Lazy Evaluation and Parameter Transfer

Scheme comes with two interesting predefined functions: **`delay`** and **`force`**. The basic purpose of **`delay`** is to postpone the evaluation of an expression. The result of the application **`(delay`** ***`expression`***`)` is an object called *promise*. The delayed ***`expression`*** is packed unevaluated within the promise. The role of **`force`** is to perform the evaluation of a promise and to return the result of the promise that is precisely the result of the expression packed in the promise. Therefore, evaluating (or forcing) a promise means evaluating the delayed expression according to the following rules:

- The delayed expression is evaluated once only, when the promise is forced for the first time. The value obtained is the *first value* of the promise.
- When the promise is subsequently evaluated, the result returned is the *first value* of the promise.

Obviously, once available, the first value of the promise must be cached within the promise. Therefore, a promise has a state, and forcing the promise for the first time has side effects. Moreover, observe that the first value of the promise is the result returned by *the first completed* **`force`** applied on the promise. Consider the **`toy`** example, where **`set!`** is the assignment operator i.e.

`(set!` ***`variable expression`***`)`

binds the ***`variable`*** to the value of the ***`expression`***.

```
(define counter 0)
(define toy
  (delay (if (< counter 3)
             (begin (set! counter (+ counter 1))
                    (force toy)
                    counter)
             100)))                    Delayed expression

(force toy)
```

The value of the delayed expression while the promise **`toy`** is forced is shown in table 1. Notice that the first completed **`force`** corresponds to **`counter=3`** and the value returned by this **`force`** operation is 100. This is the *first value* of the promise and it should stay unmodified. Notice also that when this value is computed there are other three incomplete **`force`** operations under way. The results corresponding to these subsequently completed **`force`** operations are discarded.

	apply **`force`** →				return from **`force`** →		
`(force toy)`	top-level	second	third	fourth	third	second	top-level
`counter`	0	1	2	3	3	3	3
value of the delayed expression	?	?	?	100	3	3	3
result of top-level **`(force toy)`**	?	?	?	?	?	?	100

Table 1. The behavior of **`(force toy)`**

In order to implement the evaluation of a promise and, implicitly, the implementation of **`delay`** we have to solve three problems:

- Saving the current computational context of the delayed expression. When the promise is forced, the expression must evaluate in its own original computational environment. For example, the delayed expression in the example above must see the top-level variable **counter**.
- Saving and preserving the result of the first evaluation.
- Distinguishing between the first value of the promise and its subsequent values.

Consider that a function called **make-promise** constructs a promise for a delayed ***expression***. Then saving the computational context of the ***expression*** is the easiest problem. The ***expression*** is closed within an **0**-ary function and **make-promise** is applied onto this closure. Therefore, we define the expression **(delay *expression*)** to have the same meaning as the call **(make-promise (lambda () *expression*))**. This effect can be obtained using the following macro-definition[1]:

```
(define-macro delay
  (lambda (expression)
      `(make-promise (lambda () ,expression))))
```

In general, a macro-definition has the format

(define-macro *name function-expr*)

Considering that **F** is the result of the ***function-expr***, the call of the macro proceeds as follows:

1. The *unevaluated* arguments are transferred to **F**.
2. The result **R** returned by **F** is substituted for the macro call in the computational environment of the call.
3. The computation of the macro call proceeds with the evaluation of the result **R**.

The macro call **(delay *expression*)** then is equivalent to the function call **(make-promise (lambda () *expression*))**. In order to solve the other two problems related to the evaluation of a promise, we have to consider the way **make-promise** works. The function can be defined as follows.

```
(define make-promise
  (lambda (closure)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()                                    promise
        (if result-ready?
            result
            (let ((x (closure)))
              (if result-ready?
                  result
                  (begin (set! result-ready? #t)
                         (set! result x)
                         result)))))))))
```

[1] The macro-definition uses the operator ` called quasiquote. The quasiquote is useful for constructing a list that has some elements unknown in advance. If there is no comma within the ***list_template***, the result of **`*list_template*** is the result of '***list_template***. If ***,expression*** occurs within ***list_template*** the result of ***expression*** replaces ***,expression.*** within '***list_template***. If ***,@expression*** occurs within ***list_template***, then the ***expression*** must evaluate to a list; the opening and closing parentheses of the list are stripped away and the elements of the list replace the ***,@expression*** within '***list_template***.

