CS 360: Programming Languages
Lecture 7: A Lazy Interpreter

Geoffrey Mainland

Drexel University

Section 1

Administrivia

Administrivia

- Lab 2 due **Friday, February 1 at 11:59pm**.
- Lab 2 tests updated for TravisCI compatibility. See the `lab2` repository at https://github.com/DrexelCS360/lab2.
- Homework 2 due **Friday, February 8 at 11:59pm**.
- We will take time Thursday to work on Lab 2/Homework 2 in class. Be prepared.
- Office hours **5:30–7:30pm** on Thursday. I will stay until 8pm if there are students in my office at 7:30.
- Reading for next week is from *Programming in Haskell*. I do not expect you to read 8 chapters; they are for reference! New book, would like feedback.

Section 2

Lazy evaluation

Lazy evaluation

- **Applicative-order** languages evaluate a procedure's arguments before calling the procedure. This is also called **call-by-value** evaluation.
- The interpreter we have seen—the original metacircular interpreter—implements **applicative-order** evaluation. Scheme itself is an applicative-order language.
- **Normal-order** languages delay evaluation of procedure arguments until they are needed. This is also called **lazy evaluation**.
- **Call-by-name** evaluation re-evaluates arguments when they are needed.
- **Call-by-need** evaluation evaluates arguments once and caches the result, avoiding unnecessary repeated computation.
- Question: How do purity and call-by-need evaluation relate? Would knowing a language is pure/impure complicate or simplify call-by-need evaluation?

Lazy evaluation: an example

```scheme
(define (try a b)
  (if (= a 0) 1 b))
```

- What is the result of evaluating `(try 0 (/ 1 0))`?
- What would it be if we were using a lazy version of Scheme?

Lazy evaluation: another example

```
(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value))

(unless (= b 0)
        (/ a b)
        (begin (display "exception: returning 0")
               0))
```

We can only write `unless` in a lazy language.

A lazy interpreter

- Basic idea: when performing application, the interpreter must decide which arguments are to be evaluated and which are to be delayed.
- Question: In what circumstances must an expression be evaluated? Hint: there are three (four if you count the driver loop).
- Delayed arguments are not evaluated, but instead captured as **thunks**.
- Question: What information does a thunk need to contain? Think about how we implemented the `delay` special form.
- The process of evaluating a thunk is called **forcing** the thunk.
- When should we force a thunk? When did delayed things get forced when we were working with streams?

Modifying application

Before:

```
((application? exp)
 (mcapply (mceval (operator exp) env)
          (list-of-values (operands exp) env)))
```

After:

```
((application? exp)
 (mcapply (actual-value (operator exp) env)
          (operands exp)
          env))
```

- For lazy evaluation, we call `mcapply` with the operand *expressions* rather than the arguments produced by evaluating them.
- Since we need the environment to create thunks, we pass it to `mcapply` as well.
- We still evaluate the operator because `mcapply` needs to dispatch on its "type" to apply it.

Getting the value of an expression

```
(define (actual-value exp env)
  (force-it (mceval exp env)))
```

Whenever we need the value of an expression, we call `actual-value`. The call to `force-it` will force any thunk.

Question: Why can't we just directly return the result of `mceval`?

A lazy mcapply

```
(define (mcapply procedure arguments env)
  (cond [(primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env))]  ; changed
        [(compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env) ; changed
           (procedure-environment procedure)))]
        [else
         (error
          "Unknown procedure type -- APPLY" procedure)]))
```

- We need to use `actual-value` instead of `mceval` to evaluate arguments to primitives.
- We need to delay arguments to compound procedures.
- We no longer have any use for `list-of-values`.

