Reaktive Programmierung

Vorlesung 8 vom 10.05.17: Bidirektionale Programmierung — Zippers and Lenses

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2017

Fahrplan

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 :: Editor→ String→ Editor
insert 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{col= col c+1}}
```

Mit Hilfsfunktion:

```
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
```

- Aufwand für Manipulation?
  $O(n)$ mit n Länge des gesamten Textes

Manipulation strukturierter Datentypen

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

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

- Bspw. 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"]
         ]
```

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

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)
```


Zipping 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))
 _                 → 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)
 _                 → 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

- Konstruktor (für):

```
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 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))
```


Einfügen

- 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?
  - Aufwand: go_up $O(left(n))$, 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

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' :

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

Zipping 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:

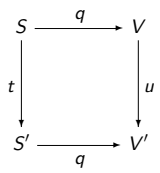
```
fastrev2 :: [a]→ [a]
fastrev2 xs = rev xs [] where
  rev :: [a]→ [a]→ [a]
  rev []     as = as
  rev (x:xs) as = rev xs (x:as)
```

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

Bidirektionale Programmierung

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

View Updates

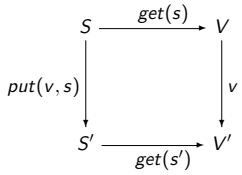


- View v durch Anfrage q (Bsp: Anfrage auf Datenbank)
- View wird verändert (Update u)
- Quelle S soll entsprechend angepasst werden (Propagation der Änderung)
- Problem: q soll beliebig sein
  - Nicht-injektiv? Nicht-surjektiv?

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:

$$\begin{aligned} get &: S \longrightarrow V \\ put &: V \times S \longrightarrow S \end{aligned}$$

- Es müssen die Linsengesetze gelten:

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

Erweiterung: Erzeugung

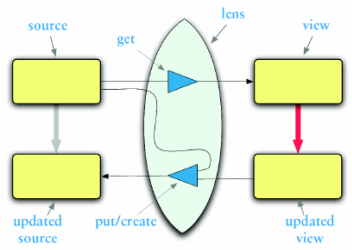
- Wir wollen auch Elemente (im Ziel) erzeugen können.
- Signatur:

$$create : V \longrightarrow S$$

- Weitere Gesetze:

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

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, δ: Double) =
    TurtleHeading.modify(t)(_ + δ)
```

- Neue Linsen aus vorhandenen konstruieren

Linsen konstruieren

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

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

- Die Identitätslinse:

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

Linsen komponieren

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

$$\begin{array}{rcl} L_2 \cdot L_1 & : & S_1 \longleftrightarrow S_3 \\ get & = & get_2 \cdot get_1 \\ put(v, s) & = & put_1(put_2(v, get_1(s)), s) \\ create & = & create_1 \cdot create_2 \end{array}$$

- Beispiel hier:

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

Mehr Linsen und Bidirektionale Progammierung

- 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: Metaprogrammierung — Programme schreiben Programme