Reaktive Programmierung
Vorlesung 12 vom 12.06.19
Funktional-Reaktive Programmierung

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

Fahrplan

Das Tagemenü

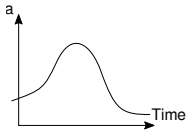
- **Funktional-Reaktive Programmierung** (FRP) ist **rein** funktionale, reaktive Programmierung.
- Sehr **abstraktes** Konzept — im Gegensatz zu Observables und Aktoren.
- Literatur: Paul Hudak, **The Haskell School of Expression**, Cambridge University Press 2000, Kapitel 13, 15, 17.
  - Andere (effizientere)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

FRP in a Nutshell

Zwei Basiskonzepte:

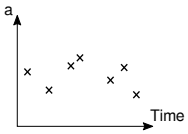
- **Kontinuierliches**, über der Zeit veränderliches **Verhalten**:

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



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

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



- Beispiel: Position eines Objektes
- Beispiel: Benutzereingabe

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

Verhalten: erste einfache Beispiele

- Ein kreisender und ein pulsierender Ball:

```
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 ff $∗ Behavior fb
lift1 f b1 = lift0 f $∗ b1
```

  - Gleiches mit `lift2`, `lift3`, ...
- Damit komplexere Liftings (für viele andere Typklassen):

```
(+) = lift2 (+)
(∗) = lift2 (∗)
```

```
pi  = lift0 pi
cos = lift1 cos
```


Reaktive Animationen: Verhaltensänderung

- Beispiel: auf Knopfdruck Farbe ändern:

```
color1 = red `untilB` lbp →⇒ blue
```

```
color3 = white `switch` (key ⇒⇒ λc →
```

- Was passiert hier?
  - `untilB` und `switch` kombinieren Verhalten:

```
Behavior fb `untilB` Event fe =
Behavior fb `switch` Event fe =
```

  - `⇒⇒` ist `map` für Ereignisse:

```
Event fe ⇒⇒ f = Event (map (fmap f) ∘ fe)
e →⇒ v = e ⇒⇒ λ_ → v
```

  - Kombination von Ereignissen:

```
Event fe1 .|. Event fe2
```


Der Springende Ball

```
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)
```

```
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:

```
integral = genIntegral 0 (+) (∗)
Event fe `snapshot` Behavior fb
  = Event (λuts → zipWith' aux (fe uts) (fb uts))
```

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))
```

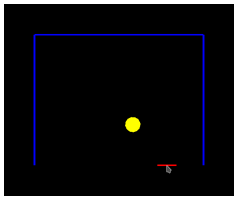

Längeres Beispiel: Pong!

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

```
walls :: Behavior Picture
```

- ... und alles zusammen:

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



Pong: der Ball

- Der Ball:

```
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))
```

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