ב״ה

NON MEMORIZING AND FORGETTING EVALUATION STRATEGIES FOR FUNCTIONAL LANGUAGES

R.B. Yehezkael*

Jerusalem College of Technology, Hawaad Haleumi 21, Jerusalem 91160, ISRAEL

E-mail address: rafi@mail.jct.ac.il, FAX:02-6422075 Tel: 02-6751111

November 1990 - כסלו תשנ״א, revised November 1991 - כסלו תשנ״ב

Formatting and minor changes made April 2001 - אייר תשס״א

ABSTRACT: We present techniques for expression evaluation for functional (list processing) languages which for a certain class of functions and expressions, evaluates the expression without creating storage structure for the final result, i.e. evaluation and output are interleaved so that no storage structure for the final result is created. This enables the creation of large files from these languages without creating list structure (cons cells) for the data in the file.

The other approach presented is to mark the memory structure for the final result created during the computation and "forget quickly" the marked memory structure in the process of output. The advantage of this approach is that the inherent paralellism in functional languages can be more effectively utilised.

KEYWORDS: Non-memorizing evaluation, Forgetting evaluation, Eager evaluation, Lazy evaluation, Functional langauge, Declarative language.

---

* Previously Haskell

CONTENTS

1 INTRODUCTION

The functional programming style has many advantages. Programs are often concise, it is easier to reason about them, there is greater oppurtunity to evaluate expressions in parallel. A disadvantage of functional languages is the memory overheads they incur, and the technique for saving stack space on tail recursive calls (calls which return the function value) are well known. Many techniques for storage optimization of the evaluation process are presented in [Harrison, 1982].

In this paper we present an alternate direction for storage reduction which we describe by an analogy. A well known phenomenom among beginning students of computer science is that when asked to write a program to reverse a list, they write instead a program to print out the list reversed without creating a memory structure for the reversed list. We present in this paper designs for interpreters which interleave evaluation and output and in a systematic way avoid creating memory structure for the final result which is output.

Regarding storage optimization which may be achievable by interleaving input and evaluation, we do not think that there are benefits to be reaped over the technique of (lazy) input streams combined with tail recursion optimization. While lazy input streams gives functional languages the ability to handle large volumes of input data, our techniques gives us the ability for handling large volumes of output data in the functional programming style. We can carry out computations with functions which return for example the projection of a relation in a database. Our approach can also be looked at as being very eager to output the computed prefix of the value of the expression to be output. Thus the combination of lazy input and eager forgetting output seems a good combination to reduce storage requirements for functional languages.

Furthermore, the interpreters presented may be executed with either an eager or lazy cons funtion, reaping the same benefit.

2 CONVENTIONS

We adopt the following conventions in the definitions we present:

v - the name of the variable.
vl - list of values typically the binding list which binds the list of parameter variables of a function to their values.
e - expression to be evaluated
el - list of expressions to be evaluated, usually the arguments of a function.
fn - function name

The following are the main functions of the interpreter.

eval(e, vl) - value of the expression e with variable bindings given by vl.
evlis(el, vl) - list of values of the expressions in el when evaluated with bindings vl.
evcond(e, vl) - like eval but for conditional expressions only.
apply(fn, el, vl) - apply the function fn to the expression list el where the values of the variables occuring in the expression list are given by the binding vl.
applybasic(fn, vl) - apply a basic function to a list of values or to a list which is waiting to receive its values.
valueof (v, vl) - value of the variable v with bindings list vl, (if v is the i'th variable in the list of parameters of the function where v is declared, then valueof(v ,vl) typically equals the i'th element of vl.)

The following are selector functions used for accessing parts of expressions and function definitions.

functionpart(e) - the function part of the expression e. (Note, the function part itself may be an expression in which case which its value should be a function)
argumentspart(e) - the arguments part of the expression e
expressionpart(fn) - the expression defining the value of fn.
conditionpart(e) - the condition of the conditional expression
thenpart(e) - the expression after the then.
elsepart(e) - the expression after the else.
first(l), second(l) - first, second element of list l.
rest(l) - the elements of a non empty list l excluding the first.
cons(e, l) - list with one more element e than l where e is the first element of the said list.

