

Reaktive Programmierung  
Vorlesung 9 vom 22.05.19  
Bidirektionale Programmierung — Zippers and Lenses

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

- ▶ Einführung
- ▶ Monaden und Monadentransformer
- ▶ Nebenläufigkeit: Futures and Promises
- ▶ Aktoren I: Grundlagen
- ▶ Aktoren II: Implementation
- ▶ Meta-Programmierung
- ▶ Bidirektionale Programmierung
- ▶ Reaktive Ströme I
- ▶ Reaktive Ströme II
- ▶ Funktional-Reaktive Programmierung
- ▶ Software Transactional Memory
- ▶ Eventual Consistency
- ▶ Robustheit und Entwurfsmuster
- ▶ Theorie der Nebenläufigkeit, Abschluss

# 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 Pos    = Int
data Editor = Ed { text    :: String
                  , cursor  :: Pos }
```

- ▶ Cursor bewegen (links)

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

- ▶ Text rechts einfügen:

```
insert :: Editor → Char → Editor
insert Ed{text= t, cursor= c} text =
  let (as, bs) = splitAt c t
  in  Ed{text= as ++ (text: bs), cursor= c+1}
```

# Aufwand

- ▶ **Aufwand** für Manipulation?  
 $O(n)$  mit  $n$  Länge des gesamten Textes
  
- ▶ Geht das auch einfacher?

# Ein einfacher Editor

- ▶ Datenstrukturen:

```
data Editor = Ed { before :: [Char] — In reverse order
                  , cursor :: Maybe Char
                  , after  :: [Char] }
```

- ▶ Invariante: cursor == Nothing gdw. before und after leer

- ▶ Cursor bewegen (links):

```
go_left :: Editor → Editor
go_left e@(Ed [] _) = e
go_left (Ed (a:as) (Just a) bs) = Ed as (Just a) (c: bs)
```

- ▶ Text unter dem Cursor löschen:

```
delete :: Editor → Editor
delete (Ed as _ (b:bs)) = Ed as (Just b) bs
delete (Ed (a:as) _ []) = Ed as (Just a) []
delete (Ed [] _ []) = Ed [] Nothing []
```

# Manipulation strukturierter Datentypen

- ▶ Anderer Datentyp:  $n$ -äre Bäume (rose trees)

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

- ▶ Bspw. abstrakte Syntax von einfachen Ausdrücken
- ▶ Update auf Beispielterm  $t = a * b - c * d$ : ersetze  $b$  durch  $x + y$

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

- ▶ Referenzierung durch Namen

```
upd1 :: Eq a => a -> Tree a -> Tree a -> Tree a
```

- ▶ Referenzierung durch Pfad: **type** Path=[Int]

```
type Path = [Int]  
upd2 :: Path -> Tree a -> Tree a -> Tree a
```

# Aufwand

- ▶ Aufwand: Mittlere Aufwand  $O(\log n)$ , worst case  $O(n)$   
 $n$  Anzahl der Knoten
- ▶ Geht das besser — wie beim einfachen Editor?
- ▶ Generalisierung der Idee



# Der Zipper

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

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

- ▶ Kontext ist 'inverse Umgebung' ("*Like a glove turned inside out*")
- ▶ Besteht aus linken Nachbarn, Knoten, Kontext darüber, rechtem Nachbarn
- ▶ Loc a ist **Baum** mit **Fokus**

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

# Ziping Trees: Navigation

## ► Fokus nach **links**

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

## ► Fokus nach **rechts**

```
go_right :: Loc a → Loc a
go_right (Loc(t, c)) = case c of
  Cons le a up (r:ri) → Loc(r, Cons (t:le) a up ri)
  _                    → error "go_right: at 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: at the top"
  Cons le a up ri →
    Loc (Node a (reverse le ++ t:ri), up)
```

## ► Fokus nach **unten**

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

# Einfügen

- ▶ **Einfügen**: Wo?
- ▶ **Überschreiben** des Fokus

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

- ▶ **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 a up ri → Loc(t, Cons (t1:le) a 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 a up ri → Loc(t, Cons le a up (t1:ri))
```

# Ersetzen und Löschen

► 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(_, c)) = case c of
  Empty → error "delete: delete at top"
  Cons le a up (r:ri) → Loc(r, Cons le a up ri)
  Cons (l:le) a up [] → Loc(l, Cons le a up [])
  Cons [] a up [] → Loc (Node a [], up)
```

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

# Schnelligkeit

- ▶ Wie **schnell** sind Operationen?

# Schnelligkeit

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

# Schnelligkeit

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



# Zipper für andere Datenstrukturen

## ► Binäre Bäume:

```
sealed trait Tree[+A]  
case class Leaf[A](value: A) extends Tree[A]  
case class Node[A](left: Tree[A],  
                    right: Tree[A]) extends Tree[A]
```

## ► Kontext:

```
sealed trait Context[+A]  
case object Empty extends Context[Nothing]  
case class Left[A](up: Context[A],  
                  right: Tree[A]) extends Context[A]  
case class Right[A](left: Tree[A],  
                   up: Context[A]) extends Context[A]
```

```
case class Loc[A](tree: Tree[A], context: Context[A])
```

# Tree-Zipper: Navigation

► Fokus nach **links**

