Fortgeschrittene Funktionale Programmierung

Teil 3: Laziness und Zirkularität

Steffen Jost

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

6. November 2011

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 3: Laziness und Zirkularität

Auswertungsstrategie

Eine **Auswertungsstrategie** beschreibt, wie ein Programmausdruck ausgewertet wird. Insbesondere:

1. In welcher Reihenfolge werden die reduzierbaren Teilausdrücke (**Redex**) bearbeitet?
2. Werden Funktionsargumente ausgewertet, bevor eine Funktion aufgerufen wird?

Achtung

- Bei Effekten (Wertzuweisung, Ausnahmen, Ein/Ausgabe) ist die Auswertungsstrategie signifikant.
- Bei *terminierenden* funktionalen Programmen ohne Seiteneffekten ist die Auswertereihenfolge egal.

  Die Auswertereihenfolge kann aber beeinflussen, ob ein Programm terminiert (wenn z.B. bei der Auswertung eines unbenötigten Funktionsargumentes ein Fehler auftritt).

Lazy Evaluation

Haskell nutzt Auswertestrategie **call-by-need** (Bedarfsauswertung), implementiert durch **lazy evaluation** (verzögerte Auswertung).

- Termination wie bei Call-By-Name
- Effizienz (fast) wie bei Call-By-Value

Idee

- Anstelle eines Redex wird ein Verweis (**Thunk**) eingesetzt
- Bei der ersten Verwendung des Verweises wird der Redex ausgewertet und durch seinen Ergebniswert ersetzt; weitere Verwendungen des Verweises liefern sofort diesen Ergebniswert.

Nur möglich, da die Auswertereihenfolge prinzipiell egal ist!

Nachteile

- Mehr Speicher notwendig zum Merken von Thunks/Redexe
- Keine **Sequentialität** der Auswertung (erst das, dann das)

Vergleich Auswertungsstrategien

Ausgabe eines Programms mit Seiteneffekten, Auswertung von z:

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

foo x y z = y + y + z

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

Call-By-Value "first" "second" "third" 7

Call-By-Name "second" "second" "third" 7

Lazy Evaluation "second" "third" 7

Ausgaben hypothetisch! GHC: immer Lazy Evaluation

Vergleich Auswertungsstrategien

Ausgabe eines Programms mit Seiteneffekten, Auswertung von z:

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

foo x y z = y + y + z

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

Call-By-Value *** Exception: Prelude.undefined

Call-By-Name "second" "second" "third" 7

Lazy Evaluation "second" "third" 7

Ausgaben hypothetisch! GHC: immer Lazy Evaluation

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 des Arguments `x`.
- Falls `x<0` wahr ist, wird der Wert von `abs x` zurückgegeben; weder `y` noch `z` werden ausgewertet.
- Falls `x<0` falsch ist, wird der Wert von `x+y` zurückgegeben; dies erfordert die Auswertung von `y`.
- `z` wird in keinem Fall ausgewertet.

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

Lazy Evaluation: Beispiel 2

```
g x y =   let  z = x `div` y
          in   if x > y * y then x else z

t2 = g 5 0
```

- `t2` liefert `5`. Da der Wert von `z` nicht benötigt wird, kommt es nicht zur Division durch Null.
- Nicht nur Funktionsargumente werden verzögert ausgewertet, sondern *beliebige Teilausdrücke*.
- **Sequentialität** der Auswertung (erst das, dann das) nicht gegeben, was sehr verwirrend sein kann.

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))
```

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))
```

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))
```

Thunks in GHCI

GHCI erlaubt die Inspektion des Speicher mit den Befehlen `:print` und `:sprint`, ohne das dabei Thunks ausgewertet werden:

```
> let x = map even [1..22]
> :sprint x
x = _
> take 3 $ drop 3 x
[True,False,True]
> :sprint x
x = _ : _ : _ : True : False : True : _
```

Beide Befehle unterscheiden sich nur in der Ausführlichkeit der gezeigten Information, so zeigt `:print` für jeden Thunk eine Referenz und den jeweiligen Typ an.

Thunks in GHCI

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

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

Beispiel: Trennung Daten & Kontrolle

Lazy Evaluation ermöglicht Trennung von Daten und Kontrollfluss, auch ohne den Einsatz von Zirkularität:

Beispiel

```
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)
```

`factors` berechnet alle Faktoren einer Zahl.

Die Berechnung von `isPrime` bricht sofort ab, so bald der erste Faktor gefunden wurde, trotz Verwendung von `factors` werden also nicht alle Faktoren berechnet!

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) = case compare x y of
  LT -> x : minus xt ys
  EQ ->     minus xt yt
  GT ->     minus xs yt
```

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

Speicherverbrauch

Der Teufel kann aber hier im Detail stecken:

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

cycle  :: [a] -> [a]
cycleA xs = xs ++ cycleA xs
cycleB xs = xs' where xs' = xs++xs'
```

`cycleA l` verbraucht potenziell unendlich viel Speicher; 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.

Für die weitere Verarbeitung spielt dies jedoch keine Rolle!

Formale Operationale Semantik

Um solche Eigenschaften zu untersuchen, muss man die Auswertung durch eine **operationale Semantik** formalisieren.
„A natural semantics for lazy evaluation“ von J. Launchbury POPL'93 wird oft als Basis verwendet, z.B. auch in unseren Forschungsarbeiten zu verzögerter Auswertung:

$$\frac{}{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m}{\underset{m}{\vdash}} w \Downarrow w, \mathcal{H}} \quad (\textsc{Whnf}_{\Downarrow})$$

$$\frac{\ell \notin \mathcal{L} \qquad \mathcal{H}[\ell \mapsto e], \mathcal{S}, \mathcal{L} \cup \{\ell\} \overset{m}{\underset{m'}{\vdash}} e \Downarrow w, \mathcal{H}'[\ell \mapsto e]}{\mathcal{H}[\ell \mapsto e], \mathcal{S}, \mathcal{L} \overset{m}{\underset{m'}{\vdash}} \ell \Downarrow w, \mathcal{H}'[\ell \mapsto w]} \quad (\textsc{Var}_{\Downarrow})$$

$$\frac{\ell \text{ is fresh} \qquad \mathcal{H}[\ell \mapsto e_1[\ell/x]], \mathcal{S}, \mathcal{L} \overset{m}{\underset{m'}{\vdash}} e_2[\ell/x] \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m+1}{\underset{m'}{\vdash}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow w, \mathcal{H}'} \quad (\textsc{Let}_{\Downarrow})$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m}{\underset{m'}{\vdash}} e \Downarrow \lambda x.\, e', \mathcal{H}' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \overset{m'}{\underset{m''}{\vdash}} e'[\ell/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m}{\underset{m''}{\vdash}} e\ \ell \Downarrow w, \mathcal{H}''} \quad (\textsc{App}_{\Downarrow})$$

$$\frac{\mathcal{H}, \mathcal{S} \cup \left(\bigcup_{i=1}^{n} \{\overline{x}_i\} \cup \mathrm{BV}(e_i)\right), \mathcal{L} \overset{m}{\underset{m'}{\vdash}} e_0 \Downarrow c_k(\overline{\ell}), \mathcal{H}' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \overset{m'}{\underset{m''}{\vdash}} e_k[\overline{\ell}/\overline{x}_k] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \overset{m}{\underset{m''}{\vdash}} \mathsf{match}\ e_0\ \mathsf{with}\ \{c_i(\overline{x}_i)\text{->}e_i\}_{i=1}^{n} \Downarrow w, \mathcal{H}''} \quad (\textsc{Match}_{\Downarrow})$$

Fig. 1. Instrumented version of Launchbury's operational semantics

Formalisierung der verzögerten Auswertung

Wir begnügen uns hier mit einer vereinfachten Version:

- Alle Teilausdrücke werden zuerst in den Speicher geschoben.
- Um mit Rekursion umzugehen, wird dabei die Variable durch einen frischen, einzigartigen Namen (Speicheradresse) ersetzt; im ursprünglichen Quelltext kommen diese nicht vor.
  Wir verwenden dafür spitze Klammern in Beispielen, z.B. $\langle 3 \rangle$
- Jede Speicherzelle wird nur *einmal* verändert: Wird der Wert der Speicherzelle gebraucht und daher ausgewertet, so ersetzen wir den Ausdruck im Speicher durch seinen Wert.
- Speicherzellen werden nie gelöscht.
  In der Praxis kümmert sich ein Garbage Collector um das freigeben unbenötigter Speicherzellen.

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA ['a','b']`			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	`['a','b']`	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩ ⟨11⟩ ++ (`cycleA` ⟨11⟩)	⟨11⟩	['a','b']	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	['a','b']	
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
	⟨13⟩	`'a'`	✓
	⟨14⟩	`['b']`	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨13⟩	`'a'`	✓
	⟨14⟩	`['b']`	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
	⟨14⟩	`['b']`	
	⟨15⟩	⟨14⟩ ++ ⟨12⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩	⟨14⟩	`['b']`	
	⟨15⟩	⟨14⟩ ++ ⟨12⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	`['b']`	
	⟨15⟩	⟨14⟩ ++ ⟨12⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
	⟨15⟩	⟨14⟩ ++ ⟨12⟩	
	⟨16⟩	`'b'`	✓
	⟨17⟩	[ ]	✓

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	⟨14⟩ ++ ⟨12⟩	
	⟨16⟩	`'b'`	✓
	⟨17⟩	[ ]	✓

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	⟨14⟩ ++ ⟨12⟩	
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	`'b'`	✓
	⟨17⟩	[ ]	✓
	⟨18⟩	⟨17⟩ ++ ⟨12⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	`'b'`	✓
	⟨17⟩	[ ]	✓
	⟨18⟩	⟨17⟩ ++ ⟨12⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	`'b'`	✓
⟨18⟩	⟨17⟩	[ ]	✓
	⟨18⟩	⟨17⟩ ++ ⟨12⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	`'b'`	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
	⟨18⟩	⟨17⟩ ++ ⟨12⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	`'b'`	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	`'b'`	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	`'b'`	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	
⟨11⟩ ++ (`cycleA` ⟨11⟩)			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	`'b'`	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ (⟨14⟩ ++ ⟨19⟩)			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	`'a'`	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	`'b'`	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	⟨14⟩ ++ ⟨19⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨20⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	⟨14⟩ ++ ⟨19⟩	

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨20⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	(:) ⟨13⟩ ⟨20⟩	✓
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	⟨14⟩ ++ ⟨19⟩	
⟨20⟩			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨20⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	(:) ⟨13⟩ ⟨20⟩	✓
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	⟨14⟩ ++ ⟨19⟩	
⟨20⟩ = ⟨14⟩ ++ ⟨19⟩			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨20⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	(:) ⟨13⟩ ⟨20⟩	✓
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	⟨14⟩ ++ ⟨19⟩	
⟨20⟩ = ⟨14⟩ ++ ⟨19⟩			
(:) ⟨16⟩ (⟨17⟩ ++ ⟨19⟩)			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨20⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	(:) ⟨13⟩ ⟨20⟩	✓
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	⟨14⟩ ++ ⟨19⟩	
⟨20⟩ = ⟨14⟩ ++ ⟨19⟩	⟨21⟩	⟨17⟩ ++ ⟨19⟩	
(:) ⟨16⟩ (⟨17⟩ ++ ⟨19⟩)			
(:) ⟨16⟩ ⟨21⟩			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨20⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	(:) ⟨13⟩ ⟨20⟩	✓
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	(:) ⟨16⟩ ⟨21⟩	✓
⟨20⟩ = ⟨14⟩ ++ ⟨19⟩	⟨21⟩	⟨17⟩ ++ ⟨19⟩	
(:) ⟨16⟩ (⟨17⟩ ++ ⟨19⟩)			
(:) ⟨16⟩ ⟨21⟩			
⋮			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨20⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	(:) ⟨13⟩ ⟨20⟩	✓
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	(:) ⟨16⟩ ⟨21⟩	✓
⟨20⟩ = ⟨14⟩ ++ ⟨19⟩	⟨21⟩	⟨19⟩	
(:) ⟨16⟩ (⟨17⟩ ++ ⟨19⟩)			
(:) ⟨16⟩ ⟨21⟩			
⋮			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨20⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	(:) ⟨13⟩ ⟨20⟩	✓
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	(:) ⟨16⟩ ⟨21⟩	✓
⟨20⟩ = ⟨14⟩ ++ ⟨19⟩	⟨21⟩	⟨19⟩	
(:) ⟨16⟩ (⟨17⟩ ++ ⟨19⟩)	⟨22⟩	`cycleA` ⟨11⟩	
(:) ⟨16⟩ ⟨21⟩			
⋮			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨20⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	(:) ⟨13⟩ ⟨20⟩	✓
⟨11⟩ ++ ⟨19⟩	⟨19⟩	`cycleA` ⟨11⟩	
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	(:) ⟨16⟩ ⟨21⟩	✓
⟨20⟩ = ⟨14⟩ ++ ⟨19⟩	⟨21⟩	⟨19⟩	
(:) ⟨16⟩ (⟨17⟩ ++ ⟨19⟩)	⟨22⟩	`cycleA` ⟨11⟩	
(:) ⟨16⟩ ⟨21⟩	⟨23⟩	⟨14⟩ ++ ⟨22⟩	
⋮			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleA` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨20⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨13⟩	'a'	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
(:) ⟨16⟩ ⟨18⟩	⟨16⟩	'b'	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩	⟨17⟩	[ ]	✓
⟨12⟩ = `cycleA` ⟨11⟩	⟨18⟩	(:) ⟨13⟩ ⟨20⟩	✓
⟨11⟩ ++ ⟨19⟩	⟨19⟩	(:) ⟨13⟩ ⟨23⟩	✓
(:) ⟨13⟩ ⟨20⟩	⟨20⟩	(:) ⟨16⟩ ⟨21⟩	✓
⟨20⟩ = ⟨14⟩ ++ ⟨19⟩	⟨21⟩	(:) ⟨13⟩ ⟨23⟩	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨19⟩)	⟨22⟩	`cycleA` ⟨11⟩	
(:) ⟨16⟩ ⟨21⟩	⟨23⟩	⟨14⟩ ++ ⟨22⟩	
⋮			

