

Funktionale Programmierung

05 Verzögerte Auswertung

Prof. Dr. David Sabel
Sommersemester 2025

Stand der Folien: 20. Mai 2025

- Informelle Betrachtung, wie Haskell Ausdrücke auswertet
- Haskell verwendet die verzögerte Auswertung.
- Synonyme: Bedarfsauswertung, nicht-strikte Auswertung mit Sharing, lazy evaluation, call-by-need evaluation
- Wesentlich Ideen:
 - Unterausdrücke werden nur dann ausgewertet, wenn sie für das Ergebnis benötigt werden
 - Mehrfache Auswertung desselben Ausdrucks wird vermieden.

- Ausdrücke werden als Graphen repräsentiert
- Ein unausgewerteter (Unter-)Ausdruck ist dann ein **Verweis** auf einen Teilgraphen und wird als **Thunk** bezeichnet.
- Bei der ersten Verwendung des Verweises wird der Ausdruck ausgewertet und durch sein Ergebniswert ersetzt, weitere Verwendungen des Verweises liefern sofort den Ergebniswert

Beispiel

Betrachte `squareAndIgnore (6+9) (1+2+3+4+5)` wobei

`squareAndIgnore x y = x*x`

- Nicht-strikte Auswertung:

`squareAndIgnore (6+9) (1+2+3+4+5)`

$\rightarrow (6+9) * (6+9)$

$\rightarrow 15 * (6+9)$

$\rightarrow 15 * 15$

$\rightarrow 225$

- Verzögerte Auswertung:

`(squareAndIgnore <1> <2>, {<1> ↦ 6+9, <2> ↦ 1+2+3+4+5})`

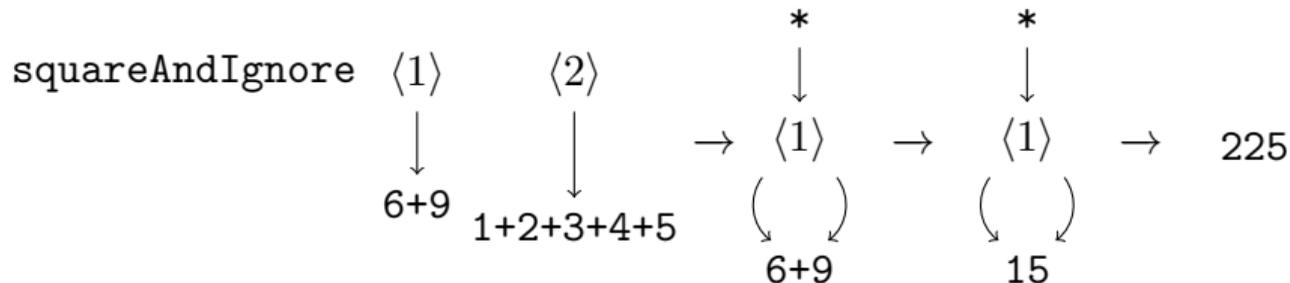
$\rightarrow (\langle 1 \rangle * \langle 1 \rangle, \{ \langle 1 \rangle \mapsto 6+9, \langle 2 \rangle \mapsto 1+2+3+4+5 \})$

$\rightarrow (15 * 15, \{ \langle 1 \rangle \mapsto 15, \langle 2 \rangle \mapsto 1+2+3+4+5 \})$

$\rightarrow (225, \{ \langle 1 \rangle \mapsto 15, \langle 2 \rangle \mapsto 1+2+3+4+5 \})$

Beispiel: Als Darstellung mit Graphen

Eine alternative Darstellung ist die Verwendung von Graphen:



Beispiel mit Trace

```
import Debug.Trace          -- von Verwendung wird abgeraten!!  
-- trace :: String -> a -> a -- Seiteneffekt: Textausgabe
```

```
foo :: Int -> Int -> Int -> Int  
foo x y z = y + y + z
```

```
z = foo (trace "first" 1)  
        (trace "second" 2)  
        (trace "third" 3)
```

Auswertung von z

- Strikt: "first" "second" "third" 7
- Nicht-strikt: "second" "second" "third" 7
- Verzögert: "second" "third" 7 .

