Laziness

Advanced functional programming summer school - Lecture 9

Gabriele Keller (& Trevor McDonell, Wouter Swierstra)

Utrecht University

Laziness

A simple expression

```
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

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 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
```

Dopes non-strict evaluation 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
```

```
-- strict
const (1 + 2) 5
=
const 3 5
=
5
```

```
– non-strict
const (1 + 2) 5
=
5
```

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)
=
* 
```

```
(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)
=
  *            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 eager evaluation

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

Lazy data-structures

Infinite lists:

```
let xs = 1 : xs
in take 10 xs
```

```
let ys = [3,6..]
in take 20 ys
```

```
let primes = …
in primes !! 23453
```

Allow to separate generators from filters

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 structure

```
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

```
(&&) :: 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

"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

Case study: `foldl`

```
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'`

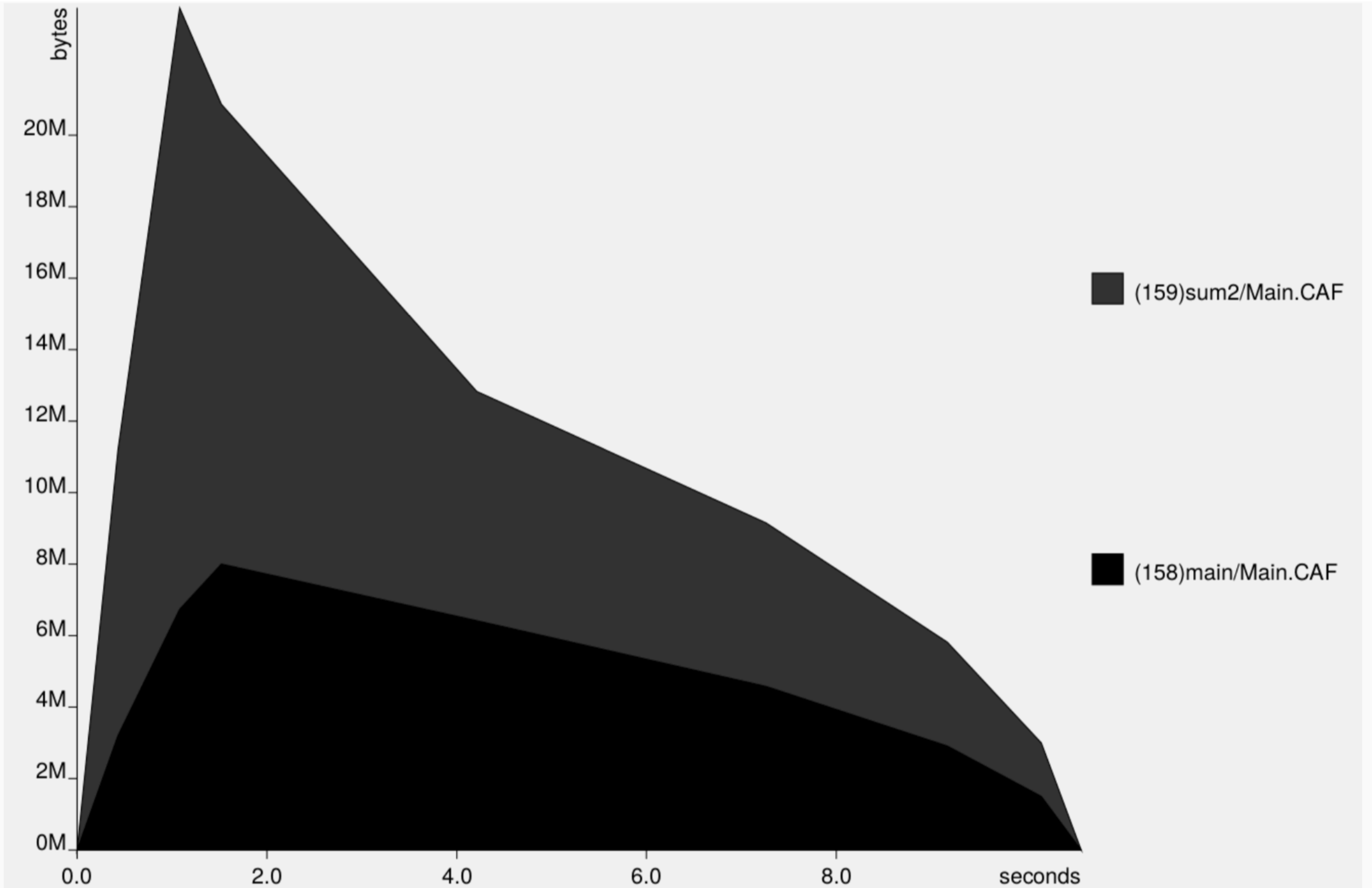
```
 foldl (+) 0 [1,2,3]
= ((0 + 1) + 2) + 3
```

Each of the additions is kept in a thunk

- Some memory need to be reserved

- They have to be GC'ed after use

Case study: `foldl'`



Case study: `foldl'`

Just performing the addition is faster!

Computers are fast at arithmetic

```
  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

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

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

Profiling

Something about (in)efficiency

We have seen that Haskell programs:

- can be very short
- and sometimes very inefficient

Question:

How to find out where time is spent

Answer:

Use profiling!

Laziness is a double-edged sword

With laziness, we are sure that things are evaluated only as much as needed to get the result.

But, being lazy means holding lots of thunks in memory:

- Memory consumption can grow quickly.
- Performance is not uniformly distributed.

Question:

How to find out where memory is spent?

How to find out where to sprinkle `seq`

Answer:

Use profiling

Example: `segs`

`segs xs` computes all the consecutive sublists of `xs`.

```
segs []     = [[]]
segs (x:xs) = segs xs ++ map (x:) (inits xs)
> segs [2,3,4]
[[],[4],[3],[3,4],[2],[2, 3],[2,3,4]]
```

This implementation is extremely inefficient.

Example: `segsinits`

We can compute `inits` and `segs` at the same time.

```
segsinits []       =  ([[]], [[]])
segsinits (x:xs)   =
  let (segsxs, initsxs)  = segsinits xs
      newinits           = map (x:) initsxs
  in (segsxs ++ newinits, [] : newinits)
segs = fst . segsinits
```