Lecture 11. Lazy evaluation

Functional Programming 2018/19

Alejandro Serrano

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Goals

- Understand the lazy evaluation strategy
  - As opposed to strict evaluation
- Work with infinite structures
- Learn about laziness pitfalls
  - Force evaluation using `seq`

A simple expression

```haskell
square :: Integer -> Integer
square x = x * x

square (1 + 2)
= -- magic happens in the computer
9
```

How do we reach that final value?

Strict or eager or call-by-value evaluation

In most programming languages:

1. Evaluate the arguments completely
2. Evaluate the function call

```
square (1 + 2)
= -- evaluate arguments
square 3
= -- go into the function body
3 * 3
=
9
```

Non-strict or call-by-name evaluation

Arguments are replaced as-is in the function body

```
square (1 + 2)
= -- go into the function body
(1 + 2) * (1 + 2)
= -- we need the value of (1 + 2) to continue
3 * (1 + 2)
=
3 * 3
=
9
```

Does call-by-name make any sense?

In the case of `square`, non-strict evaluation is worse

Is this always the case?

Does call-by-name make any sense?

In the case of `square`, non-strict evaluation is worse

Is this always the case?

```
const x y = y  -- forget about x
```

```
-- Call-by-value
const (1 + 2) 5
=
const 3 5
=
5
```

```
-- Call-by-name
const (1 + 2) 5
=
5
```

Sharing expressions

```
square (1 + 2)
=
(1 + 2) * (1 + 2)
```

Why redo the work for `(1 + 2)`?

Sharing expressions

```
square (1 + 2)
=
(1 + 2) * (1 + 2)
```

Why redo the work for `(1 + 2)`?

We can share the evaluated result

```
square (1 + 2)
=
Δ * Δ
↑___↑___ (1 + 2)
           = 3
=
9
```

Lazy evaluation

Haskell uses a **lazy** evaluation strategy

- Expressions are not evaluated *until needed*
- Duplicate expressions are *shared*

Lazy evaluation never requires more steps than call-by-value

Each of those not-evaluated expressions is called a **thunk**

Does it matter?

Is it possible to get different outcomes using different evaluation strategies?

. . .

Yes and no

Does it matter? - Correctness and efficiency

The *Church-Rosser Theorem* states that for *terminating* programs the result of the computation does *not* depend on the evaluation strategy

But...

1. Performance might be different
   - As `square` and `const` show
2. This applies only if the program terminates
   - What about infinite loops?
   - What about exceptions?

Termination

```
loop x = loop x
```

- This is a well-typed program
- But `loop 3` never terminates

```
-- Eager                  -- Lazy
const (loop 3) 5          const (loop 3) 5
=                         =
const (loop 3) 5          5
=
...
```

Lazy evaluation terminates more often than eager

Build your own control structures

```
if_ :: Bool -> a -> a -> a
if_ True  t _ = t
if_ False _ e = e
```

- In eager languages, `if_` evaluates both branches
- In lazy languages, only the one being selected

For that reason,

- In eager languages, `if` has to be *built-in*
- In lazy languages, you can build your *own control structures*

Short-circuiting

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

- In eager languages, `x && y` evaluates both conditions
  - But if the first one fails, why bother?
  - C/Java/C# include a built-in *short-circuit* conjunction
- In Haskell, `x && y` only evaluates the second argument if the first one is `True`
  - `False && (loop True)` terminates

An infinite list of ones

```
ones :: [Integer]
ones = 1 : ones
```

`ones` is infinite, but everything works fine if we only work with a *finite* part

```
  take 2 ones
= take 2 (1 : ones)
= 1 : take 1 ones
= 1 : take 1 (1 : ones)
= 1 : 1 : take 0 ones
= 1 : 1 : []
```

A list of all natural numbers

To build an infinite list of numbers, we use recursion

- This kind of recursion is trickier than the usual one

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

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

"Until needed"

How does Haskell know *how much* to evaluate?