Since a promise may be recursive, forcing such a promise may cause the promise to be subsequently forced before *its first value* is known (see the **toy**). This requires testing for the **result-ready?** flag. If the flag is true the cached result is returned and the result of the current evaluation of the delayed expression is ignored. In this way, the *first value* of the promise is always returned as the correct value of the promise.

The result of **make-promise** (a promise) is an **0**-ary closure. It closes the delayed expression – itself in the form of an **0**-ary closure – and the variables **result-ready?** and **result**. These variables are dynamically created when **make-promise** is called and are hidden within the promise. Moreover, these variables live as long as the promise lives (the extent of Scheme variables is unlimited).

The **force** function has a simple task to do: to call the promise. It is defined simply as:

```
(define force
    (lambda (promise)  (promise)))
```

A computation process that uses delayed expressions whose results are cached is termed as lazy evaluation. In the sequel we consider two applications of lazy evaluation: streams and call by need.

Streams revisited

Why lazy evaluation is important when representing infinite objects? First, the part already computed from an infinite object need not be recomputed. Second, cyclic objects can be represented conveniently. For instance, consider the infinite graph in figure 1. We have to represent the graph in such a way that the **ls** links behave like physical links. Given a node, say **n3 = (rs (rs (rs root)))**, we would like to enforce the physical equality: **(ls n3)** is the same node as **(rs (rs root))**.

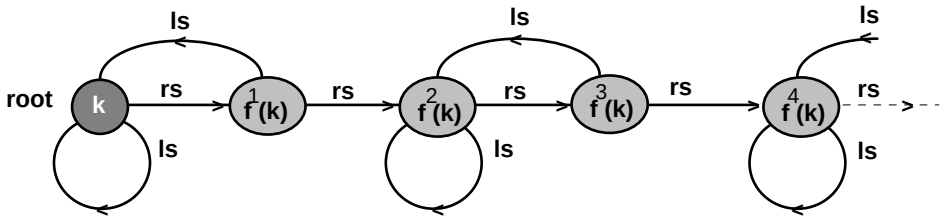
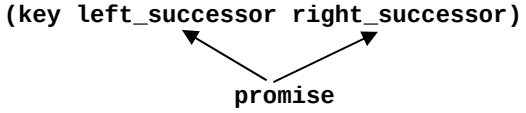


Figure 1 A cyclic infinite graph

A node of the graph is represented as a list where the first element is the key, and the second and third elements are the left, respectively, the right successor of the node. The successors are promises, which allow for a finite representation of the graph.



For clarity, we will define a basic constructor of a node and selectors of data from a node.

```
(define node list)  ; builds a node
(define key (lambda (node) (car node)))
; returns the key of node
```

```
(define ls (lambda (node) (force (cadr node))))
; returns the left successor of node
(define rs (lambda (node) (force (caddr node))))
; returns the right successor of node
```

The constructor of the graph is a function called **gen_graph**, which uses two auxiliary constructors:

- **even** for a node whose **ls** link goes back to the predecessor node in the graph. For being able to satisfy the node identity requirement, **even** gets its predecessor as a parameter. The predecessor is cached in a promise and stays unmodified.
- **odd** for a node whose **ls** link points to the node itself. Here again the self-pointer is cached in a promise.

Not only the graph is like a physical pointer structure but, moreover, once a node is computed it need not be recomputed.

```
(define gen_graph
  (lambda (k f)
    (letrec ((even (lambda (k predecessor)
                     (node k (delay predecessor) (delay (odd (f k))))))
             (odd (lambda (k)
                    (letrec ((this (node k (delay this)
                                           (delay (even (f k) this)))))
                      this))))
      (odd k))))

(define root (gen_graph 1 (lambda (x) (+ x 1))))

(let ((n3 (rs (rs (rs root)))))
   (and (eq? (ls n3) (rs (rs root)))
        (eq? (ls (ls n3)) (rs (rs root)))))
#t
```

Parameter transfer modes

The normal-order evaluation and applicative-order evaluation have several variations in programming: call by value, call by sharing, call by copying, call by reference, call by name, call by need.

