

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 6 vom 07.12.2020: Rekursive und zyklische Datenstrukturen

Christoph Lüth



Wintersemester 2020/21

Fahrplan

- ▶ **Teil I: Funktionale Programmierung im Kleinen**
 - ▶ Einführung
 - ▶ Funktionen
 - ▶ Algebraische Datentypen
 - ▶ Typvariablen und Polymorphie
 - ▶ Funktionen höherer Ordnung I
 - ▶ Rekursive und zyklische Datenstrukturen
 - ▶ Funktionen höherer Ordnung II
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- ▶ **Rekursive** Datentypen und **zyklische** Daten
 - ▶ ... und wozu sie nützlich sind
 - ▶ Fallbeispiel: Labyrinth
- ▶ Performance-Aspekte

Lernziele

- ① Wir verstehen, wie in Haskell „unendliche“ Datenstrukturen modelliert werden. Warum sind unendliche Listen nicht wirklich unendlich?
- ② Wir wissen, worauf wir achten müssen, wenn uns die Geschwindigkeit unser Haskell-Programme wichtig ist.

I. Rekursive und Zyklische Datenstrukturen

Konstruktion zyklischer Datenstrukturen

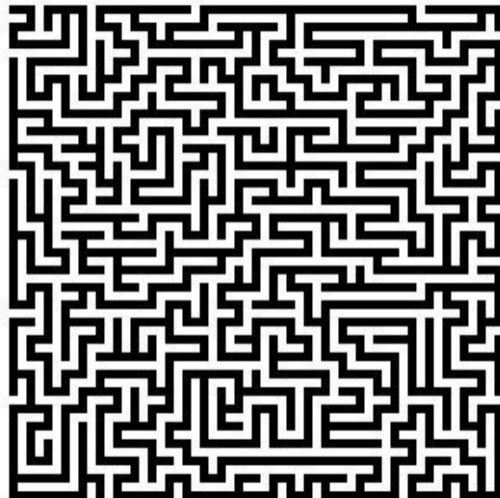
- ▶ **Zyklische** Datenstrukturen haben keine **endliche freie** Repräsentation
 - ▶ Nicht durch endlich viele Konstruktoren darstellbar
 - ▶ Sondern durch Konstruktoren und **Gleichungen**

- ▶ Einfaches Beispiel:

```
ones = 1 : ones
```

- ▶ Nicht-Striktheit erlaubt einfache Definition von Funktionen auf zyklische Datenstrukturen
- ▶ Aber: Funktionen können **divergieren**

Fallbeispiel: Zyklische Datenstrukturen



Quelle: docs.gimp.org

Modellierung eines Labyrinths

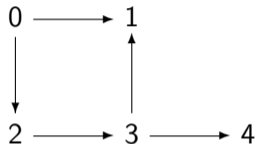
- ▶ Ein **gerichtetes** Labyrinth ist entweder
 - ▶ eine Sackgasse,
 - ▶ ein Weg, oder
 - ▶ eine Abzweigung in zwei Richtungen.

- ▶ Jeder Knoten im Labyrinth hat ein Label α .

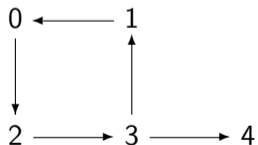
```
data Lab  $\alpha$  = Dead  $\alpha$ 
           | Pass  $\alpha$  (Lab  $\alpha$ )
           | TJnc  $\alpha$  (Lab  $\alpha$ ) (Lab  $\alpha$ )
```

Definition von Labyrinth

Ein einfaches Labyrinth ohne Zyklen:



Ein einfaches Labyrinth mit Zyklen:



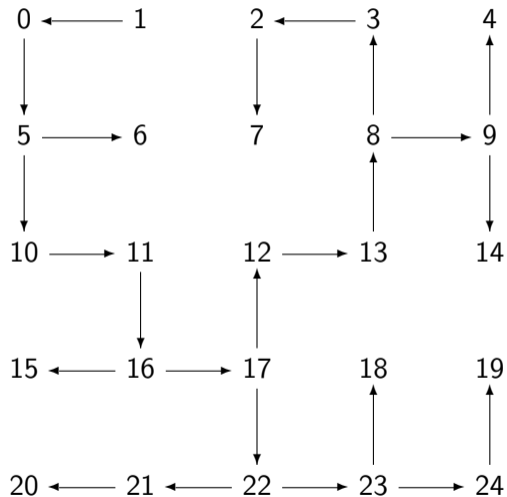
Definition in Haskell:

```
s0 = TJnc 0 s1 s2
s1 = Dead 1
s2 = Pass 2 s3
s3 = TJnc 3 s1 s4
s4 = Dead 4
```

