Advanced Functional Programming

Laziness

Lawrence Chonavel (slides also by 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
```

A simple expression

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

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

How do we reach that final value?

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?

How to make it fast?

How evaluation works

‘Strict’ aka ‘eager’ aka ‘call-by-value’ evaluation

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

‘Strict’ aka ‘eager’ aka ‘call-by-value’ evaluation

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

Most programming languages (e.g. OCaml, C, …):

‘Strict’ aka ‘eager’ aka ‘call-by-value’ evaluation

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

Most programming languages (e.g. OCaml, C, ...):

Haskell does **not** do this! Haskell was invented to experiment with *non*-strict evaluation.

‘call-by-name’ evaluation

```
square (1 + 2)
= -- copy args as-is into the function body
(1 + 2) * (1 + 2)
= -- to continue evaluting *, need value of its 1st arg
3 * (1 + 2)
= -- to continue evaluting *, need value of its 2nd arg
3 * 3
=
9
```

'call-by-name' evaluation

```
square (1 + 2)
= -- copy args as-is into the function body
(1 + 2) * (1 + 2)
= -- to continue evaluting *, need value of its 1st arg
3 * (1 + 2)
= -- to continue evaluting *, need value of its 2nd arg
3 * 3
=
9
```

e.g. LaTeX, bash, old LISPs.

‘call-by-name’ evaluation

```
square (1 + 2)
= -- copy args as-is into the function body
(1 + 2) * (1 + 2)
= -- to continue evaluting *, need value of its 1st arg
3 * (1 + 2)
= -- to continue evaluting *, need value of its 2nd arg
3 * 3
=
9
```

e.g. LaTeX, bash, old LISPs.

Haskell does **not** do this either! Too inefficient.

… but CBN faster than CBV

```
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
```

… but CBN faster than CBV

```
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
```

Best of both?

‘call-by-need’ aka ‘lazy’ aka ‘normal order’ aka ‘left outermost’ evaluation

```
square (1 + 2)
= -- go into function body
Δ * Δ
□___□___ (1 + 2)
          =
          3
=
9
```

‘call-by-need’ aka ‘lazy’ aka ‘normal order’ aka ‘left outermost’ evaluation

```
square (1 + 2)
= -- go into function body
Δ * Δ
▯___▯___ (1 + 2)
          =
          3
=
9
```

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

‘call-by-need’ aka ‘lazy’ aka ‘normal order’ aka ‘left outermost’ evaluation

```
square (1 + 2)
= -- go into function body
Δ * Δ
□___□___ (1 + 2)
          =
          3
=
9
```

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

Haskell does this

Does it matter?

Results are always the same

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

Results are always the same

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

But:

1. Performance might be different
2. (Sub-) programs do not always terminate
    - e.g. infinite loops?
    - e.g. exceptions?
    - e.g. programs that run out of memory and crash?

Non-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

```
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*

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*

- In Agda, `if_then_else_` is a library function
- In Haskell, `&&` is a library function

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

Programming with laziness

How is this evaluated?

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

main = print $ take 4 map $ capitalize "abcdefg"
```

How is this evaluated?

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

main = print $ take 4 map $ capitalize "abcdefg"

  = print $ case (4, capitalize "abcdefg") of
              (0, _)  -> []
              (_, []) -> []
              (n, (x:xs)) -> x : take (n-1) xs
```

How is this evaluated?

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

main = print $ take 4 map $ capitalize "abcdefg"

  = print $ case (4, capitalize "abcdefg") of
              (0, _)  -> []
              (_, []) -> []
              (n, (x:xs)) -> x : take (n-1) xs
```

To decide which case to take, need to know more about `capitalize "abcdefg"`

How is this evaluated?

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

main = print $ take 4 map $ capitalize "abcdefg"

  = print $ case (4, capitalize "abcdefg") of
              (0, _)  -> []
              (_, []) -> []
              (n, (x:xs)) -> x : take (n-1) xs
```

To decide which case to take, need to know more about `capitalize "abcdefg"`

But *how much* more?

