Laziness

Lecture 8 of CSE 3100
Functional Programming

Jesper Cockx

Q3 2023-2024

Technical University Delft

[*TODO*: insert joke about laziness.]

Lecture plan

- Lazy evaluation
- Forcing strictness
- Infinite data structures
- Case study: computing primes

Lazy evaluation

Evaluation strategies

An evaluation strategy gives a general way to pick a which subexpression to evaluate next.

- Call-by-value reduction: evaluate arguments before unfolding the definition of a function
- Call-by-name reduction: unfold function definition without evaluating arguments

Note. There are many other evaluation strategies 'in between' these two extremes.

Side note: innermost and outermost reduction

In Haskell, a lambda expression is a black box: its body will never be evaluated before it is applied.

Evaluation strategies that do evaluate under lambdas:

- Innermost reduction is call-by-value with evaluation under lambdas.
- Outermost reduction is call-by-name with evaluation under lambdas.

Evaluating `map`

Question. How is `head (map (1+) [1,2,3])` evaluated under call-by-value and call-by-name?

Evaluating `map`

Call-by-value

```
    head (map (1+) (1:2:3:[]))
--> head ((1+1):map (+1) (2:3:[]))
--> head (2:map (+1) (2:3:[]))
--> head (2:(2+1):map (+1) (3:[]))
--> head (2:3:map (+1) (3:[]))
--> head (2:3:(3+1):map (+1) ([]))
--> head (2:3:4:map (+1) ([]))
--> head (2:3:4:[])
--> 2
```

Evaluating `map`

Call-by-name

```
    head (map (1+) (1:2:3:[]))
--> head ((1+1):map (1+) (2:3:[]))
--> 1+1
--> 2
```

Two definitions of `fac`

```
fac 0 = 1
fac n = n * fac (n - 1)

fac' n = acc 1 n
  where
    acc x 0 = x
    acc x y = acc (x*y) (y-1)
```

Question. How are `fac 3` and `fac' 3` evaluated under call-by-name and call-by-value? Which one is more efficient?

Non-terminating programs

Some programs will go into an infinite loop with any evaluation strategy.

```
inf :: Integer
inf = 1 + inf

    inf
--> 1 + inf
--> 1 + 1 + inf
--> 1 + 1 + 1 + inf
--> ...
```

Non-terminating programs

Other programs will go into an infinite loop with call-by-value, but not with call-by-name:

```
-- with call-by-value
    fst (0, inf)
--> fst (0, 1 + inf)
--> fst (0, 1 + 1 + inf)
--> ...

-- with call-by-name
    fst (0, inf)
--> 0
```

Avoiding useless work

For functions that don't (always) use their arguments, *call-by-value* will do useless work:

```
     const 42 (2+3)
 --> const 42 5 --> 42
```

For functions that use their arguments more than once, *call-by-name* will do useless work:

```
     square (2+3)
 --> (2+3) * (2+3)
 --> 5 * (2+3) --> 5 * 5 --> 25
```

Can we get the best of both worlds? Yes!

Lazy evaluation

Lazy evaluation (aka *call-by-need*) is a variant of call-by-name that avoids double evaluation.

Each function argument is turned into a thunk:

- The first time the argument is used, the thunk is evaluated and the result is stored in the thunk.
- The next time the value stored in the thunk is used.

Lazy evaluation in a nutshell

Under lazy evaluation, programs are evaluated at most once and only as far as needed.

Lazy evaluation in other languages

Haskell is a lazy language: all evaluation is lazy by default.

Most other languages are eager (aka *strict*), but still have some form of lazy evaluation:

- *Lazy 'and'/'or'* (almost all languages): `False && b` evaluates to `False` without evaluating `b`.
- *Iterators* (e.g. Java) can produce values on-demand.
- *Generator functions* (e.g. Python) can use `yield` to lazily return values.
- *`lazy val`* (in Scala) declares a value that is computed lazily.

Advantages of lazy evaluation

- It never evaluates unused arguments.
- It always terminates if possible.
- It takes the smallest number of steps of all strategies.
- It enables use of infinite data structures.

