12. Lazy evaluation

Introduction

So far, we have not looked at the details of how Haskell expressions are evaluated.
So far, we have not looked at the details of how Haskell expressions are evaluated. The evaluation strategy is called *lazy evaluation* („verzögerte Auswertung“).

**Advantages:**
- Avoids unnecessary evaluations
- Terminates as often as possible
- Supports infinite lists
- Increases modularity

Therefore Haskell is called a *lazy functional language*. 
Expressions are evaluated \textit{(reduced)} by successively applying definitions until no further reduction is possible.

Example:

\begin{verbatim}
  sq :: Integer -> Integer
  sq n = n * n

  One evaluation:
  sq(3+4)
\end{verbatim}

\begin{verbatim}
  sq(3+4) = sq 7 = 7 * 7
\end{verbatim}
Theorem
Any two terminating evaluations of the same Haskell expression lead to the same final result.

This is not the case in languages with side effects:

Example
Let n have value 0 initially.
Two evaluations:
n + (n := 1)
Theorem
Any two terminating evaluations of the same Haskell expression lead to the same final result.

This is not the case in languages with side effects:

Example
Let \( n \) have value 0 initially.

Two evaluations:
\[
\text{\( n + (n := 1) = 0 + (n := 1) = 0 + 1 \)}
\]

Theorem
Any two terminating evaluations of the same Haskell expression lead to the same final result.

This is not the case in languages with side effects:

Example
Let \( n \) have value 0 initially.

Two evaluations:
\[
\text{\( n + (n := 1) = 0 + (n := 1) = 0 + 1 \)}
\]

Reduction strategies
An expression may have many reducible subexpressions:
\[
\text{\( sq \ (3+4) \)}
\]

Terminology: \( \text{\( \text{redex} = \text{reducible expression} \)} \)
Reduction strategies

An expression may have many reducible subexpressions:

\[ \text{sq (3+4)} \]

Terminology: *redex* = reducible expression

Two common reduction strategies:

**Innermost reduction** Always reduce an innermost redex.
Corresponds to *call by value*:

- Arguments are evaluated before they are substituted into the function body

- \[ \text{sq (3+4)} = \text{sq 7} = 7 \times 7 \]

**Outermost reduction** Always reduce an outermost redex.

- Corresponds to *call by name*:
  - The unevaluated arguments are substituted into the function body

Reduction strategies

An expression may have many reducible subexpressions:

\[ \text{sq (3+4)} \]

Terminology: *redex* = reducible expression

Two common reduction strategies:

**Innermost reduction** Always reduce an innermost redex.
Corresponds to *call by value*:

- Arguments are evaluated before they are substituted into the function body

- \[ \text{sq (3+4)} = \text{sq 7} = 7 \times 7 \]

**Outermost reduction** Always reduce an outermost redex.

- Corresponds to *call by name*:
  - The unevaluated arguments are substituted into the function body
Reduction strategies

An expression may have many reducible subexpressions:

\[
\text{sq (3+4)}
\]

Terminology: \textit{redex} = reducible expression

Two common reduction strategies:

\textbf{Innermost reduction} Always reduce an innermost redex.
Corresponds to \textit{call by value}:
Arguments are evaluated before they are substituted into the function body
\[
\text{sq (3+4)} = \text{sq 7} = 7 \times 7
\]

\textbf{Outermost reduction} Always reduce an outermost redex.
Corresponds to \textit{call by name}:
The unevaluated arguments are substituted into the function body
\[
\text{sq (3+4)} = (3+4) \times (3+4)
\]

Comparison: termination

Definition:
\[
\text{loop = tail loop}
\]

Innermost reduction:
\[
\text{fst (1,loop)}
\]

Definition:
\[
\text{loop = tail loop}
\]

Innermost reduction:
\[
\text{fst (1,loop)} = \text{fst(1,tail loop)}
\]
\[
= \text{fst(1,tail(tail loop))}
\]
\[
= \ldots
\]
Comparison: termination

Definition:
\( \text{loop} = \text{tail} \text{ loop} \)

Innermost reduction:
\[
\text{fst}(1, \text{loop}) = \text{fst}(1, \text{tail} \text{ loop}) \\
= \text{fst}(1, \text{tail}(\text{tail} \text{ loop})) \\
= \ldots
\]

Outermost reduction:
\[
\text{fst}(1, \text{loop}) = 1
\]

Theorem If expression \( e \) has a terminating reduction sequence, then outermost reduction of \( e \) also terminates.

