



## Praktische Informatik 3: Funktionale Programmierung

### Vorlesung 6 (22.11.2022): Rekursive Datenstrukturen

Christoph Lüth



Deutsches  
Forschungszentrum  
für Künstliche  
Intelligenz GmbH



Universität  
Bremen

Wintersemester 2022/23

14:10:14 2023-01-10

1 [47]



## Organisatorisches

- Die Vorlesung am **06.12.2022** findet im **NW2 A0242** statt.

1 Pi3 WS 22/23

2 [47]



## 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

1 Pi3 WS 22/23

3 [47]



## Inhalt

### ► Rekursive Datentypen und zyklische Daten

- ... und wozu sie nützlich sind
- Fallbeispiel: Labyrinth

### ► Effizienzerwägungen

#### Lernziele

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

1 Pi3 WS 22/23

4 [47]



## 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 zyklischen Datenstrukturen

### ► Aber: Funktionen können **divergieren**

1 Pi3 WS 22/23

5 [47]



## I. 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.

1 Pi3 WS 22/23

7 [47]



## Unendliche Weiten?

### ► Verschiedene Ebenen:

- Mathematisch — unendliche Strukturen (natürliche Zahlen, Listen)
- Implementierung — immer endlich (kann unendliche Strukturen **repräsentieren**)

### ► Berechnungen 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
```

1 Pi3 WS 22/23

8 [47]







## Ein einfaches Beispiel

Ein einfaches Labyrinth mit Zyklen:



Definition in Haskell:

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

► Gesucht: Pfad von 0 zu 3

► Tiefensuche: [0, 1, 2, 3]

► Breitensuche: [0, 3]

• P13 WS 22/23

25 [47]

DFU

## Tiefensuche

```
depth_first_search :: Eq α => α → VTree α → Maybe [α]
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 (λc→ (c, l: p)) ch
```

• P13 WS 22/23

26 [47]

DFU

## Breitensuche

```
breadth_first_search :: Eq α => α → VTree α → Maybe [α]
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 (λc→ (c, l: p)) ch
```

☞ Siehe Übung 6.2

• P13 WS 22/23

27 [47]

DFU

## III. Effizienzerwägungen

### Beispiel: Listen umdrehen

► Liste umdrehen, **nicht** endrekursiv:

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

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

► 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?

• P13 WS 22/23

29 [47]

DFU

### 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?!

• P13 WS 22/23

30 [47]

DFU

### Verzögerte Auswertung und Speicherlecks

► **Garbage collection** gibt **unbenutzten** Speicher wieder frei.

► **Unbenutzt**: Bezeichner 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** auswertet.

► Insbesondere ärgerlich bei **nicht-terminierenden** Funktionen.

• P13 WS 22/23

31 [47]

DFU

### Striktheit

► **Strikte Argumente** erlauben Auswertung **vor** Aufruf

► Dadurch **konstanter** Platz bei **Endrekursion**.

► Erzwungene Striktheit: **seq** ::  $\alpha \rightarrow \beta \rightarrow \beta$