Definition in Haskell:

```
t0 = Pass 0 t2
t1 = Pass 1 t0
t2 = Pass 2 t3
t3 = TJnc 3 t1 t4
t4 = Dead 4
```


Ein Labyrinth (zyklenfrei)



Traversion des Labyrinths

▶ Ziel: **Pfad** zu einem gegebenen **Ziel** finden

▶ Benötigt Pfade und Traversion

▶ Pfade: Liste von Knoten

```
type Path  $\alpha$  = [ $\alpha$ ]
```

▶ Traversion: erfolgreich (Pfad) oder nicht erfolgreich

```
type Trav  $\alpha$  = Maybe [ $\alpha$ ]
```

Traversionsstrategie

- ▶ Geht erstmal von **zyklenfreien** Labyrinth aus
- ▶ An jedem Knoten prüfen, ob Ziel erreicht, ansonsten
 - ▶ an Sackgasse: Fehlschlag (**Nothing**)
 - ▶ an Passagen: Weiterlaufen

```
cons ::  $\alpha \rightarrow \text{Trav } \alpha \rightarrow \text{Trav } \alpha$   
cons _ Nothing      = Nothing  
cons i (Just is) = Just (i: is)
```

- ▶ an Kreuzungen: Auswahl treffen

```
select ::  $\text{Trav } \alpha \rightarrow \text{Trav } \alpha \rightarrow \text{Trav } \alpha$   
select Nothing t = t  
select t _ = t
```

- ▶ Erfordert Propagation von Fehlschlägen (in **cons** und **select**)

Zyklenfreie Traversal

- ▶ Zusammengesetzt:

```
traverse_1 :: (Show α, Eq α) ⇒ α → Lab α → Trav α
traverse_1 t l
  | nid l == t = Just [nid l]
  | otherwise = case l of
    Dead _ → Nothing
    Pass i n → cons i (traverse_1 t n)
    TJnc i n m → cons i (select (traverse_1 t n)
                                (traverse_1 t m))
```



Zyklenfreie Traversal

- ▶ Zusammengesetzt:

```
traverse_1 :: (Show α, Eq α) ⇒ α → Lab α → Trav α
traverse_1 t l
  | nid l == t = Just [nid l]
  | otherwise = case l of
    Dead _ → Nothing
    Pass i n → cons i (traverse_1 t n)
    TJnc i n m → cons i (select (traverse_1 t n)
                                (traverse_1 t m))
```



- ▶ Wie mit Zyklen umgehen?
- ▶ An jedem Knoten prüfen ob schon im Pfad enthalten.

Traversion mit Zyklen

- ▶ Veränderte **Strategie**: Pfad bis hierher übergeben
 - ▶ Pfad muss **hinten** erweitert werden ($O(n)$)
 - ▶ Besser: Pfad **vorne** erweitern ($O(1)$), am Ende umdrehen
- ▶ Wenn **aktueller** Knoten in bisherigen Pfad **enthalten** ist, Fehlschlag
- ▶ Ansonsten wie oben

Traversion mit Zyklen

```
traverse_2 :: Eq  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$  Lab  $\alpha$   $\rightarrow$  Trav  $\alpha$ 
traverse_2 t l = trav_2 l [] where
  trav_2 l p
    | nid l == t = Just (reverse (nid l: p))
    | elem (nid l) p = Nothing
    | otherwise = case l of
      Dead _  $\rightarrow$  Nothing
      Pass i n  $\rightarrow$  trav_2 n (i: p)
      TJnc i n m  $\rightarrow$  select (trav_2 n (i: p)) (trav_2 m (i: p))
```

► Kritik:

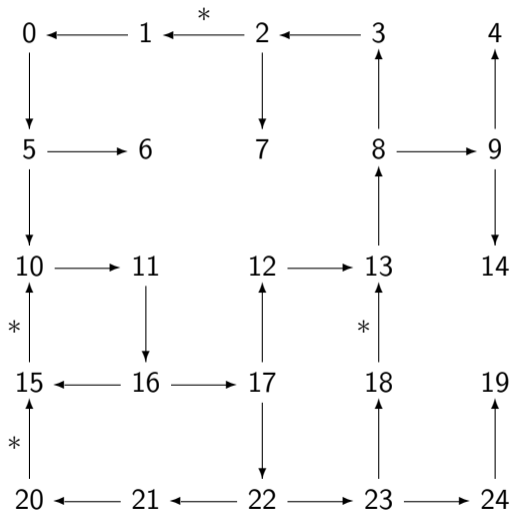
Traversion mit Zyklen

```
traverse_2 :: Eq  $\alpha$   $\Rightarrow$   $\alpha$   $\rightarrow$  Lab  $\alpha$   $\rightarrow$  Trav  $\alpha$ 
traverse_2 t l = trav_2 l [] where
  trav_2 l p
    | nid l == t = Just (reverse (nid l: p))
    | elem (nid l) p = Nothing
    | otherwise = case l of
      Dead _  $\rightarrow$  Nothing
      Pass i n  $\rightarrow$  trav_2 n (i: p)
      TJnc i n m  $\rightarrow$  select (trav_2 n (i: p)) (trav_2 m (i: p))
```

