Praktische Informatik 3: Funktionale Programmierung

Vorlesung 5 vom 13.11.2018: Rekursive und zyklische Datenstrukturen

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Organisatorisches

- Abgabefrist Übungsblätter ab jetzt bis **Dienstag 12:00**

- Termin Probeklausur: **17.12.18** ab **10:15**

Fahrplan

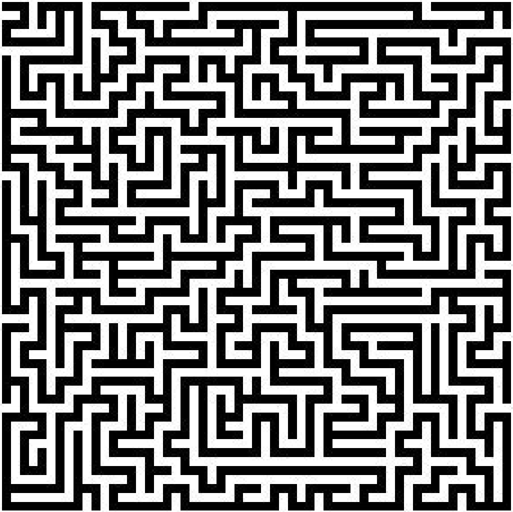
- **Teil I: Funktionale Programmierung im Kleinen**
  - Einführung
  - Funktionen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Zyklische Datenstrukturen
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale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

Inhalt

- **Rekursive** Datentypen und **zyklische** Daten
  - . . . und wozu sie nützlich sind
  - Fallbeispiel: Labyrinth
- Datentypen und Polymorphie in anderen Sprachen
- Performance-Aspekte

Rekursive und Zyklische Datenstrukturen

Fallbeispiel: Zyklische Datenstrukturen



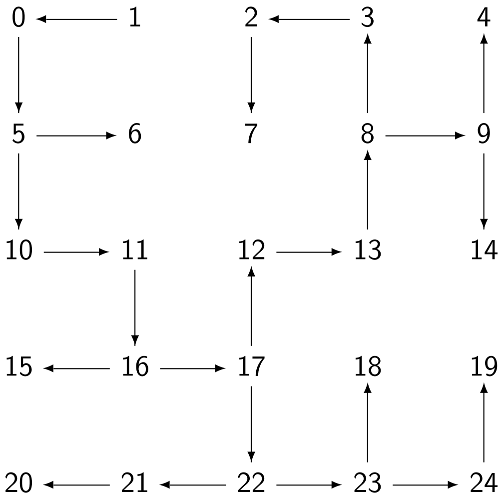
Quelle: docs.gimp.org

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.

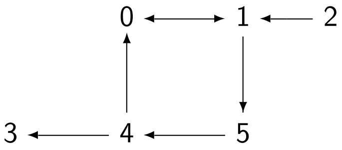
$$
\begin{aligned}
\textbf{data}\ \mathsf{Lab}\ \alpha = {} & \mathsf{Dead}\ \alpha \\
\mid {} & \mathsf{Pass}\ \alpha\ (\mathsf{Lab}\ \alpha) \\
\mid {} & \mathsf{TJnc}\ \alpha\ (\mathsf{Lab}\ \alpha)\ (\mathsf{Lab}\ \alpha)
\end{aligned}
$$

Ein Labyrinth (zyklenfrei)



Definition von Labyrinthen

Ein einfaches Labyrinth:



Definition in Haskell:

```haskell
lab0  = Pass 0 lab1
lab1  = TJnc 1 lab0 lab5
lab2  = Pass 2 lab1
lab3  = Dead 3
lab4  = TJnc 4 lab0 lab3
lab5  = Pass 5 lab4
```

- **Rekursiv**!

- Freundlichere Notation:

```
0 ⟶ 1
1 ⟶ 0 5
2 ⟶ 1
3 ⟶
4 ⟶ 0 3
5 ⟶ 4
```

Traversion des Labyrinths

- Ziel: **Pfad** zu einem gegeben **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 :: α → Trav α → Trav α
cons _ Nothing     = Nothing
cons i (Just is) = Just (i: is)
```

  - an Kreuzungen: Auswahl treffen

```
select :: Trav α → Trav α→ Trav α
select Nothing t = t
select t       _ = t
```

- Erfordert Propagation von Fehlschlägen (in cons und select)

Zyklenfreie Traversion

- Zusammengesetzt:

```
traverse_1 :: 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 Traversion

