Reaktive Programmierung
Vorlesung 13 vom 23.06.14: Bidirektionale Programmierung: Zippers and Lenses

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte
  - Bidirektionale Programmierung: Zippers and Lenses
  - Eventual Consistency
  - Robustheit, Entwurfsmuster
  - 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  = List[String]
case class Pos(line: Int, col: Int)
case class Editor(text: Text, cursor: Pos)
```

- Operationen: Cursor bewegen (links)

```
def goLeft: Editor =
  if (cursor.col == 0) sys.error("At start of line")
  else Editor(text, cursor.copy(col = cursor.col - 1))
```

Beispieloperationen

- Text rechts einfügen:

```
def insertRight(s: String): Editor = {
  val (befor,after) =
      text(cursor.line).splitAt(cursor.col)
  val newLine = before + s + after
  val newText = text.take(cursor.line) ++
    (newLine :: text.drop(cursor.line + 1))
  Editor(newText,cursor)
}
```

- Problem: Aufwand für Manipulation

Manipulation strukturierter Datentypen

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

```
sealed trait Tree[A]
case class Leaf[A](a: A) extends Tree[A]
case class Node[A](children: Tree[A]*) extends Tree[A]
```

  - Bsp: Abstrakte Syntax von einfachen Ausdrücken

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

```
val t = Node(Leaf("-"),
  Node(Leaf("*"), Leaf("a"), Leaf ("b")),
  Node(Leaf("*"), Leaf("c"), Leaf ("d"))
)
```

Der Zipper

- Idee: Kontext nicht wegwerfen!
- Nicht: `case class Path(i: Int*)`
- Sondern:

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

  - Kontext ist 'inverse Umgebung' (*"Like a glove turned inside out"*)
- `Location[A]` ist Baum mit Fokus

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

Zipping Trees: Navigation

- Fokus nach links

```
def goLeft: Location[A] = context match {
  case Cons(l::le,up,ri) ⇒
    Location(l, Cons(le,up,(t::ri)))
  case _ ⇒ sys.error("goLeft of first")
}
```

- Fokus nach rechts

```
def goRight: Location[A] = context match {
  case Cons(le,up,r::ri) ⇒
    Location(r,Cons(t::le,up,ri))
  case _ ⇒ sys.error("goRight of last")
}
```

Zipping Trees: Navigation

- Fokus nach oben

```
def goUp: Location[A] = context match {
  case Empty ⇒ sys.error("goUp of empty")
  case Cons(le,up,ri) ⇒
    Location(Node((le.reverse ++ t::ri) :_*), up)
}
```

- Fokus nach unten

```
def goDown: Location[A] = tree match {
  case Leaf(_) ⇒ sys.error("goDown at leaf")
  case Node() ⇒ sys.error("goDown at empty")
  case Node(t,ts@ _*) ⇒
    Loaction(t,Cons(Seq.empty,context,ts))
}
```

Zipping Trees: Navigation

- Hilfsfunktion (auf `Tree[A]`):

```
def top: Location[A] =
  Location(this,Empty)
```

- Damit andere Navigationsfunktionen:

```
def path(ps: List[Int]): Loaction[A] = ps match {
  case Nil  ⇒ this
  case i::ps if i == 0 ⇒ goDown.path(ps)
  case i::ps if i > 0 ⇒ goLeft.path((i-1)::ps)
}
```

Einfügen

- Einfügen: Wo?
- Links des Fokus einfügen

```
def insertLeft(t: Tree[A]): Loaction[A] = context match {
  case Empty ⇒ sys.error("insertLeft at empty")
  case Cons(le,up,ri) ⇒ Location(tree,Cons(t::le,up,ri))
}
```

- Rechts des Fokus einfügen

```
def insertRight(t: Tree[A]): Location[A] = context match {
  case Empty ⇒ sys.error("insertRight at empty")
  case Cons(le,up,ri) ⇒ Location(tree,Cons(le,up,t::ri))
}
```

Einfügen

- Unterhalb des Fokus einfügen