► Kritik:

- Prüfung `elem` immer noch $O(n)$
- Abhilfe: **Menge** der besuchten Knoten getrennt von aufgebautem **Pfad**
- Erfordert effiziente Datenstrukturen für Mengen (`Data.Set`, `Data.IntSet`) → später

Ein Labyrinth (mit Zyklen)



Der allgemeine Fall: variadische Bäume

- ▶ Labyrinth \rightarrow **Graph** oder **Baum**
- ▶ Labyrinth mit mehr als 2 Nachfolgern: **variadischer Baum**

```
data VTree  $\alpha$  = NT  $\alpha$  [VTree  $\alpha$ ]
```

- ▶ Kürzere Definition erlaubt einfachere Funktionen:

```
traverse :: Eq  $\alpha \Rightarrow \alpha \rightarrow$  VTree  $\alpha \rightarrow$  Maybe [ $\alpha$ ]  
traverse t vt = trav [] vt where  
  trav p (NT l vs)  
    | l == t = Just (reverse (l: p))  
    | elem l p = Nothing  
    | otherwise = select (map (trav (l: p)) vs)
```



Traversion verallgemeinert

- ▶ Änderung der Parameter der Traversionsfunktion `trav`:

```
trav :: Eq  $\alpha$   $\Rightarrow$  [(VTree  $\alpha$ , [ $\alpha$ ])]  $\rightarrow$  Maybe [ $\alpha$ ]
```

- ▶ Liste der nächsten **Kandidaten** mit **Pfad** der dorthin führt.
- ▶ Algorithmus:
 - 1 Wenn Liste leer, Fehlschlag
 - 2 Wenn Liste nicht leer, ist der aktuelle Knoten der Kopf der Liste.
 - 3 Prüfe, ob aktueller Knoten das Ziel ist.
 - 4 Wenn nicht am Ziel und aktueller Knoten schon besucht, nächsten Kandidaten traversieren
 - 5 Ansonsten füge Kinder des aktuellem Knotens mit aktuellem Pfad zu Kandidaten hinzu und traversiere weiter

Traversion verallgemeinert

- ▶ Änderung der Parameter der Traversionsfunktion `trav`:

```
trav :: Eq  $\alpha$   $\Rightarrow$  [(VTree  $\alpha$ , [ $\alpha$ ])]  $\rightarrow$  Maybe [ $\alpha$ ]
```

- ▶ Liste der nächsten **Kandidaten** mit **Pfad** der dorthin führt.

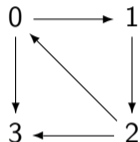
- ▶ Algorithmus:

- 1 Wenn Liste leer, Fehlschlag
- 2 Wenn Liste nicht leer, ist der aktuelle Knoten der Kopf der Liste.
- 3 Prüfe, ob aktueller Knoten das Ziel ist.
- 4 Wenn nicht am Ziel und aktueller Knoten schon besucht, nächsten Kandidaten traversieren
- 5 Ansonsten füge Kinder des aktuellem Knotens mit aktuellem Pfad zu Kandidaten hinzu und traversiere weiter

- ▶ Tiefensuche: Kinder **vorne** anfügen (Kandidatenliste ist ein **Stack**)
- ▶ Breitensuche: Kinder **hinten** anhängen (Kandidatenliste ist eine **Queue**)
- ▶ Andere Bewertungen möglich

Ein einfaches Beispiel

Ein einfaches Labyrinth mit Zyklen:



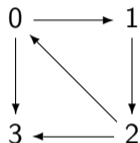
► Gesucht: Pfad von 0 zu 3

Definition in Haskell:

```
100 = NT 0 [101, 103]
101 = NT 1 [102]
102 = NT 2 [100, 103]
103 = NT 3 [100]
```

Ein einfaches Beispiel

Ein einfaches Labyrinth mit Zyklen:



- ▶ Gesucht: Pfad von 0 zu 3
- ▶ Tiefensuche: [0, 1, 2, 3]
- ▶ Breitensuche: [0, 3]

