Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 24.11.09:
Unendliche Datentypen und Graphen

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- Teil I: Monaden und fortgeschrittene Typen
- Teil II: Fortgeschrittene Datenstrukturen
  - Der Zipper
  - Ströme, Graphen, unendliche Datenstrukturen
- Teil III: Nebenläufigkeit
- Teil IV: The Future of Programming

Das Tagesmenü

- Reprise: Ströme (uendliche Listen)
- Doppelt verkettete Listen
- Graphen

Unendliche Datenstrukturen: Ströme

- Ströme: Unendliche Listen

```
data Stream a = Stream { hd :: a
                       , tl :: Stream a
                       }
```

- Observatoren:

```
hd :: Stream a → a
tl :: Stream a → Stream a
```


Bsp: Fibonacci-Zahlen

- Fibonacci-Zahlen als Strom
- Sei `fibs :: [Integer]` Strom aller Fib'zahlen:

```
            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 :: Stream Integer
fibs = Stream 1 (Stream 1 $ zipS (+) fibs (tl fibs))
```

- `n`-te Fibonaccizahl mit `ith n fibs`
- Aufwand: linear, da `fibs` nur einmal ausgewertet wird.

Doppelt Verkettete Listen

- In Haskell wie in Java/C: Zeiger auf Vorgänger, Nachfolger

```
data DList a = DLNode { prev :: DList a
                      , val  :: a
                      , next :: DList a }
             | DLEmpty
```

  - **deriving** (Eq, Show) ???
- Kein freier Datentyp: es gelten Invarianten

```
d /= DLEmpty && next d /= DLEmpty
                       ⇒ prev (next d)= d
d /= DLEmpty && prev d /= DLEmpty
                       ⇒ next (prev d)= d
```


Listen erzeugen

- Doppelt verkettete Listen erzeugen:

```
fromList :: [a]→ DList a
fromList as = mkDList DLEmpty as

mkDList :: DList a→ [a]→ DList a
mkDList currprev []     = DLEmpty
mkDList currprev (x:xs) =
  let here = DLNode currprev x (mkDList here xs)
  in  here
```

- Problem: Knoten einfügen/löschen
  - Einfache falsche Lösung
  - Richtige Lösung ist $O(n)$

Zusammenfassung

- Vorteile:
  - Vorgänger/Nachfolger $O(1)$
- Nachteile:
  - Einfügen/Löschen $O(n)$

Graphen als unendliche Datentypen

- Ein Graph ist eine Liste von Knoten

```
data Node a b   = Node a [Vertex a b]
data Vertex a b = Vertex b (Node a b)
type Graph a b  = [Node a b]
```

- Damit Beispielgraph:

```
g :: Graph String String
g = let n1 = Node "a" [Vertex "R" n2]
        n2 = Node "b" [Vertex "L" n1, Vertex "D" n3]
        n3 = Node "c" [Vertex "U" n1]
    in g
```

- Problem: Einfügen/Löschen (King & Launchbury, 1995)

Graphen als induktive Datenstrukturen

- Martin Erwig (2001)— Functional Graph Library (FGL)
- Ideen:
  1. Knoten haben explizite Identität
  2. Graph ist induktiv definiert
- Ein Graph ist
  - entweder leer
  - oder Erweiterung eines Graphen (Kontext)

```
type Node    = Int
type Adj b   = [(b, Node)]
type Ctx a b = (Adj b, Node, a, Adj b)

data Gr a b  = Empty | Ctx a b :& Gr a b
```


Pattern Matching

- Graph kein freier Datentyp — Beispiel
- Datentyp Graph muss abstrakt sein
- Fallunterscheidung auf Konstruktoren von Graph nicht möglich
- Fälle:
  - Leerer Graph

```
isEmpty :: Gr a b→ Bool
```

  - Nicht-leerer Graph: Kontext plus Rest

```
matchAny :: Gr a b→ (Ctx a b, Gr a b)
```


Einfache Funktionen

- Match auf einen bestimmten Knoten:

```
match :: Node→ Gr a b→ (Maybe (Ctx a b), Gr a b)
```

  Invariante:

$$match\ v\ g = (Just(is, w, l, os), h) \Longrightarrow v = w$$

- Map:

```
gmap :: (Ctx a b→ Ctx c d)→ Gr a b→ Gr c d
gmap f g | isEmpty g = Empty
         | otherwise = f c :& (gmap f g') where
             (c, g') = matchAny g
```

- Damit Umkehr aller Kanten:

```
swap :: Ctx a b→ Ctx a b
swap (p, v, l, s) = (s, v, l, p)
grev :: Gr a b→ Gr a b
grev = gmap swap
```


Beweise von Eigenschaften

- Datentyp Gr a b ist nicht frei, aber induktiv
- Induktion als zulässiges Beweisprinzip:

$$\frac{empty\ g \longrightarrow P\ g \quad \forall c\ g.\ P\ g \longrightarrow P(c : \& g)}{\forall g.P\ g}$$

- Damit zeigen:

$$\begin{array}{rcll} gmap\ f.gmap\ f' & = & gmap\ (f.f') & \text{(gmap fusion)} \\ grev.grev & = & id & \text{(grev inv)} \end{array}$$

Tiefensuche

- Aufspannenden Baum in Tiefensuche

```
data Tree a = Tree a [Tree a]

df :: [Node]→ Gr a b→ ([Tree Node], Gr a b)
df [] g       = ([], g)
df (v:vs) g = case match v g of
     (Just c, h) → (Tree v t1: t2, g2)
                   where (t1, g1)= df (suc c) h
                         (t2, g2)= df vs g1
     (Nothing, h) → df vs h

dff :: [Node]→ Gr a b→ [Tree Node]
dff vs g = fst (df vs g)
```

- Anwendung: SCC (stark verbundene Komponenten)

Breitensuche

- Aufspannenden Baum in Breitensuche
  - Problem: Baum wächst nach 'unten' — daher Pfade

```
type Path  = [Node]
type RTree = [Path]

bft :: Node→ Gr a b→ RTree
bft v = bf [[v]]

bf :: [Path]→ Gr a b→ RTree
bf [] g = []
bf (p@(v:_):ps) g = case match v g of
  (Just c, h)  → p:bf (ps ++ map (:p) (suc c)) h
  (Nothing, h) → bf ps h
```

- Verbesserung: Queue Path statt [Path] benutzen
- Anwendung: kürzester Pfad

FGL als Bücherei

- Viele weitere Algorithmen
- Graphen als Klasse, verschiedene Implementationen
- Hackage: Data.Graph.Inductive
- Mehr hier:
  `http://web.engr.oregonstate.edu/~erwig/fgl/haskell/`

Zusammenfassung

- Unendliche Datenstrukturen realisiert durch Referenzen
- Programmierung: Observatoren/Destruktoren vs. Konstruktoren
- Beispiel: doppelt verkettete Listen
- Graphen in Haskell
  - Beispiel für induktive, aber nicht freie Datenstruktur
  - Kompakte Darstellung, effiziente Algorithmen möglich
- Nächste Woche: Nebenläufigkeit