

Reaktive Programmierung  
Vorlesung 1 vom 02.04.19  
Was ist Reaktive Programmierung?

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Organisatorisches

- ▶ Vorlesung: Mittwochs 14-16, MZH 1110
- ▶ Übung: Donnerstags 16-18, MZH 1450 (nach Bedarf)
- ▶ Webseite: `www.informatik.uni-bremen.de/~cxl/lehre/rp.ss19`
- ▶ Scheinkriterien:
  - ▶ Voraussichtlich 6 Übungsblätter
  - ▶ Alle bearbeitet, insgesamt 40% (Notenspiegel PI3)
  - ▶ Übungsgruppen 2 – 4 Mitglieder
  - ▶ **Danach:** Fachgespräch **oder** Modulprüfung

# Warum Reaktive Programmierung?

Herkömmliche Sprachen:

- ▶ PHP, JavaScript, Ruby, Python
- ▶ C, C++, Java
- ▶ (Haskell)

Eigenschaften:

- ▶ **Imperativ** und **prozedural**
- ▶ **Sequentiell**

Zugrundeliegendes Paradigma:



# Warum Reaktive Programmierung?

Herkömmliche Sprachen:

- ▶ PHP, JavaScript, Ruby, Python
- ▶ C, C++, Java
- ▶ (Haskell)

Eigenschaften:

- ▶ **Imperativ** und **prozedural**
- ▶ **Sequentiell**

Zugrundeliegendes Paradigma:

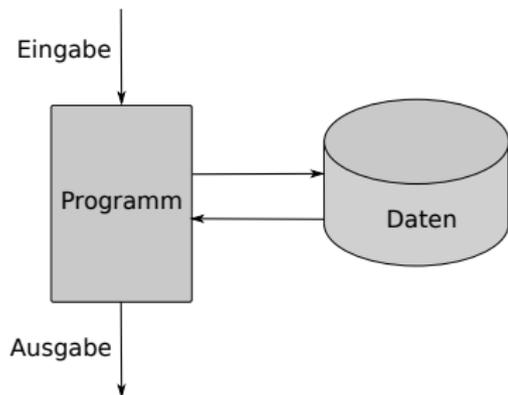


... aber die Welt ändert sich:



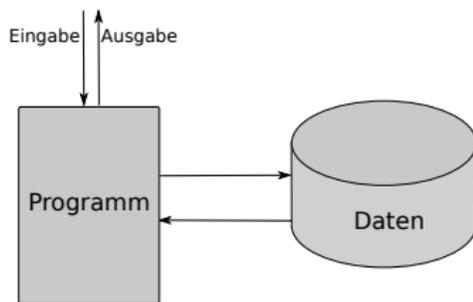
- ▶ Das **Netz** verbindet Rechner
- ▶ Selbst eingebettete Systeme sind vernetzt (Auto: ca. 130 Proz.)
- ▶ Mikroprozessoren sind **mehrkernig**
- ▶ Systeme sind **eingebettet**, **nebenläufig**, **reagieren** auf ihre Umwelt.

# Probleme mit dem herkömmlichen Ansatz



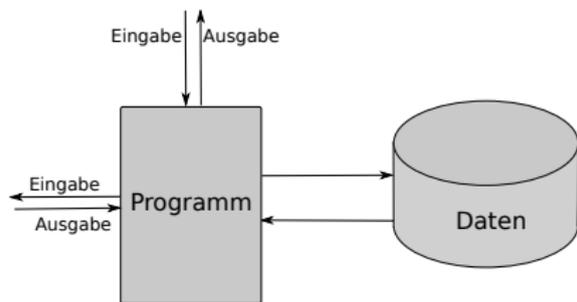
# Probleme mit dem herkömmlichen Ansatz

- ▶ Problem: **Nebenläufigkeit**

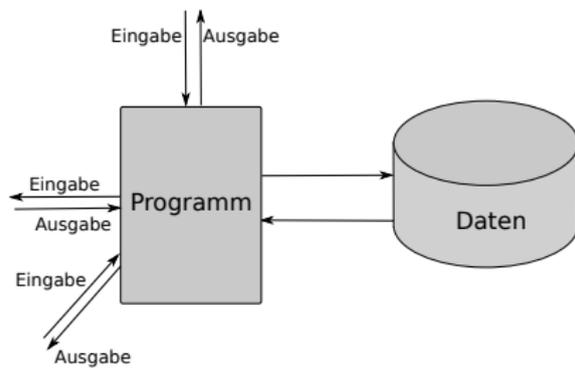


# Probleme mit dem herkömmlichen Ansatz

- ▶ Problem: **Nebenläufigkeit**



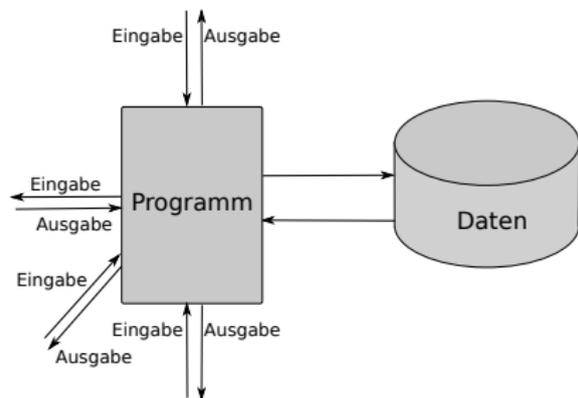
# Probleme mit dem herkömmlichen Ansatz



► Problem: **Nebenläufigkeit**

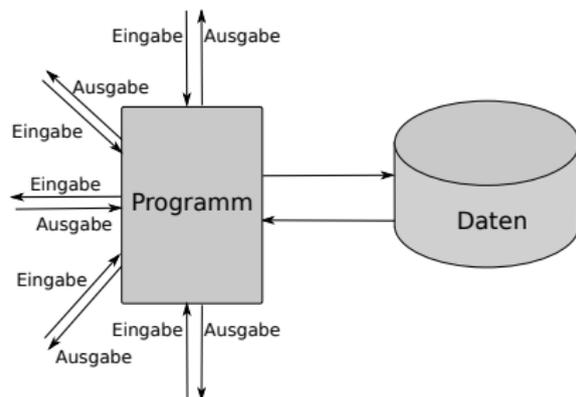
► Nebenläufigkeit verursacht **Synchronisationsprobleme**

# Probleme mit dem herkömmlichen Ansatz



- ▶ Problem: **Nebenläufigkeit**
- ▶ Nebenläufigkeit verursacht **Synchronisationsprobleme**
- ▶ Behandlung:
  - ▶ Callbacks (JavaScript, PHP)
  - ▶ Events (Java)
  - ▶ Global Locks (Python, Ruby)
  - ▶ Programmiersprachenkonstrukte: Locks, Semaphoren, Monitore

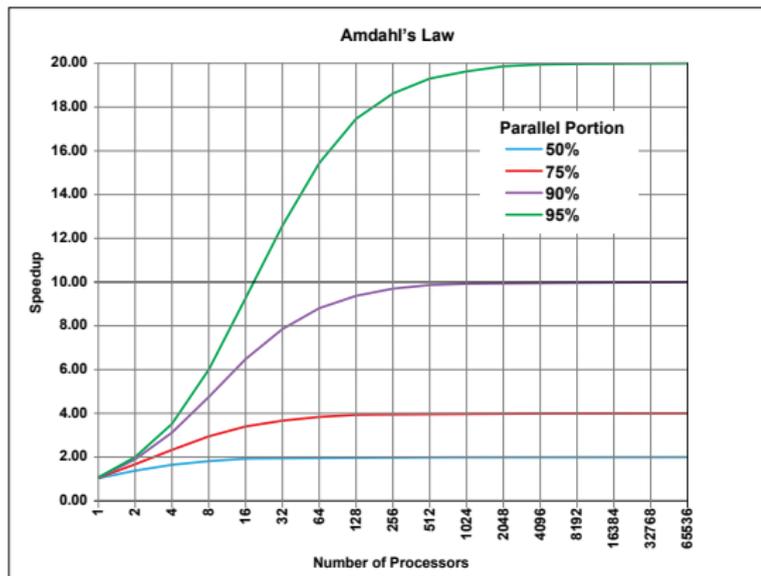
# Probleme mit dem herkömmlichen Ansatz



- ▶ Problem: **Nebenläufigkeit**
- ▶ Nebenläufigkeit verursacht **Synchronisationsprobleme**
- ▶ Behandlung:
  - ▶ Callbacks (JavaScript, PHP)
  - ▶ Events (Java)
  - ▶ Global Locks (Python, Ruby)
  - ▶ Programmiersprachenkonstrukte: Locks, Semaphoren, Monitore

# Amdahl's Law

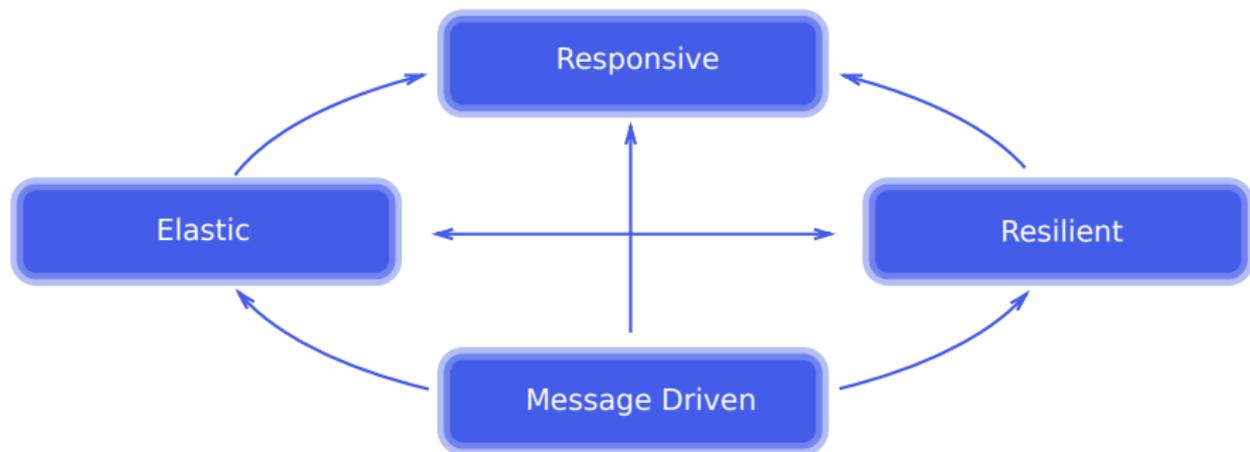
“The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be  $20\times$  as shown in the diagram, no matter how many processors are used.”



Quelle: Wikipedia

# The Reactive Manifesto

► <http://www.reactivemanifesto.org/>

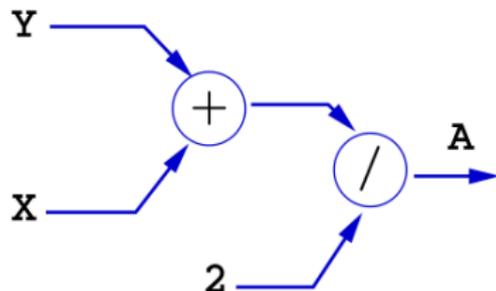


# Was ist Reaktive Programmierung?

- ▶ Imperative Programmierung: Zustandsübergang
- ▶ Prozedural und OO: Verkapselter Zustand
- ▶ Funktionale Programmierung: Abbildung (mathematische Funktion)
- ▶ **Reaktive** Programmierung:
  - ① **Datenabhängigkeit**
  - ② **Reaktiv** = funktional + nebenläufig

# Datenflusssprachen (data flow languages)

- ▶ Frühe Sprachen: VAL, SISAL, ID, LUCID (1980/1990)
- ▶ Heutige Sprachen: Esterel, Lustre (Gérard Berry, Verimag)
  - ▶ Keine **Zuweisungen**, sondern **Datenfluss**
  - ▶ **Synchron**: alle Aktionen ohne Zeitverzug
  - ▶ Verwendung in der Luftfahrtindustrie (Airbus)



```
node Average(X, Y : int)
returns (A : int);
let
    A = (X + Y) / 2 ;
tel
```

# Struktur der VL

- ▶ **Kernkonzepte** in Scala und Haskell:
  - ▶ Nebenläufigkeit: Futures, Aktoren, Reaktive Ströme
  - ▶ FFP: Bidirektionale und Meta-Programmierung, FRP, sexy types
  - ▶ Robustheit: Eventual Consistency, Entwurfsmuster
- ▶ Bilingualer **Übungsbetrieb** und **Vorlesung**
  - ▶ Kein Scala-Programmierkurs
  - ▶ Erlernen von Scala ist nützlicher Seiteneffekt

# Fahrplan

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

# Rückblick Haskell

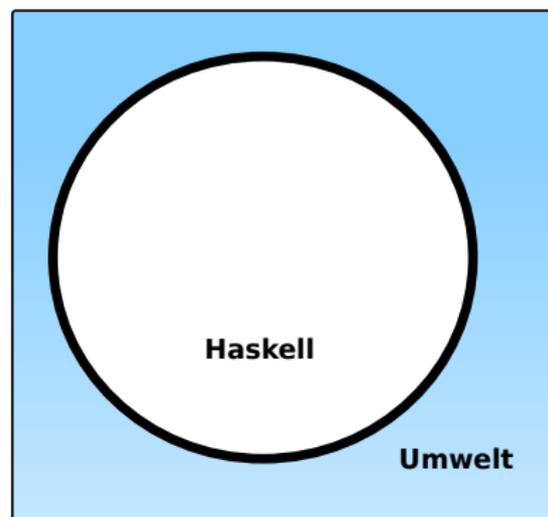
# Rückblick Haskell

- ▶ Definition von Funktionen:
  - ▶ lokale Definitionen mit **let** und **where**
  - ▶ Fallunterscheidung und guarded equations
  - ▶ Abseitsregel
  - ▶ Funktionen höherer Ordnung
- ▶ Typen:
  - ▶ Basisdatentypen: Int, Integer, Rational, Double, Char, Bool
  - ▶ Strukturierte Datentypen:  $[\alpha]$ ,  $(\alpha, \beta)$
  - ▶ Algebraische Datentypen: **data** Maybe  $\alpha = \text{Just } \alpha \mid \text{Nothing}$

# Rückblick Haskell

- ▶ Nichtstriktheit und verzögerte Auswertung
- ▶ Strukturierung:
  - ▶ Abstrakte Datentypen
  - ▶ Module
  - ▶ Typklassen

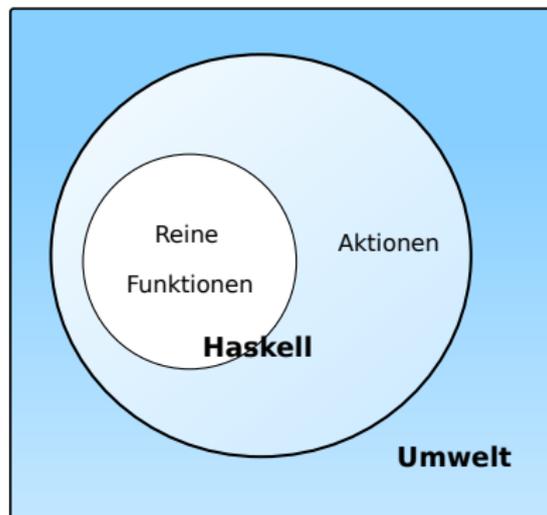
# Ein- und Ausgabe in Haskell



## Problem:

- ▶ Funktionen mit Seiteneffekten nicht referentiell transparent.
- ▶ `readString :: ... → String ??`

# Ein- und Ausgabe in Haskell



## Problem:

- ▶ Funktionen mit Seiteneffekten nicht referentiell transparent.
- ▶ `readString :: ... → String ??`

## Lösung:

- ▶ Seiteneffekte am Typ erkennbar
- ▶ **Aktionen** können *nur* mit **Aktionen** komponiert werden
- ▶ „einmal Aktion, immer Aktion“

# Aktionen als abstrakter Datentyp

- ▶ ADT mit Operationen **Komposition** und **Lifting**
- ▶ Signatur:

```
type IO  $\alpha$ 
```

```
( $\gg$ ) :: IO  $\alpha \rightarrow (\alpha \rightarrow \text{IO } \beta) \rightarrow \text{IO } \beta$ 
```

```
return ::  $\alpha \rightarrow \text{IO } \alpha$ 
```

- ▶ Plus **elementare** Operationen (lesen, schreiben etc)

# Elementare Aktionen

- ▶ Zeile von stdin lesen:

```
getLine  :: IO String
```

- ▶ Zeichenkette auf stdout ausgeben:

```
putStr   :: String → IO ()
```

- ▶ Zeichenkette mit Zeilenvorschub ausgeben:

```
putStrLn :: String → IO ()
```

# Die do-Notation

- ▶ Syntaktischer Zucker für IO:

```
echo =  
  getLine  
  >>= λs → putStrLn s  
  >> echo
```



```
echo = do  
  s ← getLine  
  putStrLn s  
  echo
```

- ▶ Rechts sind `>>=`, `>>` implizit.
- ▶ Es gilt die **Abseitsregel**.
- ▶ **Einrückung** der **ersten Anweisung** nach **do** bestimmt Abseits.

# Zustandsabhängige Berechnungen

# Funktionen mit Zustand

- ▶ Idee: Seiteneffekt **explizit** machen
- ▶ Funktion  $f : A \rightarrow B$  mit Seiteneffekt in **Zustand**  $S$ :

$$\begin{aligned} f : A \times S &\rightarrow B \times S \\ &\cong \\ f : A &\rightarrow S \rightarrow B \times S \end{aligned}$$

- ▶ Datentyp:  $S \rightarrow B \times S$
- ▶ Komposition: Funktionskomposition und uncurry

```
curry    :: (( $\alpha$ ,  $\beta$ )  $\rightarrow$   $\gamma$ )  $\rightarrow$   $\alpha \rightarrow \beta \rightarrow \gamma$   
uncurry  :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow$  ( $\alpha$ ,  $\beta$ )  $\rightarrow \gamma$ 
```

# In Haskell: Zustände **explizit**

- ▶ **Zustandstransformer:** Berechnung mit Seiteneffekt in Typ  $\sigma$  (polymorph über  $\alpha$ )

```
type State  $\sigma$   $\alpha$  =  $\sigma \rightarrow (\alpha, \sigma)$ 
```

- ▶ Komposition zweier solcher Berechnungen:

```
comp :: State  $\sigma$   $\alpha \rightarrow (\alpha \rightarrow$  State  $\sigma$   $\beta) \rightarrow$  State  $\sigma$   $\beta$   
comp f g = uncurry g  $\circ$  f
```

- ▶ Trivialer Zustand:

```
lift ::  $\alpha \rightarrow$  State  $\sigma$   $\alpha$   
lift = curry id
```

- ▶ Lifting von Funktionen:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  State  $\sigma$   $\alpha \rightarrow$  State  $\sigma$   $\beta$   
map f g = ( $\lambda(a, s) \rightarrow (f a, s)$ )  $\circ$  g
```

# Zugriff auf den Zustand

- ▶ Zustand lesen:

```
get :: ( $\sigma \rightarrow \alpha$ )  $\rightarrow$  State  $\sigma$   $\alpha$   
get f s = (f s, s)
```

- ▶ Zustand setzen:

```
set :: ( $\sigma \rightarrow \sigma$ )  $\rightarrow$  State  $\sigma$  ()  
set g s = ((), g s)
```

# Einfaches Beispiel

- ▶ Zähler als Zustand:

```
type WithCounter  $\alpha$  = State Int  $\alpha$ 
```

- ▶ Beispiel: Funktion, die in Kleinbuchstaben konvertiert und **zählt**

```
cntToL :: String  $\rightarrow$  WithCounter String
cntToL [] = lift ""
cntToL (x:xs)
  | isUpper x = cntToL xs 'comp'
                 $\lambda$ ys  $\rightarrow$  set (+1) 'comp'
                 $\lambda$ ()  $\rightarrow$  lift (toLower x: ys)
  | otherwise = cntToL xs 'comp'  $\lambda$ ys  $\rightarrow$  lift (x: ys)
```

- ▶ Hauptfunktion (verkapselt State):

```
cntToLower :: String  $\rightarrow$  (String, Int)
cntToLower s = cntToL s 0
```

# Monaden

# Monaden als Berechnungsmuster

- ▶ In cntToL werden zustandsabhängige Berechnungen verkettet.
- ▶ So ähnlich wie bei Aktionen!

State:

```
type State  $\sigma$   $\alpha$ 
```

```
comp :: State  $\sigma$   $\alpha$   $\rightarrow$   
      ( $\alpha \rightarrow$  State  $\sigma$   $\beta$ )  $\rightarrow$   
      State  $\sigma$   $\beta$ 
```

```
lift ::  $\alpha \rightarrow$  State  $\sigma$   $\alpha$ 
```

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  State  $\sigma$   $\alpha \rightarrow$   
      State  $\sigma$   $\beta$ 
```

Aktionen:

```
type IO  $\alpha$ 
```

```
( $\gg$ ) :: IO  $\alpha$   $\rightarrow$   
      ( $\alpha \rightarrow$  IO  $\beta$ )  $\rightarrow$   
      IO  $\beta$ 
```

```
return ::  $\alpha \rightarrow$  IO  $\alpha$ 
```

```
fmap :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$  IO  $\alpha \rightarrow$   
      IO  $\beta$ 
```

Berechnungsmuster: **Monade**

# Monaden als Berechnungsmuster

Eine Monade ist:

- ▶ **mathematisch**: durch Operationen und Gleichungen definiert (verallgemeinerte algebraische Theorie)
- ▶ als **Berechnungsmuster**: **verknüpfbare** Berechnungen mit einem **Ergebnis**
- ▶ in **Haskell**: durch mehrere Typklassen definierte Operationen mit **Eigenschaften**

# Monaden in Haskell

- ▶ Aktion auf Funktionen:

```
class Functor f where  
  fmap :: (a → b) → f a → f b
```

fmap bewahrt Identität und Komposition:

```
fmap id == id  
fmap (f ∘ g) == fmap f ∘ fmap g
```

- ▶ Die Eigenschaften **sollten** gelten, können aber nicht überprüft werden.
  - ▶ Standard: “*Instances of Functor should satisfy the following laws.*”

# Monaden in Haskell

► Applicative:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  <*>  :: f (a -> b) -> f a -> f b
```

Eigenschaften: links-neutralität, bewahrt Komposition,  
Homomorphismus:

```
pure id <*> v == v
pure (o) <*> u <*> v <*> w == u <*> (v <*> w)
pure f <*> pure x == pure (f x)
u <*> pure y == pure ($) y <*> u
```

# Monaden in Haskell

- ▶ Verkettung ( $\gg=$ ) und Lifting (return):

```
class Applicative m => Monad m where
  (gg=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

$\gg=$  ist assoziativ und return das neutrale Element:

```
return a gg= k == k a
m gg= return == m
m gg= (\x -> k x gg= h) == (m gg= k) gg= h
```

- ▶ Auch diese Eigenschaften können nicht geprüft werden.
- ▶ Den syntaktischen Zucker (**do**-Notation) gibt's umsonst dazu.

# Monaden mit Möglichkeiten

- ▶ Alternativen:

```
class Applicative f  $\Rightarrow$  Alternative f where  
  empty :: f a  
  <|>   :: f a  $\rightarrow$  f a  $\rightarrow$  f a
```

- ▶ Monaden mit Alternative (e.g. List):

```
class (Alternative m, Monad m)  $\Rightarrow$  MonadPlus m where  
  mzero :: m a  
  mzero = empty  
  mplus :: m a  $\rightarrow$  m a  $\rightarrow$  m a  
  mplus = (<|>)
```

- ▶ Gleichungen: mzero Identität für mplus und  $\gg=$ , mplus assoziativ.

# Beispiele für Monaden

- ▶ Zustandstransformer: Reader, Writer, State
- ▶ Fehler und Ausnahmen: Maybe, Either
- ▶ Mehrdeutige Berechnungen: List, Set

# Die Reader-Monade

- ▶ Aus dem Zustand wird nur gelesen:

```
data Reader  $\sigma$   $\alpha$  = R {run ::  $\sigma \rightarrow \alpha$ }
```

- ▶ Instanzen:

```
instance Functor (Reader  $\sigma$ ) where  
  fmap f (R g) = R (f . g)
```

```
instance Monad (Reader  $\sigma$ ) where  
  return a = R (const a)  
  R f  $\gg$ = g = R $  $\lambda s \rightarrow$  run (g (f s)) s
```

- ▶ Nur eine elementare Operation:

```
get :: ( $\sigma \rightarrow \alpha$ )  $\rightarrow$  Reader  $\sigma$   $\alpha$   
get f = R $  $\lambda s \rightarrow$  f s
```

# Fehler und Ausnahmen

- ▶ Maybe als Monade:

```
instance Functor Maybe where  
  fmap f (Just a) = Just (f a)  
  fmap f Nothing = Nothing
```

```
instance Monad Maybe where  
  Just a >>= g = g a  
  Nothing >>= g = Nothing  
  return = Just
```

- ▶ Ähnlich mit Either
- ▶ Berechnungsmodell: **Ausnahmen** (Fehler)
  - ▶  $f :: \alpha \rightarrow \text{Maybe } \beta$  ist Berechnung mit möglichem Fehler
  - ▶ Fehlerfreie Berechnungen werden verkettet
  - ▶ Fehler (Nothing oder Left x) werden propagiert

# Mehrdeutigkeit

- ▶ List als Monade:
- ▶ Können wir so nicht hinschreiben, Syntax vordefiniert

```
instance Functor [ $\alpha$ ] where  
  fmap = map
```

```
instance Monad [ $\alpha$ ] where  
  a : as  $\gg=$  g = g a ++ (as  $\gg=$  g)  
  []  $\gg=$  g = []  
  return a = [a]
```

- ▶ Berechnungsmodell: Mehrdeutigkeit
- ▶  $f :: \alpha \rightarrow [\beta]$  ist Berechnung mit **mehreren** möglichen Ergebnissen
- ▶ Verkettung: Anwendung der folgenden Funktion auf **jedes** Ergebnis (concatMap)

# Beispiel

## ► Berechnung aller Permutationen einer Liste:

### ① Ein Element überall in eine Liste einfügen:

```
ins ::  $\alpha \rightarrow [\alpha] \rightarrow [[\alpha]]$   
ins x [] = return [x]  
ins x (y:ys) = [x:y:ys] ++ do  
  is  $\leftarrow$  ins x ys  
  return $ y:is
```

### ② Damit Permutationen (rekursiv):

```
perms ::  $[\alpha] \rightarrow [[\alpha]]$   
perms [] = return []  
perms (x:xs) = do  
  ps  $\leftarrow$  perms xs  
  is  $\leftarrow$  ins x ps  
  return is
```

# Der Listenmonade in der Listenkomprehension

- ▶ Berechnung aller Permutationen einer Liste:

- 1 Ein Element überall in eine Liste einfügen:

```
ins' ::  $\alpha \rightarrow [\alpha] \rightarrow [[\alpha]]$   
ins' x [] = [[x]]  
ins' x (y:ys) = [x:y:ys] ++ map (y :) (ins' x ys)
```

- 2 Damit Permutationen (rekursiv):

```
perms' ::  $[\alpha] \rightarrow [[\alpha]]$   
perms' [] = [[]]  
perms' (x:xs) = [is | ps  $\leftarrow$  perms' xs, is  $\leftarrow$  ins' x ps ]
```

- ▶ Listenkomprehension  $\cong$  Listenmonade

# IO ist keine Magie

# Referenzen in Haskell

- ▶ Zustand als **finite map** von Referenzen auf Werte
- ▶ Ungetypt: SimpleRefs
- ▶ Typ der Werte ist Typparameter des Zustands

```
readRef :: Ref → Stateful a a  
writeRef :: Ref → a → Stateful a ()
```

- ▶ Getypt: Refs
  - ▶ Typ der Werte durch Typparameter der Referenz
  - ▶ Nutzt **dynamische Typen**:

```
readRef :: Typeable a ⇒ Ref a → Stateful a  
writeRef :: Typeable a ⇒ Ref a → a → Stateful ()
```

# Implizite vs. explizite Zustände

- ▶ Wie funktioniert jetzt IO?
- ▶ Nachteil von State: Zustand ist **explizit**
  - ▶ Kann dupliziert werden
- ▶ Daher: Zustand **implizit** machen
  - ▶ Datentyp verkapseln (kein run)
  - ▶ Zugriff auf State nur über elementare Operationen

