EDAF95/EDAN40: Functional Programming
On Laziness

Jacek Malec
Dept. of Computer Science, Lund University, Sweden

April 26th, 2023

Topic of today: evaluation, strictness, laziness

References:

- Hutton, 2nd ed., Chapter 15
- https://www.haskell.org/haskellwiki/Haskell/Lazy_evaluation
- https://www.haskell.org/haskellwiki/Lazy_vs._non-strict
- http://stackoverflow.com/questions/265392/why-is-lazy-evaluation-useful

What is evaluation?

Finding out a value:

$$(\int_0^y x dx)(2+3)$$

What is evaluation?

Finding out a value:

$$(\int_0^y x dx)(2+3)$$

Value of an expression:

$$f(a) = b$$

What is evaluation?

$$\int xdx = \frac{x^2}{2}$$

$$\int_0^y xdx = \frac{y^2}{2} - \frac{0^2}{2} = \frac{y^2}{2}$$

$$(\int_0^y xdx)(2+3) = (\frac{y^2}{2})(2+3) = \frac{(2+3)^2}{2} = \frac{5^2}{2} = \frac{25}{2} = 12,5$$

$$(\int_0^y xdx)(2+3) = (\int_0^y xdx)5 = (\int_0^5 xdx) = \frac{5^2}{2} = 12,5$$

Evaluation order

```
inc :: Int -> Int
inc n = n + 1
```

Evaluation order

```
inc :: Int -> Int
inc n = n + 1
```

```
                inc (2 * 3)

inc 6                           (2 * 3) + 1

6 + 1                              6 + 1

                        7
```

Evaluation order

Some imperative language

```
n = 0
y = n + (n = 1)
```

Evaluation order

Some imperative language

```
n = 0
y = n + (n = 1)
```

```
n + (n = 1)
```

```
0 + (n = 1)
```

```
0 + 1
```

```
1
```

```
n + 1
```

```
1 + 1
```

```
2
```

Reduction

Reducible expression: *redex*
Function application to an argument

```
mult :: (Int, Int) -> Int
mult (x, y) = x * y

mult (1 + 2, 2 + 3)
```

Three redexes!

Reduction

Reducible expression: *redex*
Function application to an argument

```
mult :: (Int, Int) -> Int
mult (x, y) = x * y

mult (1 + 2, 2 + 3)
```

Three redexes!

```
(1 + 2) * (2 + 3)
mult (3, 2 + 3)
mult (1 + 2, 5)
```

Choice -> leads to evaluation *strategies*

Evaluation strategies

1. innermost redex first (usually left-to-right) *call by value*

```
mult (1 + 2, 2 + 3)
mult (3, 2 + 3)
mult (3, 5)
3 * 5
15
```

2. outermost redex first (usually left-to-right) *call by name*

```
mult (1 + 2, 2 + 3)
(1 + 2) * (2 + 3)
3 * (2 + 3)
3 * 5
15
```

Lambda expressions

Curried `mult`:

```
mult :: Int -> Int -> Int
mult x = \y -> x * y
```

Innermost evaluation:

```
mult (1 + 2) (2 + 3)
mult 3 (2 + 3)
(\y -> 3 * y) (2 + 3)
(\y -> 3 * y) 5
3 * 5
15
```

In Haskell: **Do not reduce inside lambdas!**

Termination

```
inf :: Int
inf = 1 + inf
```

Now, irrespectively of evaluation strategy:

```
inf
1 + inf
1 + (1 + inf)
1 + (1 + (1 + inf))
...
```

Does not terminate!

