What is evaluation?

Finding out a value:

\[ \left( \int_0^y x \, dx \right) (2 + 3) \]

Value of an expression:

\[ f(a) = b \]
**Laziness**

**Evaluation order**

\[
\text{inc} :: \text{Int} \to \text{Int} \\
\text{inc} \ n = n + 1 \\
\text{inc} (2 \times 3) \\
\text{inc} \ 6 \quad (2 \times 3) + 1 \\
6 + 1 \\
7
\]

**Some imperative language**

\[
n = 0 \\
y = n + (n = 1) \\
n + (n = 1) \\
0 + (n = 1) \\
n + 1 \\
0 + 1 \\
1 + 1 \\
2
\]

**Reduction**

Reducible expression: \textit{redex}  
Function application to an argument

\[
mult :: (\text{Int}, \text{Int}) \to \text{Int} \\
mult (x, y) = x \times y \\
mult (1 + 2, 2 + 3) \\
(1 + 2) \times (2 + 3) \\
mult (3, 2 + 3) \\
mult (1 + 2, 5) \\
\]

Choice \to leads to evaluation \textit{strategies}

**Evaluation strategies**

\(\bullet\) innermost redex first (usually left-to-right) \textit{call by value}

\[
mult (1 + 2, 2 + 3) \\
mult (3, 2 + 3) \\
mult (3, 5) \\
3 \times 5 \\
15
\]

\(\bullet\) outermost redex first (usually left-to-right) \textit{call by name}

\[
mult (1 + 2, 2 + 3) \\
(1 + 2) \times (2 + 3) \\
3 \times (2 + 3) \\
3 \times 5 \\
15
\]
Lambda expressions

Curried mult:

\[
mult :: Int \to Int \to Int \\
mult x = \lambda y \to x \times y
\]

Innermost evaluation:

\[
mult (1 + 2) (2 + 3) \\
mult 3 (2 + 3) \\
(\lambda y \to 3 \times y) (2 + 3) \\
(\lambda y \to 3 \times y) 5 \\
3 \times 5 \\
15
\]

In Haskell: Do not reduce inside lambdas!

Termination

\[
inf :: Int \\
inf = 1 + inf
\]

Now, irrespectively of evaluation strategy:

\[
inf \\
1 + inf \\
1 + (1 + inf) \\
1 + (1 + (1 + inf)) \\
\ldots
\]

Does not terminate!

But consider:

\[
fst :: (a, b) \to a \\
stst (x, y) = x
\]

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

\[
stst (0, \inf) \\
stst (0, 1 + \inf) \\
stst (0, 1 + (1 + \inf)) \\
stst (0, 1 + (1 + (1 + \inf))) \\
\ldots
\]

\[
\text{while for call by name:}
\]

\[
stst (0, \inf) \\
0
\]

Number of reductions

\[
square :: Int \to Int \\
square n = n \times n
\]

Call-by-value strategy:

\[
square (1 + 2) \\
square 3 \\
3 \times 3 \\
9
\]
**Laziness**

**Number of reductions**

```haskell
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!

Solution: pointers!

```haskell
square (1 + 2)
p * p  p -> (1 + 2)
p * p  p -> 3
9
```

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

**Infinite structures**

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

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

Now consider head ones:

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

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

**Modular programming**

Separation of control and data

Be careful:

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

will not stop, while
takewhile (<= 6) [1..]

will!
```

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

Strict application

\[ f \texttt{!} x \quad \text{f \texttt{!} x} \]
\$! enforces evaluation of \textbf{top-level} of \textit{x} (WHNF)

\[
\text{square } \texttt{!} (1 + 2) \\
\text{square } \texttt{!} 3 \\
3 \times 3 \\
9
\]

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

\[
(f \texttt{!} x) y \quad -- \text{forces evaluation of } x \\
(f x) \texttt{!} y \quad -- \text{forces evaluation of } y \\
(f \texttt{!} x) \texttt{!} y \quad -- \text{forces evaluation of both } x \text{ and } y
\]

Strict evaluation saves space (sometimes)

Laziness

Strict vs. non-strict

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

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: \textit{lenient} (or optimistic) evaluation; somewhere between lazy and strict — more promising for parallelisation.
Laziness

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


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 : []

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

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