Evaluating “just enough”

Evaluate just enough to make progress

```
print $ case (4, capitalize "abcdefg") of
                  (0, _)  -> []
                  (_, []) -> []
                  (n, (x:xs)) -> x : take (n-1) xs
```

Evaluating "just enough"

Evaluate just enough to make progress

```
print $ case (4, capitalize "abcdefg") of
                  (0, _)  -> []
                  (_, []) -> []
                  (n, (x:xs)) -> x : take (n-1) xs

=

print $ case (4, 'A':capitalize "bcdefg") of
                  (0, _)  -> []
                  (_, []) -> []
                  (n, (x:xs)) -> x : take (n-1) xs
```

Evaluating “just enough”

Evaluate just enough to make progress

```
print $ case (4, capitalize "abcdefg") of
                (0, _)  -> []
                (_, []) -> []
                (n, (x:xs)) -> x : take (n-1) xs

=
print $ case (4, 'A':capitalize "bcdefg") of
                (0, _)  -> []
                (_, []) -> []
                (n, (x:xs)) -> x : take (n-1) xs

=
print $ let n = 4; x = 'A' ; xs = capitalize "bcdefg"
        in x : take (n-1) xs
```

How much is “just enough”?

- Outermost constructor
    - e.g. `True`
    - e.g. `Just (1 + 2)`

How much is “just enough”?

- Outermost constructor
  - e.g. `True`
  - e.g. `Just (1 + 2)`
- “Head Normal Form”

How much is “just enough”?

- Outermost constructor
  - e.g. `True`
  - e.g. `Just (1 + 2)`
- “Head Normal Form”
- Outermost constructor *or lambda-expression*
  - e.g. `True`
  - e.g. `Just (1 + 2)`
  - e.g. `\x -> x + 1`
  - e.g. `\x -> if_ True x x`

How much is “just enough”?

- Outermost constructor
  - e.g. `True`
  - e.g. `Just (1 + 2)`
- “Head Normal Form”
- Outermost constructor *or lambda-expression*
  - e.g. `True`
  - e.g. `Just (1 + 2)`
  - e.g. `\x -> x + 1`
  - e.g. `\x -> if_ True x x`
- “Weak Head Normal Form”

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 : seive (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 : seive (removeMultiples p ns)
```

3. Define the primes:

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

From long, long time ago...

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

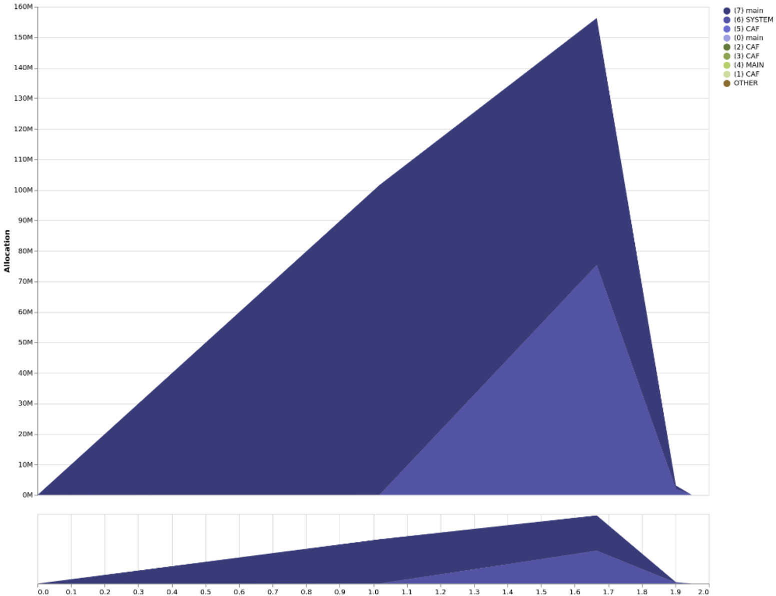
```
  foldl (+) 0 [1,2,3]
= foldl (+) (0 + 1) [2,3]
= foldl (+) ((0 + 1) + 2) [3]
= foldl (+) (((0 + 1) + 2) + 3) []
= ((0 + 1) + 2) + 3
```

Case study: `foldl'`

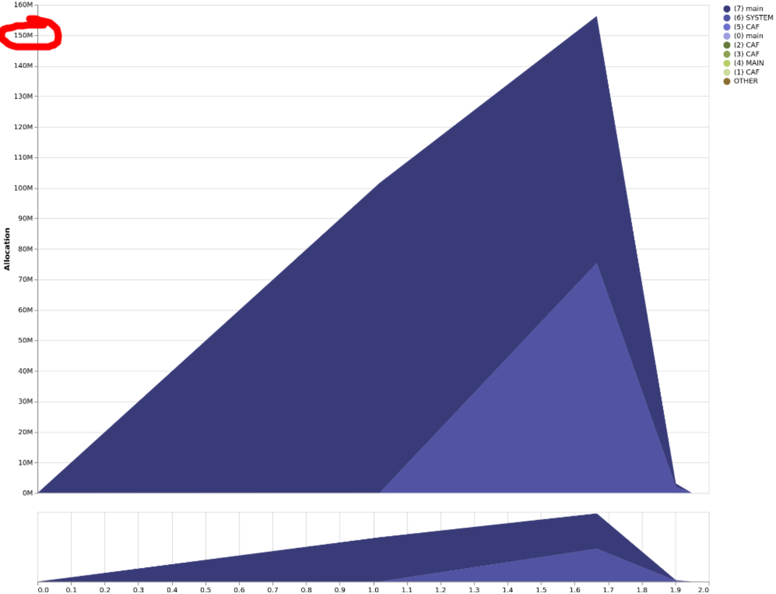
```
  foldl (+) 0 [1,2,3]
= ((0 + 1) + 2) + 3
```

- 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'`



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
```

