Advanced Functional Programming

03 - Laziness

Wouter Swierstra & Trevor L. McDonell

Utrecht University

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 = x  -- forget about y
```

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

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

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?
   - What about programs run out of memory and crash?

Termination

```
loop x = loop x
```

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

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

Lazy evaluation terminates more often than eager

Build your own control structures

```haskell
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

“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: `length` and `take`

Given the following definitions:

```
take 0 xs = []
take n xs = head xs : take (n - 1) (tail xs)

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

What is the result of evaluating `length (take 3 undefined)`?

Case study: `length` and `take`

Given the following definitions:

```
take 0 xs = []
take n xs = head xs : take (n - 1) (tail xs)

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

What is the result of evaluating `length (take 3 undefined)`?

Somewhat surprisingly – this expression evaluates to `3`!

Why?

```
length (take 3 undefined)
length (head undefined : take 2 (tail undefined))
1 + length (take 2 (tail undefined))
1 + length (head (tail undefined) : take 1 (tail (tail undefined)))
1 + 1 + length (take 1 (tail (tail undefined)))
1 + 1 + length (head (tail (tail undefined)) : take 0 (tail (tail (tail undefined
1 + 1 + 1 + length (take 0 (tail (tail (tail undefined))))
1 + 1 + 1 + length []
1 + 1 + 1 + 0
1 + 1 + 1
1 + 2
3
```

Case study: Sieve of Eratosthenes

Idea: compute the list of all prime numbers by ‘crossing out’ all the multiples of 2. The next prime number must be 3. Cross out the multiples of 3. The next prime number must be 5. Repeat…

Case study: Sieve of Eratosthenes

Idea: compute the list of all prime numbers by 'crossing out' all the multiples of 2. The next prime number must be 3. Cross out the multiples of 3. The next prime number must be 5. Repeat...

In Haskell we write this in three simple steps:

1. Remove the multiples of a given number:

```
removeMultiples n xs = filter ((/=) 0) . (`mod` n)) xs
```

Case study: Sieve of Eratosthenes

Idea: compute the list of all prime numbers by 'crossing out' all the multiples of 2. The next prime number must be 3. Cross out the multiples of 3. The next prime number must be 5. Repeat...

In Haskell we write this in three simple steps:

1. Remove the multiples of a given number:

```
removeMultiples n xs = filter ((/=) 0) . (`mod` n)) xs
```

2. Define a prime as any number that passes the sieve:

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

Case study: Sieve of Eratosthenes

Idea: compute the list of all prime numbers by 'crossing out' all the multiples of 2. The next prime number must be 3. Cross out the multiples of 3. The next prime number must be 5. Repeat...

In Haskell we write this in three simple steps:

1. Remove the multiples of a given number:

```
removeMultiples n xs = filter ((/=) 0) . (`mod` n)) xs
```

2. Define a prime as any number that passes the sieve:

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

3. Define the primes:

```
primes = sieve [2..]
```

Case study: `foldl'`

From long, long time ago...

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

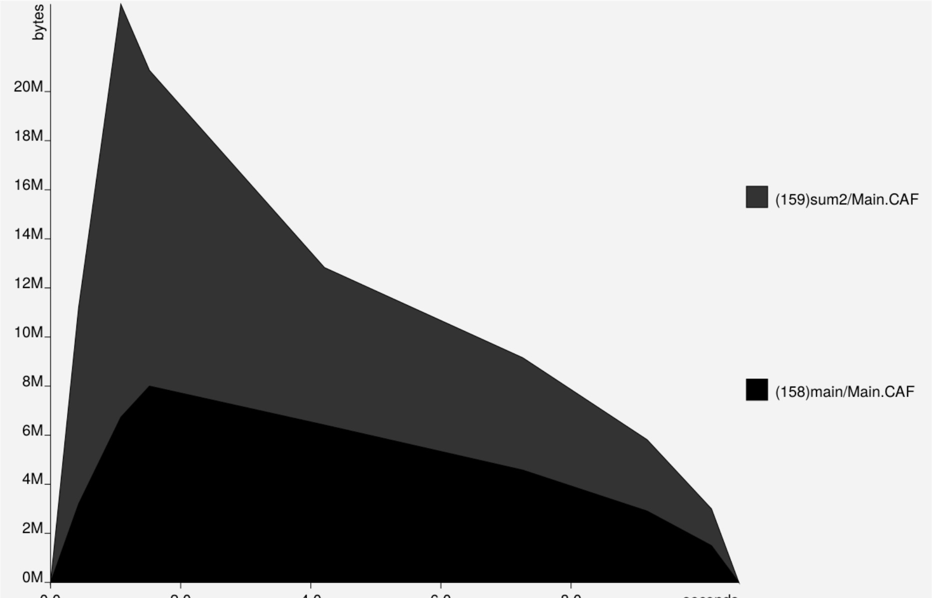
```
  foldl (+) 0 [1,2,3]
= foldl (+) (0 + 1) [2,3]
= foldl (+) ((0 + 1) + 2) [3]
= foldl (+) (((0 + 1) + 2) + 3) []
= ((0 + 1) + 2) + 3
```