For expository reasons, we present interpreter designs for a simple functional list processing language (i.e. no nested definitions, global variables or lambda expressions, but recursion, mutual recursion and expressions which evaluate to a function are permitted).

3 THE STANDARD INTERPRETER

The structure of this interpreter is similar to a standard LISP interpreter. In order that the interpreter can handle expressions which evaluate to a function, the convention is adopted that a function is like a constant and evaluates to itself. This is reflected in the definition of eval which evaluates the function part of an expression in its call to apply. The function definitions of the interpreter functions eval, evlis, evconditiomal, apply are as follows.

```
eval(e, vl) =

  if e is a constant or a function then e
  else if e is a variable then valueof(e, vl)
  else if e is a conditional expression then evconditional(e, vl)
  else if e is a function with arguments
       then apply(eval(functionpart(e), vl), argumentspart(e), vl)

  else error

evlis(el, vl) =

  let i, rl be local variables
  rl := a new list having the same length as el;
  for i := 1 to length of el do
      set the i'th component of rl equal to eval(i'th component of el, vl);
  return rl;

evconditional(e, vl) =

  if eval(conditionpart(e), vl)
  then eval(thenpart(e), vl)
  else eval(elsepart(e), vl)

apply(fn, el, vl) =

  if fn is a basic function
       then applybasic(fn, evlis(el, vl))
  else if fn is a user defined function
       then eval(expressionpart(fn), evlis(el, vl))

  else error
```

The function applybasic applies a basic function to its arguments. The functions valueof, functionpart, expressionpart, conditionpart, thenpart, elsepart are selector functions used for accessing parts of expressions and function definitions. They have straightforward definitions.

THE INTERACTIVE LOOP

The outermost loop of the interpreter controlling the interaction with the user is typically of the form:

```
while there is more input do
        print(eval(readexpression, emptyvaluelist));
```

NOTES:

1) In practice the functions evlis, evconditional and apply can be implemented as macros which saves paramater passing overheads. Evaluation would typically take place with calls returning the function result (tail recursion) optimized for stack usage.

2) Having presented the standard interpreter, a succession of interpreters will be presented all of which are based on the standard interpreter. As an aid to readability, the major differences with respect to the standard interpreter will be highlighted in bold type.

4 NON MEMORIZING EVALUATION

We now present techniques for interleaving computation and output of the partially computed result. We give examples of computation sequences of these interpreters and also indicate where benefits are to be gained.

4.1 THE NON-MEMORIZING INTERPRETER - VERSION 1

We add functions to the standard interpreter which interleave output and evaluation in such a way that no list structure for the final result is created. In this first version of this kind of interpreter, we treat the cons function only, and discuss after version 2 the handling of functions like append, list, reverse. We add the following definitions to the standard interpreter.

p_eval(...) equivalent to print(eval(...))
p_evconditional(...) equivalent to print(evconditional(...))
p_apply(...) equivalent to print(apply(...))
p_cons(...) equivalent to print(cons(...))

The first three of the above functions have a very similar structure to eval, evconditional, apply and are obtained from them by moving the print inside their definition in the obvious way.

```
p_eval(e, vl) =

  if e is a constant or a function then print(outfile, e)
  else if e is a variable then print(outfile, valueof(e, vl))
  else if e is a conditional expression then p_evconditional(e, vl)
  else if e is a function with arguments
       then p_apply(eval(functionpart(e), vl), argumentspart(e), vl)

  else error

p_evconditional(e, vl) =

  if eval(conditionpart(e), vl)
  then p_eval(thenpart(e), vl)
  else p_eval(elsepart(e), vl)

p_apply(fn, el, vl) =

  if fn = cons then p_cons(el, vl)
  else if fn is a basic function
       then print(outfile, applybasic(fn, evlis(el, vl)))
  else if fn is a user defined function
       then p_eval(expressionpart(fn), evlis(el, vl))
  else error

p_cons(el, vl) =

  print(outfile, '(');
  p_eval((first(el), vl);
  print(outfile, ' . ');
  p_eval((second(el), vl));
  print(outfile, ')');
```

THE INTERACTIVE LOOP

The outermost loop of the interpreter now becomes:

```
while there is more input do
       p_eval(readexpression, emptyvaluelist);
```

NOTES:

1) We assume sequential "left to right" evaluation.

