S4.1: Expression Evaluation: Eager and Lazy Evaluation

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

Eager and Lazy Evaluation

- These slides introduce *call-by-name* semantics and *lazy evaluation* as an alternative to the more common *call-by-value* semantics.
- We will see how lazy evaluation allows for more direct expression of solutions to some problems.
- They also show how these techniques can be used in languages such as OCaml that use call-by-value semantics.

Function definitions as equations or rules

- Consider a function definition:
  `f x = g (h x x)`
- We may read this as follows:

  "For any expression `x`, the expression `f x` can be replaced by (rewritten as) `g (h x x)` in order to evaluate it."
  - This quote and much of this material in this section comes from Chris Reade's book Elements of Functional Programming.

Let's consider the following definitions

```
g x = 5
h x y = h y x
f x = g (h x x)
```

Treating these as (directed) rules, we could evaluate this as follows:

```
f 3 = g (h 3 3), by def of f w/ x = 3
    = 5,         by def of g w/ x = h 3 3
```

Alternatively, we could evaluate it this way:

```
f 3 = g (h 3 3), by def of f w/ x = 3
    = g (h 3 3), by def of h w/ x = y = 3
    = g (h 3 3), by def of h w/ x = y = 3
    = g (h 3 3), by def of h w/ x = y = 3
    ...
```

Here we see non-terminating behavior.

Different evaluation strategies

- The non-terminating evaluation attempted to evaluate function arguments before calling the function.
- This is the traditional *call-by-value* semantics.
- In call-by-value, function arguments are evaluated down to values and the values are then passed to the function.
  - OCaml uses call-by-value semantics.
  - In functional languages, this is also called *applicative order evaluation*.
  - It is also called *eager evaluation*.

Different evaluation strategies

- The terminating evaluation *delayed* evaluating function arguments until they were needed (any they might not be) in the evaluation of the function body.
- This is called *call-by-name* semantics.
  - Haskell uses a form of call-by-name semantics.
  - In functional languages, this is also called *normal order evaluation*.
  - An optimization of this, that we will see later, is called *lazy evaluation*.

Additional valid behavior

We have seen two examples:

```
f 3                        a (3 + 4) (3 / 0)
```
where
```
g x = 5                    a x y = x + 3
h x y = h y x
f x = g (h x x)
```

Each produces a value under call-by-name semantics and does not under call-by-value semantics.

If an expression `e` evaluates to a value `v` under call by value semantics, then `e` also evaluates to `v` under call by name semantics.

But some expressions that evaluate to values under call-by-name semantics *do not* evaluate to values under call-by-value semantics - instead they fail to terminate or terminate abnormally.

So which is correct?

- call-by-value is correct ⇒ function definitions are not equations over expressions, but only over defined fully-evaluated values.
- call-by-name is correct ⇒ functions definitions are equations over any values, well-defined or not.

Lazy evaluation

Naive implementations of call-by-name semantics can be inefficient.

Consider this definition:
```
double x = x + x
```
and the evaluation of
```
double (fact 10)
```
which in one step, evaluates to
```
(fact 10) + (fact 10)
```

This would be an inefficient way to do this.

Lazy evaluation

In call-by-name evaluation above we saw evaluation as rewriting over terms (syntax trees).

Variables are replaced by expressions.

In lazy evaluation, the same variables are replaces by pointers to the expression. When it is evaluated for one use of the variable, then the other uses see the evaluated value when they look for it.

We can draw these as DAGs - directed acyclic graphs.

We can also use a `where` clause to do this textually.

See how this is done on the white board.

Lazy evaluation

Lazy evaluation and normal order evaluation (call-by-name) have the same behavior.

For any expression `e`,

- They both terminate with the same value `v` or
- They both fail to terminate.

Thus we see lazy evaluation as an optimization of normal order evaluation.

Evaluation by rewriting

- We can see call-by-value semantics as rewriting the expression from the “bottom up” or “inside out.”
  - roughly speaking, sub expressions lower in the tree are evaluated before those higher up.
- But call-by-name and lazy evaluation can be seen as rewriting the expression from the “top down” or “outside-in.”
- In both cases we typically go left-to-right for sub expressions at the same level.