Call by value. The value of the argument is assigned to the corresponding formal parameter. The formal parameter plays the role of a temporary variable used within the body of the function.

Call by sharing is a variant of call by value. It is used in languages such as Scheme, Lisp, ML where the objects are handled by implicit and explicit references. A reference is like a pointer to the referred object. What is transferred is the pointer. In this way the called function and the calling environment share the referred object. The modification of the formal parameter is not felt in the calling environment. Nevertheless, the modifications performed on the referred (pointed) object are felt in the calling environment.

```
(define L '(a b c))
(define fun (lambda (FL) (set! FL ‘(x)) FL))

(fun L)
(x)
L
(a b c)
```

The function **fun** modifies the value of its binding variable **FL**. The modification is not felt at the top-level.

```
(define foo (lambda (FL)
              (append! FL '(d e))))
(foo L)
(a b c d e)
L
(a b c d e)
```

The function **foo** appends destructively[2] the list referenced by **FL** and the local list **(d e)**. The first list, referenced by **FL**, is modified and the modification is felt at the top-level.

Some languages work with explicit references, i.e. variables can take references as values of a "reference" type. An explicit reference is like a container that stores the referred object. If a function has a formal parameter of a "reference" type then what is transferred on call is the container itself. If modifying the formal parameter means modifying the contents of the container bound to the formal parameter, then the effect will be felt in the calling environment. As an example, consider the Caml example below that uses explicit references.

```
let r = ref 1;
-: int ref = ref 1
```

A reference to the object *"integer 1"* has been created. The reference is a container which stores the object *"integer 1"*. The value of the variable **r** is the reference i.e. the container.

```
let f = fun(x) -> x:=!x+1;
f: int ref -> unit = <fun>
```

The variable **f** is bound to a function. The function takes a reference to an integer (a container that stores an integer) and modifies the value referred (the contents of the container). Here **!x** means the value referred by the reference bound to **x**; ***x:=expr*** means "store the value of ***expr*** into the reference bound to **x**", in other words store the value of ***expr*** into the container bound to **x**.

```
f(r);;
!r;;
-: int ref = 2
```

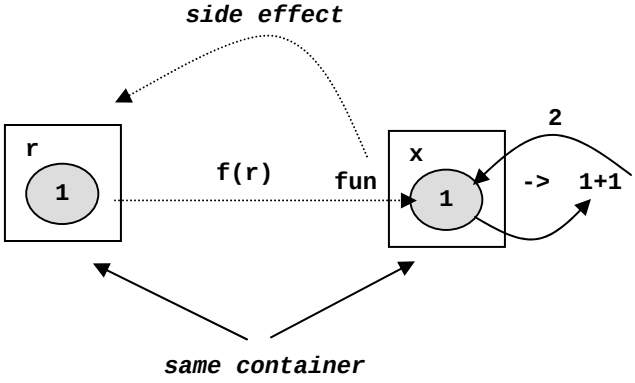


Figure 3. Working with explicit references

The argument of **f(r)** is the reference bound to **r**. During the evaluation of **f(r)** the top-level variable **r** and the formal parameter **x** share the same value: the same container that

[2] The **cdr** field of the last cell is modified to point to the list **(d e).**

stores an integer (see figure 3). The modification performed by **f**, i.e. replacing **1** by the result of **1+1** in the reference, is felt at the top-level. Implicitly the object referred by **r** will be different although the value of **r** (the container standing as the reference) is not modified.

Call by copying. The value of the argument is copied into the formal parameter when the function is called and, eventually, back to the argument when the function returns (the function has side effects). According to the copying actions performed the formal parameters can be classified as:

- **in** (the value of the argument is copied into the corresponding formal parameter when the function/procedure is applied);
- **out** (the value of the formal parameter is copied to the actual parameter when the function/procedure returns), and
- **inout** (equivalent to **in** and **out**).

See the textbook, section *Parameter Modes in ADA* for an interesting discussion on call by copying and the importance of the way it is implemented).

