# Variation on The Little Schemer Interpreter - Lazy Evaluation

Students: Aashna Shah, Gurpreet Singh, James Chou

**1. Introduction**

The Little Schemer **(TLS)** is an applicative-order language. It evaluates all procedure arguments whether or not their values are needed before entering the body of the procedure. If the argument is evaluated before the body of the procedure is entered we say that the procedure is *strict* in that argument. In this paper, we explore a variation of TLSinterpreter in which the evaluations of procedure arguments are delayed until the values are needed in the procedure. If the body of the procedure is entered before an argument has been evaluated we say that the procedure is *non-strict* in that argument. This is known as *lazy* or *Normal order* evaluation and it allows procedures to be defined that could not be defined using the normal order evaluation. This strategy has two crucial properties. First, the evaluation of argument is delayed and stored as *thunks*, until its result is needed. Second, the first time a delayed computation is executed, it is changed to an *evaluated thunk* and its resulting value is cached so that the next time it is needed, it can be looked up rather than recomputed. This caching is known as *memoization.*

**2. Advantages of Lazy Evaluation**

Lazy evaluation is an employed by many purely functional programming languages such as Haskell and Miranda.

To see some advantages of Lazy evaluation, consider the following examples below:

|  |
| --- |
| (define (try a b)  (**if** (= a 0) 1 b)) |

In applicable order TLS, if we try to evaluate (try 0 (/ 1 0)) it will generate an error because the arguments will be evaluated and the interpreter will realize that the argument *b* is invalid. But in Lazy TLS, this will not generate any error because the interpreter will not see b in the procedure body, because *a* is 0 and the conditional if will simply returns 1 and the interpreter never saw b in the body, therefor it was never evaluated. Consider another example below where the arguments do not generate errors:

|  |
| --- |
| (define (f a b)  (+ a 1)) |

If we try to evaluate (f 5 (1000000000000^9999999999)) using strict TLS, both arguments will be evaluated before entering the body of f, which we can see is a lot of work, and when the program enters the body of f, it ignores its second argument, so all that work to compute (1000000000000^9999999999) was wasted.

**3. Disadvantage of Lazy Evaluation**

Lazy evaluation is a trade-off. On one hand, it helps with making our code more modular. On the other hand, it becomes infeasible to completely understand how evaluation proceeds in a particular program. It is difficult to debug the program. Consider the factorial program:

(define (fact k)

(if (= k 0)

1

(\* k (fact (- k 1)))))

This is typical tail recursion problem and tail recursion does not need stack space, and no allocation is required. But when we use a lazy language implementation, the picture is dramatically different. For example, if we evaluate (fact 5), we will eventually create five nested thunks representing the computation. When it returns, the resulting chain of thunks will finally be forced and the multiply operations will actually be performed.

**4. TLS Interpreter with Lazy Evaluation**

In this section, we will explain our implementation of TLS-Lazy adapted from Abelson and Sussman’s *Structure and Interpretation of Computer Programs*. The primitive procedures will still be strict. The idea is that, when applying a procedure, the interpreter must determine which arguments are to be evaluated and which are to be delayed. The delayed arguments are not evaluated; instead they are converted into objects called *thunks*. The thunk contains the argument expression and the environment in which the procedure is being evaluated to produce the value of the argument when it is needed.

To evaluate the expression in a thunk, we need to force it. In general, a thunk is forced when its value is needed. It value is needed when it is passed to a primitive procedure that will use the value of the thunk or when it is the value of a predicate of a conditional, or when is the value of an operator that is about to be applied as a procedure. Our interpreter will memorize thunks, so when a thunk is forced for the first time, it stores the value that is computed.

The main difference between TLS and the TLS-Lazy is the handling of the procedure \**application* and *myapply*. The \**application* procedure will now call *myapply* with the operands without evaluating them. We will still evaluate the operator because *myapply* needs the actual procedure to be applied in order to dispatch on its type and apply it.