Comparison: termination

Definition:
\( \text{loop} = \text{tail} \text{ loop} \)

Innermost reduction:
\[
\text{fst}(1, \text{loop}) = \text{fst}(1, \text{tail} \text{ loop}) \\
= \text{fst}(1, \text{tail}(\text{tail} \text{ loop})) \\
= \ldots
\]

Outermost reduction:
\[
\text{fst}(1, \text{loop}) = 1
\]

Theorem If expression \( e \) has a terminating reduction sequence, then outermost reduction of \( e \) also terminates.

Why is this useful?

Example
Can build your own control constructs:

\[
\text{switch} :: \text{Int} \to a \to a \to a
\]
Why is this useful?

**Example**
Can build your own control constructs:

```haskell
switch :: Int -> a -> a -> a
switch n x y
  | n > 0     = x
  | otherwise = y
```

```haskell
switch :: Int -> a -> a -> a
switch n x y
  | n > 0     = x
  | otherwise = y

fac :: Int -> Int
fac n = switch n (n * fac(n-1)) 1
```

---

### Comparison: Number of steps

**Innermost reduction:**

```haskell
\( \text{sq} (3+4) = \text{sq} 7 = 7 \times 7 = 49 \)
```

**Outermost reduction:**

```haskell
\( \text{sq}(3+4) = (3+4) \times (3+4) = 7 \times (3+4) = 7 \times 7 = 49 \)
```
Comparison: Number of steps

Innermost reduction:
\[ \text{sq} (3+4) = \text{sq} 7 = 7 \times 7 = 49 \]

Outermost reduction:
\[ \text{sq} (3+4) = (3+4) \times (3+4) = 7 \times (3+4) = 7 \times 7 = 49 \]

More outermost than innermost steps!

Innermost reduction:
\[ \text{sq} (3+4) = \text{sq} 7 = 7 \times 7 = 49 \]

Outermost reduction:
\[ \text{sq} (3+4) = (3+4) \times (3+4) = 7 \times (3+4) = 7 \times 7 = 49 \]

More outermost than innermost steps!
How can outermost reduction be improved?

Sharing!
**Theorem**

Lazy evaluation never needs more steps than innermost reduction.
The principles of lazy evaluation:

- Arguments of functions are evaluated only if needed to continue the evaluation of the function.
- Arguments are not necessarily evaluated fully, but only far enough to evaluate the function. (Remember \texttt{fst (1,loop)})
- Each argument is evaluated at most once (sharing!)
**Pattern matching**

Example

```haskell
f :: [Int] -> [Int] -> Int
f []  ys = 0
f (x:xs) [] = 0
f (x:xs) (y:ys) = x+y

Lazy evaluation:
f [1..3] [7..9]
```

```
-- does f.1 match?
= f (1 : [2..3]) [7..9]
```
Pattern matching

Example

\[ f :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Int} \]
\[ f \text{[]} \quad \text{ys} = 0 \]
\[ f (x:xs) \text{[]} \quad = 0 \]
\[ f (x:xs) (y:ys) = x+y \]

Lazy evaluation:
\[ f \text{[1..3] [7..9]} \quad -- \text{does f.1 match?} \]
\[ = f (1 : [2..3]) [7..9] \quad -- \text{does f.2 match?} \]
\[ = f (1 : [2..3]) (7 : [8..9]) \quad -- \text{does f.3 match?} \]
\[ = 1+7 \]
\[ = 8 \]

Guards

Example

\[ f m n p \mid m >= n \&\& m >= p = m \]
\[ | n >= m \&\& n >= p = n \]
\[ | \text{otherwise} = p \]

Lazy evaluation:
\[ f (2+3) (4-1) (3+9) \]
Example
f m n p | m >= n && m >= p = m
     | n >= m && n >= p = n
     | otherwise = p

Lazy evaluation:
f (2+3) (4-1) (3+9)
? 2+3 >= 4-1 && 2+3 >= 3+9

Guards
f m n p | m >= n && m >= p = m
     | n >= m && n >= p = n
     | otherwise = p

Lazy evaluation:
f (2+3) (4-1) (3+9)
? 2+3 >= 4-1 && 2+3 >= 3+9
?

