A Lazy Evaluator

Peter Henderson
University of Newcastle upon Tyne

James H. Morris, Jr.
Xerox Palo Alto Research Center

Abstract: A different way to execute pure LISP programs is presented. It delays the evaluation of parameters and list structures without ever having to perform more evaluation steps than the usual method. Although the central idea can be found in earlier work this paper is of interest since it treats a rather well-known language and works out an algorithm which avoids full substitution. A partial correctness proof using Scott-Strachey semantics is sketched in a later section.

I. Introduction

This paper studies a non-standard method of performing the mechanical evaluation of expressions in a purely applicative language;i.e. one without assignment. The intuitive ideas behind this method are two:

- Perform an evaluation step only when it is necessary.

- Never perform the same step twice.

It is somewhat surprising that these objectives can be approached through the use of rather simple data structures and algorithms. The following example should serve to acquaint the reader with the basic idea.

```
integer procedure g(x,y);
    g := if x = 0 then 1 else y*y
```

An ALGOL-60 programmer who wanted to enhance this procedure's speed by choosing whether to declare y a call-by-name or a call-by-value parameter would feel most uncomfortable. The value of y is going to be used either twice or not at all, depending on the value of x. The lazy evaluation technique overcomes this dilemma because the evaluation of g(E,F) will proceed as follows:

(1) Substitute pointers, α and β, to the expressions E and F for the formal parameters x and y.

(2) Evaluate (i.e. reduce to a numeral) the contents of α.

(3) If the result is 0 return 1.

(4) Otherwise, evaluate the contents of β and replace them with the resulting numeral.

(5) Evaluate the contents of β again (this takes little time since already a numeral) and multiply the results.

Thus a lazy evaluator will perform the work to evaluate the second parameter either once or not at all. This example illustrates the call-by-need mechanism of Wadsworth [7] and and the delay rule of Vuillemin [6].

Here we shall carry this strategy one small, but important step further as suggested in [6]: list structures are evaluated incrementally. In LISP parlance, an argument of CONS is not evaluated until and unless it is selected and examined by some later operation. Thus the statement

```
car[cons[x;y]] = x
```

is always true, even if the evaluation of x or y never terminates. This extension is pragmatically important because it allows the possibility for significantly different styles of programming as the following examples illustrate.

Example 1. Infinite Lists

The function defined by

```
integers[i] = cons[i; integers[i+1]]
```

is quite useful under a lazy evaluation regime. integers[0] denotes the infinite list (0 1 2 ...) and the expression

```
car[cdr[integers[0]]]
```

will evaluate to 1 via the following intermediate steps:

```
car[cdr[cons[0; integers[0+1]]]]
car[integers[0+1]]
car[cons[0+1; integers[0+1+1]]]
0+1
1
```

In a similar way, the list defined by

```
L = cons[1; cons[2; L]]
```

is useful and computable.

Example 2. A leaf comparator

The following functions solve a problem posed by Carl Hewitt [2] to illustrate the need for co-routines.

```
EqLeaves[x;y] = EqList[Flatten[x];Flatten[y]]
```

```
Flatten[x] = if atom[x] then cons[x;NIL]
             else Append[Flatten[car[x]];
                         Flatten[cdr[x]]]

Append[x;y] = if null[x] then y
              else cons[car[x];Append[cdr[x];y]]


EqList[x;y] = if null[x] then null[y] else
              if null[y] then false else
              if eq[car[x];car[y]]
                 then EqList[cdr[x];cdr[y]]
              else false
```

In other words, EqLeaves tests two S-expressions to see if their atoms are identical, independent of structure. This obvious solution uses Flatten to eliminate the structure, then uses EqList to compare the atoms. It would be an unnecessarily slow method under normal circumstances because applied to a pair of expressions like

(A Huge1) and (B Huge2)

it would go to all the work of Flattening Huge1 and Huge2 even though the answer is false because of the first atoms in each structure differ. If a lazy evaluator is used, however, there is no need to change the solution to one involving co-routines because the same computational effect will be achieved automatically. Suppose location π_0 holds the expression to be evaluated. The computation will follow this pattern:

π_0: EqLeaves[(A Huge1);(B Huge2)]

First π_0 is updated with the definition of EqLeaves with actual parameters substituted for formals.

π_0: EqList[Flatten[(A Huge1)];Flatten[(B Huge2)]]

Now pointers, π_1 and π_2, to the parameters are substituted into the definition of EqList without any evaluation of the parameters.

π_0: if null[π_1] then null[π_2] else
if null[π_2] then false else
if eq[car[π_1];car[π_2]] then EqList[cdr[π_1];cdr[π_2]]
else false

π_1: Flatten[(A Huge1)]

π_2: Flatten[(B Huge2)]

The primitive null now forces the lazy evaluator to go to work on the contents of π_1.

π_1: if atom[(A Huge1)] then cons[(A Huge1); NIL]
else Append[Flatten[car[(A Huge1)]];
Flatten[cdr[(A Huge1)]]]

π_1: Append[Flatten[car[(A Huge1)]];
Flatten[cdr[(A Huge1)]]]

π_1: if null[π_3] then π_4
else cons[car[π_3];Append[cdr[π_3];π_4]]

π_3: Flatten[car[(A Huge1)]]

π_4: Flatten[cdr[(A Huge1)]]

Again the primitive null forces evaluation steps on π_3.

