Functional Programming

Lecture 9: Lazy evaluation

Twan van Laarhoven

14 November 2022

Outline

- Evaluation orders
- Strictness
- Dynamic programming
- Infinite data structures

Evaluation orders

Evaluation orders

- evaluation = reduction
  - select a reducible expression (redex)
  - rewrite selected redex according to its definition
- different evaluation orders are possible
  - applicative-order evaluation
  - normal-order evaluation
  - lazy evaluation

```
square :: Int → Int
square x = x * x
```

Evaluation

Recall different evaluation orders from before:

```
  square (3 + 4)
= { definition of + }
  square 7
= { definition of square }
  7 * 7
= { definition of * }
  49
```

```
square :: Int → Int
square x = x * x
```

Evaluation

Recall different evaluation orders from before:

```
square (3 + 4)
= { definition of + }
square 7
= { definition of square }
7 * 7
= { definition of * }
49
```

```
square (3 + 4)
= { definition of square }
(3 + 4) * (3 + 4)
= { definition of + }
7 * (3 + 4)
= { definition of + }
7 * 7
= { definition of * }
49
```

Non-terminating evaluations

Consider

```
three :: Integer → Integer
three _ = 3
```

```
infinity :: Integer
infinity = 1 + infinity
```

Two different evaluation orders:

```
three infinity
= { definition of infinity }
three (1 + infinity)
= { definition of infinity }
three (1 + (1 + infinity))
= { definition of  * }
...
```

Non-terminating evaluations

Consider

```
three :: Integer → Integer
three _ = 3
```

```
infinity :: Integer
infinity = 1 + infinity
```

Two different evaluation orders:

```
  three infinity
= { definition of infinity }
  three (1 + infinity)
= { definition of infinity }
  three (1 + (1 + infinity))
= { definition of  * }
  ...
```

```
  three infinity
= { definition of three }
  3
```

Non-terminating evaluations

Consider

```
three :: Integer → Integer
three _ = 3
```

```
infinity :: Integer
infinity = 1 + infinity
```

Two different evaluation orders:

```
three infinity
= { definition of infinity }
three (1 + infinity)
= { definition of infinity }
three (1 + (1 + infinity))
= { definition of  * }
...
```

```
three infinity
= { definition of three }
3
```

Not all evaluation orders terminate, which order to choose?

Applicative-order evaluation

- To reduce the application `f e`:
  1. reduce `e` to a value
  2. expand definition of `f` and continue reducing
- Simple and obvious
- Easy to implement
- May not terminate!
- Other names: innermost evaluation, call-by-value evaluation

Normal-order evaluation

- To reduce the application `f e`:
  1. expand definition of `f`, substituting `e`
  2. reduce result of expansion
- Avoids non-termination, if any evaluation order will
- May involve repeating work
- Other names: outermost evaluation, call-by-name evaluation

A third way: lazy evaluation

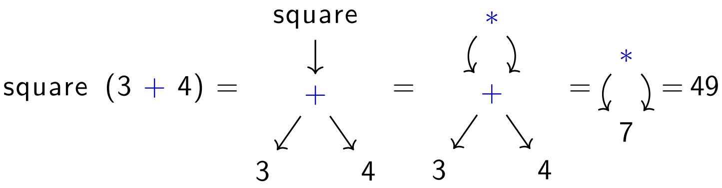
Like normal-order evaluation, but instead of copying arguments we share them

```
  square (3 + 4)
= { definition of square }
  let x = (3 + 4) in x * x
= { reduce first argument of *, definition of +}
  let x = 7 in x * x
= { definition of * }
  let x = 7 in 49
= { garbage collection }
  49
```

Sharing is expressed using **let**-expressions

Lazy evaluation

- Terms are directed graphs, not trees; graph reduction



- Best of both worlds:
  - evaluates argument only when needed, so terminating,
  - but never evaluates argument more than once, so efficient.
- The strategy used by Haskell

Normal forms

- An expression is in *normal form* (NF) when it cannot be reduced any further
- An expression is in *weak head normal form* (WHNF) if it is a constructor applied to its arguments, or a (lambda) function expecting arguments.
  For example:
  - `\n -> 2 * 3 + n`
  - `f x : map f xs`
  - `(1+2, 1-2)`
- Note: An expression in normal form is also in weak head normal form (but converse is not true)