Example
f m n p | m >= n && m >= p = m
     | n >= m && n >= p = n
     | otherwise = p

Lazy evaluation:
f (2+3) (4-1) (3+9)
? 2+3 >= 4-1 && 2+3 >= 3+9
?
?
? = True && 5 >= 3+9
?
? = 5 >= 3+9
?
? = 5 >= 12
?
? = False
?
? = 3 >= 5 && 3 >= 12
?
? = False && 3 >= 12
?
? = False
?
? otherwise = True
Example

\[
\begin{align*}
\text{guard} & : \text{Int} \times \text{Int} \times \text{Int} \rightarrow \text{Int} \\
\text{guard} (m, n, p) & = \begin{cases} 
    m & \text{if } n \geq m \\
    n & \text{if } m \geq n \\
    p & \text{otherwise}
\end{cases}
\end{align*}
\]

Lazy evaluation:

\[
\begin{align*}
\text{guard} (2+3, (4-1), (3+9)) & = 3+9 \\
& = \begin{cases} 
    2+3 & \text{if } (4-1) \geq 2+3 \\
    5 & \text{if } (4-1) \geq 5 \\
    \text{True} & \text{if } 5 \geq 3+9 \\
    \text{False} & \text{if } 3 \geq 5 \\
    \text{otherwise} & \text{True}
\end{cases}
\end{align*}
\]

where

Same principle: definitions in where clauses are only evaluated when needed and only as much as needed.

Lambda

Haskell never reduces inside a lambda

Same principle: definitions in where clauses are only evaluated when needed and only as much as needed.
Haskell never reduces inside a lambda

Example: $\lambda x \rightarrow \text{False} \&\& x$ cannot be reduced

Reasons:
- Functions are black boxes
- All you can do with a function is apply it

Example: $(\lambda x \rightarrow \text{False} \&\& x) \text{True} = \text{False} \&\& \text{True} = \text{False}$
Arithmetic operators and other built-in functions evaluate their arguments first

Example
3 * 5 is a redex

Arithmetic operators and other built-in functions evaluate their arguments first

Example
3 * 5 is a redex
0 * head(...) is not a redex

They behave like their Haskell definition:

(&&) :: Bool -> Bool -> Bool
True && y = y
False && y = False
Lazy evaluation evaluates an expression only when needed and only as much as needed.

Lazy evaluation evaluates an expression only when needed and only as much as needed.

(“Call by need”)

**12.1 Applications of lazy evaluation**

```
min = head . inSort
```
Minimum of a list

\[
\text{min} = \text{head} \cdot \text{inSort}
\]

\[
\text{inSort} : \text{Ord} \ a \Rightarrow [a] \rightarrow [a]
\]

\[
\text{inSort} [] = []
\]

\[
\text{inSort} \ (x:xs) = \text{ins} \ x \ (\text{inSort} \ xs)
\]

\[
\text{ins} : \text{Ord} \ a \Rightarrow a \rightarrow [a] \rightarrow [a]
\]

\[
\text{ins} \ x [] = [x]
\]

\[
\text{ins} \ x \ (y:ys) \ | \ x \leq y \ = \ x : y : ys
\]

\[
\ | \ \text{otherwise} \ = \ y : \text{ins} \ x \ ys
\]

\[
\implies \text{inSort} \ [6,1,7,5]
\]

\[
= \text{ins} \ 6 \ (\text{ins} \ 1 \ (\text{ins} \ 7 \ (\text{ins} \ 5 \ [])))
\]
min \{6,1,7,5\} = \text{head}(\text{insSort} \{6,1,7,5\})
= \text{head}(\text{ins} 6 \ (\text{ins} 1 \ (\text{ins} 7 \ (\text{ins} 5 [])))

\text{Minimum of a list}

min = \text{head} \cdot \text{insSort}

\text{insSort :: Ord a} \Rightarrow [a] \rightarrow [a]
\text{insSort []} = []
\text{insSort (x:xs)} = \text{ins} x \ (\text{insSort} \ xs)

\text{ins :: Ord a} \Rightarrow a \rightarrow [a] \rightarrow [a]
\text{ins} x [] = [x]
\text{ins} x (y:ys) \mid x \leq y = x : y : ys
\quad \mid \text{otherwise} = y : \text{ins} x ys