π_3: if atom[π_5] then cons[π_5;NIL]
else Append[Flatten[car[π_5]];Flatten[cdr[π_5]]]

π_5: car[(A Huge1)]

The primitive atom forces the evaluation of π_5.

π_5: A

π_3: cons[π_5;NIL] since atom[A] is true

π_1: cons[car[π_3]; Append[cdr[π_3]; π_4]] since null[cons ...] is false

π_0: if null[π_2] then false
if eq[car[π_1];car[π_2]] then EqList[cdr[π_1];cdr[π_2]]
else false

If we choose to view Flatten as a co-routine, at this point we would say that it has produced its first result, car[π_3] = A, and its context has been saved in π_4 for later reactivation.

Now the contents of π_2 is evaluated in the same way until we have

π_2: cons[car[π_6]; Append[cdr[π_6];π_7]]

π_6: cons[B;NIL]

π_7: Flatten[(B Huge1)]

π_0: if eq[car[π_1];car[π_2]] then EqList[cdr[π_1];cdr[π_2]]
else false

The primitive eq forces

π_1: cons[A; Append[cdr[π_3]; π_4]]

π_2: cons[B; Append[cdr[π_6]; π_7]]

Finally the test is made and the computation terminates with

π_0: false

Notice that Huge1 and Huge2 did not enter into any of the foregoing computation and that the work done to evaluate the subexpressions in π_1 and π_2 for the benefit of the null primitive is not repeated when the eq primitive examines the parameters.

Generalizing from this example we can see that a large class of co-routine applications can be subsumed by this technique. A producer co-routine becomes a function that produces a long (possibly infinite) list and a consumer co-routine becomes the receiver of such a list. Also, notions such as streams and the dynamic lists of POP-2 are subsumed. The purpose of these programming constructs is to allow one to describe a sequence of values with a single sub-program, yet have them computed on a hand-to-mouth basis. This assumption is built into a lazy evaluator at the most basic level so there is no need to call for it explicitly.

One the other hand, one might ask how to force a more conventional evaluation to occur. Suppose one wishes to cause the evaluation of f[s] to proceed conventionally, computing the S-expression s before invoking f. One could

say, instead of f[s], Force[f,s] where

Force[f,s] = **if** Finite[s] **then** f[s] **else** don't care

and

Finite[s] = **if** atom[s] **then true else**

if Finite[car[s]] **then** Finite[cdr[s]] **else** don't care

The function Finite simply explores the complete S-expression, forcing every part of it to be evaluated. If it ever terminates, Force invokes the function on the now-evaluated argument.

Example 3: Prime Numbers (due to P. Quarendon)

primeswrt[x;l] produces a new list from l by removing all multiples of x

primes[l] produces a new list from l by removing any element which is a multiple of a predecessor.

primeswrt[x;l] = **if** car[l] **mod** x=0 **then** primeswrt[x;cdr[l]]
else cons[car[l];primeswrt[x;cdr[l]]]

primes[l] = cons[car[l];primes[primeswrt[car[l];cdr[l]]]]

then

primes[integers[2]]

is the infinite list of prime numbers.

II. A lazy evaluator for Hyper-Pure LISP

In this section we shall describe a language and its implementation in order to crystallize the notion of lazy evaluation. Hyper-Pure LISP is a variant of LISP 1.0 which remains true to the principles of the λ-calculus [1]. Specifically, FUNARG binding is the only possibility. The syntax of expressions in this language is as follows:

<expression> ::= <variable> |
(QUOTE <atom>) |
(CONS <expression> <expression>) |
(CAR <expression>) |
(CDR <expression>) |
(ATOM <expression>) |
(EQ <expression> <expression>) |
(IF <expression> <expression> <expression>) |
(<expression> <expression>) |
(LAMBDA <variable> <expression>) |
(LABEL <variable> <expression>) |
(FUNARG <expression> <alist>)

<alist> ::= <empty> | <variable> <expression> <alist>

<atom> ::= <any string of capital letters>

<variable> ::= <any atom except CONS, CAR, CDR, ATOM, EQ, IF, LAMBDA, QUOTE, LABEL, or FUNARG>

The variations from the syntax of LISP are the replacement of COND by IF, the restriction of QUOTE to atoms, the restriction of LAMBDA-defined functions to one argument, and the elevation of the FUNARG construct from an internal bookkeeping device. The first three restrictions are inessential and the FUNARG phrase is an extension. Intuitively "(FUNARG e1 x e2 y e3)" means "e1 where x = e2 and y = e3".

It will greatly simplify the discussion if we assume that the computer memory is of a very accommodating, if unrealistic, sort: each cell is capable of holding any of the forms listed as <expression>s where addresses are used for any component of type expression. In other words, if π_0, π_1, etc. are addresses and ξ_0, ξ_1, etc. are variables, a single memory cell is capable of holding (and discriminating among) items like ξ_0 , (CONS π_1 π_2), (π_0 π_1), (LAMBDA ξ_1 π_0), and (FUNARG π_0 ξ_1 π_1 ξ_2 π_2). Naturally, any real implementation would represent such variable-sized memory cells with linked lists; but the extra pointers would only complicate this discussion.