Lazy = Demand-driven evaluation

- Pattern-matching triggers reduction of arguments to WHNF

```haskell
head [1 .. 1000000] = head (1 : [(1 + 1) .. 1000000]) = 1
```

- Patterns matched top to bottom, left to right

```haskell
False && x = False
True  && x = x
```

- guards may also trigger reduction

```haskell
f z | fst z > 0 = fst z
    | otherwise = snd z
```

- local definitions not reduced until needed

```haskell
g x = (x /= 0 && y < 10) where y = 1/x
```

```
foldl :: (b → a → b) → b → [a] → b
foldl _ z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

A demand-driven pipeline

The outermost function drives the evaluation

```
  foldl (+) 0 (map square [1..1000])
= foldl (+) 0 (map square (1:[2..1000]))
= foldl (+) 0 (square 1 : map square [2..1000])
= foldl (+) 1 (map square [2..1000])
= foldl (+) 1 (square 2 : map square [3..1000])
= ...
= foldl (+) 14 (map square [4..1000])
= ...
= 333833500
```

Note: the list [1..1000] never exists all at once

Custom control structures

Define

```
ifZeroElse :: Int → a → a → a
ifZeroElse 0 t _ = t
ifZeroElse _ _ e = e
```

Then

```
  ifZeroElse (1 + 2) huge small
= ifZeroElse 3       huge small
= small
```

Like a built-in construct: only one branch is evaluated

Demand-driven evaluation

- Lazy evaluation has useful implications for program design
- Many computations can be thought of as pipelines
- Expressed with lazy evaluation, intermediate data structures need not exist all at once
- Same effect requires major program surgery in most languages
- Slogan: lazy evaluation allows new and better means of modularizing programs

Infinite data structures

Infinite data structures

- Demand-driven evaluation means that programs can manipulate infinite data structures
- Whole structure is not evaluated at once (fortunately)
- Because of laziness, finite result can be obtained from (finite prefix of) infinite data structure
- Any recursive datatype has infinite elements, but we will consider only lists

Infinite lists

```
ones          = 1 : ones
[n..]         = [n, n + 1, n + 2, .. ]
[n,n + k..]   = [n, n + k, n + 2 * k, .. ]
repeat n      = n : repeat n
iterate f x   = x : iterate f (f x)
fibs          = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Infinite lists (continued)

Can apply functions to infinite data structures

```
filter even [1..] = [2,4,6,8...]
```

Can return finite results

```
takeWhile (< 10) [1..] = [1,2,3,4,5,6,7,8,9]
```

Note that these do not always behave like infinite sets in maths

```
filter (< 10) [1..] = [1,2,3,4,5,6,7,8,9,
```

Primes

Bounded sequences of primes

```
primes m    = [n | n ← [1..m], divisors n == [1,n]]
divisors n = [d | d ← [1..n], n `mod` d == 0]
```

Infinite sequence of primes

```
primes = [n | n ← [1..], divisors n == [1,n]]
```

Much more efficient version: *sieve of Eratosthenes*

```
primes = 2 : sieve [3,5..]
  where sieve (x : xs) = x : sieve [y | y ← xs, y `mod` x /= 0]
```

Modular programming with infinite data

Separate control

```
take 3
takeWhile (<10)
findBestMove
```

from data

```
primes
[1..]
infiniteGameTree
```

Strictness

```
foldl :: (b → a → b) → b → [a] → b
foldl _ z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

The need for strictness

Recall summing a list (simplified):

```
  foldl (+) 0 [1..100]
= foldl (+) 1 [2..100]
= foldl (+) 3 [3..100]
= ...
```

This is a lie! additions are not forced yet

```
  foldl (+) 0 [1..100]
= foldl (+) (0 + 1) [2..100]
= foldl (+) ((0 + 1) + 2) [3..100]
= ...
```

Linear memory usage, instead of constant :(
What to do about it?

Forcing evaluation with seq

The primitive `seq a b` reduces `a` to WHNF, then returns `b`

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

Example:

```haskell
strictSum :: Num a => [a] -> a
strictSum = go 0
  where
  go acc [] = acc
  go acc (x:xs) = let acc' = acc + x in acc' `seq` go acc' xs
```

Strict apply

Defined as

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

Compare:

```
  succ $ succ $ (8*5)
= (succ $ (8*5)) + 1
= ((8*5) + 1) + 1
= (40 + 1) + 1
= 41 + 1
= 42
```

```
  succ $! succ $! (8*5)
= succ $! succ $! 40
= succ $! 40 + 1
= succ $! 41
= 41 + 1
= 42
```

Undefined, and the meaning of strictness

- Some expressions have no normal form (e.g. `infinity`, `1 / 0`)
- We call these undefined
  (sometimes written "$\perp$", pronounced as "bottom")
- When evaluating such an $\perp$, evaluator may hang or may give error message
- Can apply functions to $\perp$; strict functions (`square`) give $\perp$ as a result, non-strict functions (`three`) may give some non-$\perp$ value
- A function f is *strict* iff $f\ \perp = \perp$
- A strict function always evaluates its argument
- `seq` is strict in its first argument:

$$\perp \ \text{`seq`}\ b = \perp$$
$$a \ \text{`seq`}\ b = b$$

Dynamic programming

Case study: postage in Fremont

You are a postal worker in Fremont.
Given postage denominations, 1, 10, 21, 34, 70, and 100,




dispense a given amount to customer using smallest number of stamps

Greedy approach doesn't work:

- greedy: $140 = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1$
- optimal: $140 = 70 + 70$

For simplicity, assume that we are only interested in the total number of stamps

Postage: a recursive implementation

Naive recursive implementation

```
stamps :: [Stamp] → Integer → Integer
stamps ds 0 = 0
stamps ds n = minimum [ stamps ds (n−d) + 1 | d ← ds, d ≤ n ]
```

Postage: a recursive implementation

Naive recursive implementation

```
stamps :: [Stamp] → Integer → Integer
stamps ds 0 = 0
stamps ds n = minimum [ stamps ds (n−d) + 1 | d ← ds, d ≤ n ]
```

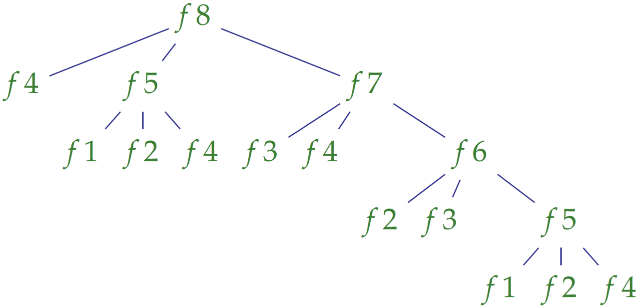
Why naive?

```
>>> stamps [4,3,1] 6
2
>>> stamps [100,70,34,21,10,1] 140
...
```

the second call is answered by a *looong* wait

Naive recursion: analysis

Recursion tree of `f = stamps [4,3,1]`:



Exponential running-time

Problem: solutions to sub-problems are computed over and over again, e.g. `f 5`

Dynamic programming

- Idea: replace a function that computes data by a look-up table that contains data
- Trade space for time: we decrease the running-time at the cost of increased space consumption
- Candidates for a look-up table
  - list: linear running time of look-up $\Theta(i)$
  - search tree: logarithmic running time of look-up $\Theta(\log i)$
  - array: constant running time of look-up $\Theta(1)$

Intermezzo: lazy functional arrays

The library Data.Array provides lazy functional arrays

```
data Array ix val
```

Based on class Ix that maps a contiguous range of indices onto integers.

```
class Ord a ⇒ Ix a where
  range :: (a, a) → [a]
  index :: (a, a) → a → Int
  ...
```

Creating an array: `array :: Ix ix ⇒ (ix, ix) → [(ix, val)] → Array ix val`
The function `array (l, u)` lazily constructs an array from a list of index/value pairs with indices within bounds (l,u)
Array indexing: `(!) :: Ix ix ⇒ Array ix val → ix → val`

Elements of many types can serve as indices: e.g. tuples of indices yield multi-dimensional arrays.

Example: Fibonacci numbers

A naive recursive implementation

```
fib :: Integer → Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n−1) + fib (n−2)
```

Dynamic programming: Fibonacci numbers