Consider our previous examples:

```
g x = 5
h x y = h y x
f x = g (h x x)
```

The underlined components are the ones that are evaluated.

$$\begin{array}{ll} & \underline{\texttt{f 3}} \\ = & \underline{\texttt{g (h 3 3)}} \\ = & \texttt{5} \end{array} \qquad \begin{array}{ll} & \underline{\texttt{f 3}} \\ = & \texttt{g } \underline{\texttt{(h 3 3)}} \\ = & \texttt{g } \underline{\texttt{(h 3 3)}} \\ = & \texttt{g } \underline{\texttt{(h 3 3)}} \\ \ldots & \end{array}$$

Minimal evaluation

As the name suggests, in lazy evaluation we evaluate expressions as little as possible.

The following exercise will illustrate this point.

(See handout `expr_eval.pdf` on GitHub in `Resources` directory.)

Strictness

We say that a function is *strict* if passing it an undefined argument causes an undefined result to be produced.

Otherwise the function is *non-strict*.

In eager evaluation, we treat all functions as if they were strict.

We can generalize to strictness in a particular argument.

In the exercise above, `take` was strict in its first argument but not its second and `makefrom` was strict in both arguments.

So, in eager evaluation we only pass values to functions.

We first evaluate expressions down to values, then pass those values.

Whereas in lazy evaluation of call-by-name evaluation (non-strict evaluation) we can pass in expressions that may or may not later evaluate down to values.

Values

What precisely are values?

- Values of primitive types: integers, strings, characters
  `1`, `'A'`
- Lists of values
  `1 :: 2 :: 4 :: []`, or in short hand notation
  `[ ['h'; 'i']; ['t'; 'h'; 'e'; 'r'; 'e'] ]`
- Other value constructors (from disjoint union types) with values as arguments
- lambda expressions
  λ `x` $\rightarrow$ `x + (1 + 3)`
  (or `fun x -> x + (1 + 3)` in OCaml's syntax)
  We do not evaluate terms under a lambda.

Values are expressions that cannot be evaluated any further. 21

Termination

We've that some functions terminate under non-strict evaluation, but don't under strict evaluation.

We'll define $\perp$ to indicate a non terminating computation for any type.

- `let` $\perp_{int}$ `= let rec f x = f x + 1 in f 0`
- `let` $\perp_{string}$ `= let rec f x = f x ^ "X" in f "A"`
- `let` $\perp_{int\ list}$ `= let rec f x = 1 :: f x in f []`
- ...

We'll drop the subscript on $\perp$ since it can be inferred.

22

Termination properties

- Does $\perp ::$ `et` $= \perp$?
- Does `eh` $:: \perp = \perp$?
- Does $\perp :: \perp = \perp$?

23

Termination properties

No - they are not equal. Consider this function:

```
let is_empty [] = true
let is_empty (x::xs) = false
```

Applied to the above values what do we get?

- `is_empty` $(\perp ::$ `et`$) =$ `false`
- `is_empty` (`eh` $:: \perp) =$ `false`
- `is_empty` $(\perp :: \perp) =$ `false`
- `is_empty` $\perp = \perp$

Termination properties

Ok, what about these?

- Does $(\perp,$ `e`$) = \perp$? , Does (`e`, $\perp) = \perp$? ,
  Does $(\perp, \perp) = \perp$?

No. Consider `let foo (x,y) = 99` in
`foo` $(\perp,$ `e`$) =$ `99`
But we can view `foo` $\perp$ as returning `99`.
This pattern is *irrefutable*. We don't need to inspect the value to know that the patter will match. We can delay binding `x` and `y` until they are used in some way.
The list patterns were *refutable* - they may not match - and thus we had to evaluate the list to some extent to see if it matched.

Induction in lazy languages

Proofs by induction can still be done on infinite and undefined values.

For some property $P(v)$ we considered

- a base case, where v is some base of "zero" value and
- an inductive case.