2) With the above definition of p_cons, lists are output in fully
bracketed dot notation, i.e. (1 2 3) would appear as (1 . (2 . (3 . ()))).
To get standard list output we need to modify the definition of p_cons and also add functions p1_eval, etc. which evaluates and prints a list without the outermost brackets. The details are given in version 2 of the interpreter.

4.2 THE NON-MEMORIZING INTERPRETER - VERSION 2

Firstly the previous definition of p_cons is modified as follows:

```
p_cons(el, vl) =

  print(outfile, '(');
  p1_cons(el,vl);
  print(outfile, ')');
```

The functions which we add to version 1 of the non memorizing interpreter are p1_eval, p1_evconditional, p1_apply(...),p1_cons(...) which are similar to p_eval, p_evconditional, p_apply(...),p_cons(...) except that lists are printed without their outermost brackets. We assume the existence of a procedure print1 which prints any evaluated value without outermost brackets. Now for their definitions.

```
p1_eval(e, vl) =

  if e is a constant or a function then print1(outfile, e)
  else if e is a variable then print1(outfile, valueof(e, vl))
  else if e is a conditional expression then p1_evconditional(e, vl)
  else if e is a function with arguments
       then p1_apply(eval(functionpart(e), vl), argumentspart(e), vl)

  else error
```

```
p1_evconditional(e, vl) =

  if eval(conditionpart(e), vl)
  then p1_eval(thenpart(e), vl)
  else p1_eval(elsepart(e), vl)

p1_apply(fn, el, vl) =

  if fn = cons then p1_cons(el, vl)
  else if fn is a basic function
       then print1(outfile, applybasic(fn, evlis(el, vl)))
  else if fn is a user defined function
       then p1_eval(expressionpart(fn), evlis(el, vl))

  else error

p1_cons(el, vl) =

  p_eval((first(el), vl);
  p1_eval((second(el), vl));
```

THE INTERACTIVE LOOP

This remains as before, that is:

```
while there is more input do
       p_eval(readexpression, emptyvaluelist);
```

4.3 HANDLING FUNCTIONS LIKE APPEND, LIST, REVERSE

Actually, nothing special needs to be done regarding the functions APPEND and LIST we can use their standard definitions unaltered. According to the context, the interpreter will simply print the appended list instead of computing it, and similarly with LIST. However, for reasons of completeness we show how to handle these functions explicitly. The function REVERSE must of course be explicitly handled. We have to modify the functions p_apply and p1_apply in the obvious way to test for these functions and activate functions p_append, p1_append, p_list, p1_list, p_reverse, p1_reverse whose definitions follow.

```
p_append(el, vl) =

  print(')');
  p1_append(el, vl);
  print('(');
```

```
p1_append(el, vl) =

  let i be a local variable;
  for i:=1 to length of el do
      p_eval(i'th element of el, vl);
```

```
p_list(el, vl) =

  print(')');
  p1_list(el, vl);
  print('(');
```

```
p1_list(el, vl) =

  let i be a local variable;
  for i:=1 to length of el do
      p1_eval(i'th element of el, vl);
```

```
p_reverse(el, vl) =

  print(')');
  p1_reverse(eval(first(el), vl));
  print('(');
```

```
p1_reverse(l) =

  if l is not ()
  then begin p1_reverse(rest(l));
             print(first(l));
       end;
```

AN EXAMPLE:

Database projection of the second component of a list of tuples.

```
(DEFUN P2 (TUPLES)
  (IF (NULL TUPLES)
      ()
      (CONS (SECOND (FIRST (TUPLES)) (P2 (REST TUPLES)))))
```

