Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 17.11.09: Der Zipper

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 Problem

- Funktional = kein Zustand
- Wie destruktiver Update?
  - Manipulation innerhalb einer Datenstruktur
  - Beispiel: ZeilenorientierterEditor, Abstrakte Syntax
- Lösung: Der Zipper
  - Keine feste Datenstruktur, sondern ein Schema

Ein einfacher Editor

- Datenstrukturen:

```haskell
type Text   = [String]
data Pos    = Pos { line :: Int, col :: Int}
data Editor = Ed  { text   :: Text
                  , cursor :: Pos }
```

- Operationen: Cursor bewegen (links)

```haskell
go_left :: Editor → Editor
go_left Ed{text= t, cursor= c}
  | col c ≡ 0 = error "At start of line"
  | otherwise =
     Ed{text= t, cursor=c{col= col c− 1}}
```

Beispieloperationen

- Text rechts einfügen:

```
insert_right :: Editor→ String→ Editor
insert_right Ed{text= t, cursor= c} text =
  let (as, bs) = splitAt (col c) (t !! line c)
  in  Ed{text= updateAt (line c) t
                        (as ++ text++ bs),
         cursor= c}
```

```
updateAt :: Int→ [a]→ a→ [a]
updateAt n as a = case splitAt n as of
  (bs, [])     → error "updateAt: list too short."
  (bs, _:cs)  → bs ++ a : cs
```

- Problem: Aufwand für Manipulation

Manipulation strukturierter Datentypen

- Anderes Beispiel: *n*-äre Bäume (rose trees)

```haskell
data Tree a = Leaf a
            | Node [Tree a]
            deriving Show
```

  - Bsp: Abstrakte Syntax von einfachen Ausdrücken

- Update auf Beispielterm $t = a * b - c * d$: ersetze b durch $x + y$

```haskell
t = Node [ Leaf "-"
         , Node [Leaf "*", Leaf "a", Leaf "b"]
         , Node [Leaf "*", Leaf "c", Leaf "d"]
         ]
```

Der Zipper

- Idee: Kontext nicht wegwerfen!
- Nicht: **type** Path = [Int]
- Sondern:

```
data Ctxt a = Empty
            | Cons [Tree a] (Ctxt a) [Tree a]
```

  - Kontext ist 'inverse Umgebung' (*"Like a glove turned inside out"*)
- Loc a ist Baum mit Fokus

```
newtype Loc a = Loc (Tree a, Ctxt a)
```

  - Warum newtype?

Zipping Trees: Navigation

- Fokus nach links

```
go_left :: Loc a→ Loc a
go_left (Loc(t, c)) = case c of
 Empty → error "go_left␣at␣empty"
 Cons (l:le) up ri → Loc(l, Cons le up (t:ri))
 Cons [] _ _        → error "go_left␣of␣first"
```

- Fokus nach rechts

```
go_right :: Loc a→ Loc a
go_right (Loc(t, c)) = case c of
 Empty → error "go_right␣at␣empty"
 Cons le up (r:ri) → Loc(r, Cons (t:le) up ri)
 Cons _ _ []       → error "go_right␣of␣last"
```

Zipping Trees: Navigation

- Fokus nach oben

```
go_up :: Loc a→ Loc a
go_up (Loc (t, c)) = case c of
  Empty → error "go up of empty"
  Cons le up ri →
    Loc (Node (reverse le ++ t:ri), up)
```

- Fokus nach unten

```
go_down :: Loc a→ Loc a
go_down (Loc (t, c)) = case t of
  Leaf _ → error "go down at leaf"
  Node [] → error "go down at empty"
  Node (t:ts) → Loc (t, Cons [] c ts)
```

Zipping Trees: Navigation

- Hilfsfunktion:

```
top :: Tree a → Loc a
top t = (Loc (t, Empty))
```

- Damit andere Navigationsfunktionen:

```
path :: Loc a → [Int] → Loc a
path l [] = l
path l (i:ps)
  | i ≡ 0 = path (go_down l) ps
  | i > 0 = path (go_left l) (i-1) ps
```

Einfügen

- Einfügen: Wo?
- Links des Fokus einfügen

```
insert_left :: Tree a→ Loc a→ Loc a
insert_left t1 (Loc (t, c)) = case c of
  Empty → error "insert_left: insert at empty"
  Cons le up ri → Loc(t, Cons (t1:le) up ri)
```

- Rechts des Fokus einfügen

```
insert_right :: Tree a→ Loc a→ Loc a
insert_right t1 (Loc (t, c)) = case c of
  Empty → error "insert_right: insert at empty"
  Cons le up ri → Loc(t, Cons le up (t1:ri))
```

- Unterhalb des Fokus einfügen

```
insert_down :: Tree a→ Loc a→ Loc a
insert_down t1 (Loc(t, c)) = case t of
  Leaf _ → error "insert_down: insert at leaf"
  Node ts → Loc(t1, Cons [] c ts)
```

Ersetzen und Löschen

- Unterbaum im Fokus ersetzen:

```
update :: Tree a→ Loc a→ Loc a
update t (Loc (_, c)) = Loc (t, c)
```

- Unterbaum im Fokus löschen: wo ist der neue Fokus?
  1. Rechter Baum, wenn vorhanden
  2. Linker Baum, wenn vorhanden
  3. Elternknoten

```
delete :: Loc a→ Loc a
delete (Loc(_, p)) = case p of
  Empty → Loc(Node [], Empty)
  Cons le up (r:ri) → Loc(r, Cons le up ri)
  Cons (l:le) up [] → Loc(l, Cons le up [])
  Cons [] up [] → Loc(Node [], up)
```

  - *"We note that delete is not such a simple operation."*

Schnelligkeit

- Wie schnell sind Operationen?