Beispiel mit Divergenz

`foo x y z = if x < 0 then abs x else x+y`

Auswertung einer Anwendung `foo e1 e2 e3`:

- e_1 , e_2 und e_3 werden nicht angefasst, sondern in den Graphen (Heap) eingefügt und x , y und z sind Verweise auf die Ausdrücke e_1 , e_2 und e_3 .
- Im Anschluss beginnt die Auswertung des Rumpfs von `foo`.
- Die Auswertung des `if`-Ausdrucks erfordert ein Auswerten von `x<0` und dieses wiederum ein Auswerten des Arguments x .
- Daher wird e_1 ausgewertet und durch das Resultat ersetzt.
- Falls `x<0` gilt, wird der Wert von `abs x` zurückgegeben; weder e_2 noch e_3 werden ausgewertet.
- Falls `x<0` falsch ist, wird der Wert von `x+y` zurückgegeben; dies erfordert die Auswertung von e_2 . Daher wird e_3 in keinem Fall ausgewertet.

Der Ausdruck `foo 1 2 (1 'div' 0)` ist daher wohldefiniert.

Noch ein Beispiel

```
g x y = let z = x 'div' y
         in if x > y * y then x else z
t2 = g 5 0
```

- Wertet man t2 aus, so erhält man 5.
- Da der Wert von z nicht benötigt wird, kommt es nicht zur Division durch Null.
- Sequentialität der Auswertung (erst das, dann das) ist daher nicht gegeben, was manchmal verwirrend sein kann.

Potentiell unendliche Datenstrukturen

Lazy Evaluation ermöglicht „unendliche“ Datenstrukturen:

```
ones = 1 : ones      -- "unendliche Listen von 1en"  
twos = map (1+) ones -- "unendliche" Liste von 2en"  
nums = iterate (1+) 0 -- Liste der natuerlichen Zahlen
```

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

```
> take 10 nums  
[0,1,2,3,4,5,6,7,8,9]
```

Es wird immer nur soviel von der Datenstruktur ausgewertet, wie benötigt wird:

```
nums = iterate (1+) 0
```

```
> take 10 nums  
[0,1,2,3,4,5,6,7,8,9]
```

Lokale Funktionsdefinitionen mit let:

```
let  f1 x1,1 ... x1,n1 = e1
     f2 x2,1 ... x2,n2 = e2
     ...
     fm xm,1 ... xm,nm = em
in ...
```

Z.B.

```
f x y = let quadrat z = z*z in quadrat x + quadrat y
```

```
quadratfakultaet x = let quadrat z = z*z
                      fakq    0 = 1
                      fakq    x = (quadrat x)*fakq (x-1)
in fakq x
```

Bindungen der Form $var = e$ werden nur einmal berechnet (es sei denn, der Compiler optimiert dies durch inlining). Z.B.

```
verdoppelfak x =  
  let fak 0 = 1  
      fak x = x*fak (x-1)  
      fakx  = fak x  
  in fakx + fakx
```

Ein Aufruf `verdoppelfak 100` wird nur einmal `fak 100` berechnen.
Im Gegensatz dazu:

```
verdoppelfakLangsam x =  
  let fak 0 = 1  
      fak x = x*fak (x-1)  
  in fak x + fak x
```

Auf einer linken Seite kann auch ein Pattern stehen: z.B. $(a, b) = \dots$, damit kann man komfortabel auf einzelne Komponenten zugreifen.

Z.B. Summe und Produkt berechnen:

```
sumprod 1 = (1,1)
```

```
sumprod n =
```

```
  let (s',p') = sumprod (n-1)
```

```
  in (s'+n,p'*n)
```

Beispiel:

```
fib 0 = 0
```

```
fib 1 = 1
```

```
fib i = fib (i-1) + fib (i-2)
```

Mit Memoization:

```
-- Fibonacci mit Memoization
```

```
fibM i =
```

```
  let fibs = [0,1] ++ [fibs!!(i-1) + fibs!!(i-2) | i <- [2..]]
```

```
  in fibs!!i -- i-tes Element der Liste fibs
```

Memoization (2)

Variante nach let-over-lambda-shifting:

```
fibM' =  
  let fibs = [0,1] ++ [fibs!!(i-1) + fibs!!(i-2) | i <- [2..]]  
  in λi -> fibs!!i -- i-tes Element der Liste fibs
```

```
*Main> fibM' 20000
```

```
...
```

```
(7.27 secs, 29757856 bytes)
```

```
*Main> fibM' 20000
```

```
...
```

```
(0.06 secs, 2095340 bytes)
```

- Möglich im GHCi: Speicher inspizieren und beobachten, welche Teile einer Datenstruktur ausgewertet wurden
- Die beiden Kommandos hierzu sind `:print` und `:sprint` (die zweite Variante liefert eine vereinfachte Ansicht).
- Aufruf ist `:sprint [<name> ...]`. Speicherzustände der definierten Namen werden angezeigt, unausgewertete Thunks als `_`

```
ghci> let x = map even [1..22]; y = (map odd [10..20],x)
ghci> :sprint x y
x = _
y = (_,_)
ghci> take 3 $ drop 3 x
[True,False,True]
ghci> :sprint x y
x = _ : _ : _ : True : False : True : _
y = (_,_ : _ : _ : True : False : True : _)
```

