Lab 12

Lazy evaluation and Infinite data structures in Haskell

Goals

In this lab you will learn to:

1. Understand lazy evaluation in Haskell
2. Control the behavior of lazy evaluation in Haskell
3. Work with infinite lists and circular data structures

Resources

Table 12.1: Lab Resources

Resource	Link
Lecture 6 : Error handling in Haskell	Lecture 6 on Moodle
Lecture 8	Lecture 8 on Moodle
Haskell wiki: Laziness	`https://wiki.haskell.org/Performance/Laziness`
Haskell wiki: Strictness	`https://wiki.haskell.org/Performance/Strictness`
Haskell wiki: foldr vs foldl vs foldl'	`https://wiki.haskell.org/Foldr_Foldl_Foldl'`

12.1 Exceptions and bottom values

12.1.1 Bottom values

Concept 12.1.1: Bottom values

The bottom value in a programming language can be assigned to *any* type, and crashes the program when evaluated at runtime.

Question 12.1.1

What is the bottom type in Elm?

In Haskell the bottom value is `undefined`. As in Elm, you can use it to temporarily make your code compile.

12.1.2 Exceptions

Question 12.1.2

How would you define exceptions? What languages that you know have exceptions? When should you use exceptions?

Note 12.1.1

In Functional Programming, the possibility of failure can be signaled in the return type of functions, so the use of exceptions is **discouraged**!

Exceptions in Haskell

Question 12.1.3 *

Which error handling method do you prefer? Raising exceptions or using types like `Maybe` and `Result`? Try to find one advantage and one disadvantage for each error handling method.

12.2 Call-by-value vs Call-by-name

Lets try to implement our own `if` expression in Elm and Haskell, called `myIf`:

Elm REPL

```
> myIf cond t f = if cond then t else f
<function> : Bool -> a -> a -> a
```

Haskell REPL

```
> myIf cond ifTrue ifFalse = if cond then ifTrue else ifFalse
```

Lets test it with some examples:

```
Elm REPL

> myIf (0 == 0) "Good" "Bad"
"Good" : String
> myIf (1 == 0) "Bad" "Good"
"Good" : String
```

```
Haskell REPL

> myIf (0 == 0) "Good" "Bad"
"Good"
> myIf (0 == 1) "Bad" "Good"
"Good"
```

So far it seems that is working as expected.

Lets try to use it with some expression that contains the bottom value:

```
Elm REPL

> if (1 == 0) then (Debug.todo "Boom") else "Ok"
"Ok" : String
> myIf (1 == 0) (Debug.todo "Boom") "Ok"
Error: TODO in module `Elm_Repl` on line 4

Boom
```

```
Haskell REPL

> if (1 == 0) then (error "Boom") else "Ok"
"Ok"
> myIf (1 == 0) (error "Boom") "Ok"
"Ok"
```

As you can see we forgot about the key property of `if` expressions in all languages: they evaluate **only one branch**, depending on the result of the condition. When calling the `myIf` function in Elm, both arguments were evaluated, crashing the program (because one was the bottom value).

Because Haskell is lazy by default, `myIf` behaves like the normal `if` would, not evaluating the bottom value until required.

12.3 Infinite ranges in Haskell

In Haskell we can create infinite ranges, by omitting the upper bound:

```
Haskell REPL

> take 10 [1..]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
> take 5 (filter (\x -> mod x 3 == 0) [1..])
[3, 6, 9, 12, 15]
```

12.4 Inspecting structures with the `:sprint` command in GHCi

GHCi offers a command named `:sprint` that shows the evaluated part of a given data structure.

Haskell REPL

```
> l = [1..] :: [Int]
> :sprint l
l = _
> head l
1
> :sprint l
l = 1 : _
> take 5 (filter (\x -> mod x 3 == 0) l)
[3,6,9,12,15]
> :sprint l
l = 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10 : 11 : 12 : 13 : 14 : 15 : _
```

Note 12.4.1

In Haskell the integer literals are *polymorphic*: they can be both `Int` and `Integer`. Why does this matter? Because when we write a list of numbers like `[1, 2, 3]` Haskell won't assign a concrete type (`Int` or `Integer`) for storing the numbers, so if we try to use `:sprint` with such a list (i.e. for which the concrete type is not known), the result won't be valid.

Exercise 12.4.1 *

What will be printed by the `:sprint l` after running the following commands:

Haskell REPL

```
> l = [1,3..] :: [Int]
> take 3 (filter (\x -> mod (x - 1) 2 == 0) l)
```

Exercise 12.4.2 *

What will be printed by the `:sprint l` after running the following commands:

Haskell REPL

```
> l = [1,10..] :: [Int]
> take 3 (filter (\x -> mod x 10 == 0) l)
```

12.5 Controlling laziness in Haskell

12.5.1 Showing execution time in ghci

To view the time it took to evaluate an expression we can use the `:set +s` command to instruct ghci to display the execution time and memory usage of each evaluation.

12.5.2 Normal Form and Weak Head Normal Form

Concept 12.5.1: Normal Form

An expression is in Normal Form when it is fully evaluated.

Concept 12.5.2: Weak Head Normal Form

An expression is in Weak Head Normal Form when it is lambda expression or function waiting to be applied to some arguments or an expression evaluated up to the outermost constructor.

12.5.3 The seq function

The `seq :: a -> b -> b` function has a special meaning in Haskell, because it evaluates the first argument to WHNF and returns the second argument.

