BAHUG 101 - Lecture 6

15th December 2015

Outline of Today's Lecture

- Strict evaluation
- Side effects and purity
- Adding strictness
- Short-circuiting
- Infinite Data Structures
- Profiling

Strict evaluation

Strict evaluation in other languages

Let's pretend Haskell is strict. If we have the following function

```
f x y = x + 2
```

and then evaluate

```
f 5 (29^35792)
```

we would

Strict evaluation in other languages

Let's pretend Haskell is strict. If we have the following function

```
f x y = x + 2
```

and then evaluate

```
f 5 (29^35792)
```

we would

- evaluate 5 to 5

Strict evaluation in other languages

Let's pretend Haskell is strict. If we have the following function

```
f x y = x + 2
```

and then evaluate

```
f 5 (29^35792)
```

we would

- evaluate 5 to 5
- evaluate 29^{35792} to something 52,343 digits long.

Strict evaluation in other languages

Let's pretend Haskell is strict. If we have the following function

```
f x y = x + 2
```

and then evaluate

```
f 5 (29^35792)
```

we would

- evaluate 5 to 5
- evaluate 29^{35792} to something 52,343 digits long.
- Pass both *values* to f.

Strict evaluation in other languages

Let's pretend Haskell is strict. If we have the following function

```
f x y = x + 2
```

and then evaluate

```
f 5 (29^35792)
```

we would

- evaluate 5 to 5
- evaluate 29^{35792} to something 52,343 digits long.
- Pass both *values* to f.

However, for the definition of `f`, we never use `y`!

Why use strict evaluation

Strict evaluation is convenient in many cases.

- The programmer can reason about the order of execution of a program.
- The programmer can easily perform order-dependent state changes.

```
f (release_monkeys(), increment_counter())
```

Lazy Evaluation

Unlike strict evaluation, in lazy evaluation the evaluation of a function’s arguments is delayed as long as possible.

Lazy Evaluation

Unlike strict evaluation, in lazy evaluation the evaluation of a function's arguments is delayed as long as possible.

Unevaluated expressions are known as *thunks*.

Lazy Evaluation

Unlike strict evaluation, in lazy evaluation the evaluation of a function's arguments is delayed as long as possible.

Unevaluated expressions are known as *thunks*.

In the case of

```
f 5 (29^35792)
```

29^{35792} is turned into a thunk and not evaluated.

Pattern matching drives evaluation

The following function does not cause `m` to be evaluated because the value of `m` is not needed by the function.

```haskell
f1 :: Maybe a -> [Maybe a]
f1 m = [m,m]
```

Pattern matching drives evaluation

The following function does not cause `m` to be evaluated because the value of `m` is not needed by the function.

```
f1 :: Maybe a -> [Maybe a]
f1 m = [m,m]
```

However, it is necessary to evaluate a value to it's outer constructor when pattern matching.

```
f2 :: Maybe a -> [a]
f2 Nothing  = []
f2 (Just x) = [x]
```

Note here that for `f2`, `x` is not evaluated.

Example of pattern matching driven evaluation

```haskell
f2 :: Maybe a -> [a]
f2 Nothing  = []
f2 (Just x) = [x]
```

Let's try with the following simple value.

```haskell
Prelude> let x = Just $ map (+1) [1..10] :: Maybe [Int]
Prelude> let y = f2 x
```

Example of pattern matching driven evaluation

```haskell
f2 :: Maybe a -> [a]
f2 Nothing  = []
f2 (Just x) = [x]
```

Let's try with the following simple value.

```
Prelude> let x = Just $ map (+1) [1..10] :: Maybe [Int]
Prelude> let y = f2 x
```

We can call the `:sprint` GHCi function to show the current evaluation of an expression.

```
Prelude> :sprint y
y = _
Prelude> :sprint x
x = _
```

Example of pattern matching driven evaluation

```
f2 :: Maybe a -> [a]
f2 Nothing  = []
f2 (Just x) = [x]
```

Let's try with the following simple value.

```
Prelude> let x = Just $ map (+1) [1..10] :: Maybe [Int]
Prelude> let y = f2 x
```

Let's find how many elements **f2** created (note **length** does not need the values inside a list)

```
Prelude> length y
1
Prelude> :sprint y
y = [_]
Prelude> :sprint x
x = Just _
```

Example of pattern matching driven evaluation

```haskell
f2 :: Maybe a -> [a]
f2 Nothing  = []
f2 (Just x) = [x]
```

Let's try with the following simple value.

```
Prelude> let x = Just $ map (+1) [1..10] :: Maybe [Int]
Prelude> let y = f2 x
```

If we print `y`, we need the value of `x` as well.

```
Prelude> y
[[2,3,4,5,6,7,8,9,10,11]]
Prelude> :sprint x
x = Just [2,3,4,5,6,7,8,9,10,11]
```

Takeaway slogan

The slogan to remember is *pattern matching drives evaluation*. To reiterate the important points:

- Expressions are only evaluated when pattern-matched

Takeaway slogan

The slogan to remember is *pattern matching drives evaluation*. To reiterate the important points:

- Expressions are only evaluated when pattern-matched
- only as far as necessary for the match to proceed, and no farther!