- By default, everything is kept in a thunk
- When we have a case distinction, we evaluate enough to distinguish which branch to follow

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

- If the number is `0` we do not need the list at all
- Otherwise, we need to distinguish `[]` from `x:xs`

Weak Head Normal Form

An expression is in **weak head normal form** (WHNF) if it is:

- A constructor with (possibly non-evaluated) data inside
  - `True` or `Just (1 + 2)`
- An anonymous function
  - The body might be in any form
  - `\x -> x + 1` or `\x -> if_ True x x`
- A built-in function applied to too few arguments

Every time we need to distinguish the branch to follow the expression is evaluated until its WHNF

A list of all Fibonacci numbers

Remember the usual definition of `fib`,

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

A list of all Fibonacci numbers

Remember the usual definition of `fib`,

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Here is a list containing all Fibonacci numbers:

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

fib :: Integer -> Integer
fib n = fibs !! n  -- Take the n-th element
```


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

A list of all Fibonacci numbers

```
  0 : 1  : ...
+ 1 : ...
----------------------------------
  1 : ...
```


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

A list of all Fibonacci numbers

```
  0 : 1 : 1  :  ...
+ 1 : 1 :  ...
-----------------------------
  1 : 2 :  ...
```

A list of all Fibonacci numbers

```
  0 : 1 : 1 : 2   : ...
+ 1 : 1 : 2 : ...
----------------------------
  1 : 2 : 3 : ...
```


Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Sieve of Erastosthenes

An algorithm to compute the list of all primes

- Already known in Ancient Greece

1. Lay all numbers in a list starting with 2
2. Take the first next number p in the list
3. Remove all the multiples of p from the list
   - $2p$, $3p$, $4p\ldots$
   - Alternatively, remove n if the remainder with p is 0
4. Go back to step 2 with the first remaining number

Sieve of Erastosthenes

1. Lay all numbers in a list starting with 2

   ```
   primes :: [Int]
   primes = sieve [2 .. ]  -- an infinite list
   ```

2. Take the first number p in the list

   ```
   sieve (p:ns) = ...
   ```

3. Remove n if the remainder with p is 0
4. Go back to step 2 with the first remaining number

   ```
   sieve (p:ns)
     = p : sieve [n | n <- ns, n `mod` p /= 0]
   ```


Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Strict versus lazy functions

Note the difference between these two functions

```
  loop 2 + 3
= -- definition of loop
  loop 2 + 3
= -- never-ending sequence
  ...

  const 3 (loop 2)
= -- definition of const
  3
  -- and that's it!
```

Strict versus lazy functions

A function is **strict** on one argument if the result of the function is non-terminating given a non-terminating value for that argument

- `(+)` is strict on its first and second arguments
- `const` is not strict on its second argument, but strict on the first

We represent non-termination by $\perp$ or `undefined`

- We also call $\perp$ a *diverging* computation
- f is strict if $f\ \perp = \perp$

Some (tricky) questions

What is the result of these expressions?

1. `(\x -> x) True`
2. `(\x -> x) undefined`
3. `(\x -> 0) undefined`
4. `(\x -> undefined) 0`
5. `(\x f -> f x) undefined`
6. `undefined undefined`
7. `length (map undefined [1,2])`

Some (tricky) questions

What is the result of these expressions?

1. `(\x -> x) True              = True`
2. `(\x -> x) undefined         = undefined`
3. `(\x -> 0) undefined         = 0`
4. `(\x -> undefined) 0         = undefined`
5. `(\x f -> f x) undefined     = \f -> f undefined`
6. `undefined undefined         = undefined`
7. `length (map undefined [1,2]) = 2`

Garbage collection

- Thunks are managed by the run-time system
  - They are created when you need a value
  - But are not reclaimed right after evaluation
- Haskell uses **garbage collection** (GC)
  - Every now and them Haskell takes back all the memory used by thunks which are not needed anymore
  - *Pro*: we do not need to care about memory
  - *Pro*: GC enables fancy distributed algorithms
  - *Con*: GC takes time, so lags can occur
- Most modern languages nowadays use GC
  - Java, Scala, C#, Ruby, Python…
  - Swift uses Automatic Reference Counting (ARC)

Performance

Lists are sloooooooow

You have to traverse them for almost every operation

- To find an element you need to compare all the ones in front of it
- To append two lists you traverse the first one

We have seen some techniques that help

- Use an accumulator for reversing a list
- Use a search tree to find things quickly

Still, lists are convenient for small data sets

Use better data structures

`containers` contains many general purpose data structures

- `Map k v` hold values `v` indexed by keys of type `k`
- `Set a` holds values of `a` without repetition
- `Seq a` is similar to a list, but more efficient
  - Match and build from both sides
  - Useful as a queue or as a stack
- `Tree a` implement rose trees

The interface is almost *identical* to that of lists

Use better data structures

`String` is just a synonym for `[Char]`

- Simple to handle, very inefficient

If your application uses a lot of them, you should consider

- `ByteString` to treat it as an array of words
- `Text` to represent Unicode strings

Space leaks

Space leak = data structure which grows bigger, or lives longer than expected

- More memory in use means more GC
- As a result, performance decreases

The most common source of space leaks are thunks

- Thunks are essential for lazy evaluation
- But they also take some amount of memory

Case study: `foldl'`

From long, long time ago...

