Programmierung und Modellierung mit Haskell

Teil 10: Laziness & Striktheit

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

20. Juni 2018

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 10: Laziness & Striktheit

Substitutionsmodell

Das Substitutionsmodell (Folien 3.5ff.) erklärt das Verhalten von rein funktionalen Sprachen, solange keine Fehler auftreten (und das Programm terminiert).

- Man wählt einen beliebigen Teilausdruck und wertet diesen gemäß den geltenden Rechenregeln aus.
- Eine Funktionsanwendung wird durch den definierenden Rumpf ersetzt. Dabei werden die formalen Parameter im Rumpf durch die Argumentausdrücke ersetzt ("substitutiert").
- Dies wiederholt man, bis keine auswertbaren Teilausdrücke mehr vorhanden sind.

Es gibt *unterschiedliche* Strategien, den als nächstes auszuwertenden Teilausdruck auszuwählen.

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

```
bar 1 2 (succ 3)
```

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

`bar 1 2 (succ 3)` $\rightsquigarrow$ `bar 1 2 (3+1)`

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

`bar 1 2 (succ 3)` $\rightsquigarrow$ `bar 1 2 (3+1)` $\rightsquigarrow$ `bar 1 2 4`

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

`bar 1 2 (succ 3)` $\leadsto$ `bar 1 2 (3+1)` $\leadsto$ `bar 1 2 4`
$\leadsto$ `if 1 < succ 1 then 4 else 1+2`

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

`bar 1 2 (succ 3)` $\rightsquigarrow$ `bar 1 2 (3+1)` $\rightsquigarrow$ `bar 1 2 4`
$\rightsquigarrow$ `if 1 < succ 1 then 4 else 1+2`
$\rightsquigarrow$ `if 1 < 1+1 then 4 else 1+2`

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

`bar 1 2 (succ 3)` $\rightsquigarrow$ `bar 1 2 (3+1)` $\rightsquigarrow$ `bar 1 2 4`
$\rightsquigarrow$ `if 1 < succ 1 then 4 else 1+2`
$\rightsquigarrow$ `if 1 < 1+1 then 4 else 1+2`
$\rightsquigarrow$ `if 1 < 2 then 4 else 1+2`

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

`bar 1 2 (succ 3)` $\rightsquigarrow$ `bar 1 2 (3+1)` $\rightsquigarrow$ `bar 1 2 4`
$\rightsquigarrow$ `if 1 < succ 1 then 4 else 1+2`
$\rightsquigarrow$ `if 1 < 1+1 then 4 else 1+2`
$\rightsquigarrow$ `if 1 < 2 then 4 else 1+2` $\rightsquigarrow$ `if True then 4 else 1+2`

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

`bar 1 2 (succ 3)` $\rightsquigarrow$ `bar 1 2 (3+1)` $\rightsquigarrow$ `bar 1 2 4`
$\rightsquigarrow$ `if 1 < succ 1 then 4 else 1+2`
$\rightsquigarrow$ `if 1 < 1+1 then 4 else 1+2`
$\rightsquigarrow$ `if 1 < 2 then 4 else 1+2` $\rightsquigarrow$ `if True then 4 else 1+2`
$\rightsquigarrow$ `if True then 4 else 3`

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

`bar 1 2 (succ 3)` $\rightsquigarrow$ `bar 1 2 (3+1)` $\rightsquigarrow$ `bar 1 2 4`
$\rightsquigarrow$ `if 1 < succ 1 then 4 else 1+2`
$\rightsquigarrow$ `if 1 < 1+1 then 4 else 1+2`
$\rightsquigarrow$ `if 1 < 2 then 4 else 1+2` $\rightsquigarrow$ `if True then 4 else 1+2`
$\rightsquigarrow$ `if True then 4 else 3` $\rightsquigarrow$ `4`

Es wäre aber auch möglich so auszuwerten:

`bar 1 2 (succ 3)`

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

`bar 1 2 (succ 3)` $\leadsto$ `bar 1 2 (3+1)` $\leadsto$ `bar 1 2 4`
$\leadsto$ `if 1 < succ 1 then 4 else 1+2`
$\leadsto$ `if 1 < 1+1 then 4 else 1+2`
$\leadsto$ `if 1 < 2 then 4 else 1+2` $\leadsto$ `if True then 4 else 1+2`
$\leadsto$ `if True then 4 else 3` $\leadsto$ `4`

Es wäre aber auch möglich so auszuwerten:

`bar 1 2 (succ 3)` $\leadsto$ `if 1 < succ 1 then succ 3 else 1+2`

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

```
bar 1 2 (succ 3) ⇝ bar 1 2 (3+1) ⇝ bar 1 2 4
⇝ if 1 < succ 1 then 4 else 1+2
⇝ if 1 < 1+1 then 4 else 1+2
⇝ if 1 < 2 then 4 else 1+2 ⇝ if True then 4 else 1+2
⇝ if True then 4 else 3 ⇝ 4
```