Another example

Let's go through another example.

```
take 3 (repeat 7)
```

As a reminder, here are the definitions of **repeat** and **take**.

```
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _       | n <= 0 =  []
take _ []               =  []
take n (x:xs)           =  x : take (n-1) xs
```

Another example stepthrough

```haskell
repeat :: a → [a]
repeat x = x : repeat x

take :: Int → [a] → [a]
take n _       | n ⩽ 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs
```

Step through

```haskell
take 3 (repeat 7)
```

Another example stepthrough

```
repeat :: a → [a]
repeat x = x : repeat x

take :: Int → [a] → [a]
take n _        | n ⩽ 0 =  []
take _ []               =  []
take n (x:xs)           =  x : take (n-1) xs
```

Step through

```
  take 3 (repeat 7)
```

Evaluate **take**. First two pattern matches fail, but the third matches on expansion.

```
= take 3 (7 : repeat 7)
```

Another example stepthrough

```
repeat :: a → [a]
repeat x = x : repeat x

take :: Int → [a] → [a]
take n _        | n ⩽ 0 =  []
take _ []               =  []
take n (x:xs)           =  x : take (n-1) xs
```

Step through

```
= take 3 (7 : repeat 7)
```

Another example stepthrough

```
repeat :: a → [a]
repeat x = x : repeat x

take :: Int → [a] → [a]
take n _       | n ⩽ 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs
```

Step through

```
= take 3 (7 : repeat 7)
```

Replace with the definition of `take` in the third pattern match.

```
= 7 : take (3-1) (repeat 7)
```

Another example stepthrough

```haskell
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _       | n <= 0 =  []
take _ []               =  []
take n (x:xs)           =  x : take (n-1) xs
```

Step through

```haskell
= 7 : take (3-1) (repeat 7)
```

Another example stepthrough

```
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs
```

Step through

```
= 7 : take (3-1) (repeat 7)
```

Attempt to evaluate **take** again. The attempt to match the first clause cause the substraction to occur.

```
= 7 : take 2 (repeat 7)
```

Another example stepthrough

```
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _       | n <= 0 =  []
take _ []               =  []
take n (x:xs)           =  x : take (n-1) xs
```

Step through

```
= 7 : take 2 (repeat 7)
```

Another example stepthrough

```
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _       | n <= 0 =  []
take _ []               =  []
take n (x:xs)           =  x : take (n-1) xs
```

Step through

```
= 7 : take 2 (repeat 7)
```

Evaluate **take**. First two pattern matches fail, but the third matches on expansion.

```
= 7 : take 2 (7 : repeat 7)
```

Another example stepthrough

```
repeat :: a → [a]
repeat x = x : repeat x

take :: Int → [a] → [a]
take n _        | n ⩽ 0 =  []
take _ []               =  []
take n (x:xs)           =  x : take (n-1) xs
```

Step through

And the rest

```
= 7 : 7 : take (2-1) (repeat 7)
= 7 : 7 : take 1 (repeat 7)
= 7 : 7 : take 1 (7 : repeat 7)
= 7 : 7 : 7 : take (1-1) (repeat 7)
= 7 : 7 : 7 : take 0 (repeat 7)
= 7 : 7 : 7 : []
```

Consequences of purity

Purity

In order to enable lazy evaluation, the language is pure. Otherwise not knowing the evaluation order is very challenging to program

Space Usage

Laziness can have good, and poor, space usage.

```
badSum :: Num a => [a] -> a
badSum []     = 0
badSum (x:xs) = x + badSum xs
```

`badSum` is not tail recursive and processes the list right to left.

Space Usage can even be bad for tail recursion

A tail recursive function can also have poor space usage.

```
lazySum :: Num a ⇒ [a] → a
lazySum = go 0
  where go acc []     = acc
        go acc (x:xs) = go (x + acc) xs
```

The problem is that all those uses of `(+)` never get evaluated, until the very end.

lazySum stepthrough

```
  lazySum [1,2,3,4]
= go 0 [1,2,3,4]
= go (1 + 0) [2,3,4]
= go (2 + (1 + 0)) [3,4]
= go (3 + (2 + (1 + 0))) [4]
= go (4 + (3 + (2 + (1 + 0)))) []
= (4 + (3 + (2 + (1 + 0))))
= (4 + (3 + (2 + 1)))
= (4 + (3 + 3))
= (4 + 6)
= 10
```

Forcing evaluation using **seq**

Here is a version with good space performance by forcing the accumulator to be evaluated.

```
strictSum :: Num a => [a] -> a
strictSum = go 0
  where go acc []     = acc
        go acc (x:xs) = acc `seq` go (x + acc) xs
```

where `seq :: a -> b -> b` forces the value of the first parameter before returning the second.