Thunks im GHCi

Mit `:print`:

```
ghci> let x = map even [1..22]
ghci> let y = (map odd [10..20],x)
ghci> :print x y
x = (_t1::[Bool])
y = ((_t2::[Bool]),(_t3::[Bool]))
ghci> take 3 $ drop 3 x
[True,False,True]
ghci> :print x y
x = (_t4::Bool) : (_t5::Bool) : (_t6::Bool) : True : False : True :
    (_t7::[Bool])
y = ((_t8::[Bool]),(_t9::Bool) : (_t10::Bool) : (_t11::Bool) :
    True : False : True : (_t12::[Bool]))
```

Thunks im GHCi (3)

Man beachte, dass das Anzeigen nur für monomorphe Typen funktioniert, z.B. ergibt

```
ghci> let z = map (*2) [1..6] :: [Int]
```

```
ghci> take 3 $ drop 3 z
```

```
[8,10,12]
```

```
ghci> :sprint z
```

```
z = _ : _ : _ : 8 : 10 : 12 : _
```

```
ghci> let z = map (*2) [1..6]
```

```
ghci> take 3 $ drop 3 z
```

```
[8,10,12]
```

```
ghci> :sprint z
```

```
z = _
```

```
ghci> :print z
```

```
z = (_t30 :: (Num b, Enum b) => [b])
```

Thunks im GHCi (4)

Mit `:force` kann man die Auswertung von Thunks außerhalb der Reihenfolge erzwingen:

```
ghci> let x = map even [1..22]
ghci> :sprint x
x = _
ghci> take 3 $ drop 3 x
[True,False,True]
ghci> :print x
x = (_t1::Bool) : (_t2::Bool) : (_t3::Bool) : True : False : True :
    (_t4::[Bool])
ghci> :force _t2
_t2 = True
ghci> :print x
x = (_t5::Bool) : True : (_t6::Bool) : True : False : True :
    (_t7::[Bool])
```

```
factors :: Integral a => a -> [a]
factors n = filter (\m -> n `mod` m == 0) [2 .. (n - 1)]
```

```
isPrime :: Integral a => a -> Bool
isPrime n = n > 1 && null (factors n)
```

Die Funktion `factors` berechnet alle Faktoren einer Zahl. Die Berechnung von `isPrime` bricht sofort ab, sobald der erste Faktor gefunden wurde, trotz Verwendung von `factors` werden also nicht alle Faktoren berechnet!

Trennung von Daten und Kontrollfluss (2)

Weiteres Beispiel: Sieb des Erathostenes

```
primes :: [Integer]
```

```
primes = sieve [2..]
```

```
sieve :: [Integer] -> [Integer]
```

```
sieve (p:xs) = p : sieve (xs 'minus' [p,p+p..])
```

```
minus xs@(x:xt) ys@(y:yt) = case compare x y of
```

```
  LT -> x : minus xt ys
```

```
  EQ ->   minus xt yt
```

```
  GT ->   minus xs yt
```

```
(++) :: [a] -> [a] -> [a]
(++) []      ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

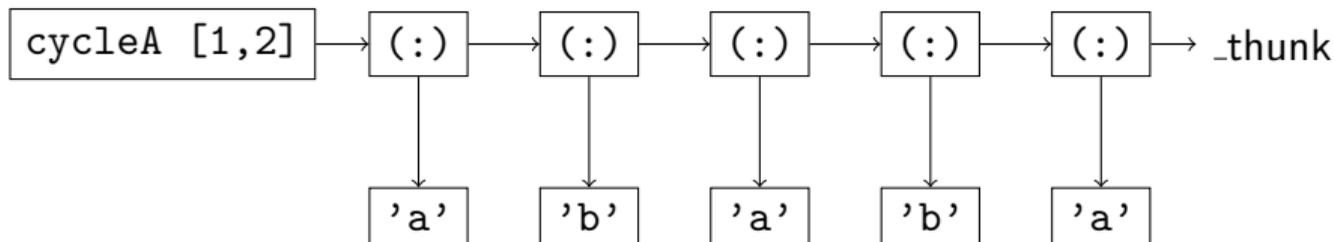
```
cycleA xs = xs ++ cycleA xs
```