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
```

Where the seq primitive encourages the GHC optimizer to evaluate z before `foldl' f z xs`.

This version solves the problem with addition

Case study: `foldl'`

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

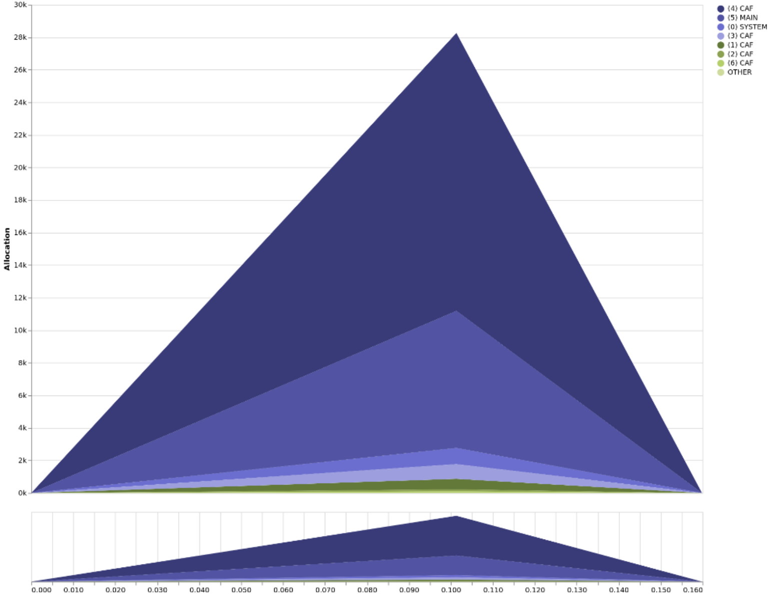
```
foldl' _ v []     = v
foldl' f v (x:xs) = let z = f v x
                    in  z `seq` foldl' f z xs
```

Where the seq primitive encourages the GHC optimizer to evaluate `z` before `foldl' f z xs`.

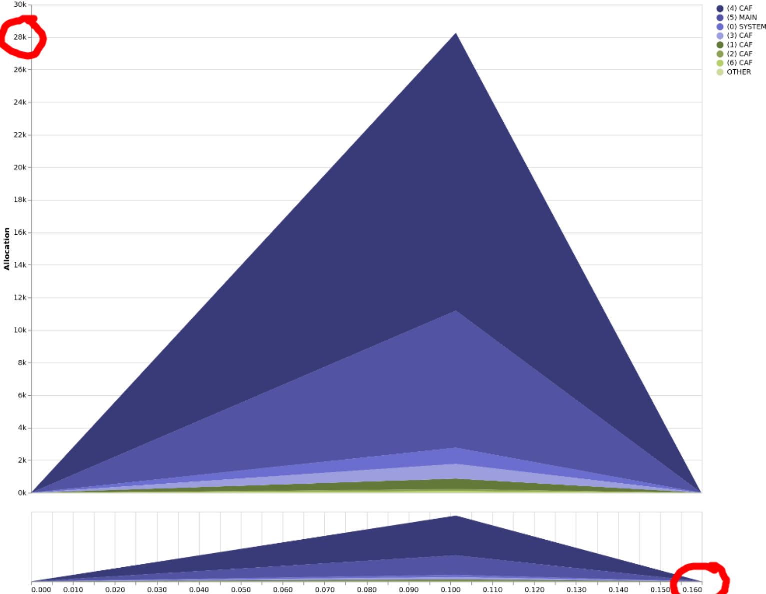
This version solves the problem with addition

N.B. exactly what seq does is too complicated for this course. Read the full story at https://stackoverflow.com/a/66965677

Case study: `foldl'`



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
```