# Aktionen als Zustandstransformationen

- ▶ **Idee:** Aktionen sind Transformationen auf Systemzustand  $S$
- ▶  $S$  beinhaltet
  - ▶ Speicher als Abbildung  $A \rightarrow V$  (Adressen  $A$ , Werte  $V$ )
  - ▶ Zustand des Dateisystems
  - ▶ Zustand des Zufallsgenerators
- ▶ In Haskell: Typ `RealWorld`
  - ▶ “Virtueller” Typ, Zugriff nur über elementare Operationen
  - ▶ Entscheidend nur Reihenfolge der Aktionen

# Zusammenfassung

- ▶ War das jetzt **reaktiv**?
- ▶ Haskell ist **funktional**
- ▶ Für eine reaktive Sprache fehlt **Nebenläufigkeit**
- ▶ Nächstes Mal:
  - ▶ Monaden **komponieren** — Monadentransformer
- ▶ Danach: Nebenläufigkeit in Haskell und Scala

Reaktive Programmierung  
Vorlesung 2 vom 10.04.2019  
Monaden und Monadentransformer

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Inhalt

- ▶ Monaden zusammensetzen
- ▶ Monadentransformer
- ▶ Monaden in Scala

# Monaden

# Beispiele für Monaden

- ▶ Zustandstransformer: Reader, Writer, State
- ▶ Fehler und Ausnahmen: Maybe, Either
- ▶ Mehrdeutige Berechnungen: List, Set

# Fallbeispiel: Auswertung von Ausdrücken

# Monaden im Einsatz

- ▶ Auswertung von Ausdrücken:

```
data Expr = Var String
         | Num Double
         | Plus Expr Expr
         | Minus Expr Expr
         | Times Expr Expr
         | Div Expr Expr
```

- ▶ Mögliche Arten von Effekten:
  - ▶ Partialität (Division durch 0)
  - ▶ Zustände (für die Variablen)
  - ▶ Mehrdeutigkeit

# Monaden im Einsatz

- ▶ Auswertung von Ausdrücken:

```
data Expr = Var String
          | Num Double
          | Plus Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
          | Div Expr Expr
```

- ▶ Mögliche Arten von Effekten:

- ▶ Partialität (Division durch 0)
- ▶ Zustände (für die Variablen)
- ▶ Mehrdeutigkeit

- ▶ Auswertung ohne Effekte:

```
eval :: Expr → Double
eval (Var _) = 0
eval (Num n) = n
eval (Plus a b) = eval a + eval b
eval (Minus a b) = eval a - eval b
eval (Times a b) = eval a * eval b
eval (Div a b) = eval a / eval b
```

# Auswertung mit Fehlern

## ► Partialität durch Maybe-Monade

```
eval :: Expr → Maybe Double
eval (Var _) = return 0
eval (Num n) = return n
eval (Plus a b) = do x ← eval a; y ← eval b; return $ x + y
eval (Minus a b) = do x ← eval a; y ← eval b; return $ x - y
eval (Times a b) = do x ← eval a; y ← eval b; return $ x * y
eval (Div a b) = do
  x ← eval a; y ← eval b; if y == 0 then Nothing else Just $ x / y
```

# Auswertung mit Zustand

- ▶ Zustand durch Reader-Monade

```
import ReaderMonad
import qualified Data.Map as M
type State = M.Map String Double
eval :: Expr → Reader State Double
eval (Var i) = get (M.! i)
eval (Num n) = return n
eval (Plus a b) = do x ← eval a; y ← eval b; return $ x + y
eval (Minus a b) = do x ← eval a; y ← eval b; return $ x - y
eval (Times a b) = do x ← eval a; y ← eval b; return $ x * y
eval (Div a b) = do x ← eval a; y ← eval b; return $ x / y
```

# Mehrdeutige Auswertung

- Dazu: Erweiterung von Expr:

```
data Expr = Var String
          | ...
          | Pick Expr Expr
```

```
eval :: Expr → [Double]
eval (Var i) = return 0
eval (Num n) = return n
eval (Plus a b) = do x ← eval a; y ← eval b; return $ x + y
eval (Minus a b) = do x ← eval a; y ← eval b; return $ x - y
eval (Times a b) = do x ← eval a; y ← eval b; return $ x * y
eval (Div a b) = do x ← eval a; y ← eval b; return $ x / y
eval (Pick a b) = do x ← eval a; y ← eval b; [x, y]
```

# Kombination der Effekte

- ▶ Benötigt **Kombination** der Monaden.
- ▶ Monade Res:
  - ▶ Zustandsabhängig
  - ▶ Mehrdeutig
  - ▶ Fehlerbehaftet

```
data Res  $\sigma$   $\alpha$  = Res { run ::  $\sigma \rightarrow$  [Maybe  $\alpha$ ] }
```

- ▶ Andere Kombinationen möglich:

```
data Res  $\sigma$   $\alpha$  = Res ( $\sigma \rightarrow$  Maybe [ $\alpha$ ])
```

# Kombination der Effekte

- ▶ Benötigt **Kombination** der Monaden.
- ▶ Monade Res:
  - ▶ Zustandsabhängig
  - ▶ Mehrdeutig
  - ▶ Fehlerbehaftet

```
data Res  $\sigma$   $\alpha$  = Res { run ::  $\sigma \rightarrow$  [Maybe  $\alpha$ ] }
```

- ▶ Andere Kombinationen möglich:

```
data Res  $\sigma$   $\alpha$  = Res ( $\sigma \rightarrow$  Maybe [ $\alpha$ ])
```

```
data Res  $\sigma$   $\alpha$  = Res ( $\sigma \rightarrow$  [ $\alpha$ ])
```

# Kombination der Effekte

- ▶ Benötigt **Kombination** der Monaden.
- ▶ Monade Res:
  - ▶ Zustandsabhängig
  - ▶ Mehrdeutig
  - ▶ Fehlerbehaftet

```
data Res  $\sigma$   $\alpha$  = Res { run ::  $\sigma \rightarrow$  [Maybe  $\alpha$ ] }
```

- ▶ Andere Kombinationen möglich:

```
data Res  $\sigma$   $\alpha$  = Res ( $\sigma \rightarrow$  Maybe [ $\alpha$ ])
```

```
data Res  $\sigma$   $\alpha$  = Res ( $\sigma \rightarrow$  [ $\alpha$ ])
```

```
data Res  $\sigma$   $\alpha$  = Res ([ $\sigma \rightarrow$   $\alpha$ ])
```

# Res: Monadeninstanz

- ▶ Functor durch Komposition der fmap:

```
instance Functor (Res  $\sigma$ ) where  
  fmap f (Res g) = Res $ fmap (fmap f). g
```

- ▶ Monad ist Kombination

```
instance Monad (Res  $\sigma$ ) where  
  return a = Res (const [Just a])  
  Res f  $\gg$ = g = Res $  $\lambda$ s  $\rightarrow$  do ma  $\leftarrow$  f s  
                                case ma of  
                                  Just a  $\rightarrow$  run (g a) s  
                                  Nothing  $\rightarrow$  return Nothing
```

# Res: Operationen

- ▶ Zugriff auf den Zustand:

```
get :: ( $\sigma \rightarrow \alpha$ )  $\rightarrow$  Res  $\sigma$   $\alpha$   
get f = Res $  $\lambda s \rightarrow$  [Just $ f s]
```

- ▶ Fehler:

```
fail :: Res  $\sigma$   $\alpha$   
fail = Res $ const [Nothing]
```

- ▶ Mehrdeutige Ergebnisse:

```
join ::  $\alpha \rightarrow \alpha \rightarrow$  Res  $\sigma$   $\alpha$   
join a b = Res $  $\lambda s \rightarrow$  [Just a, Just b]
```

# Auswertung mit Allem

- ▶ Im Monaden Res können alle Effekte benutzt werden:

```
type State = M.Map String Double

eval :: Expr → Res State Double
eval (Var i) = get (M.! i)
eval (Num n) = return n
eval (Plus a b) = do x ← eval a; y ← eval b; return $ x + y
eval (Minus a b) = do x ← eval a; y ← eval b; return $ x - y
eval (Times a b) = do x ← eval a; y ← eval b; return $ x * y
eval (Div a b) = do x ← eval a; y ← eval b
                  if y == 0 then fail else return $ x / y
eval (Pick a b) = do x ← eval a; y ← eval b; join x y
```

- ▶ Systematische Kombination durch **Monadentransformer**
- ▶ Monade mit Platzhalter für weitere Monaden

# Kombination von Monaden

# Das Problem

- ▶ Monaden sind nicht **kompositional**:

```
type mn a = m (n a)
```

```
instance (Monad m, Monad n) => Monad mn
```

- ▶ Warum?
  - ▶ Wie wären  $\gg=$ , `return` definiert?
- ▶ Funktoren **sind** kompositional.

# Die “Lösung”

- ▶ Monadentransformer
  
- ▶ Monaden mit einem “Loch” (i.e. parametrisierte Monaden)

# Beispiel

- ▶ Zustandsmonadentransformer: `StateMonadT`

```
data StateT m s a = St { runSt :: s → m (a, s) }
```

- ▶ Ausnahmenmonadentransformer: `ExnMonadT`

```
data ExnT m e a = ExnT { runEx :: m (Either e a) }
```

- ▶ Komposition:

```
type ResMonad a = StateT (ExnT Identity Error) State a
```

# Probleme

- ▶ “Lifting” von Hand
- ▶ Komposition muss fallweise entschieden werden:
  - ▶ Exception und Writer kann kanonisch mit allen kombiniert werden
  - ▶ State und List nicht mit allen, oder unterschiedlich

# Monadtransformer in Haskell: mtl

- ▶ Klassendeklarationen erlauben Typinferenz für automatisches Lifting
- ▶ Zustandsmonaden, Exceptions, Reader, Writer, Listen, IO
- ▶ Fallbeispiel: Interpreter für eine imperative Sprache

# Zusammenfassung

- ▶ Monaden sind **Muster** für **Berechnungen** mit **Seiteneffekten**
- ▶ Beispiele:
  - ▶ Zustandstransformer
  - ▶ Fehler und Ausnahmen
  - ▶ Nichtdeterminismus
- ▶ Kombination von Monaden: **Monadentransformer**
  - ▶ Monadentransformer: parametrisierte Monaden
  - ▶ mtl-Bücherei erleichtert Kombination
  - ▶ Prinzipielle Begrenzungen
- ▶ Grenze: Nebenläufigkeit → Nächste Vorlesung

Reaktive Programmierung  
Vorlesung 3 vom 24.04.2019  
Nebenläufigkeit: Futures and Promises

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Inhalt

- ▶ Konzepte der Nebenläufigkeit
- ▶ Nebenläufigkeit in Scala und Haskell
- ▶ Futures and Promises

# Konzepte der Nebenläufigkeit

# Begrifflichkeiten

- | ▶ <b>Thread (lightweight process)</b>                           | vs. | <b>Prozess</b>         |
|---|-----|------------------------|
| Programmiersprache/Betriebssystem<br>(z.B. Java, Haskell/Linux) |     | Betriebssystem         |
| gemeinsamer Speicher  |     | getrennter Speicher    |
| Erzeugung <b>billig</b>   |     | Erzeugung <b>teuer</b> |
| mehrere pro Programm  |     | einer pro Programm     |
- 
- ▶ Multitasking:
    - ▶ **präemptiv:** Kontextwechsel wird **erzungen**
    - ▶ **kooperativ:** Kontextwechsel nur **freiwillig**

# Threads in Java

- ▶ Erweiterung der Klassen `Thread` oder `Runnable`
- ▶ Gestartet wird Methode `run()` — durch eigene überladen
- ▶ Starten des Threads durch Aufruf der Methode `start()`
- ▶ Kontextwechsel mit `yield()`
- ▶ Je nach JVM kooperativ **oder** präemptiv.
- ▶ Synchronisation mit **Monitoren** (`synchronize`)

# Threads in Scala

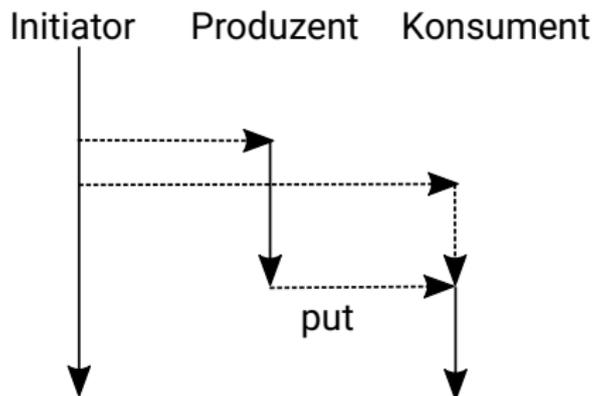
- ▶ Scala nutzt das Threadmodell der JVM
  - ▶ Kein sprachspezifisches Threadmodell
- ▶ Daher sind Threads vergleichsweise **teuer**.
- ▶ Synchronisation auf unterster Ebene durch Monitore (`synchronized`)
- ▶ Bevorzugtes Abstraktionsmodell: **Aktoren** (dazu später mehr)

# Threads in Haskell: Concurrent Haskell

- ▶ **Sequentielles Haskell:** Reduktion eines Ausdrucks
  - ▶ Auswertung
- ▶ **Nebenläufiges Haskell:** Reduktion eines Ausdrucks an **mehreren Stellen**
  - ▶ `ghc` implementiert Haskell-Threads
  - ▶ Zeitscheiben (Default 20ms), Kontextwechsel bei Heapallokation
  - ▶ Threaderzeugung und Kontextswitch sind **billig**
- ▶ Modul `Control.Concurrent` enthält Basisfunktionen
- ▶ Wenige Basisprimitive, darauf aufbauend Abstraktionen
- ▶ Synchronisation mit Futures

# Futures

- ▶ Futures machen Nebenläufigkeit **explizit**
- ▶ Grundprinzip:
  - ▶ Ausführung eines Threads wird **verzögert**
  - ▶ Konsument startet erst, wenn Ergebnis vorhanden.



Note: Not a UML sequence diagram

# Futures in Scala

# Futures in Scala

- ▶ Antwort als **Callback**:

```
trait Future[+T] {  
  def onComplete(f: Try[T] => Unit): Unit  
  def map[U](f: T => U): Future[U]  
  def flatMap[U](f: T => Future[U]): Future[U]  
  def filter(p: T => Boolean): Future[T]  
}
```

- ▶ `map`, `flatMap`, `filter` für monadische Notation
- ▶ Factory-Methode für einfache Erzeugung
- ▶ Vordefiniert in `scala.concurrent.Future`, Beispielimplementation `Future.scala`

## Beispiel: Robot.scala

- ▶ Roboter, kann sich um n Positionen bewegen:

```
if (n ≤ 0) this
else if (battery > 0) {
  Thread.sleep(100*Random.nextInt(10));
  Robot(id, pos+1, battery- 1).mv(n-1)
} else throw new LowBatteryException
```

```
def move(n: Int): Future[Robot] = Future { mv(n) }
```

```
override def toString = s"Robot # $id at $pos [battery: $battery]"
```

## Beispiel: Moving the robots

```
object Examples {  
  def ex1 = {  
    val robotSwarm = List.range(1,6).map{i⇒ Robot(i,0,10)}  
    val moved = robotSwarm.map(_.move(10))  
    moved.map(_.onComplete(println))  
    println("Started moving...")  
  }  
}
```

- ▶ 6 Roboter erzeugen, alle um zehn Positionen bewegen.
- ▶ Wie lange dauert das?
  - ▶ 0 Sekunden (nach spät. 10 Sekunden Futures erfüllt)
- ▶ Was wir verschweigen: `ExecutionContext`

# Compositional Futures

- ▶ Wir können Futures komponieren
  - ▶ “Spekulation auf die Zukunft”
- ▶ Beispiel: Roboterbewegung

```
def ex2 = { val r = Robot(99, 0, 20); for {  
  r1 ← r.move(3)  
  r2 ← r1.move(5)  
  r3 ← r2.move(2)
```

- ▶ Fehler (Failure) werden propagiert

# Promises

- ▶ Promises sind das Gegenstück zu Futures

```
trait Promise {  
  def complete(result: Try[T])  
  def success(result: T)  
  def future: Future[T]  
}  
  
object Promise {  
  def apply[T]: Promise[T] = ...  
}
```

- ▶ Das Future eines Promises wird durch die `complete` Methode **erfüllt**.

# Futures in Haskell

# Concurrent Haskell: Wesentliche Typen und Funktionen

- ▶ Jeder Thread hat einen Identifier: abstrakter Typ `ThreadId`
- ▶ Neuen Thread erzeugen: `forkIO :: IO () → IO ThreadId`
- ▶ Thread stoppen: `killThread :: ThreadId → IO ()`
- ▶ Kontextwechsel: `yield :: IO ()`
- ▶ Eigener Thread: `myThreadId :: IO ThreadId`
- ▶ Warten: `threadDelay :: Int → IO ()`

# Concurrent Haskell — erste Schritte

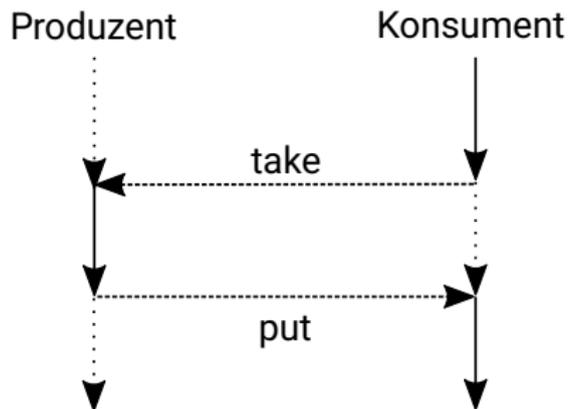
- ▶ Ein einfaches Beispiel:

```
write :: Char → IO ()  
write c = do putChar c; write c  
  
main :: IO ()  
main = do forkIO (write 'X'); write 'O'
```

- ▶ Ausgabe ghc:  $(X^*|O^*)^*$

# Futures in Haskell: MVars

- ▶ **Basissynchronisationsmechanismus** in Concurrent Haskell
  - ▶ Alles andere abgeleitet
- ▶ Grundprinzip:



Note: Not a UML sequence diagram

# Futures in Haskell: MVars

- ▶ MVar  $\alpha$  ist **polymorph** über dem Inhalt
- ▶ Entweder **leer** oder **gefüllt** mit Wert vom Typ  $\alpha$
- ▶ Verhalten beim Lesen und Schreiben:

Zustand vorher:	leer	gefüllt
Lesen	<b>blockiert</b> (bis gefüllt)	danach leer
Schreiben	danach gefüllt	<b>blockiert</b> (bis leer)

- ▶ NB. **Aufwecken** blockierter Prozesse **einzeln** in **FIFO**

# Basisfunktionen MVars

- ▶ Neue Variable erzeugen (leer oder gefüllt):

```
newEmptyMVar :: IO (MVar  $\alpha$ )  
newMVar     ::  $\alpha \rightarrow$  IO (MVar  $\alpha$ )
```

- ▶ Lesen:

```
takeMVar :: MVar  $\alpha \rightarrow$  IO  $\alpha$ 
```

- ▶ Schreiben:

```
putMVar :: MVar  $\alpha \rightarrow \alpha \rightarrow$  IO ()
```

- ▶ Es gibt noch weitere (nicht-blockierend lesen/schreiben, Test ob gefüllt, map etc.)

# Ein einfaches Beispiel: Robots Revisited

```
data Robot = Robot {id :: Int, pos :: Int, battery :: Int}
```

- ▶ Hauptfunktion: MVar anlegen, nebenläufig Bewegung starten

```
move :: Robot → Int → IO (MVar Robot)
```

```
move r n = do
```

```
  m ← newEmptyMVar; forkIO (mv m r n); return m
```

- ▶ Bewegungsfunktion:

```
mv :: MVar Robot → Robot → Int → IO ()
```

```
mv v r n
```

```
  | n ≤ 0 = putMVar v r
```

```
  | otherwise = do
```

```
    m ← randomRIO(0,10); threadDelay(m*100000)
```

```
    mv v r {pos = pos r + 1, battery = battery r - 1} (n-1)
```

# Abstraktion von Futures

- ▶ Aus  $MVar\ \alpha$  konstruierte Abstraktionen
- ▶ Semaphoren (`QSem` aus `Control.Concurrent.QSem`):

```
waitQSem    :: QSem → IO ()  
signalQSem  :: QSem → IO ()
```

- ▶ Siehe `Sem.hs`
- ▶ Damit auch `synchronized` wie in Java (huzzah!)
- ▶ Kanäle (`Chan  $\alpha$`  aus `Control.Concurrent.Chan`):

```
writeChan  :: Chan  $\alpha$  →  $\alpha$  → IO ()  
readChan   :: Chan  $\alpha$  → IO  $\alpha$ 
```

# Asynchrone Ausnahmen

- ▶ Ausnahmen unterbrechen den sequentiellen Kontrollfluß
- ▶ In Verbindung mit Nebenläufigkeit **überraschende Effekte**:

```
m ← newEmptyMVar
forkIO (do {s← takeMVar m; putStrLn s})
threadDelay (100000)
putMVar m (error "FOO!")
```

- ▶ In welchem Thread wird die Ausnahme geworfen?

# Asynchrone Ausnahmen

- ▶ Ausnahmen unterbrechen den sequentiellen Kontrollfluß
- ▶ In Verbindung mit Nebenläufigkeit **überraschende Effekte**:

```
m ← newEmptyMVar
forkIO (do {s← takeMVar m; putStrLn s})
threadDelay (100000)
putMVar m (error "FOO!")
```

- ▶ In welchem Thread wird die Ausnahme geworfen?
- ▶ Wo kann sie gefangen werden?

# Asynchrone Ausnahmen

- ▶ Ausnahmen unterbrechen den sequentiellen Kontrollfluß
- ▶ In Verbindung mit Nebenläufigkeit **überraschende Effekte**:

```
m ← newEmptyMVar
forkIO (do {s← takeMVar m; putStrLn s})
threadDelay (100000)
putMVar m (error "FOO!")
```

- ▶ In welchem Thread wird die Ausnahme geworfen?
- ▶ Wo kann sie gefangen werden?
- ▶ Deshalb haben in Scala die Future-Callbacks den Typ:

```
trait Future[+T] { def onComplete(f: Try[T] ⇒ Unit): Unit
```

# Explizite Fehlerbehandlung mit Try

- ▶ Die Signatur einer Methode verrät nichts über mögliche Fehler:

```
if (n ≤ 0) this
else if (battery > 0) {
```

- ▶ Try[T] macht Fehler explizit (**Materialisierung** oder Reifikation):

```
sealed abstract class Try[+T] {
  def flatMap[U](f: T ⇒ Try[U]): Try[U] = this match {
    case Success(x) ⇒
      try f(x) catch { case NonFatal(ex) ⇒ Failure(ex) }
    case fail: Failure ⇒ fail }
}
```

```
case class Success[T](x: T) extends Try[T]
case class Failure(ex: Throwable) extends Try[Nothing]
```

- ▶ Ist Try eine Monade?

# Explizite Fehlerbehandlung mit Try

- ▶ Die Signatur einer Methode verrät nichts über mögliche Fehler:

```
if (n ≤ 0) this
else if (battery > 0) {
```

- ▶ Try[T] macht Fehler explizit (**Materialisierung** oder Reifikation):

```
sealed abstract class Try[+T] {
  def flatMap[U](f: T ⇒ Try[U]): Try[U] = this match {
    case Success(x) ⇒
      try f(x) catch { case NonFatal(ex) ⇒ Failure(ex) }
    case fail: Failure ⇒ fail }

```

```
case class Success[T](x: T) extends Try[T]
case class Failure(ex: Throwable) extends Try[Nothing]
```

- ▶ Ist Try eine Monade? Nein, Try(e) flatMap f ≠ f e

# Zusammenfassung

- ▶ **Nebenläufigkeit in Scala** basiert auf der JVM:
  - ▶ Relativ schwergewichtige Threads, Monitore (`synchronized`)
- ▶ **Nebenläufigkeit in Haskell**: Concurrent Haskell
  - ▶ Leichtgewichtige Threads, `MVar`
- ▶ **Futures**: Synchronisation über veränderlichen Zustand
  - ▶ In Haskell als `MVar` mit Aktion (`IO`)
  - ▶ In Scala als `Future` mit Callbacks
- ▶ Explizite Fehler bei Nebenläufigkeit **unverzichtbar**
- ▶ Morgen: Scala Collections, nächste VL: das Aktorenmodell

Reaktive Programmierung  
Vorlesung 4 vom 25.04.19  
The Scala Collection Library

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Heute: Scala Collections

- ▶ Sind **nicht** in die Sprache eingebaut!
- ▶ Trotzdem komfortabel

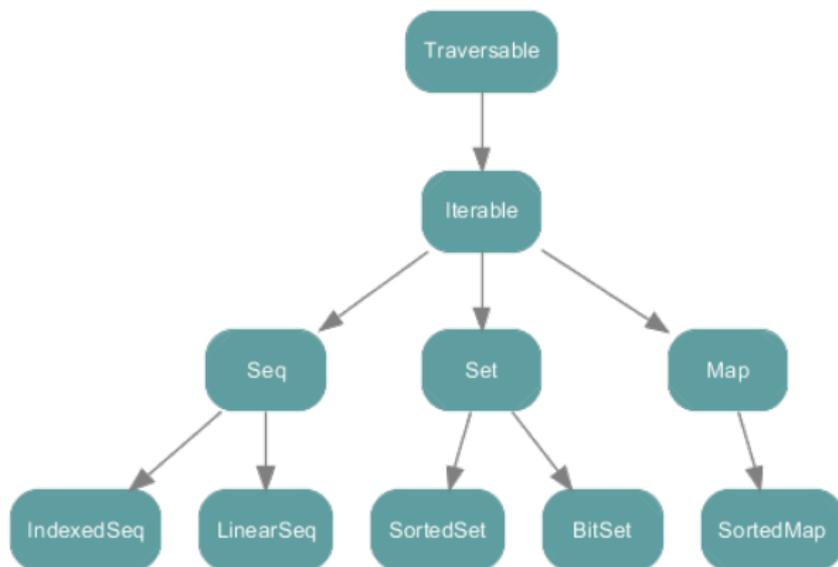
```
val ages = Map("Homer" → 36, "Marge" → 34)
ages("Homer") // 36
```

- ▶ Sehr vielseitig (Immutable, Mutable, Linear, Random Access, Read Once, Lazy, Strict, Sorted, Unsorted, Bounded...)
- ▶ Und sehr generisch

```
val a = Array(1,2,3) ++ List(1,2,3)
a.flatMap(i ⇒ Seq(i, i+1, i+2))
```

# Scala Collections Bücherei

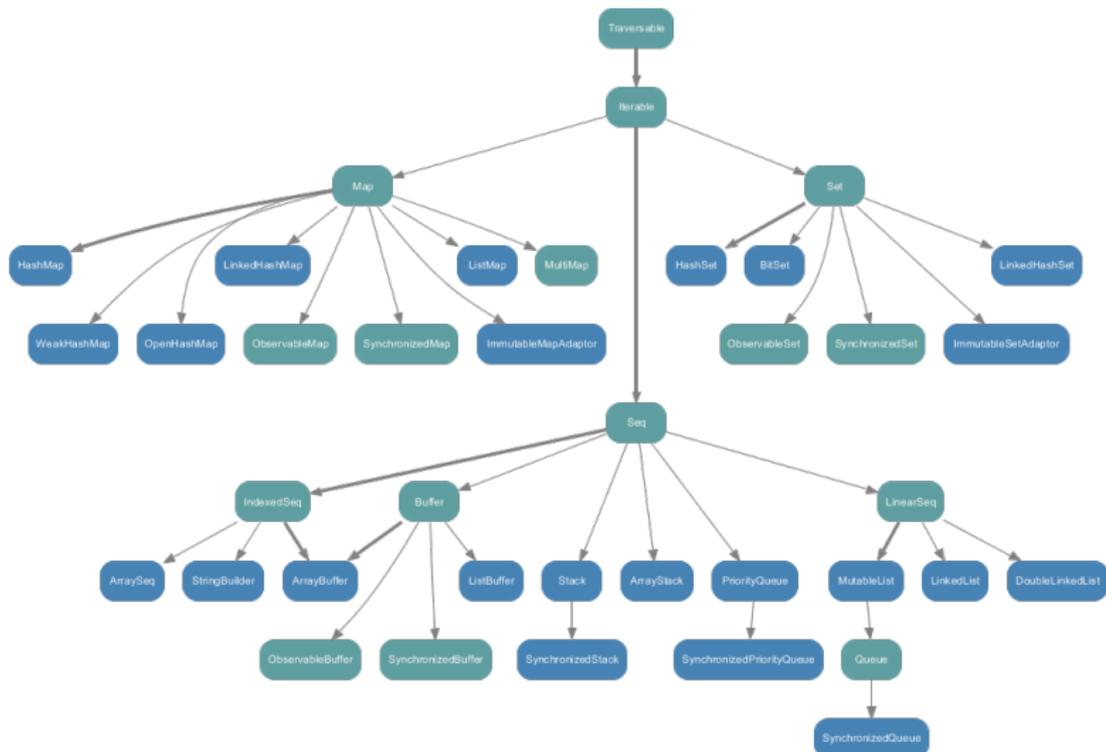
Sehr einheitliche Schnittstellen aber komplexe Bücherei:



# Scala Collections Bücherei - Immutable



# Scala Collections Bücherei - Mutable



# Konstruktoren und Extraktoren

## ► Einheitliche Konstruktoren:

```
Traversable(1, 2, 3)
Iterable("x", "y", "z")
Map("x" → 24, "y" → 25, "z" → 26)
Set(Color.red, Color.green, Color.blue)
SortedSet("hello", "world")
Buffer(x, y, z)
IndexedSeq(1.0, 2.0)
LinearSeq(a, b, c)
...
```

## ► Einheitliche Extraktoren:

```
val Seq(a,b,c) = Seq(1,2,3)
// a = 1; b = 2; c = 3
...
```

# Exkurs: Funktionen in Scala

- ▶ Scala ist rein Objektorientiert.
  - ▶ jeder Wert ist ein Objekt
  - ▶ jede Operation ist ein Methodenaufruf
- ▶ Also ist eine Funktion ein Objekt
- ▶ und ein Funktionsaufruf ein Methodenaufruf.

```
trait Function1[-T1,+R] {  
  def apply(v1: T1): R  
}
```

- ▶ Syntaktischer Zucker: `f(5)` wird zu `f.apply(5)`

## Exkurs: Konstruktoren in Scala

- ▶ Der syntaktische Zucker für Funktionen erlaubt uns Konstruktoren ohne **new** zu definieren:

```
trait Person {  
  def age: Int  
  def name: String  
}  
  
object Person {  
  def apply(a: Int, n: String) = new Person {  
    def age = a  
    def name = n  
  }  
}  
  
val homer = Person(36, "Homer")
```

- ▶ Vgl. Case Classes

## Exkurs: Extraktoren in Scala

- ▶ Das Gegenstück zu `apply` ist `unapply`.
  - ▶ `apply` (Konstruktor): Argumente  $\rightarrow$  Objekt
  - ▶ `unapply` (Extraktor): Objekt  $\rightarrow$  Argumente
- ▶ Wichtig für Pattern Matching (Vgl. Case Classes)

```
object Person {  
  def apply(a: Int, n: String) = <...>  
  def unapply(p: Person): Option[(Int, String)] =  
    Some((p.age, p.name))  
}  
  
homer match {  
  case Person(age, name) if age < 18  $\Rightarrow$  s"hello young  
    $name"  
  case Person(_, name)  $\Rightarrow$  s"hello old $name"  
}  
  
val Person(a, n) = homer
```

## scala.collection.Traversable[+A]

- ▶ Super-trait von allen anderen Collections.
- ▶ Einzige abstrakte Methode:

```
def foreach[U](f: Elem => U): Unit
```

- ▶ Viele wichtige Funktionen sind hier schon definiert:
  - ▶ ++[B](that: Traversable[B]): Traversable[B]
  - ▶ map[B](f: A => B): Traversable[B]
  - ▶ filter(f: A => Boolean): Traversable[A]
  - ▶ foldLeft[B](z: B)(f: (B,A) => B): B
  - ▶ flatMap[B](f: A => Traversable[B]): Traversable[B]
  - ▶ take, drop, exists, head, tail, foreach, size, sum, groupBy, takeWhile ...

