# Reaktive Programmierung
## Vorlesung 3 vom 21.04.15: Funktional-Reaktive Programmierung

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015



## Fahrplan

- Teil I: Grundlegende Konzepte
  - Was ist Reaktive Programmierung?
  - Nebenläufigkeit und Monaden in Haskell
  - Funktional-Reaktive Programmierung
  - Einführung in Scala
  - Die Scala Collections
  - ScalaTest und ScalaCheck
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte



## Das Tagemenü

- Funktional-Reaktive Programmierung (FRP) ist rein funktionale, reaktive Programmierung.
- Sehr abstraktes Konzept — im Gegensatz zu Observables und Aktoren.
- Literatur: Paul Hudak, The Haskell School of Expression, Cambridge University Press 2000, Kapitel 13, 15, 17.
  - Andere (effizientere) Implementierung existieren.



## FRP in a Nutshell

- Zwei Basiskonzepte
- Kontinuierliches, über der Zeit veränderliches Verhalten:

```
type Time = Float
type Behaviour a = Time → a
```

- Diskrete Ereignisse zu einem bestimmten Zeitpunkt:

```
type Event a = [(Time, a)]
```

- Obige Typdefinitionen sind Spezifikation, nicht Implementation



## Verhalten: erste einfache Beispiele

- Ein kreisender und ein pulsierender Ball:

```
circ, pulse :: Behavior Region
circ    = translate (cos time, sin time) (ell 0.2 0.2)
pulse   = ell (cos time ∗ 0.5) (cos time ∗ 0.5)
```

- Was passiert hier?
  - Basisverhalten: time :: Behaviour Time, constB :: a → Behavior a
  - Grafikbücherei: Datentyp Region, Funktion Ellipse
  - Liftings (∗, 0.5, sin, ...)



## Reaktive Animationen: Verhaltensänderung

- Beispiel: auf Knopfdruck Farbe ändern:

```
color1 :: Behavior Color
color1 = red `untilB` lbp −≫ blue
```

```
color2r = red `untilB` ce where
          ce = (lbp −≫ blue `untilB` ce) .|.
               (key −≫ yellow `untilB` ce)
```

- Was passiert hier?
  - untilB kombiniert Verhalten:

```
untilB :: Behavior a → Event (Behavior a) → Behavior a
```

  - =≫ ist map für Ereignisse:

```
(=≫) :: Event a → (a→b) → Event b
(−≫) :: Event a→ b→ Event b
e −≫ v = e =≫ λ_ → v
```

  - Kombination von Ereignissen:



## Der Springende Ball

```
ball2 = paint red (translate (x,y) (ell 0.2 0.2))
    where g = −4
          x = −3 + integral 0.5
          y = 1.5 + integral v
          v = integral g `switch`
                (hit `snapshot_` v =≫ λv'→
                 lift0 (−v') + integral g)
          hit = when (y <∗ −1.5)
```

- Nützliche Funktionen:

```
integral :: Behavior Float → Behavior Float
```

```
snapshot :: Event a → Behavior b → Event (a,b)
```

- Erweiterung: Ball ändert Richtung, wenn er gegen die Wand prallt.



## Implementation

- Verhalten, erste Annäherung:

```
data Beh1 a = Beh1 ([(UserAction, Time)]→ Time→ a)
```

- Problem: Speicherleck und Ineffizienz
- Analogie: suche in sortierten Listen

```
inList :: [Int] → Int → Bool
inList xs y = elem y xs

manyInList' :: [Int]→ [Int]→ [Bool]
manyInList' xs ys = map (inList xs) ys
```

- Besser Sortiertheit direkt nutzen

```
manyInList :: [Int] → [Int] → [Bool]
```

## Implementation

- Verhalten werden inkrementell abgetastet:

```
data Beh2 a
    = Beh2 ([(UserAction, Time)] → [Time] → [a])
```

- Verbesserungen:
  - Zeit doppelt, nur einmal
  - Abtastung auch ohne Benutzeraktion
  - Currying

```
data Behavior a
    = Behavior (([Maybe UserAction], [Time]) → [a])
```

- Ereignisse sind im Prinzip optionales Verhalten:

```
data Event a = Event (Behaviour (Maybe a))
```



## Längeres Beispiel: Paddleball

- Das Paddel:

```
paddle = paint red (translate (fst mouse, −1.7) (rec 0.5 0.05))
```

- Der Ball:

```
pball vel =
 let xvel    = vel `stepAccum` xbounce −≫ negate
     xpos    = integral xvel
     xbounce = when (xpos >∗ 2 ||∗ xpos <∗ −2)
     yvel    = vel `stepAccum` ybounce −≫ negate
     ypos    = integral yvel
     ybounce = when (ypos >∗ 1.5
                ||∗ ypos       `between` (−2.0,−1.5) &&∗
                    fst mouse `between` (xpos−0.25,xpos+0.25))
 in paint yellow (translate (xpos, ypos) (ell 0.2 0.2))
```

- Die Mauern:

```
walls :: Behavior Picture
```

- ... und alles zusammen:

```
paddleball vel = walls `over` paddle `over` pball vel
```



## Warum nicht in Scala?

- Lifting und Typklassen für syntaktischen Zucker
- Aber: zentrales Konzept sind unendliche Listen (Ströme) mit nicht-strikte Auswertung
  - Implementation mit Scala-Listen nicht möglich
  - Benötigt: Ströme als unendliche Listen mit effizienter, nicht-strikter Auswertung
  - Möglich, aber nicht für diese Vorlesung
- Generelle Schwäche:
  - Fundamental nicht-kompositional — ist gibt eine Hauptfunktion
  - Fehlerbehandlung, Nebenläufigkeit?



## Zusammenfassung

- Funktional-Reaktive Programmierung am Beispiel FAL (Functional Animation Library)
- Zwei Kernkonzepte: kontinuierliches Verhalten und diskrete Ereignisse
- Implementiert in Haskell, Systemverhalten als unendlicher Strom von Zuständen
- Erlaubt abstrakte Progammierung von reaktiven Animationen
- Problem ist mangelnde Kompositionalität
- Nächste Vorlesungen: Scala!