Profiling

How did I generate those graphs?

```haskell
-- tmp.hs
main = print $ foldl' (+) 0 [1..2_000_000]
```

```
$ ghc tmp.hs -prof -fprof-auto
[1 of 2] Compiling Main             ( tmp.hs, tmp.o ) [Flags changed]
[2 of 2] Linking tmp [Objects changed]
$ ./tmp +RTS -l -hc
2000001000000
$ eventlog2html tmp.eventlog
```

Compile-time options

```
$ ghc tmp.hs -prof -fprof-auto
```

`tmp.hs` | Name of file to compile |

`-prof` | Instrument binary with profiling information |

`-fprof-auto` | Mark all functions as billable ‘cost centers’ |

Manual at http://downloads.haskell.org/ghc/latest/docs/users_guide/profiling.html

Run-time options

```
$ ./tmp +RTS -l -hc
```

`+RTS` | Pass following options to the Haskell run-time system (rather than `main`) |

`-hc` | Generate ***h**eap profile*, labelled by ***c**ost centre* |

`-l` | Emit *event log* (needed by the eventlog2html tool) |

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`

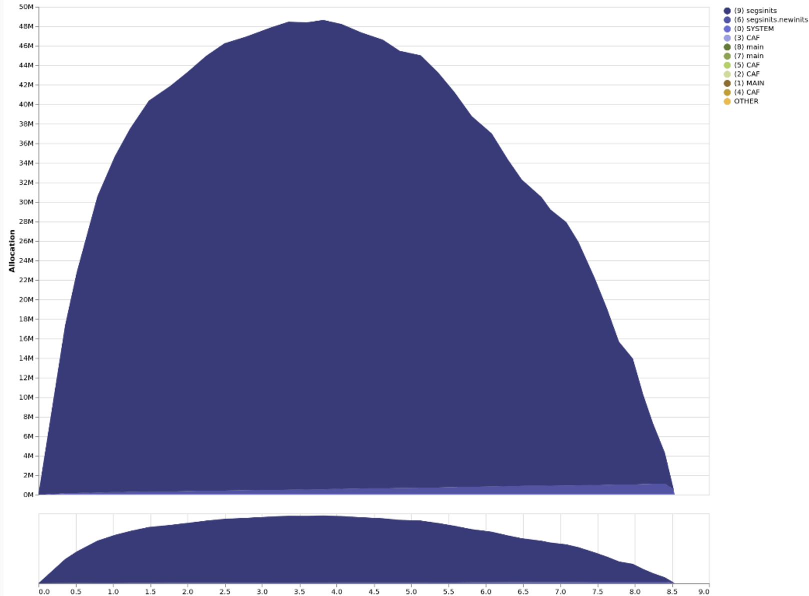
We can compute `inits` and `segs` at the same time.

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



main = print $ segs [1..300]
```

Heap profile for `segsinit`

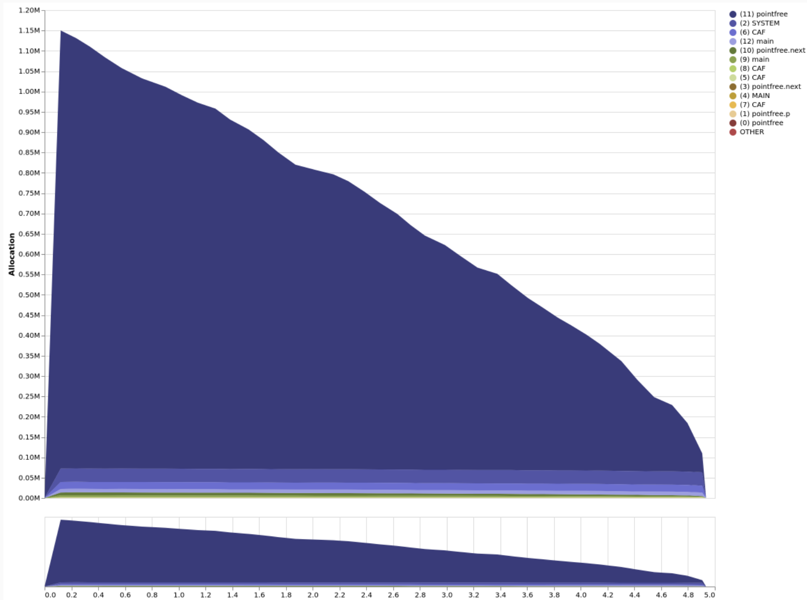


Example: `pointfree`

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

main = print $ pointfree [1..300]
```