Case study: `foldl'`

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

- Each of the additions is kept in a thunk
  - Some memory needs to be reserved
  - This has to be GC'ed after use

Case study: `foldl'`



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 evaluation

```
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

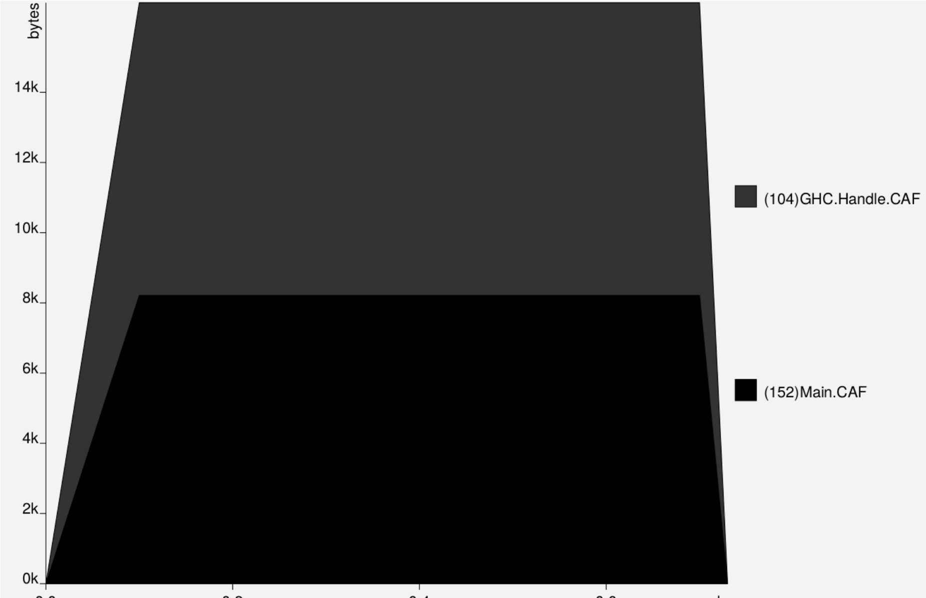
Case study: `foldl'`

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

Case study: `foldl'`



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
```

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?

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`s?

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`s?

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]]
```

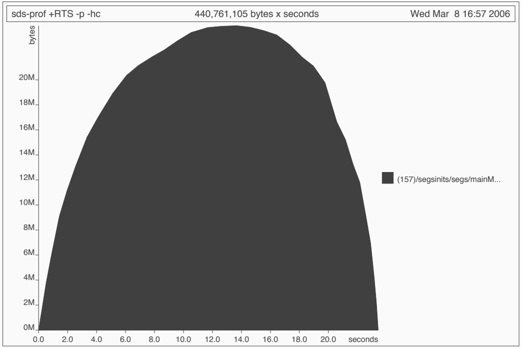
This implementation is extremely inefficient.

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
```

