ITERATIVE INTERPRETATION OF TAIL-RECURSIVE LISP PROCEDURES

Patrick GREUSSAY

Departement d' Informatique

University of Vincennes

September 1976

ABSTRACT

The design of a LISP interpreter that allows tail-recursive procedures to be interpreted iteratively is presented at the machine-language level. Iterative interpretation means that, without any program transformations, no environments and continuations will be stacked unless necessary.

We apply a specific modification within a traditional stack-oriented version of LISP interpreter, without any non-recursive control structure. The design is compatible with value-cells as well as a-lists LISP processors.

We present a complete modified interpreter written itself in LISP and an informal proof that it meets its requirements.

1.0 INTRODUCTION

It is well-known that tail-recursive procedures (TR for short) are formally equivalent to iterative procedures [1]. Unfortunately, when interpreted, TR code behaves as ordinary recursive code, and we do not obtain the benefit of the formal equivalence. We need a way to make the formal equivalence also a practical one.

The problem of the computation of TR procedures can be stated in terms of program transformations, or in terms of interpreters specially designed to handle them properly.

Static program transformations [4,10] from TR to iterative programs are mainly used on compiler-oriented LISP systems, and therefore do not allow full access to interpreter-oriented tools of debugging, breaks, traces etc. Moreover, as they are name-sensitive, they cannot handle procedures with circular types.

Interpreter-oriented processing of TR procedures uses non-recursive control structures like message-passing as in Hewitt's ACTORS system [7,8] or generalized FUNARG devices with static a-lists, as in Sussman's SCHEME [12]. Also delayed evaluation of CONS [6] has been proposed in which partial building of a structure is triggered by invocations to the decomposition primitives CAR and CDR applied to this virtual structure.
As interesting as they are, there do not seem to be any evident ways to adapt such methods to ordinary recursive stack-oriented LISP interpreters.

In contrast, we propose a simple way to process TR procedures iteratively, without any program transformations or non-recursive control structures. Our scheme can be used with a-lists as well as value-cells stack-oriented LISP interpreters.

2.0 TAIL RECURSIVE SCHEMATA

In [1], a TR schema in iterative form is defined as a recursion equation

$$f(x1,\ldots,xn) = g(f,x1,\ldots,xn,h1,\ldots,hm)$$

where g is a conditional expression defining f in terms of the functions h1,...,hm; g is said to be iterative if f occurs exactly in terms of g in the form THEN f(...) or ELSE f(...).
For example, this is the recursive form of the well-known program for addition (NOTE 1)

```
(de plus (x y) (if (= x 0) y
                   (add1 (plus (sub1 x) y))))
```

and this is the corresponding iterative form

```
(de plus (x y) (if (= x 0) y
                   (plus (sub1 x) (add1 y))))
```

NOTE 1 : (if c e1 e2 ... en) is the LISP equivalent to the form *if* c *then* e1 *else* e2; ... ; en *fi* .

We must generalize the previous definition of iterative schema to any named λ-expression such that

(1) f = (λ(x1...xn) ... (f a1...an)) (NOTE 2)

i.e. the last term of the λ-expression body is a call of this λ-expression.

(2) f = (λ(x1...xn) ... (if c (f a1...an) ...))
and
f = (λ(x1...xn) ... (if c e1 e2 ... (f a1...an)))

i.e. the THEN-part or the last term of the ELSE-part of an if-form is a call of this λ-expression.
Nested if-forms are valid under this schema, e.g.

f = (λ(x1...xn) ... (if c1 (if c2 ... (if cm (f a1...an) ...) ...)))

(3) f = (λ(x1...xn) ... (cond ... (c e1 ... em-1 (f a1...an)) ...))

Note that the condition c, even if it consists solely of the constant T, must be mentionned explicitely, in contrast with for example INTERLISP [13] style of writing CONDs.

3.0 RETURN CONTINUATIONS

We notice that these schemata share a common property: all of them are instances of forms interpreted by the LISP system internal function PROGN.

These forms will be interpreted recursively if PROGN is defined as follows. Let us suppose the variable EXP is the name of a register which contains the form to be evaluated and the result after the evaluation. TEMP is a working register, SAVE and RESTORE push and pop respectively their argument onto a stack, REC pushes a return continuation (NOTE 3), and UNREC restores the current continuation from the stack.

```
PROGN = temp←exp;
        while not(null(temp))
              do save(temp); exp←car(temp);
                 rec EVAL; temp←restore();
                 temp←cdr(temp)
              od
        unrec();
```

NOTE 2: This part of the definition means that we allow non-terminating procedures.

NOTE 3: Here we use the continuation concept [11] the same way as in [9], where a continuation is just a list of instructions to be executed.