- Zusammengesetzt:

```
traverse_1 :: 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 α⇒ α→ Lab α→ Trav α
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 _ → Nothing
      Pass i n → trav_2 n (i: p)
      TJnc i n m → select (trav_2 n (i: p)) (trav_2 m (i: p))
```

- Kritik:

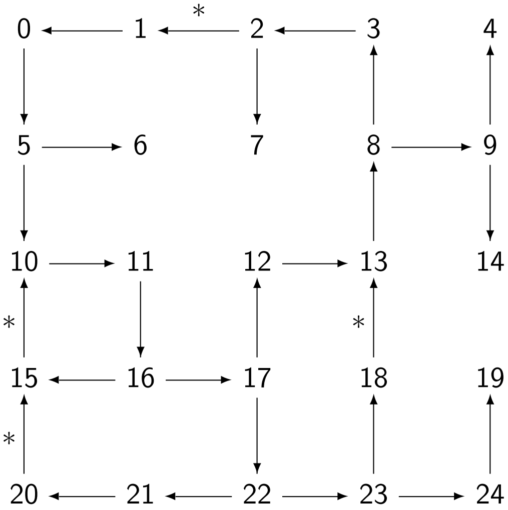
Traversion mit Zyklen

```
traverse_2 :: Eq α⇒ α→ Lab α→ Trav α
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 _ → Nothing
      Pass i n → trav_2 n (i: p)
      TJnc i n m → select (trav_2 n (i: p)) (trav_2 m (i: p))
```

- Kritik:
  - Prüfung elem immer noch $O(n)$
  - Knoten können mehrfach besucht werden
  - Abhilfe: Menge der besuchten Knoten getrennt von aufgebautem Pfad
  - Erfordert effiziente Datenstrukturen für Mengen (Data.Set, Data.IntSet)

  $\longrightarrow$ später

Ein Labyrinth (mit Zyklen)



Labyrinthe lesen

- Zerlegung in zwei Schritte:

```
readLab :: (Read α, Eq α, Show α)⇒ String→ α→ Lab α
readLab = makeLab ∘ parseString
```

- parseString zerlegt Zeichenkette in Zeilenbestandteile

```
parseString :: String→ [(String, [String])]
```

- makeLab erzeugt daraus das Labyrinth

```
makeLab :: (Read α, Eq α, Show α) ⇒
           [(String, [String])]→ α→ Lab α
```

Labyrinthe konstruieren

- Problem: Zyklische Referenzen
- Abbildung von α auf Lab α wird aus Argument konstruiert
- Das gesamte Labyrinth ist der Fixpunkt der Konstruktion

Labyrinthe konstruieren

- Problem: Zyklische Referenzen
- Abbildung von α auf Lab α wird aus Argument konstruiert
- Das gesamte Labyrinth ist der Fixpunkt der Konstruktion

```
makeLab :: (Read α, Eq α, Show α) ⇒
           [(String, [String])]→ α→ Lab α
makeLab vs id =
  let mk_lab map [] = []
      mk_lab map ((s, ts):rest) =
        let src = read s
            get v = fromJust (lookup (read v) map)
            l = case ts of
                  [] → Dead src
                  [v] → Pass src (get v)
                  [v1, v2] → TJnc src (get v1) (get v2)
                  _ → error ("Too many edges from "++ show src++ ": "
        in  (src, l): mk_lab map rest
      map = mk_lab map vs
  in fromMaybe (error ("Undefined label: "++ show id)) (lookup id map)
```

Der allgemeine Fall: variadische Bäume

- Labyrinth ⟶ **Graph** oder **Baum**
- Labyrinth mit mehr als 2 Nachfolgern: **variadischer Baum**

```
data VTree α = NT α [VTree α]
```

  - Kürzere Definition erlaubt einfachere Funktionen:

```
traverse :: Eq α⇒ α→ VTree α→ Maybe [α]
traverse t vt = trav vt [] where
  trav (NT l vs) p
    | l == t = Just (reverse (l: p))
    | elem l p = Nothing
    | otherwise = select (travList (l: p) vs)
  travList p [] = []
  travList p (nt: nts) = trav nt p : travList p nts
```

Vorteile der Nicht-Strikten Auswertung

Unendliche Listen

- Auch Listen müssen nicht **endlich repräsentierbar** sein:
  - E.g. Unendliche Liste [2,2,2,...]

```
twos = 2 : twos
```

  - Liste der natürlichen Zahlen:

```
nat  = [1..]
```

  - Bildung von unendlichen Listen:

```
cycle :: [a]→ [a]
cycle xs = xs ++ cycle xs
```

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

```
cycle (trace "Foo!" [5])
```

- Nützlich für Listen mit a priori unbekannter Länge

Berechnung der ersten n Primzahlen

- Eratosthenes — aber bis wo sieben?