Now p_eval((P2 ((john eleceng) (mary compsci) (jacob biology))) ()) would result in the following computation sequence where the output is indicated on the right.

```
                                                                    OUTPUT
p_eval( (P2 ((john eleceng) (mary compsci) (jacob biology))), ()) =

p_apply(P2, (((john eleceng) (mary compsci) (jacob biology))), ()) =

p_eval((IF (NULL TUPLES)
         ()
         (CONS (SECOND (FIRST (TUPLES)) (P2 (REST TUPLES)))),
      (((john eleceng) (mary compsci) (jacob biology))) ) =
p_evconditional((IF (NULL TUPLES)
                    ()
                    (CONS (SECOND (FIRST (TUPLES)) (P2 (REST TUPLES)))),
                (((john eleceng) (mary compsci) (jacob biology))) ) =

p_eval((CONS (SECOND (FIRST (TUPLES)) (P2 (REST TUPLES))),
       (((john eleceng) (mary compsci) (jacob biology))) ) =

p_apply(CONS, ((SECOND (FIRST (TUPLES)) (P2 (REST TUPLES))),
        (((john eleceng) (mary compsci) (jacob biology))) ) =

p_cons((SECOND (FIRST (TUPLES)),
       (P2 (REST TUPLES)),
       (((john eleceng) (mary compsci) (jacob biology))) ) =

print(outfile, '(');                                                          '('
p1_cons((SECOND (FIRST (TUPLES)),
        (P2 (REST TUPLES)),
        (((john eleceng) (mary compsci) (jacob biology))) )
print(outfile, ')'); =

p1_cons((SECOND (FIRST (TUPLES)),
        (P2 (REST TUPLES)),
        (((john eleceng) (mary compsci) (jacob biology))) )
print(outfile, ')'); =

p_eval((SECOND (FIRST (TUPLES)),
       (((john eleceng) (mary compsci) (jacob biology))) );
p1_eval((P2 (REST TUPLES)),
        (((john eleceng) (mary compsci) (jacob biology))) );
print(outfile, ')'); =

p_apply(SECOND, ((FIRST (TUPLES))),
        (((john eleceng) (mary compsci) (jacob biology))) );
p1_eval((P2 (REST TUPLES)),
        (((john eleceng) (mary compsci) (jacob biology))) );
print(outfile, ')'); =
print(applybasic(SECOND, ((john eleceng))))
p1_eval((P2 (REST TUPLES)),
        (((john eleceng) (mary compsci) (jacob biology))) );
print(outfile, ')'); =

print(outfile, eleceng)                                                              eleceng
p1_eval((P2 (REST TUPLES)),
        (((john eleceng) (mary compsci) (jacob biology))) );
print(outfile, ')'); =

p1_eval( (P2 (REST TUPLES)),
        (((john eleceng) (mary compsci) (jacob biology))) );
print(outfile, ')'); =

p1_apply(P2, ((REST TUPLES)),
         (((john eleceng) (mary compsci) (jacob biology))) );
print(outfile, ')'); =

p1_eval((IF (NULL TUPLES)
            ()
            (CONS (SECOND (FIRST (TUPLES)) (P2 (REST TUPLES)))),
        (((mary compsci) (jacob biology))) );
print(outfile, ')'); =
...
...
p1_cons((SECOND (FIRST (TUPLES)),
        (P2 (REST TUPLES)),
        (((mary compsci) (jacob biology))) ) =
print(outfile, ')'); =
...
...
p_eval((SECOND (FIRST (TUPLES)),
       (((mary compsci) (jacob biology))) );
p1_eval( (P2 (REST TUPLES)),
       (((mary compsci) (jacob biology))) );
print(outfile, ')'); =
...
...
print(outfile, compsci)                                                              compsci
```

```
p1_eval( (P2 (REST TUPLES)),
      (((mary compsci) (jacob biology))) );
print(outfile, ')'); =
...
...
p1_eval((IF (NULL TUPLES)
          ()
          (CONS (SECOND (FIRST (TUPLES)) (P2 (REST TUPLES)))),
       (((jacob biology))) );
print(outfile, ')'); =
...
...
print(outfile, biology)                                                    biology
p1_eval( (P2 (REST TUPLES)),
      (()) );
print(outfile, ')'); =
...
...
p1_eval((),
       (()) );
print(outfile, ')'); =

print1(outfile, ())  {NO OUTPUT as outermost brackets not printed}
print(outfile, ')');                                                       ')'
```

