

Reaktive Programmierung  
Vorlesung 12 vom 08.07.14: Bidirektionale Programmierung:  
Zippers and Lenses

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2014

# Fahrplan

- ▶ Teil I: Grundlegende Konzepte
- ▶ Teil II: Nebenläufigkeit
- ▶ Teil III: Fortgeschrittene Konzepte
  - ▶ Bidirektionale Programmierung: Zippers and Lenses
  - ▶ Robustheit, Fehlertoleranz und Fehlerbehandlung
  - ▶ Theorie der Nebenläufigkeit

# Was gibt es heute?

- ▶ Motivation: funktionale Updates
  - ▶ Akka ist *stateful*, aber im allgemeinen ist funktional besser
  - ▶ Globalen Zustand vermeiden hilft der Skalierbarkeit und der Robustheit
- ▶ Der Zipper
  - ▶ Manipulation innerhalb einer Datenstruktur
- ▶ Linsen
  - ▶ Bidirektionale Programmierung

# Ein einfacher Editor

- ▶ Datenstrukturen:

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

- ▶ Operationen: Cursor **bewegen** (links)

```
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)

```
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  $x + y$

```
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?

# Ziping Trees: Navigation

- ▶ Fokus nach **links**

`go_left :: Loc a → Loc a`

`go_left (Loc(t, c)) = case c of`

`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`

`Cons le up (r:ri) → Loc(r, Cons (t:le) up ri)`

`Cons _ _ [] → error "go_right_of_last"`



# Ziping 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)
```

# Ziping 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`  $O(\text{left}(n))$ , alle anderen  $O(1)$ .
- ▶ **Warum** sind Operationen so schnell?

# Schnelligkeit

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

# 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

# Zippping Lists

- ▶ Listen:

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

- ▶ Damit:

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

- ▶ Listen sind ihr 'eigener Kontext' :

List a  $\cong$  Ctxt a

# Ziping Lists: Fast Reverse

- ▶ Listenumkehr **schnell**:

$\text{fastrev} :: [a] \rightarrow [a]$

$\text{fastrev } xs = \text{rev } xs []$

$\text{rev} :: [a] \rightarrow [a] \rightarrow [a]$

$\text{rev } [] \quad \text{as} = \text{as}$

$\text{rev } (x:xs) \text{ as} = \text{rev } xs (x:\text{as})$

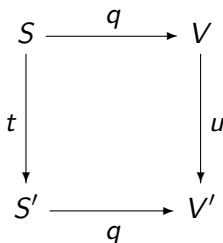
- ▶ Zweites Argument von `rev`: **Kontext**

- ▶ Liste der Elemente davor in **umgekehrter** Reihenfolge

# Bidirektionale Programmierung

- ▶ Motivierendes Beispiel: Update in einer Datenbank
- ▶ Weitere Anwendungsfelder:
  - ▶ Software Engineering (round-trip)
  - ▶ Benutzerschnittstellen (MVC)
  - ▶ Datensynchronisation

# View Updates



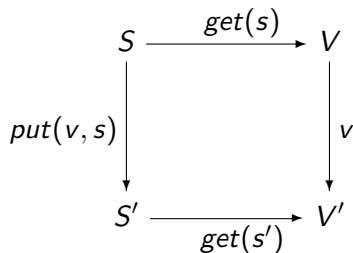
- ▶ View  $v$  durch Anfrage  $q$  (Bsp: Anfrage auf Datenbank)
- ▶ View wird **verändert** (Update  $u$ )
- ▶ Quelle  $S$  soll entsprechend angepasst werden (**Propagation** der Änderung)
- ▶ Problem:  $q$  soll **beliebig** sein
  - ▶ Nicht-injektiv? Nicht-surjektiv?

# Lösung

- ▶ Eine Operation *get* für den View
- ▶ Inverse Operation *put* wird automatisch erzeugt (wo möglich)
- ▶ Beide müssen invers sein — deshalb **bidirektionale Programmierung**



# Putting and Getting



- ▶ Signatur der Operationen:

$$\text{get} : S \longrightarrow V$$

$$\text{put} : V \times S \longrightarrow S$$

- ▶ Es müssen die **Linsengesetze** gelten:

$$\text{get}(\text{put}(v, s)) = v$$

$$\text{put}(\text{get}(s), s) = s$$

$$\text{put}(v, \text{put}(w, s)) = \text{put}(v, s)$$

## Erweiterung: Erzeugung

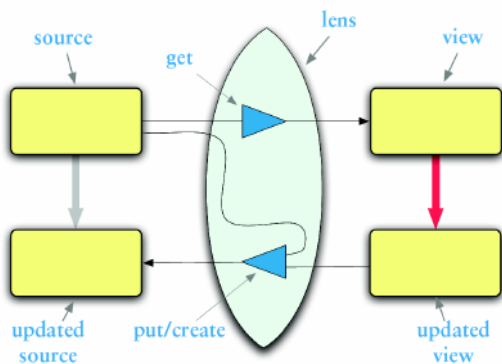
- ▶ Wir wollen auch Elemente (im Ziel) erzeugen können.
- ▶ Signatur:

$$\text{create} : V \longrightarrow S$$

- ▶ Weitere **Gesetze**:

$$\begin{aligned} \text{get}(\text{create}(v)) &= v \\ \text{put}(v, \text{create}(w)) &= \text{create}(w) \end{aligned}$$