Berechnung der ersten n Primzahlen

- Eratosthenes — aber bis wo sieben?
- Lösung: Berechnung **aller** Primzahlen, davon die ersten n.

```
sieve :: [Integer]→ [Integer]
sieve (p:ps) = p: sieve (filter ps) where
  filter (q: qs)
    | q `mod` p ≠ 0 = q: filter qs
    | otherwise     = filter qs
```

```
allprimes :: [Integer]
allprimes = sieve [2..]
```

- Von allen Primzahlen die ersten:

```
primes :: Int→ [Integer]
primes n = take n allprimes
```

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:

```haskell
fibs :: [Integer]
fibs = 1 : 1 : zipPlus fibs (tail fibs) where
  zipPlus (a:as) (b:bs) = a+b: zipPlus as bs
```

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

```haskell
fib2 :: Integer→ Integer
fib2 n = genericIndex fibs n
```

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

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:

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

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

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

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

- Beispiel: `last (rev [1..10000])`
- **Schneller** — warum?

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:

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

- Fakultät endrekursiv:

```haskell
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**: Bezeichner nicht mehr 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**: Bezeichner nicht mehr 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.

Striktheit

- **Strikte Argumente** erlauben Auswertung **vor** Aufruf
  - Dadurch **konstanter** Platz bei **Endrekursion**.
- Erzwungene Striktheit: seq :: $\alpha \to \beta \to \beta$

$$\perp \text{ `seq` } b = \perp$$
$$a \text{ `seq` } b = b$$

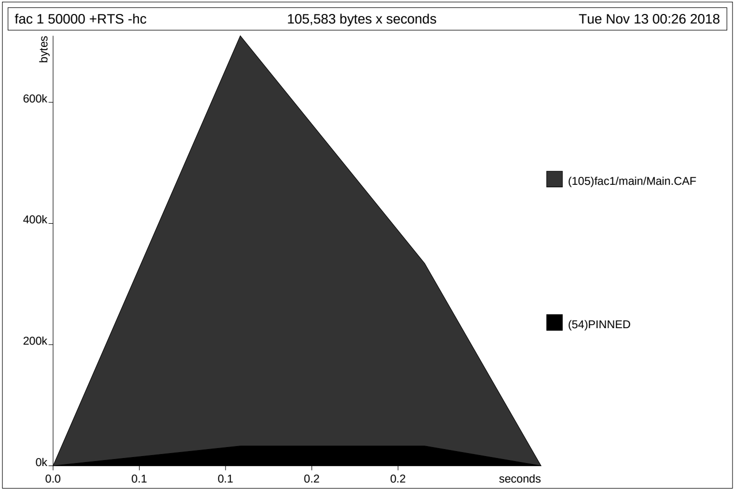
  - seq vordefiniert (nicht in Haskell definierbar)
  - (\$!) :: (a→ b)→ a→ b strikte Funktionsanwendung

```haskell
f $! x = x `seq` f x
```

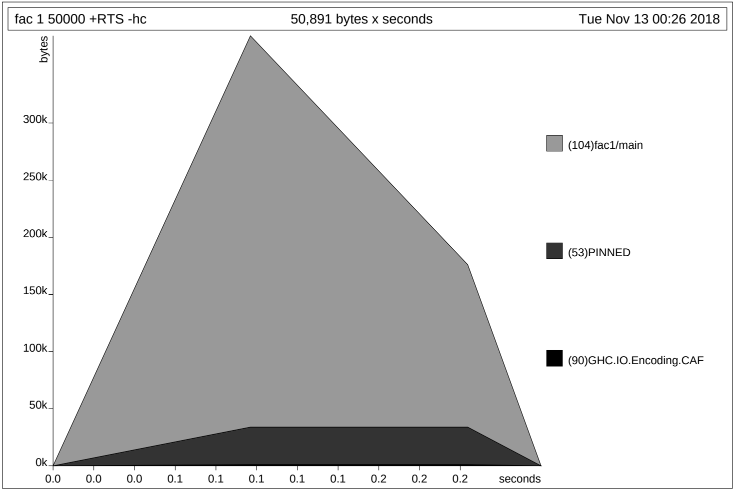
  - ghc macht Striktheitsanalyse
- Fakultät in konstantem Platzaufwand