The state of a computation is described by a partial memory function, μ, which has the following consistency property: If a particular address, π, occurs as a component anywhere in the memory (i.e. in μ's range), then $\mu(\pi)$ is defined. Thus the addresses for which μ is undefined are the free cells and no non-free cell points at a free one. A computation is started by loading the memory with the expression in the obvious way, performing transformations for a while, and then examining the root cell of the expression.

As the first step in describing the lazy evaluator we describe a set of reduction rules which transform the memory, μ, to produce a new memory μ'. Each rule changes just a few cells. To describe such an altered function we introduce some notation:

$$\mu[\ \varepsilon\ /\ \pi_0\ ] = \lambda\pi.\ \textbf{if}\ \pi{=}\pi_0\ \textbf{then}\ \varepsilon\ \textbf{else}\ \mu(\pi)$$

In other words the new function differs from the old at just one argument, π_0 where its value has become ε. Furthermore

$$\mu[\ \varepsilon_1/\pi_1,\ \varepsilon_2/\pi_2\ ] = (\mu[\ \varepsilon_1/\pi_1\ ])[\ \varepsilon_2/\pi_2\ ]$$

In other words the rightmost pair represents the last change to the memory.

There are seven transformation rules.

[C] $\mu(\pi_0)$ = (CAR π_1) and $\mu(\pi_1)$ = (CONS π_2 π_3)

$\Rightarrow$ $\mu' = \mu[\ \mu(\pi_2)\ /\ \pi_0\ ]$

In programming terms, the contents of π_2 are copied into π_0. When convenient we shall dispense with mention of extra references by convening that $\mu(\pi_0)$ = (CAR (CONS π_2 π_3)) implicitly asserts the existence of a π_1 for which the above is true. The rule for CDR is similar.

$$\mu(\pi_0) = (\text{CDR } (\text{CONS } \pi_2\ \pi_3)) \Rightarrow \mu' = \mu[\mu(\pi_3)\ /\ \pi_0]$$

[A] $\mu(\pi_0)$ = (ATOM(QUOTE α))
$\Rightarrow$ $\mu' = \mu$[(QUOTE T) / π_0]

$\mu(\pi_0)$ = (ATOM(CONS π_1 π_2))
$\Rightarrow$ μ'= μ[(QUOTE NIL) / π_0]

I.e. "atom-hood" is simply the property of being QUOTEd.

[E] $\mu(\pi_0)$ = (EQ(QUOTE α_1)(QUOTE α_2)) ⇒

$\mu' = \mu$[if $\alpha_1 = \alpha_2$ then(QUOTE T)else(QUOTE NIL) / π_0]

[I] $\mu(\pi_0)$ = (IF (QUOTE T) π_1 π_2) ⇒ $\mu' = \mu[\mu(\pi_1) / \pi_0]$

$\mu(\pi_0)$ = (IF (QUOTE NIL) π_1 π_2) ⇒ $\mu' = \mu[\mu(\pi_2) / \pi_0]$

Thus IF is just an alternate form of the usual COND primitive.

The following rule is a sort of incremental substitution operation which allows FUNARG expressions to bubble down through the memory. When the specific list of variable-address pairs in a FUNARG expression is not relevant we use θ to denote it.

[F] (a) $\mu(\pi_0)$ = (FUNARG π' ξ_1 π_1 ... ξ_n π_n)

and $\mu(\pi') = \xi_i$ for some $1 \le i \le n$
(choose the smallest i)

⇒ $\mu' = \mu[\mu(\pi_i) / \pi_0$]

(b) $\mu(\pi_0)$ = (FUNARG (QUOTE α) θ)

⇒ $\mu' = \mu$[(QUOTE α) / π_0]

(c) $\mu(\pi_0)$ = (FUNARG (π_1 π_2) θ)
⇒
$\mu' = \mu$[(π_1' π_2') / π_0,
(FUNARG π_1 θ) / π_1',
(FUNARG π_2 θ) / π_2']

where π_1' and π_2' are distinct free cells in μ

(d) If α is one of CONS, CAR, CDR, ATOM, EQ, or IF then

$\mu(\pi_0)$ = (FUNARG (α π_1 ... π_m) θ)
⇒

$\mu' = \mu$[(α π_1' ... π_m') / π_0,
(FUNARG π_1 θ) / π_1',
....,(FUNARG π_m θ) / π_m']

where π_1', π_1',..., π_m' are distinct free cells in μ

Rules Fc and Fd are the only ones which use additional storage. Although it appears here that the list of pairs θ is being duplicated, in a real implementation which represents a single, variable-sized cell with multiple linked cells, it would not be; only a pointer to the list would be duplicated.

Notice that rule F does not tell what to do when a LAMBDA or LABEL construct is encountered. The following rules give an answer for some cases. The first one shows how variables are bound to values in FUNARG lists, and the second shows how recursion is implemented.

[G] $\mu(\pi_0)$ = ((FUNARG (LAMBDA ξ π_1) θ) π_2)
⇒
$\mu' = \mu$[(FUNARG π_1 ξ π_2 θ) / π_0]

The following rule creates a circular structure.