Call by reference. The address of the argument is transferred. The argument does not necessarily need be a variable (as required in Pascal, for example). If the argument is an expression a temporary variable is created to hold the value of the expression and the address of the variable is passed to the called function/procedure. The modifications of the formal parameter in the function/procedure body are felt in the calling environment. The call has side effects.

Call by name. The effect is as the unevaluated argument is textually "substituted" for the formal parameter in the body of the function. Each time the value of the formal parameter is needed the argument is evaluated. The implementation avoids substitution by using the address of an anonymous subroutine, called *thunk*, corresponding to the unevaluated argument. Each time the value of the argument is needed the thunk is executed. The thunk is like a closure which packs the argument.

However, although closures can be used to simulate the call by name, a subtle difference exists. In the call by name the argument is evaluated in the referencing context of the formal parameter it replaces. Therefore, the computational environment of the argument is the computational environment of the formal parameter replaced. If the argument is implemented as a closure, the referencing context of the closure is that corresponding to the closure construction.

To clarify the call by name and the discussion above, consider the Jensen mechanism, used to compute the sum $\sum_{x=1}^{\lim it} term$. The function **sum** is written in an ad-hoc C, considering that the parameter **term** is transferred by name.

```
int sum (by_name int term, int limit) {
   int x,s=0;
   for(x=1; x <=limit; x++) s += term;
   return s;
}
```

The call **sum(x*x+2*x+3,10)** is equivalent to executing the function body shown below, where the formal parameter **term** is *macro expanded* by textually substituting it by the unevaluated argument **x*x + 2*x+3**.

```
int x,s=0;
for(x=1; x <=10; x++) s += x*x + 2*x+3;
return s;
```

Notice that the variable **x** from the argument is - by force of textual substitution - the same **x** from the body of the function. Remember that the argument is evaluated in the referencing context of the formal parameter it replaces. The function call **sum(x*x+2*x+3,10)** computes

$$\sum_{x=1}^{10} (x^2 + 2x + 3)$$

The function **sum** can be simulated in Scheme by a function that gets **0**-ary closures as arguments. The closure **stop** tests if **x > limit**; the closure **next** increments **x** by **1** and as then returns itself as the result; the closure **term** computes x^2+2x+3. Notice the essential difference from the call by name. In the simulated function the variable **x** does not belong to **sum**. It is local to the referencing context of the functions **stop**, **term** and **next**.

```
(define sum
  (lambda (stop term next)
    (if (stop) 0 (+ (term) (sum stop term (next))))))

(define sum_init
  (lambda (limit)
    (letrec ((x 1)
             (stop (lambda() (> x limit)))
             (next (lambda() (set! x (+ x 1)) next))
             (term (lambda() (+ (* x x) (* 2 x) 3))))
      (sum stop term next))))

(sum_init 10)
525
```

The call by name coincides with the “call by closure” only when:

- The scope of the variables is static and the applied function does not use variables whose names coincide with the names of the free variables from the argument.
- The scope of the variables is dynamic.

The call by name has historical roots in Algol-60, where it was the default parameter transfer mode, and helps achieving subtle effects by apparently simple computations. However, in practical situations call by name is inefficient, as proved by the problem below.

Let’s compute **(power x n)** = x^n using the formula: $x^{2n} = (x^2)^n$ and $x^{2n+1} = x\,(x^2)^n$. The function must finish in $\Theta(\log_2 n)$ steps. Assume that the parameter **x** is transferred by name, here perfectly simulated by transferring a closure.

```
(define power
  (lambda(x n)
    (cond ((= n 0)  1)
          ((= n 1) (x))
          ((odd? n) (* (x) (power (lambda() (* (x) (x))) (quotient n 2))))
          (else (power (lambda() (* (x) (x))) (quotient n 2))))))

(define x 2)

(power (lambda() (display "eval ") (+ x 1)) 8)
eval eval eval eval eval eval eval eval
6561
```

For the call **(power (lambda()(+ x 1)) 8)** there must be exactly **3** steps of computation. Unfortunately, the call by name spoils the speed of the algorithm. The expression **(+ x 1)** is evaluated **8** times. The hidden complexity of the function is **Θ(n)** instead of the apparent **Θ(log$_2$ n)**.