\Rightarrow \text{insSort} \{6,1,7,5\}
= \text{ins} 6 \ (\text{ins} 1 \ (\text{ins} 7 \ (\text{ins} 5 [])))
min [6,1,7,5] = head(inSort [6,1,7,5])
= head(ins 6 (ins 1 (ins 7 (ins 5 []))))
= head(ins 6 (ins 1 (ins 7 (5 : []))))
= head(ins 6 (ins 1 (5 : ins 7 [])))

min [6,1,7,5] = head(inSort [6,1,7,5])
= head(ins 6 (ins 1 (ins 7 (ins 5 []))))
= head(ins 6 (ins 1 (ins 7 (5 : []))))
= head(ins 6 (ins 1 (5 : ins 7 [])))
= head(ins 6 (1 : 5 : ins 7 []))


```
min [6,1,7,5] = head(inSort [6,1,7,5])
= head(ins 6 (ins 1 (ins 7 (ins 5 []))))
= head(ins 6 (ins 1 (ins 7 (5 : []))))
= head(ins 6 (ins 1 (5 : ins 7 [])))
= head(ins 6 (1 : 5 : ins 7 []))
= head(1 : ins 6 (5 : ins 7 []))
= 1
```

Lazy evaluation needs only linear time
although `inSort` is quadratic

```
min [6,1,7,5] = head(inSort [6,1,7,5])
= head(ins 6 (ins 1 (ins 7 (ins 5 []))))
= head(ins 6 (ins 1 (ins 7 (5 : []))))
= head(ins 6 (ins 1 (5 : ins 7 [])))
= head(ins 6 (1 : 5 : ins 7 []))
= head(1 : ins 6 (5 : ins 7 []))
= 1
```

Lazy evaluation needs only linear time
because the sorted list is never constructed completely
```
min [6,1,7,5] = head(ins 6 (ins 1 (ins 7 (ins 5 []))))
= head(ins 6 (ins 1 (ins 7 (5 : []))))
= head(ins 6 (ins 1 (ins 7 [])))
= head(1 : ins 6 (5 : ins 7 []))
= 1
```

Lazy evaluation needs only linear time
although `insSort` is quadratic
because the sorted list is never constructed completely

Warning: this depends on the exact algorithm and does not work
so nicely with all sorting functions!

---

Maximum of a list

```
max = last . insSort
```

Complexity?

---

Example

A recursive definition
```
ones :: [Int]
ones = 1 : ones
```
A recursive definition

ones :: [Int]
ones = 1 : ones

that defines an infinite list of 1s:
ones

Example

But Haskell can compute with infinite lists, thanks to lazy evaluation:

ones = 1 : ones
ones = 1 : 1 : ones = ...
But Haskell can compute with infinite lists, thanks to lazy evaluation:

\[ \text{head \ ones} = \text{head \ (1 : \ ones)} = 1 \]

Remember:

Lazy evaluation evaluates an expression only as much as needed

Outermost reduction: \( \text{head \ ones} = \text{head \ (1 : \ ones)} = 1 \)

Haskell lists are never actually infinite but only potentially infinite
Haskell lists are never actually infinite but only potentially infinite
Lazy evaluation computes as much of the infinite list as needed

This is how partially evaluated lists are represented internally:
\[ 1 : 2 : 3 : \text{code pointer to compute rest} \]

In general: finite prefix followed by code pointer

Why (potentially) infinite lists?

- They come for free with lazy evaluation
- They increase modularity:
  list producer does not need to know how much of the list the consumer wants
Example: The sieve of Eratosthenes

1. Create the list 2, 3, 4, ...
2. Output the first value \( p \) in the list as a prime.
3. Delete all multiples of \( p \) from the list.
4. Goto step 2

2 3 4 5 6 7 8 9 10 11 12 ...

In Haskell:

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

sieve :: [Int] -> [Int]
sieve (p:xs) = p : sieve [x | x <- xs, x \( \not\equiv \) p]
```
In Haskell:

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

sieve :: [Int] -> [Int]
sieve (p:xs) = p : sieve [x | x <- xs, x \mod\ p /= 0]

Lazy evaluation:

primes = sieve [2..] = sieve (2:[3..]) = 2 : sieve [x | x <- [3..], x \mod\ 2 /= 0]
In Haskell:

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

sieve :: [Int] -> [Int]
sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p /= 0]
```

Lazy evaluation:

```haskell
primes = sieve [2..] = sieve (2:[3..])
= 2 : sieve [x | x <- [3..], x `mod` 2 /= 0]
= 2 : sieve [x | x <- 3:[4..], x `mod` 2 /= 0]
= 2 : sieve (3 : [x | x <- [4..], x `mod` 2 /= 0])
```