Definition in Haskell:

```
100 = NT 0 [101, 103]
101 = NT 1 [102]
102 = NT 2 [100, 103]
103 = NT 3 [100]
```

Tiefensuche

```
depth_first_search :: Eq  $\alpha$   $\Rightarrow$   $\alpha \rightarrow$  VTree  $\alpha \rightarrow$  Maybe [ $\alpha$ ]  
depth_first_search t vt = trav [(vt, [])] where  
  trav [] = Nothing  
  trav ((NT l ch, p):rest)  
    | l == t    = Just (reverse (l:p))  
    | elem l p  = trav rest  
    | otherwise = trav (more++ rest) where  
                        more = map ( $\lambda c \rightarrow$  (c, l: p)) ch
```

Breitensuche

```
breadth_first_search :: Eq  $\alpha$   $\Rightarrow$   $\alpha \rightarrow$  VTree  $\alpha \rightarrow$  Maybe [ $\alpha$ ]  
breadth_first_search t vt = trav [(vt, [])] where  
  trav [] = Nothing  
  trav ((NT l ch, p):rest)  
    | l == t    = Just (reverse (l:p))  
    | elem l p  = trav rest  
    | otherwise = trav (rest ++ more) where  
                      more = map ( $\lambda c \rightarrow$  (c, l: p)) ch
```


Was zum Nachdenken

Übung 6.1: Wo ist der Stack?

Wo ist der Stack bei `traverse`, und warum läßt sich `traverse` nicht zu Breitensuche verallgemeinern?

Was zum Nachdenken

Übung 6.1: Wo ist der Stack?

Wo ist der Stack bei `traverse`, und warum läßt sich `traverse` nicht zu Breitensuche verallgemeinern?

Lösung: Der Stack ist bei `traverse` der Aufruf-Stack, implizit in dieser Zeile:

```
select (map (trav (l: p)) vs)
```

Hier werden die Kinder in Stack-Order aufgerufen (Kinder der Kinder vor Geschwistern). Die Traversionsfunktion `trav` der Tiefen/Breitensuche hat dagegen keinen Aufruf-Stack; sie ist **endrekursiv** (und damit potenziell effizienter).

II. Vorteile der Nicht-Strikten Auswertung

Zyklische Listen

- ▶ Durch Gleichungen können wir **zyklische** Listen definieren.

```
nats :: [Integer]
nats = natsfrom 0 where
  natsfrom i = i: natsfrom (i+1)
```

- ▶ Repräsentation durch endliche, zyklische Datenstruktur
- ▶ Kopf wird nur **einmal** ausgewertet.

```
fives :: [Integer]
fives = trace "***_Foo!_***" 5 : fives
```

- ▶ Es gibt keine **unendlichen** Listen, es gibt nur Berechnungen von Listen, die nicht terminieren.



Unendliche Weiten?

- ▶ Verschiedene Ebenen:
 - ▶ Mathematisch — unendliche Strukturen (natürliche Zahlen, Listen)
 - ▶ Implementierung — immer endlich (kann unendliche Strukturen **repräsentieren**)
- ▶ Berechnung auf unendlichen Strukturen: Vereinigung der Berechnungen auf allen **endlichen** Teilstrukturen
- ▶ Jede Berechnung hat **endlich** viele Parameter.
- ▶ Daher nicht entscheidbar, ob Liste „unendlich“ (zyklisch) ist:

```
isCyclic :: [a] → Bool
```

Unendliche Listen und Nicht-Striktheit

- ▶ Nicht-Striktheit macht den Umgang mit zyklischen Datenstrukturen einfacher
- ▶ Beispiel: Sieb des Eratosthenes:
 - ▶ Ab wo muss ich sieben, um die n -Primzahl zu berechnen?
 - ▶ Einfacher: Liste **aller** Primzahlen berechnen, davon n -te selektieren.

Fibonacci-Zahlen

- ▶ Aus der Kaninchenzucht.
- ▶ Sollte jeder Informatiker kennen.

```
fib1 :: Integer → Integer
fib1 0 = 1
fib1 1 = 1
fib1 n = fib1 (n-1) + fib1 (n-2)
```

- ▶ Problem: **exponentieller Aufwand**.