```
cycleA xs = xs ++ (cycleA xs)
```

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB ['a','b']`			

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	`['a','b']`	

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	['a','b']	
⟨12⟩	⟨12⟩	⟨11⟩ ++ ⟨12⟩	

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	`['a','b']`	
⟨12⟩	⟨12⟩	⟨11⟩ ++ ⟨12⟩	
⟨11⟩ ++ ⟨12⟩			

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	⟨11⟩ ++ ⟨12⟩	
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
	⟨14⟩	['b']	

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	⟨11⟩ ++ ⟨12⟩	
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	['b']	

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	⟨11⟩ ++ ⟨12⟩	
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	['b']	
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	⟨14⟩ ++ ⟨12⟩	

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	['b']	
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	⟨14⟩ ++ ⟨12⟩	

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	['b']	
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	⟨14⟩ ++ ⟨12⟩	
⟨15⟩			

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	['b']	
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	⟨14⟩ ++ ⟨12⟩	
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩			

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	⟨14⟩ ++ ⟨12⟩	
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨16⟩	'b'	✓
	⟨17⟩	[ ]	✓

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	⟨14⟩ ++ ⟨12⟩	
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨16⟩	'b'	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨17⟩	[ ]	✓

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	⟨14⟩ ++ ⟨12⟩	
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨16⟩	'b'	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨17⟩	[ ]	✓
(:) ⟨16⟩ ⟨18⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	`'a'`	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨16⟩	`'b'`	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨17⟩	[ ]	✓
(:) ⟨16⟩ ⟨18⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	
⟨18⟩			

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨16⟩	'b'	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨17⟩	[ ]	✓
(:) ⟨16⟩ ⟨18⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩			

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨16⟩	'b'	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨17⟩	[ ]	✓
(:) ⟨16⟩ ⟨18⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩			
⟨12⟩			

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
cycleB ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨16⟩	'b'	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨17⟩	[]	✓
(:) ⟨16⟩ ⟨18⟩	⟨18⟩	⟨17⟩ ++ ⟨12⟩	
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩			
⟨12⟩			
(:) ⟨13⟩ ⟨15⟩			

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

Beispiel

Ausdruck zur Auswertung	Adresse	Thunk/Wert	Ausgewertet
`cycleB` ⟨11⟩	⟨11⟩	(:) ⟨13⟩ ⟨14⟩	✓
⟨12⟩	⟨12⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨11⟩ ++ ⟨12⟩	⟨13⟩	'a'	✓
(:) ⟨13⟩ (⟨14⟩ ++ ⟨12⟩)	⟨14⟩	(:) ⟨16⟩ ⟨17⟩	✓
(:) ⟨13⟩ ⟨15⟩	⟨15⟩	(:) ⟨16⟩ ⟨18⟩	✓
⟨15⟩ = ⟨14⟩ ++ ⟨12⟩	⟨16⟩	'b'	✓
(:) ⟨16⟩ (⟨17⟩ ++ ⟨12⟩)	⟨17⟩	[ ]	✓
(:) ⟨16⟩ ⟨18⟩	⟨18⟩	(:) ⟨13⟩ ⟨15⟩	✓
⟨18⟩ = ⟨17⟩ ++ ⟨12⟩			
⟨12⟩			
(:) ⟨13⟩ ⟨15⟩			
Alles vollständig ausgewertet!			

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