(define \*application

(lambda (e table)

(myapply (actual-value (operator e) table)

(operands e)

table)))

To get the actual-value, we use the following procedure:

(define (actual-value e table)

(force-it (meaning e table)))

If the meaning of the expression is a thunk then the thunk will be forced otherwise it will return the meaning. The new version of *myapply* can be seen below:

(define myapply

(lambda (procedure arguments table)

(cond

((primitive? procedure) (myapply-primitive

(second procedure)

(list-of-args-value arguments table)))

((non-primitive? procedure)

(eval-seq (proc-body procedure)

(extend-env (proc-params procedure)

(list-of-delayed-args arguments table)

(proc-env procedure)))))))

For primitive procedures we will call myapply-primitive with the procedure to be applied to the arguments and the arguments are evaluated using the list-of-args-value function.

(define (list-of-args-value arguments table)

(**if** (null? arguments)

'()

(cons (actual-value (car arguments) table)

(list-of-args-value (cdr arguments) table)

)))

The arguments can be thunks, in which case they will be forced and memorized, or the arguments may also be evaluated thunks or simple primitives or constants. For non-primitive procedures passed into *myapply*, the arguments are delayed using list-of-delayed-args, and they are converted to thunks using the delay-it and then the current environment is extended to include the environment of the procedure.

(define (list-of-delayed-args exp env)

(**if** (null? exp)

'()

(cons (delay-it (car exp) env)

(list-of-delayed-args (cdr exp) env))))

(define (delay-it exp env)

(list (quote thunk) exp env))

After the environment is extended then eval-seq will get the call meaning on the body of the procedure. In the body of the procedure, as arguments are used to evaluate expression, the interpreter will need to force the thunk of the argument to be evaluated and change it to an evaluated thunk. As we explained earlier, thunks are evaluated using force-it and the implementation of force-it is shown below:

(define (force-it obj)

(cond ((thunk? obj)

(let ((result (actual-value

(thunk-exp obj)

(thunk-env obj))))

(set-car! obj 'evaluated-thunk)

(set-car! (cdr obj) result)

(set-cdr! (cdr obj) '())

result))

((evaluated-thunk? obj)

(thunk-value obj))

(**else** obj)))

When we force objects to be evaluated, the first condition to check is if it is a thunk. If it is a thunk then the interpreter will extract the environment and the expression from the thunk using thunk-exp and thunk-env.

(define (thunk-exp thunk) (cadr thunk))

(define (thunk-env thunk) (caddr thunk))

Next actual-value is called using the expression and the environment because the thunk expression may have other arguments that have been stored as thunks. Once this returns, the result is memorized and the thunk will be changed to an evaluated-thunk. If the object passed to force-it is an evaluated-thunk then the result that was memorized will be return. Otherwise, the object is returned if it is neither a thunk? or evaluated-thunk?.

All these changes lead to changes in our *cond* function. Now, the test expression in the cond lines may be a thunk or an evaluated thunk. Therefore we need to check if it is a thunk or evaluated thunk. Next, we will get the actual-value of the thunk, which will then be forced to evaluate. See the code below of the next *cond* function.

|  |
| --- |
| (define evcon  (lambda (lines table)  (cond  ((**else**? (question-of (car lines)))  (cond ((atom? (meaning (answer-of (car lines)) table))  (meaning (answer-of (car lines)) table))  (**else** (**if** (or (eq? (car (meaning (answer-of (car lines)) table)) 'thunk)  (eq? (car (meaning (answer-of (car lines)) table)) 'evaluated-thunk))  (force-it (meaning (answer-of (car lines)) table))  (meaning (answer-of (car lines))table)))))  ((meaning (question-of (car lines)) table)  (cond ((atom? (meaning (answer-of (car lines)) table))  (meaning (answer-of (car lines)) table))  (**else** (**if** (or (eq? (car (meaning (answer-of (car lines)) table)) 'thunk)  (eq? (car (meaning (answer-of (car lines)) table)) 'evaluated-thunk))  (force-it (meaning (answer-of (car lines)) table))  (meaning (answer-of (car lines))table)))))  (**else** (evcon (cdr lines) table))))) |