Es wäre aber auch möglich so auszuwerten:

```
bar 1 2 (succ 3) ⇝ if 1 < succ 1 then succ 3 else 1+2
⇝ if 1 < 1+1 then succ 3 else 1+2
```

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

```
bar 1 2 (succ 3) ⇝ bar 1 2 (3+1) ⇝ bar 1 2 4
⇝ if 1 < succ 1 then 4 else 1+2
⇝ if 1 < 1+1 then 4 else 1+2
⇝ if 1 < 2 then 4 else 1+2 ⇝ if True then 4 else 1+2
⇝ if True then 4 else 3 ⇝ 4
```

Es wäre aber auch möglich so auszuwerten:

```
bar 1 2 (succ 3) ⇝ if 1 < succ 1 then succ 3 else 1+2
⇝ if 1 < 1+1 then succ 3 else 1+2
⇝ if 1 < 2 then succ 3 else 1+2
```

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

```
bar 1 2 (succ 3) ⇝ bar 1 2 (3+1) ⇝ bar 1 2 4
⇝ if 1 < succ 1 then 4 else 1+2
⇝ if 1 < 1+1 then 4 else 1+2
⇝ if 1 < 2 then 4 else 1+2 ⇝ if True then 4 else 1+2
⇝ if True then 4 else 3 ⇝ 4
```

Es wäre aber auch möglich so auszuwerten:

```
bar 1 2 (succ 3) ⇝ if 1 < succ 1 then succ 3 else 1+2
⇝ if 1 < 1+1 then succ 3 else 1+2
⇝ if 1 < 2 then succ 3 else 1+2
⇝ if True then succ 3 else 1+2
```

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

```
bar 1 2 (succ 3) ⇝ bar 1 2 (3+1) ⇝ bar 1 2 4
⇝ if 1 < succ 1 then 4 else 1+2
⇝ if 1 < 1+1 then 4 else 1+2
⇝ if 1 < 2 then 4 else 1+2 ⇝ if True then 4 else 1+2
⇝ if True then 4 else 3 ⇝ 4
```

Es wäre aber auch möglich so auszuwerten:

```
bar 1 2 (succ 3) ⇝ if 1 < succ 1 then succ 3 else 1+2
⇝ if 1 < 1+1 then succ 3 else 1+2
⇝ if 1 < 2 then succ 3 else 1+2
⇝ if True then succ 3 else 1+2 ⇝ succ 3
```

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

```
bar 1 2 (succ 3) ⇝ bar 1 2 (3+1) ⇝ bar 1 2 4
⇝ if 1 < succ 1 then 4 else 1+2
⇝ if 1 < 1+1 then 4 else 1+2
⇝ if 1 < 2 then 4 else 1+2 ⇝ if True then 4 else 1+2
⇝ if True then 4 else 3 ⇝ 4
```

Es wäre aber auch möglich so auszuwerten:

```
bar 1 2 (succ 3) ⇝ if 1 < succ 1 then succ 3 else 1+2
⇝ if 1 < 1+1 then succ 3 else 1+2
⇝ if 1 < 2 then succ 3 else 1+2
⇝ if True then succ 3 else 1+2 ⇝ succ 3 ⇝ 3+1
```

Beispiel

```
succ x = x + 1
bar x y z = if 1 < succ x then z else x+y
```

Beispielauswertung des Ausdrucks `bar 1 2 (succ 3)`:

```
bar 1 2 (succ 3) ⇝ bar 1 2 (3+1) ⇝ bar 1 2 4
⇝ if 1 < succ 1 then 4 else 1+2
⇝ if 1 < 1+1 then 4 else 1+2
⇝ if 1 < 2 then 4 else 1+2 ⇝ if True then 4 else 1+2
⇝ if True then 4 else 3 ⇝ 4
```

Es wäre aber auch möglich so auszuwerten:

```
bar 1 2 (succ 3) ⇝ if 1 < succ 1 then succ 3 else 1+2
⇝ if 1 < 1+1 then succ 3 else 1+2
⇝ if 1 < 2 then succ 3 else 1+2
⇝ if True then succ 3 else 1+2 ⇝ succ 3 ⇝ 3+1 ⇝ 4
```

Bemerke: Anzahl Auswerteschritte kann sich unterscheiden!

Reduzierbare Teilausdrücke

Ein **Redex** eines Programmausdrucks ist ein Teilausdruck davon, welcher weiter ausgewertet werden kann.