## scala.collection.Traversable[+A]

- ▶ Super-trait von allen anderen Collections.
- ▶ Einzige abstrakte Methode:

```
def foreach[U](f: Elem => U): Unit
```

- ▶ Viele wichtige Funktionen sind hier schon definiert:
  - ▶ ++[B](that: Traversable[B]): Traversable[B]
  - ▶ map[B](f: A => B): Traversable[B]
  - ▶ filter(f: A => Boolean): Traversable[A]
  - ▶ foldLeft[B](z: B)(f: (B,A) => B): B
  - ▶ flatMap[B](f: A => Traversable[B]): Traversable[B]
  - ▶ take, drop, exists, head, tail, foreach, size, sum, groupBy, takeWhile ...
- ▶ Problem: So funktionieren die Signaturen nicht!
- ▶ Die folgende Folie ist für Zuschauer unter 16 Jahren nicht geeignet...

# Die wahre Signatur von `map`

```
def map[B,That](f: A ⇒ B)(implicit bf:  
  CanBuildFrom[Traversable[A], B, That]): That
```

# Die wahre Signatur von `map`

```
def map[B,That](f: A ⇒ B)(implicit bf:  
  CanBuildFrom[Traversable[A], B, That]): That
```

Was machen wir damit?

- ▶ Schnell wieder vergessen
- ▶ Aber im Hinterkopf behalten: Die Signaturen in der Dokumentation sind “geschönt”!

## Seq[+A], IndexedSeq[+A], LinearSeq[+A]

- ▶ Haben eine Länge (`length`)
- ▶ Elemente haben feste Positionen (`indexOf`, `indexOfSlice`, ...)
- ▶ Können sortiert werden (`sorted`, `sortedWith`, `sortBy`, ...)
- ▶ Können umgedreht werden (`reverse`, `reverseMap`, ...)
- ▶ Können mit anderen Sequenzen verglichen werden (`startsWith`, ...)
- ▶ Nützliche Subtypen: `List`, `Stream`, `Vector`, `Stack`, `Queue`, `mutable.Buffer`
- ▶ Welche ist die richtige für mich?  
<http://docs.scala-lang.org/overviews/collections/performance-characteristics.html>

## Set [+A]

- ▶ Enthalten keine doppelten Elemente
- ▶ Unterstützen Vereinigungen, Differenzen, Schnittmengen:

```
Set("apple", "strawberry") ++ Set("apple", "peach")  
> Set("apple", "strawberry", "peach")
```

```
Set("apple", "strawberry") — Set("apple", "peach")  
> Set("strawberry")
```

```
Set("apple", "strawberry") & Set("apple", "peach")  
> Set("apple")
```

- ▶ Nützliche Subtypen: SortedSet, BitSet

## Map [K, V]

- ▶ Ist eine Menge von Schlüssel-Wert-Paaren:  
Map[K,V] <: Iterable[(K,V)]
- ▶ Ist eine partielle Funktion von Schlüssel zu Wert:  
Map[K,V] <: PartialFunction[K,V]
- ▶ Werte können "nachgeschlagen" werden:

```
val ages = Map("Homer" → 39, "Marge" → 34)
```

```
ages("Homer")
```

```
> 39
```

```
ages.isDefinedAt "Bart" // ages contains "Bart"
```

```
> false
```

```
ages.get "Marge"
```

```
> Some(34)
```

- ▶ Nützliche Subtypen: mutable.Map

# Array

- ▶ Array sind “special”:
  - ▶ Korrespondieren zu Javas Arrays
  - ▶ Können aber auch **generisch** sein `Array[T]`
  - ▶ Und sind kompatibel zu Sequenzen
- ▶ Problem mit Generizität:

```
def evenElems[T](xs: Vector[T]): Array[T] = {  
  val arr = new Array[T]((xs.length + 1) / 2)  
  for (i ← 0 until xs.length by 2)  
    arr(i / 2) = xs(i)  
  arr }
```

# Array

- ▶ *Type erasure* zur Laufzeit — daher: **Class tag** benötigt

```
def evenElems[T](xs: Vector[T])(implicit m: ClassTag[T]):  
  Array[T] = ...  
def evenElems[T: ClassTag](xs: Vector[T]): Array[T] = ...
```

- ▶ Generische Arrays erzeugen overhead: *“You can expect accesses to generic arrays to be three to four times slower than accesses to primitive or object arrays.”*

# String

- ▶ Scala-Strings sind `java.lang.String`
- ▶ Unterstützen aber alle Sequenz-Operationen
- ▶ Beste aller Welten: effiziente Repräsentation, viele Operationen
  - ▶ Vergleiche Haskell: **type** `String = [Char]` bzw. `ByteString`
- ▶ Wird erreicht durch implizite Konversionen `String` to `WrappedString` und `String` to `StringOps`

# Vergleiche von Collections

- ▶ Collections sind in Mengen, Maps und Sequenzen aufgeteilt.
- ▶ Collections aus verschiedenen Kategorien sind niemals gleich:

```
Set(1,2,3) == List(1,2,3) // false
```

- ▶ Mengen und Maps sind gleich wenn sie die selben Elemente enthalten:

```
TreeSet(3,2,1) == HashSet(2,1,3) // true
```

- ▶ Sequenzen sind gleich wenn sie die selben Elemente in der selben Reihenfolge enthalten:

```
List(1,2,3) == Stream(1,2,3) // true
```

# Scala Collections by Example - Part I

- ▶ Problem: Namen der erwachsenen Personen in einer Liste

```
case class Person(name: String, age: Int)
val persons = List(Person("Homer",39), Person("Marge",34),
                    Person("Bart",10), Person("Lisa",8),
                    Person("Maggie",1), Person("Abe",80))
```

# Scala Collections by Example - Part I

- ▶ Problem: Namen der erwachsenen Personen in einer Liste

```
case class Person(name: String, age: Int)
val persons = List(Person("Homer",39), Person("Marge",34),
                   Person("Bart",10), Person("Lisa",8),
                   Person("Maggie",1), Person("Abe",80))
```

- ▶ Lösung:

```
val adults = persons.filter(_.age ≥ 18).map(_.name)
> List("Homer", "Marge", "Abe")
```

## Scala Collections by Example - Part II

- ▶ Problem: Fibonacci Zahlen so elegant wie in Haskell?

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

## Scala Collections by Example - Part II

- ▶ Problem: Fibonacci Zahlen so elegant wie in Haskell?

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- ▶ Lösung:

```
val fibs: Stream[BigInt] =  
  BigInt(0) #:: BigInt(1) #:: fibs.zip(fibs.tail).map(  
    n => n._1 + n._2)
```

```
fibs.take(10).foreach(println)
```

```
> 0
```

```
> 1
```

```
> ...
```

```
> 21
```

```
> 34
```

## Option [+A]

- ▶ Haben **maximal** 1 Element

```
sealed trait Option[+A]  
case object None extends Option[Nothing]  
case class Some(get: A) extends Option[A]
```

- ▶ Entsprechen Maybe in Haskell
- ▶ Sollten dort benutzt werden wo in Java null im Spiel ist

```
def get(elem: String) = elem match {  
  case "a" ⇒ Some(1)  
  case "b" ⇒ Some(2)  
  case _ ⇒ None  
}
```

- ▶ Hilfreich dabei:

```
Option("Hallo") // Some("Hallo")  
Option(null) // None
```

## Option [+A]

- ▶ An vielen Stellen in der Standardbücherei gibt es die Auswahl:

```
val ages = Map("Homer" → 39, "Marge" → 34)
```

```
ages("Bart") // NoSuchElementException
```

```
ages.get("Bart") // None
```

- ▶ Nützliche Operationen auf Option

```
val x: Option[Int] = ???
```

```
x.getOrElse 0
```

```
x.foldLeft("Test")(_.toString)
```

```
x.exists(_ == 4)
```

```
...
```

# Ranges

- ▶ Repräsentieren Zahlensequenzen

```
class Range(start: Int, end: Int, step: Int)
class Inclusive(start: Int, end: Int, step: Int) extends
  Range(start, end + 1, step)
```

- ▶ Int ist “gepimpt” (RichInt):

```
1 to 10 // new Inclusive(1,10,1)
1 to (10,5) // new Inclusive(1,10,5)
1 until 10 // new Range(1,10)
```

- ▶ Werte sind berechnet und nicht gespeichert
- ▶ Keine “echten” Collections
- ▶ Dienen zum effizienten Durchlaufen von Zahlensequenzen:

```
(1 to 10).foreach(println)
```

# For Comprehensions

- ▶ In Scala ist `for` nur syntaktischer Zucker

```
for (i ← 1 to 10) println(i)  
⇒ (1 to 10).foreach(i ⇒ println(i))
```

```
for (i ← 1 to 10) yield i * 2  
⇒ (1 to 10).map(i ⇒ i * 2)
```

```
for (i ← 1 to 10 if i > 5) yield i * 2  
⇒ (1 to 10).filter(i ⇒ i > 5).map(i ⇒ i * 2)
```

```
for (x ← 1 to 10, y ← 1 to 10) yield (x,y)  
⇒ (1 to 10).flatMap(x ⇒ (1 to 10).map(y ⇒ (x,y)))
```

- ▶ Funktioniert mit allen Typen die die nötige Untermenge der Funktionen (`foreach`, `map`, `flatMap`, `withFilter`) implementieren.

## Scala Collections by Example - Part III

- ▶ Problem: Wörter in allen Zeilen in allen Dateien in einem Verzeichnis durchsuchen.

```
def files(path: String): List[File]
def lines(file: File): List[String]
def words(line: String): List[String]

def find(path: String, p: String ⇒ Boolean) = ???
```

## Scala Collections by Example - Part III

- ▶ Problem: Wörter in allen Zeilen in allen Dateien in einem Verzeichnis durchsuchen.

```
def files(path: String): List[File]
def lines(file: File): List[String]
def words(line: String): List[String]

def find(path: String, p: String ⇒ Boolean) = ???
```

- ▶ Lösung:

```
def find(path: String, p: String ⇒ Boolean) = for {
  file ← files(path)
  line ← lines(file)
  word ← words(line) if p(word)
} yield word
```

# Zusammenfassung

- ▶ Scala Collections sind ziemlich komplex
- ▶ Dafür sind die Operationen sehr generisch
- ▶ Es gibt keine in die Sprache eingebauten Collections:  
Die Collections in der Standardbücherei könnte man alle selbst implementieren
- ▶ Für fast jeden Anwendungsfall gibt es schon einen passenden Collection Typ
- ▶ `for`-Comprehensions sind in Scala nur syntaktischer Zucker

Reaktive Programmierung  
Vorlesung 5 vom 02.05.19  
Das Aktorenmodell

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Das Aktorenmodell



- ▶ Eingeführt von Carl Hewitt, Peter Bishop und Richard Steiger (1973)
- ▶ Grundlage für nebenläufige Programmiersprachen und Frameworks. (Unter anderem Akka)
- ▶ Theoretisches Berechnungsmodell

# Das Aktorenmodell



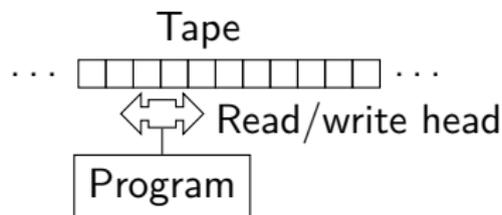
- ▶ Eingeführt von Carl Hewitt, Peter Bishop und Richard Steiger (1973)
- ▶ Grundlage für nebenläufige Programmiersprachen und Frameworks. (Unter anderem Akka)
- ▶ Theoretisches Berechnungsmodell

Warum ein weiteres Berechnungsmodell? Es gibt doch schon die Turingmaschine!

# Die Turingmaschine



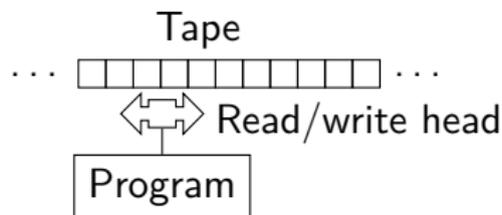
“the behavior of the computer at any moment is determined by the symbols which he [the computer] is observing, and his ‘state of mind’ at that moment”  
— Alan Turing



# Die Turingmaschine



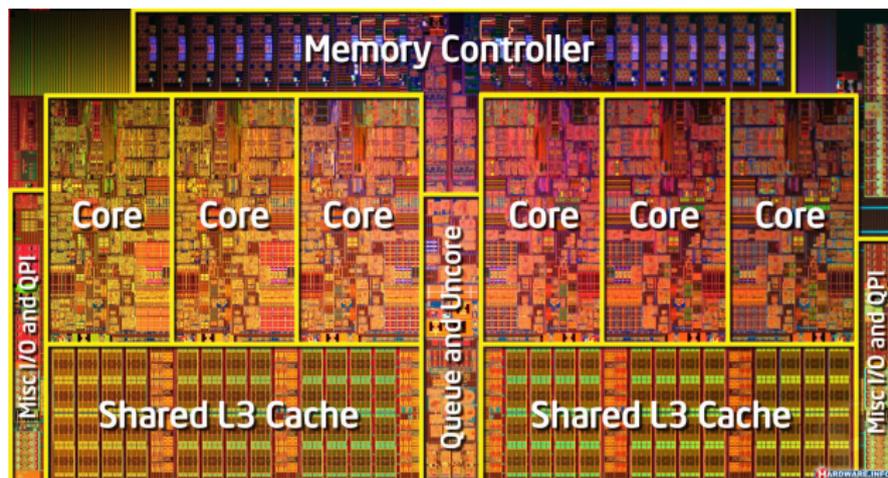
“the behavior of the computer at any moment is determined by the symbols which he [the computer] is observing, and his ‘state of mind’ at that moment”  
— Alan Turing



It is “absolutely impossible that anybody who understands the question [What is computation?] and knows Turing’s definition should decide for a different concept.” — Kurt Gödel



# Die Realität



- ▶  $3\text{GHz} = 3'000'000'000\text{Hz} \implies \text{Ein Takt} = 3,333 * 10^{-10}\text{s}$
- ▶  $c = 299'792'458 \frac{\text{m}}{\text{s}}$
- ▶ Maximaler Weg in einem Takt  $< 0,1\text{m}$  (Physikalische Grenze)

# Synchronisation



- ▶ Während auf ein Signal gewartet wird, kann nichts anderes gemacht werden
- ▶ Synchronisation ist nur in engen Grenzen praktikabel! (Flaschenhals)

# Der Arbiter

- ▶ Die Lösung: **Asynchrone Arbiter**



- ▶ Wenn  $I_1$  und  $I_2$  fast ( $\approx 2fs$ ) gleichzeitig aktiviert werden, wird entweder  $O_1$  oder  $O_2$  aktiviert.
- ▶ Physikalisch unmöglich in konstanter Zeit. Aber Wahrscheinlichkeit, dass keine Entscheidung getroffen wird nimmt mit der Zeit exponentiell ab.
- ▶ Idealer Arbiter entscheidet in  $O(\ln(1/\epsilon))$
- ▶ kommen in modernen Computern überall vor

# Unbounded Nondeterminism

- ▶ In Systemen mit Arbitern kann das Ergebnis einer Berechnung **unbegrenzt** verzögert werden,
- ▶ wird aber **garantiert** zurückgegeben.
- ▶ Nicht modellierbar mit (nichtdeterministischen) Turingmaschinen.

## Beispiel

Ein Arbitr entscheidet in einer Schleife, ob ein Zähler inkrementiert wird oder der Wert des Zählers als Ergebnis zurückgegeben wird.

# Das Aktorenmodell

Quantum mechanics indicates that the notion of a universal description of the state of the world, shared by all observers, is a concept which is physically untenable, on experimental grounds. — Carlo Rovelli

- ▶ Frei nach der relationalen Quantenphysik

## Drei Grundlagen

- ▶ Verarbeitung
  - ▶ Speicher
  - ▶ **Kommunikation**
- 
- ▶ Die (nichtdeterministische) Turingmaschine ist ein Spezialfall des Aktorenmodells
  - ▶ Ein **Aktorensystem** besteht aus **Aktoren** (Alles ist ein Aktor!)

# Aktoren

- ▶ Ein Aktor verarbeitet Nachrichten

Während ein Aktor eine Nachricht verarbeitet kann er

- ▶ neue Aktoren erzeugen
  - ▶ Nachrichten an bekannte Aktor-Referenzen versenden
  - ▶ festlegen wie die nächste Nachricht verarbeitet werden soll
- 
- ▶ Aktor  $\neq$  ( Thread | Task | Channel | ... )

# Aktoren

- ▶ Ein Aktor verarbeitet Nachrichten

Während ein Aktor eine Nachricht verarbeitet kann er

- ▶ neue Aktoren erzeugen
  - ▶ Nachrichten an bekannte Aktor-Referenzen versenden
  - ▶ festlegen wie die nächste Nachricht verarbeitet werden soll
- 
- ▶ Aktor  $\neq$  ( Thread | Task | Channel | ... )

Ein Aktor kann (darf) **nicht**

- ▶ auf globalen Zustand zugreifen
- ▶ veränderliche Nachrichten versenden
- ▶ irgendetwas tun während er keine Nachricht verarbeitet

# Aktoren (Technisch)

- ▶  $\text{Aktor} \approx \text{Schleife über unendliche Nachrichtenliste} + \text{Zustand}$   
(Verhalten)

## Aktoren (Technisch)

- ▶  $\text{Aktor} \approx \text{Schleife über unendliche Nachrichtenliste} + \text{Zustand}$   
(Verhalten)
- ▶  $\text{Behavior} : (\text{Msg}, \text{State}) \rightarrow \text{IO State}$
- ▶ oder  $\text{Behavior} : \text{Msg} \rightarrow \text{IO Behavior}$

# Aktoren (Technisch)

- ▶  $\text{Aktor} \approx \text{Schleife über unendliche Nachrichtenliste} + \text{Zustand}$   
(Verhalten)
- ▶  $\text{Behavior} : (\text{Msg}, \text{State}) \rightarrow \text{IO State}$
- ▶ oder  $\text{Behavior} : \text{Msg} \rightarrow \text{IO Behavior}$
- ▶ Verhalten hat Seiteneffekte (IO):
  - ▶ Nachrichtenversand
  - ▶ Erstellen von Aktoren
  - ▶ Ausnahmen

# Verhalten vs. Protokoll

## Verhalten

Das Verhalten eines Aktors ist eine seiteneffektbehaftete Funktion

*Behavior* :  $Msg \rightarrow IO\ Behavior$

## Protokoll

Das Protokoll eines Aktors beschreibt, wie ein Aktor auf Nachrichten reagiert und resultiert implizit aus dem Verhalten.

► Beispiel:

```
case (Ping, a) =>
  println("Hello")
  counter += 1
  a ! Pong
```

$\square(a(Ping, b) \rightarrow \diamond b(Pong))$

# Kommunikation

- ▶ Nachrichten sind **unveränderliche** Daten, **reine** Funktionen oder **Futures**
- ▶ Die Zustellung von Nachrichten passiert höchstens einmal (Best-effort)
- ▶ Wenn z.B. die Netzwerkverbindung abbricht, wird gewartet, bis der Versand wieder möglich ist
- ▶ Wenn aber z.B. der Computer direkt nach Versand der Nachricht explodiert (oder der Speicher voll läuft), kommt die Nachricht möglicherweise niemals an
- ▶ Über den Zeitpunkt des Empfangs kann keine Aussage getroffen werden (Unbounded indeterminacy)
- ▶ Über die Reihenfolge der Empfangenen Nachrichten wird im Aktorenmodell keine Aussage gemacht (In vielen Implementierungen allerdings schon)
- ▶ Nachrichtenversand  $\neq$  ( Queue | Lock | Channel | ... )

# Kommunikation (Technisch)

- ▶ Der Versand einer Nachricht  $M$  an Aktor  $A$  bewirkt, dass zu **genau einem** Zeitpunkt in der Zukunft, das Verhalten  $B$  von  $A$  mit  $M$  als Nachricht ausgeführt wird.
- ▶ Über den Zustand  $S$  von  $A$  zum Zeitpunkt der Verarbeitung können wir begrenzte Aussagen treffen:
  - ▶ z.B. Aktor-Invariante: Vor und nach jedem Nachrichteneingang gilt  $P(S)$
- ▶ Besser: Protokoll
  - ▶ z.B. auf Nachrichten des Typs  $T$  reagiert  $A$  immer mit Nachrichten des Typs  $U$

# Identifikation

- ▶ Akteure werden über **Identitäten** angesprochen

## Akteure kennen Identitäten

- ▶ aus einer empfangenen Nachricht
  - ▶ aus der Vergangenheit (Zustand)
  - ▶ von Akteuren die sie selbst erzeugen
- 
- ▶ Nachrichten können weitergeleitet werden
  - ▶ Eine Identität kann zu mehreren Akteuren gehören, die der Halter der Referenz äußerlich nicht unterscheiden kann
  - ▶ Eindeutige Identifikation bei verteilten Systemen nur durch Authentisierungsverfahren möglich

# Location Transparency

- ▶ Eine Aktoridentität kann irgendwo hin zeigen
  - ▶ Gleicher Thread
  - ▶ Gleicher Prozess
  - ▶ Gleicher CPU Kern
  - ▶ Gleiche CPU
  - ▶ Gleicher Rechner
  - ▶ Gleiches Rechenzentrum
  - ▶ Gleicher Ort
  - ▶ Gleiches Land
  - ▶ Gleicher Kontinent
  - ▶ Gleicher Planet
  - ▶ ...

# Sicherheit in Aktorsystemen

- ▶ Das Aktorenmodell spezifiziert nicht wie eine Aktoridentität repräsentiert wird
- ▶ In der Praxis müssen Identitäten aber **serialisierbar** sein
- ▶ Serialisierbare Identitäten sind auch **synthetisierbar**
- ▶ Bei Verteilten Systemen ein potentiellles Sicherheitsproblem
- ▶ Viele Implementierungen stellen **Authentisierungsverfahren** und **verschlüsselte** Kommunikation zur Verfügung.

# Inkonsistenz in Aktorsystemen

- ▶ Ein Aktorsystem hat **keinen** globalen Zustand (Pluralismus)
- ▶ Informationen in Aktoren sind global betrachtet **redundant**, **inkonsistent** oder **lokal**
- ▶ Konsistenz  $\neq$  Korrektheit
- ▶ Wo nötig müssen duplizierte Informationen konvergieren, wenn "*längere Zeit*" keine Ereignisse auftreten (**Eventual consistency**)

# Eventual Consistency

## Definition

In einem verteilten System ist ein repliziertes Datum **schließlich Konsistent**, wenn über einen längeren Zeitraum keine Fehler auftreten und das Datum nirgendwo verändert wird

- ▶ Konvergente (oder Konfliktfreie) Replizierte Datentypen (CRDTs) garantieren diese Eigenschaft:
  - ▶  $(\mathbb{N}, \{+\})$  oder  $(\mathbb{Z}, \{+, -\})$
  - ▶ Grow-Only-Sets
- ▶ Strategien auf komplexeren Datentypen:
  - ▶ Operational Transformation
  - ▶ Differential Synchronization
- ▶ dazu später mehr ...

# Fehlerbehandlung in Aktorsystemen

- ▶ Wenn das Verhalten eines Aktors eine unbehandelte Ausnahme wirft:
  - ▶ Verhalten bricht ab
  - ▶ Aktor existiert nicht mehr
- ▶ Lösung: Wenn das Verhalten eine Ausnahme nicht behandelt, wird sie an einen überwachenden Aktor (**Supervisor**) weitergeleitet (**Eskalation**):
  - ▶ Gleiches Verhalten wird wiederbelebt
  - ▶ oder neuer Aktor mit gleichem Protokoll kriegt Identität übertragen
  - ▶ oder Berechnung ist fehlgeschlagen

# "Let it Crash!" (Nach Joe Armstrong)

- ▶ Unbegrenzter Nichtdeterminismus ist statisch kaum analysierbar
- ▶ **Unschärfe** beim Testen von verteilten Systemen
- ▶ Selbst wenn ein Programm fehlerfrei ist kann Hardware ausfallen
- ▶ Je verteilter ein System umso wahrscheinlicher geht etwas schief
- ▶ Deswegen:
  - ▶ Offensives Programmieren
  - ▶ Statt Fehler zu vermeiden, Fehler behandeln!
  - ▶ Teile des Programms kontrolliert abstürzen lassen und bei Bedarf neu starten



# Das Aktorenmodell in der Praxis

- ▶ Erlang (Aktor-Sprache)
  - ▶ Ericsson - GPRS, UMTS, LTE
  - ▶ T-Mobile - SMS
  - ▶ WhatsApp (2 Millionen Nutzer pro Server)
  - ▶ Facebook Chat (100 Millionen simultane Nutzer)
  - ▶ Amazon SimpleDB
  - ▶ ...
- ▶ Akka (Scala Framework)
  - ▶ ca. 50 Millionen Nachrichten / Sekunde
  - ▶ ca. 2,5 Millionen Aktoren / GB Heap
  - ▶ Amazon, Cisco, Blizzard, LinkedIn, BBC, The Guardian, Atos, The Huffington Post, Ebay, Groupon, Credit Suisse, Gilt, KK, ...