Pitfalls of lazy evaluation

- Creation and management of thunks has some runtime overhead.
- It is hard to predict the order of evaluation.[1]
- Big intermediate expressions sometimes lead to a drastic increase in memory usage.

[1] Usually not a problem, unless you use `unsafePerformIO`.

Forcing strict evaluation

Performance drawbacks of lazy evaluation

The number of steps is not the only thing that matters for performance: the size of intermediate terms is also important:

- For small expressions that evaluate to a large data structure, call-by-need is better

  ```
  (replicate 100000000 "spam") !! 5
  ```

- For big expressions that evaluate to a small value, call-by-value is better

  ```
  foldl (+) 0 [1..100000000]
  ```

Summing a long list

Let's take a closer look:

```
    foldl (+) 0 [1..100000000]
--> foldl (+) (0+1) [2..100000000]
--> foldl (+) ((0+1)+2) [3..100000000]
--> foldl (+) (((0+1)+2)+3) [4..1000000
--> ...
```

What happens if you try to evaluate `foldl (+) 0 [1..100000000]` in GHCi?

The problem with large intermediate expressions

Recursive functions (like `foldl`) can create large intermediate expressions during evaluation, which is bad for performance:

- Each intermediate expression requires a new thunk to be allocated.
- Too large intermediate expressions cause stack overflows.

Maybe being a *little* less lazy would help?

Forcing strict evaluation

Haskell provides a built-in function `seq`:

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

The expression `seq u v` will evaluate `u` before returning `v`.

```
(1+2) `seq` 5 --> 3 `seq` 5 --> 5

replicate 5 'c' `seq` 42
--> 'c':(replicate 4 'c') `seq` 42
--> 42
```

Strict application

Using `seq`, we can define strict application:

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

"Please evaluate `x` before applying `f`!"

```
    square $! (1+2)       [          ]
--> x `seq` square x      [ x := 1+2 ]
--> x `seq` square x      [ x := 3   ]
--> square x              [ x := 3   ]
--> x*x                   [ x := 3   ]
--> 9
```

Forcing evaluation of multiple arguments

```haskell
-- force evaluation of x:
(f $! x) y

-- force evaluation of y:
(f x) $! y

-- force evaluation of x and y:
(f $! x) $! y
```

A strict version of `foldl`

We can define a version of `foldl` that is strict in its second argument:

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' (#) v [] = v
foldl' (#) v (x:xs) =
  (foldl' (#) $! (v # x)) xs
```

Now we can evaluate

`foldl' (+) 0 [1..100000000]`

without running out of memory!

Infinite data structures

An infinite list

```
ones :: [Int]
ones = 1 : ones

ones --> 1 : ones
     --> 1 : (1 : ones)
     --> 1 : (1 : (1 : ones))
     --> ...

head ones --> head (1 : ones)
          --> 1
```

Infinite data structures

An infinite data structure is an expression that would contain an infinite number of constructors if it is fully evaluated.

Intuition. An infinite list is a stream of data that produces as much elements as required by its context.

Quiz question

Question. Which of the following defines the infinite list `evens = 0:2:4:6:...`?

1. `evens = 0 : 2 : tail evens`
2. `evens = 0 : map (+2) (tail evens)`
3. `evens = 0 : map (+2) evens`
4. `evens = map (+2) [0..]`

Syntactic sugar for (infinite) lists

`[m..]` denotes the list of all integers starting from `m`:

```
> [1..]
[1,2,3,4,5,6,7,{Interrupted}
> zip [1..] "hallo"
[(1,'h'),(2,'a'),(3,'l'),(4,'l'),(5,'o')]
```

In fact, `[m..]` is syntactic sugar for `enumFrom m`:

```
enumFrom :: (Enum a) => a -> [a]
```

Infinite list of prime numbers

```
sieve (x:xs) =
  let xs' = [ y | y <- xs, y `mod` x /= 0 ]
  in  x : sieve xs'

primes :: [Int]
primes = sieve [2..]

> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
> primes !! 10000
104743
> head (dropWhile (<2023) primes)
2027
```

Separating data and control

With infinite data structures, we can separately define:

- *what* we want to compute (the data)
- *how* it will be used (the control flow)

We can get the data we need for each situation by applying the right function to the infinite list: `take`, `!!`, `takeWhile`, `dropWhile`, ...

Functions for constructing infinite lists

```
repeat :: a -> [a]
repeat x = xs
    where xs = x : xs

cycle :: [a] -> [a]
cycle xs = xs'
    where xs' = xs ++ xs'

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Filtering infinite lists

Warning. Filtering an infinite list will loop forever, even if the result is finite:

```
> filter (<5) [1..]
[1,2,3,4,<loop>
```

Instead, use `takeWhile` to get an initial fragment of an infinite list:

```
> takeWhile (<5) [1..]
[1,2,3,4]
```

The tree labeling problem

Remember the datatype of labeled trees:

```
data Tree a =
  Leaf | Node (Tree a) a (Tree a)
```

Exercise. Given a tree and an infinite list of labels `xs :: [Int]`, define a function `label :: [Int] -> Tree a -> Tree (Int, a)` that labels the tree with `xs`, using each label at most once.

Other infinite data structures

Any (recursive) datatype in Haskell can have infinite structures, not just lists:

```
data Tree a =
  Leaf | Node (Tree a) a (Tree a)

infTree :: Int -> Tree Int
infTree n = Node subTree n subTree
  where subTree = infTree (n+1)
```

See exercises on WebLab!

Case study: computing fast primes

Working with infinite ascending lists

Exercise 1. Define a function

```
merge :: Ord a => [a] -> [a] -> [a]
```

that merges two *ascending* infinite lists into one (removing duplicate entries).

```
> take 10 (merge [2,4..] [3,6..])
[2,3,4,6,8,9,10,12,14,15]
```

Exercise 2. Define a function

```
(\\) :: Ord a => [a] -> [a] -> [a]
```

that takes two *ascending* infinite lists and returns the list of elements that are in the first but not in the second list.

A faster way to calculate primes (1/5)

We could define the infinite list of prime numbers is to first define the infinite list `composites` of *non*-prime numbers:

```
primesV2 :: [Integer]
primesV2 = [2..] \\ composites
```

Question. How to compute `composites`?

A faster way to calculate primes (2/5)

```
multiples = [ map (*n) [n..] | n <- [
mergeAll (xs:xss) = merge xs (mergeAl
composites = mergeAll multiples
```

This loops forever!

A faster way to calculate primes (3/5)

We can fix the loop by using the fact that the smallest element is always in the first list:

```
multiples = [ map (*n) [n..] | n <- [2..] ]
xmerge (x:xs) ys = x : merge xs ys
mergeAll (xs:xss) = xmerge xs (mergeAll xss)
composites = mergeAll multiples
```

`primesV2` is faster than `primes`!

A faster way to calculate primes (4/5)

We can avoid a lot of work by only considering multiples of prime numbers in the calculation of `composites`:

```
primesV3 = [2..] \\ composites
  where
    composites = mergeAll primeMultiples
    primeMultiples = [ map (p*) [p..]
                     | p <- primesV3  ]
```

```
> take 10 primesV3
<loop>
```

Oh no, it's looping again!

A faster way to calculate primes (5/5)

To get the recursion started, we need to specify that `2` is the first prime number:

```
primesV3 = 2 : ([3..] \\ composites)
  where
    composites = mergeAll primeMultiples
    primeMultiples =
      [ map (p*) [p..] | p <- primesV3  ]
```

```
> take 10 primesV3
[2,3,5,7,11,13,17,19,23,29]
```

This one is much faster than V1!

What's next?

Next lecture: Getting started with Agda

To do:

- Read the book:
    - This lecture: sections 15.1–15.5, 15.7
    - Next lecture: section 1 of Agda lecture notes
- Finish week 4 exercises on Weblab
- Install Agda on your PC (see instructions on Brightspace)