Beispiel

$$\underline{\texttt{if True then}}\ 4\ \texttt{else}\ \underline{1+2}$$

Dieser Ausdruck enthält zwei Redexe:

1. Fallunterscheidung (Konditional) ausführen
2. Numerische Rechnung ausführen

Verschachtelte Redexe

Im Beispiel ist der Redex der numerischen Rechnung selbst ein Teilausdruck des Konditional-Redex. Wir unterscheiden:

- **Innerster Redex** enthält keinen weiteren Redex als Teilausdruck
- **Äußerster Redex** ist in keinem anderen Redex enthalten.

Der Begriff "Redex" kommt von "reduzieren", auch wenn die Auswertung machmal einen Ausdruck größer machen kann!

Reduzierbare Teilausdrücke (2)

Verschachtelte Redexe bieten manchmal eine Wahl, z.B. Ausdruck `if 1>2 then 4+1 else 5*2` hat drei Redexe

Der Ausdruck `if 1>2 then 4 else 5` hat aber nur einen Redex, da Konditional erst auswertbar ist, wenn Bedingung ein Wert ist!

Keine Reduktion unter einem Lambda

Wir verbieten Reduktion innerhalb eines Funktionsrumpfes, z.B. der Ausdruck `\x -> 1 + 2` enthält keinen Redex.

Hinweis: Reduktion in Funktionsrümpfen kann durchaus erlaubt werden. Wir vereinfachen hier, weil Haskell es auch so macht. Philosophie: Funktionen kann man nur anwenden, aber nicht hineinschauen, d.h. Funktionsrumpf muss erst eingesetzt werden.

Auswertungsstrategie

Eine **Auswertungsstrategie** beschreibt, wie ein Programmausdruck auszuwerten ist.

1. In welcher Reihenfolge werden die Teilausdrücke bearbeitet?
2. Werden Funktionsargumente ausgewertet, bevor diese in den Funktionsrumpf eingesetzt werden?

Achtung

- Bei Effekten (Wertzuweisung, Ausnahmen, Ein/Ausgabe) ist die Auswertungsstrategie signifikant.

- Bei *terminierenden* Programmen ohne Seiteneffekten ist die Auswertereihenfolge egal falls das Programm terminiert

Seiteneffekte vs. Auswertereihenfolge

In imperativen Sprachen ist die Auswertereihenfolge wichtig.

Beispiel

Der imperative Programmausdruck der Zuweisung $x := e$
weist als Seiteneffekt der Variablen x den Wert des Ausdrucks e zu,
und wertet selbst zu diesem Wert aus.

Am Anfang gelte `x=0`.

$$\underline{\texttt{x}}\ \texttt{+}\ \texttt{(x:=1)} \leadsto \texttt{0}\ \texttt{+}\ \underline{\texttt{(x:=1)}} \leadsto \texttt{0}\ \underline{\texttt{+}}\ \texttt{1} \leadsto \texttt{1}$$

Alternative Auswertereihenfolge führt zu anderen Ergebnis:

$$\texttt{x}\ \texttt{+}\ \underline{\texttt{(x:=1)}} \leadsto \underline{\texttt{x}}\ \texttt{+}\ \texttt{1} \leadsto \texttt{1}\ \underline{\texttt{+}}\ \texttt{1} \leadsto \texttt{2}$$

$\Rightarrow$ Neben dem *was* man berechnen will, muss man in imperativen Sprachen darauf achten, *wie* ausgewertet wird!

Monaden vs. Auswertereihenfolge

Fragen

- Wenn in Haskell die Auswertereihenfolge generell egal ist, wieso funktionieren dann Monaden?
- Warum beachtet die DO-Notation die Auswertereihenfolge?

Antwort

Erfolgt ganz automatisch wegen des Durchfädeln des Kontexts!

Zum Beispiel kann ein if-then-else, dessen Bedingung von dem Kontext der Monade abhängt, erst dann ausgewertet werden, wenn diese Bedingung und damit der Kontext ausgewertet wurden.

Die Auswertereihenfolge für Haskell bleibt weiterhin beliebig: es wird immer der gleiche Wert berechnet, egal wie ausgewertet wird!

Termination vs. Auswertereihenfolge

Egal wie ausgewertet wird, Haskell liefert den gleichen *Wert*
Manchmal kommt aber eben gar kein Wert heraus:

Beispiel

```
bar x y z = if 1 < succ x then z else x+y
```

```
bar 0 2 (3 `div` 0) ⇝ *** Exception: divide by zero
```

Es wäre aber auch möglich so auszuwerten:

```
bar 0 2 (3 `div` 0) ⇝
  if 1 < succ 0 then (3 `div` 0) else 0+2 ⇝
  if 1 < 1 then (3 `div` 0) else 0+2 ⇝
  if False then (3 `div` 0) else 0+2 ⇝ 0+2 ⇝ 2
```

⇒ Auswertestrategie kann über Termination entscheiden!