[L] Given

$\mu(\pi)$ = (FUNARG (LABEL ξ π_0) θ)

⇒

$\mu' = \mu$[(FUNARG π_0 ξ π θ) / π]

What can be said of these rules? The reader may wonder whether they "work". For example, one of the authors (Morris) thought that rule G might suffer from the usual capture of free variables problem, and was surprised to find it was not the case. This issue is taken up in the next section. Presently we shall continue with the description of the evaluator.

Note that at most one rule can be applicable to any particular location, so the only choices left open to an evaluator are where to perform the next reduction and when to halt. The procedure for a lazy evaluator is defined recursively as follows:

Eval(π,μ) : π is the location to be reduced μ is the memory
Eval returns a new memory

```
if μ(π) = (QUOTE α) then return μ
if μ(π) = (CONS ...) then return μ
if μ(π) = (FUNARG (LAMBDA ...) ...) then return μ
[C1]
if μ(π) = (CAR π')
      then let μ1 = Eval(π',μ)
           if μ1(π') = (CONS π1 π2)
                then let μ2 = Eval(π1,μ1)
                     return μ2[μ2(π1) / π]
                else there is an error, CAR applied to
                     non-pair
[C2]
if μ(π) = (CDR π')
      then let μ1 = Eval(π',μ)
           if μ1(π') = (CONS π1 π2)
                then let μ2 = Eval(π2,μ1)
                     return μ2[μ2(π2) / π]
                else there is an error, CDR applied to
                     non-pair
[A]
if μ(π) = (ATOM π')
     then let μ1 = Eval(π',μ)
          if μ1(π') = (QUOTE α)
            then return μ1[ (QUOTE T) / π];
          if μ1(π') = (CONS ...)
            then return μ1[(QUOTE NIL) / π];
          else there is an error
[E]
if μ(π) = (EQ π1 π2)
     then let μ1 = Eval(π2,Eval(π1,μ))
          if μ1(π1)=(QUOTE α1) ∧ μ1(π2)= (QUOTE α2)
          then if α1 = α2 then return μ1[(QUOTE T) / π]
                          else return μ1[(QUOTE NIL) / π]
          else there is an error
```

```
[I]
if μ(π) = (IF π0 π1 π2)
    then let μ1 = Eval(π0,μ)
        if μ1(π0) = (QUOTE T)
          then let μ2 = Eval(π1,μ1)
               return μ2[μ2(π1) / π ]
          if μ1(π0) = (QUOTE NIL)
          then let μ2 = Eval(π2,μ1)
               return μ2[μ2(π2) / π ]
          else there is an error
[Fa]
if μ(π) = (FUNARG π0 ξ1 π1 ...ξn πn ) and μ(π0) = ξ0
      then
          if i is the smallest such that ξ0 = ξi
               then let μ1 = Eval(πi,μ)
                         return μ1[μ1(πi) / π ]
                 else there is no such i
                   and there is an unbound
                      variable error
[Fb]
if μ(π) = (FUNARG (QUOTE α) ...)
      then return μ[ (QUOTE α) / π]
[L]
if μ(π) = (FUNARG (LABEL ξ π') θ)
       then return  Eval(π,μ[(FUNARG π' ξ π θ) / π ])
[Fc]
if μ(π) = (FUNARG (π0 π1) ...)
      then apply rule Fc to yield μ'
           return Eval(π,μ')
[Fd]
if μ(π) = (FUNARG (α ... πn) ...)
      then apply rule Fd to yield μ'
           return Eval(π,μ')
[G]
if μ(π) = (π0 π1)
     then let μ1 = Eval(π0,μ)
              if μ1(π0) = (FUNARG (LAMBDA ...) ...)
                then apply rule G to location π to get μ2
                          return Eval(π,μ2)
                else there is an error

else there is an error
```

The most subtle aspect of this algorithm centers about rules C, I, and Fa. For example, in C1, the first component of the pair is evaluated *in situ* before being copied into π. This policy maximizes the number of paths through the data structure which "see" the change.

A conventional evaluator differs from a lazy evaluator in that the parameters of CONS are evaluated as soon as they are encountered and the actual parameter of a function call are evaluated before they are bound. In terms of the foregoing definition of Eval this means that the second case is changed to

```
if μ(π) = (CONS π1 π2 )
      then return Eval(π2 ,Eval(π1 ,μ))
```

and the case [G] is changed to

```
if μ(π) = (π0  π1)
    then let μ1 = Eval(π1,Eval(π0,μ))
           if μ1(π0) = (FUNARG (LAMBDA ...) ...)
              then apply rule G to location π and μ1
                        to get μ2
                   return Eval(π,μ2)
            else there is an error
```

These two changes make certain other invocations of Eval unnecessary. Specifically, the case C1 becomes

```
if μ(π) = (CAR π')
      then let μ1 =  Eval(π',μ)
           if μ1(π') = (CONS π1 π2)
                then return μ1[μ1(π1) / π]
                else there is an error, CAR applied to
                     non-pair
```

Case C2 changes analogously, and case Fa becomes

```
if μ(π) = (FUNARG π0  ξ1 π1 ...ξn πn ) and μ(π0) = ξ0
     then
         if i is the smallest such that ξ0 = ξi
              then return μ[μ(πi) / π ]
                else there is no such i
                  and there is an unbound
                     variable error
```

All the other cases remain the same. We believe that the lazy evaluator never performs more reduction steps than the conventional one but shall not attempt to prove it here.