# Zusammenfassung

- ▶ Das Aktorenmodell beschreibt **Aktorensysteme**
- ▶ Aktorensysteme bestehen aus **Aktoren**
- ▶ Aktoren kommunizieren über **Nachrichten**
- ▶ Aktoren können überall liegen (**Location Transparency**)
- ▶ Inkonsistenzen können nicht vermieden werden: **Let it crash!**
- ▶ Vorteile: Einfaches Modell; keine Race Conditions; Sehr schnell in Verteilten Systemen
- ▶ Nachteile: Informationen müssen dupliziert werden; Keine vollständige Implementierung

Reaktive Programmierung  
Vorlesung 6 vom 27.04.17  
ScalaTest and ScalaCheck

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Was ist eigentlich Testen?

Myers, 1979

Testing is the process of executing a program or system with the intent of finding errors.

- ▶ Hier: testen ist **selektive, kontrollierte** Programmausführung.
- ▶ **Ziel** des Testens ist es immer, Fehler zu finden wie:
  - ▶ Diskrepanz zwischen Spezifikation und Implementation
  - ▶ strukturelle Fehler, die zu einem fehlerhaften Verhalten führen (Programmabbruch, Ausnahmen, etc)

EW. Dijkstra, 1972

Program testing can be used to show the presence of bugs, but never to show their absence.

# Testmethoden

- ▶ Statisch vs. dynamisch:
  - ▶ **Statische** Tests **analysieren** den Quellcode ohne ihn auszuführen (**statische Programmanalyse**)
  - ▶ **Dynamische** Tests führen das Programm unter **kontrollierten** Bedingungen aus, und prüfen das Ergebnis gegen eine gegebene Spezifikation.
- ▶ Zentrale Frage: wo kommen die **Testfälle** her?
  - ▶ **Black-box**: Struktur des s.u.t. (hier: Quellcode) unbekannt, Testfälle werden aus der Spezifikation generiert;
  - ▶ **Grey-box**: Teile der Struktur des s.u.t. ist bekannt (z.B. Modulstruktur)
  - ▶ **White-box**: Struktur des s.u.t. ist offen, Testfälle werden aus dem Quellcode abgeleitet

# Spezialfall des Black-Box-Tests: Monte-Carlo Tests

- ▶ Bei Monte-Carlo oder Zufallstests werden **zufällige** Eingabewerte generiert, und das Ergebnis gegen eine Spezifikation geprüft.
- ▶ Dies erfordert **ausführbare** Spezifikationen.
- ▶ Wichtig ist die **Verteilung** der Eingabewerte.
  - ▶ Gleichverteilt über erwartete Eingaben, Grenzfälle beachten.
- ▶ Funktioniert gut mit **high-level-Spachen** (Java, Scala, Haskell)
  - ▶ Datentypen repräsentieren Informationen auf **abstrakter** Ebene
  - ▶ Eigenschaft gut **spezifizierbar**
  - ▶ Beispiel: Listen, Listenumkehr in C, Java, Scala
- ▶ **Zentrale Fragen:**
  - ▶ Wie können wir **ausführbare Eigenschaften** formulieren?
  - ▶ Wie **Verteilung** der Zufallswerte steuern?

# ScalaTest

- ▶ Test Framework für Scala

```
import org.scalatest.FlatSpec

class StringSpec extends FlatSpec {
  "A String" should "reverse" in {
    "Hello".reverse should be ("olleH")
  }

  it should "return the correct length" in {
    "Hello".length should be (5)
  }
}
```

# ScalaTest Assertions 1

- ▶ ScalaTest Assertions sind Makros:

```
import org.scalatest.Assertions._  
val left = 2  
val right = 1  
assert(left == right)
```

- ▶ Schlägt fehl mit "2 did not equal 1"
- ▶ Alternativ:

```
val a = 5  
val b = 2  
assertResult(2) {  
  a - b  
}
```

- ▶ Schlägt fehl mit "Expected 2, but got 3"

## ScalaTest Assertions 2

- ▶ Fehler manuell werfen:

```
fail("I've got a bad feeling about this")
```

- ▶ Erwartete Exceptions:

```
val s = "hi"  
val e = intercept[IndexOutOfBoundsException] {  
  s.charAt(-1)  
}
```

- ▶ Assumptions

```
assume(database.isAvailable)
```

# ScalaTest Matchers

- ▶ Gleichheit überprüfen:

```
result should equal (3)
result should be (3)
result shouldBe 3
result shouldEqual 3
```

- ▶ Länge prüfen:

```
result should have length 3
result should have size 3
```

- ▶ Und so weiter...

```
text should startWith ("Hello")
result should be a [List[Int]]
list should contain noneOf (3,4,5)
```

- ▶ Siehe [http://www.scalatest.org/user\\_guide/using\\_matchers](http://www.scalatest.org/user_guide/using_matchers)

# ScalaTest Styles

- ▶ ScalaTest hat viele verschiedene Styles, die über Traits eingemischt werden können
- ▶ Beispiel: FunSpec (Ähnlich wie RSpec)

```
class SetSpec extends FunSpec {  
  describe("A Set") {  
    describe("when empty") {  
      it("should have size 0") {  
        assert(Set.empty.size == 0)  
      }  
  
      it("should produce NoSuchElementException when head  
        is invoked") {  
        intercept[NoSuchElementException] {  
          Set.empty.head  
        }  
      }  
    }  
  }  
}
```

- ▶ Übersicht unter

# Blackbox Test

- ▶ Überprüfen eines Programms oder einer Funktion ohne deren Implementierung zu nutzen:

```
def primeFactors(n: Int): List[Int] = ???
```

- ▶ z.B.

```
"primeFactors" should "work for 360" in {  
  primeFactors(360) should contain theSameElementsAs  
    List(2,2,2,3,3,5)  
}
```

- ▶ Was ist mit allen anderen Eingaben?

# Property based Testing

- ▶ Überprüfen von **Eigenschaften** (Properties) eines Programms / einer Funktion:

```
def primeFactors(n: Int): List[Int] = ???
```

- ▶ Wir würden gerne so was schreiben:

```
forall x ≥ 1 → primeFactors(x).product = x  
          && primeFactors(x).forall(isPrime)
```

# Property based Testing

- ▶ Überprüfen von **Eigenschaften** (Properties) eines Programms / einer Funktion:

```
def primeFactors(n: Int): List[Int] = ???
```

- ▶ Wir würden gerne so was schreiben:

```
forall x ≥ 1 → primeFactors(x).product = x  
          && primeFactors(x).forall(isPrime)
```

- ▶ Aber wo kommen die Eingaben her?

# Testen mit Zufallswerten

▶ `def primeFactors(n: Int): List[Int] = ???`

▶ Zufallszahlen sind doch einfach!

```
"primeFactors" should "work for many numbers" in {  
  (1 to 1000) foreach { _ =>  
    val x = Math.max(1, Random.nextInt.abs)  
    assert(primeFactors(x).product == (x))  
    assert(primeFactors(x).forall(isPrime))  
  }  
}
```

# Testen mit Zufallswerten

▶ `def primeFactors(n: Int): List[Int] = ???`

▶ Zufallszahlen sind doch einfach!

```
"primeFactors" should "work for many numbers" in {  
  (1 to 1000) foreach { _ =>  
    val x = Math.max(1, Random.nextInt.abs)  
    assert(primeFactors(x).product == (x))  
    assert(primeFactors(x).forall(isPrime))  
  }  
}
```

▶ Was ist mit dieser Funktion?

```
def sum(list: List[Int]): Int = ???
```

# ScalaCheck

- ▶ ScalaCheck nutzt Generatoren um Testwerte für Properties zu generieren

```
forall { (list: List[Int]) =>
  sum(list) == list.foldLeft(0)(_ + _)
}
```

- ▶ Generatoren werden über implicits aufgelöst
- ▶ Typklasse Arbitrary für viele Typen vordefiniert:

```
abstract class Arbitrary[T] {
  val arbitrary: Gen[T]
}
```

# Zufallsgeneratoren

- ▶ Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }  
  
object Generator {  
  def apply[T](f: => T) = new Generator[T] {  
    def generate = f }  
}
```

# Zufallsgeneratoren

- ▶ Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }  
  
object Generator {  
  def apply[T](f: => T) = new Generator[T] {  
    def generate = f }  
}
```

- ▶ `val integers = Generator(Random.nextInt)`

# Zufallsgeneratoren

- ▶ Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }  
  
object Generator {  
  def apply[T](f: => T) = new Generator[T] {  
    def generate = f }  
}
```

- ▶ **val** integers = Generator(Random.nextInt)

- ▶ **val** booleans = Generator(integers.generate > 0)

# Zufallsgeneratoren

- ▶ Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }  
  
object Generator {  
  def apply[T](f: => T) = new Generator[T] {  
    def generate = f }  
}
```

- ▶ `val integers = Generator(Random.nextInt)`
- ▶ `val booleans = Generator(integers.generate > 0)`
- ▶ `val pairs =  
 Generator((integers.generate, integers.generate))`

# Zufallsgeneratoren Kombinieren

- ▶ Ein generischer, **kombinierbarer** Zufallsgenerator:

```
trait Generator[+T] { self =>
  def generate: T
  def map[U](f: T => U) = new Generator[U] {
    def generate = f(self.generate)
  }
  def flatMap[U](f: T => Generator[U]) = new Generator[U] {
    def generate = f(self.generate).generate
  }
}
```

# Einfache Zufallsgeneratoren

- ▶ Einelementige Wertemenge:

```
def single[T](value: T) = Generator(value)
```

- ▶ Eingeschränkter Wertebereich:

```
def choose(lo: Int, hi: Int) =  
  integers.map(x ⇒ lo + x % (hi - lo))
```

- ▶ Aufzählbare Wertemenge:

```
def oneOf[T](xs: T*): Generator[T] =  
  choose(0, xs.length).map(xs)
```

## Beispiel: Listen Generieren

- ▶ Listen haben zwei Konstruktoren: Nil und :::

```
def lists: Generator[List[Int]] = for {  
  isEmpty ← booleans  
  list ← if (isEmpty) emptyLists else nonEmptyLists  
}
```

## Beispiel: Listen Generieren

- ▶ Listen haben zwei Konstruktoren: Nil und :::

```
def lists: Generator[List[Int]] = for {  
  isEmpty ← booleans  
  list ← if (isEmpty) emptyLists else nonEmptyLists  
}
```

- ▶ Die Menge der leeren Listen enthält genau ein Element:

```
def emptyLists = single(Nil)
```

## Beispiel: Listen Generieren

- ▶ Listen haben zwei Konstruktoren: Nil und :::

```
def lists: Generator[List[Int]] = for {  
  isEmpty ← booleans  
  list ← if (isEmpty) emptyLists else nonEmptyLists  
}
```

- ▶ Die Menge der leeren Listen enthält genau ein Element:

```
def emptyLists = single(Nil)
```

- ▶ Nicht-leere Listen bestehen aus einem Element und einer Liste:

```
def nonEmptyLists = for {  
  head ← integers  
  tail ← lists  
} yield head :: tail
```

# ScalaCheck

- ▶ ScalaCheck nutzt Generatoren um Testwerte für Properties zu generieren

```
forall { (list: List[Int]) =>
  sum(list) == list.foldLeft(0)(_ + _)
}
```

- ▶ Generatoren werden über implicits aufgelöst
- ▶ Typklasse Arbitrary für viele Typen vordefiniert:

```
abstract class Arbitrary[T] {
  val arbitrary: Gen[T]
}
```

# Kombinatoren in ScalaCheck

```
object Gen {  
  def choose[T](min: T, max: T)(implicit c: Choose[T]):  
    Gen[T]  
  def oneOf[T](xs: Seq[T]): Gen[T]  
  def sized[T](f: Int  $\Rightarrow$  Gen[T]): Gen[T]  
  def someOf[T](gs: Gen[T]*); Gen[Seq[T]]  
  def option[T](g: Gen[T]): Gen[Option[T]]  
  ...  
}
```

```
trait Gen[+T] {  
  def map[U](f: T  $\Rightarrow$  U): Gen[U]  
  def flatMap[U](f: T  $\Rightarrow$  Gen[U]): Gen[U]  
  def filter(f: T  $\Rightarrow$  Boolean): Gen[T]  
  def suchThat(f: T  $\Rightarrow$  Boolean): Gen[T]  
  def label(l: String): Gen[T]  
  def |(that: Gen[T]): Gen[T]
```

## Wertemenge einschränken

- ▶ Problem: Vorbedingungen können dazu führen, dass nur wenige Werte verwendet werden können:

```
val prop = forAll { (l1: List[Int], l2: List[Int]) =>
  l1.length == l2.length => l1.zip(l2).unzip() == (l1, l2)
}
```

```
scala> prop.check
```

```
Gave up after only 4 passed tests. 500 tests were
discarded.
```

- ▶ Besser:

```
forAll(myListPairGenerator) { (l1, l2) =>
  l1.zip(l2).unzip() == (l1, l2)
}
```

# Kombinatoren für Properties

- ▶ Properties können miteinander kombiniert werden:

```
val p1 = forAll (...)  
val p2 = forAll (...)  
val p3 = p1 && p2  
val p4 = p1 || p2  
val p5 = p1 == p2  
val p6 = all(p1, p2)  
val p7 = atLeastOne(p1, p2)
```

# ScalaCheck in ScalaTest

- ▶ Der Trait `Checkers` erlaubt es, ScalaCheck in beliebigen ScalaTest Suiten zu verwenden:

```
class IntListSpec extends FlatSpec with PropertyChecks {  
  "Any list of integers" should "return its correct sum"  
    in {  
      forall { (x: List[Int]) => x.sum == x.foldLeft(0)(_ +  
        _) }  
    }  
}
```

# Zusammenfassung

- ▶ ScalaTest: DSL für Tests in Scala
  - ▶ Verschiedene Test-Stile durch verschiedene Traits
  - ▶ Matchers um Assertions zu formulieren
- ▶ ScalaCheck: Property-based testing
  - ▶ `Gen[+T]` um Zufallswerte zu generieren
  - ▶ Generatoren sind ein monadischer Datentyp
  - ▶ Typklasse `Arbitrary[+T]` stellt generatoren implizit zur Verfügung
- ▶ Nächstes mal endlich Nebenläufigkeit: Futures und Promises

Reaktive Programmierung  
Vorlesung 7 vom 08.05.19  
Actors in Akka

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Aktoren in Scala

- ▶ Eine kurze Geschichte von Akka:
  - ▶ 2006: Aktoren in der Scala Standardbücherei (Philipp Haller, `scala.actors`)
  - ▶ 2010: Akka 0.5 wird veröffentlicht (Jonas Bonér)
  - ▶ 2012: Scala 2.10 erscheint ohne `scala.actors` und Akka wird Teil der Typesafe Platform
- ▶ Auf Akka aufbauend:
  - ▶ Apache Spark
  - ▶ Play! Framework
  - ▶ Akka HTTP (Früher Spray Framework)

# Akka

- ▶ Akka ist ein Framework für Verteilte und Nebenläufige Anwendungen
- ▶ Akka bietet verschiedene Ansätze mit Fokus auf Aktoren
- ▶ Nachrichtengetrieben und asynchron
- ▶ Location Transparency
- ▶ Hierarchische Aktorenstruktur

# Rückblick

- ▶ Aktor Systeme bestehen aus Aktoren
- ▶ Aktoren
  - ▶ haben eine Identität,
  - ▶ haben ein veränderliches Verhalten und
  - ▶ kommunizieren mit anderen Aktoren ausschließlich über unveränderliche Nachrichten.

# Aktoren in Akka

```
trait Actor {  
  type Receive = PartialFunction[Any, Unit]  
  
  def receive: Receive  
  
  implicit val context: ActorContext  
  implicit final val self: ActorRef  
  final def sender: ActorRef  
  
  def preStart()  
  def postStop()  
  def preRestart(reason: Throwable, message: Option[Any])  
  def postRestart(reason: Throwable)  
  
  def supervisorStrategy: SupervisorStrategy  
  def unhandled(message: Any)  
}
```

# Aktoren Erzeugen

```
object Count
```

```
class Counter extends Actor {  
  var count = 0  
  def receive = {  
    case Count => count += 1  
  }  
}
```

# Aktoren Erzeugen

```
object Count
```

```
class Counter extends Actor {  
  var count = 0  
  def receive = {  
    case Count => count += 1  
  }  
}
```

```
val system = ActorSystem("example")
```

# Aktoren Erzeugen

```
object Count
```

```
class Counter extends Actor {  
  var count = 0  
  def receive = {  
    case Count => count += 1  
  }  
}
```

```
val system = ActorSystem("example")
```

Global:

```
val counter = system.actorOf(Props[Counter], "counter")
```

# Aktoren Erzeugen

```
object Count
```

```
class Counter extends Actor {  
  var count = 0  
  def receive = {  
    case Count => count += 1  
  }  
}
```

```
val system = ActorSystem("example")
```

Global:

```
val counter = system.actorOf(Props[Counter], "counter")
```

In Aktoren:

```
val counter = context.actorOf(Props[Counter], "counter")
```

# Nachrichtenversand

```
object Counter { object Count; object Get }
```

```
class Counter extends Actor {  
  var count = 0  
  def receive = {  
    case Counter.Count => count += 1  
    case Counter.Get   => sender ! count  
  }  
}
```

```
val counter = actorOf(Props[Counter], "counter")
```

```
counter ! Count
```

“!” ist asynchron – Der Kontrollfluss wird sofort an den Aufrufer zurückgegeben.

# Eigenschaften der Kommunikation

- ▶ Nachrichten von einer Aktor identität zu einer anderen kommen in der Reihenfolge des Versands an. (Im Aktorenmodell ist die Reihenfolge undefiniert)
- ▶ Abgesehen davon ist die Reihenfolge des Nachrichteneempfangs undefiniert.
- ▶ Nachrichten sollen unveränderlich sein. (Das kann derzeit allerdings nicht überprüft werden)

# Verhalten

```
trait ActorContext {  
  def become(behavior: Receive, discardOld: Boolean = true):  
    Unit  
  def unbecome(): Unit  
  ...  
}
```

```
class Counter extends Actor {  
  def counter(n: Int): Receive = {  
    case Counter.Count ⇒ context.become(counter(n+1))  
    case Counter.Get   ⇒ sender ! n  
  }  
  def receive = counter(0)  
}
```

Nachrichten werden sequenziell abgearbeitet.

# Modellieren mit Aktoren

Aus “Principles of Reactive Programming” (Roland Kuhn):

- ▶ Imagine giving the task to a group of people, dividing it up.
- ▶ Consider the group to be of very large size.
- ▶ Start with how people with different tasks will talk with each other.
- ▶ Consider these “people” to be easily replaceable.
- ▶ Draw a diagram with how the task will be split up, including communication lines.

# Beispiel

# Aktorpfade

- ▶ Alle Aktoren haben eindeutige absolute Pfade. z.B. "akka://exampleSystem/user/countService/counter1"
- ▶ Relative Pfade ergeben sich aus der Position des Aktors in der Hierarchie. z.B. "../counter2"
- ▶ Aktoren können über ihre Pfade angesprochen werden

```
context.actorSelection("../sibling") ! Count  
context.actorSelection("../*") ! Count // wildcard
```

- ▶ ActorSelection  $\neq$  ActorRef

# Location Transparency und Akka Remoting

- ▶ Aktoren in anderen Aktorsystemen auf anderen Maschinen können über absolute Pfade angesprochen werden.

```
val remoteCounter = context.actorSelection(  
    "akka.tcp://otherSystem@214.116.23.9:9000/user/counter")  
  
remoteCounter ! Count
```

- ▶ Aktorsysteme können so konfiguriert werden, dass bestimmte Aktoren in einem anderen Aktorsystem erzeugt werden

```
src/resource/application.conf:
```

```
> akka.actor.deployment {  
>   /remoteCounter {  
>     remote = "akka.tcp://otherSystem@127.0.0.1:2552"  
>   }  
> }
```

# Supervision und Fehlerbehandlung in Akka

## ► OneForOneStrategy vs. AllForOneStrategy

```
class RootCounter extends Actor {  
  override def supervisorStrategy =  
    OneForOneStrategy(maxNrOfRetries = 10,  
                      withinTimeRange = 1 minute) {  
      case _: ArithmeticException           ⇒ Resume  
      case _: NullPointerException         ⇒ Restart  
      case _: IllegalArgumentException       ⇒ Stop  
      case _: Exception                     ⇒ Escalate  
    }  
}
```

# Aktorsysteme Testen

- ▶ Um Aktorsysteme zu testen müssen wir eventuell die Regeln brechen:

```
val actorRef = TestActorRef[Counter]
val actor = actorRef.underlyingActor
```

- ▶ Oder: Integrationstests mit TestKit

```
"A counter" must {
  "be able to count to three" in {
    val counter = system.actorOf[Counter]
    counter ! Count
    counter ! Count
    counter ! Count
    counter ! Get
    expectMsg(3)
  }
}
```

## Event-Sourcing (Akka Persistence)

- ▶ Problem: Aktoren sollen Neustarts überleben, oder sogar dynamisch migriert werden.
- ▶ Idee: Anstelle des Zustands, speichern wir alle Ereignisse.

```
class Counter extends PersistentActor {  
  var count = 0  
  def receiveCommand = {  
    case Count =>  
      persist(Count)(_ => count += 1)  
    case Snap => saveSnapshot(count)  
    case Get => sender ! count  
  }  
  def receiveRecover = {  
    case Count => count += 1  
    case SnapshotOffer(_, snapshot: Int) => count = snapshot  
  }  
}
```

## akka-http (ehemals Spray)

- ▶ Akteure sind ein hervorragendes Modell für **Webserver**
- ▶ akka-http ist ein **minimales** HTTP interface für Akka

```
val serverBinding = Http(system).bind(  
  interface = "localhost", port = 80)  
  
...  
  
val requestHandler: HttpRequest => HttpResponse = {  
  case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>  
    HttpResponse(entity = "PONG!")  
  ...  
}
```

- ▶ Vorteil: Vollständig in Scala implementiert, keine Altlasten wie *Jetty*

# Bewertung

- ▶ Vorteile:
  - ▶ Nah am Aktorenmodell (Carl-Hewitt-approved)
  - ▶ keine Race Conditions
  - ▶ Effizient
  - ▶ Stabil und ausgereift
  - ▶ Umfangreiche Konfigurationsmöglichkeiten
- ▶ Nachteile:
  - ▶ Nah am Aktorenmodell  $\Rightarrow$  `receive` ist untypisiert
  - ▶ Aktoren sind nicht komponierbar
  - ▶ Tests können aufwendig werden
  - ▶ Unveränderlichkeit kann in Scala nicht garantiert werden
  - ▶ Umfangreiche Konfigurationsmöglichkeiten

# Zusammenfassung

- ▶ Unterschiede Akka / Aktormodell:
  - ▶ Nachrichtenordnung wird pro Sender / Receiver Paar garantiert
  - ▶ Futures sind keine Aktoren
  - ▶ ActorRef identifiziert einen eindeutigen Aktor
  - ▶ Die Regeln können gebrochen werden (zu Testzwecken)
- ▶ Fehlerbehandlung steht im Vordergrund
- ▶ Verteilte Aktorensystem können per Akka Remoting miteinander kommunizieren
- ▶ Mit Event-Sourcing können Zustände über Systemausfälle hinweg wiederhergestellt werden.

Reaktive Programmierung  
Vorlesung 8 vom 15.05.19  
Meta-Programmierung

Christoph Lüth, Martin Ring

Universität Bremen

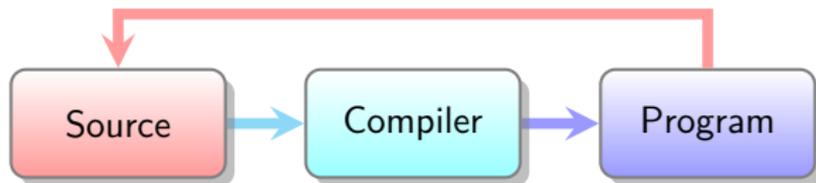
Sommersemester 2019

# Fahrplan

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

# Was ist Meta-Programmierung?

“Programme höherer Ordnung” / Makros



# Was sehen wir heute?

- ▶ Anwendungsbeispiel: JSON Serialisierung
- ▶ Meta-Programmierung in Scala:
  - ▶ Scala Meta
- ▶ Meta-Programmierung in Haskell:
  - ▶ Template Haskell
- ▶ Generische Programmierung in Scala und Haskell

# Beispiel: JSON Serialisierung

Scala

```
case class Person(  
  names: List[String],  
  age: Int  
)
```

Haskell

```
data Person = Person {  
  names :: [String],  
  age :: Int  
}
```

Ziel: Scala  $\xleftrightarrow{\text{JSON}}$  Haskell

# JSON: Erster Versuch

JSON1.scala

# JSON: Erster Versuch

JSON1.scala

- ▶ Unpraktisch: Für jeden Typ muss manuell eine Instanz erzeugt werden
- ▶ Idee: Makros for the win

# Klassische Metaprogrammierung (Beispiel C)

```
#define square(n) ((n)*(n))  
#define UpTo(i, n) for((i) = 0; (i) < (n); (i)++)
```

```
UpTo(i,10) {  
    printf("i squared is: %d\n", square(i));  
}
```

- ▶ Eigene Sprache: C Präprozessor
- ▶ Keine Typsicherheit: einfache String Ersetzungen

# Metaprogrammierung in Scala: Scalameta

- ▶ Idee: Der Compiler ist im Programm verfügbar

```
> "x + 2 * 7".parse[Term].get.structure
```

```
Term.ApplyInfix(Term.Name("x"), Term.Name("+"), Nil,  
  Seq(Term.ApplyInfix(Lit.Int(2), Term.Name("*"), Nil,  
    Seq(Lit.Int(7))))))
```

- ▶ Abstrakter syntaxbaum (AST) als algebraischer Datentyp → typsicher
- ▶ Sehr komplexer Datentyp ...

# Quasiquotations

- ▶ Idee: Programmcode statt AST
- ▶ Zur Konstruktion ...

```
> val p = q"case class Person(name: String)"  
p: meta.Defn.Class = case class Person(name: String)
```

- ▶ ... und zur Extraktion

```
> val q"case class $name($param)" = p  
name: meta.Type.Name = Person  
param: scala.meta.Term.Param = name: String
```

# Makro Annotationen

- ▶ Idee: Funktion  $AST \rightarrow AST$  zur Compilezeit ausführen
- ▶ Werkzeug: Annotationen

```
class hello extends StaticAnnotation {  
  inline def apply(defn: Any): Any = meta { defn match {  
    case q"object $name { ..$members }" =>  
      q"""object $name {  
        ..$members  
        def hello: Unit = println("Hello")  
      }"""  
    case _ => abort("@hello must annotate an object")  
  } }  
}
```

@hello object Test

# JSON: Zweiter Versuch

JSON2.scala

# JSON: Zweiter Versuch

JSON2.scala

- ▶ Generische Ableitungen für **case classes**
- ▶ Funktioniert das für alle algebraischen Datentypen?

# Generische Programmierung

- ▶ Beispiel: YAML statt JSON erzeugen
- ▶ Idee: Abstraktion über die Struktur von Definitionen
- ▶ Erster Versuch: `ToMap.scala`

# Generische Programmierung

- ▶ Beispiel: YAML statt JSON erzeugen
- ▶ Idee: Abstraktion über die Struktur von Definitionen
- ▶ Erster Versuch: `ToMap.scala`
  - ▶ Das klappt so nicht ...
  - ▶ Keine geeignete Repräsentation!

# Heterogene Listen

- ▶ Generische Abstraktion von Tupeln

```
> val l = 42 :: "foo" :: 4.3 :: HNil  
l: Int :: String :: Double :: HNil = ...
```

- ▶ Viele Operationen normaler Listen vorhanden:
- ▶ Was ist der parameter für flatMap?