Fibonacci-Zahlen

- ▶ Lösung: zuvor berechnete **Teilergebnisse wiederverwenden**.
- ▶ Sei `fibs :: [Integer]` Strom aller Fibonaccizahlen:

```
fibs    ~> [1, 1, 2, 3, 5, 8, 13, 21, 34, 55 .. ]  
tail fibs ~> [1, 2, 3, 5, 8, 13, 21, 34, 55 .. ]  
tail (tail fibs) ~> [2, 3, 5, 8, 13, 21, 34, 55...]
```


Fibonacci-Zahlen

- ▶ Lösung: zuvor berechnete **Teilergebnisse wiederverwenden**.
- ▶ Sei `fibs :: [Integer]` Strom aller Fibonaccizahlen:

```
fibs    ~> [1, 1, 2, 3, 5, 8, 13, 21, 34, 55 .. ]  
tail fibs ~> [1, 2, 3, 5, 8, 13, 21, 34, 55 .. ]  
tail (tail fibs) ~> [2, 3, 5, 8, 13, 21, 34, 55...]
```

- ▶ Damit ergibt sich:

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

- ▶ n -te Fibonaccizahl mit `fibs !! n`:

```
fib2 :: Integer -> Integer  
fib2 n = genericIndex fibs n
```

- ▶ **Aufwand: linear**, da `fibs` nur einmal ausgewertet wird.

Was zum Nachdenken.

Übung 6.1: Fibonacci

Es gibt eine geschlossene Formel für die Fibonacci-Zahlen:

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right)$$

In Haskell (zählt ab 0):

```
fib3 :: Integer -> Integer
```

```
fib3 n = round ((1/sqrt 5)*(((1+ sqrt 5)/2)^(n+1)-((1- sqrt 5)/2)^(n+1)))
```

Was ist hier das Problem?

Was zum Nachdenken.

Übung 6.1: Fibonacci

Es gibt eine geschlossene Formel für die Fibonacci-Zahlen:

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right)$$

In Haskell (zählt ab 0):

```
fib3 :: Integer → Integer
```

```
fib3 n = round ((1/sqrt 5)*(((1+ sqrt 5)/2)^(n+1)-((1- sqrt 5)/2)^(n+1)))
```

Was ist hier das Problem?

Lösung: Die Fließkommaarithmetik wird irgendwann (ab 74) ungenau.

III. Effizienzerwägungen

Beispiel: Listen umdrehen

- ▶ Liste umdrehen, **nicht** endrekursiv:

```
rev' :: [a] → [a]
rev' []      = []
rev' (x:xs) = rev' xs ++ [x]
```

- ▶ Hängt auch noch **hinten** an — $O(n^2)$!

Beispiel: Listen umdrehen

- ▶ Liste umdrehen, **nicht** endrekursiv:

```
rev' :: [a] → [a]
rev' []      = []
rev' (x:xs) = rev' xs ++ [x]
```

- ▶ Hängt auch noch **hinten** an — $O(n^2)$!

- ▶ Liste umdrehen, **endrekursiv** und $O(n)$:

```
rev :: [a] → [a]
rev xs = rev0 xs [] where
  rev0 []      ys = ys
  rev0 (x:xs) ys = rev0 xs (x:ys)
```

- ▶ Schneller weil geringere Aufwandsklasse, nicht nur wg. Endrekursion
- ▶ Frage: ist Endrekursion immer schneller?

Beispiel: Fakultät

- ▶ Fakultät **nicht** endrekursiv:

```
fac1 :: Integer → Integer
fac1 n = if n == 0 then 1 else n * fac1 (n-1)
```

Beispiel: Fakultät

- ▶ Fakultät **nicht** endrekursiv:

```
fac1 :: Integer → Integer
fac1 n = if n == 0 then 1 else n * fac1 (n-1)
```

- ▶ Fakultät endrekursiv:

```
fac2 :: Integer → Integer
fac2 n = fac' n 1 where
  fac' :: Integer → Integer → Integer
  fac' n acc = if n == 0 then acc
               else fac' (n-1) (n*acc)
```

- ▶ `fac1` verbraucht Stack, `fac2` nicht.

Beispiel: Fakultät

- ▶ Fakultät **nicht** endrekursiv:

```
fac1 :: Integer -> Integer
fac1 n = if n == 0 then 1 else n * fac1 (n-1)
```

- ▶ Fakultät endrekursiv:

```
fac2 :: Integer -> Integer
fac2 n = fac' n 1 where
  fac' :: Integer -> Integer -> Integer
  fac' n acc = if n == 0 then acc
               else fac' (n-1) (n*acc)
```

- ▶ `fac1` verbraucht Stack, `fac2` nicht.
- ▶ Ist **nicht** merklich schneller?!

Verzögerte Auswertung und Speicherlecks

- ▶ **Garbage collection** gibt unbenutzten Speicher wieder frei.
 - ▶ **Unbenutzt**: Bezeichnet nicht mehr Speicher im erreichbar
- ▶ Verzögerte Auswertung **effizient**, weil nur bei Bedarf ausgewertet wird
 - ▶ Aber Achtung: Speicherleck!