Example 4.

To illustrate the lazy evaluator we shall perform a full evaluation of the expression from example 1. To get things started it is necessary to embed the expression to be evaluated in a FUNARG expression with an empty alist. Also, we shall assume that PLUS is a primitive operator with the same general characteristics as EQ and that numerals are understood to be QUOTEd atoms. Not all addresses are explicit; each pair of parentheses indicates the presence of an unnamed address.

```
α:(FUNARG
   ((LAMBDA INTS (CAR(CDR(INTS 0))))
    (LABEL INTEGERS
          (LAMBDA I
            (CONS I(INTEGERS (PLUS I 1)))))))
```

Apply Fc to α

```
α: ((FUNARG (LAMBDA INTS (CAR(CDR(INTS 0)))))
    β)

β: (FUNARG
   (LABEL INTEGERS
         (LAMBDA I
           (CONS I(INTEGERS (PLUS I 1))))))
```

Apply G to α

```
α: (FUNARG
     (CAR(CDR(INTS 0)))
     INTS β)
```

Apply Fd to α

```
α: (CAR γ)

γ: (FUNARG (CDR (INTS 0))
                INTS β)
```

Stack α, Apply Fd to γ

```
γ: (CDR δ)
```

δ: (FUNARG (INTS 0) INTS β)

Stack γ,Apply Fc to δ

δ: (ε φ)

ε: (FUNARG INTS INTS β)

φ: (FUNARG 0 INTEGERS β)

Stack δ, ε, Apply L to β

β: (FUNARG
(LAMBDA I
(CONS I(INTEGERS (PLUS I 1))))
INTEGERS β)

Return to ε, Apply Fa

ε: (FUNARG
(LAMBDA I
(CONS I (INTEGERS(PLUS I 1))))
INTEGERS β)

Return to δ, Apply G

δ: (FUNARG
(CONS I (INTEGERS(PLUS I 1)))
I φ
INTEGERS β)

Apply Fd to δ

δ: (CONS ξ η)

ξ: (FUNARG I I φ INTEGERS β)

η: (FUNARG (INTEGERS (PLUS I 1))
I φ INTEGERS β)

Return to γ to note CDR, Stack γ again,
Apply Fc to η

η: (ι s)

ι: (FUNARG INTEGERS I φ
INTEGERS β)

s: (FUNARG (PLUS I 1) I φ
INTEGERS β)

Stack η, Apply Fa to ι

ι: (FUNARG (LAMBDA I (CONS I
(INTEGERS (PLUS I 1))))
INTEGERS β)

Return to η, Apply G

η: (FUNARG (CONS I (INTEGERS
(PLUS I 1)))
I s
INTEGERS β)

Apply Fd to η

η: (CONS κ λ)

κ: (FUNARG I I s
INTEGERS β)

λ: (FUNARG (INTEGERS (PLUS I 1))
I s INTEGERS β)

Return to γ, Apply C

γ: (CONS κ λ)

Return to α to note CAR, Stack α again,
Stack κ, Apply Fd to s

s: (PLUS μ ν)

μ:(FUNARG I I φ INTEGERS β)

ν:(FUNARG 1 I φ INTEGERS β)

Stack s, Stack μ, Apply Fb to φ

φ: 0

Return to μ , Apply Fa to μ

μ: 0

Return to s to note PLUS, Stack s,
Apply Fb to ν

ν: 1

Return to s, Apply P

s: 1

Return to κ, Apply Fa

κ: 1

Return to α, Apply C

α: 1

III. Semantic Considerations

There are several questions one might ask about the foregoing transformation rules and evaluator.

(1) Is the final answer independent of the order in which the rules are applied; i.e. does the system have the Church-Rosser property?

(2) Are there enough rules to allow an answer to be computed in all cases we consider legal?

(3) Is the evaluator complete in the sense that will compute an answer whenever any application of the rules will do so?

(4) Is the evaluator optimal in the sense that it performs the minimum needed steps to compute an answer?

We believe that the answer to the first three is yes, and know that the fourth is not true. However, they are not very meaningful questions unless we have an independent definition of what an "answer" is. To get one we shall first define the meaning of an expression in terms of Scott-Strachey semantics [4] and then define an answer as a certain finite amount of information about that meaning. An analogous approach for arithmetic would be as follows:

(1) Given the class of arithmetic expressions involving numerals and no variables (e.g. "4", "4+5", "5-(6+2)"), consider the domain of integers (i.e. ...,-2,-1,0,1,2,...).

(2) Define a semantic function, V, mapping expressions into domain elements.

$$V(\text{"9"}) = 9 \quad \text{and} \quad V(\text{"4+5"}) = 9$$

(3) In this case the definition of an answer is obvious: Any computer should reduce its input expression to the (possibly signed) numeral which has the same value as the original expression.

The central idea is that the computer does not find the value for the expression, but only reduces the input to a more comprehensible form which has the same value. When, as in the case of arithmetic, there is a unique, finitely representable canonical form for any value the distinction between a value and the output expression is not interesting. On the other hand, here we are dealing with values that can be infinite list structures and functions so the distinction between a value and an answer is real.