Heap profile for `pointfree`



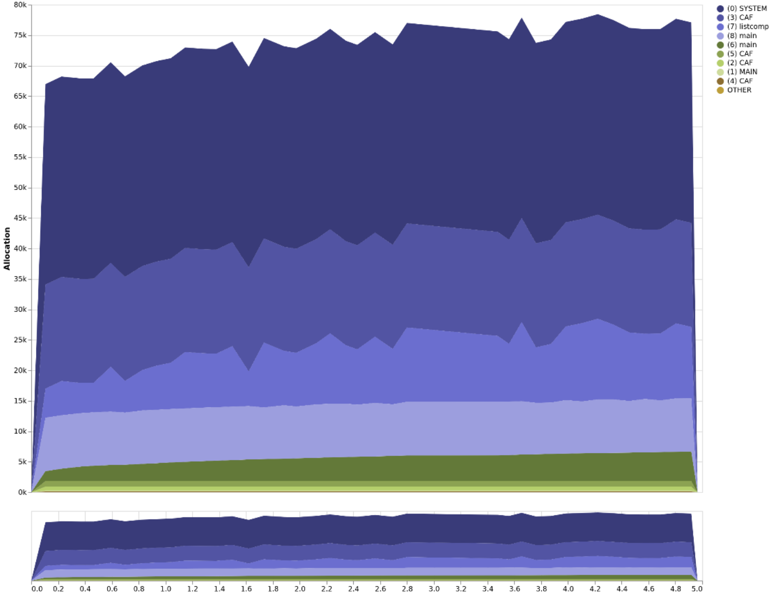
Example: `listcomp`

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

```
listcomp xs =
  [] : [ t | i <- inits xs
           , t <- tails i
           , not (null t) ]

main = print $ listcomp [1..300]
```

Heap profile for `listcomp`



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:

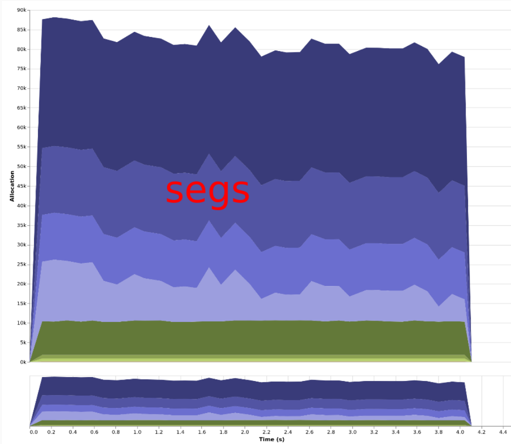
How to find out where memory is spent?

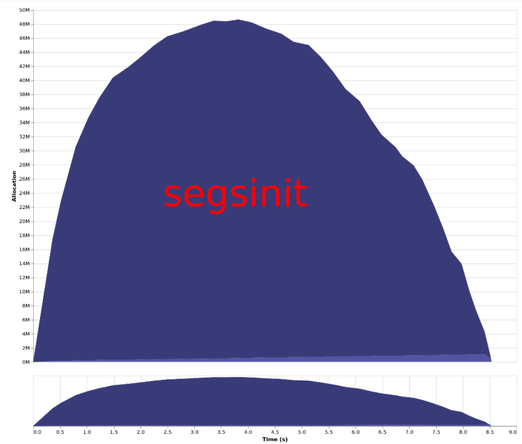
How to find out where to sprinkle seqs?

Answer:

Use profiling

Beware premature optimization





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

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

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

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