```
foldl _ v []     = v
foldl f v (x:xs) = foldl f (f v x) xs
```

```
  foldl (+) 0 [1,2,3]
= foldl (+) (0 + 1) [2,3]
= foldl (+) ((0 + 1) + 2) [3]
= foldl (+) (((0 + 1) + 2) + 3) []
= ((0 + 1) + 2) + 3
```

Case study: `foldl'`

```
  foldl (+) 0 [1,2,3]
= ((0 + 1) + 2) + 3
```

- Each of the additions is kept in a thunk
  - Some memory need to be reserved
  - They have to be GC'ed after use

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Case study: `foldl'`

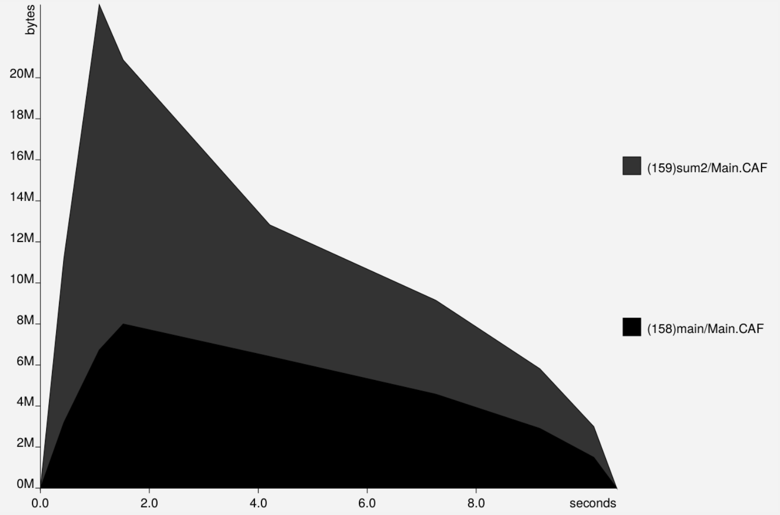


Figure 1:

Case study: `foldl'`

Just performing the addition is faster!

- Computers are fast at arithmetic
- We want to *force* additions before going on

```
  foldl (+) 0 [1,2,3]
= foldl (+) (0 + 1) [2,3]
= foldl (+) 1 [2,3]
= foldl (+) (1 + 2) [3]
= foldl (+) 3 [3]
= foldl (+) (3 + 3) []
= foldl (+) 6 []
= 6
```

Forcing evaluation

Haskell has a primitive operation to force

```
seq :: a -> b -> b
```

A call of the form `seq x y`

- First evaluates `x` up to WHNF
- Then it proceeds normally to compute `y`

Usually, `y` depends on `x` somehow

Universiteit Utrecht

[Faculty of Science
Information and Computing
Sciences]

Case study: `foldl'`

We can write a new version of `foldl` which forces the accumulated value before recursion is unfolded

```
foldl' _ v []     = v
foldl' f v (x:xs) = let z = f v x
                     in z `seq` foldl' f z xs
```

This version solves the problem with addition

Case study: `foldl'`

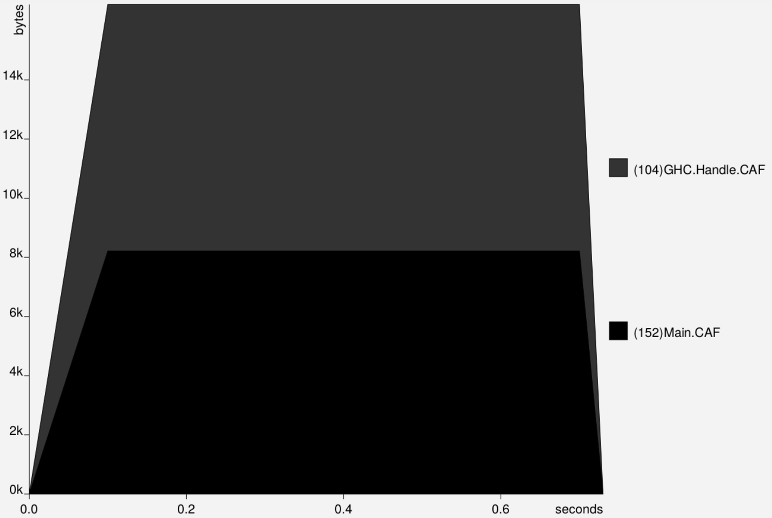


Figure 2:

Strict application

Most of the times we use `seq` to force an argument to a function, that is, *strict application*

```
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

Because of sharing, `x` is evaluated only once

```
foldl' _ v []     = v
foldl' f v (x:xs) = ((foldl' f) $! (f v x)) xs
```

More (tricky) questions

What is the result of these expressions?

