Lazy Evaluation

Lazy Evaluation

Auswertung von Ausdrücken: Faul (lazy)

- Nur falls wirklich benötigt

```
f x y = x + 42

f 3 (1 `div` 0)
⇒ 3 + 42
⇒ 45
```

- Abhängig von anderen Parametern

```
g x y n
  | n > 0       = (x + 42)
  | otherwise   = (y + 42)

   g (1 `div` 0) 3 (-1)
⇒+ 3 + 42
⇒  45
```

```
   g (1 `div` 0) 3 1
⇒+ (1 `div` 0) + 42
⇒+ ⊥
*** Exception: divide by zero
```

Lazy Evaluation

Auswertung von Ausdrücken: Faul (lazy)

- Strukturierte Daten: Nur soweit wie wirklich benötigt

```
   head [fak 3, 1 `div` 0]
⇒+ 6

   fst (True, 1 `div` 0)
⇒ True
```

- Duplizierte Argumente: maximal einmalig (sharing)

```
   f x y = x + y
   g x = f x x

   g (42*10)
⇒ f (42*10) (42*10)
⇒ (42*10) + (42*10)
⇒ 420 + 420 ⇒ 840
```

Notwendigkeit: Auswertung

Auswertung notwendig: Vergleichs-Operatoren, arithmetische Operatoren

- `x+y`, `x==y`, `x<y`

Auswertung notwendig: In Bedingungen

```
choose n m x y z
 | n==1 = x
 | m==1 = y
 | otherwise = z
```

- Auswertung von `n`: immer
- Auswertung von `x`: nur, falls `n==1` $\Rightarrow^+$ `True`
- Auswertung von `m`: nur, falls `n==1` $\Rightarrow^+$ `False`
- Auswertung von `y`: nur, falls `n==1` $\Rightarrow^+$ `False`
  und `m==1` $\Rightarrow^+$ `True`
- Auswertung von `z`: nur, falls `n==1` $\Rightarrow^+$ `False`
  und `m==1` $\Rightarrow^+$ `False`

Notwendigkeit: Auswertung

Auswertung notwendig: Beim Pattern-Matching

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

   f [1..4] [2..10]
⇒ f 1:[2..4] [2..10]
⇒ f 1:[2..4] 2:[3..10]
⇒ f 1:[2..4] 2:3:[4..10]
⇒ f 1+2 ⇒ 3
```

- Auswertung der Listen: So weit wie nötig, bis passendes Muster gematched

Lazyness: Beispiele

Lazy Boolsche Operatoren: Short-circuit-Auswertung

```
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && (and xs)

and [True,True,False,True,True]
```

```
⇒ True && (and [True,False,True,True])
⇒         (and [True,False,True,True])
⇒          True && (and [False,True,True])
⇒                  (and [False,True,True])
⇒                  (False && [True,True])
⇒ False
```

- Auswertung: nur bis zum ersten `False`

Lazyness: Beispiele

Lazy Boolsche Operatoren: Short-circuit-Auswertung

```
and :: [Bool] -> Bool
and = foldr (&&) True
```

```
   and [True,True,False,True,True]
⇒ foldr (&&) True [True,True,False,True,True])
⇒ True && (foldr (&&) True [True,False,True,True])
⇒         (foldr (&&) True [True,False,True,True])
⇒          True && (foldr (&&) True [False,True,True])
⇒                  (foldr (&&) True [False,True,True])
⇒                   False && (foldr (&&) True [False,True,True])
⇒ False
```

- Auswertung: nur bis zum ersten `False`

Lazyness: Beispiele

Lazy Boolsche Operatoren: Short-circuit-Auswertung

```
and :: [Bool] -> Bool
and = foldl (&&) True
```

```
   and [True,True,False,True,True]
⇒ foldl (&&)       True [True,True,False,True,True])
⇒ foldl (&&)      (True && True) [True,False,True,True]
⇒ foldl (&&)     ((True && True) && True) [False,True,True]
⇒ foldl (&&)    (((True && True) && True) && False) [True,True]
⇒ foldl (&&)   ((((True && True) && True) && False) && True) [True]
⇒ foldl (&&) (((((True && True) && True) && False) && True) && True) []
⇒ ((((True && True) && True) && False) && True) && True
⇒           (((True && True) && False) && True) && True
⇒                  ((True && False) && True) && True
⇒                         (False && True) && True
⇒                                False && True
⇒ False
```

- Auswertung: Auswertung der gesamten Listenstruktur!

Lazyness: Beispiele

Minimales Listenelement: `minimum = head . sort`

k minimale Elemente: `minimal k = (take k) . sort`

- Komplett ineffizient?
- Lazyness: Keine vollständige Sortierung!

Insertion-Sort:

```
ins x []          = [x]
ins x (y : ys)
    | x < y       = x : y : ys
    | otherwise   = y : ins x ys