The definition of a semantics for the λ- calculus has already been carried out by Wadsworth and others, see [5,7]. Our approach follows theirs but extends it somewhat to introduce the notion of a semantic memory. This approach allows us more easily to make the connection between the semantics and the evaluator. In particular, it allows us to deal with the sharing and sometimes circular data structures more directly.

The Domains

A - the primitive domain of atoms. α,α',α_1, etc. denote atoms.

C - the primitive domain of variables. ξ denotes a variable. There is no reason why atoms cannot be used as variables, but things seem clearer if they are kept distinct.

R - the primitive domain of references (addresses). π denotes a reference.

$E = C + A + R\times R + R + R + R + R\times R + R\times R\times R + R\times R + C\times R + C\times R + R\times(C\times R)^*$ - the domain of expressions corresponding to the twelve possibilities listed in section II. In other words (CAR π) is a member of the fourth part of the disjoint union. ε denotes an expression.

$M = R \rightarrow E$ - the domain of memories, each cell of which is capable of holding an expression. μ denotes a memory.

$V = A + (V \times V) + (V \rightarrow V)$ - the domain of values. A value can be an atom, a pair of values, or a function from a value to a value. This domain is the one which requires Scott's theory as it contains its own function space. Furthermore we use Reynolds's [3] version of the disjoint union operator so that $\perp$, (QUOTE $\perp$), (CONS $\perp$ $\perp$), and (LAMBDA X $\perp$) all have distinct values, with $\perp$ being the bottom of the whole lattice.

$N = C \rightarrow V$ - the domain of environments. ρ denotes an environment.

$S = R \rightarrow (N \rightarrow V)$ - a semantic memory, mapping references and environments into values. σ denotes a semantic memory. A semantic memory has addresses just like a conventional one but its cells can hold genuinely infinite objects. Roughly speaking the semantic object that a cell π holds is what one gets by tracing out, in the conventional memory, the structure of pointers emanating from that cell.

The Semantic Function V

In the following, to reduce parentheses, we shall adopt the convention that $f\alpha\beta\gamma$ means $((f(\alpha))(\beta))(\gamma)$;i.e. f is applied to α, the result is applied to β, etc.

V maps conventional memories into semantic memories;i.e.

$$V : M \rightarrow S$$

it is defined recursively by $V\mu\pi\rho = U(V\mu)(\mu\pi)\rho$ where $U\sigma\varepsilon\rho =$

if $\varepsilon \in C$ **then** $\rho\varepsilon$ **else**

if ε = (QUOTE α) **then** α

if ε = (CONS π_1 π_2) **then** $\langle\sigma\pi_1\rho, \sigma\pi_2\rho\rangle$ **else**

if ε = (CAR π') **then** $(\sigma\pi'\rho)_1$ **else**

if ε = (CDR π') **then** $(\sigma\pi'\rho)_2$ **else**

if ε = (ATOM π') **then**
[**if** $\sigma\pi'\rho \in A$ **then** "T" **else**
if $\sigma\pi'\rho \in V \times V$ **then** "NIL" **else**
$\perp$
] **else**

if ε = (EQ π_1 π_2) **then**
[**if** $\sigma\pi_1\rho \in A \wedge \sigma\pi_2\rho \in A$
then if $\sigma\pi_1\rho = \sigma\pi_2\rho$ **then** "T" **else** "NIL"
else $\perp$
] **else**

if ε = (IF π_0 π_1 π_2) **then**
[**if** $\sigma\pi_0\rho$ = "T" **then** $\sigma\pi_1\rho$ **else**
if $\sigma\pi_0\rho$ = "NIL" **then** $\sigma\pi_2\rho$ **else**
$\perp$
] **else**

if ε = (π_0 π_1) **then** $\sigma\pi_0\rho$ $(\sigma\pi_1\rho)$ **else**

if ε = (LAMBDA ξ π') **then** $\lambda x.\ \sigma\pi'\rho[\ x/\xi\ ]$ **else**

if ε = (LABEL ξ π') **then** $Y\ \{\lambda x.\ \sigma\pi'\rho[\ x/\xi\ ]\}$ **else**

if ε = (FUNARG π_0 ξ_1 π_1 ... ξ_n π_n) **then**
$\sigma\pi_0\ \rho[\ \sigma\pi_n\rho/\xi_n,\ldots,\sigma\pi_1\rho/\xi_1\ ]$

else $\perp$

This definition depends upon several informal semantic operations: <,> forms pairs, a subscript selects components,etc. The most noteworthy are the last four: The application of $\sigma\pi_0\rho$ to $\sigma\pi_1\rho$ is the function application which required Scott's construction to justify since both values reside in the same domain. Note that the λ is also an informal notion and that the bracket notation is used to define the changed environment $\rho[\ x/\xi\ ]$. Y is the minimal fixed point operator. It happens that mapping a FUNARG construction into its semantics involves a complete reversal of the list of bindings.

It is instructive to compare this function with the Eval function of section II. The major difference is that V is quite happy to deal with completed infinite objects like functions. The operation of applying a function to its argument is taken as primitive here, while it was done very slowly and incompletely by Eval.

This particular way of describing the function isolates the application of the memory function to a single place, namely the $\mu\pi$ in V's definition; U uses only the semantic memory σ.