Termination vs. Auswertereihenfolge

Egal wie ausgewertet wird, Haskell liefert den gleichen *Wert*
Manchmal kommt aber eben gar kein Wert heraus:

Beispiel

```
bar x y z = if 1 < succ x then z else x+y
```

```
bar 0 2 (3 `div` 0) ⇝ *** Exception: divide by zero
```

Es wäre aber auch möglich so auszuwerten:

```
bar 0 2 (3 `div` 0) ⇝
  if 1 < succ 0 then (3 `div` 0) else 0+2 ⇝
  if 1 < 1 then (3 `div` 0) else 0+2 ⇝
  if False then (3 `div` 0) else 0+2 ⇝ 0+2 ⇝ 2
```

⇒ Auswertestrategie kann über Termination entscheiden!

Call-By-Name

Die Auswertestrategie **Call-By-Name** ist festgelegt durch:

- Argumente unausgewertet in den Funktionsrumpf einsetzen!
- Nicht in Funktiosrümpfen (unter Lambda) reduzieren.
- Nicht in den Zweigen einer Fallunterscheidung reduzieren.
  (also `if` oder Pattern-Match, z.B. `case`)

Vorteile Unbenutzte Argumente werden nicht ausgewertet!
Terminiert immer fehlerfrei, wenn eine andere Auswertestrategie das auch könnte.

Nachteil Ineffizient, falls Argumente mehrfach benötigt werden.

Beispiel

$$\frac{\texttt{(\textbackslash x -> \textbackslash y -> (y,y)) (1 `div` 0) (succ 1)} \leadsto \quad \frac{\texttt{(\textbackslash y -> (y,y)) (succ 1)} \leadsto}{\texttt{(succ 1, succ 1)} \leadsto \texttt{(2, succ 1)} \leadsto \texttt{(2,2)}}}$$

Call-By-Value

Die Auswertestrategie **Call-By-Value** ist festgelegt durch:

- Argumente nur vollständig ausgewertet einsetzen!
- Nicht in Funktiosrümpfen (unter Lambda) reduzieren.
- Nicht in den Zweigen einer Fallunterscheidung reduzieren.
  (also `if` oder Pattern-Match, z.B. `case`)

Vorteil Jedes Argument wird nur einmal ausgewertet!

Nachteil Auswertung scheitert auch an *unbenötigten*, fehlerhaften oder nicht-terminierenden Teilausdrücken.

Beispiele

```
(\x -> \y -> (y,y)) (1 `div` 0) (succ 1)
  ⇝ *** Exception: divide by zero
```

```
(\x -> \y -> (y,y)) 44 (succ 1) ⇝
  (\x -> \y -> (y,y)) 44 2 ⇝
  (\y -> (y,y)) 2 ⇝ (2, 2)
```

Call-By-Value

Die Auswertestrategie **Call-By-Value** ist festgelegt durch:

- Argumente nur vollständig ausgewertet einsetzen!
- Nicht in Funktiosrümpfen (unter Lambda) reduzieren.
- Nicht in den Zweigen einer Fallunterscheidung reduzieren.
  (also `if` oder Pattern-Match, z.B. `case`)

Vorteil Jedes Argument wird nur einmal ausgewertet!

Nachteil Auswertung scheitert auch an *unbenötigten*, fehlerhaften oder nicht-terminierenden Teilausdrücken.

Beispiele

```
(\x -> \y -> (y,y)) (1 `div` 0) (succ 1)
  ⇝ *** Exception: divide by zero
```

```
(\x -> \y -> (y,y)) 44 (succ 1) ⇝
  (\x -> \y -> (y,y)) 44 2 ⇝
  (\y -> (y,y)) 2 ⇝ (2, 2)
```

Call-By-Need

Haskell verwendet **Bedarfsauswertung** (engl. **Call-by-need**), oft implementiert durch **verzögerte Auswertung** (engl. **lazy evaluation**):

Idee Anstelle eines Redex wird ein Verweis (engl. Pointer) eingesetzt. Benötigte Ausdrücke werden *einmal* ausgewertet; damit wird die Speicherstelle des Verweises aktualisiert.
Bei erneuter Verwendung ist der Wert dank Verweis sofort vorhanden!

Nur möglich, wenn die Auswertereihenfolge egal ist!

Ein unausgewerteter Ausdruck im Speicher wird **Thunk** genannt.

Vorteile
- Terminierung wie bei Call-By-Name
- Effizienz (fast) wie bei Call-By-Value

Nachteile
- Verweise kosten auch etwas Speicher und Zeit
- **Sequentialität** der Auswertung (erst dieses, dann jenes) geht verloren, was verwirren kann.

