Lazy evaluation

Evaluation

- **Evaluation** refers to the algorithm that runs your program
- In Haskell evaluation is best understood as ‘replacing function names by their definitions (right hand sides)’
- But most programs contain more than one function name, raising the question of which *order* your program will evaluate in
- This can be significant for termination, for speed, and for the timing of any side effects that happen during your program’s run

These slides give an informal description of Haskell’s approach to evaluation.

A simple example

Consider the program

```
addOne :: Int -> Int
addOne x = x + 1
```

How should we evaluate

```
addOne (2 * 3)
```

A simple example

Option one: evaluate the input to the function first:

```
addOne (2 * 3) = addOne 6
               = 6 + 1
               = 7
```

Option two: put the input expression into the function, evaluating it later:

```
addOne (2 * 3) = (2 * 3) + 1
               = 6 + 1
               = 7
```

Strict vs Lazy

The first approach, where we evaluate all inputs before we touch the function, is called **strict**

The second approach, where we insert the unevaluated input into the function and only evaluate it when needed (if at all!) is called **lazy**

Haskell is, by default, lazy

- You can force strict evaluation if you wish
- More languages are the other way around – strict by default, but with the capacity to mimic lazy evaluation

`const` and laziness

Consider the Prelude function

```
const :: a -> b -> a
const x _ =  x
```

`const` never uses its second input. In combination with laziness, this creates some interesting behaviour.

Moral – inputs that are not used are never evaluated

Programming with streams

A **stream** is an infinite list

In Haskell infinite lists have the same list type we're used to using

I.e. while we have usually intended our lists to be finite with base case `[ ]`, there is nothing stopping us defining a list of form

```
x1 : x2 : x3 : x4 : x5 …
```

with no base case!

zeros

```
zeros :: [Int]
zeros  = 0 : zeros                                    (definition)
       = 0 : 0 : zeros
       = 0 : 0 : 0 : zeros
       = 0 : 0 : 0 : 0 : zeros
       = 0 : 0 : 0 : 0 : 0 : zeros
       = …
```

Using streams

If you run zeros it will not terminate

- Click Ctrl+C to kill the process

Other programs like `sum zeros` will also fail to terminate. But…

```
> const True zeros
True
> take 10 zeros
[0,0,0,0,0,0,0,0,0,0]
> zeros!!1000000000
0
```

A more interesting example: `nats`

```
nats :: [Integer]
nats  = 0 : map (+1) nats                                      (definition)
      = 0 : map (+1) (0 : map (+1) nats)
      = 0 : 1 : map (+1) (map (+1) nats)
      = 0 : 1 : map (+1) (map (+1) (0 : map (+1) nats))
      = 0 : 1 : map (+1) (1 : map (+1) (map (+1) nats))
      = 0 : 1 : 2 : map (+1) (map (+1) (map (+1) nats))
```

Even more interesting

```
mystery :: [Integer]
mystery = 0 : 1 : zipWith (+) mystery (tail mystery)
```

What might this do?

A problem with lazy evaluation?

Lazy evaluation could be very costly if an input is used many times:

```
replicate :: Int -> a -> [a]
replicate n x
  | n <= 0    = []
  | otherwise = x : replicate (n-1) x
```

With strict evaluation `replicate n foo` will calculate `foo` once.
With lazy, it might calculate it n times!

One more clever Haskell trick

In fact Haskell does not fall into this trap: if it can see that a variable is used many times, it will calculate it once and refer to that value every time it is used.

But this only works for variables (e.g. `x` in the last example). Haskell will not detect that you have repeated larger amounts of code!

Therefore you need to avoid repeating code, e.g. using variables defined by `where` clauses.

Summary

- You need to understand what order programs evaluates in to completely understand the termination behaviour and speed of your code
- Haskell is **lazy** – doesn’t evaluate inputs until it has to
- Laziness has advantages
    - More likely to terminate
    - Can avoid expensive computations except when you need them
    - Allows working with infinite data
- But also disadvantages
    - More unpredictable with respect to side effects
    - Harder to reason about space efficiency (working memory), and sometimes time efficiency
    - Sometimes wastes space
    - ‘Clever tricks’ can be expensive in their own right