Lazy Evaluation*

Andreas Lochbihler

Department of Computer Science
ETH Zurich

*Thanks to David Basin for slide material

Evaluation Strategy

- Evaluation strategy has, until now, been unimportant
- Example: `map (\x -> x * x) ([1,2,3] ++ [2*2])`

$$[1, 4, 9, 16]$$

- Haskell is **lazy**: expressions evaluated only when necessary

```
loop x = (loop x) + 1

divZero = 1 `div` 0

f g x = g 7
```

```
? f (*2) (loop 0)
14
? divZero
*** Exception: divide by zero
? f (+1) divZero
8
```

- Subtle consequences such as data-driven computation

Lazy evaluation

- Evaluation based on function application and substitution

$$f\ x = ...x...x... \quad \Rightarrow \quad f\ a = ...a...a...$$

- Example for $f\ x\ y = x + y$

$$f\ (9-3)\ (f\ 34\ 3) = (9-3) + (f\ 34\ 3)$$

  - In Haskell, substitution occurs **without** argument evaluation
  - Evaluation of arguments is postponed

$$\cdots = 6 + (f\ 34\ 3) = 6 + (34 + 3) = 6 + 37 = 43$$

- Sometimes expressions are never evaluated
  This can save arbitrarily large amounts of time

Example in ghc

```
g :: Int -> Int -> Int
g x y = x + 12

switch :: Bool -> Int -> Int -> Int
switch True x _  = x
switch False _ y = y

?  g 7 (loop 0)
19 :: Int

? switch True 8 (loop 0)
8 :: Int

? switch False 8 (loop 0)
*** Exception: stack overflow

? switch False 0 divZero
*** Exception: divide by zero
```

Lazy evaluation (cont.)

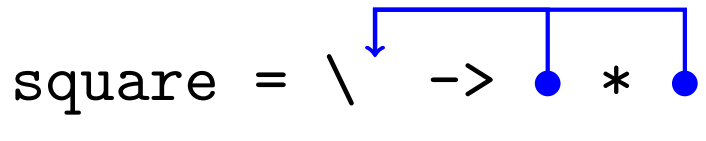
- Potential problem: duplicated computation, e.g., `square x = x * x`

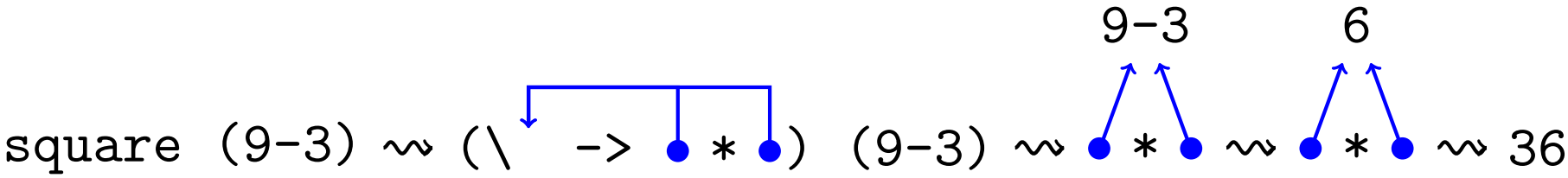
  ```
  square (9-3) = (9-3) * (9-3) = 6 * (9-3) = 6 * 6 = 36
  ```

  The same expression $9 - 3$ is evaluated twice here

- Duplication avoided by simultaneously reducing both occurrences

  Implementation based on **sharing**:
  terms represented as **directed graphs**





- Summary: function arguments are evaluated only when needed and at most once

Evaluation — further details

Typical function

```
f p1 p2 ... pk
  | g1  = e1
  | g2  = e2
    :
  | otherwise = en
  where v1 ... = r1
        :
f pi ... pj
  | gm  = em
    :
  where ...
```

Built using patterns, guards, and local definitions

Evaluation — pattern matching

- Arguments evaluated as far as needed to determine pattern match

```
f []     _      = 0          -- (f.1)
f _      []     = 0          -- (f.2)
f (a:_) (b:_) = a + b       -- (f.3)
```

- Haskell notation: `[n .. m] == enumFromTo n m`

```
? enumFromTo 1 10
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] :: [Int]
```