NOTES:

1) No cons cells for the output were created - the result list was output directly.
2)The size of the expressions in the computation sequence essentially represents the stack depth. Note the cyclic behaviour in the evaluation where:

```
print(second element of tuple1)
p1_eval((P2 (REST TUPLES)) ((tuple1 tuple2 tuple3 tuple4...)))
print(')');
```

becomes:

```
print(second element of tuple2)
p1_eval((P2 (REST TUPLES)) ((tuple2 tuple3 tuple4...)))
print(outfile, ')');
```

This can of course be repeated indefinitely without increasing expression size (stack depth). We therefore conclude that the stack depth is bounded and no further stack is needed regardless of the number of tuples. In other words, the interpreter could project a very large list of tuples without requiring large amounts of memory for the stack or for the cons cells to be output.

ANOTHER EXAMPLE:

The evaluation of p_eval((APPEND (1 2) (REVERSE (CONS 3 (4))), ()) is given below. Output is given on the extreme right.

```
                                                                  OUTPUT
p_eval((APPEND (1 2) (REVERSE (CONS 3 (4)))), ()) =
...
...
print(outfile, '(');                                              ')'
p1_eval((1 2) ());
p1_eval((REVERSE (CONS 3 (4))), ())
print(outfile, ')'); =
...
...
print1(outfile, (1 2));                                           1 2
p1_eval((REVERSE (CONS 3 (4))), ())
print(outfile, ')'); =
...
...
p1_eval((REVERSE (CONS 3 (4))), ())
print(outfile, ')'); =
...
...
p1_reverse((3 4), ());
print(outfile, ')'); =
...
...
p1_reverse((4), ());
print(outfile, 3);
print(outfile, ')'); =
...
...
print(outfile, 4);                                                      4
print(outfile, 3);                                                      3
print(outfile, ')');                                                    ')'
```

NOTE: Only one cons took place in the above evaluation, when the parameter of reverse is calculated. No cons took place in either the append or in actually doing the reverse. The following more elaborate definition of p_reverse and p1_reverse would avoid even this cons.

4.4 A MORE ELABORATE TECHNIQUE TO HANDLE REVERSE

```
p_reverse(el, vl) =

  print(')');
  p1_reverse(el, vl);
  print('(');

p1_reverse(el, vl) =

  case first(el) of  {According to the parameter of reverse}
    cons(...): begin p1_eval(second parameter of cons, vl);
                     p_eval(first parameter of cons, vl);
               end;
    append(...): begin let i be a local variable
                       for i:= number of parameters of list downto 1 do
                       p1_reverse(i'th parameter of list, vl);
                 end;
    list(...): begin let i be a local variable
                     for i:= number of parameters of list downto 1 do
                     p_reverse(i'th parameter of list, vl);
               end;
    reverse(...): begin p1_eval(parameter of reverse, vl); {reverse of reverse}
                  end;
    otherwise p11_reverse(eval(first(el), vl)); {Essentially previous defn.}
```

```
p11_reverse(l) =

if l is not ()
then begin p11_reverse(rest(l));
           print(first(l));
     end;
```

4.5 WHERE ARE BENIFITS GAINED ?

Benifits are gained in those places in a function definition which contribute to the final result to be output. Forms such as cons(...), or append(...) or list(...) or reverse(...) or their combinations which contribute to the output would require no cons cells for their results and possibly for their parameters when evaluated by the previous interpreter.