# Heterogene Listen

- ▶ Generische Abstraktion von Tupeln

```
> val l = 42 :: "foo" :: 4.3 :: HNil  
l: Int :: String :: Double :: HNil = ...
```

- ▶ Viele Operationen normaler Listen vorhanden:
- ▶ Was ist der parameter für flatMap?  
⇒ Polymorphe Funktionen

# Records

- ▶ Uns fehlen namen
- ▶ Dafür: Records

```
> import shapeless._; record._; import syntax.singleton._  
> val person = ("name" →> "Donald") :: ("age" →> 70) ::  
  HNil
```

```
person: String with KeyTag[String("name"),String] :: Int  
  with KeyTag[String("age"),Int] :: HNil = Donald :: 70  
  :: HNil
```

```
> person("name")
```

```
res1: String = Donald
```

# Die Typklasse Generic

- ▶ Typklasse `Generic[T]`

```
trait Generic[T] {  
  type Repr  
  def from(r: Repr): T  
  def to(t: T): Repr  
}
```

- ▶ kann magisch abgeleitet werden:

```
> case class Person(name: String, age: Int)  
> val gen = Generic[Person]  
gen: shapeless.Generic[Person]{type Repr = String :: Int  
  :: shapeless.HNil} = ...
```

- ▶ → Makro Magie
- ▶ Funktioniert allgemein für algebraische Datentypen

# JSON Serialisierung: Teil 3

JSON3.scala

# Automatische Linsen

```
case class Address(street: String, city: String, zip: Int)
case class Person(name: String, age: Int, address: Address)

val streetLens = lens[Person] >> 'address >> 'street
```

# Zusammenfassung

- ▶ Meta-Programmierung: “Programme Höherer Ordnung”
- ▶ Scalameta: Scala in Scala manipulieren
- ▶ Quasiquotations: Reify and Splice
- ▶ Macros mit Scalameta:  $AST \rightarrow AST$  zur Compilezeit
- ▶ Äquivalent in Haskell: TemplateHaskell
- ▶ Generische Programmierung in Shapeless
- ▶ Äquivalent in Haskell: GHC.Generic

Reaktive Programmierung  
Vorlesung 9 vom 22.05.19  
Bidirektionale Programmierung — Zippers and Lenses

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# 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 Pos    = Int
data Editor = Ed { text    :: String
                  , cursor  :: Pos }
```

- ▶ Cursor bewegen (links)

```
go_left :: Editor → Editor
go_left Ed{text= t, cursor= c}
  | c == 0 = error "At start of line"
  | otherwise = Ed{text= t, cursor= c- 1}
```

- ▶ Text rechts einfügen:

```
insert :: Editor → Char → Editor
insert Ed{text= t, cursor= c} text =
  let (as, bs) = splitAt c t
  in  Ed{text= as ++ (text: bs), cursor= c+1}
```

# Aufwand

- ▶ **Aufwand** für Manipulation?  
 $O(n)$  mit  $n$  Länge des gesamten Textes
  
- ▶ Geht das auch einfacher?

# Ein einfacher Editor

- ▶ Datenstrukturen:

```
data Editor = Ed { before :: [Char] — In reverse order
                  , cursor :: Maybe Char
                  , after  :: [Char] }
```

- ▶ Invariante: cursor == Nothing gdw. before und after leer
- ▶ Cursor **bewegen** (links):

```
go_left :: Editor → Editor
go_left e@(Ed [] _) = e
go_left (Ed (a:as) (Just a) bs) = Ed as (Just a) (c: bs)
```

- ▶ Text unter dem Cursor löschen:

```
delete :: Editor → Editor
delete (Ed as _ (b:bs)) = Ed as (Just b) bs
delete (Ed (a:as) _ []) = Ed as (Just a) []
delete (Ed [] _ []) = Ed [] Nothing []
```

# Manipulation strukturierter Datentypen

- ▶ Anderer Datentyp:  $n$ -äre Bäume (rose trees)

```
data Tree a = Node a [Tree a]
```

- ▶ Bspw. abstrakte Syntax von einfachen Ausdrücken
- ▶ Update auf Beispielterm  $t = a * b - c * d$ : ersetze  $b$  durch  $x + y$

```
t = Node "-" [ Node "*" [Node "a" [], Node "b" []],  
              , Node "*" [Node "c" [], Node "d" []]  
            ]
```

- ▶ Referenzierung durch Namen

```
upd1 :: Eq a => a -> Tree a -> Tree a -> Tree a
```

- ▶ Referenzierung durch Pfad: **type** Path=[Int]

```
type Path = [Int]  
upd2 :: Path -> Tree a -> Tree a -> Tree a
```

# Aufwand

- ▶ Aufwand: Mittlere Aufwand  $O(\log n)$ , worst case  $O(n)$   
 $n$  Anzahl der Knoten
- ▶ Geht das besser — wie beim einfachen Editor?
- ▶ Generalisierung der Idee

# Der Zipper

- ▶ Idee: **Kontext** nicht **wegwerfen!**

- ▶ Nicht: **type** Path=[Int]

- ▶ Sondern:

```
data Ctxt a = Empty
           | Cons [Tree a] a (Ctxt a) [Tree a]
```

- ▶ Kontext ist 'inverse Umgebung' ("*Like a glove turned inside out*")

- ▶ Besteht aus linken Nachbarn, Knoten, Kontext darüber, rechtem Nachbarn

- ▶ Loc a ist **Baum** mit **Fokus**

```
newtype Loc a = Loc (Tree a, Ctxt a)
```

# Ziping Trees: Navigation

## ► Fokus nach **links**

```
go_left :: Loc a → Loc a
go_left (Loc(t, c)) = case c of
  Cons (l:le) a up ri → Loc(l, Cons le a up (t:ri))
  _                    → error "go_left: at first"
```

## ► Fokus nach **rechts**

```
go_right :: Loc a → Loc a
go_right (Loc(t, c)) = case c of
  Cons le a up (r:ri) → Loc(r, Cons (t:le) a up ri)
  _                    → error "go_right: at last"
```

# Ziping Trees: Navigation

## ► Fokus nach **oben**

```
go_up :: Loc a → Loc a
go_up (Loc (t, c)) = case c of
  Empty → error "go_up: at the top"
  Cons le a up ri →
    Loc (Node a (reverse le ++ t:ri), up)
```

## ► Fokus nach **unten**

```
go_down :: Loc a → Loc a
go_down (Loc (t, c)) = case t of
  Node _ [] → error "go_down: at leaf"
  Node a (t:ts) → Loc (t, Cons [] a c ts)
```

# Einfügen

- ▶ **Einfügen**: Wo?
- ▶ **Überschreiben** des Fokus

```
update :: Tree a → Loc a → Loc a
update t (Loc (_, c)) = Loc (t, c)
```

- ▶ **Links** des Fokus einfügen

```
insert_left :: Tree a → Loc a → Loc a
insert_left t1 (Loc (t, c)) = case c of
  Empty → error "insert_left: insert at empty"
  Cons le a up ri → Loc(t, Cons (t1:le) a up ri)
```

- ▶ **Rechts** des Fokus einfügen

```
insert_right :: Tree a → Loc a → Loc a
insert_right t1 (Loc (t, c)) = case c of
  Empty → error "insert_right: insert at empty"
  Cons le a up ri → Loc(t, Cons le a up (t1:ri))
```

# Ersetzen und Löschen

► Unterbaum im Fokus löschen: wo ist der neue Fokus?

- 1 Rechter Baum, wenn vorhanden
- 2 Linker Baum, wenn vorhanden
- 3 Elternknoten

```
delete :: Loc a → Loc a
delete (Loc(_, c)) = case c of
  Empty → error "delete: delete at top"
  Cons le a up (r:ri) → Loc(r, Cons le a up ri)
  Cons (l:le) a up [] → Loc(l, Cons le a up [])
  Cons [] a up [] → Loc (Node a [], up)
```

► *"We note that delete is not such a simple operation."*

# Schnelligkeit

- ▶ Wie **schnell** sind Operationen?

# Schnelligkeit

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

# Schnelligkeit

- ▶ Wie **schnell** sind Operationen?
  - ▶ Aufwand: `go_up`  $O(\text{left}(n))$ , 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[A](value: A) extends Tree[A]  
case class Node[A](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 Loc[A](tree: Tree[A], context: Context[A])
```

# Tree-Zipper: Navigation

## ► Fokus nach **links**

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

## ► Fokus nach **rechts**

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

## ► Fokus nach **unten links**

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

## ► Fokus nach **unten rechts**

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

# Tree-Zipper: Einfügen und Löschen

## ▶ Einfügen links

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

## ▶ Einfügen rechts

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

## ▶ Löschen

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

## ▶ Neuer Fokus: anderer Teilbaum

# Ziping Lists

- ▶ Listen:

```
data List a = Nil | Cons a (List a)
```

- ▶ Damit:

```
data Ctxt a = Empty | Snoc (Ctxt a) a
```

- ▶ Listen sind ihr 'eigener Kontext' :

$$\text{List } a \cong \text{Ctxt } a$$

# Ziping Lists: Fast Reverse

- ▶ Listenumkehr **schnell**:

```
fastrev1 :: List a → List a
fastrev1 xs = rev (top xs) where
  rev :: Loc a → List a
  rev (Loc (Nil, as))      = as
  rev (Loc (Cons x xs, as)) = rev (Loc (xs, Cons x as))
```

- ▶ Vergleiche:

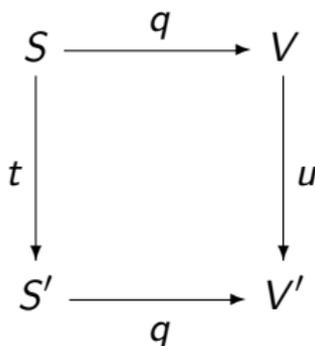
```
fastrev2 :: [a] → [a]
fastrev2 xs = rev xs [] where
  rev :: [a] → [a] → [a]
  rev []      as = as
  rev (x:xs) as = rev xs (x:as)
```

- ▶ Zweites Argument von rev: **Kontext**
  - ▶ Liste der Elemente davor in **umgekehrter** Reihenfolge

# Bidirektionale Programmierung

- ▶ Verallgemeinerung der Idee des Kontext
- ▶ Motivierendes Beispiel: Update in einer Datenbank
- ▶ Weitere Anwendungsfelder:
  - ▶ Benutzerschnittstellen (MVC)
  - ▶ Datensynchronisation

# View Updates

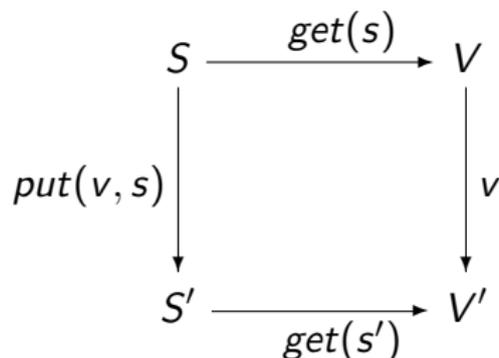


- ▶ View  $v$  durch Anfrage  $q$  (Bsp: Anfrage auf Datenbank)
- ▶ View wird **verändert** (Update  $u$ )
- ▶ Quelle  $S$  soll entsprechend angepasst werden (**Propagation** der Änderung)
- ▶ Problem:  $q$  soll **beliebig** sein
  - ▶ Nicht-injektiv? Nicht-surjektiv?

# 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:

$$get : S \longrightarrow V$$

$$put : V \times S \longrightarrow S$$

- ▶ Es müssen die **Linsengesetze** gelten:

$$get(put(v, s)) = v$$

$$put(get(s), s) = s$$

$$put(v, put(w, s)) = put(v, s)$$

## Erweiterung: Erzeugung

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

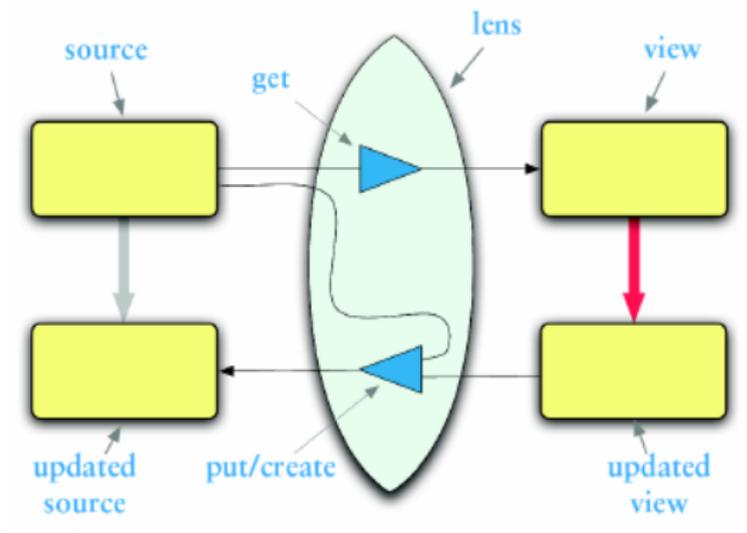
- ▶ Signatur:

$$\text{create} : V \longrightarrow S$$

- ▶ Weitere **Gesetze**:

$$\begin{aligned} \text{get}(\text{create}(v)) &= v \\ \text{put}(v, \text{create}(w)) &= \text{create}(w) \end{aligned}$$

# 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

# Linse konstruieren

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

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

- ▶ Die **Identitätslinse**:

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

# Linsen komponieren

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

$$\begin{aligned}L_2 \cdot L_1 & : S_1 \longleftrightarrow S_3 \\ \text{get} & = \text{get}_2 \cdot \text{get}_1 \\ \text{put}(v, s) & = \text{put}_1(\text{put}_2(v, \text{get}_1(s)), s) \\ \text{create} & = \text{create}_1 \cdot \text{create}_2\end{aligned}$$

- ▶ Beispiel hier:

$$\text{TurtleX} = \text{TurtlePosition} \cdot \text{PointX}$$

# Mehr Linsen und Bidirektionale Programmierung

- ▶ 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: Reaktive Ströme

Reaktive Programmierung  
Vorlesung 10 vom 28.05.19  
Reactive Streams (Observables)

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Klassifikation von Effekten

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

- ▶ Try macht **Fehler** explizit
- ▶ Future macht **Verzögerung** explizit
- ▶ Explizite Fehler bei Nebenläufigkeit **unverzichtbar**
- ▶ Heute: Observables

## Future[T] ist dual zu Try[T]

```
trait Future[T] {  
  def onComplete(callback: Try[T] => Unit): Unit  
}
```

## Future[T] ist dual zu Try[T]

```
trait Future[T] {  
  def onComplete(callback: Try[T] => Unit): Unit  
}
```

►  $(\text{Try}[T] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$

# Future[T] ist dual zu Try[T]

```
trait Future[T] {  
  def onComplete(callback: Try[T] => Unit): Unit  
}
```

- ▶  $(\text{Try}[T] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$
- ▶ Umgedreht:  
 $\text{Unit} \Rightarrow (\text{Unit} \Rightarrow \text{Try}[T])$

# Future[T] ist dual zu Try[T]

```
trait Future[T] {  
  def onComplete(callback: Try[T] => Unit): Unit  
}
```

- ▶  $(\text{Try}[T] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$
- ▶ Umgedreht:  
 $\text{Unit} \Rightarrow (\text{Unit} \Rightarrow \text{Try}[T])$
- ▶  $() \Rightarrow ( () \Rightarrow \text{Try}[T] )$

# Future[T] ist dual zu Try[T]

```
trait Future[T] {  
  def onComplete(callback: Try[T] => Unit): Unit  
}
```

- ▶  $(\text{Try}[T] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$
- ▶ Umgedreht:  
 $\text{Unit} \Rightarrow (\text{Unit} \Rightarrow \text{Try}[T])$
- ▶  $() \Rightarrow ( () \Rightarrow \text{Try}[T] )$
- ▶  $\approx \text{Try}[T]$

# Try vs Future

- ▶ Try[T]: Blockieren  $\longrightarrow$  Try[T]
- ▶ Future[T]: Callback  $\longrightarrow$  Try[T] (**Reaktiv**)

# Was ist dual zu Iterable ?

```
trait Iterable[T] { def iterator(): Iterator[T] }  
trait Iterator[T] { def hasNext: Boolean  
                   def next(): T }
```

## Was ist dual zu Iterable ?

```
trait Iterable[T] { def iterator(): Iterator[T] }  
trait Iterator[T] { def hasNext: Boolean  
                  def next(): T }
```

▶ () =>

## Was ist dual zu Iterable ?

```
trait Iterable[T] { def iterator(): Iterator[T] }  
trait Iterator[T] { def hasNext: Boolean  
                  def next(): T }
```

▶ () => () => Try[Option[T]]

# Was ist dual zu Iterable ?

```
trait Iterable[T] { def iterator(): Iterator[T] }  
trait Iterator[T] { def hasNext: Boolean  
                  def next(): T }
```

- ▶  $() \Rightarrow () \Rightarrow \text{Try}[\text{Option}[T]]$
- ▶ Umgedreht:  
 $(\text{Try}[\text{Option}[T]] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$

## Was ist dual zu Iterable ?

```
trait Iterable[T] { def iterator(): Iterator[T] }  
trait Iterator[T] { def hasNext: Boolean  
                  def next(): T }
```

- ▶  $() \Rightarrow () \Rightarrow \text{Try}[\text{Option}[T]]$
- ▶ Umgedreht:  
 $(\text{Try}[\text{Option}[T]] \Rightarrow \text{Unit}) \Rightarrow \text{Unit}$
- ▶  $( T \Rightarrow \text{Unit}, \text{Throwable} \Rightarrow \text{Unit}, () \Rightarrow \text{Unit} ) \Rightarrow \text{Unit}$

# Observable[T] ist dual zu Iterable [T]

```
trait Iterable[T] {  
  def iterator:  
    Iterator[T]  
}
```

```
trait Iterator[T] {  
  def hasNext: Boolean  
  def next(): T  
}
```

```
trait Observable[T] {  
  def subscribe(Observer[T]  
    observer):  
    Subscription  
}
```

```
trait Observer[T] {  
  def onNext(T value): Unit  
  def onError(Throwable error): Unit  
  def onComplete(): Unit  
}
```

```
trait Subscription {  
  def unsubscribe(): Unit  
}
```

# Warum Observables?

```
class Robot(var pos: Int, var battery: Int) {  
  def goldAmounts = new Iterable[Int] {  
    def iterator = new Iterator[Int] {  
      def hasNext = world.length > pos  
      def next() = if (battery > 0) {  
        Thread.sleep(1000)  
        battery -= 1  
        pos += 1  
        world(pos).goldAmount  
      } else sys.error("low battery")  
    }  
  }  
}  
  
(robotA.goldAmounts zip robotB.goldAmounts)  
  .map(_ + _).takeUntil(_ > 5)
```

# Observable Robots

```
class Robot(var pos: Int, var battery: Int) {  
  def goldAmounts = Observable { obs =>  
    var continue = true  
    while (continue && world.length > pos) {  
      if (battery > 0) {  
        Thread.sleep(1000)  
        pos += 1  
        battery -= 1  
        obs.onNext(world(pos).gold)  
      } else obs.onError(new Exception("low battery"))  
    }  
    obs.onCompleted()  
    Subscription(continue = false)  
  }  
}
```

```
(robotA.goldAmounts zip robotB.goldAmounts)
```

DEMO

# Observable Contract

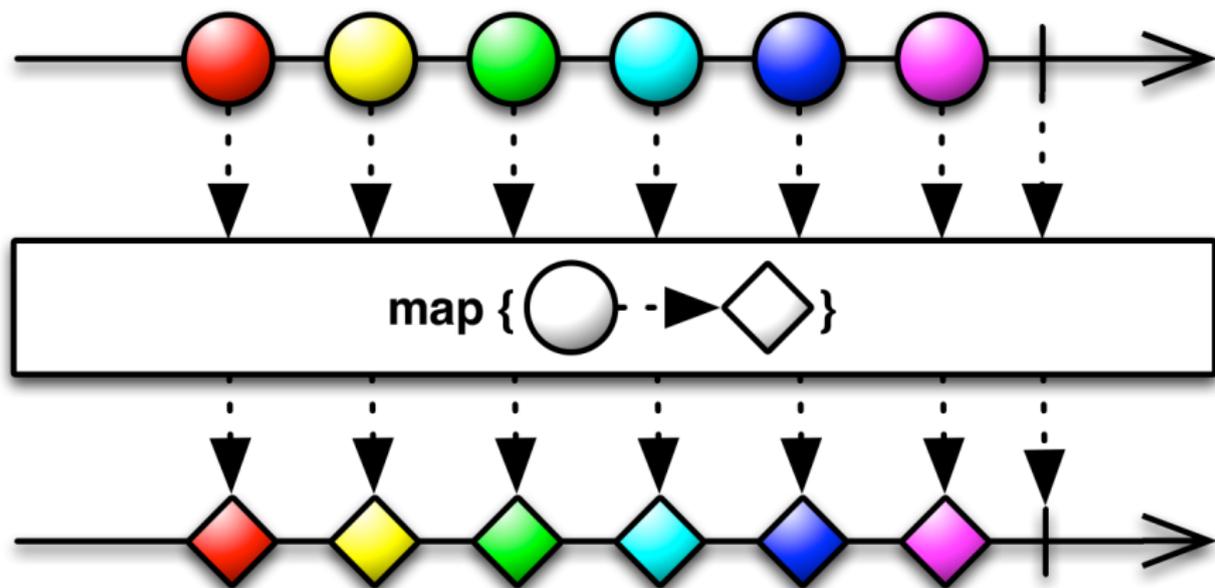
- ▶ die `onNext` Methode eines Observers wird beliebig oft aufgerufen.
- ▶ `onCompleted` oder `onError` werden nur einmal aufgerufen und schließen sich gegenseitig aus.
- ▶ Nachdem `onCompleted` oder `onError` aufgerufen wurde wird `onNext` nicht mehr aufgerufen.

`onNext*(onCompleted|onError)?`

- ▶ Diese Spezifikation wird durch die Konstruktoren erzwungen.

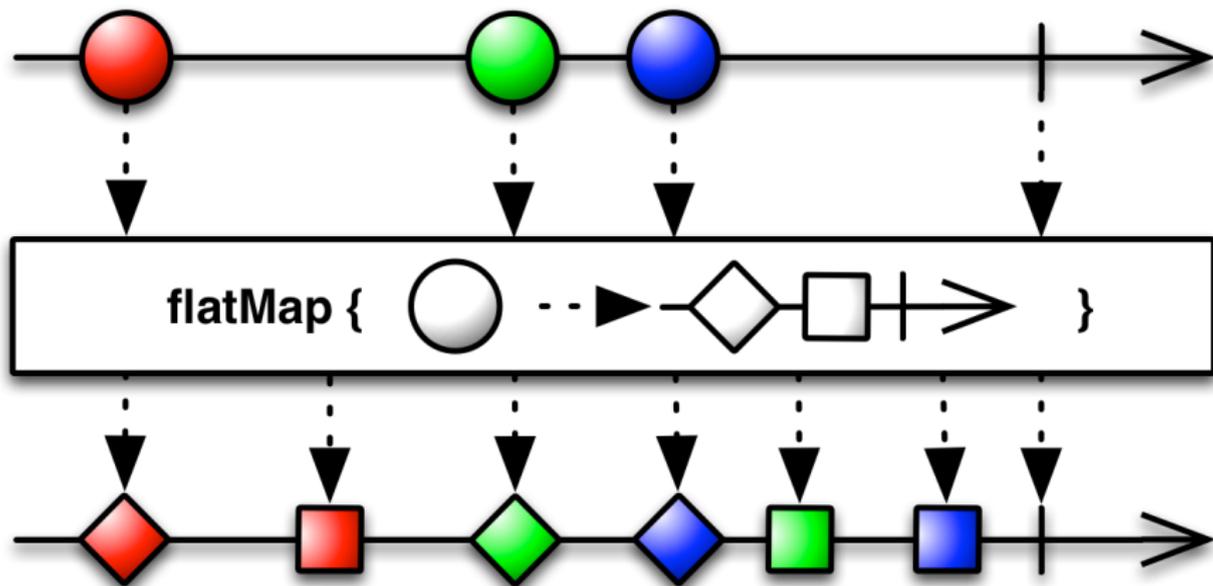
# map

```
def map[U]( f: T => U ): Observable[U]
```



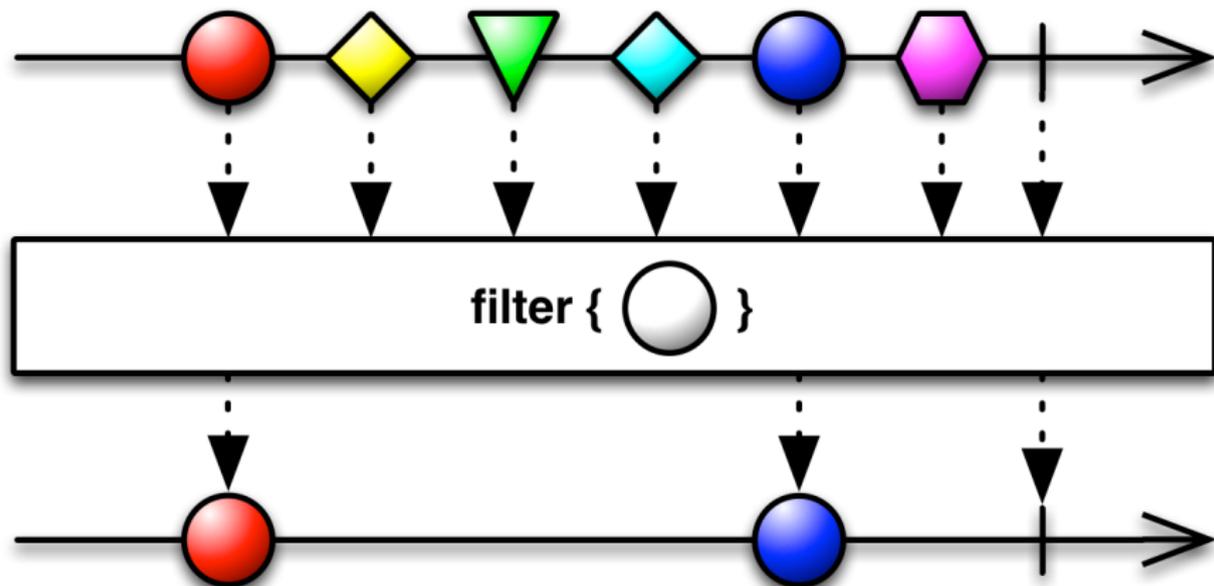
# flatMap

```
def flatMap[U](f: T => Observable[U]): Observable[U]
```



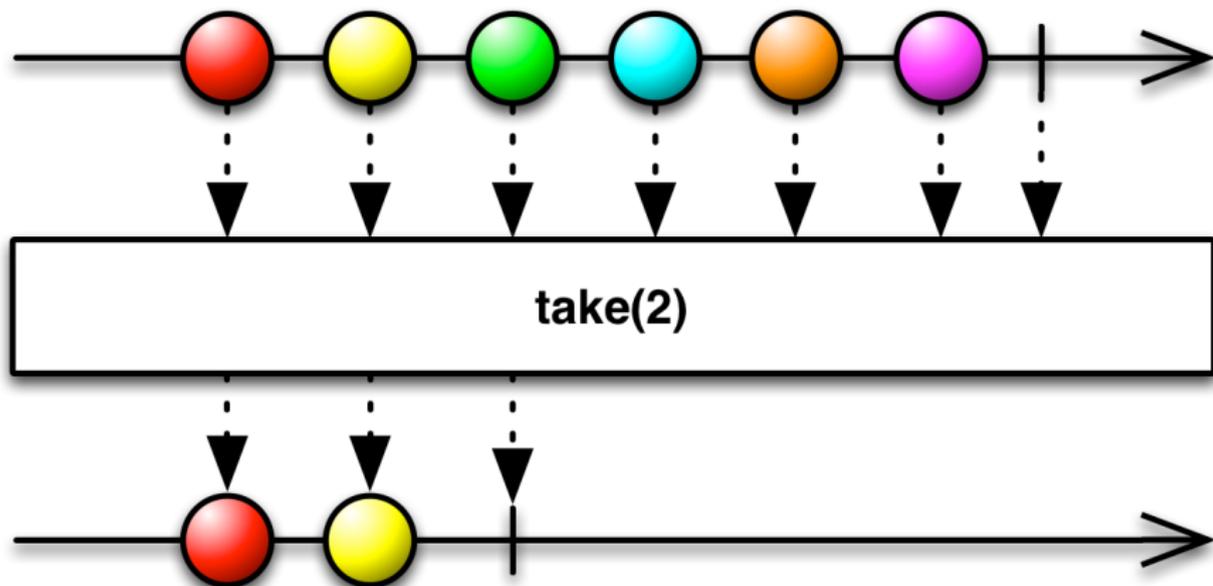
# filter

```
def filter(f: T => Boolean): Observable[T]
```