```
def goLeft: Loc[A] = context match {  
  case Empty ⇒ sys.error("goLeft at empty")  
  case Left(_,_) ⇒ sys.error("goLeft of left")  
  case Right(l,c) ⇒ Loc(l, Left(c, tree))  
}
```

► Fokus nach **rechts**

```
def goRight: Loc[A] = context match {  
  case Empty ⇒ sys.error("goRight at empty")  
  case Left(c,r) ⇒ Loc(r, Right(tree, c))  
  case Right(_,_) ⇒ sys.error("goRight of right")  
}
```

# Tree-Zipper: Navigation

## ► Fokus nach **oben**

```
def goUp: Loc[A] = context match {  
  case Empty ⇒ sys.error("goUp of empty")  
  case Left(c, r) ⇒ Loc(Node(tree, r), c)  
  case Right(l, c) ⇒ Loc(Node(l, tree), c)  
}
```

## ► Fokus nach **unten links**

```
def goDownLeft: Loc[A] = tree match {  
  case Leaf(_) ⇒ sys.error("goDown at leaf")  
  case Node(l, r) ⇒ Loc(l, Left(context, r))  
}
```

## ► Fokus nach **unten rechts**

```
def goDownRight: Loc[A] = tree match {  
  case Leaf(_) ⇒ sys.error("goDown at leaf")  
  case Node(l, r) ⇒ Loc(r, Right(l, context))  
}
```

# Tree-Zipper: Einfügen und Löschen

## ▶ Einfügen links

```
def insertLeft(t: Tree[A]): Loc[A] =  
  Loc(tree, Right(t, context))
```

## ▶ Einfügen rechts

```
def insertRight(t: Tree[A]): Loc[A] =  
  Loc(tree, Left(context, t))
```

## ▶ Löschen

```
def delete: Loc[A] = context match {  
  case Empty => sys.error("delete of empty")  
  case Left(c, r) => Loc(r, c)  
  case Right(l, c) => Loc(l, c)  
}
```

## ▶ Neuer Fokus: anderer Teilbaum

# Ziping Lists

- ▶ Listen:

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

- ▶ Damit:

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

- ▶ Listen sind ihr 'eigener Kontext' :

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

# Ziping Lists: Fast Reverse

- ▶ Listenumkehr **schnell**:

```
fastrev1 :: List a → List a
fastrev1 xs = rev (top xs) where
  rev :: Loc a → List a
  rev (Loc (Nil, as))      = as
  rev (Loc (Cons x xs, as)) = rev (Loc (xs, Cons x as))
```

- ▶ Vergleiche:

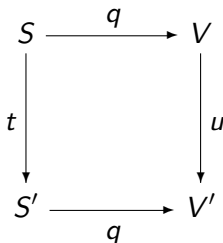
```
fastrev2 :: [a] → [a]
fastrev2 xs = rev xs [] where
  rev :: [a] → [a] → [a]
  rev []      as = as
  rev (x:xs) as = rev xs (x:as)
```

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

# Bidirektionale Programmierung

- ▶ Verallgemeinerung der Idee des Kontext
- ▶ Motivierendes Beispiel: Update in einer Datenbank
- ▶ Weitere Anwendungsfelder:
  - ▶ Benutzerschnittstellen (MVC)
  - ▶ Datensynchronisation

# View Updates



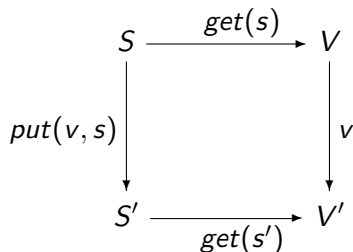
- ▶ View  $v$  durch Anfrage  $q$  (Bsp: Anfrage auf Datenbank)
- ▶ View wird **verändert** (Update  $u$ )
- ▶ Quelle  $S$  soll entsprechend angepasst werden (**Propagation** der Änderung)
- ▶ Problem:  $q$  soll **beliebig** sein
  - ▶ Nicht-injektiv? Nicht-surjektiv?



# 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:

$$get : S \longrightarrow V$$

$$put : V \times S \longrightarrow S$$

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

$$get(put(v, s)) = v$$

$$put(get(s), s) = s$$

$$put(v, put(w, s)) = put(v, s)$$

## Erweiterung: Erzeugung

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

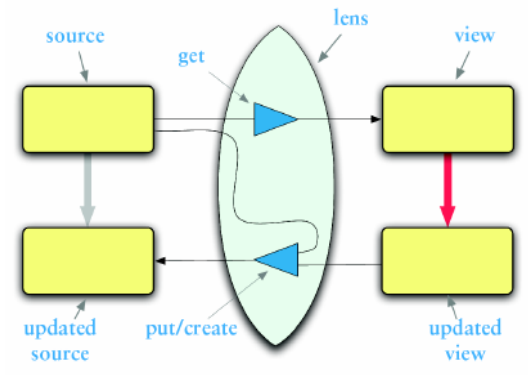
- ▶ Signatur:

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

- ▶ 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:

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

- ▶ Beispiel hier:

$$\text{TurtleX} = \text{TurtlePosition} \cdot \text{PointX}$$

# 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: Reaktive Ströme