5 FORGETTING EVALUATION

We now discuss techniques for marking and forgetting the final result, particularly a technique which exploit parallelism and forget the final result on the fly. We also indicate where benifits are to be expected.

5.1 THE FORGETTING INTERPRETER - VERSION 1 (INTRODUCING SOME PARALLELISM)

The principle behind this interpreter is that the cons cells created for the output are marked or tagged and immediately released when output by the print procedure. Furthermore, the print and evaluation procedures now run in parallel such that the cons cells produced requiring output, would be released on the fly by the print process after output. In order to describe this clearly, we have to add to the definition of the interpreter the procedure spawn(e, result) which evaluates the expression e as a subprocess and puts its value in result. It is similar to the assignment result := e except that the calling process can continue immediately with result initially set to a "waiting for a value code" and when e returns a value, result gets assigned. A useful variant is the form spawn(e, HERE) which spawns the evaluation of e as a subprocess and returns the value at the point where the spawn is written. An example will clarify what we mean. Let f(x,y) and g(a,b) be functions with formal parameters x,y and a,b. The expression f(1,spawn(g(2,3),HERE)) will be equivalent to simultaneously evaluating f(1,y) and spawn(g(2,3),y) that is, the spawn puts the result of g(2,3) as the value of the formal parameter y of f.

We additionally assume that variables and list cells and components are initialized to a "waiting for a value code" and that we can therefore determine whether or not a variable or list component has received its value yet. Assignment is of course once only. We also assume functions like first, second, rest will wait for a value if the list component is waiting for a value. Similarly, applybasic(fn, vl) will wait until all the elements of vl have been evaluated and only then apply the function. Similarly a wait will occur if a test or condition is waiting for a value.

We now give the definition of this interpreter.

```
f_eval(e, vl) =

  if e is a constant or a function then e
  else if e is a variable then valueof(e, vl)
  else if e is a conditional expression then f_evconditional(e, vl)
  else if e is a function with arguments
        then f_apply(eval(functionpart(e), vl), argumentspart(e), vl)

  else error

f_evconditional(e, vl) =

  if f_eval(conditionpart(e), vl)
  then f_eval(thenpart(e), vl)
  else f_eval(elsepart(e), vl)

f_apply(fn, el, vl) =

  if fn = cons
        then f_cons(el, vl)
  else if fn is a basic function
        then applybasic(fn, evlis(el, vl));
  else if fn is a user defined function
        then f_eval(expressionpart(fn), evlis(el, vl));

  else error
```

```
f_cons(el, vl) =

  let result be a local variable
  result := address of a new cons cell;
  mark the cons cell for release after printing;
  spawn(f_eval(first(el), vl), first field of result);
  spawn(f_eval(second(el), vl), rest field of result );
  return result
```

THE INTERACTIVE LOOP

The outermost loop of the interpreter controlling the interaction with the user now becomes:

```
while there is more input do
      f_print(spawn(f_eval(readexpression, emptyvaluelist), HERE));
```

f_print prints and releases all cons cells which were marked to be freed after printing. It can free the cons cell after printing the first component. It looks something like

```
f_print(l) is

  let address be a local variable;
  if l is an atom then write(outfile, l)
  else begin write(outfile,'(');
          while l is not empty do
              begin
                address := l;
                f_print(first(l));
                l := rest(l);
                   if cons cell pointed at by address is marked to be
                     freed after printing
                   then free cons cell pointed at by address;
              end;
          write(outfile,')');
        end;
```

NOTE: Though it is not explicitly indicated, waiting occurs during the printing process if a value is waiting to be computed.

5.2 THE FORGETTING INTERPRETER - VERSION 2 (A HIGHLY PARALLEL VERSION)

Parallelism was introduced in the previous interpreter only when evaluating a cons of the final result. As the function f_apply used the standard interpreter function evlis to evaluate the parameters of the function, no parallelism occurs when evaluating the arguments of any other cons or or any other function. By modifying the definition of f_apply to activate the highly parallel interpreter below, we are able utilize the inherent parallism in function parameter evaluation. When we do this, the function f_eval gains the benefits of parallelism and still marks cons cells created during the computation for the output.

