

19. Funktional-reaktive Programmierung

Graphische Figuren

```
type Dimension = Int
data Figure = Rect Dimension Dimension
            | Triangle Dimension Angle Dimension
            | Polygon [Point]
            | Circle Dimension
            | Translate Point Figure
            | Scale Double Figure
            | Rotate Angle Figure
            deriving (Eq, Ord, Show)
```

Anfangsposition ist (0,0)

Beispiele

- `Rect 10 20` – Rechteck mit Höhe 10, Länge 20
- `Rotate (pi/2) (Triangle 10 (pi/3) 10)` –
Gleichschenkliges Dreieck, auf dem Kopf stehend.
- `Rotate pi (Circle 20)` – rotierter Kreis

Drehung um einen Punkt:

`rotate :: Point -> Angle -> Figure -> Figure`

`rotate (px, py) w =`

`Translate (px, py) . Rotate w . Translate (-px, -py)`

Graphic

Schnittstelle zur Grafikprogrammierung in Hugs

Operationen:

- `openWindow :: Title -> Size -> IO Window`
- `drawInWindow :: Window -> Graphic -> IO ()`
- `ellipse, line, text :: ... -> Graphic`
- `withColor, withFont :: ... -> Graphic -> Graphic`
- `overGraphics :: [Graphic] -> Graphic`
- `overGraphic :: Graphic -> Graphic -> Graphic`
- ...

Hier wichtig: `draw :: Figure -> Graphic`

Beispiel

```
drawFigs :: [Figure]-> IO ()
drawFigs f =
  runGraphics (do
    w<- openWindow "Beispiel" (500, 500)
    drawInWindow w (overGraphics (zipWith withColor
      (cycle [Blue ..])
      (map (draw. Translate (250, 250)) f)))
    getKey w
    closeWindow w)

swirl :: Figure-> [Figure]
swirl = iterate (Rotate (pi/17). Scale 1.123)

main = do
  drawFigs (take 300 (swirl
    (Triangle 6 (pi/3) 6)))
  drawFigs (take 300 (swirl (Rect 4 3)))
```

Animation

Idee: Animation ist eine Funktion der Zeit die Bilder liefert

- `type Animation a = Time -> a`
- `type Time = Float`

Beispiele: `Animation Graphic`, `Animation Figure`

Anzeigen von Animation

```
animate :: String-> Animation Graphic-> IO ()
```

```
animFig :: String-> Animation Figure -> IO ()
```

```
animFig t a = animate t (drawCentered. a)
```

```
drawCentered :: Figure -> Graphic
```

```
drawCentered = draw . Translate (250, 250)
```

Pulsierender Ball

```
pulsatingBall :: Animation Figure
pulsatingBall t = Circle (round (190* sin t))

animFig "pulsatingBall" pulsatingBall
```

Drehender Ball

```
revolvingBall :: Animation Figure
revolvingBall t = Translate (round (210* cos t),
                             round (210* sin t))
                             (Circle 20)

animFig "revolvingBall" revolvingBall
```

Planeten

```
planets :: Animation Graphic
planets t =
  draw (pulsatingBall t) 'overGraphic'
  draw (revolvingBall t)

animate "planets" planets
```

Modul `Fa1`: Functional-Animations-Library

Erweiterung von `Animation`

```
newtype Beh a = Beh (Time -> a)
```

Zeitabhängige Werte

Anzeigen von `Beh Graphic`:

```
reactimate :: String -> Beh Graphic -> IO ()
```

Anzeigen von `Beh Figure`:

```
reactimateF :: String -> Beh Figure -> IO ()
```

Jeweils in 500x500-Fenster

Liften

Konstanten zu konstanten Behaviors machen:

```
lift0 :: a -> Beh a  
lift0 x = Beh (\t -> x)
```

Funktionen zu Funktionen auf Behaviors machen:

```
lift1 :: (a -> b) -> (Beh a -> Beh b)  
lift1 f (Beh b1)  
    = Beh (\t -> f (b1 t))  
  
lift2 :: (a -> b -> c) -> Beh a -> Beh b -> Beh c  
lift2 f (Beh b1) (Beh b2) =  
    Beh (\t -> f (b1 t) (b2 t))
```

Liften der Standard-Funktionen

```
round' :: Beh Double -> Beh Int
```

```
round' = lift1 round
```

```
constB :: a -> Beh a
```

```
constB x = Beh (\_ -> repeat x)
```

```
mapB :: (a -> b) -> Beh [a] -> Beh [b]
```

```
mapB f = lift1 (map f)
```

```
pairB :: Beh a -> Beh b -> Beh (a,b)
```

```
pairB = lift2 (,)
```

```
fstB :: Beh (a,b) -> Beh a
```

```
fstB = lift1 fst
```

```
sndB :: Beh (a,b) -> Beh b
```

```
sndB = lift1 snd
```