Termination

But consider:

```
fst :: (a, b) -> a
fst (x, y) = x
```

If we apply some `f` to `(0, inf)`, then for call by value:

```
fst (0, inf)
fst (0, 1 + inf)
fst (0, 1 + (1 + inf))
fst (0, 1 + (1 + (1 + inf)))
...
```

Termination

But consider:

```
fst :: (a, b) -> a
fst (x, y) = x
```

If we apply some `f` to `(0, inf)`, then for call by value:

```
fst (0, inf)
fst (0, 1 + inf)
fst (0, 1 + (1 + inf))
fst (0, 1 + (1 + (1 + inf)))
...
```

while for call by name:

```
fst (0, inf)
0
```

Number of reductions

```
square :: Int -> Int
square n = n * n
```

Call-by-value strategy:

```
square (1 + 2)
square 3
3 * 3
9
```

Number of reductions

```
square :: Int -> Int
square n = n * n
```

Call-by-name strategy:

```
square (1 + 2)
(1 + 2) * (1 + 2)
3 * (1 + 2)
3 * 3
9
```

Call-by-name may evaluate an argument more than once!

Number of reductions

Solution: pointers!

```
square (1 + 2)

p * p              p -> (1 + 2)

p * p              p -> 3

9
```

Sharing

call-by-name + sharing = *lazy evaluation*

Infinite structures

```
ones :: [Int]
ones = 1 : ones
```

evaluating `ones`:

```
ones
1 : ones
1 : (1 : ones)
1 : (1 : (1 : ones))
...
```

Now consider `head ones`:

```
head (x : _) = x
```

Call-by-value does not terminate. Call-by-name does!

Modular programming

Separation of *control* and *data*
Be careful:

```
filter (<= 6) [1..]
```

will not stop, while

```
takewhile (<= 6) [1..]
```

will!

```
primes :: [Integer]
primes = sieve [2..]
sieve :: [Int] -> [Int]
sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p /= 0]
```

Strict application

```
f $! x                    f $ x
```

`$!` enforces evaluation of **top-level** of `x` (WHNF)

```
square $! (1 + 2)
square $! 3
square 3
3 * 3
9
```

Strict application

We get three possibilities for two arguments of a curried function:

```
(f $! x) y       -- forces evaluation of x

(f x) $! y       -- forces evaluation of y

(f $! x) $! y    -- forces evaluation of both x and y
```

Strict evaluation saves space (sometimes)

Strict vs. non-strict

- Property of the *semantics* of the language
- Related to *reductions* (evaluations) of expressions
  - top-down
  - bottom-up
- If something evaluates to *bottom* (an error or endless loop) then
  - strict languages will always find the bottom value
  - non-strict languages not need to!

Laziness

Property of an implementation!
Evaluate an expression only when its value is needed.
Common implementation technique for non-strict languages.
Not generally amenable to parallelisation.

Alternative: *lenient* (or optimistic) evaluation; somewhere between lazy and strict — more promising for parallelisation.