On the contrary no return continuation will be stacked at an instance of a TR call of our iterative λ-expressions if PROGN is designed as:

```
PROGN = while not(null(cdr(exp)))
            do save(exp); exp←car(temp);
               rec EVAL; exp←restore();
               exp←cdr(exp)
            od
        exp←car(exp); jumpTo EVAL;
```

The last clause of a PROGN argument (a list of expressions to be evaluated) will be passed directly to EVAL, which obtains the definitive control of the continuation.

If LISP TR procedures had no arguments, the interpreter would automatically handle calls of forms like

```
f = ( λ() (if c e1 e2 ... en-1 (f)))
```

as

```
while not c do eval(e2); ...; eval(en-1) od;
eval(e1);
```

Then the program writer could design its procedures in the most natural way, without paying attention to what are necessary or unnecessary recursions [14]. But, as LISP procedures generally use argument passing, we need a way to omit unnecessary saving of environments (NOTE 4) by the internal LISP system function APPLY, in the case of TR procedures.

4.0 THE HANDLING OF ENVIRONMENTS

A redundant environment is defined as a new environment caused by the call of a TR procedure, the old environment being unnecessarily saved, in spite of the fact that it will be never used again.

To avoid redundant environments, we have to modify APPLY in the following manner.
When APPLY has discovered that it must handle a λ-expression, first it examines the stack at a definite place to see if the same λ-expression has been called before. If this is the case, it does not save the current environment onto the stack, and just binds every variable of its formal arguments list to its value, then gives the body of the λ-expression to PROGN. If this is not the case, then APPLY saves the current environment (which means that the λ-expression is called for the first time, as far as APPLY can see) onto the stack, then saves the list which represents the λ-expression being called, then builds a new environment as before, and finally saves a return continuation to a part of APPLY which restores environments. The control is then given to PROGN.

The definite element which APPLY examines is then the next-to-last item saved in the stack.
This part of APPLY can be designed in the following manner (the field CVAL being the value-cell of LISP variables) :

```
PART-OF-APPLY = if car(exp)=λ then
                    if STACK[TOP-1]=exp then
                       for-each x in cadr(exp) do
                                x.CVAL←car(arglist);
                                arglist←cdr(arglist)
                                od
                       exp←cddr(exp); jumpTo PROGN
                    else
                       save(sentinel);
                       for-each x in cadr(exp) do
                                save(x.CVAL); save(x);
                                x.CVAL←car(arglist);
                                arglist←cdr(arglist)
                                od
                       save(exp); exp←cddr(exp); rec PROGN;
                       restore();
                       while STACK[TOP]≠ sentinel do
                             x←restore();
                             x.EVAL←restore()
                       od
                       restore(); unrec()
                    fi
                fi
```

NOTE 4: By environment is meant an association of variables with values. In LISP, saved environments can take the form of a-lists, where the value of every variable is found (with the possibility of multiple instances of the same variable), or can take the form of value-cells, in which case the value associated with the variable is unique and immediately accessible, old associations being saved on the stack.

5.0 EXAMPLES OF TR-PROCEDURES

Here are some examples to illustrate programming in TR style, with the modified interpreter.

(1) An iterative Ackermann function :

```
(de ack (x y) (a x y nil))

(de a (x y p) (cond
                ((= x 0) (if p (a (car p) (add1 y) (cdr p))
                               (add1 y)))
                ((= y 0) (a (sub1 x) 1 p))
                (T (a x (sub1 y) (cons (sub1 x) p)))))
```

(2) Building a list of factorials :

```
(de factlist (n) (g n 1 (list 1)))

(de g (n x r)
    (if (= x n) r
          (g n (add1 x)
               (cons (times (add1 x) (car r)) r))))
```

(3) Building a factorial procedure with circular type :

```
(setq g '((λ (x y f)
            (if (= x 0) y
                  ((car f) (sub1 x) (times x y) f)))))
```

to obtain factorial n we call ((car g) n 1 g)

Our modified interpreter appears to be "name-insensitive" and can run this example iteratively, which cannot be handled by program transformations.

(4) Notice that forms like

```
((λ (x) (x x)) '(λ (x) (x x)))
```

being run iteratively do not cause overflow of the stack.