Now, given the definition of Eval, it should be clear what an answer is. Suppose we load the memory with an expression so that the root of the expression occurs in π_0, and we start the lazy evaluator on that location. If it ever halts (ignoring the possibility of error stops) we know that π_0 will contain an expression with one of the three forms (QUOTE α), (CONS ..), or (FUNARG(LAMBDA...)...). Thus the answer tells which of the three components of V the value lies in; and, if it is an atom, what atom it is. In the other cases, nothing more is revealed, or computed. This fact indicates how we should define the correctness of an evaluator:

If $V\mu\pi_0\perp = \perp$
then Eval(π_0,μ) does not halt.
If $V\mu\pi_0\perp = \alpha$
then Eval(π_0,μ)(π_0) = (QUOTE α).
If $V\mu\pi_0\perp$ is a pair
then Eval(π_0,μ)(π_0) =(CONS ...).
If $V\mu\pi_0\perp$ is a function
then Eval(π_0,μ)(π_0) =(FUNARG(LAMBDA..)...).

We use the empty environment, $\perp$, in these statements because it is assumed that the expression to be reduced does not contain free variables at the outermost level. Therefore, an environment function is not needed initially.

Soundness

For the present we shall content ourselves with sketching a partial correctness proof;i.e. that the last three clauses hold if Eval halts. This can be done by showing that each of the seven transformation rules leaves the semantic memory unchanged (except at newly allocated cells). Then if the evaluator starts with memory μ at location π_0 and halts with memory μ' we know that $V\mu\pi_0\perp = V\mu'\pi_0\perp$, and the last three clauses are immediate since the Eval function halts only on the three forms in question.

This method of proof also proves a qualified "yes" to the first question in this section: it doesn't matter in what order the transformations are applied. The argument goes as follows: On semantic grounds we know that (QUOTE α), (CONS ...) and (FUNARG(LAMBDA ...)...) all denote distinct objects, and that atoms that look different are different. If we then show that the transformations cannot change the semantic value in a cell, we know that all sequences of transformations which produces one of those configurations in a cell must produce the same one.

The proofs for the various rules are very similar. We shall give two.

Rule C: Suppose $\mu(\pi_0)$ = (CAR π_1) and $\mu(\pi_1)$ = (CONS π_2 π_3) and rule C is applied. We shall prove that $V\mu = V\mu'$ where $\mu' = \mu\ [\ \mu(\pi_2)\ /\ \pi_0\ ]$. First we define the truncations of V by

$$V_i = \perp \text{ for } i \le 0$$

$$V_{i+1}\mu\pi\rho = U(V_i\mu)(\mu\pi)\rho$$

so $V = \lim_i V_i$

Now a straight-forward computation shows that

$$V_i\mu = \lambda\pi\rho.\textbf{if } \pi=\pi_0 \textbf{ then } U(V_{i-1}\mu)(\text{CAR}(\text{CONS } \pi_2\ \pi_3))\rho \textbf{ else } U(V_{i-1}\mu)(\mu\pi)\rho$$

$$= \lambda\pi\rho.\textbf{if } \pi=\pi_0 \textbf{ then } \{U(V_{i-2}\mu)(\text{CONS } \pi_2\ \pi_3)\rho\}_1 \textbf{ else } U(V_{i-1}\mu)(\mu\pi)\rho$$

$$= \lambda\pi\rho.\textbf{if } \pi=\pi_0 \textbf{ then } U(V_{i-3}\mu)(\mu\pi_2)\rho \textbf{ else } U(V_{i-1}\mu)(\mu\pi)\rho$$

and

$$V_j\mu' = \lambda\pi\rho.\ \textbf{if } \pi=\pi_0 \textbf{ then } U(V_{j-1}\mu')(\mu\pi_2)\rho \textbf{ else } U(V_{j-1}\mu')(\mu\pi)\rho$$

Now it is easy to show that $(\forall i)(\exists j)[\ V_i\mu \subseteq V_j\mu'\ ]$ and vice versa. Thus $\lim_i V_i\mu = \lim_j V_j\mu'$.

Rule L: Here the proof is somewhat different since it involves comparing a circular data structure with a minimal fixed-point. Suppose

$$\mu(\pi_0) = (\text{FUNARG (LABEL } \xi\ \pi_1)\ \xi_2\ \pi_2 \ldots\ \xi_n\ \pi_n)$$

and $\mu' = \mu[(\text{FUNARG } \pi_1\ \xi\ \pi_0\ \xi_2\ \pi_2 \ldots\ \xi_n\ \pi_n)/\pi_0\ ]$

First, we claim without proof that $V\mu$ and $V\mu'$ are the minimal solutions to the following equations, respectively:

$$V\mu = \lambda\pi\rho.\ \textbf{if } \pi=\pi_0 \textbf{ then } Y\{\lambda x.\ V\mu\pi_1\rho[\ V\mu\pi_n\rho/\xi_n \ldots V\mu\pi_2\rho/\xi_2,\ x/\xi]\} \textbf{ else } U(V\mu)(\mu\pi)\rho$$

$$V\mu' = \lambda\pi\rho.\ \textbf{if } \pi=\pi_0 \textbf{ then } V\mu'\pi_1\rho[\ V\mu'\pi_n\rho/\xi_n \ldots V\mu'\pi_2\rho/\xi_2,\ V\mu'\pi_0\rho/\xi\ ] \textbf{ else } U(V\mu')(\mu\pi)\rho$$