```
(++) []     ys = ys
(++) (x:xs) ys = (:) x (xs++ys)
```

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 **strikt**, 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 einen Mechanismus, welcher die **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 ⇝ [0
```

Striktheit erzwingen

Haskell bietet einen Mechanismus, welcher die **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 einen Mechanismus, welcher die **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

`$!` könnte mit Primitiv `seq :: a -> b -> b` definiert werden:

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

`seq` erzwingt die Auswertung seiner beiden Argumente (Reihenfolge undefiniert; $\Rightarrow$ `pseq`) und liefert danach das 2. 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** Weak Head Normal Form

WHNF

`Control.DeepSeq` bietet Kontrolle über Auswertungsreihenfolge.

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.

Striktheit erzwingen

Eine weitere Alternative bietet die Spracherweiterung `BangPatterns`. Damit ist `($!)` tatsächlich definiert:

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

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

```
ghci -XBangPatterns

> let foo (x,!y) = [x,y]
> drop 1 $ foo (undefined,2)
[2]
> take 1 $ foo (2,undefined)
*** Exception: Prelude.undefined
```

Strikte Datentypen

GHC erlaubt auch die Deklaration von strikten Datentypen:

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

> let (FP u v) = FP undefined True in v
True
> 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.

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

```
data FaulesPaar   = FP ~Integer ~Bool
```

Probleme mit Striktheit

Einführen von Striktheit kann funktionierende Programme zerstören:

```
stack ghci -- +RTS -M1g
> foldr  (&&) True (repeat False)
False
> :module + Data.Foldable
> foldr' (&&) True (repeat False)
*** Exception: heap overflow
```

Probleme mit Striktheit

Einführen von Striktheit kann funktionierende Programme zerstören:

```
stack ghci -- +RTS -M1g
> foldr  (&&) True (repeat False)
False
> :module + Data.Foldable
> foldr' (&&) True (repeat False)
*** Exception: heap overflow


> let foo = const 42                    -- const x y = x
> foo $ undefined
42
> foo $! undefined
*** Exception: Prelude.undefined
```

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

Faulheit erzwingen

Man kann auch faule Pattern-Matches erzwingen:

```
> let bar  (x,y) = 42
> bar undefined
*** Exception: Prelude.undefined
> let baz ~(x,y) = 42
> baz undefined
42
```

Auch in verschachtelten Patterns möglich, z.B. `(x,~y)`.

Fallstrick: Faules Patterns sind irrefutable, müssen also immer als letzte geprüft werden. Verwendung kann man sich so vorstellen:

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

qux ~(h:t) = 2*h:(qux t)
```

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

qux l= 2*(head l):(qux(tail l))
```

Faulheit erzwingen: Beispiel

Demo: splitAt_lazypattern.hs

```
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)
```

Damit steht der erste Teil des Ergebnis-Paares sofort zur Verfügung; der Rest muss dagegen noch später ausgewertet werden.

Faulheit erzwingen: Beispiel

Demo: `splitAt_lazypattern.hs`

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

Hinweis Da das Tupel-Matching sowieso irrefutable ist, können wir hier auch `let` verwenden. Dadurch wird das Matching automatisch verzögert.