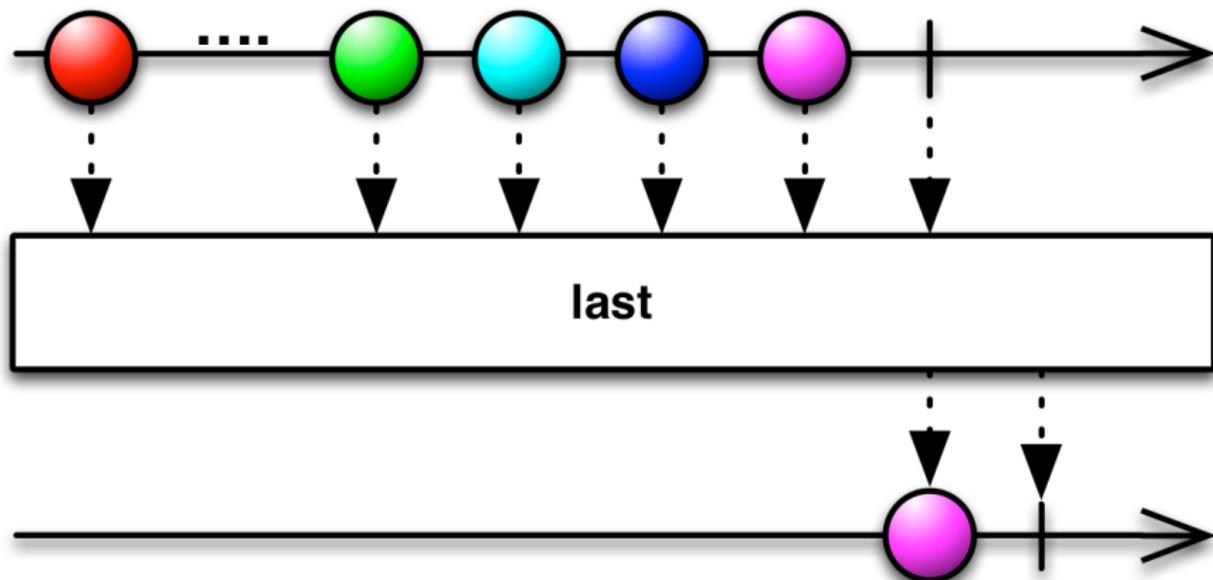
# take

```
def take(count: Int): Observable[T]
```



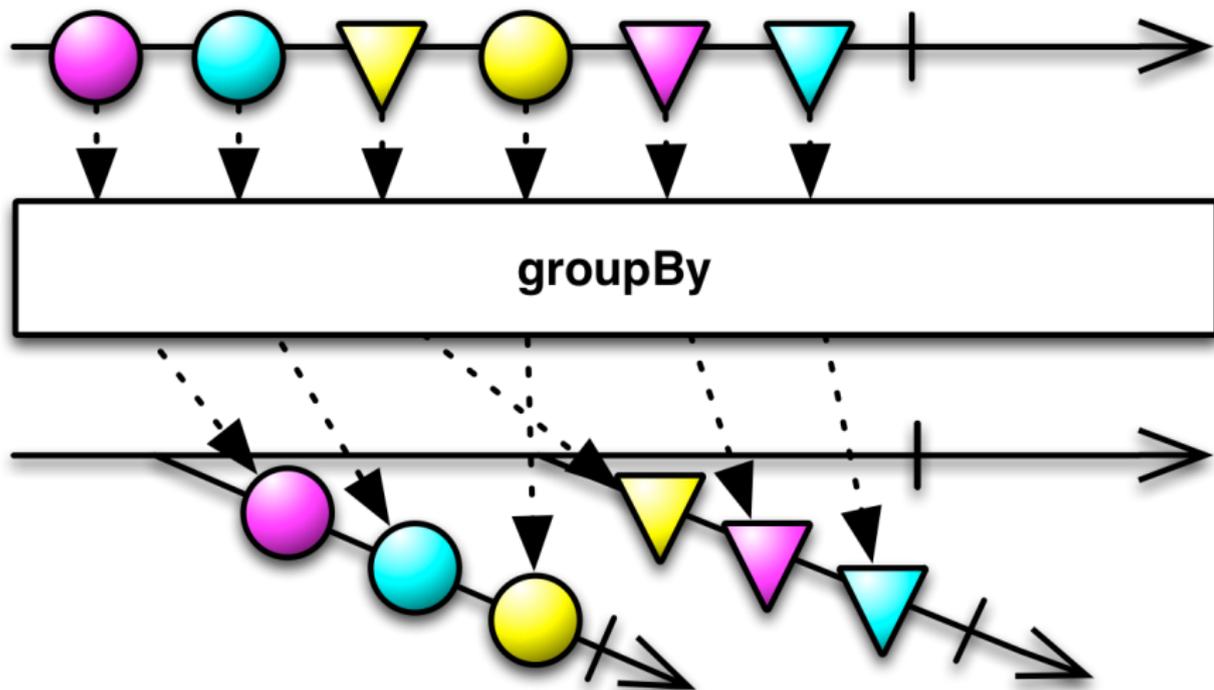
# last

```
def last: Observable[T]
```



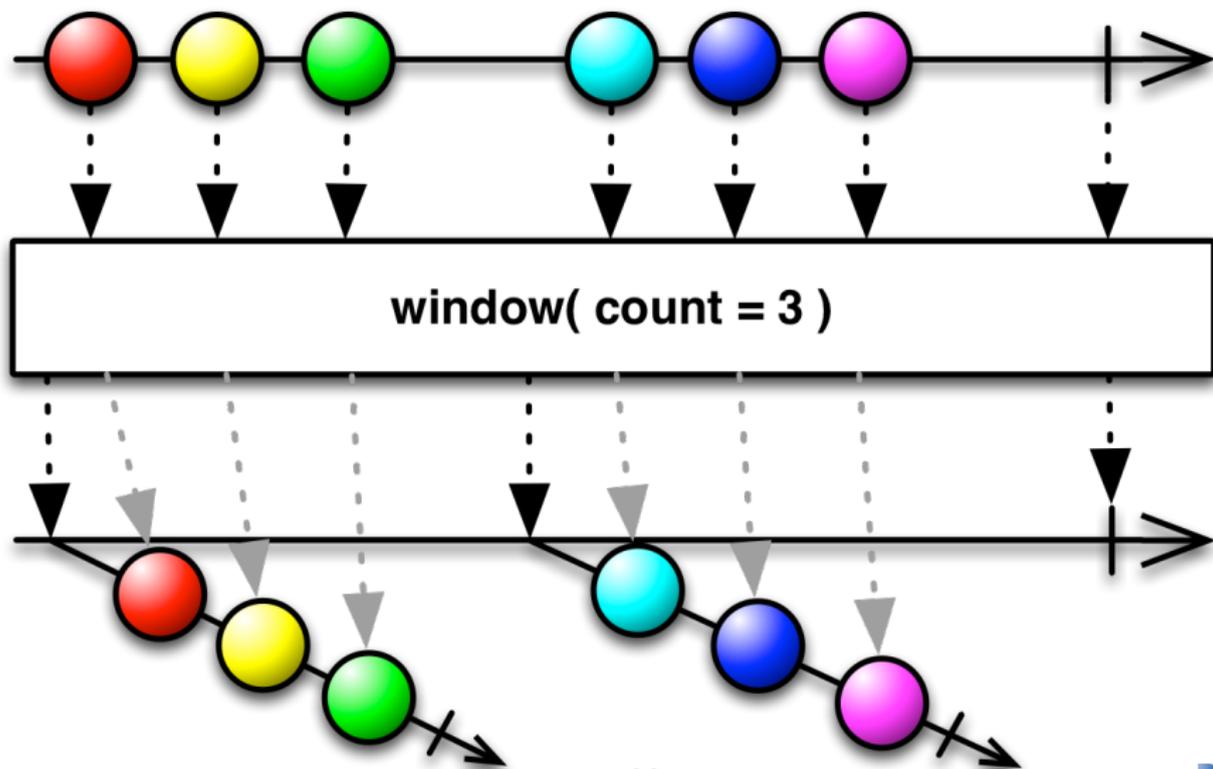
## groupBy

```
def groupBy[U](T => U): Observable[Observable[T]]
```



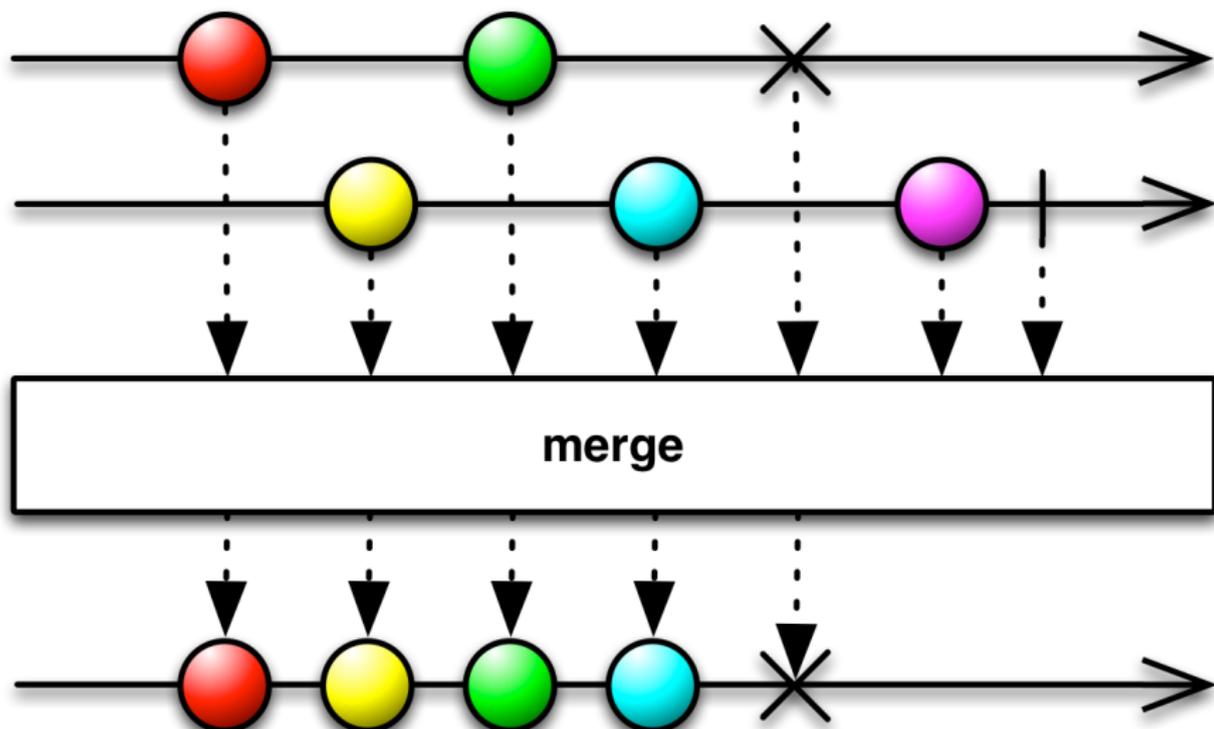
# window

```
def window(count: Int): Observable[Observable[T]]
```



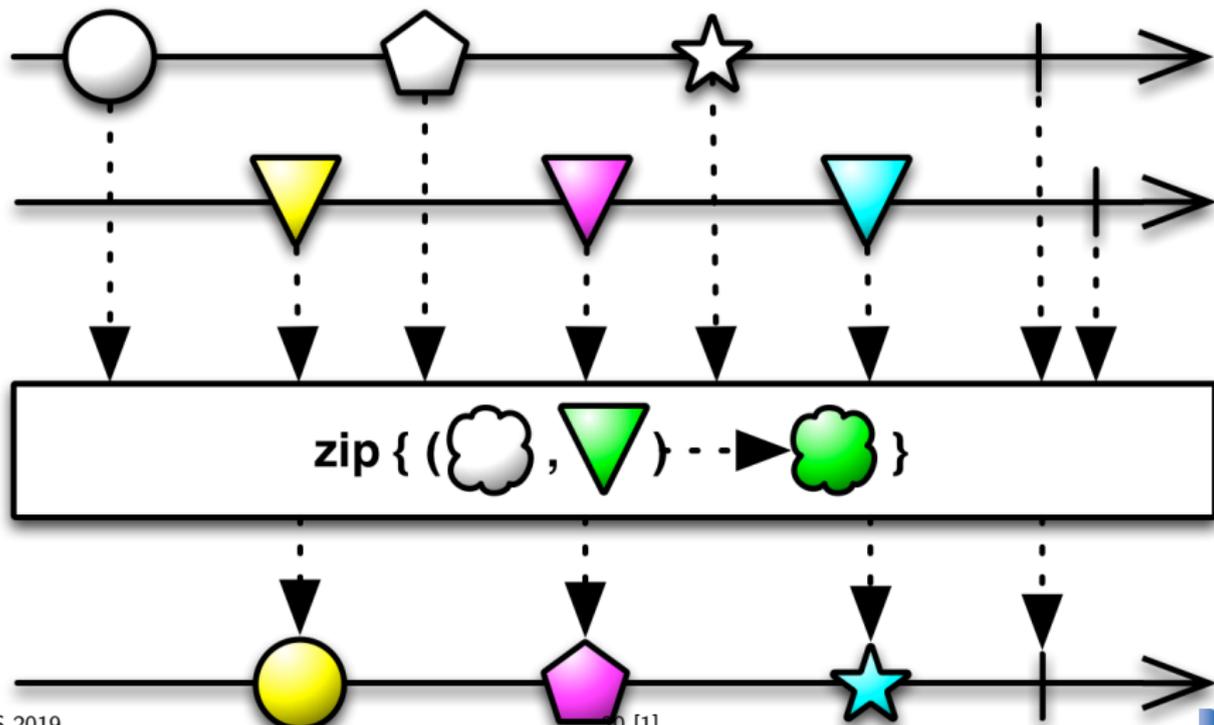
# merge

```
def merge[T](obss: Observable[T]*): Observable[T]
```



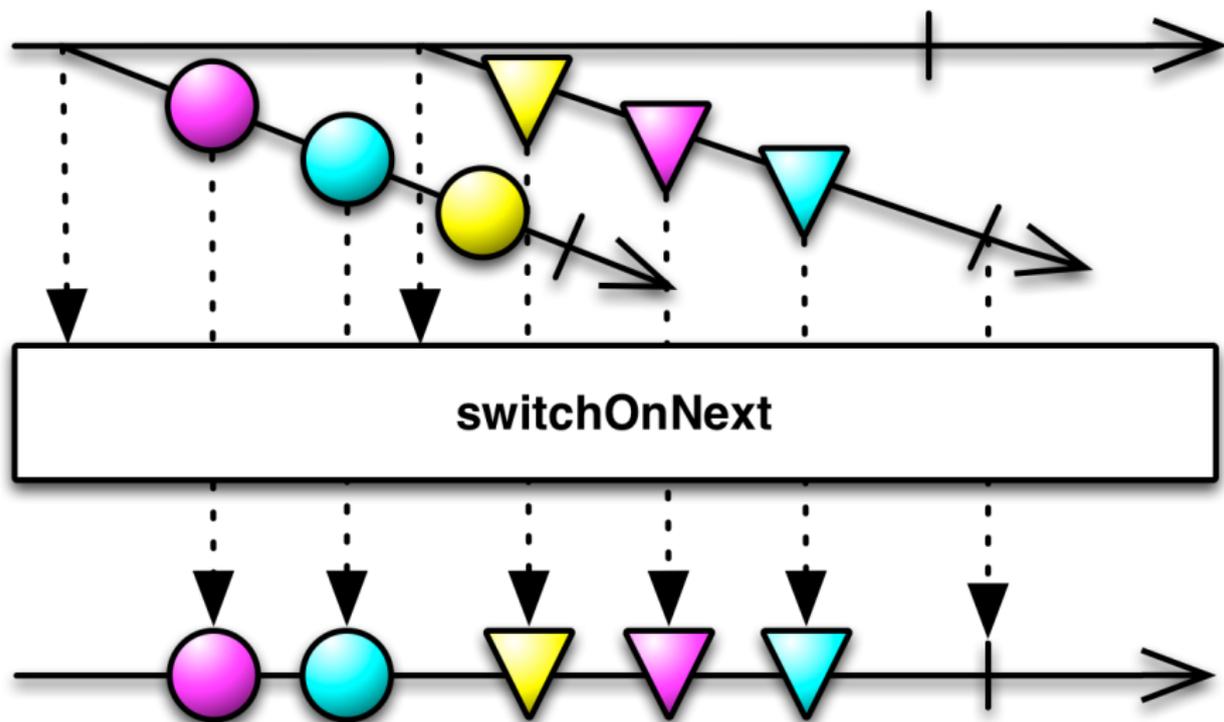
# zip

```
def zip[U,S](obs: Observable[U], f: (T,U) => S):  
  Observable[S]
```



# switch

```
def switch(): Observable[T]
```



# Subscriptions

- ▶ Subscriptions können mehrfach gecancelt werden. Deswegen müssen sie idempotent sein.

```
Subscription(cancel: ⇒ Unit)
```

```
BooleanSubscription(cancel: ⇒ Unit)
```

```
class MultiAssignmentSubscription {  
  def subscription_=(s: Subscription)  
  def subscription: Subscription  
}
```

```
CompositeSubscription(subscriptions: Subscription*)
```

# Schedulers

- ▶ Nebenläufigkeit über Scheduler

```
trait Scheduler {  
  def schedule(work:  $\Rightarrow$  Unit): Subscription  
}  
  
trait Observable[T] {  
  ...  
  def observeOn(schedule: Scheduler): Observable[T]  
}
```

- ▶ `Subscription.cancel()` muss synchronisiert sein.

# Hot vs. Cold Streams

- ▶ **Hot Observables** schicken allen Observern die gleichen Werte zu den gleichen Zeitpunkten.

z.B. Maus Klicks

- ▶ **Cold Observables** fangen erst an Werte zu produzieren, wenn man ihnen zuhört. Für jeden Observer von vorne.

z.B. `Observable.from(Seq(1,2,3))`

# Observables Bibliotheken

- ▶ Observables sind eine Idee von Eric Meijer
- ▶ Bei Microsoft als .net *Reactive Extension* (Rx) entstanden
- ▶ Viele Implementierungen für verschiedene Plattformen
  - ▶ RxJava, RxScala, RxClosure (Netflix)
  - ▶ RxPY, RxJS, ... (ReactiveX)
- ▶ Vorteil: Elegante Abstraktion, Performant
- ▶ Nachteil: Push-Modell ohne Bedarfsrückkopplung

# Zusammenfassung

- ▶ Futures sind dual zu Try
- ▶ Observables sind dual zu Iterable
- ▶ Observables abstrahieren viele Nebenläufigkeitsprobleme weg:  
Außen **funktional** (Hui) - Innen **imperativ** (Pfui)
- ▶ Nächstes mal: **Back Pressure** und noch mehr reaktive Ströme

Reaktive Programmierung  
Vorlesung 11 vom 05.06.19  
Reactive Streams II

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Rückblick: Observables

- ▶ Observables sind „asynchrone Iterables“
- ▶ Asynchronität wird durch **Inversion of Control** erreicht
- ▶ Es bleiben drei Probleme:
  - ▶ Die Gesetze der Observable können leicht verletzt werden.
  - ▶ Ausnahmen beenden den Strom - **Fehlerbehandlung**?
  - ▶ Ein zu schneller Observable kann den Empfangenden Thread **überfluten**

# Datenstromgesetze

- ▶ `onNext*(onError|onComplete)`
- ▶ Kann leicht verletzt werden:

```
Observable[Int] { observer =>
  observer.onNext(42)
  observer.onCompleted()
  observer.onNext(1000)
  Subscription()
}
```

- ▶ Wir können die Gesetze erzwingen: CODE DEMO

# Fehlerbehandlung

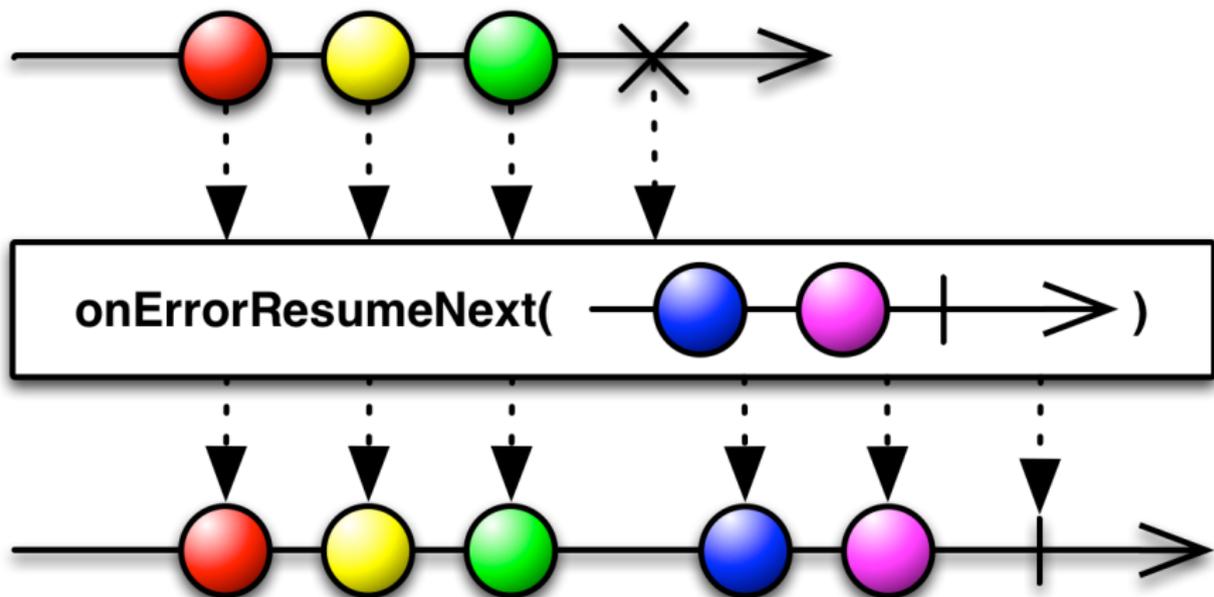
- ▶ Wenn Datenströme Fehler produzieren, können wir diese möglicherweise behandeln.
- ▶ Aber: *Observer.onError* beendet den Strom.

```
observable.subscribe(  
  onNext = println ,  
  onError = ??? ,  
  onCompleted = println("done"))
```

- ▶ *Observer.onError* ist für die Wiederherstellung des Stroms ungeeignet!
- ▶ Idee: Wir brauchen mehr Kombinatoren!

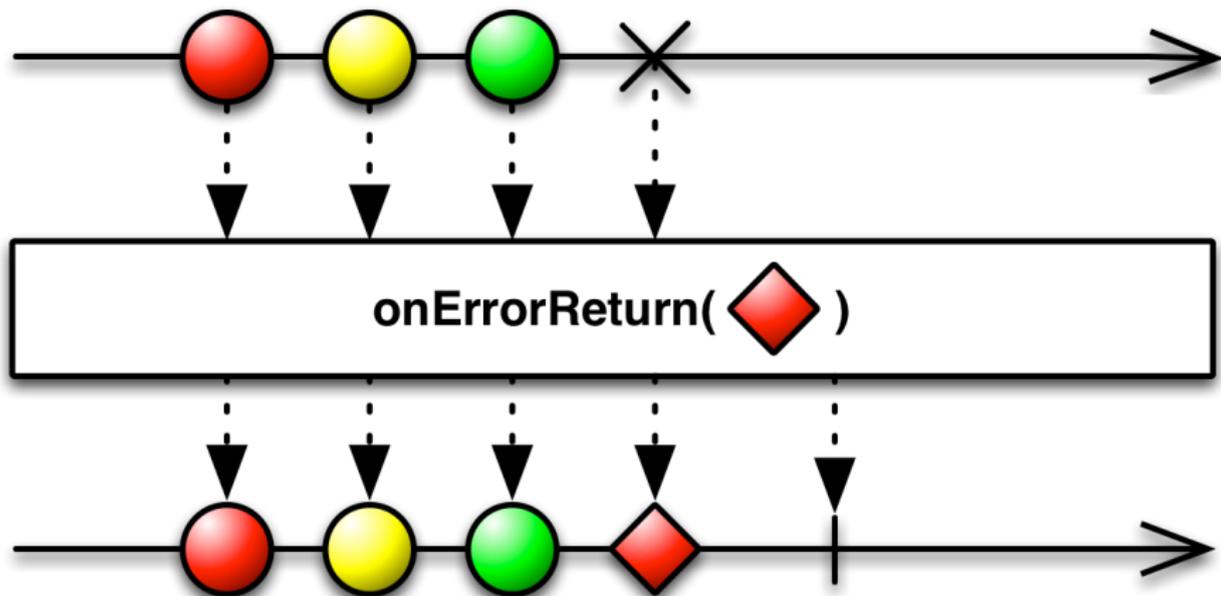
## onErrorResumeNext

```
def onErrorResumeNext(f: => Observable[T]): Observable[T]
```



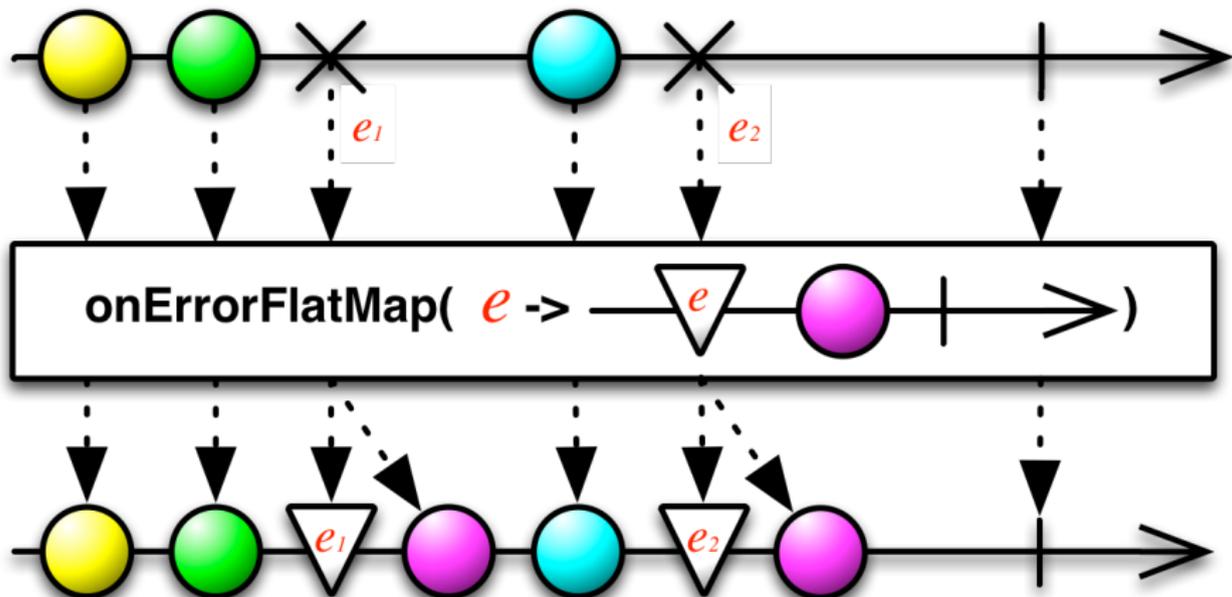
# onErrorReturn

```
def onErrorReturn(f: => T): Observable[T]
```



# onErrorFlatMap

```
def onErrorFlatMap(f: Throwable => Observable[T]):  
  Observable[T]
```



# Schedulers

- ▶ Nebenläufigkeit über Scheduler

```
trait Scheduler {  
  def schedule(work: ⇒ Unit): Subscription  
}  
  
trait Observable[T] {  
  ...  
  def observeOn(schedule: Scheduler): Observable[T]  
}
```

- ▶ CODE DEMO

# Little's Gesetz

- ▶ In einer stabilen Warteschlange gilt:

$$L = \lambda \times W$$

- ▶ Länge der Warteschlange = Ankunftsrate  $\times$  Durchschnittliche Wartezeit
- ▶ Ankunftsrate =  $\frac{\text{Länge der Warteschlange}}{\text{Durchschnittliche Wartezeit}}$

# Little's Gesetz

- ▶ In einer stabilen Warteschlange gilt:

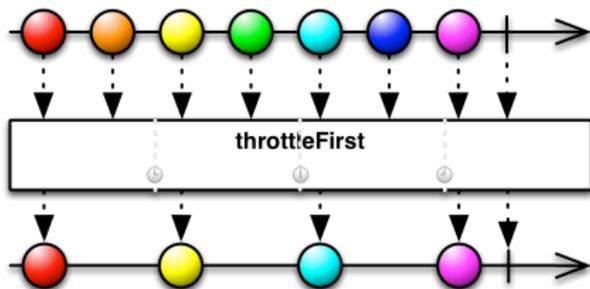
$$L = \lambda \times W$$

- ▶ Länge der Warteschlange = Ankunftsrate  $\times$  Durchschnittliche Wartezeit
- ▶ Ankunftsrate =  $\frac{\text{Länge der Warteschlange}}{\text{Durchschnittliche Wartezeit}}$
- ▶ Wenn ein Datenstrom über einen längeren Zeitraum mit einer Frequenz  $> \lambda$  Daten produziert, haben wir ein Problem!