6.0 THE MODIFIED INTERPRETER

Here is the complete modified LISP interpreter (NOTE 5). It is written itself in LISP in the machine language style of Sussman's SCHEME [12]. We use a global environment and we do not use any recursive features.

```
(de run () (setq pc 'toplevel) (loop))

(de loop () (while t (apply pc nil)))
```

We use a non-terminating control loop which runs the "next" procedure, in which the non-modified LISP internal function *apply* is called over and over again. The variable pc plays the role of a program counter. Here is the top-level loop :

```
(de toplevel () (setq link nil stack nil) (save 'top1)
                (setq exp (read) pc 'eval))

(de top1 () (print exp) (setq pc 'toplevel))
```

Here are the "pipe-lined" procedures eval, evlis and apply :

```
(de eval () (cond
        ((numberp exp) (unrec))
        ((atom exp) (setq exp (car exp)) (unrec))
        (T (setq hdexp (car exp) exp (cdr exp) pc 'eval1))))

(de eval1 () (cond
         ((listp hdexp) (save hdexp) (setq pc 'evlis))
         ((or (get hdexp 'expr) (get hdexp 'subr))
          (save hdexp) (setq pc 'evlis))
         ((setq temp (get hdexp 'fexpr))
          (setq arglist (list exp) exp temp pc 'apply))
         ((get hdexp 'fsubr) (setq pc hdexp))
         (T (setq hdexp (car hdexp)))))


(de evlis () (setq built nil pc 'evlis1))

(de evlis1 () (if (null exp)
                  (setq arglist (reverse built) exp (restore) pc 'apply)
                  (save exp) (save built) (save 'evlis2)
                  (setq exp (car exp) pc 'eval)))

(de evlis2 ()
          (setq built (cons exp (restore)) exp (cdr (restore)) pc 'evlis1))
```

NOTE 5 : This interpreter is in fact a simplification of the one running under the name of VLISP at the University of Vincennes [2,5] on a PDP 10 and a T1600 computer.

```
(de apply () (cond
        ((atom exp) (cond
              ((setq temp (or (get exp 'expr) (get exp 'fexpr)))
               (setq exp temp))
              ((or (get exp 'subr) (get exp 'fsubr))
               (setq pc exp))
              (T (setq exp (car exp)))))
        ((or (eq (car exp) (setq temp 'λ))
             (eq (car exp) (setq temp 'γ)))
         (setq λ-γ-exp exp)
         (if (eq temp 'γ) (setq arglist (car arglist)))
         (if (eq (cadr stack) λ-γ-exp)
             (rebind (cadr λ-γ-exp) arglist)
             (bind (cadr λ-γ-exp) arglist) (save λ-γ-exp) (save 'apply3))
         (setq exp (cddr λ-γ-exp) pc 'progn))
        (T (save arglist) (save 'apply2) (setq pc 'eval))))
```

A form (γ(x1...xn) e1 e2 ... en) is like a λ-expression but when applied to an argument which is a list, it distributes the elements of this list over the formal arguments x1...xn . This is very efficient for handling multiple values recursive procedures.

```
(de apply2 () (setq arglist (restore) pc 'apply))

(de apply3 () (restore) (unbind) (unrec))
```

The internal procedure *unbind* restores old environments, *rebind* simply builds a new environment without saving the current one in contrast to *bind* which saves the old environment.

Next we come to the "sequencer" procedure *progn* :

```
(de progn () (if (cdr exp) nil (save exp) (save 'progn1))
             (setq exp (car exp) pc 'eval))

(de progn1 () (setq exp (cdr (restore)) pc 'progn))
```

Finally, as an illustration of control procedure of FSUBR type, we give the code for *if* :

```
(df if () (save exp) (save 'if1) (setq exp (car exp) pc 'eval))

(de if1 () (if exp (setq exp (cadr (restore)) pc 'eval)
                   (setq exp (cddr (restore)) pc 'progn)))
```

7.0 CHECKING-RULES

We must now devise a means of insuring that our interpreter meets its requirements. We shall use "checking-rules" of the form

S1{P1}P2:S2

where S1 is the state of the stack when entering procedure P1, S2 is the state of the stack when leaving procedure P1, and P2 is the name of the next procedure to enter. When P2 is the label "retcont" it means that P1 has no next procedure to enter, so a recursive return to the head of the stack has to be performed.

Examination of the interpreter yields the following rules, which constitute in a sense an abstract version of the interpreter, the enter and exit states of the stack playing the role of an history [3].

α{eval}retcont:α ∨ eval1:α

α{eval1}evlis:hdexp:α ∨ apply:α ∨ fsubr:α ∨ eval1:α

hdexp:α{evlis}evlis1:hdexp:α

hdexp:α{evlis1}eval:evlis2:built:exp:hdexp:α ∨ apply:α

built:exp:hdexp:α{evlis2}evlis1:hdexp:α

α{apply}apply:α ∨ subr:α ∨ fsubr:α
∨ progn:α ∨ progn:apply3:λ-γ-exp:oldbindings:α
∨ eval:apply2:arglist:α

arglist:α{apply2}apply:α

λ-γ-exp:oldbindings:α{apply3}retcont:α

β{progn}eval:progn1:exp:β ∨ eval:β

exp:β{progn1}progn:β

β{if}eval:if1:exp:β (NOTE 6)

exp:β{if1}eval:β ∨ progn:β

Using the checking-rules, we can show that the interpreter handles TR programs correctly.

let foo = (λ (x1 ... xn) e1 ... em)

with em = (foo a1 ... an) .