- `f [1 .. 3] [4 .. 6]` executes as follows:

```
f [1 .. 3] [4 .. 6]                    -- Does (f.1) match?
 = f (1 : [2 .. 3]) [4 .. 6]           -- No. Does (f.2) match?
 = f (1 : [2 .. 3]) (4 : [5 .. 6])     -- No. Does (f.3) match?
 = 1 + 4                               -- Yes!
 = 5
```

Evaluation — guards

Execution proceeds sequentially, until success

```
f a b c
  | a >= b && a >= c = a
  | b >= a && a >= c = b
  | otherwise        = c
```

Example

```
f (2+3) (4-1) (3+9)
   ?? (2+3) >= (4-1) && (2+3) >= (3+9)     -- try 1st guard
   ??   = 5 >= 3     && 5 >= (3+9)
   ??   = True       && 5 >= (3+9)
   ??   = 5 >= (3+9)
   ??   = 5 >= 12
   ??   = False
   ?? 3 >= 5 && 5 >= 12  -- try 2nd guard, already partially evaluated
   ??   = False && 5 >= 12
   ??   = False
   ?? otherwise        -- try final guard (= True)
 = 12
```

Evaluation — local definitions

Local definitions (with `where`) are also lazily evaluated

```
f a b
  | notNil l  = front l
  | otherwise = b
  where
    l = [a .. b]

front (c:d:_) = c+d
front [c] = c

notNil [] = False
notNil _  = True
```

```
f 3 5
    ?? notNil l
    ??    where l = [3 .. 5]
    ?? = notNil ([3 .. 5])
    ?? = notNil (3:[4 .. 5])
    ?? = True
  = front l
      where
      l = 3:[4 .. 5]
        = 3:4:[5]
  = 3+4
  = 7
```

Evaluation — misc.

- Functions are evaluated top-down (outermost operator first)

$$\underline{f\ e_1\ (f\ e_2\ 17)}$$

- and otherwise usually from left to right, depending on operator precedence

$$f\ e_1 + f\ e_2$$
$$f\ e_1 + f\ e_2 * f\ e_3$$

- This kind of evaluation is as natural as "eager evaluation"

  But the consequences (and possibilities) are surprising

Tracing evaluation

- `trace :: String -> a -> a` (in module `Debug.Trace`)
  prints a message when it evaluates and returns second argument.

- **Use only for debugging!** It breaks referential transparency.

```
import Debug.Trace (trace)

f x = trace "called f" (x * 2)
g y = trace "called g" (y + 3)
```

```
? g (f 5)
called g
called f
13
```

- Tracing can change the evaluation order

```
import Debug.Trace (trace)

f x = trace "called f" (x * 2)
g y = trace ("called g " ++ show y) (y + 3)
```

```
? g (f 5)
called f
called g 10
13
```

Application 1: data-driven programming

- Data can be generated lazily (on demand)

  The result is improved runtime complexity

- Example: sum the 4th powers of the numbers 1 through n

- Data-driven solution
  - Construct the list of numbers $[1 .. n]$
  - Compute each 4th power, resulting in $[1, 16, \ldots, n^4]$
  - Sum the list of powers

- Resulting program: `sumFourthPowers n = sum (map (^4) [1 .. n])`

  Would a loop (e.g., in Java) be better?

Example (cont.)

```
sum []     = 0
sum (x:xs) = x + sum xs

map f []     = []
map f (x:xs) = f x : map f xs

sumFourthPowers n = sum (map (^4) [1 .. n])
```

Execution as follows

```
sumFourthPowers n
 = sum (map (^4) [1 .. n])
 = sum (map (^4) (1:[2 .. n]))
 = sum ((^4) 1 : map (^4) [2 .. n])
 = (^4) 1 + sum (map (^4) [2 .. n])
 = 1 + sum (map (^4) [2 .. n])
 = 1 + ((^4) 2 + map (^4) [3 .. n])
 = 1 + (16 + sum (map (^4) [3 .. n]))
 :
 = 1 + (16 + (81 + ... + n^4))
```

Intermediate lists are **not** fully constructed,
head is immediately turned into an addition.