A lazy apply

```
(define (list-of-arg-values exps env)
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps) env)
            (list-of-arg-values (rest-operands exps)
                                env))))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps) env)
            (list-of-delayed-args (rest-operands exps)
                                  env))))
```

One more change we need to make…

```
(define (eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (mceval (if-consequent exp) env)
      (mceval (if-alternative exp) env)))
```

Representing thunks

A thunk must package together an expression and an environment.

```
(define (delay-it exp env)
  (mlist 'thunk exp env))

(define (thunk? obj)
  (tagged-mlist? obj 'thunk))

(define (thunk-exp thunk) (mcadr thunk))

(define (thunk-env thunk) (mcaddr thunk))
```

Forcing thunks

```scheme
(define (force-it obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj))

(define (actual-value exp env)
  (force-it (mceval exp env)))
```

But we want to memoize thunks...

```
(define (evaluated-thunk? obj)
  (tagged-mlist? obj 'evaluated-thunk))

(define (thunk-value evaluated-thunk) (mcadr evaluated-thunk))

(define (force-it obj)
  (cond [(thunk? obj)
         (let ((result (actual-value
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-mcar! obj 'evaluated-thunk)
           (set-mcar! (mcdr obj) result)  ; replace exp with its value
           (set-mcdr! (mcdr obj) '())     ; forget unneeded env
           result)]
        [(evaluated-thunk? obj)
         (thunk-value obj)]
        [else obj]))
```

Summary: Lazy Interpreter

- The lazy interpreter delays evaluating expression until they are needed.
- There are 3 (4) places where we actually need the value of an expression. What are they?
- Delayed expression are represented using thunks. What does a thunk contain, i.e., what do we need in order to evaluate an expression at some point in the future?
- Question: When are thunks created?

Section 3

Understanding the metacircular interpreter

Extending the interpreter

There are three ways to extend the interpreter. How do you make use of each?

1. Add a primitive (see Problem 1.2).
2. Add a top-level definition (see Problem 1.5).
3. Add a special form.

What is the difference between...

```
(top-mceval (+ 2 3))
(top-mceval '(+ 2 3))
```

```
(top-mceval +)
(top-mceval '+)
```

```
(top-mceval (define (id x) x))
(top-mceval '(define (id x) x))
```

Section 4

Review of `force`, `delay`, and streams

Implementing streams with `force` and `delay`

- To implement streams, we need a way to prevent the stream's tail from being evaluated.
- Idea: new syntax.

```
(delay <exp>)
(force <exp>)
```

- `delay` does not evaluate its argument, but returns a *delayed object*, which is a "promise" to evaluate <exp> at some point in the future.
- `force` takes a delayed object and evaluates it—it forces the delay to fulfill its promise.
- Question: Does `delay` need to be a special form?

Implementing streams

- Question: How can we implement stream-cons?
- Would this work?

```
(define (stream-cons x s) (cons x (delay s)))
```

- (stream-cons <a> <b>) $\equiv$ (cons <a> (delay <b>))
- The idea is to **rewrite** (stream-cons <a> <b>) into an expression we *already* know how to evaluate.
- What about stream-first and stream-rest?

```
(define (stream-first s) (car s))
(define (stream-rest s) (force (cdr s)))
```

Implementing `force` and `delay`

- We already decided that `delay` needs to be a special form. How can we represent `(delay <exp>)` in terms of Scheme constructs we've already seen?
- `(delay <exp>)` is syntactic sugar for `(lambda () <exp>)`.
- Once again, the solution is to **rewrite** `(delay <exp>)` into an expression we already know how to evaluate.
- `force` can simply call the procedure produced by `delay` (it doesn't need to be a special form):

  ```
  (define (force delayed-object)
    (delayed-object))
  ```
- Do you see an efficiency issue with `delay`?
- How could we use state to solve this problem?

Implementing `delay` efficiently

```
(define (memo-proc proc)
  (let ((already-run? false)
        (result null))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

Now we can define `(delay <exp>)` as syntactic sugar for

```
(memo-proc (lambda () <exp>))
```

Homework 2: Implementing `force`, `delay`, and streams

- Do **not** attempt to implement `force` and `delay` with thunks. We just reviewed the most straightforward way to implement them!
- Think before you write!
  - What method will you use to implement each construct? Remember, there are 3 ways to extend the interpreter.
  - Where is code being executed? By Racket? By your interpreter?
- For all `e`, `(force (delay e))` $\equiv$ `e`
- Question: Can you think of an obviously *incorrect* implementation of `force` and `delay` that satisfies this equation.