```
def insertDown(t: Tree[A]): Location[A] = tree match {
  case Leaf(_) ⇒ sys.error("insertDown at leaf")
  case Node(ts @_*) ⇒ Location(t,Cons(Nil,context,ts))
}
```

Ersetzen und Löschen

- Unterbaum im Fokus ersetzen:

```
def update(t: Tree): Location[A] =
  Location(t,context)
```

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

```
def delete: Location[A] = context match {
  case Empty ⇒ Location(Node(),Empty)
  case Cons(le,up,r::ri) ⇒ Location(r,Cons(le,up,ri))
  case Cons(l:le,up,Nil) ⇒ Location(l,Cons(le,up,Nil))
  case Cons(Nil,up,Nil) ⇒ Location(Node(),up)
}
```

- *“We note that `delete` is not such a simple operation.”*

Schnelligkeit

- Wie schnell sind Operationen?

Schnelligkeit

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

Schnelligkeit

- Wie schnell sind Operationen?
  - Aufwand: `goLeft` $O(\mathit{left}(n))$, 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(value: A) extends Tree[A]
case class Node(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 Location[A](tree: Tree[A], context:
   Context[A])
```

Tree-Zipper: Navigation

- Fokus nach links

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

- Fokus nach rechts

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

- Fokus nach unten links

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

- Fokus nach unten rechts

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

Tree-Zipper: Einfügen und Löschen

- Einfügen links

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

- Einfügen rechts

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

- Löschen

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

  - Neuer Fokus: anderer Teilbaum

Zipping Lists

- Listen:

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class ::[A](head: A, tail: List[A])
  extends List[A]
```

- Damit:

```
sealed trait Context[+A]
case object Empty extends Context[Nothing]
case class Snoc[A](init: Context[A], last: A)
  extends Context[A]
```

- Listen sind ihr 'eigener Kontext' :

$$\text{List}[A] \cong \text{Context}[A]$$

Zipping Lists: Fast Reverse

- Listenumkehr schnell:

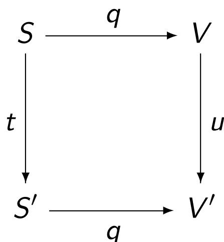
```
def reverse(init: List[A] = Nil) = this match {
  case Nil  ⇒ init
  case x::xs ⇒ xs.reverse(x::init)
}
```

- Argument von `reverse`: 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

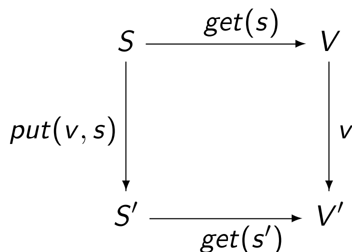


- View v durch Anfrage q (Bsp: Anfrage auf Datenbank)
- View wird verändert (Update u)
- Quelle S soll entsprechend angepasst werden (Propagation der Änderung)
- Problem: q soll beliebig sein
  - Nicht-injektiv? Nicht-surjektiv?

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

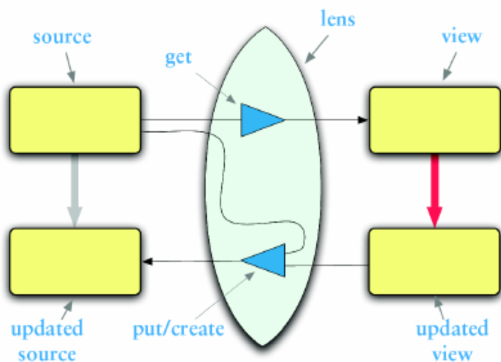
- Wir wollen auch Elemente (im Ziel) erzeugen können.
- Signatur:

$$create : V \longrightarrow S$$

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

- Neue Linsen aus vorhandenen konstruieren

Linsen konstruieren

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

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

- Die Identitätslinse:

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

Linsen komponieren

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

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

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: Eventual Consistency