Verzögerte Auswertung und Speicherlecks

- ▶ **Garbage collection** gibt unbenutzten Speicher wieder frei.
 - ▶ **Unbenutzt**: Bezeichnet nicht mehr Speicher im erreichbar
- ▶ Verzögerte Auswertung **effizient**, weil nur bei Bedarf ausgewertet wird
 - ▶ Aber Achtung: Speicherleck!
- ▶ Eine Funktion hat ein **Speicherleck**, wenn Speicher **unnötig** lange im Zugriff bleibt.
 - ▶ “Echte” Speicherlecks wie in C/C++ nicht möglich.
- ▶ Beispiel: `fac2`
 - ▶ Zwischenergebnisse werden nicht ausgewertet.
 - ▶ Insbesondere ärgerlich bei nicht-terminierenden Funktionen.

Striktheit

- ▶ **Strikte Argumente** erlauben Auswertung **vor** Aufruf
 - ▶ Dadurch **konstanter** Platz bei **Endrekursion**.
- ▶ Erzwungene Striktheit: $\text{seq} :: \alpha \rightarrow \beta \rightarrow \beta$

\perp 'seq' b = \perp

a 'seq' b = b

- ▶ seq vordefiniert (nicht in Haskell definierbar)
- ▶ ($\$!$) :: (a \rightarrow b) \rightarrow a \rightarrow b strikte Funktionsanwendung

```
f $! x = x 'seq' f x
```

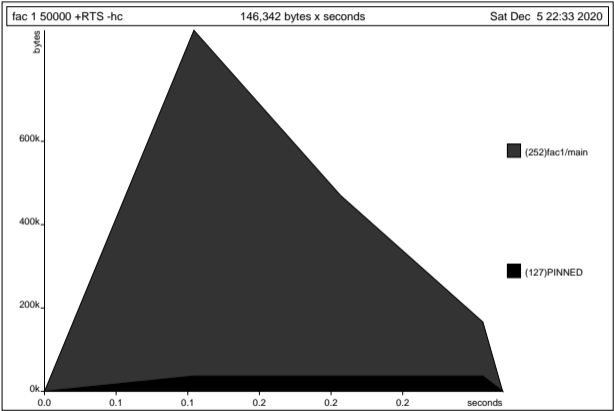
- ▶ ghc macht Striktheitsanalyse
- ▶ Fakultät in konstantem Platzaufwand

```
fac3 :: Integer  $\rightarrow$  Integer
```

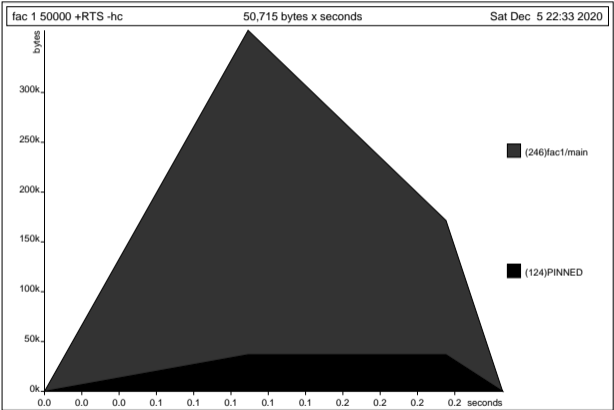
```
fac3 n = fac' n 1 where
```

```
  fac' n acc = seq acc (if n == 0 then acc  
                       else fac' (n-1) (n*acc))
```

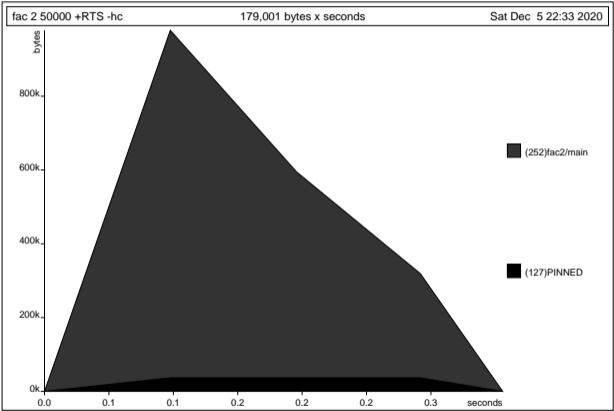
Speicherprofil: fac1 50000, nicht optimiert



