CSCI 2041: Lazy Evaluation

Chris Kauffman

Last Updated:
Wed Dec 5 12:32:32 CST 2018

Logistics

Reading

- Module Lazy on lazy evaluation
- Module Stream on streams

Lambdas/Closures

Briefly discuss these as they pertain Calculon

Goals

- Eager Evaluation
- Lazy Evaluation
- Streams

Lab13: Lazy/Streams

Covers basics of delayed computation

A5: Calculon

- Arithmetic language interpreter
- 2X credit for assignment
- 5 Required Problems 100pts
- 5 Option Problems 50pts
- Milestone due Wed 12/5
- Final submit Tue 12/11

Evaluation Strategies

Eager Evaluation

- Most languages employ **eager evaluation**
- Execute instructions as control reaches associated code
- Corresponds closely to actual machine execution

Lazy Evaluation

- An alternative is **lazy evaluation**
- Execute instructions only as expression results are needed (*call by need*)
- Higher-level idea with advantages and disadvantages

- In **pure computations**, evaluation strategy doesn't matter: will produce the same results
- With **side-effects**, when code is run matter, particular for I/O which may see different printing orders

Exercise: Side-Effects and Evaluation Strategy

Most common place to see differences between Eager/Lazy eval is when functions are called

- Eager eval: eval argument expressions, call functions with results
- Lazy eval: call function with un-evaluated expressions, eval as results are needed

Consider the following expression

```
let print_it expr =
  printf "Printing it\n";
  printf "%d\n" expr;
;;

print_it (begin
            printf "Evaluating\n";
            5;
          end);;
```

Predict results and output for both Eager and Lazy Eval strategies

Answers: Side-Effects and Evaluation Strategy

```
let print_it expr =
  printf "Printing it\n";
  printf "%d\n" expr;
;;

print_it (begin
            printf "Evaluating\n";
            5;
          end);;
```

Evaluation

```
> ocamlc eager_v_lazy.ml
> ./a.out
Eager Eval                      # ocaml's default
Evaluating
Printing it
5

Lazy Eval
Printing it
Evaluating
5
```

OCaml and explicit lazy Computations

- OCaml's default model is eager evaluation BUT...
- Can introduce lazy portions via the `lazy` keyword which produces a `'a lazy_t` type
- The `'a` is the type that will be produced on evaluation of the expression
- `Lazy.force expr` is used to evaluate an `lazy_t` expression to obtain its result

```
# lazy (printf "hello\n");;
- : unit lazy_t = <lazy>

# let result = lazy (printf "hello\n"; 5);;
val result : int lazy_t = <lazy>

# Lazy.force result;;
hello
- : int = 5
```

Code Example: eager_v_lazy.ml

```
open Printf;;

printf "Eager Eval\n";;

let print_it expr =
  printf "Printing it\n";
  printf "%d\n" expr;                          (* already evaluated *)
;;

print_it (begin                                (* pass a normal expression *)
            printf "Evaluating\n";             (* which will be eval'd  *)
            5;                                 (* before the call *)
          end);;

printf "Lazy Eval\n";;

let print_it_lazy expr =
  printf "Printing it\n";
  printf "%d\n" (Lazy.force expr);             (* force required to eval *)
;;

print_it_lazy (lazy (begin                     (* pass a lazy expression *)
                       printf "Evaluating\n";
                       5;
                     end));;
```

Exercise: Predict Output

- Consider the following REPL session using `lazy/force`
- Identify the type and value of each expression
- Indicate where output will result

```
# lazy (printf "hello\n"; 5);;                       (*1 *)

# Lazy.force (lazy (printf "hello\n"; 5));;   (*2 *)

# Lazy.force (lazy (printf "hello\n"; 5));;   (*3 *)

# let result = lazy (printf "hello\n"; 5);;   (*4 *)

# Lazy.force result;;           (*5 *)

# Lazy.force result;;           (*6  *)

# Lazy.force result;;           (*7  *)
```

Answers: Predict Output

```
# lazy (printf "hello\n"; 5);;                   (*1 lazy: no printing *)
- : int lazy_t = <lazy>

# Lazy.force (lazy (printf "hello\n"; 5));;  (*2 force: printing *)
hello
- : int = 5

# Lazy.force (lazy (printf "hello\n"; 5));;  (*3 force: printing *)
hello
- : int = 5

# let result = lazy (printf "hello\n"; 5);;  (*4 named lazy expr *)
val result : int lazy_t = <lazy>

# Lazy.force result;;              (*5 first evaluation: need result *)
hello                              (* side-effects produced during eval *)
- : int = 5                        (* answer saved for later use *)

# Lazy.force result;;              (*6 second evaluation *)
- : int = 5                        (* just return saved answer *)

# Lazy.force result;;              (*7 third eval *)
- : int = 5                        (* return saved answer *)
```

Exercise: Principle of Efficient Lazy Eval

- A `lazy` expression is not immediately evaluated
- When `force` is used, evaluate the expression **saving the result**
- If `force` is called again on the same expression, don't evaluate again, just return the saved result
- This opens up some efficiencies in lazy evaluation

Questions

1. Saving the results of evaluation for later should remind you of something we covered in a lab a while back...
2. To save the results of expression, what quality of must `lazy_t` data possess?

