COMP105 Lecture 26

Evaluation Strategies

Motivating example

```
inc x = x + 1

square x = x * x
```

So `square (inc 2) = (2 + 1) * (2 + 1) = 9`

How does Haskell **evaluate** this?

Strict Evaluation

In strict evaluation, we always apply the **innermost** functions first

```
square (inc 2)
```

→ `square (3)`

→ `3 * 3`

→ `9`

First apply `inc` then apply `square`

Lazy Evaluation

In lazy evaluation, we always apply the **outermost** functions first

```
square (inc 2)

→ inc 2 * inc 2

→ 3 * 3

→ 9
```

First apply `square` then apply `inc`

Lazy Evaluation

In the following, nothing is computed until we ask for the value of z

```
ghci> let x = 1 + 1
ghci> let y = x + x
ghci> let z = y / 2
ghci> z
2.0

ghci> let x = 1 + undefined
ghci> let y = x + x
ghci> let z = y / 2
ghci> z
*** Exception: Prelude.undefined
```

Lazy Evaluation

If a value is **never used**, it is **never computed**

```
ghci> let f x = 1

ghci> f 2
1

ghci> f 3
1

ghci> f undefined
1
```

Imperative languages

Imperative languages are **always** strict

- So programmers know the order of side effects

```
def inc(x):
    print "hi from inc"
    return x + 1

def square(x):
    print "hi from square"
    return x * x
```

```
>>> print square(inc(2))
hi from inc
hi from square
9
```

Functional languages

For pure functions the order of evaluation is **irrelevant**.

- You will always get the same answer
- No need to worry about side effects

Some functional programming languages are strict by default

- eg. ML, Ocaml

Others are lazy by default

- eg. Haskell

Lazy vs. Strict

If strict evaluation finds an answer, then lazy evaluation will find the **same** answer

Sometimes lazy evaluation can find an answer when strict **can't**

```
ghci> fst (1, 1 `div` 0)
1
```

Strict evaluation would crash on this example

Lazy vs. Strict

Sometimes lazy evaluation can be more efficient than strict

- Particularly if some values are computed but never used

```
double (x, y) = (2*x, 2*y)
```

```
ghci> fst (double (3, 4))
6
```

Lazy evaluation produces:

```
fst (double (3, 4))
→ 2 * 3
→ 6
```

Lazy computation

Lazy evaluation only ever computes a value when it is needed

```
ghci> let x = 1 `div` 0
ghci> x
*** Exception: divide by zero
```

This can lead to big efficiency savings, eg. in

```
ghci> head (map (*2) [1..100])
2
```

Lazy evaluation just computes `2 * 1`

Lazy vs. Strict summary

Strict evaluation

- Evaluate things as **soon** as possible
- Gives certainty over order of side effects

Lazy evaluation

- Evaluate things only when they are **needed**
- Potentially eliminates unneeded computation

Exercise

What are the outputs for the following queries?

```
ghci> fst (1, error "error")
```

```
ghci> show [10 `div` 2, 10 `div` 1, 10 `div` 0]
```

```
ghci> let x = []
ghci> let y = head x
ghci> let z = y `div` 0
ghci> z
```