```
cycleB xs = xs' where xs' = xs++xs'
```

- `cycleA l` verbraucht potenziell unendlich viel Speicher, bzw. genauer so viel Speicherzellen, wie Elemente von `cycleA l` gelesen werden.
- Für eine Liste `l` mit Länge `n` verbraucht `cycleB` jedoch nur maximal `n` Speicherzellen.

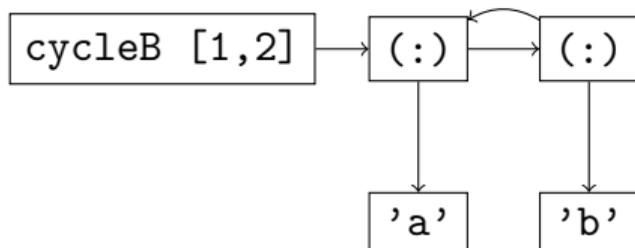
Zirkularität (2)

Call-by-Need-Auswertung von `cycleA [1,2]` nach Anforderung einiger Elemente:



Zirkularität (3)

Call-by-Need-Auswertung von `cycleB [1,2]` nach Anforderung einiger Elemente:



- Bereits gesehen: `seq` und `$!-`Operator
- Dabei wird nur bis zur WHNF ausgewertet
- Tiefer auswerten: `Control.DeepSeq`

Spracherweiterung `BangPatterns`:

Patterns, welche mit `!` beginnen, müssen immer zuerst ausgewertet werden:

```
stack exec -- ghci -XBangPatterns
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
ghci> let foo (x,!y) = [x,y]
ghci> drop 1 $ foo (undefined,2)
[2]
ghci> take 1 $ foo (2,undefined)
*** Exception: Prelude.undefined
```

Der GHC erlaubt auch die Deklaration von strikten Datentypen:

```
data FaulesPaar   = FP Integer Bool deriving Show
data StriktesPaar = SP !Integer !Bool deriving Show
```

```
ghci> let (FP u v) = FP undefined True in v
True
ghci> let (SP u v) = SP undefined True in v
*** Exception: Prelude.undefined
```

- Der Konstruktor `SP` wird hier immer mit ausgewerteten Argumenten abgespeichert;
- die Argumente von `FP` können dagegen thunks sein.

Strikte Datentypen (2)

Die Spracherweiterung `StrictData` macht alle Datentypen strikt; lazy Argumente erhält man dann mit einer Tilde:

```
stack exec -- ghci -XStrictData
GHCi, version 8.6.5: http://www.haskell.org/ghc/ :? for help
ghci> data Paar = P Integer Bool
ghci> let (P a b) = P undefined True in b
*** Exception: Prelude.undefined
ghci> data FaulesPaar = FP ~Integer ~Bool
ghci> let (FP a b) = FP undefined True in b
True
```

Lazy Pattern

Lazy Pattern-Matches: Versehe das Pattern mit ~:

```
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
ghci> let bar (x,y) = 42
ghci> bar undefined
*** Exception: Prelude.undefined
ghci> let baz ~(x,y) = 42
ghci> baz undefined
42
```

- Lazy Patterns matchen immer (sind daher *irrefutable*)
- Diese Pattern immer nur als letzte Möglichkeit verwenden.

Falsch:

```
mylength ~[] = 0
mylength (_:xs) = 1 + mylength xs
```

- Beachte: Pattern auf linken Seiten von `let`-Ausdrücken sind stets lazy.

Lazy-Pattern (2)

Die Verwendung der Lazy-Pattern kann man sich so vorstellen: Aus

```
quu ~(x,y) = x+y
```

wird

```
quu p = let (x,y) = p in x+y
```

was wiederum übersetzt wird in

```
quu p = let x = case p of (x,_) -> x  
           y = case p of (_,y) -> y  
           in (x+y)
```

oder kürzer geschrieben:

```
quu p = (fst p) +(snd p)
```

Beispiel

Funktion splitAt:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n ls
  | n <= 0      = ([], ls)
splitAt _ []   = ([], [])
splitAt n (y:ys) =
  case splitAt (n-1) ys of
    ~(prefix, suffix) -> (y : prefix, suffix)
```

Durch den Lazy-Pattern-Match steht der erste Teil des Ergebnis-Paares sofort zur Verfügung. Z.B. liefert

```
head $ fst $ splitAt 10000000 (repeat 'a')
```

sofort 'a' zurück.

Bei Verwendung eines normalen Pattern Matches, dauert dies viel länger (da erst 10 Millionen rekursive Aufrufe berechnet werden)