```
fibF :: Integer → Integer
fibF n = fibArray ! n
  where
   fib 0 = 1
   fib 1 = 1
   fib n = fibArray ! (n−1) + fibArray ! (n−2)

   fibArray = array (0,n) [(i, fib i) | i ← [0..n]]
```

Dynamic programming: abstraction

```haskell
fibF :: Integer → Integer
fibF n = fibM n
  where
   fib 0 = 1
   fib 1 = 1
   fib n = fibM (n−1) + fibM (n−2)

   fibM = memo (0,n) fib

memo :: Ix ix ⇒ (ix,ix) → (ix → a) → (ix → a)
memo bounds f = (\i → table ! i)
  where table = array bounds [(i, f i) | i ← range bounds]
```

Computing fib

fibF 4

```
fibF n = fibArray ! n
  where
   fib 0 = 1
   fib 1 = 1
   fib n = fibArray ! (n-1) +
           fibArray ! (n-2)

   fibArray = array (0,n)
       [(i, fib i) | i <- [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	fib 0
1	fib 1
2	fib 2
3	fib 3
4	fib 4

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n−1) +
          fibArray ! (n−2)

  fibArray = array (0,n)
     [(i, fib i) | i ← [0..n]]
```

Computing fib

$$\text{fibF } 4 = \text{fibArray} \; ! \; 4$$
where

i	fibArray ! i
0	fib 0
1	fib 1
2	fib 2
3	fib 3
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) +
          fibArray ! (n-2)

  fibArray = array (0,n)
      [(i, fib i) | i <- [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	fib 0
1	fib 1
2	fib 2
3	fibArray !2 + fibArray !1
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n−1) +
          fibArray ! (n−2)

  fibArray = array (0,n)
     [(i, fib i) | i ← [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	fib 0
1	fib 1
2	fibArray !1 + fibArray !0
3	fibArray !2 + fibArray !1
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n−1) +
          fibArray ! (n−2)

  fibArray = array (0,n)
     [(i, fib i) | i ← [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	fib 0
1	1
2	fibArray !1 + fibArray !0
3	fibArray !2 + fibArray !1
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) +
          fibArray ! (n-2)

  fibArray = array (0,n)
     [(i, fib i) | i ← [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	fib 0
1	1
2	1 + fibArray !0
3	fibArray !2 + fibArray !1
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) +
          fibArray ! (n-2)

  fibArray = array (0,n)
      [(i, fib i) | i <- [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	1
1	1
2	1 + fibArray !0
3	fibArray !2 + fibArray !1
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
   fib 0 = 1
   fib 1 = 1
   fib n = fibArray ! (n−1) +
           fibArray ! (n−2)

   fibArray = array (0,n)
       [(i, fib i) | i ← [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	1
1	1
2	1 + 1
3	fibArray !2 + fibArray !1
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) +
          fibArray ! (n-2)

  fibArray = array (0,n)
     [(i, fib i) | i <- [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	1
1	1
2	2
3	fibArray !2 + fibArray !1
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n−1) +
          fibArray ! (n−2)

  fibArray = array (0,n)
     [(i, fib i) | i ← [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	1
1	1
2	2
3	2 + fibArray !1
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) +
          fibArray ! (n-2)

  fibArray = array (0,n)
     [(i, fib i) | i ← [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	1
1	1
2	2
3	2 + 1
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) +
          fibArray ! (n-2)

  fibArray = array (0,n)
      [(i, fib i) | i ← [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
    where

i	fibArray ! i
0	1
1	1
2	2
3	3
4	fibArray !3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) +
          fibArray ! (n-2)

  fibArray = array (0,n)
      [(i, fib i) | i ← [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	1
1	1
2	2
3	3
4	3 + fibArray !2

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n−1) +
          fibArray ! (n−2)

  fibArray = array (0,n)
      [(i, fib i) | i ← [0..n]]
```

```haskell
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) +
          fibArray ! (n-2)

  fibArray = array (0,n)
     [(i, fib i) | i <- [0..n]]
```