# Throttling / Debouncing

- ▶ Wenn wir  $L$  und  $W$  kennen, können wir  $\lambda$  bestimmen. Wenn  $\lambda$  überschritten wird, müssen wir etwas unternehmen.
- ▶ Idee: Throttling

```
stream.throttleFirst(lambda)
```

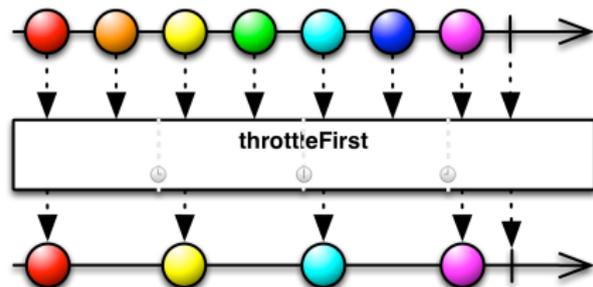
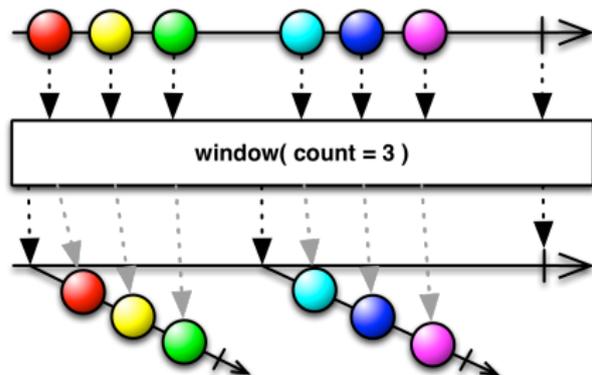


- ▶ Problem: Kurzzeitige Überschreitungen von  $\lambda$  sollen nicht zu Throttling führen.

# Throttling / Debouncing

- ▶ Besser: Throttling erst bei längerer Überschreitung der Kapazität:

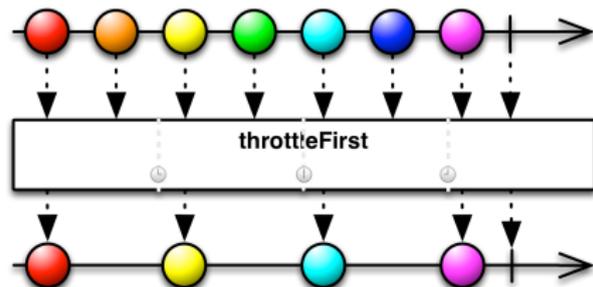
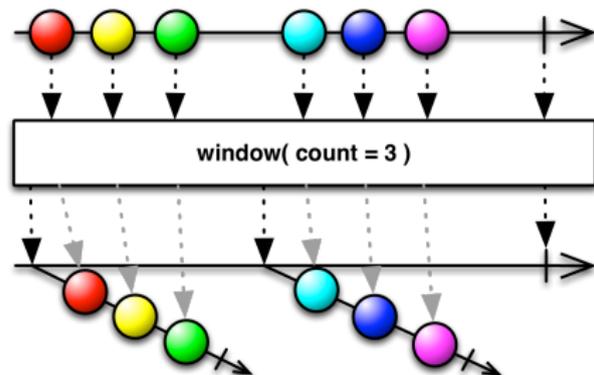
```
stream.window(count = L)  
    .throttleFirst(lambda * L)
```



# Throttling / Debouncing

- ▶ Besser: Throttling erst bei längerer Überschreitung der Kapazität:

```
stream.window(count = L)  
    .throttleFirst(lambda * L)
```



- ▶ Was ist wenn wir selbst die Daten Produzieren?

# Back Pressure

- ▶ Wenn wir Kontrolle über die Produktion der Daten haben, ist es unsinnig, sie wegzuworfen!
- ▶ Wenn der Konsument keine Daten mehr annehmen kann soll der Produzent aufhören sie zu Produzieren.
- ▶ Erste Idee: Wir können den produzierenden Thread blockieren

```
observable.observeOn(producerThread)  
                .subscribe(onNext = someExpensiveComputation)
```

- ▶ Reaktive Datenströme sollen aber gerade verhindern, dass Threads blockiert werden!

# Back Pressure

- ▶ Bessere Idee: der Konsument muss mehr Kontrolle bekommen!

```
trait Subscription {  
  def isUnsubscribed: Boolean  
  def unsubscribe(): Unit  
  def requestMore(n: Int): Unit  
}
```

- ▶ Aufwändig in Observables zu implementieren!
- ▶ Siehe <http://www.reactive-streams.org/>

# Reactive Streams Initiative

- ▶ Ingenieure von Kaazing, Netflix, Pivotal, RedHat, Twitter und Typesafe haben einen offenen Standard für reaktive Ströme entwickelt
- ▶ Minimales Interface (Java + JavaScript)
- ▶ Ausführliche Spezifikation
- ▶ Umfangreiches **Technology Compatibility Kit**
- ▶ Führt unterschiedlichste Bibliotheken zusammen
  - ▶ JavaRx
  - ▶ **akka streams**
  - ▶ Slick 3.0 (Datenbank FRM)
  - ▶ ...
- ▶ Außerdem in Arbeit: Spezifikationen für Netzwerkprotokolle

# Reactive Streams: Interfaces

- ▶ Publisher [O] – Stellt eine potentiell unendliche Sequenz von Elementen zur Verfügung. Die Produktionsrate richtet sich nach der Nachfrage der Subscriber
- ▶ Subscriber [I] – Konsumiert Elemente eines Pubilshers
- ▶ Subscription – Repräsentiert ein eins zu eins Abonnement eines Subscribers an einen Publisher
- ▶ Processor [I, O] – Ein Verarbeitungsschritt. Gleichzeitig Publisher und Subscriber

# Reactive Streams: 1. Publisher [T]

def subscribe(s: Subscriber [T]): Unit

- 1 The total number of onNext signals sent by a Publisher to a Subscriber MUST be less than or equal to the total number of elements requested by that Subscriber's Subscription at all times.
- 2 A Publisher MAY signal less onNext than requested and terminate the Subscription by calling onComplete or onError.
- 3 onSubscribe, onNext, onError and onComplete signaled to a Subscriber MUST be signaled sequentially (no concurrent notifications).
- 4 If a Publisher fails it MUST signal an onError.
- 5 If a Publisher terminates successfully (finite stream) it MUST signal an onComplete.
- 6 If a Publisher signals either onError or onComplete on a Subscriber, that Subscriber's Subscription MUST be considered cancelled.

# Reactive Streams: 1. Publisher [T]

```
def subscribe(s: Subscriber[T]): Unit
```

- 7 Once a terminal state has been signaled (`onError`, `onComplete`) it is REQUIRED that no further signals occur.
- 8 If a Subscription is cancelled its Subscriber MUST eventually stop being signaled.
- 9 `Publisher.subscribe` MUST call `onSubscribe` on the provided Subscriber prior to any other signals to that Subscriber and MUST return normally, except when the provided Subscriber is null in which case it MUST throw a `java.lang.NullPointerException` to the caller, for all other situations the only legal way to signal failure (or reject the Subscriber) is by calling `onError` (after calling `onSubscribe`).
- 10 `Publisher.subscribe` MAY be called as many times as wanted but MUST be with a different Subscriber each time.
- 11 A Publisher MAY support multiple Subscribers and decides whether each Subscription is unicast or multicast.

## Reactive Streams: 2. Subscriber [T]

```
def onComplete(): Unit
def onError(t: Throwable): Unit
def onNext(t: T): Unit
def onSubscribe(s: Subscription): Unit
```

- 1 A Subscriber **MUST** signal demand via `Subscription.request(long n)` to receive `onNext` signals.
- 2 If a Subscriber suspects that its processing of signals will negatively impact its Publisher's responsiveness, it is **RECOMMENDED** that it asynchronously dispatches its signals.
- 3 `Subscriber.onComplete()` and `Subscriber.onError(Throwable t)` **MUST NOT** call any methods on the `Subscription` or the `Publisher`.
- 4 `Subscriber.onComplete()` and `Subscriber.onError(Throwable t)` **MUST** consider the `Subscription` cancelled after having received the signal.
- 5 A Subscriber **MUST** call `Subscription.cancel()` on the given `Subscription` after an `onSubscribe` signal if it already has an active `Subscription`.

## Reactive Streams: 2. Subscriber [T]

```
def onComplete: Unit
def onError(t: Throwable): Unit
def onNext(t: T): Unit
def onSubscribe(s: Subscription): Unit
```

- 6 A Subscriber MUST call `Subscription.cancel()` if it is no longer valid to the Publisher without the Publisher having signaled `onError` or `onComplete`.
- 7 A Subscriber MUST ensure that all calls on its `Subscription` take place from the same thread or provide for respective external synchronization.
- 8 A Subscriber MUST be prepared to receive one or more `onNext` signals after having called `Subscription.cancel()` if there are still requested elements pending. `Subscription.cancel()` does not guarantee to perform the underlying cleaning operations immediately.
- 9 A Subscriber MUST be prepared to receive an `onComplete` signal with or without a preceding `Subscription.request(long n)` call.
- 10 A Subscriber MUST be prepared to receive an `onError` signal with or without a preceding `Subscription.request(long n)` call.

## Reactive Streams: 2. Subscriber [T]

```
def onComplete: Unit
def onError(t: Throwable): Unit
def onNext(t: T): Unit
def onSubscribe(s: Subscription): Unit
```

- ❶ A Subscriber MUST make sure that all calls on its `onXXX` methods happen-before the processing of the respective signals. I.e. the Subscriber must take care of properly publishing the signal to its processing logic.
- ❷ `Subscriber.onSubscribe` MUST be called at most once for a given Subscriber (based on object equality).
- ❸ Calling `onSubscribe`, `onNext`, `onError` or `onComplete` MUST return normally except when any provided parameter is null in which case it MUST throw a `java.lang.NullPointerException` to the caller, for all other situations the only legal way for a Subscriber to signal failure is by cancelling its Subscription. In the case that this rule is violated, any associated Subscription to the Subscriber MUST be considered as cancelled, and the caller MUST raise this error condition in a fashion that is adequate for the runtime environment.

## Reactive Streams: 3. Subscription

```
def cancel(): Unit
def request(n: Long): Unit
```

- 1 Subscription .request and Subscription .cancel MUST only be called inside of its Subscriber context. A Subscription represents the unique relationship between a Subscriber and a Publisher.
- 2 The Subscription MUST allow the Subscriber to call Subscription .request synchronously from within onNext or onSubscribe.
- 3 Subscription .request MUST place an upper bound on possible synchronous recursion between Publisher and Subscriber.
- 4 Subscription .request SHOULD respect the responsivity of its caller by returning in a timely manner.
- 5 Subscription .cancel MUST respect the responsivity of its caller by returning in a timely manner, MUST be idempotent and MUST be thread-safe.
- 6 After the Subscription is cancelled, additional Subscription .request(long n) MUST be NOPs.

## Reactive Streams: 3. Subscription

```
def cancel(): Unit  
def request(n: Long): Unit
```

- 7 After the Subscription is cancelled, additional Subscription.cancel() MUST be NOPs.
- 8 While the Subscription is not cancelled, Subscription.request(long n) MUST register the given number of additional elements to be produced to the respective subscriber.
- 9 While the Subscription is not cancelled, Subscription.request(long n) MUST signal onError with a java.lang.IllegalArgumentException if the argument is  $\leq 0$ . The cause message MUST include a reference to this rule and/or quote the full rule.
- 10 While the Subscription is not cancelled, Subscription.request(long n) MAY synchronously call onNext on this (or other) subscriber(s).
- 11 While the Subscription is not cancelled, Subscription.request(long n) MAY synchronously call onComplete or onError on this (or other) subscriber(s).

## Reactive Streams: 3. Subscription

```
def cancel(): Unit  
def request(n: Long): Unit
```

- 12 While the `Subscription` is not cancelled, `Subscription.cancel()` MUST request the `Publisher` to eventually stop signaling its `Subscriber`. The operation is NOT REQUIRED to affect the `Subscription` immediately.
- 13 While the `Subscription` is not cancelled, `Subscription.cancel()` MUST request the `Publisher` to eventually drop any references to the corresponding subscriber. Re-subscribing with the same `Subscriber` object is discouraged, but this specification does not mandate that it is disallowed since that would mean having to store previously cancelled subscriptions indefinitely.
- 14 While the `Subscription` is not cancelled, calling `Subscription.cancel` MAY cause the `Publisher`, if stateful, to transition into the shut-down state if no other `Subscription` exists at this point.

## Reactive Streams: 3. Subscription

```
def cancel(): Unit  
def request(n: Long): Unit
```

- 16 Calling `Subscription.cancel` MUST return normally. The only legal way to signal failure to a Subscriber is via the `onError` method.
- 17 Calling `Subscription.request` MUST return normally. The only legal way to signal failure to a Subscriber is via the `onError` method.
- 18 A `Subscription` MUST support an unbounded number of calls to `request` and MUST support a demand (sum requested - sum delivered) up to  $2^{63} - 1$  (`java.lang.Long.MAX_VALUE`). A demand equal or greater than  $2^{63} - 1$  (`java.lang.Long.MAX_VALUE`) MAY be considered by the Publisher as “effectively unbounded”.

## Reactive Streams: 4. Processor[I, O]

```
def onComplete: Unit
def onError(t: Throwable): Unit
def onNext(t: I): Unit
def onSubscribe(s: Subscription): Unit
def subscribe(s: Subscriber[O]): Unit
```

- 1 A Processor represents a processing stage — which is both a Subscriber and a Publisher and **MUST** obey the contracts of both.
- 2 A Processor **MAY** choose to recover an `onError` signal. If it chooses to do so, it **MUST** consider the `Subscription` cancelled, otherwise it **MUST** propagate the `onError` signal to its `Subscribers` immediately.

# Akka Streams

- ▶ Vollständige Implementierung der **Reactive Streams** Spezifikation
- ▶ Basiert auf **Datenflussgraphen** und **Materialisierern**
- ▶ Datenflussgraphen werden als **Aktornetzwerk** materialisiert

# Akka Streams - Grundkonzepte

**Datenstrom (Stream)** – Ein Prozess der Daten überträgt und transformiert

**Element** – Recheneinheit eines Datenstroms

**Back-Pressure** – Konsument signalisiert (asynchron) Nachfrage an Produzenten

**Verarbeitungsschritt (Processing Stage)** – Bezeichnet alle Bausteine aus denen sich ein Datenfluss oder Datenflussgraph zusammensetzt.

**Quelle (Source)** – Verarbeitungsschritt mit genau einem Ausgang

**Abfluss (Sink)** – Verarbeitungsschritt mit genau einem Eingang

**Datenfluss (Flow)** – Verarbeitungsschritt mit jeweils genau einem Ein- und Ausgang

**Ausführbarer Datenfluss (RunnableFlow)** – Datenfluss der an eine Quelle und einen Abfluss angeschlossen ist

# Akka Streams - Beispiel

```
implicit val system = ActorSystem("example")  
implicit val materializer = ActorFlowMaterializer()  
  
val source = Source(1 to 10)  
val sink = Sink.fold[Int, Int](0)(_ + _)  
val sum: Future[Int] = source runWith sink
```

# Datenflussgraphen

- ▶ Operatoren sind Abzweigungen im Graphen
- ▶ z.B. Broadcast (1 Eingang, n Ausgänge) und Merge (n Eingänge, 1 Ausgang)
- ▶ Scala DSL um Graphen darzustellen

```
val g = FlowGraph.closed() { implicit builder =>
  val in = source
  val out = sink
  val bcast = builder.add(Broadcast[Int](2))
  val merge = builder.add(Merge[Int](2))
  val f1, f2, f3, f4 = Flow[Int].map(_ + 10)

  in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out
      bcast ~> f4 ~> merge
}
```

# Operatoren in Datenflussgraphen

- ▶ Auffächern
  - ▶ Broadcast[T] – Verteilt eine Eingabe an  $n$  Ausgänge
  - ▶ Balance[T] – Teilt Eingabe gleichmäßig unter  $n$  Ausgängen auf
  - ▶ UnZip[A,B] – Macht aus [(A,B)]-Strom zwei Ströme [A] und [B]
  - ▶ FlexiRoute[In] – DSL für eigene Fan-Out Operatoren
- ▶ Zusammenführen
  - ▶ Merge[In] – Vereinigt  $n$  Ströme in einem
  - ▶ MergePreferred[In] – Wie Merge, hat aber einen präferierten Eingang
  - ▶ ZipWith[A,B,...,Out] – Fasst  $n$  Eingänge mit einer Funktion  $f$  zusammen
  - ▶ Zip[A,B] – *ZipWith* mit zwei Eingängen und  $f = (\_, \_)$
  - ▶ Concat[A] – Sequentialisiert zwei Ströme
  - ▶ FlexiMerge[Out] – DSL für eigene Fan-In Operatoren

# Partielle Datenflussgraphen

- ▶ Datenflussgraphen können partiell sein:

```
val pickMaxOfThree = FlowGraph.partial() {  
  implicit builder =>  
  
  val zip1 = builder.add(ZipWith[Int, Int, Int](math.max))  
  val zip2 = builder.add(ZipWith[Int, Int, Int](math.max))  
  
  zip1.out ~> zip2.in0  
  
  UniformFanInShape(zip2.out, zip1.in0, zip1.in1, zip2.in1)  
}
```

- ▶ Offene Anschlüsse werden später belegt

# Sources, Sinks und Flows als Datenflussgraphen

- ▶ Source — Graph mit genau einem offenen Ausgang

```
Source() { implicit builder =>
  outlet
}
```

- ▶ Sink — Graph mit genau einem offenen Eingang

```
Sink() { implicit builder =>
  inlet
}
```

- ▶ Flow — Graph mit jeweils genau einem offenen Ein- und Ausgang

```
Flow() { implicit builder =>
  (inlet, outlet)
}
```

# Zyklische Datenflussgraphen

- ▶ Zyklen in Datenflussgraphen sind erlaubt:

```
val input = Source(Stream.continually(readLine()))

val flow = FlowGraph.closed() { implicit builder =>
  val merge = builder.add(Merge[String](2))
  val bcast = builder.add(Broadcast[String](2))
  val print = Flow.map{s => println(s); s}

  input ~> merge ~> print ~> bcast ~> Sink.ignore
           merge      <~      bcast
}

```

- ▶ Hört nach kurzer Zeit auf etwas zu tun — Wieso?

# Zyklische Datenflussgraphen

► Besser:

```
val input = Source(Stream.continually(readLine()))

val flow = FlowGraph.closed() { implicit builder =>
  val merge = builder.add(Merge[String](2))
  val bcast = builder.add(Broadcast[String](2))
  val print = Flow.map{s => println(s); s}
  val buffer = Flow.buffer(10, OverflowStrategy.dropHead)

  input ~> merge ~> print ~> bcast ~> Sink.ignore
      merge <~ buffer <~ bcast
}
```

# Pufferung

- ▶ Standardmäßig werden bis zu **16 Elemente** gepuffert, um parallele Ausführung von Streams zu erreichen.
- ▶ Danach: Backpressure

```
Source(1 to 3)
  .alsoTo(Sink.foreach(i => println(s"A: $i")))
  .alsoTo(Sink.foreach(i => println(s"B: $i")))
  .alsoTo(Sink.foreach(i => println(s"C: $i")))
  .to(Sink.foreach(i => println(s"D: $i")))
  .run()
```

- ▶ Ausgabe nicht deterministisch, wegen paralleler Ausführung
- ▶ Puffergrößen können angepasst werden (Systemweit, Materialisierer, Verarbeitungsschritt)

# Fehlerbehandlung

- ▶ Standardmäßig führen Fehler zum Abbruch:

```
val source = Source(0 to 5).map(100 / _)
val result = source.runWith(Sink.fold(0)(_ +_))
```

- ▶ `result = Future(Failure(ArithmeticException))`
- ▶ Materialisierer kann mit Supervisor konfiguriert werden:

```
val decider: Supervisor.Decider = {
  case _ : ArithmeticException => Resume
  case _ => Stop
}
implicit val materializer = ActorFlowMaterializer(
  ActorFlowMaterializerSettings(system)
  .withSupervisionStrategy(decider))
```

- ▶ `result = Future(Success(228))`

# Integration mit Aktoren - ActorPublisher

- ▶ ActorPublisher ist ein Aktor, der als Source verwendet werden kann.

```
class MyActorPublisher extends ActorPublisher[String] {  
  def receive = {  
    case Request(n) =>  
      for (i ← 1 to n) onNext("Hallo")  
    case Cancel =>  
      context.stop(self)  
  }  
}
```

```
Source.actorPublisher(Props[MyActorPublisher])
```

# Integration mit Aktoren - ActorSubscriber

- ▶ ActorSubscriber ist ein Aktor, der als Sink verwendet werden kann.

```
class MyActorSubscriber extends ActorSubscriber {  
  def receive = {  
    case OnNext(elem) =>  
      log.info("received {}", elem)  
    case OnError(e) =>  
      throw e  
    case OnComplete =>  
      context.stop(self)  
  }  
}
```

```
Source.actorPublisher(Props[MyActorPublisher])
```

# Integration für einfache Fälle

- ▶ Für einfache Fälle gibt es `Source.actorRef` und `Sink.actorRef`

```
val source: Source[Foo, ActorRef] = Source.actorRef[Foo](  
  bufferSize = 10,  
  overflowStrategy = OverflowStrategy.backpressure)
```

```
val sink: Sink[Foo, Unit] = Sink.actorRef[Foo](  
  ref = myActorRef,  
  onCompleteMessage = Bar)
```

- ▶ Problem: Sink hat kein Backpressure. Wenn der Aktor nicht schnell genug ist, explodiert alles.

## Anwendung: akka-http

- ▶ Minimale HTTP-Bibliothek (Client und Server)
- ▶ Basierend auf *akka-streams* — reaktiv
- ▶ From scratch — **keine Altlasten**
- ▶ **Kein Blocking** — Schnell
- ▶ Scala DSL für Routen-Definition
- ▶ Scala DSL für Webaufrufe
- ▶ Umfangreiche Konfigurationsmöglichkeiten

# Low-Level Server API

- ▶ HTTP-Server wartet auf Anfragen:

```
Source[IncomingConnection, Future[ServerBinding]]
```

```
val server = Http.bind(interface = "localhost", port = 8080)
```

- ▶ Zu jeder Anfrage gibt es eine Antwort:

```
val requestHandler: HttpRequest => HttpResponse = {  
  case HttpRequest(GET, Uri.Path("/ping"), _, _, _) =>  
    HttpResponse(entity = "PONG!")  
}  
  
val serverSink =  
  Sink.foreach(_.handleWithSyncHandler(requestHandler))  
  
serverSource.to(serverSink)
```

# High-Level Server API

► Minimalbeispiel:

```
implicit val system = ActorSystem("example")
implicit val materializer = ActorFlowMaterializer()

val routes = path("ping") {
  get {
    complete { <h1>PONG!</h1> }
  }
}

val binding =
  Http().bindAndHandle(routes, "localhost", 8080)
```

# HTTP

- ▶ HTTP ist ein Protokoll aus den frühen 90er Jahren.
- ▶ Grundidee: Client sendet **Anfragen** an Server, Server **antwortet**
- ▶ Verschiedene Arten von Anfragen
  - ▶ GET — Inhalt abrufen
  - ▶ POST — Inhalt zum Server übertragen
  - ▶ PUT — Resource unter bestimmter URI erstellen
  - ▶ DELETE — Resource löschen
  - ▶ ...
- ▶ Antworten mit Statuscode. z.B.:
  - ▶ 200 — Ok
  - ▶ 404 — Not found
  - ▶ 501 — Internal Server Error
  - ▶ ...

# Das Server-Push Problem

- ▶ HTTP basiert auf der Annahme, dass der Webclient den (statischen) Inhalt **bei Bedarf** anfragt.
- ▶ Moderne Webanwendungen sind alles andere als statisch.
- ▶ Workarounds des letzten Jahrzehnts:
  - ▶ **AJAX** — Eigentlich *Asynchronous JavaScript and XML*, heute eher **AJAJ**  
— Teile der Seite werden dynamisch ersetzt.
  - ▶ **Polling** — "Gibt's etwas Neues?", "Gibt's etwas Neues?", ...
  - ▶ **Comet** — Anfrage mit langem Timeout wird erst beantwortet, wenn es etwas Neues gibt.
  - ▶ **Chunked Response** — Server antwortet stückchenweise

# WebSockets

- ▶ TCP-Basiertes **bidirektionales** Protokoll für Webanwendungen
- ▶ Client öffnet nur **einmal** die Verbindung
- ▶ Server und Client können **jederzeit** Daten senden
- ▶ Nachrichten ohne Header (1 Byte)
- ▶ **Ähnlich** wie Aktoren:
  - ▶ JavaScript Client sequentiell mit lokalem Zustand ( $\approx$  Actor)
  - ▶ `WebSocket.onmessage`  $\approx$  `Actor.receive`
  - ▶ `WebSocket.send(msg)`  $\approx$  `sender ! msg`
  - ▶ `WebSocket.onclose`  $\approx$  `Actor.postStop`
  - ▶ Außerdem `onerror` für Fehlerbehandlung.