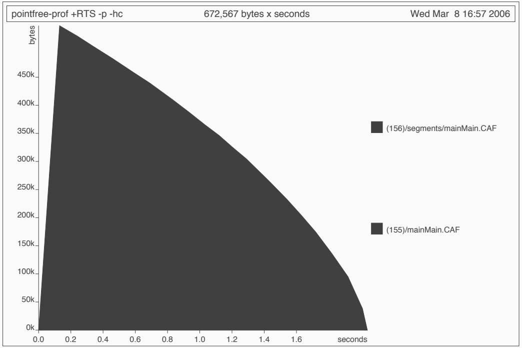
Heap profile for `segsinits`



Example: `pointfree`

```
pointfree =
  let p    = not . null
      next = filter p . map tail . filter p
  in concat . takeWhile p . iterate next . inits
```

Heap profile for `pointfree`

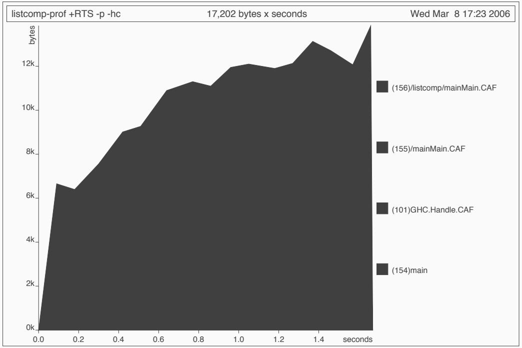


Example: `listcomp`

`segs` are just the `tails` of the `inits`!

```
listcomp xs =
  [] : [ t | i <- inits xs
           , t <- tails i
           , not (null t) ]
main =
  print (length (concat
    (listcomp [1 :: Int .. 300])))
```

Heap profile for `listcomp`



How to produce these?

```
prompt> ghc -prof -fprof-auto -o listcomp-prof -O2 Segments.hs
prompt> ./listcomp-prof +RTS -hc -p
4545100
prompt> hp2ps listcomp-prof.hp
```

Semantics

The idea behind lazy evaluation stems back at least as far as 1976, when Henderson and Morris published their paper 'A lazy evaluator'.

This paper describes an implementation of LISP, using pointers, to lazily share intermediate results when possible.

But what are the exact semantics?

Semantics

The idea behind lazy evaluation stems back at least as far as 1976, when Henderson and Morris published their paper ‘A lazy evaluator’.

This paper describes an implementation of LISP, using pointers, to lazily share intermediate results when possible.

But what are the exact semantics?

Defining such semantics was surprisingly hard!

It took until 2000 until there was a satisfactory operational semantics for lazy evaluation.

A natural semantics for lazy evaluation

```
e ::= x                    (variables)
    | e x                  (application)
    | λ x -> e             (abstraction)
    | let x = e in e'      (let bindings)
```

Note that we only ever apply *expressions* to variables!

We may need to rewrite arbitrary programs into this form, where any non-variable argument is let-bound.

A natural semantics for lazy evaluation

$$\frac{}{\Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e}\textsc{Lam} \qquad \frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \qquad \Delta : e'[x/y] \Downarrow \Theta : v}{\Gamma : e\ x \Downarrow \Theta : v}\textsc{App}$$

$$\frac{\Gamma : e \Downarrow \Delta : v}{\Gamma, x \mapsto e : x \Downarrow \Delta, x \mapsto v : v}\textsc{Var} \qquad \frac{\Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n : e \Downarrow \Delta : v}{\Gamma : \mathsf{let}\ x_1 = e_1, \ldots, x_n = e_n\ \mathsf{in}\ e \Downarrow \Delta : v}\textsc{Let}$$

Figure 1: Semantics

References

- Real World Haskell has a chapter on profiling:

`https://book.realworldhaskell.org/read/profiling-and-optimization.html`

- A natural semantics for lazy evaluation, John Launchbury
- The flame war between Bob Harper and Lennart Augustsson is both amusing and insightful:

`https://augustss.blogspot.com/2011/05/more-points-for-lazy-evaluation-in.html`

- More recently – Hackett & Hutton, *Call-by-Need Is Clairvoyant Call-by-Value*, ICFP 2019