Instanzdeklarationen für Standard-Typklassen

```
instance Num a => Num (Beh a) where
  (+) = lift2 (+)
  (*) = lift2 (*)
  negate = lift1 negate
  abs = lift1 abs
  signum = lift1 signum
  fromInteger = lift0 . fromInteger
```

Ebenso für Fractional und Floating

Zeit als Behavior

```
time :: Beh Time
time = Beh (\t -> t)
```

Damit:

```
time + 5
```

Liften von Figure

```
circle :: Beh Int-> Beh Figure
```

```
circle = lift1 Circle
```

```
poly    :: Beh [Point] -> Beh Figure
```

```
poly    = lift1 Polygon
```

```
rect    :: Beh Int-> Beh Int-> Beh Figure
```

```
rect    = lift2 Rect
```

```
rot      :: Beh Double-> Beh Figure-> Beh Figure
```

```
scale    :: Beh Double-> Beh Figure-> Beh Figure
```

```
translate :: Beh Point-> Beh Figure-> Beh Figure
```

```
rot      = lift2 Rotate
```

```
scale    = lift2 Scale
```

```
translate = lift2 Translate
```

Hilfsfunktionen

Figure zeichnen:

```
drawB :: Beh Figure-> Beh Graphic
drawB = lift1 drawCentered
```

Grafiken überlagern:

```
overB :: Beh [Graphic]-> Beh Graphic
overB = lift1 overGraphics
over :: Beh Graphic-> Beh Graphic-> Beh Graphic
over = lift2 overGraphic
```

Beispiel: Planet mit Mond

```
goRound :: Beh Int-> Beh Figure-> Beh Figure
goRound r = rot time. translate (pairB r 0)
```

```
moon :: Beh Figure
moon = goRound 150 (goRound 40 (circle 10))
```

```
planet :: Beh Figure
planet = goRound 150 (circle 20)
```

```
solar :: Beh Graphic
solar = (paint red planet) 'over' (paint white moon)
reactimate "Sola" solar
```

Zufälliges Behavior

```
randomB :: Random a=> Int-> Double-> (a, a)-> Beh a  
randomB seed freq (from, to) = ...
```

Verändert sich alle `freq` Sekunden im Bereich zwischen `from` und `to`. `seed` ist Startwert.

Reaktives Programmieren

Benutzereingaben als Ereigniss, geschehen zu bestimmter Zeit:

```
data Ev a = (Time, a)
```

Mögliche Ereignisse:

- Druck auf Mausknopf `lbp, rbp :: Ev ()`
- Tastaturereignis `key :: Ev Char`
- Mausbewegung `mm :: Ev Point`

Anmerkung: Beh wird damit komplizierter

Verhalten nach Ereignis ändern

`untilB :: Beh a -> Ev (Beh a) -> Beh a`

Erstes Verhalten bis Ereignis, dann zweites Verhalten

`(->>) :: Ev a-> b-> Ev b`

Wenn erstes Ereignis eintritt, Ereignis aus zweitem Argument bilden

Beispiel:

`color1 :: Beh Color`

`color1 = red 'untilB' (lbp ->> blue)`

Beispiel

```
rbplanet :: Beh Graphic
```

```
rbplanet = paint color1 planet
```

```
paint :: Beh Color-> Beh Figure-> Beh Graphic
```

Ereignisse können unendlich oft auftreten

Rekursive Definition

```
color1r :: Beh Color
```

```
color1r = red 'untilB' lbp ->> blue 'untilB' lbp ->> color1r
```

⇒ Ev a entspricht Strom von Ereignissen

Ereignisauswahl

`(.|.) :: Ev a-> Ev a-> Ev a`

Tritt ein, wenn erstes oder zweites Ereignis eintritt

`color2 :: Beh Color`

`color2 = red 'untilB' (lbp ->> blue .|. rbp ->> yellow)`

Rekursiv:

`color2r = red 'untilB' colEv where`

`colEv = (lbp ->> blue 'untilB' colEv) .|.`

`(rbp ->> yellow 'untilB' colEv)`

switch

Abstraktion über Rekursion

```
switch :: Beh a -> Ev (Beh a) -> Beh a
a 'switch' b = (a 'untilB' b) 'switch' b
```

Damit:

```
color2r' :: Beh Color
color2r' = red 'switch' (lbp ->> blue .|. rbp ->> yellow)
```