Now, we must also consider undefined values: $\perp$.

So our proofs require an additional case: $P(\perp)$.

We are primarily interested in laziness as a programming technique and thus won't do any inductive proofs over lazy data structures.

Uses for lazy evaluation.

Short-circuit evaluation

All languages already have some constructs that use a form of lazy evaluation called *short circuit evaluation*.

Consider the `&&` operation.

- `False && e` evaluates to `False` without evaluating `e`
- `True || e` evaluates to `True` without evaluating `e`.

In a lazy language we can write functions with the same behavior:

```
let and_f b1 b2 = if b1 then b2 else false
```

We *cannot* write such a function in a eager language.

Short-circuit evaluation

The `if-then-else` construct in OCaml also uses short-circuit evaluation. It is a lazy operation.

The same is true for the `? :` operator in C.

Of the three arguments, in which ones if `if-then-else` strict?

Thus, we can write a `cond` function that behaves just like `if-then-else`

Ease of expression

Some problem solutions are easier to express in lazy languages.

Consider comparing the leaves of trees in
`compare_bintrees.ml`

Specifically, the two functions for testing equality of trees.

```
let rec equal_tree_v1 t1 t2 =
   equal_list (flatten t1) (flatten t2)

let rec equal_tree_v2 t1 t2 =
  let rec comparestacks f1 f2 = match f1, f2 with
  ...
```

Ease of expression

Under lazy evaluation, `equal_tree_v1` performs as efficiently as the convoluted `equal_tree_v2` does.

In an eager language, for this performance we have to intertwine the logic of checking for equality and extracting the leaves from the tree, as seen in `equal_tree_v2`.

This is an example of how laziness lets us pull apart these separate concerns.

We can evaluate these by hand to see this.

Laziness and folds

```
let rec foldr f l v = match l with
  | [] -> v
  | x::xs -> f x (foldr f xs v)

let rec foldl f v l = match l with
  | [] -> v
  | x::xs ->  foldl f (f v x) xs

let and_f b1 b2 = if b1 then b2 else false
let and_l l = foldl and_f true l
let and_r l = foldr and_f l true
```

Does lazy evaluation help either of these?

```
let rec foldr f l v = match l with
  | [] -> v
  | x::xs -> f x (foldr f xs v)

let rec foldl f v l = match l with
  | [] -> v
  | x::xs ->  foldl f (f v x) xs
```

When `f` is not strict in its second argument, then `foldr` need not process the entire list.

This is not the case for `foldl`.

A comment on naming folds

```
let rec foldl f v l =
  match l with
  | [] -> v
  | x::xs ->  foldl f (f v x) xs
```

`foldl` is sometimes called `accumulate`

For some, it is more intuitive to consider processing elements from the front of the list.

The argument `v` is the accumulator that is eventually returned.

A comment on naming folds

```
let rec foldr f l v =
  match l with
  | [] -> v
  | x::xs -> f x (foldr f xs v)
```

`foldr` is sometimes called `reduce` because the evaluation essentially replaces

- `::` with `f` and
- `[]` with `v`

`foldr (+) [1;2;3;4;] 0` is
`foldr (+) (1::(2::(3::(4::[])))) 0`
which can be seen as
`1 + (2 + (3 + (4 + 0)))`.

Infinite data structures

Consider wanting the first 10 squares of positive integers beginning at 3. In an eager language we might write:

```
let rec some_squares_from n v =
  if n = 0
  then [ ]
  else v * v :: some_squares_from (n-1) (v+1)
let answer = some_squares_from 10 3
```

In a lazy language, we might write

```
let squares_from v = v*v :: squares_from (v+1)
let answer = take 10 (squares_from 3)
```

Streams in OCaml

Streams as a form of lazy lists in which the head of the list has been evaluated (it isn't lazy) but the tail is evaluated lazily.

We accomplish this by placing the tail inside a lambda expression.

This then isn't evaluated until we need it.

```
type 'a stream = Cons of 'a * (unit -> 'a stream)
```

See `streams.ml` in the code examples repository for examples and descriptions of uses of this type.