Schnelligkeit

- Wie schnell sind Operationen?
  - Aufwand: go_left $O(\mathit{left}(n))$, alle anderen $O(1)$.
- Warum sind Operationen so schnell?

Schnelligkeit

- Wie schnell sind Operationen?
  - Aufwand: go_left $O(\mathit{left}(n))$, alle anderen $O(1)$.
- Warum sind Operationen so schnell?
  - Kontext bleibt erhalten
  - Manipulation: reine Zeiger-Manipulation

Der Zipper als Monade

- LocM ist Zustandsmonade mit Zustand Loc

```
newtype LocM a b = LocM {locSt :: State (Loc a) b}
  deriving (Functor, Monad, MonadState (Loc a))
```

- Startfunktion

```
run :: LocM a b→ Tree a→ (b, Loc a)
run l t = runState (locSt l) (top t)
```

- Zugriff auf Baum im Fokus

```
current :: LocM a (Tree a)
current = do Loc(l, c)← get; return l
```

- Navigation

```
left :: LocM a ()
left = modify go_left
-- etc.
```

Zipper Monad: Manipulation

- Löschen:

```
del :: LocM a ()
del = modify delete
```

- Update:

```
upd :: Tree a → LocM a ()
upd t = modify (update t)
```

- Einfügen (bsp rechts):

```
insr :: Tree a → LocM a ()
insr t = modify (insert_right t)
```

- Hilfsprädikat: right möglich?

```
has_right :: Loc a → Bool
has_right (Loc(l, c)) = case c of
  (Cons (_:_) _ _) → True; _ → False
```

```
has_ri :: LocM a Bool
has_ri = gets has_right
```

Zipper für andere Datenstrukturen

- Binäre Bäume:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

- Kontext:

```
data Ctxt a = Empty
            | Le (Ctxt a) (Tree a)
            | Ri (Tree a) (Ctxt a)
```

```
newtype Loc a = Loc (Tree a, Ctxt a)
```

Tree-Zipper: Navigation

- Fokus nach links

```
go_left :: Loc a → Loc a
go_left (Loc(t, ctx)) = case ctx of
 Empty  → error "go_left at empty"
 Le c r → error "go_left of left"
 Ri l c → Loc(l, Le c t)
```

- Fokus nach rechts

```
go_right :: Loc a → Loc a
go_right (Loc(t, ctx)) = case ctx of
 Empty  → error "go_right at empty"
 Le c r → Loc(r, Ri t c)
 Ri _ _ → error "go_right of right"
```

Tree-Zipper: Navigation

- Fokus nach oben

```
go_up :: Loc a→ Loc a
go_up (Loc(t, ctx)) = case ctx of
  Empty   → error "go_up␣of␣empty"
  Le c r  → Loc(Node t r, c)
  Ri l c  → Loc(Node l t, c)
```

- Fokus nach unten links

```
go_down_left :: Loc a→ Loc a
go_down_left (Loc(t, c)) = case t of
  Leaf _     → error "go_down␣at␣leaf"
  Node l r   → Loc(l, Le c r)
```

- Fokus nach unten rechts

```
go_down_right :: Loc a→ Loc a
go_down_right (Loc(t, c)) = case t of
  Leaf _     → error "go_down␣at␣leaf"
  Node l r   → Loc(r, Ri l c)
```

Tree-Zipper: Einfügen und Löschen

- Einfügen links

```
ins_left :: Tree a→ Loc a→ Loc a
ins_left t1 (Loc(t, ctx)) = Loc(t, Ri t1 ctx)
```

- Einfügen rechts

```
ins_right :: Tree a→ Loc a→ Loc a
ins_right t1 (Loc(t, ctx)) = Loc(t, Le ctx t1)
```

- Löschen

```
delete :: Loc a→ Loc a
delete (Loc(_, c)) = case c of
  Empty  → error "delete of empty"
  Le c r → Loc(r, c)
  Ri l c → Loc(l, c)
```

  - Neuer Fokus: anderer Teilbaum

Tree-Zipper: Variation

- Binäre Bäume, Werte im Knoten:

```
data Tree a = Nil | Node (Tree a) a (Tree a)
```

- Kontext enthält Knotenwert

```
data Ctxt a  = Empty
             | Le (Ctxt a) a (Tree a)
             | Ri (Tree a) a (Ctxt a)
```

- Funktionen ähnlich, aber:
  - delete total, löscht Knoteninhalt im Kontext

Zipping Lists

- Listen:

```
data List a = Nil | Cons a (List a)
```

- Damit:

```
data Ctxt a = Empty | Snoc (Ctxt a) a
```

- Listen sind ihr 'eigener Kontext' :

$$\mathsf{List}\ a \cong \mathsf{Ctxt}\ a$$

Zipping Lists: Fast Reverse

- Listenumkehr schnell:

```
fastrev :: [a]→ [a]
fastrev xs = rev xs []

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

- Zweites Argument von rev: Kontext
  - Liste der Elemente davor in umgekehrter Reihenfolge

Zusammenfassung

- Der Zipper
  - Manipulation von Datenstrukturen
  - Zipper = Kontext + Fokus
  - Effiziente destruktive Manipulation
- Nachteile
  - Nicht richtig generisch — Schema, keine Bücherei
  - Viel schematischer Code für jeden Datentyp
  - Abhilfe: Generic Zipper
- Nächstes Mal: Graphen, Ströme, unendliche Datenstrukturen