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

Lazy Auswertung:

```
    sort [8,6,1,7,5]
⇒+ ins 8 (ins 6 (ins 1 (ins 7 (ins 5 []))))
⇒+ ins 8 (ins 6 (ins 1 (ins 7 [5])))
⇒+ ins 8 (ins 6 (ins 1 (5 : ins 7 [])))
⇒+ ins 8 (ins 6 (1 : (5 : ins 7 [])))
⇒+ ins 8 (1 : (ins 6 (5 : ins 7 [])))
⇒+ 1 : (ins 8 (ins 6 (5 : ins 7 [])))
```

minimum `[8,6,1,7,5]` $\Rightarrow^+$ `1`

Unendliche Listen: Streams

Unendliche Liste von Einsen:

```
ones = 1 : ones
```

ones $\Rightarrow^{+}$ `1:1:1:1:1:1:1:1:...`

Liste aller ungeraden Zahlen:

```
odds = 1 : map (+2) odds
```

odds $\Rightarrow^{+}$ `1:3:5:7:9:11:13:15:17:...`

`take 5 odds` $\Rightarrow^{+}$ `[1,3,5,7,9]`

```
head (tail odds)   ⇒ head (tail (1 : map (+2) odds))
                   ⇒ head (map (+2) odds)
                   ⇒ head (map (+2) (1 : map (+2) odds))
                   ⇒ head ((1+2) : (map (+2) ( map (+2) odds)))
                   ⇒ (1+2) ⇒ 3
```

Gemeinsames Schema: Funktionsiteration

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

Es gilt: `iterate f (f a) = map f (iterate f a)`.

$\Rightarrow$ `ones = iterate id 1`

$\Rightarrow$ `odds = iterate (+2) 1`

`iterate f x !! 23` führt „Schleife“ `f` 23 Mal aus.

Sieb des Eratosthenes mit Streams

Liste aller Primzahlen:

```
oddPrimes (p : ps) = p : (oddPrimes [p' | p' <- ps, p' `mod` p /= 0])
primes = 2 : oddPrimes (tail odds)
```

```
    primes
⇒   2:oddPrimes (tail odds)
⇒+  2:oddPrimes (tail (iterate (+2) 1))
⇒   2:oddPrimes (tail (1:iterate (+2) (1+2)))
⇒   2:oddPrimes (iterate (+2) (1+2))
⇒   2:oddPrimes (1+2:iterate (+2) ((1+2)+2))
⇒   2:1+2:oddPrimes [p' | p' <- iterate (+2) ((1+2)+2)
                        , p' `mod` (1+2) /= 0]
⇒   2:1+2:oddPrimes [p' | p' <- (1+2)+2:iterate (+2) (((1+2)+2)+2)
                        , p' `mod` (1+2) /= 0]
⇒+  2:3:oddPrimes (5:[p' | p' <- iterate (+2) (5+2), p' `mod` 3 /= 0])
⇒   2:3:5:oddPrimes [p'' | p'' <- [p' | p' <- iterate (+2) (5+2)
                                      , p' `mod` 3 /= 0]
                         , p'' `mod` 5 /= 0]
    ...
    take 5 primes ⇒+ [2,3,5,7,11]
```

Approximationen als Streams

Approximation von π: $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots = \sum_{i=0}^{\infty} (-1)^i \frac{1}{2i+1}$

Folge der Partialsummen von $\sum_{i=0}^{\infty} x_i$:

```
partialSums :: [Double] -> [Double]
partialSums (x : xs) = pSumsAcc xs x
  where
    pSumsAcc (x : xs) acc = acc : (pSumsAcc xs (x + acc))

approxPi = map (*4) (partialSums piSeq)
  where piSeq = zipWith (/) (iterate negate 1) odds
```

```
> take 5 approxPi
[4.0,
 2.666666666666667,
 3.466666666666667,
 2.895238095238095,
 3.339682539682540]
```

Beschleuniger von Approximationen

Euler-Beschleuniger: $(x_i)_i$ wird zu $(y_i)_i$, wobei $y_i = x_{i+2} - \frac{(x_{i+2}-x_{i+1})^2}{x_i-2x_{i+1}+x_{i+2}}$

```
eulerTransform (x : y : z : rest) =
  x - square (z - y) / (x - 2 * y + z) : eulerTransform (y : z : rest)

>take 5 (eulerTransform approxPi)
[3.166666666666667,
 3.133333333333337,
 3.145238095238096,
 3.13968253968254,
 3.142712842712435]
```

Liste der beschleunigten Beschleuniger: **`iterate`** `eulerTransform approxPi`

$\Rightarrow$ Superbeschleunigte Sequenz:

```
fastApproxPi = map head (iterate eulerTransform approxPi)

>take 5 fastApproxPi
[4.0,
 3.166666666666667,
 3.142105263157895,
 3.141599357319005,
 3.141592714033785]
```