In Haskell:

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

sieve :: [Int] -> [Int]
sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p /= 0]
```

Lazy evaluation:

```haskell
primes = sieve [2..] = sieve (2:[3..])
= 2 : sieve [x | x <- [3..], x `mod` 2 /= 0]
= 2 : sieve [x | x <- 3:[4..], x `mod` 2 /= 0]
= 2 : sieve (3 : [x | x <- [4..], x `mod` 2 /= 0])
= 2 : 3 : sieve [x | x <- [3], x `mod` 2 /= 0]
    x `mod` 3 /= 0]
```

Modularity!
The first 10 primes:

> `take 10 primes`  
\[2,3,5,7,11,13,17,19,23,29]\n
The primes between 100 and 150:

> `takeWhile (<150) (dropWhile (<100) primes)`
\[101,103,107,109,113,127,131,137,139,149]\n
All twin primes:

> `[(p,q) | (p,q) <- zip primes (tail primes), p+2==q]`
> 101 `elem` primes
   True

> 102 `elem` primes
   nontermination

> 101 `elem` primes
   True

> 102 `elem` primes
   nontermination

prime n = n == head (dropWhile (<n) primes)
There is only one copy of primes

Every time part of primes needs to be evaluated
  Example: when computing take 5 primes
  primes is (invisibly!) updated to remember the evaluated part
  Example: primes = 2 : 3 : 5 : 7 : 11 : sieve ...

The next uses of primes are faster:
  Example: now primes !! 2 needs only 3 steps
Sharing!

There is only one copy of primes

Every time part of primes needs to be evaluated
  Example: when computing take 5 primes
primes is (invisibly!) updated to remember the evaluated part
  Example: primes = 2 : 3 : 5 : 7 : 11 : sieve ...
The next uses of primes are faster:
  Example: now primes !! 2 needs only 3 steps

Nothing special, just the automatic result of sharing

The list of Fibonacci numbers

Idea: 0 1 1 2 ...

+ 0 1 1 ...

= 0 1 2 3 ...
The list of Fibonacci numbers

Idea: \[0 \quad 1 \quad 2 \ldots\]  
+  \[0 \quad 1 \quad 1 \ldots\]  
=  \[0 \quad 1 \quad 2 \quad 3 \ldots\]  

From Prelude: `zipWith`

Example: `zipWith f [a_1, a_2, \ldots] [b_1, b_2, \ldots]`  
\[= [f \ a_1 \ a_2, \ f \ a_2 \ b_2, \ldots]\]

fibs :: [Integer]  
fibs = 0 : 1 :
The list of Fibonacci numbers

Idea: 0 1 1 2 ...
     + 0 1 1 ...
     = 0 1 2 3 ...

From Prelude: zipWith
Example: zipWith f [a1, a2, ...] [b1, b2, ...]
         = [f a1 a2, f a2 b2, ...]

fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

How about
fibs = 0 : 1 : [x+y | x <- fibs, y <- tail fibs]

Hamming numbers

Game tree

data Tree p v = Tree p v [Tree p v]

Separates move computation and valuation from move selection
Game tree

data Tree p v = Tree p v [Tree p v]

Separates move computation and valuation from move selection

Laziness:

- The game tree is computed incrementally, as much as is needed
- No part of the game tree is computed twice
- Supports infinitely broad and deep trees (useful??)

gameTree :: (p -> [p]) -> (p -> v) -> p -> Tree p v
Game tree

data Tree p v = Tree p v [Tree p v]

Separates move computation and valuation from move selection

Laziness:
- The game tree is computed incrementally, as much as is needed
- No part of the game tree is computed twice
- Supports infinitely broad and deep trees (useful??)

gameTree :: (p -> [p]) -> (p -> v) -> p -> Tree p v
gameTree next val = tree where
tree p = Tree p (val p) (map tree (next p))
chessTree = gameTree ...

minimax :: Ord v => Int -> Bool -> Tree p v -> v

minimax d player1 (Tree p v ts) =
    if d == 0 || null ts then v
    else let vs = map (minimax (d-1) (not player1)) ts
         in if player1 then maximum vs else minimum vs

> minimax 3 True chessTree
Generates chessTree up to level 3

> minimax 4 True chessTree
Needs to search 4 levels, but only level 4 needs to be generated