Vergleich Auswertungsstrategien

Ausgabe eines Programms mit Seiteneffekten, Auswertung von `z`:

```
import Debug.Trace              -- für "Print-Debugging" von
trace :: String -> a -> a       -- Verwendung wird abgeraten!

foo x y z = y + y + z

z = foo (trace "first" 1)               -- wird 0x verwendet
        (trace "second" 2)              -- wird 2x verwendet
        (trace "third" 3)               -- wird 1x verwendet
```

Call-By-Value: `"first second third" 7`
Call-By-Name: `"second second third" 7`
Lazy Evaluation: `"second third" 7`

Vergleich Auswertungsstrategien

Ausgabe eines Programms mit Seiteneffekten, Auswertung von `z`:

```
impor                                          ng" von
trace                                          geraten!

foo x

z = f                                          rwendet
                                               rwendet
                                               rwendet
```

Hinweise:

- Haskell verwendet immer Lazy Evaluation!

Diese Folie zeigt nur was passieren würde, *wenn* es anders wäre – so wie in anderen Programiersprachen.

- Das Ergebnis der Berechnung ist immer 7;

unabhängig von der Auswertestrategie! Die Ausgabe von `first`, `second`, `third` ist nur ein Seiteneffekt!

Call-By-Value:	`"first second third" 7`
Call-By-Name:	`"second second third" 7`
Lazy Evaluation:	`"second third" 7`

Lazy Evaluation: Beispiel

Beispiel für Bedarfsauswertung:

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

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von `x<0` und dieses wiederum ein Auswerten von Argument `x`.
- Falls `x<0` wahr ist, wird Verweis auf `abs x` zurückgegeben; weder `y` noch `z` werden deshalb hier ausgewertet.
- Falls `x<0` falsch ist, wird Verweis auf `x+y` zurückgegeben; dies könnte später noch die Auswertung von `y` erfordern.
- `z` wird in keinem Fall hier ausgewertet.

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

Potentiell unendliche Datenstrukturen

Lazy Evaluation ermöglicht "unendliche" Datenstrukturen:

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

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

take :: Int -> [a] -> [a]
take n (x:xs) | n > 0 =  x : take (n-1) xs
take _   _            =  []

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

In endlicher Zeit und mit begrenztem Speicher kann man sich natürlich nur endliche Teile davon anschauen.

Potentiell unendliche Datenstrukturen

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]
```

$\Rightarrow$ Kontrollfluss unabhängig von Daten!

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
take 3 nums  ↝  take 3 (iterate (+1) 0)
↝ take 3 (0 : iterate (+1) (0+1))
↝ 0 : take (3-1) (iterate (+1) (0+1))
↝ 0 : take  2    (iterate (+1) (0+1))
↝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
↝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
↝ 0 : (0+1) : take  1    (iterate (+1) (0+1+1))
↝ 0 : (0+1) : take  1    ((0+1+1): iterate (+1) (0+1+1+1))
↝ 0 : (0+1) : (0+1+1) : take (1-1) (iterate (+1) (0+1+1+1))
↝ 0 : (0+1) : (0+1+1) : take  0    (iterate (+1) (0+1+1+1))
↝ 0 : (0+1) : (0+1+1) : []
```

Bemerke: Im Speicher verweist `nums` danach ebenfalls auf

```
0 : (0+1) : (0+1+1) : (iterate (+1) (0+1+1+1))
```

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
```

Um diesen Ausdruck jetzt weiter auszuwerten, muss die Fallunterscheidung von `take` durchgeführt werden.
Dazu müssen folgende Fragen geklärt werden:

- Ist `n` größer 0?
- Ist die Liste nicht-leer?

Für letztere Frage muss jetzt das zweite Argument weiter ausgewertet werden; durch Einsetzen des Funktionsrumpfes von `iterate`.

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums   ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
```

Um diesen Ausdruck jetzt weiter auszuwerten, muss die Fallunterscheidung von `take` durchgeführt werden.
Dazu müssen folgende Fragen geklärt werden:

- Ist `n` größer 0?
- Ist die Liste nicht-leer?

Für letztere Frage muss jetzt das zweite Argument weiter ausgewertet werden; durch Einsetzen des Funktionsrumpfes von `iterate`.

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums   ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝
⇝
⇝
⇝
```

Die Fallunterscheidung wurde durchgeführt und der entsprechende Funktionsrumpf von `take` wurde eingesetzt.

Falls die Auswertung fortgesetzt werden soll, so muss jetzt erneut die Fallunterscheidung von `take` durchgeführt werden: Ist `n>0` und ist die Liste leer?
Zuerst wird die Subtraktion durchgeführt, dann wieder `iterate` eingesetzt.

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums   ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝
⇝
⇝
⇝
```

Die Fallunterscheidung wurde durchgeführt und der entsprechende Funktionsrumpf von `take` wurde eingesetzt.