Die Prelude Definition von `splitAt` erreicht das Gleiche mit `where`

Faulheit in strikten Sprachen

In anderen Programmiersprachen ist die Idee nicht unbekannt:
z.B. bieten C und Java faule Varianten von logischen Operatoren:
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
```

Zusammenfassung

- Haskell verwendet Auswertestrategie Lazy Evaluation: Maximal einmal auswerten, wenn überhaupt
- Ein Funktionsargument, welches auf jeden Fall ausgewertet wird, heißt strikt.
- Auswertestrategie kann Terminierung und Ausnahmeverhalten entscheiden. *Faustregel:* Faulheit liefert mehr Ergebnisse
- Auswertestrategie beeinflusst Speicherverbrauch. *Faustregel:* Striktheit spart Stack, Faulheit spart Heap.
- Annotationen erlauben es Striktheit zu beeinflussen: Striktes Pattern: `!p` faules Pattern: `~p` (irrefutable)
- Lazy Evaluation erlaubt unendliche Datenstrukturen (bzw. zirkuläre Datenstrukturen ohne explizite Pointer). Dadurch wird eine gute Modularisierung durch Trennung von Daten und Kontrollfluss erlaubt.

Definition

Ein *zirkuläres* Programm erzeugt eine Datenstruktur, deren Berechnung von sich selbst abhängt.

- Solche Programme erfordern nicht-strikte Datenstrukturen
  oder explizites Pointer-Management in imperativen Sprachen.
  Vorsicht: doppelt verkettete Liste hat zirkuläre Verweise,
  aber die Berechnung ist meist nicht zirkulär!
- Sprachen mit nicht-strikter Semantik unterstützen solche zirkulären Datenstrukturen ganz natürlich; in Sprachen mit strikter Semantik können diese simuliert werden.
- Zirkuläre Programme werden oft verwendet um mehrfaches Traversieren einer Datenstruktur oder den Aufbau von Zwischen-Datenstrukturen zu vermeiden.

Zirkuläre Datenstrukturen

Ein einfaches Beispiel einer zirkulären Datenstruktur ist die Liste *aller* natürlichen Zahlen:

```
nums2 = 0 : (map (+1) nums2)
```

Beachte:
Programm benutzt die erzeugte Datenstruktur selbst als Eingabe!

Rekursive Variante zur Erinnerung:

```
nums1  =  iterate (+1) 0

iterate f x  =  x : iterate f (f x)
```

Beispiel: Eliminierung von Duplikaten

Die Funktion `nub1` eliminiert Duplikate aus einer Liste, ohne die Reihenfolge zu verändern:

```
nub1 :: [Integer] -> [Integer]
nub1 []          =  []
nub1 (x : xs)    =  x : (nub1 (filter (x/=) xs))
```

Beispiel: Eliminierung von Duplikaten

Die Funktion `nub1` eliminiert Duplikate aus einer Liste, ohne die Reihenfolge zu verändern:

```
nub1 :: [Integer] -> [Integer]
nub1 []         =  []
nub1 (x : xs)   =  x : (nub1 (filter (x/=) xs))
```

Diese Funktion erzeugt für jedes Listenelement der Eingabeliste eine Liste als Zwischen-Datenstruktur:

1. Liste, bei der all Doppelten des ersten Elements entfernt wurden
2. Liste, bei der alle Doppelten des ersten und zweiten Elements entfernt wurden;

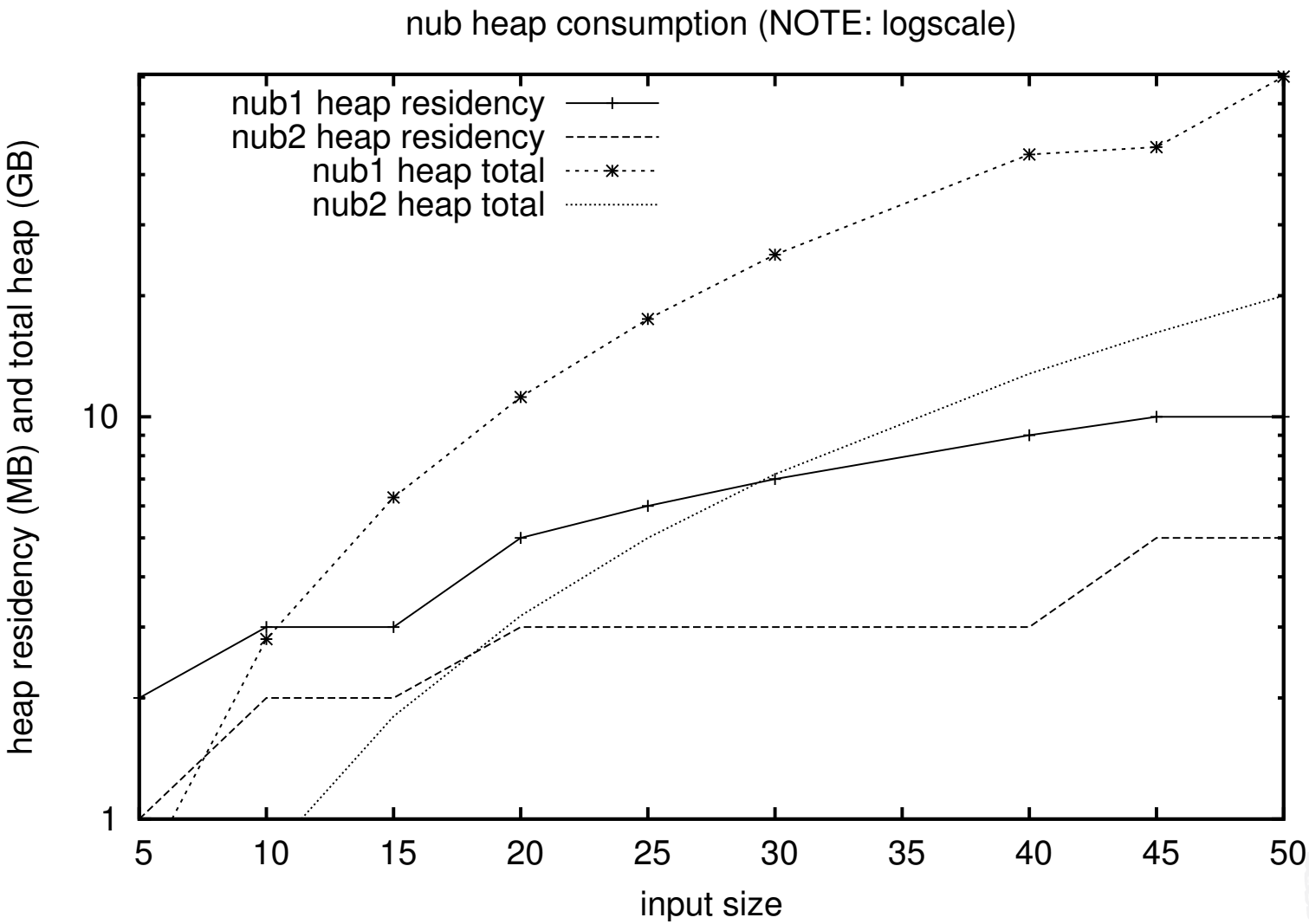
usw. ☹

Beispiel: Eliminierung von Duplikaten

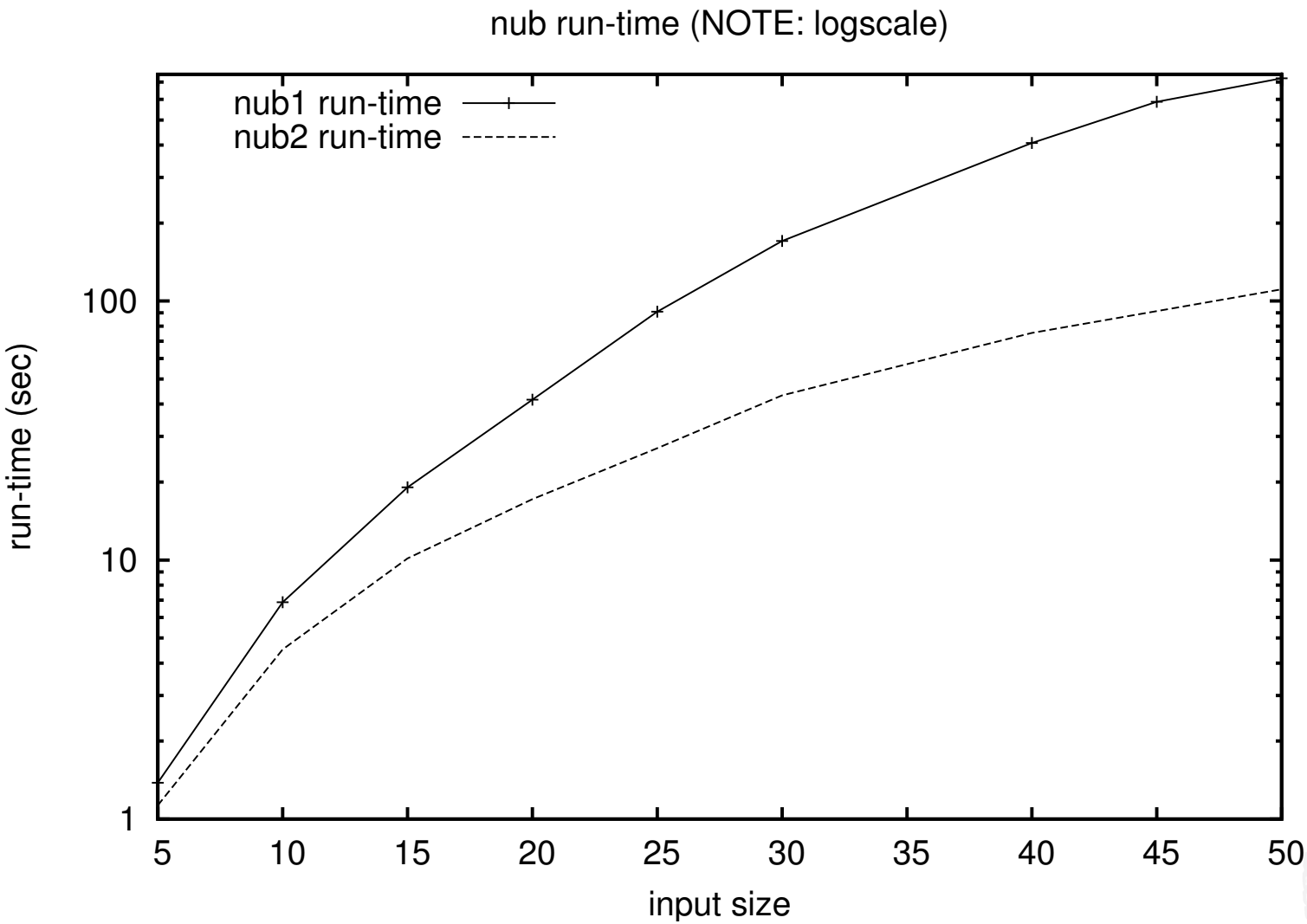
Eine bessere Implementierung vermeidet diese Zwischen-Listen:

```
nub2 :: [Integer] -> [Integer]
nub2 xs = res
  where
    res  =  build xs 0

    build    []   _  =  []
    build (x:xs)  n
      | mem x res n  =  build xs n
      | otherwise    =  x : build xs (n + 1)

    mem _   _ys  0   =  False
    mem x (y:ys) n
      | x == y       =  True
      | otherwise    =  mem x ys (n - 1)
```

Performanz


nub heap consumption (NOTE: logscale)
nub1 heap residency
nub2 heap residency
nub1 heap total
nub2 heap total
heap residency (MB) and total heap (GB)
10
1
5
10
15
20
25
30
35
40
45
50
input size

Performanz


nub run-time (NOTE: logscale)
nub1 run-time
nub2 run-time
run-time (sec)
100
10
1
5
10
15
20
25
30
35
40
45
50
input size

Beispiel: Eliminierung von Duplikaten

Die Standardbibliothek definiert dagegen ohne Zirkularität:

```
nub3 :: [Integer] -> [Integer]
nub3 l = nubBy' l []
  where
    nub3'     [] _ys  = []
    nub3' (x:xs) ys
      | mem x ys      = nub3' xs ys
      | otherwise     = x : nub3' xs (x:ys)

    mem _     []      = False
    mem x (y:ys)
      | x == y        = True
      | otherwise     = x `mem` ys
```

$\Rightarrow$ Viel schneller, insbesondere wenn viele Doppelte vorkommen; dafür Speicherbedarf für Liste mit einzigartigen Elementen.

Beispiel: repmin

Gegeben: Binärer Baum von Integer Werten.

Gesucht: Binärer Baum mit der gleichen Struktur in dem die Werte der Blätter durch das kleinste Element im Baum ersetzt sind.

Naive Implementierung

Die naive Implementierung verwendet 2 separate Funktionen:

```
treemin :: (Ord a) => BinTree a -> a
treemin (Leaf x)    =  x
treemin (Node l r)  =  min (treemin l) (treemin r )

replace :: BinTree a -> a -> BinTree a
replace (Leaf _) m   = Leaf m
replace (Node l r) m = Node (replace l m) (replace r m)

transform2p :: (Ord a) => BinTree a -> BinTree a
transform2p t  =  replace t (treemin t)
```

Dieser Code traversiert den Baum 2-mal: einmal in `treemin` und einmal in `replace`

Verbesserte Implementierung

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.
Wir verschmelzen beide Funktionen zu einer:

```
repmin (Leaf x)   m  =  (Leaf m, x)
repmin (Node l r) m  =
  let (l', ml)  = repmin l m
      (r', mr)  = repmin r m
  in  (Node l' r', min ml mr)
```

Verbesserte Implementierung

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.
Wir verschmelzen beide Funktionen zu einer:

```
repmin (Leaf x)   m  =  (Leaf m, x)
repmin (Node l r) m  =  (l', ml) `comb` (r', mr)
  where
    (l', ml)  = repmin l m
    (r', mr)  = repmin r m
    comb (l', ml) (r', mr) = (Node l' r', min ml mr)
```

Verbesserte Implementierung

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.
Wir verschmelzen beide Funktionen zu einer:

```
repmin (Leaf x)   m  =  (Leaf m, x)
repmin (Node l r) m  =  (l', ml) `comb` (r', mr)
  where
    (l', ml)  = repmin l m
    (r', mr)  = repmin r m
    comb (l', ml) (r', mr) = (Node l' r', min ml mr)
```

Mittels Gleichheitsschließen lässt sich zeigen dass:

```
repmin t (treemin t) = (replace t (treemin t), treemin t)
```

Verbesserte Implementierung

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.
Wir verschmelzen beide Funktionen zu einer:

```
repmin (Leaf x)   m  =  (Leaf m, x)
repmin (Node l r) m  =  (l', ml) `comb` (r', mr)
  where
    (l', ml)  = repmin l m
    (r', mr)  = repmin r m
    comb (l', ml) (r', mr) = (Node l' r', min ml mr)
```

Mittels Gleichheitsschließen lässt sich zeigen dass:

```
repmin t (treemin t) = (replace t (treemin t), treemin t)
```

Nun können wir eine zirkuläre Datenstruktur verwenden um das Resultat der `treemin` Komponente mit der Eingabe der `replace` Komponente zu verknüpfen:

```
transform1p t = fst p where p = repmin t (snd p)
```

Memoisation

- **Memoisation** ist eine Programmier- oder Implementierungstechnik, in der die Resultate früherer Aufrufe einer Funktion gespeichert ("memoised") werden, um wiederholtes Auswerten zu vermeiden.
- Im Allgemeinen verwendet man dazu eine Datenstruktur, wie eine Hash-Tabelle, und legt dort die bereits berechneten Werte ab.
- Dieser Ansatz ist von Vorteil, wenn die einzelnen Berechnungen sehr teuer sind, und den Administrationsaufwand der Tabellenverwaltung rechtfertigen.

Light-weight Memoisation

Zirkuläre Datenstrukturen bieten eine leichtgewichtige Alternative zu generellen Hash-Tabellen.

Anstatt eine Funktion direkt zu definieren, wird eine unendliche Liste aller Resultatwerte definiert. Der Funktionsaufruf wird durch eine Indexing-Operation auf der Liste ersetzt.

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

fib :: Int -> Integer
fib n  =  fibs !! n
```

Wiederholte Aufrufe liefern deutlich schneller ein Ergebnis!