Data-driven programming

- Example 2: list minimum

```
isort []     = []
isort (x:xs) = ins x (isort xs)

lmin = head . isort
```

```
ins a [] = [a]
ins a (x:xs)
  | a <= x    = a : (x:xs)
  | otherwise = x : ins a xs
```

- `lmin [8,1,7,6]` executes as follows (focusing on `isort`)

```
isort [8,1,7,6]
  = ins 8 (isort [1,7,6])
  = ins 8 (ins 1 (isort [7,6]))
  = ins 8 (ins 1 (ins 7 (isort [6])))
  = ins 8 (ins 1 (ins 7 (ins 6 (isort []))))
  = ins 8 (ins 1 (ins 7 (ins 6 [])))
  = ins 8 (ins 1 (ins 7 [6]))
  = ins 8 (ins 1 (6 : ins 7 []))          -- next evaluation step
  = ins 8 (1 : (6 : ins 7 []))            -- suspended evaluation
  = 1 : ins 8 (6 : ins 7 [])              -- remains unevaluated
```

- Thus `lmin l` executes in linear time!

Application 2: infinite data

- Lazy evaluation enables finite representation of infinite data
- Example: infinite lists (streams)

```
ones = 1 : ones
from n = n : from (n+1)

? ones
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ^C{Interrupted!}

? from 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ^C{Interrupted!}
```

- Example: infinite trees

```
data Tree a = Leaf | Node a (Tree a) (Tree a)    deriving (Show, Eq)
t = Node 1 t t

? t
Node 1 (Node 1 (Node 1 (Node 1 ^C{Interrupted!}
```

Infinite data (cont.)

- One can compute with infinite data in finite time

```
addFirstTwo (a:b:_) = a + b

? addFirstTwo ones
2 :: Int
```

- Executes as follows

```
addFirstTwo ones                          -- ones = 1 : ones
 = addFirstTwo (1:ones)
 = addFirstTwo (1:1:ones)
 = 1 + 1
 = 2
```

- Conceptually elegant: we describe an infinite stream (tree, etc.) and compute with arbitrarily large finite prefixes of it

Example: prime numbers

- One of the oldest algorithms: the Sieve of Eratosthenes
  1. Generate the list of all natural numbers, starting with 2
  2. Mark the first unmarked number
  3. Cross out all multiples of the last marked number
  4. Go to step 2
- N.B.
  - Infinitely many prime numbers: but each is eventually marked
  - Strictly speaking, this is **not** an algorithm since the steps cannot be carried out to completion in finite time
- Note that careful analysis of the complexity of the sieve presented on the next slide leads to doubt about its faithfulness.

Implementing the Sieve of Eratosthenes

1. Generate list: `[2 .. ]`
2. Marking: function `head :: [a] -> a` determines first element
3. Cross out all multiples: `dropMults`

   ```
   dropMults x ys = filter (\y -> y `mod` x /= 0) ys
   ```

4. Repetition via recursion:

   ```
   sieve xs = head xs : sieve (dropMults (head xs) (tail xs))
   ```

The result

```
primes = sieve [2 ..]

? take 50 primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
 67,71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137,
 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211,
 223, 227, 229] :: [Int]
```

Example: Newton's algorithm

Problem: compute square roots

Input: The radicand $r \in \mathcal{R}$, with $r \geq 0$, and the first approximation $a_0 \in \mathcal{R}$, where $a_0 > 0$