```haskell
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))
```

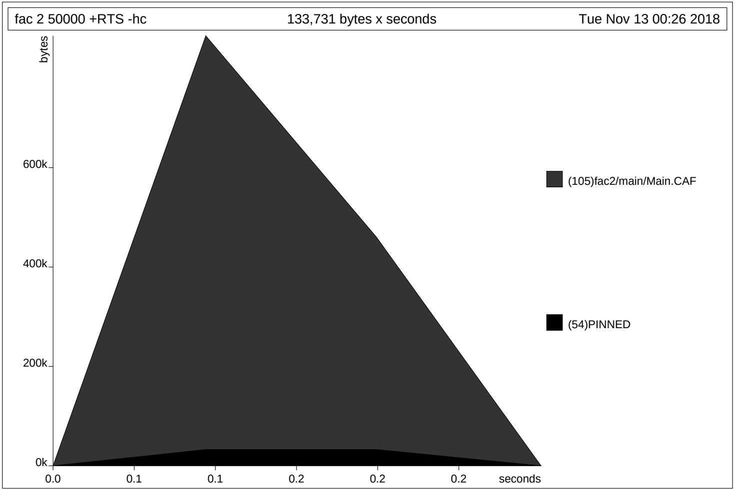
Speicherprofil: `fac1 50000`, nicht optimiert



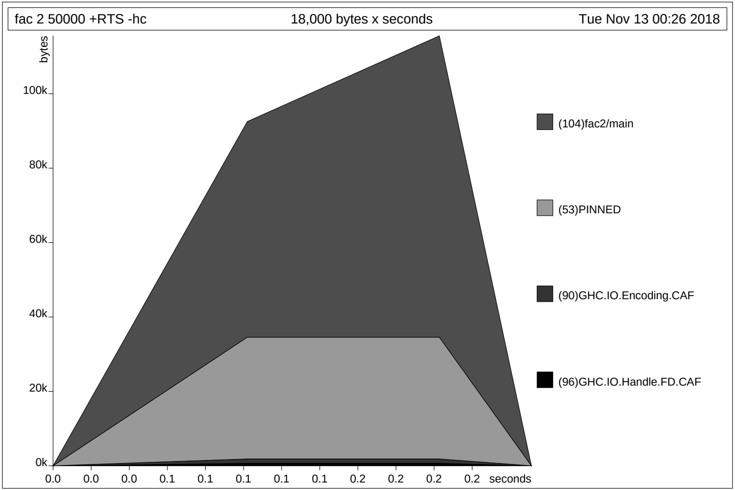
Speicherprofil: `fac1 50000`, optimiert



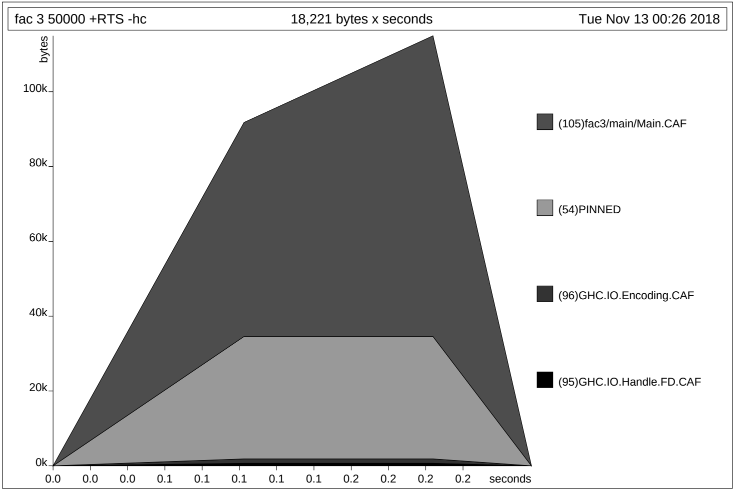
Speicherprofil: `fac2 50000`, nicht optimiert



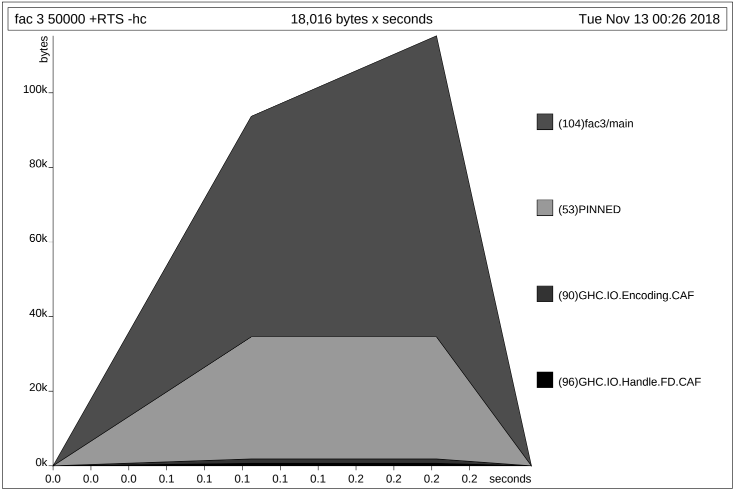
Speicherprofil: `fac2 50000`, optimiert



Speicherprofil: `fac3 50000`, nicht optimiert



Speicherprofil: `fac3 50000`, optimiert



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

Datentypen in anderen Programmiersprachen

Datentypen in C

- **C**: Produkte, Aufzählungen, keine rekursiven Typen
- Rekursion **nur** durch **Zeiger**