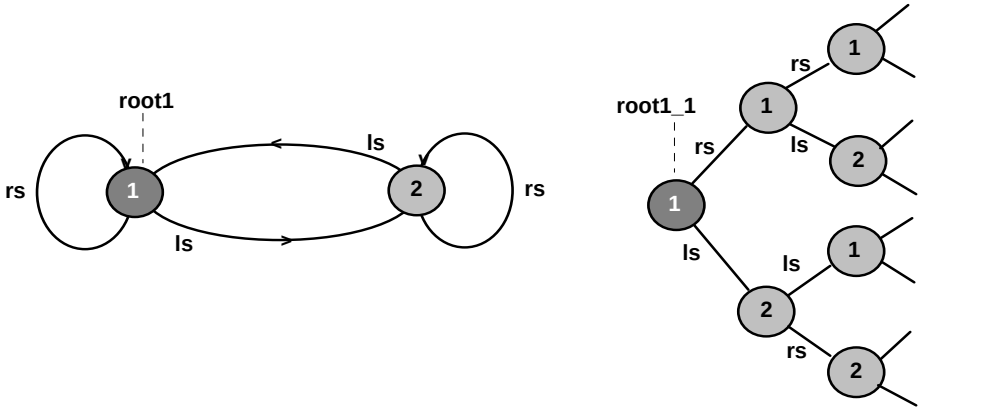
Call by need. A promise is built from the unevaluated argument and the address of the promise is transferred. Each time the value of the argument is needed, the promise is evaluated. The argument is effectively evaluated once only, the first time its value is needed.

Consider rewriting the function power such as the parameter **x** is transferred by need. For the call **(power (delay (+ x 1)) 8)** the expression **(+ x 1)** is evaluated once only.

```
(define powerr
  (lambda(x n)
    (cond ((= n 0)  1)
          ((= n 1) (force x))
          ((odd? n)
           (* (force x)
              (powerr (delay (* (force x) (force x))) (quotient n 2))))
          (else (powerr (delay (* (force x) (force x))) (quotient n 2))))))

(define x 2)
(powerr (delay (begin (display "eval ") (+ x 1))) 8)
eval
6561
```

The call by need is a safe and efficient way of transferring parameters when writing non-strict functions. It also is essential for representing objects, eventually infinite, in a functional way, yet simulating the effect of physical pointers.



a) Lazy representation b) Pure functional representation

Figure 2. A finite graph and two possible representations.

Consider the finite graph in figure 2a. Using lazy evaluation, the graph can be represented exactly as in figure 2a.

```
(define root1
  (letrec ((a (delay (list 1 b a)))
           (b (delay (list 2 a b))))
    (force a)))

(define ls (lambda (node) (force (cadr node))))
(define rs (lambda (node) (force (caddr node))))
```

We have the physical equalities:

```
(and (eq? (ls (ls root1))  root1)  (eq? (rs root1) root1)
     (eq? (ls root1) (rs (ls root1))))
#t
```

On the other hand we can represent the graph in a pure functional style, as below.

```
(define root1
  (letrec ((a (lambda() (list 1 b a)))
           (b (lambda() (list 2 a b))))
    (a)))

(define ls1 (lambda (node) ((cadr node))))
(define rs1 (lambda (node) ((caddr node))))

(and (eq? (ls (ls root1))  root1)
     (eq? (rs root1) root1)
     (eq? (ls root1) (rs (ls root1))))
#f
```

What is the effect of this representation is illustrated in figure 2b. Each time we are advancing on an **ls** or an **rs** link we are creating brand new nodes. It is like unfolding forever the finite graph.

An application: lazy problem solving by searching

Consider a problem which can accept many solutions, eventually an infinity of solutions. The goal is to write a solution generator. A generator is a function which gets an integer **n** and returns the list of the next **n** solutions of the problem. In fact the generator is a closure which closes the stream of the problem solutions.