Daten aus Ereignis lesen

`(=>>) :: Ev a -> (a -> b) -> Ev b`

Funktion berechnet Wert aus Daten im Ereignis

`color3 :: Beh Color`

```
color3 = white 'switch' (key =>> \c ->
    case c of 'r' -> red
              'g' -> green
              'y' -> yellow
              _   -> white)
```

snapshot

```
snapshot :: Ev a -> Beh b -> Ev (a,b)
```

Bei Ereignis Wert eines Behaviors einfangen

```
color4 :: Beh Color
```

```
color4 = white 'switch'
```

```
  (key 'snapshot' color4 ==>> \(c, old)->
```

```
    case c of 'r' -> red
```

```
              'g' -> green
```

```
              'y' -> yellow
```

```
              _   -> constB old)
```

Variante: `snapshot_ :: Ev a -> Beh b -> Ev b`

Boolsche Ereignisse

`when :: Beh Bool -> Ev ()`

Ereignis tritt in dem Moment ein, wenn Behavior wahr wird

`while :: Beh Bool -> Ev ()`

Ereignis tritt unendlich oft ein, sobald Behavior wahr ist

Operationen auf booleschen Behaviors

$(\&\&*)$, $(||*)$:: Beh Bool \rightarrow Beh Bool \rightarrow Beh Bool
 $(<*)$, $(>*)$:: Ord a \Rightarrow Beh a \rightarrow Beh a \rightarrow Beh Bool
cond :: Beh Bool \rightarrow Beh a \rightarrow Beh a \rightarrow Beh a

Damit:

```
color5 = red 'untilB' (when (time >* 5) ->> blue)
```

Integrieren und Ableiten

Gleichungen für Position und Geschwindigkeit unter Einfluss einer Kraft f :

$s, v :: \text{Beh Double}$

$s = s_0 + \text{integral } v$

$v = v_0 + \text{integral } f$

$\text{integral} :: \text{Beh Double} \rightarrow \text{Beh Double}$

$\text{derive} :: \text{Beh Double} \rightarrow \text{Beh Double}$

Integration von 0 bis t

Springender Ball

bouncy :: Beh Figure

```
bouncy = translate (pairB (round' x) (round' y))  
                (circ 20) where
```

```
g = 200
```

```
x = -250 + integral 50
```

```
y = -250 + integral v
```

```
v = integral g 'switch'
```

```
    (hit 'snapshot_' v ==>>
```

```
        \w-> lift0 (-w) + integral g)
```

```
hit = when (y >* 250)
```

Weitere Operatoren

`step :: a -> Ev a -> Beh a`

`a 'step' e = constB a 'switch' e ==>> constB`

Werte der Ereignisse als Behavior

`stepAccum :: a-> Event (a -> a)-> Beh a`

`stepAccum :: a -> Ev (a->a) -> Beh a`

`a 'stepAccum' e = b`

`where b = a 'step' (e 'snapshot' b ==>> \ (f, a)-> f a)`

Funktion im Event wird auf letzten Wert angewendet

`counter :: Beh Int`

`counter = 0 'stepAccum' lbp ->> (+1)`

Paddle-Ball (1)

```
walls :: Beh Graphic
```

```
walls = upper 'over' left 'over' right where
```

```
    upper = paint blue (trans (0, -230)
                               (rect 480 20))
```

```
    left  = paint blue (trans (-230, 0)
                               (rect 20 440))
```

```
    right = paint blue (trans (230, 0)
                               (rect 20 440))
```

```
trans :: (Beh Int, Beh Int) -> Beh Figure -> Beh Figure
```

```
trans (x, y) = translate (pairB x y)
```

Paddle-Ball (2)

```
paddle :: Beh Graphic
paddle = paint red (trans (fst mouse- 250, 220)
                          (rect 50 10))

mouse :: (Beh Int, Beh Int)
mouse = (fstB m, sndB m)
        where m = (0,0) 'step' mm
```

Paddle-Ball (3)

```
ball :: (Double, Double) -> Beh Graphic
ball (vx, vy) = paint yellow (trans (round' xpos,
                                     round' ypos) (circ 20)) where
  xvel      = vx 'stepAccum' xbounce ->> negate
  xpos      = 0+ integral xvel
  xbounce   = when (xpos >* 200 ||* xpos <* -200)
  yvel      = vy 'stepAccum' ybounce ->> negate
  ypos      = -220+ integral yvel
  ybounce   = when (ypos <* -200
                   ||* ypos 'between' (200, 220) &&*
                   (lift1 fromInt (fst mouse) - 250)
                   'between' (xpos- 25, xpos+ 25))
```

```
between :: Beh Double -> (Beh Double, Beh Double) -> Beh Bool
x 'between' (a,b) = x >* a &&* x <* b
```

Paddle-Ball (4)

```
paddleball :: (Double, Double)-> Beh Graphic
paddleball v = walls 'over' paddle 'over' ball v

pong :: Double-> IO ()
pong vel = do vx <- randomRIO (vel/2, vel)
             sx <- randomRIO (-1,1)
             reactimate "PONG"
                    (paddleball (signum sx * vx,
                                  - abs vel))
```