Beispiel: Hamming Numbers

Liste aller Vielfachen von 2, 3 und 5

auch bekannt als harmonische Zahlen

```
hamming = 1 : (merge (map (2*) hamming)
                     (merge (map (3*) hamming)
                            (map (5*) hamming)))

merge :: Ord a => [a] -> [a] -> [a]
merge xs@(x:xt) ys@(y:yt)
  | x==y  =  x : merge xt yt
  | x< y  =  x : merge xt ys
  | x> y  =  y : merge xs yt
```

Termination von zirkulären Programmen

Der Beweis der Termination von zirkulären Programmen ist oft nicht trivial.

Im vorangegangen Beispiel kann man zeigen, dass immer nur auf vorherige Elemente der Datenstruktur zugegriffen wird, i.e. das n-te Element hängt von allen i-ten Elementen, mit $i < n$ ab.

Verallgemeinert nutzt man das Konzept der Produktivität um die Termination von zirkulären Programmen zu zeigen.

Produktivität

Definition (maximal)

Gegeben eine partielle Ordnung $\sqsubseteq$ auf einer Menge D. Ein Element $x \in D$ ist *maximal* wenn

$$\forall y \in D.y \sqsubseteq x$$

$\sqsubseteq$ auf Funktionen und Listen wird komponentenweise definiert.
Produktivität von $x \in \sigma$ induktiv über Struktur des Typs σ:

Definition (produktiv)

$x \in \sigma$ ist *produktiv*, wenn

- $\sigma = Int \vee \sigma = Bool$: x ist maximal
- $\sigma = \tau \to \tau'$: x bildet produktive Elemente in produktive Elemente ab
- $\sigma = [\tau]$: $\forall i \;.\; 0 \leq i \leq \#x \implies x!!i$ ist produktiv

Produktivität

Definition (Mengen-Produktivität)

Eine Liste $x \in [\sigma]$ ist A-produktiv ($A \subseteq \mathbb{N}$), gdw. $\forall a \in A \;.\; x!!a$ ist produktiv.

Intuition:

A ist die Menge der Indizes von berechneten Listen-Elementen.

Produktivität

Definition (Segment-Produktivität)

Eine Liste $x \in [\sigma]$ ist n-produktiv ($n \in \mathbb{N}$), gdw. x ist $\{0, 1, \ldots, n-1\}$ produktiv.

Intuition: die ersten n Elemente der Liste sind berechnet.

Definition (Segment-Produktivität)

Eine Funktion $f \in [\sigma] \to [\tau]$ ist v-produktiv für $v \in \mathbb{N} \to \mathbb{N}$, gdw. $\forall k \in \mathbb{N}, x \in [\sigma]$. x ist k produktiv $\implies$ $(f\ x)$ ist $(v\ k)$ produktiv

Intuition: die v Funktion gibt an, wie sich die Größe des berechneten Segments verändert.

Produktivität

Beispiel 1:

Die Funktion `map f` ist `id` produktiv,
d.h. die Segmentgröße bleibt unverändert.

Beispiel 2:

Die List-cons Funktion `(:)` ist $(+1)$ produktiv,
d.h. die Segmentgröße steigt um 1.

Produktivität

Theorem (Listen-Produktivität)

Sei eine Funktion $f \in [\sigma] \to [\sigma]$ *welche* $v \in \mathbb{N} \to \mathbb{N}$*-produktiv ist, und* $\forall k \in \mathbb{N} \,.\, (v\ k) > k$*. Dann ist der Fixpunkt von* f *eine produktive, unendliche Liste.*

Intuition: Um zu zeigen, dass alle Element einer unendlichen Liste berechenbar sind, muss man zeigen, dass die Änderung der Segmentgröße immer ansteigt. D.h. in jeder Iteration steigt der berechnete Teil der Liste.

Beispiel: Beweis der Produktivität der memoisierenden Fibonacci Funktion.

- Die Funktion `zipWith` ist min produktiv, da sie eine binäre Funktion f komponentenweise auf die beiden Argumentlisten `xs`, `ys` anwendet. Wenn die ersten m Elemente von `xs` und die ersten n Elemente von `ys` berechnet sind, dann sind die ersten `min` m n Elemente der Resultatliste berechnet.
- Die List-cons Funktion `(:)` ist $(+1)$ produktiv.
- Sei `fibs` eine k produktive Liste, und *vtl* die Produktivitätsfunktion von `tail`. Dann ist der Rumpf

  ```
  1 : 1 : (zipWith (+) fibs (tail fibs))
  ```

  $1 + 1 + \texttt{min}\ k\ (\mathit{vtl}\ k)$ produktiv.
- Weiterhin ist $\mathit{vtl}\ k = k - 1$ für alle $k > 0$.
- Daher ist die Produktivität des Rumpfs von `fibs`:
  $1 + 1 + \texttt{min}\ k\ (\mathit{vtl}\ k) = 2 + k - 1 = k + 1 > k$ □

Zusammenfassung

- Zirkularität erfordert nicht-strikte Datenstrukturen
- Zirkuläre Programme können wiederholtes Traversieren von Datenstrukturen vermeiden
- Zirkularität ermöglicht eine einfache Form von Memoisation
- Man muss bei zirkulären Programmen auf die Produktivität achten, d.h. es werden pro konsumierten Element einer Datenstruktur mindestens ebenso so viele Elemente erzeugt.