Output: $\sqrt{r} \in \mathcal{R}$

Procedure: The sequence of approximations is defined by

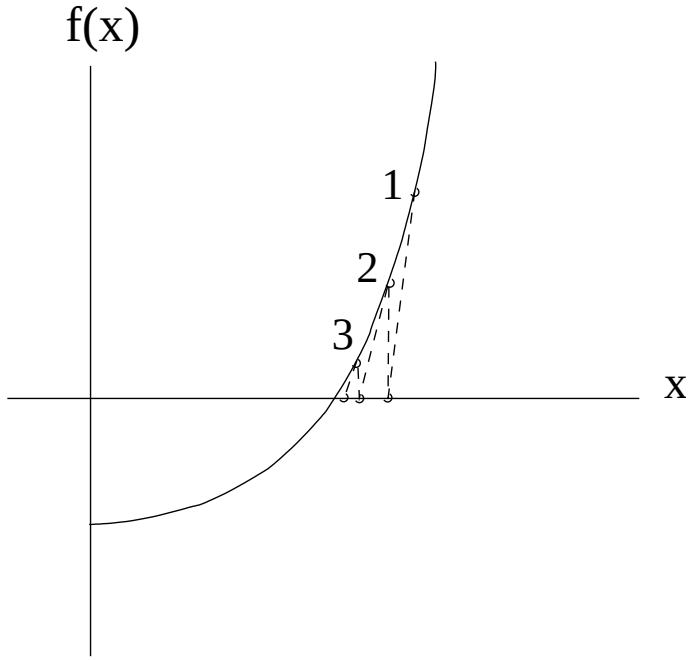
$$a_{i+1} = (a_i + r/a_i)/2$$

If the sequence of a_i converges to a, then $a = (a + r/a)/2$.
I.e., $a = \sqrt{r}$.

Numerical test: If $\left|\frac{a_{i+1} - a_i}{a_i}\right| < \epsilon$, then a_{i+1} is the result.

Newton (cont.)

- Example: $a_0 = 5$, $r = 2$, $\sqrt{r} = 1.4142135623\ldots$
- Iterative search for root of $f(x) = x^2 - 2$



Sequence of approximations is

$$[5.0, 2.7, 1.72037, 1.44146, 1.41447, 1.41421, \ldots]$$

- For correctness, convergence criteria, etc. see other courses.

Traditional implementations

Imperative

```
static double EPS = 0.001;


double root(double r, double a0) {
  double a, a';
  a' = a0;
  do {
    a = a';
    a' = (a + r / a) / 2.0;
  } while (!(abs((a'-a)/a)<EPS));
  return a';
}
```

Functional (cf. exercise sheet 2)

```
eps :: Double
eps = 0.001

root :: Double -> Double -> Double
root r a0 = iter a0
  where
    iter a
      | goodEnough a' a = a'
      | otherwise       = iter a'
      where a' = (a + r / a) / 2.0

    goodEnough a' a =
      abs ((a' - a) / a) < eps
```

Correct implementation, but as a monolithic unit!

Implementation — Haskell

```
eps = 0.001

improve r x = (x + r / x) / 2        --- a[i+1] = (a[i] + r/a[i]) / 2

iterate f x = x : iterate f (f x)  --- [x, f x, f(f x), ...]

within (x:(x':xs))
  | goodEnough x' x = x'
  | otherwise                  = within (x':xs)
  where
    goodEnough x' x = abs ((x' - x) / x) < eps

root x0 r = within (iterate (improve r) x0)
```

Direct implementation: generate and test

Program simpler to understand and modify
(e.g. with other convergence tests)

Correctness

- Lazy evaluation is powerful.

  But it complicates analyzing program complexity and correctness

- Types like $[Int]$ actually include

  1. Finite, everywhere defined lists like $[1, 3, 5]$
  2. Finite lists with "undefined" elements

```
undef :: t
undef = undef

? [1,2,undef]
[1,2,^C{Interrupted!}]
```

  3. Infinite lists with defined or undefined elements
     e.g. $[1..]$ or $[1, \mathit{undef}, 2, \mathit{undef}, 3, \mathit{undef}, \ldots]$

Correctness of lazy programs

- Induction is only sound for (1): finite, everywhere defined data.
  - When we show by induction that

$$\forall xs\, ys :: [t].\, \mathit{map}\ \ f\ (xs +\!\!+\ \ ys) = \mathit{map}\ f\ xs +\!\!+\ \ \mathit{map}\ f\ ys$$

    we have proven the equality only for all finite lists!
  - But data of kind (2) and (3) also belong to $[t]$
- We will not consider this correctness question further in this class.

  Thus, when we prove a proposition by induction, we mean only for all data of kind (1).

Summary

- Lazy evaluation enables new ways of writing programs

  Data is created or further evaluated only on demand!

- We can describe algorithms that (potentially) produce and operate on infinite data
  - Infinite data of course is never generated
  - But arbitrarily large quantities can be produced on demand
- Lazy evaluation is simple but exciting and has wide scope
  - Many real programs are not algorithms in the strict sense
  - E.g. reactive systems, operating systems, ... shouldn't terminate
  - Such systems can be implemented as (lazy) stream processors!
  - Establishing correctness requires, however, other techniques