Haskell REPL

```
> t = 1 + 1 :: Int
> :sprint t
t = _
> t `seq` ()
()
> :sprint t
t = 2
```

Haskell REPL

```
> t = [1..] :: [Int]
> :sprint t
t = _
> t `seq` ()
()
> :sprint t
t = 1 : _
```

12.5.4 Case study: foldl in Haskell is actually not perfect

When learning about `foldlr` and `foldl` in Elm, we noted that `foldl` should be used because `foldr` starts evaluating the fold from the back of the list, so it needs to know the full list to return a result, `foldl` starts from the front of the list, so it can process the list incrementally.

While the above is also true in Haskell, due to it's lazy nature, it still won't save us from stack overflows.

Haskell REPL

```
> foldr (+) 0 [1..10^7] :: Integer
50000005000000
> foldr (+) 0 [1..10^8] :: Integer
*** Exception: stack overflow
> foldl (+) 0 [1..10^8] :: Integer
*** Exception: stack overflow
```

While the first stack overflow for `foldl` is expected, the second one for `foldl` is certainly a surprise. This happens because Haskell will build up the whole expression of $(((0 + 1) + 2) + 3) + \ldots$ in memory and evaluate it only when it is fully built (because by then the list will be fully processed and there will be nothing else left to do).

To force Haskell to evaluate the partial results instead of delaying the evaluation, we can use the `seq` function:

Listing 12.5.1: Strict foldl — Haskell code

```
foldl' _ start [] = start
foldl' op start (x:xs) = next `seq` foldl' op next xs where next = op start x
```

Not only will this version of `foldl` succeed where the others failed:

Haskell REPL

```
> foldl' (+) 0 [1..10^8] :: Integer
5000000050000000
```

but it will be faster and use less memory too:

Haskell REPL

```
>:set +s
> foldl (+) 0 [1..10^7] :: Integer
50000005000000
(6.41 secs, 1,612,360,464 bytes)
> foldl' (+) 0 [1..10^7] :: Integer
50000005000000
(2.45 secs, 880,060,176 bytes)
```

12.6 Circular lists and data streams in Haskell

Listing 12.6.1 of `Circular.hs` — Haskell code

```haskell
module Circular where

twos :: [Integer]
twos = 2:twos

rep :: t -> [t]
rep e = e:(rep e)

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

count :: [Integer]
count = 1:(map (+1) count)

powsOf2 :: [Integer]
powsOf2 = 2:(map (*2) powsOf2)

oneList :: [[Integer]]
oneList = [1]:(map (1:) oneList)

primes :: [Integer]
primes = sieve [2..] where
  sieve (x:xs) = x:sieve [ y | y <- xs, mod y x /= 0]
```

Exercise 12.6.1 *

For each function above, try to predict its output and then run it to see if you were correct.

Exercise 12.6.2 *

Trace the `fibs`, `powsOf2` and `primes` functions for the first 3 elements.

12.7 Review questions

Question 12.7.1

Of the following functions, which are guaranteed to always terminate, even on infinite lists?

1. `sort`
2. `take`
3. `min` (find the minimum value)
4. `elem` (check if a value is in the list)
5. `head`
6. `filter`

12.8 Practice problems

Exercise 12.8.1 *

Write a function called `cycl :: [a] -> [a]` that takes list as parameter and repeats the list infinitely.

Haskell REPL

```
> take 10 (cycl [1, 2, 3])
[1, 2, 3, 1, 2, 3, 1]
```

Exercise 12.8.2 *

Write a function called `series :: [[Int]]` that generates the following list: `[[1], [2, 1], [3, 2, 1], ...]`

Exercise 12.8.3 **

Write a function called `iter :: (a -> a) -> a -> [a]` which takes a function `f` and a starting value `a` and returns an infinite list which contains: `[a, f a, f (f a), f (f (f a)), ...]`

Haskell REPL

```
> take 10 (iter (+1) 0)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
> take 5 (iter (\x -> x / 2) 100)
[100.0,50.0,25.0,12.5,6.25]
```

Exercise 12.8.4 **

Write a function `squareSums :: (Num a) => a -> [(a, a, a)]` that uses a list comprehension to generate the first `n` triplets (a, b, c) that satisfy $a^2 + b^2 = c^2$.

Exercise 12.8.5 optional

Write a function called `approxSqrt eps x`, which approximates the square root of `x`, with precision `eps`, using the `iter` function.

1. First, write a function `nextSqrt x y`, which calculates the next approximation for $\sqrt{x}$, denoted y_n, by using x and the previous approximation y_{n-1} using the following formula:
$$y_n = nextSqrt(x, y_{n-1}) = (x/y_{n-1} + y_{n-1})/2$$
2. Write a function `approxS x`, which uses the `iter` and `nextSqrt` functions to return an infinite list of successive approximations of the square root of `x`. The starting value for `iter` should be 1 (in the formula above, y_0 would represent the starting value).
3. Write the `goodEnough eps (x, y)` function, which checks whether the absolute difference between `x` and `y` is less than `eps` (i.e. $|x - y| < \epsilon$). **The second parameter is a tuple for a good reason!**
4. Write the `genApprox eps x` function, which uses the `goodEnough` and `approxS` functions. It should take elements from the list generated by the `approxS` function, using the `closeEnough` function to decide when to stop (i.e. the difference between two successive elements is less than `eps`).
5. Write the `approxSqrt eps x` function, which returns the last approximation generated by the `genApprox` function.

Hints:
You might find the following functions useful:

- `snd` - Get the second element of a tuple
- `zip` - Return a list of tuples from 2 lists
- `takeWhile` and `dropWhile` - Take or drop elements until a predicate is satisfied (same meaning as in Elm)