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	1
1	1
2	2
3	3
4	3 + 2

Computing fib

fibF 4 = fibArray ! 4
  where

i	fibArray ! i
0	1
1	1
2	2
3	3
4	5

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) +
          fibArray ! (n-2)

  fibArray = array (0,n)
      [(i, fib i) | i ← [0..n]]
```

Computing fib

fibF 4 = 5
    where

i	fibArray ! i
0	1
1	1
2	2
3	3
4	5

```
fibF n = fibArray ! n
  where
  fib 0 = 1
  fib 1 = 1
  fib n = fibArray ! (n-1) +
          fibArray ! (n-2)

  fibArray = array (0,n)
      [(i, fib i) | i ← [0..n]]
```

Postage: dynamic programming

Replace recursive calls by table look-ups

```
stampsDP :: [Stamp] → Integer → Integer
stampsDP ds n = stampsArray!n where
  stamps 0 = 0
  stamps i = minimum [ stampsArray ! (i−d) + 1 | d ← ds, d ≤ i ]

  stampsArray = array (0,n) [(i,stamps i) | i ← [0..n]]
```

Postage: dynamic programming

Replace recursive calls by table look-ups

```
stampsDP :: [Stamp] → Integer → Integer
stampsDP ds n = stampsArray!n where
  stamps 0 = 0
  stamps i = minimum [ stampsArray ! (i−d) + 1 | d ← ds, d ≤ i ]

  stampsArray = array (0,n) [(i,stamps i) | i ← [0..n]]
```

- Lazy evaluation at work: look-up table is filled in a demand-driven fashion
- Linear running time $\Theta(dn)$ where d is the number of denominations and n is the target

```
>>> stampsDP [100,70,34,21,10,1] 140
2
```

Case study 2: knapsack problem

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

Again a greedy approach doesn't work (in general)

Knapsack: a recursive implementation

The inventory

```
type Weight  = Int
type Value   = Int
type Item    = (String, Weight, Value)

items =
   [("map",9,150),("compass",13,35),("water",153,200),("sandwich",50,160),("glucose",15,60),
    ("banana",27,60),("apple",39,40),("cheese",23,30),("beer",52,10),("cream",11,70),
    ("tshirt",24,15),("trousers",48,10),("umbrella",73,40),("trousers",42,70)]
```

A recursive implementation

```
fillKS :: [Item] → Weigth → (Value, [String])
fillKS []                  capa = (0,[])
fillKS ((n, w, v) : items) capa
  | w ≤ capa = let (vt, itst) = fillKS items (capa − w)
                   (vs, itss) = fillKS items capa
               in if vt+v > vs then (vt+v, n:itst) else (vs, itss)
  | otherwise = fillKS items capa
```

Knapsack: dynamic programming

We again replace recursive calls by table look-ups

```
fillKSA :: [Item] → Weigth → (Value, [String])
fillKSA items capa = fillArray ! (length its, capa)
  where
   fill 0 capa = (0,[])
   fill i capa
     | w ≤ capa = let (vt, itst) = fillArray ! (i−1, capa − w)
                      (vs, itss) = fillArray ! (i−1, capa)
                  in if vt+v > vs then (vt+v, n:itst) else (vs, itss)
     | otherwise = fillArray ! (i−1, capa)
    where (n, w, v) = items !! (i−1)

   fillArray = array ((0,0),(length its,capa))
           [((i,j), fill i j) | i ← [0..length its], j ← [0..capa]]
```

Knapsack: dynamic programming

Or by memoization

```
fillKSA :: [Item] → Weigth → (Value, [String])
fillKSA items capa = fillMemo (length its, capa)
  where
   fill 0 capa = (0,[])
   fill i capa
     | w ≤ capa = let (vt, itst) = fillMemo (i−1, capa − w)
                      (vs, itss) = fillMemo (i−1, capa)
                  in if vt+v > vs then (vt+v, n:itst) else (vs, itss)
     | otherwise  = fillMemo (i−1,capa)
    where (n, w, v) = items !! (i−1)

   fillMemo = memo ((0,0),(length its,capa)) fill
```

Take away

Summary

- Evaluation strategies:
  - Applicative order: efficient, but may not terminate
  - Normal order: avoids non-termination if possible, but work possibly duplicated
  - Lazy evaluation: best of both worlds
- Enables infinite data structures
- Lazy values can happen inside data structures
- Better modularity: creation and traversal of structures can be cleanly separated (e.g. game trees)