Falls die Auswertung fortgesetzt werden soll, so muss jetzt erneut die Fallunterscheidung von `take` durchgeführt werden: Ist `n>0` und ist die Liste leer?
Zuerst wird die Subtraktion durchgeführt, dann wieder `iterate` eingesetzt.

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝
⇝
⇝
⇝
```

Die Fallunterscheidung wurde durchgeführt und der entsprechende Funktionsrumpf von `take` wurde eingesetzt.

Falls die Auswertung fortgesetzt werden soll, so muss jetzt erneut die Fallunterscheidung von `take` durchgeführt werden: Ist `n>0` und ist die Liste leer?
Zuerst wird die Subtraktion durchgeführt, dann wieder `iterate` eingesetzt.

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    (iterate (+1) (0+1+1))
⇝
⇝
⇝
⇝
```

Fallunterscheidung von `take` konnte jetzt durchgeführt werden!

Falls die Auswertung weiter fortgesetzt werden soll, läuft alles wieder analog weiter...

Bemerke: Im Speicher verweist `nums` danach ebenfalls auf

```
0 : (0+1) : (0+1+1) : (iterate (+1) (0+1+1+1))
```

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    ((0+1+1): iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take (1-1) (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take  0    (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : []
```

Bemerke: Im Speicher verweist nums danach ebenfalls auf

```
0 : (0+1) : (0+1+1) : (iterate (+1) (0+1+1+1))
```

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    ((0+1+1): iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take (1-1) (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take  0    (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : []
```

Bemerke: Im Speicher verweist `nums` danach ebenfalls auf

```
0 : (0+1) : (0+1+1) : (iterate (+1) (0+1+1+1))
```

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    ((0+1+1): iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take (1-1) (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take  0    (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : []
```

Bemerke: Im Speicher verweist `nums` danach ebenfalls auf

```
0 : (0+1) : (0+1+1) : (iterate (+1) (0+1+1+1))
```

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
   take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    (iterate (+1) (0+1+1))
⇝ 0 : (
⇝ 0 : (0+1) : (0+1+1) : take (1-1) (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take  0    (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : []
```

Ausfaltung von `take` nimmt dieses Mal den anderen Fall!

Bemerke: Im Speicher verweist `nums` danach ebenfalls auf

```
0 : (0+1) : (0+1+1) : (iterate (+1) (0+1+1+1))
```

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
   take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    (iterate (+1) (0+1+1))
⇝ 0 : (
⇝ 0 : (0+1) : (0+1+1) : take (1-1) (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take  0    (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : []
```

Ausfaltung von `take` nimmt dieses Mal den anderen Fall!

Bemerke: Im Speicher verweist `nums` danach ebenfalls auf

```
0 : (0+1) : (0+1+1) : (iterate (+1) (0+1+1+1))
```

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    ((0+1+1): iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take (1-1) (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take  0    (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : []
```

Bemerke: Im Speicher verweist `nums` danach ebenfalls auf

```
0 : (0+1) : (0+1+1) : (iterate (+1) (0+1+1+1))
```

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    ((0+1+1): iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take (1-1) (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take  0    (iterate (+1) (0+1+1+1))
⇝ 0 : 1 : (0+1+1) : []
```

Bemerke: Im Speicher verweist `nums` danach ebenfalls auf

```
0 : 1 : (0+1+1) : (iterate (+1) (0+1+1+1))
```

Lazy Evaluation: Beispiel

```
take n (x:xs)|n>0 = x : take (n-1) xs
take _   _        = []
```

```
nums = iterate (1+) 0
iterate f x = x : iterate f (f x)
```

```
  take 3 nums  ⇝  take 3 (iterate (+1) 0)
⇝ take 3 (0 : iterate (+1) (0+1))
⇝ 0 : take (3-1) (iterate (+1) (0+1))
⇝ 0 : take  2    (iterate (+1) (0+1))
⇝ 0 : take  2    ((0+1) : iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take (2-1) (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    (iterate (+1) (0+1+1))
⇝ 0 : (0+1) : take  1    ((0+1+1): iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take (1-1) (iterate (+1) (0+1+1+1))
⇝ 0 : (0+1) : (0+1+1) : take  0    (iterate (+1) (0+1+1+1))
⇝ 0 : 1 : 2 : []
```

Bemerke: Im Speicher verweist `nums` danach ebenfalls auf

```
0 : 1 : 2 : (iterate (+1) (0+1+1+1))
```

Beispiel: Sieb des Erathostenes

Unendliche Liste aller Primzahlen:

```
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) | LT == cmp = x : minus xt ys
                          | EQ == cmp =     minus xt yt
                          | GT == cmp =     minus xs yt
                         where cmp = compare x y
```

- Wir müssen uns nur um die Daten kümmern, also *wie* wir die Primzahlen berechnen.
- Kontrolle über Anzahl der benötigten Primzahlen erfolgt später!
- Effizienz: siehe Haskell-Wiki: Prime Numbers

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1              ⇝
sumWithL   [3,4]   (1+2)           ⇝
sumWithL     [4]  ((1+2)+3)        ⇝
sumWithL      [] (((1+2)+3)+4)     ⇝
                 ((1+2)+3)+4       ⇝
                   ((3)+3)+4       ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h

sumWithL [2,3,4]    1              ⇝
sumWithL   [3,4]   (1+2)           ⇝
sumWithL     [4]  ((1+2)+3)        ⇝
sumWithL      [] (((1+2)+3)+4)     ⇝
                 ((1+2)+3)+4       ⇝
                   ((3)+3)+4       ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1          ⇝
sumWithL   [3,4]   (1+2)       ⇝
sumWithL     [4]  ((1+2)+3)    ⇝
sumWithL      [] (((1+2)+3)+4) ⇝
                  ((1+2)+3)+4  ⇝
                    ((3)+3)+4  ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1          ⇝
sumWithL   [3,4]   (1+2)       ⇝
sumWithL     [4]  ((1+2)+3)    ⇝
sumWithL      [] (((1+2)+3)+4) ⇝
                 ((1+2)+3)+4   ⇝
                   ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                  ((1+2)+3)+4   ⇝
                    ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                  ((1+2)+3)+4   ⇝
                    ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                  ((1+2)+3)+4   ⇝
                    ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                  ((1+2)+3)+4   ⇝
                    ((3)+3)+4   ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                   ((1+2)+3)+4  ⇝
                     ((3)+3)+4  ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹

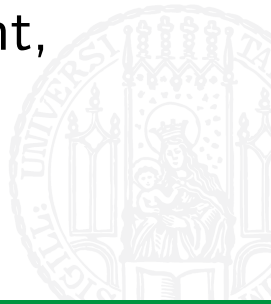
Speicherverbrauch Lazy Evaluation

Beispiel

```
sumWithL :: [Int] -> Int -> Int
sumWithL   []  acc = acc
sumWithL (h:t) acc = sumWithL t $ acc+h
```

```
sumWithL [2,3,4]    1           ⇝
sumWithL   [3,4]   (1+2)        ⇝
sumWithL     [4]  ((1+2)+3)     ⇝
sumWithL      [] (((1+2)+3)+4)  ⇝
                   ((1+2)+3)+4  ⇝
                     ((3)+3)+4  ⇝ (6)+4 ⇝ 10
```

Obwohl `sumWithL` endrekursiv ist, wird viel Speicher verbraucht, da ein großer Summen-Ausdruck für jedes Element der Liste erstellt wird. ☹



Striktheit

Eine Argument einer Funktion heißt , wenn es auf jeden Fall ausgewertet wird, egal welchen Wert die anderen Argumente haben.

Beispiel:

```
bar x y z = if 1 < succ x then z else x+y
```

Hier ist x strikt, y und z aber nicht.

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h
```

```
sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS    [] 10 ⇝
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS    [] 10 ⇝
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

Haskell bietet daher einen Mechanismus, welcher **strikte Auswertung** erzwingt:

```
($!) :: (a -> b) -> a -> b
```

Der Ausdruck `f $! x` erzwingt die Auswertung von `x` vor der Anwendung von `f`.

Beispiel:

```
sumWithS :: [Int] -> Int -> Int
sumWithS   []  acc = acc
sumWithS (h:t) acc = sumWithS t $! acc+h

sumWithS [2,3,4]     1    ⇝
sumWithS   [3,4] $! (1+2) ⇝ sumWithS [3,4] 3
sumWithS     [4] $! (3+3) ⇝ sumWithS   [4] 6
sumWithS      [] $! (6+4) ⇝ sumWithS   [] 10 ⇝ 10
```

Striktheit erzwingen

`$!` ist definiert durch das Primitiv `seq :: a -> b -> b`

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

`seq` erzwingt die Auswertung seines ersten Argumentes und liefert danach das zweite Argument zurück.