`strictSum` stepthrough

```
strictSum [1,2,3,4]
go 0 [1,2,3,4]
go (1 + 0) [2,3,4]
go (2 + 1) [3,4]
go (3 + 3) [4]
go (4 + 6) []
(4 + 6)
10
```

Another way to write strictSum

Another way to write **strictSum** is using the **BangPatterns** language extension, enabled at the top of this file

```
strictSum' :: Num a ⇒ [a] → a
strictSum' = go 0
  where go  acc []     = acc
        go !acc (x:xs) = go (x + acc) xs
```

Note the **!** before **acc** in the second equation for **go**. Just like **seq**, this forces **acc** to be evaluated.‘

Short-circuiting operators

In many languages (C, Java, etc) have certain operators that *short circuit*: they evaluate one argument and then potentially return without evaluating the second. Examples include `&&` and `||`.

Short-circuiting operators

In many languages (C, Java, etc) have certain operators that *short circuit*: they evaluate one argument and then potentially return without evaluating the second. Examples include `&&` and `||`.

In Haskell, we get that ability for free! No special semantics required.

```haskell
(&&) :: Bool → Bool → Bool
True  && x = x
False && _ = False
```

Regular (&&) operator

We could have defined `(&&)`, and it would produce the same values.

```
(&&!) :: Bool → Bool → Bool
True  &&! True  = True
True  &&! False = False
False &&! True  = False
False &&! False = False
```

Regular (&&) operator

We could have defined (&&), and it would produce the same values.

```haskell
(&&!) :: Bool → Bool → Bool
True  &&! True  = True
True  &&! False = False
False &&! True  = False
False &&! False = False
```

But it would not short circuit.

```haskell
False &&  (34^9784346 > 34987345) — Time to evaluate: 0.02 secs
False &&! (34^9784346 > 34987345) — Time to evaluate: 0.54 secs
```

Regular (&&) operator

We could have defined (&&), and it would produce the same values.

```
(&&!) :: Bool → Bool → Bool
True  &&! True  = True
True  &&! False = False
False &&! True  = False
False &&! False = False
```

But it would not short circuit.

```
False &&  (34^9784346 > 34987345) — Time to evaluate: 0.02 secs
False &&! (34^9784346 > 34987345) — Time to evaluate: 0.54 secs
```

And this would fail

```
False &&  (head [] == 'x') — False
False &&! (head [] == 'x') — *** Exception: head: empty list
```

User defined control structure

In Haskell, we could define our own **`if`** without syntactic sugar.

```
if' :: Bool -> a -> a -> a
if' True  x _ = x
if' False _ y = y
```

There are other control structures that can be built, but we will discuss those in a later lecture.

Infinite Data Structures

We can work on infinite data structures, like the following. We only use what we need

```
repeat 7 :: [Int]
```

Infinite Data Structures

We can work on infinite data structures, like the following. We only use what we need

```
repeat 7 :: [Int]
```

We can also work with practically infinite structures, like a tree representing the number of games in chess ($10^{10^{50}}$). Only the nodes visited are evaluated.

Infinite Data Structures

We can work on infinite data structures, like the following. We only use what we need

```
repeat 7 :: [Int]
```

We can also work with practically infinite structures, like a tree representing the number of games in chess ($10^{10^{50}}$). Only the nodes visited are evaluated.

Working with infinite lists is particularly common.

```
withIndices :: [a] -> [(a,Integer)]
withIndices xs = zip xs [0..]
```

We could also define the `[0..]` list this way:

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

Profiling

How to enable profiling

To profile a program it needs a **`main`** function. For example

```
main = print (lazySum [1..1000000])
```

We can then compile with the **`-rtsopts`** flag.

```
ghc main.hs -rtsopts
```

How to call the profiled program

Then, when calling the executable, pass in the RTS options

```
./main +RTS -s -h -i0.01
```

where

- `-s` is to record memory and time usage,
- `-h` is to create a heap profile, and
- `-i` is to set the heap sampling interval.

The resulting **main.hp** can be visualized with the **hp2ps** utility.

Profiling `lazySum`

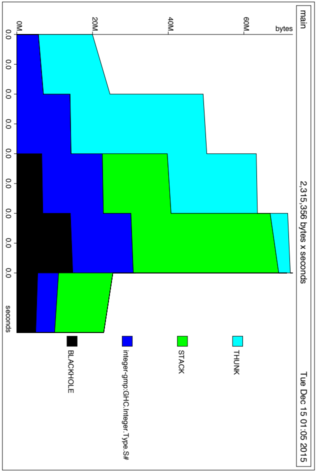

Figure 1: `lazySum` profile

Profiling `strictSum`

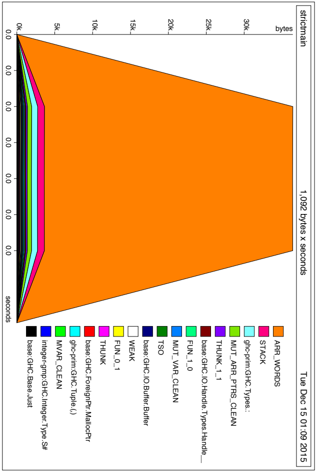

Figure 2:`strictSum` profile