```c
typedef struct list_t {
      int elem;
      struct list_t *next;
      } *list;
```

- Konstruktoren **nutzerimplementiert**

```c
list cons(int hd, list tl)
{ list l;
  if ((l= (list)malloc(sizeof(struct list_t)))== NULL) {
    printf("Out of memory\n"); exit(-1);
  }
  l-> elem= hd; l-> next= tl;
  return l;
}
```

Polymorphie in anderen Programmiersprachen: C

- "Polymorphie" in C: **void** *

```
typedef struct list_t {
      void *head;
      struct list_t *next;
      } *list;
```

- Gegeben:

```
int x = 7;
struct list_t l1  = { &x, NULL};
```

- l2.head hat Typ **void** *:

```
int y;
y = *(int *)l1.head;
```

- Nicht möglich: head direkt als Skalar (e.g. int)
- C++: Templates

Datentypen in Java

- Nachbildung durch Klassen, z.B. für Listen:

```
class List {
  public List(Object el, List tl) {
    this.elem= el;
    this.next= tl;
  }
  public Object elem;
  public List   next;
```

- Länge (iterativ):

```
  int length() {
    int i= 0;
    for (List cur= this; cur != null; cur= cur.next)
      i++;
    return i;
  }
```

Polymorphie in Java

- Polymorphie in **Java**: Methode auf alle Subklassen anwendbar
  - Manuelle Typkonversion nötig, fehleranfällig
- Neu ab Java 1.5: **Generics**
  - Damit **parametrische Polymorphie** möglich

```
class AbsList<T> {
  public AbsList(T el, AbsList<T> tl) {
    this.elem= el;
    this.next= tl;
  }
  public T elem;
  public AbsList<T> next;
```

Polymorphie in Java

- Typkorrekte Konkatenenation:

```
void concat(AbsList<T> o)
{
  AbsList<T> cur= this;
  while (cur.next != null) cur= cur.next;
  cur.next= o;
}
```

- **Nachteil:** Benutzung umständlich, weil keine Typherleitung

```
AbsList<Integer> l=
  new AbsList<Integer>(new Integer(1),
    new AbsList<Integer>(new Integer(2), null));
```

- **Vorteil:** Typkorrektheit sichergestellt:

```
AbsList<Character> l3 =
    new AbsList<Character>(new Character('a'), null);
l.concat(l3); // Does not work
```

Ad-Hoc Polymorphie in Java

- **interface** und **abstract class**
- Flexibler in Java: beliebig viele Parameter etc.
- Eingeschränkt durch Vererbungshierarchie
- Ähnliche Standardklassen
  - toString
  - equals und ==, keine abgeleitete strukturelle Gleichheit

Datentypen in Python

- **Listen** und **Tupel** fest eingebaut
- Diverse Funktionen auf Listen
  - Methoden (**stateful**) vs. Funktionen
  - Bsp. sort vs. **sorted**
- Definition eigener Typen über Klassen

Polymorphie in Pyton

- In Python werden Typen zur **Laufzeit** geprüft (**dynamic typing**)
- **duck typing**: strukturell gleiche Typen sind gleich
- Polymorphie durch Klassen
- Statt Interfaces kennt Python **Mixins**
  - Abstrakte Klassen ohne Oberklasse

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