```
foo x =
  let zwischenwert = bar x
      ergebnis     = goo zwischenwert
  in  seq zwischenwert ergebnis
```

Von einigen Funktionen bietet die Standardbibliothek auch strikte Varianten, z.B. macht `foldl'` das gleiche wie `foldl`, erzwingt jedoch die Auswertung des Akkumulators in jedem Schritt.

Striktheit erzwingen

Das erste Argument von `seq` wird nur soweit ausgewertet, bis dessen äußere Form klar ist, darin kann auf weitere Thunks verwiesen werden:

Int, Bool: werden vollständig ausgewertet

Listen: ausgewertet bis klar ist, ob Liste leer ist oder nicht. Kopf und der Rumpf werden nicht ausgewertet!

Tupel: ausgewertet bis Tupel-Konstruktor fest steht, d.h. Elemente des Tupels werden nicht ausgewertet.

Maybe: ausgewertet bis `Nothing` oder `Just`, das Argument von Just wird noch nicht ausgewertet.

Generell wertet `seq` bis zum äußeren Konstruktor aus. Argumente des Konstruktors werden nicht weiter ausgewertet.

Schwache Kopf-Normalform, engl. Weak Head Normal Form

Demo

Demo `sumWith.hs`:
Wir führen `sumWithL` und `sumWithS` für große Listen mit GHC aus und werden überrascht!

Demo

Demo `sumWith.hs`:
Wir führen `sumWithL` und `sumWithS` für große Listen mit GHC aus und werden überrascht!

- Wir haben gesehen, dass der Kompilier automatisch eine Striktheit-Analyse durchführt, d.h. wir müssen nur selten eingreifen.

- Wenn ein Stack-Overflow eintritt, dann sollte man über Endrekursion und auch über Striktheit nachdenken.

Probleme mit Striktheit

Einfügen von `seq` kann ein funktionierendes Programm zerstören:

```
> foldr  (&&) True (repeat False)
False

> foldr' (&&) True (repeat False)  -- aus Data.Foldable
<interactive>: Heap exhausted;
```

Probleme mit Striktheit

Einfügen von `seq` kann ein funktionierendes Programm zerstören:

```
> foldr  (&&) True (repeat False)
False

> foldr' (&&) True (repeat False)  -- aus Data.Foldable
<interactive>: Heap exhausted;

> let foo x y = x
> foo 42 $  undefined
42

> foo 42 $! undefined
*** Exception: Prelude.undefined
```

Eine Funktion, welche für `undefined` in Argument *n* immer einen Fehler liefern, nennt man auch **strikt im *n*-ten Argument**.

Faulheit in strikten Sprachen

In anderen Programmiersprachen ist die Idee nicht unbekannt, z.B. C und Java bieten faule Varianten von logischen Operatoren an: Der Java-UND-Operator `&` wertet immer beide Argumente aus; aber `&&` wertet das zweite Argument nicht aus, wenn das erste bereits `False` ergibt. $\Rightarrow$ short-circuit (Kurzschluß)

Programmiersprachen, in denen im Gegensatz zur verzögerten Auswertung alle Ausdrücke sofort ausgewertet werden, nennt man **strikt** (engl. **eager**). Fast alle imperativen Sprachen sind strikt.

Faule Auswertung kann in strikten Sprachen simuliert werden, in dem man Ausdrücke zu Funktionen mit Scheinargumenten macht:

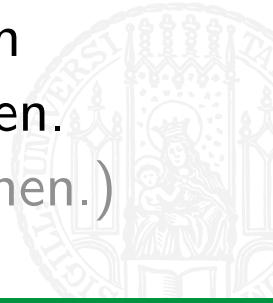
```
foo x = let e' = (\_ -> e) -- keine Auswertung von e
        in ... e' () ...   -- Auswertung von e
```

Übersicht Auswertestrategien

Strategie		*Auswertung*	*Beispiele*
Call-By-Value	Übergabe durch Werte; unbenötigte Argumente werden trotzdem ausgewertet	1 mal	C Java
Call-By-Reference	Übergabe durch Verweis auf Werte; Änderungen wirken sich auf Aufrufer aus.	1 mal	C++ Java
Call-By-Name	Ausdrücke werden in Rumpf substituiert; ignoriert alle unbenötigten Argumente	0–n mal	Algol60 Makrosp.
Call-By-Need	Memoisierende Variante von Call-By-Name	0–1 mal	Haskell

Weiterhin wird noch unterschieden, ob Funktionsparameter von *links-nach-rechts* oder von *rechts-nach-links* ausgewertet werden.

(Nur relevant, wenn Seiteneffekte auftreten können.)



Zusammenfassung

- Terminations- und Ausnahmeverhalten können von der Auswertestrategie abhängen

  In nicht-funktionalen Sprachen sogar das Ergebnis
- Auswertestrategien call-by-name und call-by-value: Besseres Terminationsverhalten versus Effiziente Ausführung
- Lazy evaluation als effiziente Implementierung von call-by-name
- Lazy Evaluation erlaubt Verwendung unendlicher oder zirkulärer Datenstrukturen
- Lazy Evaluation erlaubt gute Modularisierung durch Trennung von Daten und Kontrollfluss
- Ein Funktionsargument, welches auf jeden Fall ausgewertet wird, heißt strikt

  Haskell kennt Annotationen zur Erzwingung von Striktheit
- Mangelnde Striktheit kann zu erhöhtem Speicherbedarf führen