Relation to coroutines and generators

- A *coroutine* is a procedure that runs concurrently with other coroutines.
- They can suspend execution, handing control over to another coroutine.
- This allows two coroutines to interleave their execution, communicating back and forth between them.

Relation to coroutines and generators

- For example, a client-server architecture in which
  - the client sends requests to a server,
  - which then sends back a response
- Python's `yield` statement and *generators* are examples of a restricted form of coroutines.
- We will see how both can be realized using lazy streams.

Client-Server architecture

$$\begin{tikzcd} \boxed{\text{Client}} \arrow[r, shift left, "\rightarrow\rightarrow\ \texttt{request}\ \rightarrow\rightarrow"] & \boxed{\text{Server}} \arrow[l, shift left, "\leftarrow\leftarrow\ \texttt{response}\ \leftarrow\leftarrow"] \end{tikzcd}$$

- A sequence of requests is generated and sent from the client to the server.
- These requests are effected by previously received responses.
- For example, a client function `next_request` takes the last response from the server and generates the next request..
- The server generates a sequence of responses based on the requests that it gets.
- For example, a server function `next_response` takes the last request from the client and generates the next response to send to it.

Client server architecture using lazy streams

We can define a `client` and `server` as intertwined consumers and producers of streams of data.

A `client` consumes a stream of responses and produces a stream of requests.

A `server` consumes a stream of requests and produces a stream of responses.

Since these streams are lazily evaluated we need not create, for example, the entire stream of requests before the stream of responses can be produced.

The generation of them can be interleaved. See implementation in `client_server.ml`.

Memory usage

- How much memory to these streams take?
  - They represent infinite lists, so perhaps a lot of memory is consumed.
- But our OCaml implementation is quite memory efficient.
  - After a response (or request) is consumed it is deallocated by the garbage collector.
  - The rest of the stream is essentially just a function waiting to generate more data.
    So it takes very little memory as well.

Generators in Python

A generator is a form of coroutine that responds to calls by returning values to the calling coroutine.

It is limited in that it can only yield control back to the calling coroutine, not to a different one.

Consider the use of Python's `yield` statement.

See example in `generators.py` in the code examples repository.

Simulating laziness in strict languages

Our lazy streams were an attempt to get some of the benefits of lazy evaluation of lists in a call-by-value/eager/strict language.

But how do we do this more generally?

Study the comments and examples in `lazy.ml` in the `SamplePrograms` directory of the public repository.

Simulating laziness in strict languages

Our lazy streams were an attempt to get some of the benefits of lazy evaluation of lists in a call-by-value/eager/strict language. But how do we do this more generally?

Consider defining a `lazy` type constructor so that

- `int lazy` is a type for a lazily evaluated integer.
- `mk_lazy :: (unit -> 'a) -> 'a lazy` takes an expression and delays its evaluation
- `force :: 'a lazy -> 'a` causes the expression to be evaluated.

Can we do this in OCaml using what we've learned so far of the language?

A possible implementation:

```
type 'a lazy = Unevaluated of (unit -> 'a)
let mk_lazy e = Unevaluated e
let force e = match e with
  | Unevaluated f -> f ()
```

But this doesn't save the evaluated value for use later.

```
let l1 = mk_lazy
     ( fun () ->
         (print_endline "A lazy computation" ;
          sqrt 16.0) )
let l2 () = force l1 +. force l1
```

We see this in

```
utop # l2 () ;;
A lazy computation
A lazy computation
- : float = 8.
```

We need side effects.

We need the first call to force to change the state of the lazy value from an unevaluated value to an evaluated value.

```
type 'a lazy = Unevaluated of (unit -> 'a)
             | Evaluated of 'a
```

And thus we need mutable references.

```
let l1 = ref (mk_lazy
     ( fun () ->
         (print_endline "A lazy computation" ;
          sqrt 16.0) ) )
```

We need a way to *change* the reference to a lazy value from an unevaluated expression to a value so that the second access of that reference finds the value, not an unevaluated expression.

For this, we'll need effects. See S5.1 material.