For sake of brevity we consider that the problem is solvable by breadth-first search, although there is no restriction on the search strategy that is used. The conventional code for the breadth-first search is:

```
(define breadth_search
  (lambda (ini_state  expand  is_solution)
    (letrec ((search
              (lambda (states)
                (if (null? states) '()
                    (let ((state (car states))
                          (states (cdr states)))
                      (if (is_solution state) state
                          (search (append states (expand state)))))))))
      (search (list ini_state)))))
```

The function **breadth_search** has the following parameters: **ini_state** – the initial state of the problem; **expand** – a function that gets a state and returns the list of the successor states; **is_solution** - a predicate that decides if a state is a solution. The function returns the first solution found. What if you would like to see some more solutions, and you do not know in advance how many?

The trick is to build the stream of all the problem solutions. A term of the stream is

```
(solution . the_continuation_of_the_search)
```

where **the_continuation_of_the_search** is a function that can build the next stream term, i.e. the pair of the next solution and the search continuation mechanism. As a matter of fact

the_continuation_of_the_search is a promise that closes the entire computation status of the current search process.

```
(define lazy_breadth_search
  (lambda (ini_state expand is_solution)
    (letrec ((search
              (lambda (states)
                (if (null? states) '()
                    (let ((state (car states)) (states (cdr states)))
                      (if (is_solution state)
                          (cons state
                                (delay (search (append states
                                                       (expand state)))))
                          (search (append states (expand state))))))))))
      (search (list ini_state)))))
```

The expression

```
(cons state
      (delay (search (append states (expand state)))))
```

from the function **lazy_breadth_search** is doing the job. The promise is **(delay (search (append states (expand state))))** and closes the continuation of the search process.

Building the stream of palindromes longer than a minimum given length, and made with elements from a list of symbols, is now easy. First we consider that a palindrome is represented as a list, e.g. **(a b b a)** is a palindrome of length four. Second we build the stream of palindromes calling the **lazy_breadth_search** with the appropriate parameters.

```
(define palindromes
  (lambda (symbols min_length)
    (lazy_breadth_search
                    '()
                    (lambda(state) (map (lambda(s) (cons s state)) symbols)
                    (lambda(state) (and (equal? state (reverse state))
                                        (>= (length state) min_length))))))
```

The **is_solution** predicate corresponds to the anonymous function

```
(lambda(solution)
       (and (equal? state (reverse solution))
            (>= (length solution) min_length))))))
```

which tests if a potential solution (a list) is equal to its reversed format and if it is of the required length.

The **expand** function corresponds to

```
(lambda(state) (map (lambda(s) (cons s state)) symbols)
```

and uses the Scheme predefined high-order function **map**. What **map** does is to take a function **f** and a list **(e1 e2 . . . en)**, and to compute the list **((f e1) (f e1) . . .(f en))**

The initial state of the search process is obvious: the empty palindrome, i.e. the empty list **'()**.

The generator of palindromes is built using a function called **gen_gen**. The call **(gen_gen stream)** returns a generator of the **stream**. Recall that the generator is a function that gets a number **N** and returns the list of the next **N** terms from the **stream**. The generator uses two auxiliary functions that work on streams: **take** and **drop**.

```
; (take N stream) returns the list of the
; first N elements of the stream.
; ____________________________________________________
(define take
  (lambda(n stream)
    (cond ((= n 0) '())
          ((null? stream) '())
          (else (cons (car stream) (take (- n 1) (force (cdr stream))))))))


; (drop N stream) returns the stream resulted
; from cutting off the first N elements of the stream.
; ____________________________________________________
(define drop
  (lambda(n stream)
    (cond ((= n 0) stream)
          ((null? stream) '())
          (else (drop (- n 1) (force (cdr stream)))))))


(define gen_gen
  (lambda (stream)
    (lambda (n)
      (let ((sol (take n stream)))
        (set! stream (drop n stream))
        sol))))



; A generator of palindromes
; ____________________________

(define gen (gen_gen (palindromes '(A B C) 4)))

(gen 4)
((a a a a) (a b b a) (a c c a) (b a a b))

(gen 3)
((b b b b) (b c c b) (c a a c))

(gen 7)
((c b b c)
 (c c c c)
 (a a a a a)
 (a a b a a)
 (a a c a a)
 (a b a b a)
 (a b b b a))

(gen 9)
((a b c b a)
 (a c a c a)
 (a c b c a)
 (a c c c a)
 (b a a a b)
 (b a b a b)
 (b a c a b)
 (b b a b b)
 (b b b b b))
```