```
λ 'seq' b = ⊥
a 'seq' b = b
```

► seq vordefiniert (nicht in Haskell definierbar)

►  $(\$!) :: (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$  strikte Funktionsanwendung

►  $f \$! x = x 'seq' f x$

► **ghc** macht **Striktheitsanalyse**

► Fakultät in konstantem Platzaufwand

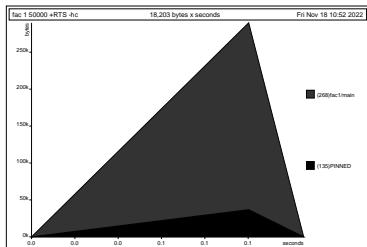
```
fac3 :: Integer → Integer
fac3 n = fac' n 1 where
  fac' n acc = seq acc (if n == 0 then acc
                        else fac' (n-1) (n*acc))
```

• P13 WS 22/23

32 [47]

DFU

### Speicherprofil: fac1 50000, nicht optimiert

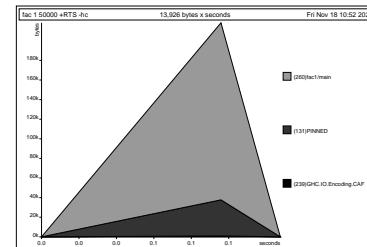


PI3 WS 22/23

33 [47]

DFU

### Speicherprofil: fac1 50000, optimiert

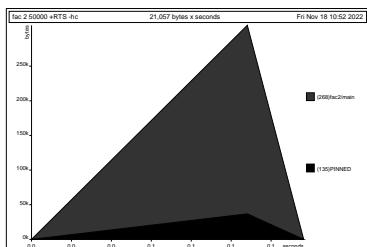


PI3 WS 22/23

34 [47]

DFU

### Speicherprofil: fac2 50000, nicht optimiert

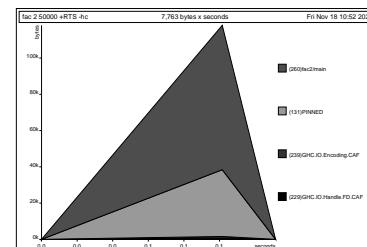


PI3 WS 22/23

35 [47]

DFU

### Speicherprofil: fac2 50000, optimiert

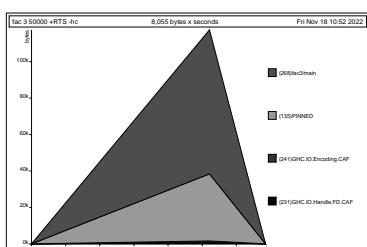


PI3 WS 22/23

36 [47]

DFU

### Speicherprofil: fac3 50000, nicht optimiert

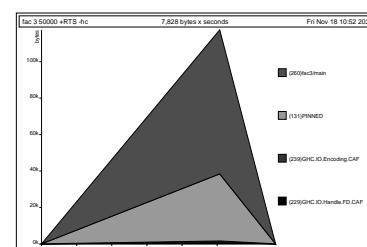


PI3 WS 22/23

37 [47]

DFU

### Speicherprofil: fac3 50000, optimiert



PI3 WS 22/23

38 [47]

DFU

### 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]
```

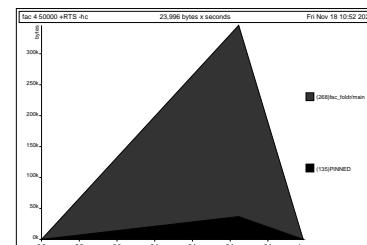
- **Exakt** die gleichen Ergebnisse!

PI3 WS 22/23

39 [47]

DFU

### Speicherprofil: foldr 50000, nicht optimiert

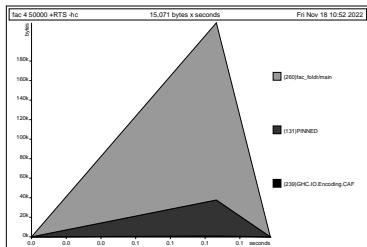


PI3 WS 22/23

40 [47]

DFU

## Speicherprofil: foldr 50000, optimiert

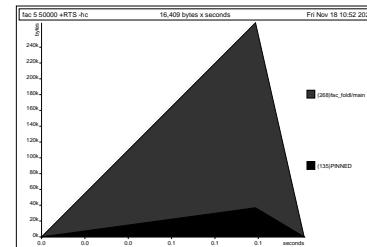


PI3 WS 22/23

41 [47]

DFU

## Speicherprofil: foldl 50000, nicht optimiert

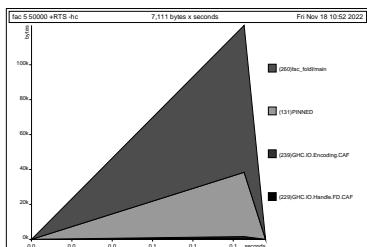


PI3 WS 22/23

42 [47]

DFU

## Speicherprofil: foldl 50000, optimiert

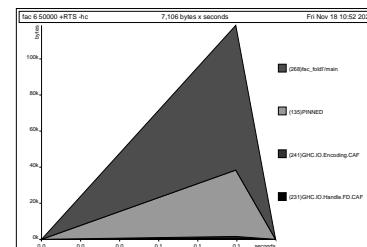


PI3 WS 22/23

43 [47]

DFU

## Speicherprofil: foldl' 50000, nicht optimiert

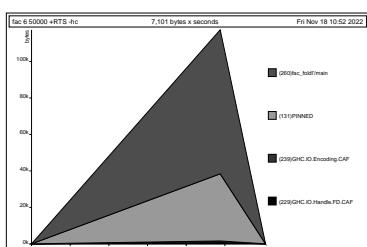


PI3 WS 22/23

44 [47]

DFU

## Speicherprofil: foldl' 50000, optimiert



PI3 WS 22/23

45 [47]

DFU

## Fazit Speicherprofile

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

PI3 WS 22/23

46 [47]

DFU

## 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

PI3 WS 22/23

47 [47]

DFU