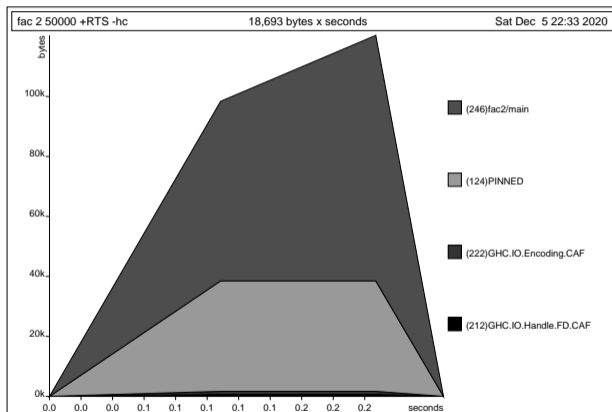
Speicherprofil: fac1 50000, optimiert



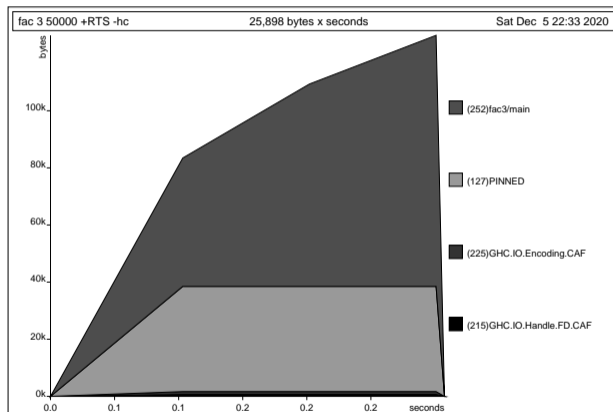
Speicherprofil: fac2 50000, nicht optimiert



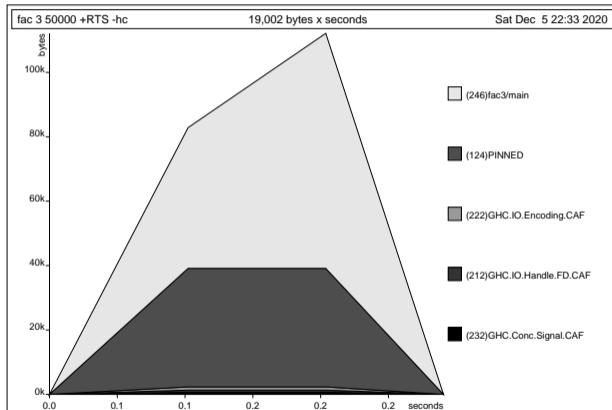
Speicherprofil: fac2 50000, optimiert



Speicherprofil: fac3 50000, nicht optimiert



Speicherprofil: fac3 50000, optimiert



Fakultät als Funktion höherer Ordnung

- ▶ Nicht end-rekursiv mit `foldr`:

```
fac_foldr :: Integer → Integer
fac_foldr i = foldr (*) 1 [1.. i]
```

- ▶ End-rekursiv mit `foldl`:

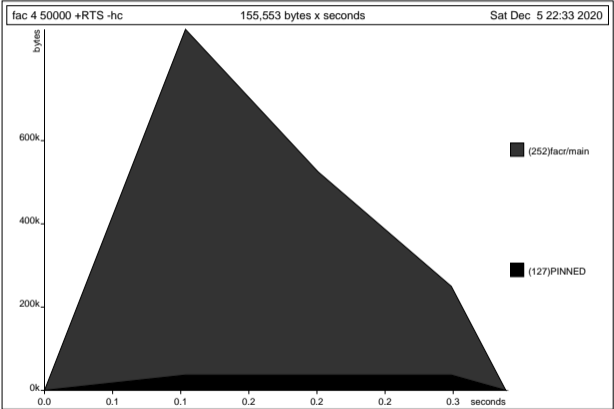
```
fac_foldl :: Integer → Integer
fac_foldl i = foldl (*) 1 [1.. i]
```

- ▶ End-rekursiv und strikt mit `foldl'`:

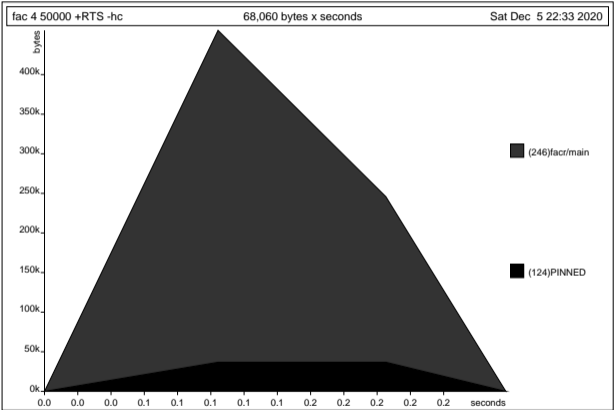
```
fac_foldl' :: Integer → Integer
fac_foldl' i = foldl' (*) 1 [1.. i]
```

- ▶ **Exakt** die gleichen Ergebnisse!