MODIFIED DEFINITION OF f_apply

```
f_apply(fn, el, vl) =

  **if fn = cons**
       **then f_cons(el, vl)**
  else if fn is a basic function
       **then applybasic(fn, ll_evlis(el, vl));**
  else if fn is a user defined function
       **then f_eval(expressionpart(fn), ll_evlis(el, vl));**

  else error
```

The other interpreter functions are now given. ll_eval and ll_evconditional are similar to eval and evconditional of the standard interpreter. The definition of ll_evlis is new and can exploit the parallelism inherent in function parameter evaluation. The definition of ll_apply is similar to the modified version of f_apply except that the cons cell is not marked.

```
ll_eval(e, vl) =

  if e is a constant or a function then e
  else if e is a variable then valueof(e, vl)
  else if e is a conditional expression then ll_evconditional(e, vl)
  else if e is a function with arguments
       then ll_apply(eval(functionpart(e), vl), argumentspart(e), vl)

  else error
```

```
ll_evlis(el, vl) =

  let i, rl be local variables
  rl := address of a new list having the same length as el;
  {recall that all components of rl are set to "waiting for a value code"};
  for i := 1 to length of el do
      spawn(ll_eval(i'th component of el, vl), i'th component of rl);
  return rl;

ll_evconditional(e, vl) =

  if ll_eval(conditionpart(e), vl)
  then ll_eval(thenpart(e), vl)
  else ll_eval(elsepart(e), vl)

ll_apply(fn, el, vl) =

  if fn = cons
       then ll_cons(el, vl)
  else if fn is a basic function
       then applybasic(fn, ll_evlis(el, vl));
  else if fn is a user defined function
       then ll_eval(expressionpart(fn), ll_evlis(el, vl));

  else error

ll_cons(el, vl) =

  let result be a local variable
  result := address of a new cons cell;
  spawn(f_eval(first(el), vl), result^.first);
  spawn(f_eval(second(el), vl), result^.rest);
  return result
```

THE INTERACTIVE LOOP

The interactive loop remains unchanged, namely:

```
while there is more input do
        f_print(spawn(f_eval(readexpression, emptyvaluelist), HERE));
```

where f_print is as before.

5.3 HANDLING FUNCTIONS LIKE APPEND, LIST, REVERSE

Regarding the functions APPEND and LIST, nothing special needs to be done as the interpreter will mark the cons cells created by these functions with no further special action. The function REVERSE needs explicit handling and we must check for it in the interpreter and activate f_reverse when appropriate. The function f_reverse would just return a coded and marked cons cell to indicate that the following list should be printed backwards, for example, f_reverse(el,vl) = f_cons("reverse code",f_eval(first(el),vl)). Then only at the time of printing would the list be printed out backwards and marked cons cells released. The changes to f_print are not given as they are of a similar kind to the changes made in the non-memorizing interpreter. This also seems to be the best way of handling reverse as it does not interfere with the parallel execution.

5.4 WHERE ARE BENIFITS GAINED ?

As before, benifits are gained in those places in a function definition which contribute to the final result to be output. Forms such as cons(...), or append(...) or list(...) or reverse(...) or their combinations which contribute to the output would have their cons cells marked and "quickly forgotten" after printing.

6 CONCLUSION

We have presented techniques for memory optimization at the cost of having an interpreters about twice or thrice as large as the standard interpreters. We think that this cost is justified, as memory optimization in declarative language is extremely important so as to enable a wider class of applications to be handled by these languages.

7 REFERENCES

Eisenbach, S. editor [1987] "FUNCTIONAL PROGRAMMING: Languages, Tools and Architectures", publ. Ellis Horwood.

Harrison, P.G. [1982] "Efficient Storage Management for Functional Languages", Computer Journal Vol. 25 No. 2 pp. 264-271.

Henderson, P. [1980] "Functional Programming: Application and Implementation", publ. Prentice Hall.