It is simple to show that $V\mu$ satisfies the equation for $V\mu'$ so $V\mu' \subseteq V\mu$. The proof in the other direction is more difficult. The difficulty seems to be that $V\mu$ involves a loop, represented by the Y, within a loop, represented by the definition of V, while $V\mu'$ involves just one loop. To overcome this problem we will "cut" the loop represented by Y at the same time we cut the loop represented by V. First, we claim (based on continuity) that the following equations are equivalent to the ones above for defining $V\mu$.

$$V_i\mu = \perp \text{ for } i \le 0$$

$$V_i\mu = \lambda\pi\rho.\ \textbf{if } \pi=\pi_0 \textbf{ then } Y_i\{\lambda x.V_{i-1}\mu\pi_1\rho[\ V_{i-1}\mu\pi_n\rho/\xi_n \ldots V_{i-1}\mu\pi_2\rho/\xi_2,\ x/\xi]\} \textbf{ else } U(V_{i-1}\mu)(\mu\pi)\rho$$

$$Y_i = \perp \text{ for } i \le 0$$

$$Y_i = \lambda f.f(Y_{i-1}f)$$

$$V = \lim_i V_i$$

Now we prove $V_i\mu \subseteq V\mu'$ by induction on i. It is vacuously

true for $i \leq 0$, so assume it for all k<i. Then, by the induction hypothesis

$$V_i\mu \subseteq \lambda\pi\rho.\ \text{if } \pi=\pi_0 \text{ then } Y_iF \text{ else } U(V\mu')(\mu\pi)\rho$$
where $F = \{\lambda x.\ V\mu'\pi_1\rho[\ V\mu'\pi_n\rho/\xi_n \ldots V\mu'\pi_2\rho/\xi_2,\ x/\xi]\}$

The proof will be complete if we can show

$$Y_iF \subseteq V\mu'\pi_0\rho$$

This fact can also be shown by and induction on i. Assume the result for all k<i,then

$$Y_iF = V\mu'\pi_1\rho[\ V\mu'\pi_n\rho/\xi_n \ldots V\mu'\pi_2\rho/\xi_2,\ Y_{i-1}F/\xi\ ]$$
$$\subseteq V\mu'\pi_1\rho[\ V\mu'\pi_n\rho/\xi_n \ldots V\mu'\pi_2\rho/\xi_2,\ V\mu'\pi_0\rho/\xi\ ] \quad \text{by the induction hypothesis}$$
$$= V\mu'\pi_0\rho$$

IV. Remarks

The two objectives presented at the beginning of the paper can now be given more substance.

First, the general question of when an evaluation step is necessary needs to be answered. Initially, some external consideration must indicate that a particular location's value must be pursued; e.g. the user would like that value to be printed. Then, the "need to be evaluated" propagates itself to the descendants of that location according to rules peculiar to the semantics of the language. These rules were straightforward for the language studied here. Sometimes they are less so. For example, changing the semantics of IF so that

(IF p x x) = x

even when p is undefined would require simultaneous evaluation of all three parts of an IF clause. In any case, it appears that performing "outer-most" reductions, i.e. those closest to the initial source of the need, is a good heuristic. The reason is that these reductions may make some parts of the structure which are farther away unnecessary.

Second, the notion of "same step" needs clarification. Here, all that has been achieved through the use of pointers is that the advantages of evaluating an expression earlier have been retained. If the expression "3+7" arose twice in an evaluation from entirely different places there is no simple way to avoid its recomputation.

Finally, a comment on the usefulness of Scott-Strachey semantics. We can hardly claim that the lazy evaluator is "right" because it is correct with respect to the semantics defined in section III. It is obvious that the semantics can be adjusted to fit any mechanical evaluation method one chooses. One the other hand the use of a semantic model is a great aid in studying the implications of various evaluation rules without gettting involved in too many details.

Acknowledgement

Section III of this paper was primarily the work of the second author with significant help from Howard Sturgis of Xerox PARC.

References

[1] Curry,H.B. and Feys,R., *Combinatory Logic*, vol I. North-Holland, 1958.

[2] Hewitt, Carl, et. al., Behavioral semantics of non-recursive control structures, *Proceedings, Colloque sur la Programmation*, Springer-Verlag Lecture Notes in Computer Science, No. 19, 1974.

[3] Reynolds, J. R., Notes on a lattice-theoretic approach to the theory of computation, Lecture notes, Syracuse Unversity, 1971.

[4] Scott, D. and Strachey, C., Toward a mathematical semantics for computer languages, *Proc. of the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn*, and PRG-6 Oxford University Computing Laboratory, 1971

[5] Stoy, J., The Scott-Strachey approach to the mathematical semantics of programming languages, Course notes at M.I.T. Project MAC, 1973.

[6] Vuillemin, J., Correct and optimal implementations of recursion in a simple programming language, *Journal of Computer and System Sciences*, vol. 9, No. 3, December 1974.

[7] Wadsworth, Christopher, Semantics and Pragmatics of the Lambda-calculus, PhD. thesis, Oxford, 1971