Contents

Lazy Evaluation

Haskell uses *Lazy* (*non-strict*) *Evalulation*

- Expressions are only evaluated when they're needed
- Argument expressions arent evaluated before a function is applied
- We find this approach allows us to write sensible programs not possible if strict-evaluation is used
- However, is comes at a price…

isOdd

- We define a function checking for 'oddness' as follows:

```
isOdd n = n `mod` 2 == 1
```

- Consider the call `isOdd(1+2)`
- A strict evaluation would be as follows:

```
isOdd(2+3)
  = isOdd(3)
  = 3 `mod` 2 == 1
  = 1==1
  = True
```

- A lazy evaluation would be as follows:

```
isOdd(2+3)
  = (1+2) `mod` 2 == 1
  = 3 `mod` 2 == 1
  = 1==1
  = True
```

`1+2` is only evaluated when `mod` needs its value to proceed

len and down

- We have a length function `len`:

```
len xs = if null xs then 0 else 1+len (tail xs)
```

- We have a function `down` that generates a list, counting down from its numeric argument:

```
down n = if n <= 0 then [] else n : (down (n-1))
```

For example, `down 3 = [3, 2, 1]`

- We shall consider pattern matching versions shortly

Strict Evaluation of len (down 1)

```
len (down 1)
= len (if 1 <= 0 then [] else 1 : (down (1-1))
= len (1 : (down (1-1))
= len (1 : (down 0))
= len (1 : (if 0 <= 0 then [] else 0 : (down (0-1))))
= len (1 : [])
= if null (1 : []) then 0 else 1 + len (tail (1 : []))
= 1 + len (tail (1 : []))
= 1 + len []
= 1 + len (if null [] then 0 else 1 + len (tail []))
= 1 + 0
= 1
```

Lazy Evaluation of len (down 1)

```
len (down 1)
= if null xs1 then 0 else 1 + len (tail xs1)
  where xs1 = down 1
= if null xs1 then 0 else 1 + len (tail xs1)
  where xs1 = if 1 <= 0 then [] else 1 : (down (1-1))
= if null xs1 then 0 else 1 + len (tail xs1)
  where xs1 =  1 : (down (1-1))
= 1 + len (tail xs1) where xs1 =  1 : (down (1-1))
= 1 + ( if null xs2 then 0 else 1 + len (tail xs2)
    where xs2 = tail xs1)
  where xs1 =  1 : (down (1-1))
= 1 + ( if null xs2 then 0 else 1 + len (tail xs2)
    where xs2 = tail xs1)
  where xs1 =  1 : (down (1-1))
= 1 + ( if null xs2 then 0 else 1 + len (tail xs2)
    where xs2 =  tail (1 : (down (1-1))))
= 1 + ( if null xs2 then 0 else 1 + len (tail xs2)
    where xs2 =  down (1-1))
= 1 + ( if null xs2 then 0 else 1 + len (tail xs2)
    where xs2 = (if (1-1) <= 0
        then [] else (1-1) : (down ((1-1)-1))) )
= 1 + ( if null xs2 then 0 else 1 + len (tail xs2)
    where xs2 = (if 0 <= 0
        then [] else (1-1): (down ((1-1)-1))) )
= 1 + ( if null xs2 then 0 else 1 + len (tail xs2)
    where xs2 = [] )
= 1 + 0
= 1
```

Why the xs1 = ... ?

- Consider the first step

```
len (down 1)
  = if null xs1 then 0 else 1 + len (tail xs1)
    where xs1 = down 1
```

- We don't evaluate `down 1` - we bind it to formal parameter `xs1`
- Parameter `xs` occurs twice, but we don't copy `...down 1...down 1....` Instead we share the reference, indicated by the `where` clause: `...xs1....xs1...where xs1 = down 1`
- Function `len` is recursive, so we get different instances of `xs` which we label as `xs1 xs2`, ...

- The grouping of an (unevaluated) expressions (`down 1`) with a binding (`xs1 = down 1`) is called eith er a "*closure*", or a "*thunk*"
- Building thunks is a *necessary* overhead for implementing lazy evaluation

Costs

- Lazy evaluation has overhead: building thunks
- Memory consumption per reduction of step is typically slightly higher
- In our examples so far we needed to evaluate almost everything

```
isOdd (1+2)
len (down 1)
```

- So far we have observed no advantage to lazy evaluation

Advantages

- Imagine we have a function definition as follows:

```
myfun carg struct1 struct 2
  = if f carg
    then g struct1
    else h struct2
```

where `f`, `g` and `h` are internal functions

- Consider the following call

```
myfun val s1Expr s2Expr
```

where both `s1Expr` and `s2Expr` are very expensive to evaluate

- With strict evaluation we would have to compue both before applying `myfun`
- With lazy evaluation we evaluate `f val`, and then only evaluate one of either `s1Expr` or `s2Expr`, and then, only if `g` or `h` requires its value
- Prelude function `take n xs` returns the first `n` elements of `xs`

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

- Function `from n` generates an *infinite* ascending list starting with `n`

```
from n = n : (from (n+1))
```

- Evaluating `from n` will fail to terminate for any `n`
- Evaluation of `take 2 (from 0)` depends on the evaluation method

Laziness and Pattern Matching

- Consider a pattern matching version of `len`

```
len [] = 0
len (x:xs) = 1 + len xs
```

- How is call `len aListExpression` evaluated?
- In order to pattern match we need to know it `aListExpression` is empty, or a cons-node
- We evaluate `aListExpression`, but only to the point where we know the different.
  - If it is not null, we do not evaluate the head element, or the tail list
- e.g. is `aListExpression` = map f (1:2:3:[]), where

```
map f [] = []
map f (x:xs) = f x : map f xs
```

then we only evaluate `map f` as far as `f 1 : map f (2:3:[])`

Evaluation Strategy and Termination

We can summarise the relationship between evaluation strategy and termination as:

- There are programs that simply do not terminate, no matter how they are evaluated
  - e.g. `from 0`
- There are programs that terminate if evaluated lazily, but fail to terminate if evaluated strictly
  - e.g. `take 2 (from 0)`

- There are programs that terminate regardless of chosen evaluation strategy
  - e.g. `len (down 1)`
- However, there are *no* programs that terminate if evaluated strictly, but fail to terminate if evaluated lazily