# Die Linse im Überblick



## Linsen im Beispiel

- ▶ Updates auf strukturierten Datenstrukturen:

```
case class Turtle(  
  position: Point = Point(),  
  color: Color = Color(),  
  heading: Double = 0.0,  
  penDown: Boolean = false)
```

```
case class Point(  
  x: Double = 0.0,  
  y: Double = 0.0)
```

```
case class Color(  
  r: Int = 0,  
  g: Int = 0,  
  b: Int = 0)
```

- ▶ Ohne Linsen: functional record update

```
scala> val t = new Turtle();  
t: Turtle = Turtle(Point(0.0,0.0),Color(0,0,0),0.0,false)
```

```
scala> t.copy(penDown = ! t.penDown);  
res5: Turtle = Turtle(Point(0.0,0.0),Color(0,0,0),0.0,true)
```

## Linsen im Beispiel

- ▶ Das wird sehr schnell sehr aufwändig:

```
scala> def forward(t: Turtle) : Turtle =  
    t.copy(position= t.position.copy(x= t.position.x+ 1));
```

```
forward: (t: Turtle) Turtle
```

```
scala> forward(t);
```

```
res6: Turtle = Turtle(Point(1.0,0.0), Color(0,0,0), 0.0, false)
```

- ▶ Linsen helfen, das besser zu organisieren.

# Abhilfe mit Linsen

- ▶ Zuerst einmal: die **Linse**.

```
object Lenses {  
  case class Lens[O, V](  
    get: O ⇒ V,  
    set: (O, V) ⇒ O  
  ) }  
}
```

- ▶ Linsen für die Schildkröte:

```
val TurtlePosition =  
  Lens[Turtle, Point](_.position,  
    (t, p) ⇒ t.copy(position = p))
```

```
val PointX =  
  Lens[Point, Double](_.x,  
    (p, x) ⇒ p.copy(x = x))
```

# Benutzung

- ▶ Längliche Definition, aber einfache Benutzung:

```
scala> StandaloneTurtleLenses.TurtleX.get(t);  
res12: Double = 0.0
```

```
scala> StandaloneTurtleLenses.TurtleX.set(t, 4.3);  
res13: Turtles.Turtle = Turtle(Point(4.3,0.0),Color(0,0,0),0.0,false)
```

- ▶ Viel *boilerplate*, aber:
- ▶ Definition kann **abgeleitet** werden

# Abgeleitete Linsen

- ▶ Aus der Shapeless-Bücherei:

```
object ShapelessTurtleLenses {  
  
  import Turtles._  
  import shapeless._, Lens._, Nat._  
  
  val TurtleX = Lens[Turtle] >> _0 >> _0  
  val TurtleHeading = Lens[Turtle] >> _2  
  
  def right(t: Turtle,  $\delta$ : Double) =  
    TurtleHeading.modify(t)(_ +  $\delta$ )  
}
```

- ▶ Neue Linsen aus vorhandenen konstruieren



# Linse konstruieren

- ▶ Die **konstante** Linse (für  $c \in V$ ):

$$\begin{aligned} \text{const } c & : S \longleftrightarrow V \\ \text{get}(s) & = c \\ \text{put}(v, s) & = s \\ \text{create}(v) & = s \end{aligned}$$

- ▶ Die **Identitätslinse**:

$$\begin{aligned} \text{copy } c & : S \longleftrightarrow S \\ \text{get}(s) & = s \\ \text{put}(v, s) & = v \\ \text{create}(v) & = v \end{aligned}$$

# Linsen komponieren

- ▶ Gegeben Linsen  $L_1 : S_1 \longleftrightarrow S_2, L_2 : S_2 \longleftrightarrow S_3$
- ▶ Die Komposition ist definiert als:

$$\begin{aligned}L_2 \cdot L_1 & : S_1 \longleftrightarrow S_3 \\ \text{get} & = \text{get}_2 \cdot \text{get}_1 \\ \text{put}(v, s) & = \text{put}_1(\text{put}_2(v, \text{get}_1(s)), s) \\ \text{create} & = \text{create}_1 \cdot \text{create}_2\end{aligned}$$

# Mehr Linsen und Bidirektionale Programmierung

- ▶ Die Shapeless-Bücherei in Scala
- ▶ Linsen in Haskell
- ▶ DSL für bidirektionale Programmierung: Boomerang

# Zusammenfassung

- ▶ Der **Zipper**
  - ▶ Manipulation von Datenstrukturen
  - ▶ Zipper = Kontext + Fokus
  - ▶ Effiziente destruktive Manipulation
- ▶ **Bidirektionale Programmierung**
  - ▶ Linsen als Paradigma: *get*, *put*, *create*
  - ▶ Effektives funktionales Update
  - ▶ In Scala/Haskell mit abgeleiteter Implementierung, sonst als DSL.
- ▶ Nächstes Mal: Robustheit und Fehlerbehandlung
- ▶ Die Vorlesung und Übung in der nächsten Woche **fallen aus!**