Another change is in the *lookup-in-entry-help*. When the interpreter searches through the table, it may find that the value of the symbol being search for is a thunk or an evaluated thunk, so we will need to force that as well. The change in the code is shown below:

(define lookup-in-entry-help

(lambda (name names vals entry-f)

(cond

((null? names) (entry-f name))

((eq? (car names) name)

(cond ((eq? 'thunk (car (car vals))) (force-it (car vals)))

((eq? 'evaluated-thunk (car (car vals))) (force-it (car vals)))

(**else** (car vals))

))

(**else** (lookup-in-entry-help name

(cdr names)

(cdr vals)

entry-f)))))

**5. Streams in TLS-Lazy**

Laziness in functional languages makes it easy to support recursively defining infinite data structures, which is not available in eager or applicative order evaluation. Streams, conceptually infinite lists, are a classic example of infinite data structures. They expand only as far as their users require. Lazy evaluation is important when representing infinite objects because the part already computed from an infinite object need not be recomputed. To explain the concept of infinite data structures, consider the example of computing integers from 1 up to n. Theoretically if we have infinite hardware this result would never return, but using streams and lazy evaluation, we don’t necessary have to compute the entire result, we can save a state that the procedure is in and then restart from that state.

**6. Half Lazy V. Full Lazy**

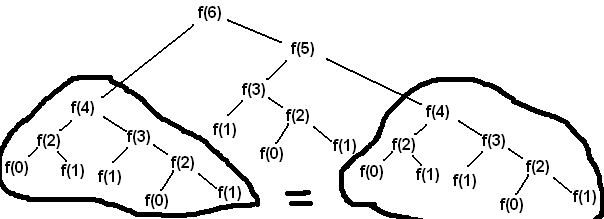
Half lazy interpreter delays only the car but the full lazy interpreter delays both *car* and *cdr* and allows us to create data structures such as infinite tree or any other structure built from pairs. If we recall how trees were implemented as lists earlier we can see how lazy *car* and *cdr* allow us to convert those to infinite trees. Consider the example of computing the Fibonacci sequence defined as:

fib(1) = 1

fib(2) = 1

fib(n+1) = fib(n) + fib(n-1)

Using a full lazy interpreter, we can delay both fib(n) and fib(n-1). It will only compute as much as we want and return. The speed of the program will also increase drastically because of the memoization paradigm.



Looking at the picture above, in a half lazy interpreter, we will have computed the right sub-tree and delayed the left sub-tree, but going down the left side of the tree, we see that parts of the right sub-tree appears in the left sub-tree, so we did several computation over again, which increases the complexity of the algorithm. Half lazy is still better than non-lazy interpreter because there are some sub-trees that repeat in the left sub tree.

**7. Conclusion**

Changing the evaluation rules in of an interpreter can produce an entire new language. It enables programmers to approach problems in a new way because changing the evaluation rules changes what programs mean. In this paper, we have described how to convert the applicative order TLS into the TLS-Lazy. We have shown the advantages and disadvantages of using lazy evaluation and how it introduces a concept of infinite lists. With lazy evaluation you, there is a lot of flexibility because rather than decide ahead of time how large your data needs to be, if you write it using a lazy infinite structure you simply don't need to worry. Lazy evaluation can be used to improve the algorithm performance by only calculating data values as and when they are needed - and potentially not at all. So you can potentially avoid a large amount of unnecessary computation.

# Resources

Abelson, Harold, Gerald J. Sussman, and Julie Sussman. "Structure and Interpretation of Computer Programs." Structure and Interpretation of Computer Programs. The MIT Press, 1996. Web. 11 May 2015.

Friedman, Daniel P., Matthias Felleisen, Duane Bibby, and Gerald Sussman. "The Little Schemer (4th Edition)." The Little Schemer. The MIT Press, 21 Dec. 1996. Web. 21 May 2015.

**APPENDIX**