First of all we must show that the state of the stack is the same when evaluating em and when entering progn with exp = (e1 ... em) .

NOTE 6 : Recall that *if* is of FSUBR type.

Let exp = (e1 ... em) with α{progn}.

Suppose m = 1, we have

α{progn}eval:α and therefore α{eval}.

If m > 1 we have

α{progn}eval:progn1:exp:α

and if the evaluation of the head of exp does not enter into an infinite loop, we shall obtain

exp:α{progn1}progn:α

followed by

α{progn} , now with the length of exp being m-1 □.

Further, we must show that when

apply3:foo:oldbindings:α{eval}

then exp = em , i.e. it is only when evaluating em that we can find foo as the next-to-top item in the stack.

Suppose exp = ek with k ≠ m. The state of the stack when entering eval will be

progn1:(ek ... em):β{eval}

and (ek ... em) being a tail of foo cannot be equal to foo □.

Finally we must show that, if there is not an infinite loop when evaluating one of the ei , 1 ≤ i ≤ m, the old bindings will be restored.

As before, if exp = em, with

apply3:foo:oldbindings:α{eval}

when eval returns, the state of the stack is

foo:oldbindings:α{apply3}retcont:α

and apply3 restores the environment immediately preceding the first call of foo □.

8.0 CONCLUDING REMARKS

We have proposed a LISP interpreter in which TR code behaves at run-time as efficiently as well-written iterative code, with the extra benefit of avoiding explicit side-effects as well as manual or automatic program transformations.
Another advantage is that it is insensible to renaming, e.g. if we have

```
(de foo (x) ... (foo (g x)))
```

and we perform

```
(put 'fie (get 'foo 'expr) 'expr)
```

then the call (fie a) will be interpreted exactly the same way as a call of foo.

The modification we have proposed does not depend on particular implementations of environments. This one more encouragement to write programs in recursive style, particularly since our modification can be applied very easily with no apparent drawbacks to any LISP interpreter.

Ed. Note: The whole thing has been improved since 1976. It can run the same way mutual co-recursive procedures, and also what I call "enveloped" tail-recursions, as in

```
(DE foo (x)
    (IF (ZEROP x) 0
        (+ x (foo (SUB1 x)))))
```

ACKNOWLEDGEMENTS

Many discussions and advices from J. CHAILLOUX have been very helpful, as well as the material help he provided during the preparation of this report.

1. J. McCARTHY : "Toward a mathematical science of computation."
Proc. IFIP 1962 21-28

2. CHAILLOUX J. : "VLISP 10" RT 17-76
Computer Sc Dept. University of Vincennes. France.

3. M. CLINT : "Program Proving : Coroutines"
Acta Informatica 2, 50-63 (1973)

4. DARLINGTON J., BURSTALL R.M. : "A system which automatically improves programs." (1973)
Proc. of 3d International Joint Conference on Artificial Intelligence.
Stanford. 537-542

5. GREUSSAY P. : "Descriptions compactes d'interprètes implémentables."
Programming Symposium. Paris. Avril 1976.
B. Robinet ed. 281-297

6. FRIEDMAN D.P., WISE D.S. : "Output Driven Interpretation of Recursive Programs."
TR n°50. Indiana University. July 1976.

7. C. HEWITT : "Behavioral semantics of nonrecursive control structures"
Proc. Programming Symposium. Paris. 1974.
B. Robinet ed. Springer-Verlag. 385-407

8. C. HEWITT : "Viewing control structures as patterns of passing messages"
Working Paper 92. April 1976. MIT AI Lab.

9. REYNOLDS J.C. : "Definitional interpreters for higher-order programming languages."
Proc of 1972 ACM Nat. Conf. Boston. 1972. 717-740

10. RISCH T.: "REMREC : A Program for automatic recursion removal in LISP."
Uppsala University. 1973

11. STRACHEY C. : "A mathematical semantics which can deal with full jumps"
Séminaires IRIA "Théorie des algorithmes, des langages et de la programmation" ed. M. NIVAT. Mai 1973. 175-191

12. SUSSMAN G.J., STEELE Jr G.L. : "SCHEME : An interpreter for extended lambda-calculus."
MIT AI Lab. AI Memo n° 349. Dec 1975

13. TEITELMAN W. : "Interlisp Reference Manual."
XEROX Palo Alto Research Center. 1974

14. WIRTH N. : "Algorithms + Data Structures = Programs"
Prentice Hall. 1976

SEND MORE TECHNICAL NOTES)