1. `(\x -> 0) $! undefined`
2. `seq (undefined, undefined) 0`
3. `snd $! (undefined, undefined)`
4. `(\x -> 0) $! (\x -> undefined)`
5. `undefined $! undefined`
6. `length $! map undefined [1,2]`
7. `seq (undefined + undefined) 0`
8. `seq (foldr undefined undefined) 0`
9. `seq (1 : undefined) 0`

More (tricky) questions

What is the result of these expressions?

1. `(\x -> 0) $! undefined = undefined`
2. `seq (undefined, undefined) 0 = 0`
3. `snd $! (undefined, undefined) = undefined`
4. `(\x -> 0) $! (\x -> undefined) = 0`
5. `undefined $! undefined = undefined`
6. `length $! map undefined [1,2] = 2`
7. `seq (undefined + undefined) 0 = undefined`
8. `seq (foldr undefined undefined) 0 = 0`
9. `seq (1 : undefined) 0 = 0`

`seq` only evaluates up to WHNF

Universiteit Utrecht

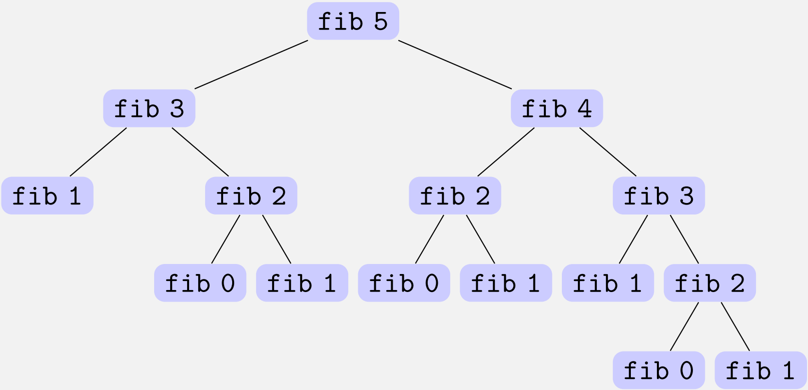
[Faculty of Science
Information and Computing
Sciences]

Case study: Fibonacci numbers

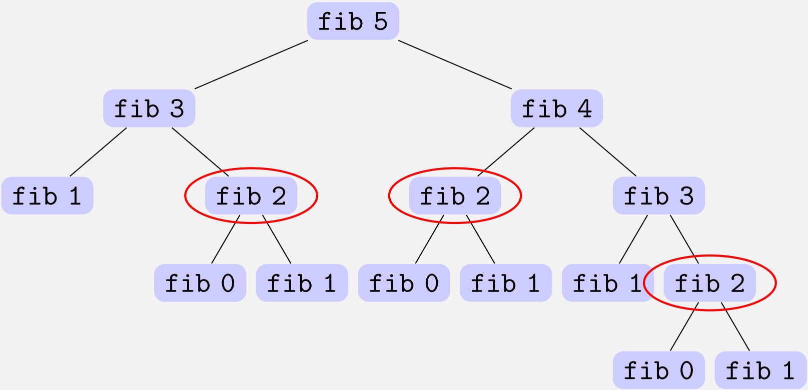
```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

What happens when we ask for `fib 5`?

Case study: Fibonacci numbers



Case study: Fibonacci numbers



Local memoization

Idea: remember the result for function calls

- We build a list of partial results
- Sharing takes care of evaluating only once

```
memo_fib n = map fib [0 .. ] !! n
  where fib 0 = 0
        fib 1 = 1
        fib n = memo_fib (n-1) + memo_fib (n-2)
```

You can get even faster by using a better data structure

- For example, `IntMap` from `containers`

Summary

- Laziness = evaluate only as much as needed
  - As opposed to the more common *eager* evaluation
- Evaluation is guided by pattern matching
  - We need WHNF to choose a branch
  - Some arguments may not even be evaluated
- Laziness is tricky when it fails
  - Too many thunks lead to a space leak
  - `seq` is used to *force* evaluation