Normal forms

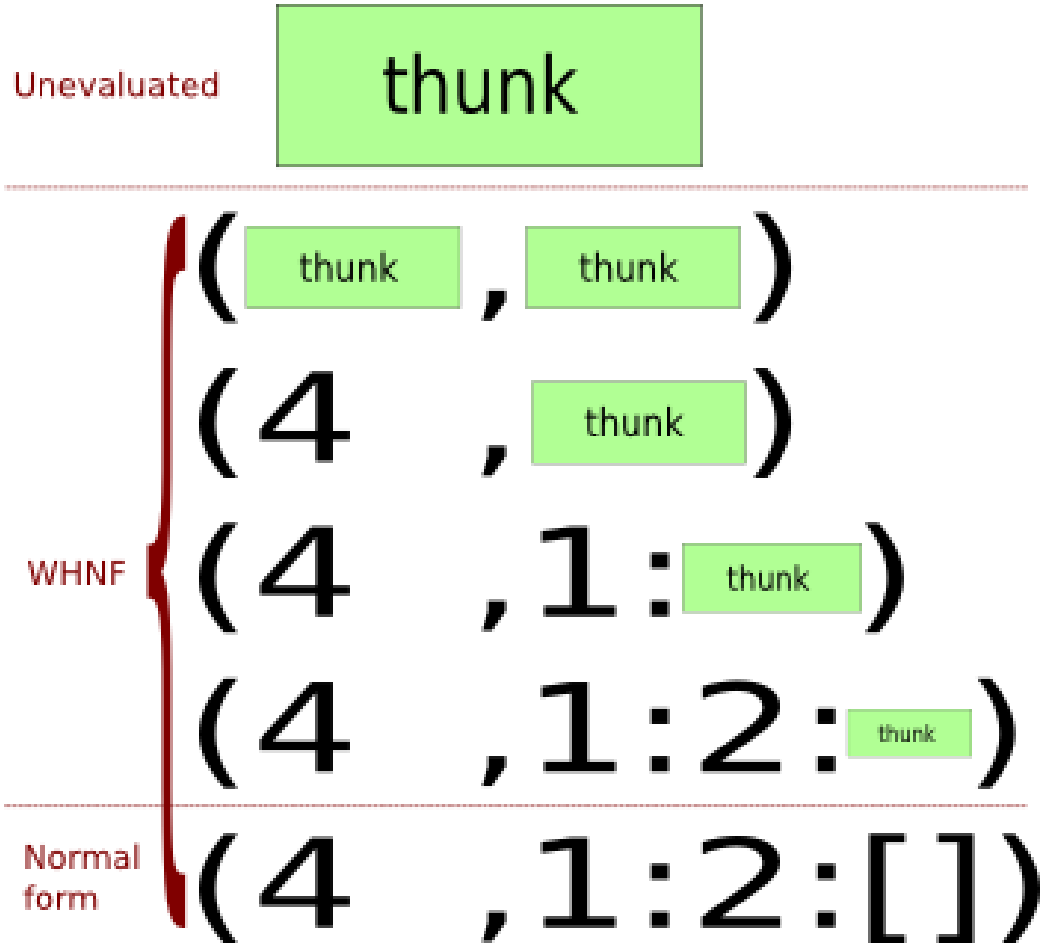
- NF (RNF) – normal form (reduced normal form)
- HNF – head normal form
- WHNF – weak head normal form

Describe the amount of evaluation performed:

- NF – evaluated
- WHNF – evaluated only up to the outermost constructor

Check `http://stackoverflow.com/questions/6872898/haskell-what-is-weak-head-normal-form`

Normal forms



/Taken from Haskell Wikibook/

Normal forms

Example:

```
replicate = \ n x -> case n of
    0 -> []
    n -> x : replicate (n - 1) x
```

Let us evaluate `replicate 3`

- WHNF: `x -> case 3 of  0 -> []; n -> x :  replicate (n - 1) x`
- HNF: `x -> x :  replicate (3 - 1) x`
- NF: `x -> x :  x :  x :  []`

Laziness again

Haskell is not completely lazy!

E.g. pattern matching (a very common situation in any non-trivial piece of code) drives evaluation

Consequences of laziness

- purity (although there exist impure lazy languages, e.g., R)
- space leaks
- short-circuiting operators by default
- infinite data structures
- efficient pipelining
- dynamic programming “for free” (Assignment N2)

Space leaks (or foldl vs. foldl')

```
foldl (+) 0 (1:2:3:[])
          == foldl (+) (0 + 1)              (2:3:[])
          == foldl (+) ((0 + 1) + 2)        (3:[])
          == foldl (+) (((0 + 1) + 2) + 3)  []
          ==           (((0 + 1) + 2) + 3)
```

Thunks stored until needed.
How can we force evaluation?

```
seq ::  a -> b -> b
```

```
foldl' f a []     = a
foldl' f a (x:xs) = let a' = f a x
                    in a' `seq` foldl' f a' xs
```

or

```
foldl' f a (x:xs) = ((foldl' f) $! (f a x)) xs
```

Short-circuiting

- In strict languages: a special-case mechanism wired into language standards
- in lazy languages: the default

```
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && _ = False
```