

Reaktive Programmierung
Vorlesung 9 vom 14.06.2022
Funktional-Reaktive Programmierung

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

Fahrplan

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

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.

Dialektik des Programmierens

	Einer	Viele
Synchron	Try [T]	Iterable [T]

► **These:** Synchron

Dialektik des Programmierens

	Einer	Viele
Synchron	Try [T]	Iterable [T]
Asynchron	Future [T]	Observable [T]

▶ **These:** Synchron

▶ **Antithese:** Asynchron

Dialektik des Programmierens

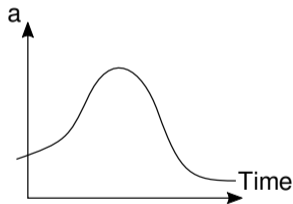
	Einer	Viele
Synchron	Try [T]	Iterable [T]
Asynchron	Future [T]	Observable [T]
Reaktiv	Event a	Behaviour a

- ▶ **These:** Synchron
- ▶ **Antithese:** Asynchron
- ▶ **Synthese:** Reaktiv

FRP in a Nutshell: Zwei Basiskonzepte

- ▶ **Kontinuierliches**, über der Zeit veränderliches **Verhalten**:

```
type Time = Float
type Behaviour  $\alpha$  = Time  $\rightarrow$   $\alpha$ 
```

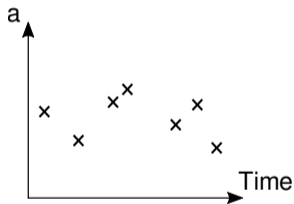


- ▶ Beispiel: Position eines Objektes

Obige Typdefinitionen sind **Spezifikation**, nicht **Implementation**

- ▶ **Diskrete Ereignisse** zu einem bestimmten Zeitpunkt:

```
type Event  $\alpha$  = [(Time,  $\alpha$ )]
```



- ▶ Beispiel: Benutzereingabe

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, ...)`

Lifting

- ▶ Um einfach mit `Behaviour` umgehen zu können, werden Funktionen zu `Behaviour` **geliftet**:

```
($*) :: Behavior (a→b) → Behavior a → Behavior b  
lift1 :: (a → b) → (Behavior a → Behavior b)
```

- ▶ Gleiches mit `lift2`, `lift3`, ...

- ▶ Damit komplexere Liftings (für viele andere Typklassen):

```
instance Num a => Num (Behavior a) where  
  (+) = lift2 (+)
```

```
instance Floating a => Floating (Behavior a) where  
  sin    = lift1 sin
```

Reaktive Animationen: Verhaltensänderung

- ▶ Beispiel: auf Knopfdruck Farbe ändern:

```
color1 :: Behavior Color
color1 = red 'untilB' lbp → blue
```

- ▶ Was passiert hier?

- ▶ `untilB` und `switch` kombinieren Verhalten

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

- ▶ `Event` ist ein `Functor`:

```
instance Functor Event where
e → v = fmap (const v) e
```

Reaktive Animationen: Verhaltensänderung

- ▶ Beispiel: auf Knopfdruck Farbe ändern:

```
color2h = red 'switch' ((lbp → blue) .|. (key → yellow))
```

- ▶ Was passiert hier?

- ▶ `untilB` und `switch` kombinieren Verhalten

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

- ▶ Event ist ein Functor:

```
instance Functor Event where  
e → v = fmap (const v) e
```

- ▶ Kombination von Ereignissen:

```
(.|.) :: Event a → Event a → Event a
```

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 vy
  vy = integral g 'switch'
      (hity 'snapshot_' vy ==>> λv'→ lift0 (-v') + integral g)
  hity = when (y <= -1.5)
```

► Nützliche Funktionen:

```
when :: Behavior Bool → Event ()
integral :: Behavior Float → Behavior Float
snapshot :: Event a → Behavior b → Event (a,b)
snapshot_ :: Event a → Behavior b → Event b
```

Der Springende Ball

```
ball2x = paint red (translate (x,y) (ell 0.2 0.2)) where
  g = -4
  x = -3 + integral vx
  vx = 1 'switch' (hitx 'snapshot_' vx ==>> λv' → lift0 (-v'))
  hitx = when (x < * -3 || * x > * 3)
  y = 1.5 + integral vy
  vy = integral g 'switch'
      (hity 'snapshot_' vy ==>> λv' → lift0 (-v')) + integral g)
  hity = when (y < * -1.5)
```

► Nützliche Funktionen:

```
when :: Behavior Bool → Event ()
integral :: Behavior Float → Behavior Float
snapshot :: Event a → Behavior b → Event (a,b)
snapshot_ :: Event a → Behavior b → Event 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: Pong!

- ▶ Pong besteht aus Paddel, Mauern und einem Ball.
- ▶ Das Paddel:

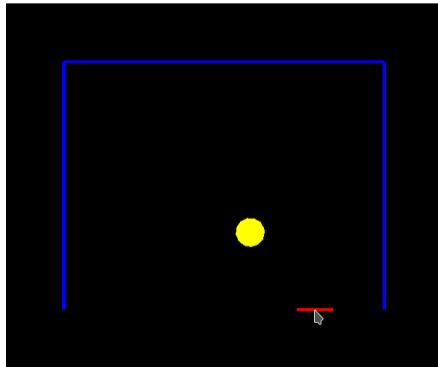
```
paddle = paint red (translate (fst mouse, -1.7) (rec 0.5 0.05))
```

- ▶ Die Mauern:

```
walls :: Behavior Picture
```

- ▶ ... und alles zusammen:

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



Pong: der Ball

► Der Ball:

```
pball vel =  
  let xvel = vel 'stepAccum' xbcnc → negate  
      xpos = integral xvel  
      xbcnc = when (xpos >* 2 || * xpos <* -2)  
      yvel = vel 'stepAccum' ybcnc → negate  
      ypos = integral yvel  
      ybcnc = 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))
```

► Ball völlig unabhängig von Paddel und Wänden

► Nützliche Funktionen:

```
while, when :: Behavior Bool → Event ()  
step :: a → Event a → Behavior a  
stepAccum :: a → Event (a→a) → Behavior a
```

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 aufwändig

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
- ▶ Stärke: Erlaubt **abstrakte** Programmierung von **reaktiven Animationen**
- ▶ Schwächen:
 - ▶ Fundamental **nicht-kompositional** — ist gibt **eine** Hauptfunktion
 - ▶ Debugging, Fehlerbehandlung, Nebenläufigkeit?
- ▶ Nächste Vorlesung: Software Transactional Memory (STM)