Answers: Principle of Efficient Lazy Eval

1. Saving the results of evaluation for later should remind you of something we covered in a lab a while back...
   Memoization used the same trick: evaluate once and save the results for later.
2. To save the results of expression, what quality of must `lazy_t` data possess?
   Using `force` must change `lazy_t` data so it must be mutable. A simple implementation would likely look like:

```
type 'a lazy_expr = {                  (* type for lazy expressions *)
    expr           : unit -> 'a;       (* expression to evaluate *)
    mutable result : 'a option;        (* saved results, None if uneval'd *)
};;
```

Haskell and Laziness

- OCaml allows some laziness via `lazy/force`, defaults to eager
- Haskell is the most well-known language with default lazy eval
- Enforces **pure computations only**: side-effects are prevented except in tightly controlled circumstances via **monads**
  - *A monad is just a monoid in the category of endofunctors, what's the problem?*[1]
  - DO NOT ask me about monads, monoids, or endofunctors
- Advantage: Enforcing pure computations with lazy evaluation potentially enables more efficiency if the programmer/compiler is sufficiently smart
- Disadvantage: I/O is difficult, iterative algorithms awkward, efficient mutable data structures are discouraged
- Haskell is interesting and fairly extreme for these reasons, likely attributing to its single implementation and lack of widespread use

[1] A Brief, Incomplete, and Mostly Wrong History of Programming Languages by James Iry

Lazy Relatives: Futures/Promises

- **Concurrent programming** performing instructions in an unpredictable order
- A standard model employs **threads** of instructions which are separately executed, may pause at any point, interleave instructions between threads, execute in simultaneously in parallel
- A **promise or future** is like a lazy expression that may execute in a different thread; execute code concurrently/later
- *CSCI 4061: Intro to Operating Systems* studies concurrency issues (in C)

Exercise Lazy Relatives: Streams / Generators

- **Streams or generators** abstract the idea of a data source
- Usually allow "give me the next thing" and or "anything left?"
- Internally, many details for efficiency specific to the source can be hidden including state, buffers, **delayed computations**
- Streams may not explicitly store all their data in memory, delaying storage until actually needed (like lazy expressions)
- Most file I/O is implemented as streams
  - Calls to `read chan` yield data and move ahead in the stream
  - Internally, chunks of input are usually cached/buffered but the whole file is **not** read into memory until needed

Questions

1. OCaml uses **channels** for input from files; how does reading from channels signal "no more input"?
2. Where else have we seen this idea before: a data source that provides only a way to get the "next" thing?
3. From lab, demonstrate a useful module for creation of streams; show different sources for the streams

Answers: Lazy Relatives: Streams

1. OCaml uses **channels** for input from files; how does reading from channels signal to "no more input"?
   An `End_of_file` exception is usually raised on reading from a channel that is out of input.
2. Where else have we seen this idea before: a data source only provides only a way to get the "next" thing?
   Aside from file input, saw it associated with Lexing Buffers which only provided a next-like function to produce a token.
3. From lab, demonstrate a useful module for creation of streams; show different sources for the streams

```
let crew_list = ["Mal"; "Zoe"; "Wash";] in
let crew_stream = Stream.of_list crew_list in   (* from list *)
let captainy      = Stream.next crew_stream in
let badass        = Stream.next crew_stream in
let leafonthewind = Stream.next crew_stream in
...
let always_one _ = 1 in
let one_stream = Stream.from always_one in      (* from func *)
let one  = Stream.next one_stream in
let uno  = Stream.next one_stream in
let hana = Stream.next one_stream in
...
```

Streams from Functions

- As seen, can build a stream from a function
- This allows the stream to be **generated** on the fly
- Stream could represent an extremely large or even infinite amount of data
- Clever function definition represents this in **constant memory space** rather than create an array/list which would take $O(N)$ memory

```
(* range.ml :create a stream of
   numbers with a function from 0
   to stop-1; O(1) memory usage *)
let range stop =
  let i = ref 0 in
  let advance _ =
    if !i < stop then
      let ret = !i in
      i := !i + 1;
      Some ret
    else
      None
  in
  Stream.from advance
;;

let _ =
  printf "0 to 9\n";
  let r10 = range 10 in
  while Stream.peek r10 <> None do
    printf "%d\n" (Stream.next r10);
  done;
  ...
```

Streams/Generators in other Languages

- Python's **generators** are streams, appear everywhere associated with `for` syntax
- `range()` is a generator for a stream of numbers

```
print("0 to 9")
for i in range(1,10):     # standard for loop with a range
  print(i)

r100 = range(1,100)       # range's are objects
print(r100)               # which prints as
                          # "range(1, 100)"
for i in r100:            # and can be iterated over
  print(i)
```

- Java's Iterator interface is similar providing an `iter.next()` function to move ahead and produce data
- `for(x :  thing)` syntax creates and advances an iterator
- Most often associated with iterating over a data structure
- Clojure's lazy sequences are . . . well, that one is obvious

Summary

- All programming languages choose an **evaluation strategy** which dictates the order in which instructions are executed
- Eager eval is used by most PLs and feels fairly natural but is not the only game in town
- **Lazy eval** can lead to some interesting possibilities and potential efficiencies if implemented "smartly"
- OCaml uses eager eval but can introduce lazy expressions via `lazy` with the `Lazy` module providing other ops like `force`
- Generally, delaying computation until needed is useful as demonstrated in **streams** which appear in many programming languages under different names (generators, iterators, etc.)