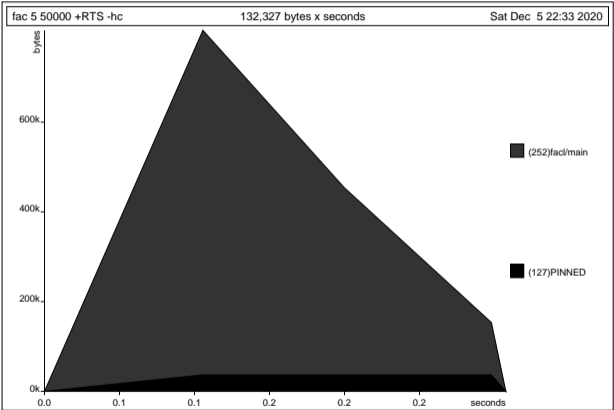
Speicherprofil: foldr 50000, nicht optimiert



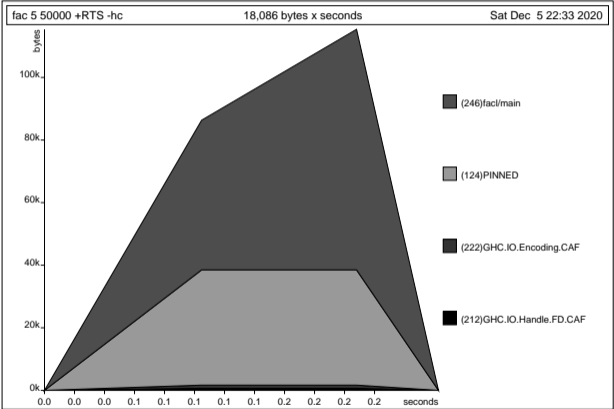
Speicherprofil: foldr 50000, optimiert



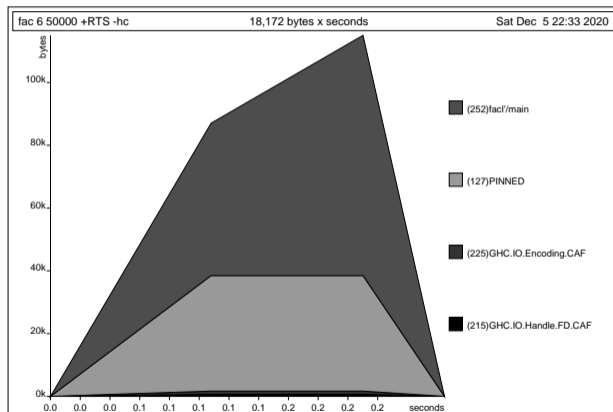
Speicherprofil: foldl 50000, nicht optimiert



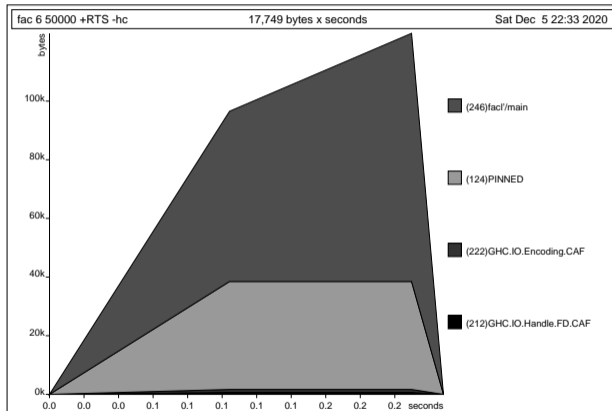
Speicherprofil: foldl 50000, optimiert



Speicherprofil: foldl' 50000, nicht optimiert



Speicherprofil: foldl' 50000, optimiert



Fazit Speicherprofile

- ▶ Endrekursion **nur** bei **strikten Funktionen** schneller
- ▶ Optimierung des *ghc*
 - ▶ Meist ausreichend für Striktheitsanalyse
 - ▶ Aber **nicht** für Endrekursion
- ▶ Deshalb:
 - ▶ **Manuelle** Überführung in Endrekursion **sinnvoll**
 - ▶ **Compiler-Optimierung** für Striktheit nutzen

Zusammenfassung

- ▶ Rekursive Datentypen können **zyklische Datenstrukturen** modellieren
 - ▶ Das Labyrinth — Sonderfall eines **variadischen Baums**
 - ▶ Unendliche Listen — nützlich wenn Länge der Liste nicht im voraus bekannt
- ▶ Effizienzerwägungen:
 - ▶ Überführung in Endrekursion sinnvoll, Striktheit durch Compiler