CSci 2041
Advanced Programming Principles
L13: Expression Evaluation: Eager and Lazy Evaluation

Eric Van Wyk

Fall 2014

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.
- We will see how laziness relates to the "yield" command in Python and other languages.

Function definitions as 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 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. optimization of call by name

Exercise L13, #1.

Above we saw that evaluation can be seen as a sequence of rewrites of the expression. Using this approach, evaluate

```
a (3 + 4) (3 / 0)
```

where

```
a x y = x + 3
```

using

1. call by name semantics, and then
2. call by value semantics

Additional valid behavior

We have seen two examples

```
f 3
```

where

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

```
a (3 + 4) (3 / 0)
```

where

```
a x y = x + 3
```

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 $\Rightarrow$ function definitions are not equations over expressions, but only over defined fully-evaluated values.
- Call by name is correct $\Rightarrow$ functions definitions are equations over any values, well-defined or not.

Lazy evaluation

Naive implementations of call by name semantics can be inefficient.

so we need to optimize

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

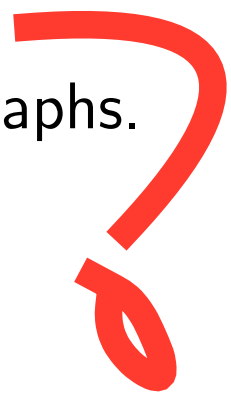
In call by name evaluation above we saw evaluation as rewriting over terms (syntax trees).

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.

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.

```
  f 3
= g (h 3 3)
= 5
```

```
  f 3
= g (h 3 3)
= g (h 3 3)
= g (h 3 3)
...
```

Minimal evaluation

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

The following exercise will illustrate this point.

Exercise L13, #2.

Using call by value evaluation and then lazy evaluation, write out the first 10 steps of the evaluation of the following:

```
take 2 (makefrom 4 5)
```

Use the following definitions (clearly not OCaml syntax):

```
take n [] = []
take 0 (x::xs) = []
take n (x::xs) = x::take (n-1) xs

makefrom 0 v = []
makefrom n v = v ::makefrom (n-1) (v+1)
```

(See the handout "Expression_Evaluation" on Moodle.)

Strictness

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

undefined cause undifined

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.

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 termination computation for any type.

- `let` $\perp_{int}$ `= let rec f x = f x in f 1`
- `let` $\perp_{char}$ `= let rec f x = f x in f 'a'`
- `let` $\perp_{'a\ list}$ `= let rec f x = f x in f []`
- ...

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

Termination properties

- Does $\bot :: e = \bot$?
- Does $e :: \bot = \bot$?
- Does $\bot :: \bot = \bot$?

No.

Consider this function:

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

Applied to the above values what do we get?

- `is_empty (⊥::e)` = `false`
- `is_empty (e::⊥)` = `false`
- `is_empty (⊥::⊥)` = `false`
- `is_empty ⊥` = ~~`true`~~ typo: should be nothing here

Termination properties

Ok, what about these?

- Does `(⊥, e)` $= \bot$?
- Does `(e, ⊥)` $= \bot$?
- Does `(⊥, ⊥)` $= \bot$?

No. Consider `let foo (x,y) = 99`
in
`foo (⊥, e)` $= 99$
But we can view `foo ⊥` as returning `99`. why???
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. !!! list is refutable; but tuple is not

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: $\bot$.

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

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

Short-circuit evaluation

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

Consider the `&&` operation.

- `False && _` evaluates to `False` without evaluating the second.
- `True || _` evaluates to `True` without evaluating the second.

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.

really?

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`

Exercise L13, #3.

Write a function named `cond` that has the same behavior as OCaml's if-then-else construct.

Ease of expression

Some problem solutions are easier to express in lazy languages.

Consider comparing the leaves of trees in `ordered_btree.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 compare_forests 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. how?

Laziness and folds

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

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 true l
```

Does lazy evaluation help either of these?

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

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

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

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

`foldr (+) 0 [1;2;3;4;]` is
`foldr (+) 0 (1::(2::(3::(4::[]))))`
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 squares_from n v =
  if n = 0
  then [ ]
  else v * v :: squares_from (n-1) (v+1)
let answer1 = squares 10 3
```

In a lazy language, we might write

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

Exercise L13, #4.

Evaluate the first doze steps or so of the following:

```
sum (take 3 (squares ones)
```

where

```
let rec sum lst = match lst with  | [] -> 0
  | x::xs -> x + sum xs

squares v = v*v :: squares (v+1)
```

Exercise L13, #5.

Define an infinite list containing all natural numbers, starting at 0.
Recall our definition of squares:

```
squares v = v*v :: squares (v+1)
```

Then evaluate `drop 3 nats` where

```
let rec drop n l = match l with
  | [] -> []
  | x::xs -> if n > 0 then drop (n-1) xs else l
```

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.