# WebSockets in akka-http

- ▶ WebSockets ist ein `Flow[Message,Message,Unit]`
- ▶ Können über bidirektional Flows gehandhabt werden
  - ▶ `BidiFlow[-I1, +O1,-I2, +O2, +Mat]` – zwei Eingänge, zwei Ausgänge: Serialisieren und deserialisieren.
- ▶ Beispiel:

```
def routes = get {  
  path("ping")(handleWebsocketMessages(wsFlow))  
}  
  
def wsFlow: Flow[Message,Message,Unit] =  
  BidiFlow.fromFunctions(serialize, deserialize)  
    .join(Flow.collect {  
      case Ping => Pong  
    })
```

# Zusammenfassung

- ▶ Die Konstruktoren in der Rx Bibliothek wenden viel **Magie** an um Gesetze einzuhalten
- ▶ Fehlerbehandlung durch Kombinatoren ist einfach zu implementieren
- ▶ Observables eignen sich nur bedingt um **Back Pressure** zu implementieren, da Kontrollfluss unidirektional konzipiert.
- ▶ Die *Reactive Streams*-Spezifikation beschreibt ein minimales Interface für Ströme mit Back Pressure
- ▶ Für die Implementierung sind Aktoren sehr gut geeignet ⇒ akka streams

# Zusammenfassung

- ▶ **Datenflussgraphen** repräsentieren reaktive Berechnungen
  - ▶ Geschlossene Datenflussgraphen sind ausführbar
  - ▶ Partielle Datenflussgraphen haben **unbelegte** ein oder ausgänge
  - ▶ **Zyklische** Datenflussgraphen sind erlaubt
- ▶ Puffer sorgen für **parallele Ausführung**
- ▶ Supervisor können bestimmte Fehler ignorieren
- ▶ *akka-stream* kann einfach mit *akka-actor* integriert werden
- ▶ Anwendungsbeispiel: *akka-http*
  - ▶ Low-Level API: Request =>Response
  - ▶ HTTP ist **pull basiert**
  - ▶ WebSockets sind **bidirektional** → Flow

## Bonusfolie: WebWorkers

- ▶ JavaScript ist singlethreaded.
- ▶ Bibliotheken machen sich keinerlei Gedanken über Race-Conditions.
- ▶ Workaround: Aufwändige Berechnungen werden gestückelt, damit die Seite responsiv bleibt.
- ▶ Lösung: HTML5-WebWorkers (Alle modernen Browser)
  - ▶ `new Worker(file)` startet neuen Worker
  - ▶ Kommunikation über `postMessage`, `onmessage`, `onerror`, `onclose`
  - ▶ Einschränkung: Kein Zugriff auf das DOM — lokaler Zustand
  - ▶ WebWorker können weitere WebWorker erzeugen
  - ▶ *"Poor-Man's Actors"*

Reaktive Programmierung  
Vorlesung 12 vom 12.06.19  
Funktional-Reaktive Programmierung

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Das Tagemenü

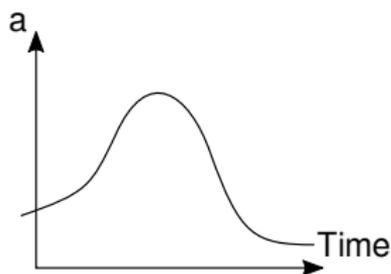
- ▶ **Funktional-Reaktive Programmierung** (FRP) ist **rein** funktionale, reaktive Programmierung.
- ▶ Sehr **abstraktes** Konzept — im Gegensatz zu Observables und Aktoren.
- ▶ Literatur: Paul Hudak, **The Haskell School of Expression**, Cambridge University Press 2000, Kapitel 13, 15, 17.
- ▶ Andere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

# FRP in a Nutshell

Zwei Basiskonzepte:

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

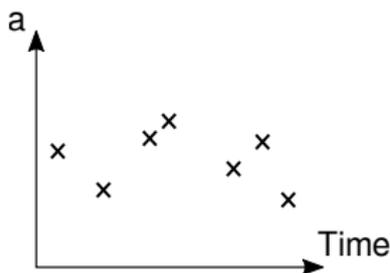
```
type Time = Float
type Behaviour a = Time → a
```



- ▶ Beispiel: Position eines Objektes

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

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



- ▶ 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
```

- ▶ Was passiert hier?

- ▶ `untilB` und `switch` kombinieren Verhalten:

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

- ▶  $\Rightarrow\Rightarrow$  ist `map` für Ereignisse:

```
Event fe  $\Rightarrow\Rightarrow$  f = Event (map (fmap f) o fe)  
e → v = e  $\Rightarrow\Rightarrow$   $\lambda\_ \rightarrow v$ 
```

# Reaktive Animationen: Verhaltensänderung

- ▶ Beispiel: auf Knopfdruck Farbe ändern:

```
color3 = white 'switch' (key ==>> λc →
```

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

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

- ▶  $\Rightarrow\Rightarrow$  ist map für Ereignisse:

```
Event fe ==>> f = Event (map (fmap f) o 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)
```

## ► Nützliche Funktionen:

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

# Der Springende Ball

```
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))
```

- ▶ **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.

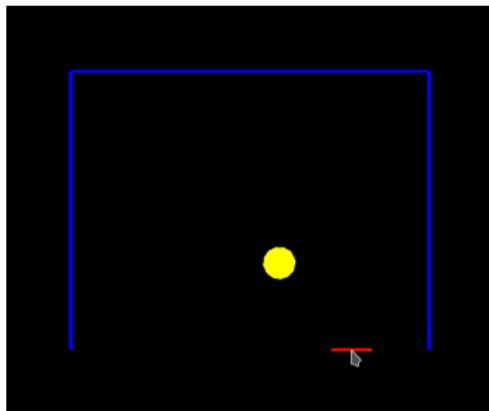
▶ Das Paddel:

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

Reaktive Programmierung  
Vorlesung 13 vom 19.06.19  
Software Transactional Memory

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Heute gibt es:

- ▶ Motivation: Nebenläufigkeit tut not!

# Heute gibt es:

- ▶ Motivation: Nebenläufigkeit tut not!
- ▶ Einen fundamental anderen Ansatz nebenläufiger Datenmodifikation
  - ▶ Keine **Locks** und **Conditional variables**
  - ▶ Sondern: **Transaktionen!**
  - ▶ Software transactional memory (STM)
- ▶ Implementierung in Haskell: `atomically`, `retry`, `orElse`
- ▶ Fallbeispiele:
  - ▶ Puffer: Reader-/Writer
  - ▶ Speisende Philosophen
  - ▶ Weihnachtlich: das Santa Claus Problem

# Aktueller Stand der Technik

- ▶ C: Locks und conditional variables

```
pthread_mutex_lock(&mutex)
pthread_mutex_unlock(&mutex)
pthread_cond_wait(&cond, &mutex)
pthread_cond_broadcast(&cond)
```

- ▶ Java (Scala): Monitore

```
synchronized public void workOnSharedData() { ... }
```

- ▶ Haskell: MVars

```
newMVar :: a → IO (MVar a)
takeMVar :: MVar a → IO a
putMVar :: MVar a → a → IO ()
```

# Stand der Technik: Locks und Conditional variables

- ▶ Grundlegende Idee: Zugriff auf gemeinsame Ressourcen nur innerhalb **kritischer Abschnitte**
  - ① Vor Betreten um Erlaubnis fragen (Lock an sich reißen)
  - ② Arbeiten
  - ③ Beim Verlassen Meldung machen (Lock freigeben)

# Stand der Technik: Locks und Conditional variables

- ▶ Grundlegende Idee: Zugriff auf gemeinsame Ressourcen nur innerhalb **kritischer Abschnitte**
  - ① Vor Betreten um Erlaubnis fragen (Lock an sich reißen)
  - ② Arbeiten
  - ③ Beim Verlassen Meldung machen (Lock freigeben)
- ▶ Verfeinerung: Auf Eintreten von Bedingungen warten (Kommunikation)
  - ① Im kritischen Abschnitt **schlafengehen**, wenn Bedingung nicht erfüllt (Lock freigeben!)
  - ② Andere Threads machen Bedingung wahr und **melden** dies
  - ③ Sobald Lock verfügbar: **aufwachen**

# Stand der Technik: Locks und Conditional variables

- ▶ Grundlegende Idee: Zugriff auf gemeinsame Ressourcen nur innerhalb **kritischer Abschnitte**
  - ① Vor Betreten um Erlaubnis fragen (Lock an sich reißen)
  - ② Arbeiten
  - ③ Beim Verlassen Meldung machen (Lock freigeben)
- ▶ Verfeinerung: Auf Eintreten von Bedingungen warten (Kommunikation)
  - ① Im kritischen Abschnitt **schlafengehen**, wenn Bedingung nicht erfüllt (Lock freigeben!)
  - ② Andere Threads machen Bedingung wahr und **melden** dies
  - ③ Sobald Lock verfügbar: **aufwachen**
- ▶ Semaphoren & Monitore bauen essentiell auf demselben Prinzip auf

# Kritik am Lock-basierten Ansatz

- ▶ Kritische Abschnitte haben eine pessimistische Lebenseinstellung:
  - ▶ Möglicherweise will ein anderer Thread gerade dieselben Daten verändern
  - ▶ Darum: Sperrung des Abschnitts in **jedem** Fall
  - ▶ Möglicherweise gar nicht nötig: Effizienz?

# Kritik am Lock-basierten Ansatz

- ▶ Kritische Abschnitte haben eine pessimistische Lebenseinstellung:
  - ▶ Möglicherweise will ein anderer Thread gerade dieselben Daten verändern
  - ▶ Darum: Sperrung des Abschnitts in **jedem** Fall
  - ▶ Möglicherweise gar nicht nötig: Effizienz?
- ▶ Gefahr des Deadlocks:
  - ▶ A betritt kritischen Abschnitt  $S_1$ ; gleichzeitig betritt B  $S_2$
  - ▶ A will nun  $S_2$  betreten, während es Lock für  $S_1$  hält
  - ▶ B will dasselbe mit  $S_1$  tun.
  - ▶ The rest is silence. . .

# Kritik am Lock-basierten Ansatz

- ▶ Kritische Abschnitte haben eine pessimistische Lebenseinstellung:
  - ▶ Möglicherweise will ein anderer Thread gerade dieselben Daten verändern
  - ▶ Darum: Sperrung des Abschnitts in **jedem** Fall
  - ▶ Möglicherweise gar nicht nötig: Effizienz?
- ▶ Gefahr des Deadlocks:
  - ▶ A betritt kritischen Abschnitt  $S_1$ ; gleichzeitig betritt B  $S_2$
  - ▶ A will nun  $S_2$  betreten, während es Lock für  $S_1$  hält
  - ▶ B will dasselbe mit  $S_1$  tun.
  - ▶ The rest is silence. . .
- ▶ Richtige Granularität schwer zu bestimmen
  - ▶ Grobkörnig: ineffizient; feinkörnig: schwer zu analysieren

## Kritik am Lock-basierten Ansatz (2)

- ▶ Größtes Problem: **Lock-basierte Programme sind nicht komponierbar!**
  - ▶ Korrekte Einzelbausteine können zu fehlerhaften Programmen zusammengesetzt werden
- ▶ Klassisches Beispiel: Übertragung eines Eintrags von einer Map in eine andere
  - ▶ Map-Bücherei explizit thread-safe, d.h. nebenläufiger Zugriff sicher
  - ▶ Implementierung der Übertragung:

```
transferItem item c1 c2 = do
  delete c1 item
  insert c2 item
```

- ▶ Problem: Zwischenzustand, in dem item in keiner Map ist
- ▶ Plötzlich doch wieder Locks erforderlich! Welche?

## Kritik am Lock-basierten Ansatz (3)

- ▶ Ein ähnliches Argument gilt für Komposition von Ressourcen-Auswahl:
- ▶ **Mehrfachauswahl** in Posix (Unix/Linux/Mac OS X):
  - ▶ `select ()` wartet auf mehrere I/O-Kanäle gleichzeitig
  - ▶ Kehrt zurück sobald mindestens einer verfügbar
- ▶ Beispiel: Prozeduren `foo()` und `bar()` warten auf unterschiedliche Ressourcen(-Mengen):

```
void foo(void) {  
...  
    select(k1, r1, w1, e1, &t1);  
...  
}
```

```
void bar(void) {  
...  
    select(k2, r2, w2, e2, &t2);  
...  
}
```

- ▶ **Keine** Möglichkeit, `foo()` und `bar()` zu komponieren, so dass bspw. auf `r1` und `r2` gewartet wird

# STM: software transactional memory

## Grundidee: Drei Eigenschaften

- 1 Transaktionen sind **atomar**
- 2 Transaktionen sind **bedingt**
- 3 Transaktionen sind **komponierbar**

- ▶ Eigenschaften entsprechen Operationen:
  - ▶ Atomare Transaktion
  - ▶ Bedingte Transaktion
  - ▶ Komposition von Transaktionen
- ▶ Typ STM von Transaktionen (Monad)
- ▶ Typsystem stellt sicher, dass Transaktionen reversibel sind

# Transaktionen sind atomar

- ▶ Ein **optimistischer** Ansatz zur nebenläufigen Programmierung
- ▶ Prinzip der **Transaktionen** aus Datenbank-Domäne entliehen
- ▶ Kernidee: `atomically ( ... )` Blöcke werden **atomar** ausgeführt
  - ▶ (Speicher-)änderungen erfolgen entweder vollständig oder gar nicht
  - ▶ Im letzteren Fall: Wiederholung der Ausführung
  - ▶ Im Block: konsistente Sicht auf Speicher
  - ▶ A(tomicity) und I(solation) aus ACID
- ▶ Damit **deklarative** Formulierung des Elementtransfers möglich:

```
atomically $  
do { removeFrom c1 item; insertInto c2 item }
```

# Blockieren / Warten (blocking)

- ▶ Atomarität allein reicht nicht: STM muss **Synchronisation** von Threads ermöglichen
- ▶ Klassisches Beispiel: Produzenten + Konsumenten:
  - ▶ Wo nichts ist, kann nichts konsumiert werden
  - ▶ Konsument **wartet** auf Ergebnisse des Produzenten

```
consumer buf = do
  item ← getItem buf
  doSomethingWith item
```

- ▶ getItem blockiert, wenn keine Items verfügbar

# Transaktionen sind bedingt

- ▶ Kompositionales “Blockieren” mit `retry`
- ▶ Idee: ist notwendige Bedingung innerhalb einer Transaktion nicht erfüllt, wird Transaktion abgebrochen und **erneut versucht**

```
atomically $ do
  ...
  if (Buffer.empty buf) then retry else...
```

- ▶ Sinnlos, sofern andere Threads Zustand nicht verändert haben!
- ▶ Daher: warten (worauf?)

# Transaktionen sind bedingt

- ▶ Kompositionales “Blockieren” mit `retry`
- ▶ Idee: ist notwendige Bedingung innerhalb einer Transaktion nicht erfüllt, wird Transaktion abgebrochen und **erneut versucht**

```
atomically $ do
  ...
  if (Buffer.empty buf) then retry else...
```

- ▶ Sinnlos, sofern andere Threads Zustand nicht verändert haben!
- ▶ Daher: warten
  - ▶ Auf Änderung an in Transaktion **gelesenen** Variablen!
  - ▶ Genial: System verantwortlich für Verwaltung der Aufweckbedingung
- ▶ Keine lost wakeups, keine händische Verwaltung von conditional variables

# Transaktionen sind kompositional

- ▶ Dritte Zutat für erfolgreiches kompositionales Multithreading: **Auswahl** möglicher Aktionen
- ▶ Beispiel: Event-basierter Webserver liest Daten von mehreren Verbindungen
- ▶ Kombinator `orElse` ermöglicht linksorientierte Auswahl (ähnlich `||`):

```
webServer = do
  ...
  news ← atomically $ orElse spiegelRSS cnnRSS
  req ← atomically $ foldr1 orElse clients
  ...
```

- ▶ Wenn linke Transaktion misslingt, wird rechte Transaktion versucht

# Einschränkungen an Transaktionen

- ▶ Transaktionen dürfen nicht beliebige Seiteneffekte haben
  - ▶ Nicht jeder reale Seiteneffekt lässt sich rückgängig machen:
  - ▶ Bsp: `atomically $ do { if (done) delete_file (important); S2 }`
  - ▶ Idee: Seiteneffekte werden auf **Transaktionsspeicher** beschränkt
- ▶ Ideal: Trennung wird **statisch** erzwungen
  - ▶ In Haskell: Trennung im **Typsystem**
  - ▶ IO-Aktionen vs. STM-Aktionen (Monaden)
  - ▶ Innerhalb der STM-Monade nur **reine** Berechnungen (kein IO!)
  - ▶ STM Monade erlaubt **Transaktionsreferenzen** TVar (ähnlich IORef)

# Software Transactional Memory in Haskell

- ▶ Kompakte Schnittstelle:

```
newtype STM a
instance Monad STM
atomically :: STM a → IO a
retry      :: STM a
orElse     :: STM a → STM a → STM a

data TVar
newTVar    :: a → STM (TVar a)
readTVar   :: TVar a → STM a
writeTVar  :: TVar a → a → STM ()
```

- ▶ Passt auf eine Folie!

# Gedankenmodell für atomare Speicheränderungen

## Mögliche Implementierung

- ▶ Thread  $T_1$  im atomically-Block nimmt keine Speicheränderungen vor, sondern in schreibt Lese-/Schreiboperationen in **Transaktions-Log**
- ▶ Leseoperationen konsultieren zunächst Log
- ▶ Beim Verlassen des atomically-Blocks:
  1. **globales Lock** greifen
  2. konsistenter Speicher gelesen?
  - 3t. änderungen einpflegen
  - 4t. Lock freigeben
  - 3f. änderungen verwerfen
  - 4f. Lock freigeben, Block wiederholen

# Gedankenmodell für atomare Speicheränderungen

## Mögliche Implementierung

- ▶ Thread  $T_1$  im atomically-Block nimmt keine Speicheränderungen vor, sondern in schreibt Lese-/Schreiboperationen in **Transaktions-Log**
- ▶ Leseoperationen konsultieren zunächst Log
- ▶ Beim Verlassen des atomically-Blocks:
  1. **globales Lock** greifen
  2. konsistenter Speicher gelesen?
  - 3t. änderungen einpflegen
  - 4t. Lock freigeben
  - 3f. änderungen verwerfen
  - 4f. Lock freigeben, Block wiederholen

## Konsistenter Speicher

- ▶ Jede zugriffene Speicherstelle hat zum Prüfzeitpunkt denselben Wert wie beim **ersten** Lesen

# Puffer mit STM: Modul MyBuffer

- ▶ Erzeugen eines neuen Puffers: newTVar mit leerer Liste

```
newtype Buf a = B (TVar [a])
```

```
new :: STM (Buf a)
```

```
new = do tv ← newTVar []  
      return $ B tv
```

- ▶ Elemente zum Puffer hinzufügen (immer möglich):
  - ▶ Puffer lesen, Element hinten anhängen, Puffer schreiben

```
put :: Buf a → a → STM ()
```

```
put (B tv) x = do xs ← readTVar tv  
                writeTVar tv (xs ++ [x])
```

## Puffer mit STM: Modul MyBuffer (2)

- ▶ Element herausnehmen: Möglicherweise keine Elemente vorhanden!
  - ▶ Wenn kein Element da, **wiederholen**
  - ▶ Ansonsten: Element entnehmen, Puffer verkleinern

```
get :: Buf a → STM a
get (B tv) = do xs ← readTVar tv
             case xs of
               [] → retry
               (y:xs') → do writeTVar tv xs'
                          return y
```

## Puffer mit STM: Anwendungsbeispiel

```
useBuffer :: IO ()
useBuffer = do
  b ← atomically $ new
  forkIO $ forever $ do
    n ← randomRIO(1,5)
    threadDelay (n*106)
    t ← getCurrentTime
    mapM_ (λx → atomically $ put b $ show x) (replicate n t)
  forever $ do x ← atomically $ get b
               putStrLn $ x
```

# Anwendungsbeispiel Philosophers.hs

- ▶ Gesetzlich vorgeschrieben als Beispiel
- ▶ Gabel als TVar mit Zustand Down oder Taken, und einer Id:

```
data FS = Down | Taken deriving Eq  
data Fork = Fork { fid :: Int, tvar :: TVar FS }
```

- ▶ Am Anfang liegt die Gabel auf dem Tisch:

```
newFork :: Int → IO Fork  
newFork i = atomically $ do  
  f ← newTVar Down  
  return $ Fork i f
```

Uses code from  
[http://rosettacode.org/wiki/Dining\\_philosophers#Haskell](http://rosettacode.org/wiki/Dining_philosophers#Haskell)

# Anwendungsbeispiel Philosophers.hs

- ▶ Transaktionen:
- ▶ Gabel aufnehmen— kann fehlschlagen

```
takeFork :: Fork → STM ()  
takeFork (Fork _ f) = do  
  s ← readTVar f  
  when (s == Taken) retry  
  writeTVar f Taken
```

- ▶ Gabel ablegen— gelingt immer

```
releaseFork :: Fork → STM ()  
releaseFork (Fork _ f) = writeTVar f Down
```

# Anwendungsbeispiel Philosophers.hs

- ▶ Ein Philosoph bei der Arbeit (putStrLn elidiert):

```
runPhilosopher :: String → (Fork, Fork) → IO ()
runPhilosopher name (left, right) = forever $ do
  delay ← randomRIO (1, 50)
  threadDelay (delay * 100000) — 1 to 5 seconds
  atomically $ do {takeFork left; takeFork right}
  delay ← randomRIO (1, 50)
  threadDelay (delay * 100000) — 1 to 5 seconds.
  atomically $ do {releaseFork left; releaseFork right}
```

- ▶ Atomare Transaktionen: beide Gabeln aufnehmen, beide Gabeln ablegen

# Santa Claus Problem

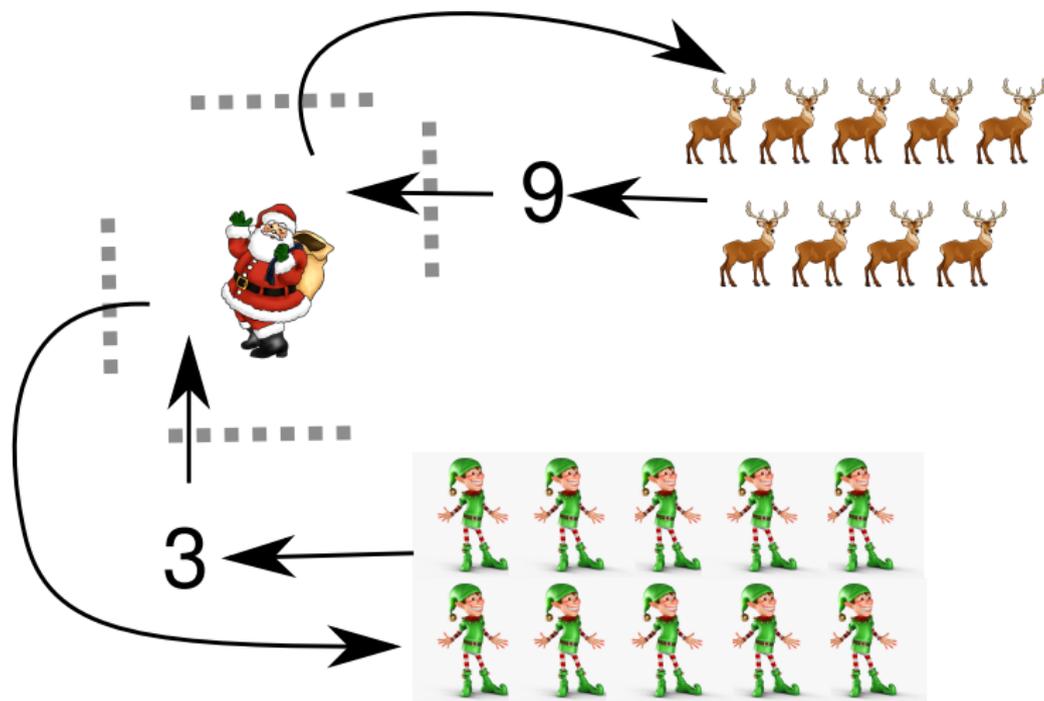
Ein modernes Nebenläufigkeitsproblem:

*Santa **repeatedly sleeps** until wakened by either all of his nine reindeer, [. . .], or by a group of three of his ten elves. If **awakened** by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them ([. . .]). If awakened by a group of elves, he shows each of the group into his study, consults with them [. . .], and finally shows them each out ([. . .]). Santa should give **priority** to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.*

aus:

J. A. Trono, *A new exercise in concurrency*, SIGCSE Bulletin, 26:8–10, 1994.

# Santa Claus Problem, veranschaulicht



# Lösungsstrategie

- ▶ Modellieren jede Elfe, jedes Rentier, und den Weihnachtsmann als **Faden**
- ▶ Santa wartet und koordiniert, sobald genügend “Teilnehmer” vorhanden
- ▶ Elfen und Rentiere tun fortwährend dasselbe: Sammeln, arbeiten, herumstehen

# Lösungsstrategie

- ▶ Modellieren jede Elfe, jedes Rentier, und den Weihnachtsmann als **Faden**
  - ▶ Santa wartet und koordiniert, sobald genügend "Teilnehmer" vorhanden
  - ▶ Elfen und Rentiere tun fortwährend dasselbe: Sammeln, arbeiten, herumstehen
- ▶ Verwenden **Gruppen** (Group) als Sammelplätze für Elfen und Rentiere
  - ▶ 3er-Gruppe für Elfen, 9er-Gruppe für Rentiere
  - ▶ Santa wacht auf, sobald Gruppe vollzählig

# Lösungsstrategie

- ▶ Modellieren jede Elfe, jedes Rentier, und den Weihnachtsmann als **Faden**
  - ▶ Santa wartet und koordiniert, sobald genügend “Teilnehmer” vorhanden
  - ▶ Elfen und Rentiere tun fortwährend dasselbe: Sammeln, arbeiten, herumstehen
- ▶ Verwenden **Gruppen** (Group) als Sammelplätze für Elfen und Rentiere
  - ▶ 3er-Gruppe für Elfen, 9er-Gruppe für Rentiere
  - ▶ Santa wacht auf, sobald Gruppe vollzählig
- ▶ **Gatterpaare** (Gate) erlauben koordinierten Eintritt in Santas Reich
  - ▶ Stellt geordneten Ablauf sicher (kein überholen übereifriger Elfen)

## Vorarbeiten: (Debug-)Ausgabe der Aktionen in Puffer

```
{- Actions of elves and deer -}  
meetInStudy :: Buf → Int → IO ()  
meetInStudy buf id = bput buf $  
  "Elf "+show id+" meeting in the study"  
  
deliverToys :: Buf → Int → IO ()  
deliverToys buf id = bput buf $  
  "Reindeer "+show id+" delivering toys"
```

- ▶ Puffer wichtig, da `putStrLn` nicht thread-sicher!
- ▶ Lese-Thread liest Daten aus `Buf` und gibt sie sequentiell an `stdout` aus

# Arbeitsablauf von Elfen und Rentieren

- Generisch: Tun im Grunde dasselbe, parametrisiert über task

```
helper1 :: Group → IO () → IO ()  
helper1 grp task = do  
  (inGate, outGate) ← joinGroup grp  
  passGate inGate  
  task  
  passGate outGate
```

```
elf1, reindeer1 :: Buf → Group → Int → IO ()  
elf1 buf grp elfId =  
  helper1 grp (meetInStudy buf elfId)  
reindeer1 buf grp reinId =  
  helper1 grp (deliverToys buf reinId)
```

# Gatter: Erzeugung, Durchgang

- ▶ Gatter haben aktuelle sowie Gesamtkapazität
- ▶ Anfänglich leere Aktualkapazität (Santa kontrolliert Durchgang)

```
data Gate = Gate Int (TVar Int)

newGate :: Int → STM Gate
newGate n = do tv ← newTVar 0
              return $ Gate n tv

passGate :: Gate → IO ()
passGate (Gate n tv) =
  atomically $ do c ← readTVar tv
                  check (c > 0)
                  writeTVar tv (c - 1)
```

# Nützliches Design Pattern: check

- ▶ Nebenläufiges assert:

```
check :: Bool → STM ()  
check b | b = return ()  
        | not b = retry
```

- ▶ Bedingung `b` muss gelten, um weiterzumachen
- ▶ Im STM-Kontext: wenn Bedingung nicht gilt: wiederholen
- ▶ Nach `check`: Annahme, dass `b` gilt
  
- ▶ Wunderschön deklarativ!

# Santas Aufgabe: Gatter betätigen

- ▶ Wird ausgeführt, sobald sich eine Gruppe versammelt hat
- ▶ **Zwei** atomare Schritte
  - ▶ Kapazität hochsetzen auf Maximum
  - ▶ Warten, bis Aktualkapazität auf 0 gesunken ist, d.h. alle Elfen/Rentiere das Gatter passiert haben

```
operateGate :: Gate → IO ()
operateGate (Gate n tv) = do
  atomically $ writeTVar tv n
  atomically $ do c ← readTVar tv
                check (c == 0)
```

- ▶ Beachte: Mit nur einem `atomically` wäre diese Operation niemals ausführbar! (Starvation)

## Gruppen: Erzeugung, Beitritt

```
data Group = Group Int (TVar (Int, Gate, Gate))
```

```
newGroup :: Int → IO Group
```

```
newGroup n = atomically $ do
```

```
  g1 ← newGate n
```

```
  g2 ← newGate n
```

```
  tv ← newTVar (n, g1, g2)
```

```
  return $ Group n tv
```

```
joinGroup :: Group → IO (Gate, Gate)
```

```
joinGroup (Group n tv) =
```

```
  atomically $ do (k, g1, g2) ← readTVar tv
```

```
    check (k > 0)
```

```
    writeTVar tv (k - 1, g1, g2)
```

```
    return $ (g1, g2)
```

# Eine Gruppe erwarten

- ▶ Santa erwartet Elfen und Rentiere in entsprechender Gruppengröße
- ▶ Erzeugt neue Gatter für nächsten Rutsch
  - ▶ Verhindert, dass Elfen/Rentiere sich “hineinmogeln”

```
awaitGroup :: Group → STM (Gate, Gate)
awaitGroup (Group n tv) = do
  (k, g1, g2) ← readTVar tv
  check (k == 0)
  g1' ← newGate n
  g2' ← newGate n
  writeTVar tv (n, g1', g2')
  return (g1, g2)
```

# Elfen und Rentiere

- ▶ Für jeden Elf und jedes Rentier wird ein eigener Thread erzeugt
- ▶ Bereits gezeigte `elf1`, `reindeer1`, gefolgt von Verzögerung (für nachvollziehbare Ausgabe)

— An elf does his elf thing, indefinitely.

```
elf :: Buf → Group → Int → IO ThreadId
elf buf grp id = forkIO $ forever $
  do elf1 buf grp id
     randomDelay
```

— So does a deer.

```
reindeer :: Buf → Group → Int → IO ThreadId
reindeer buf grp id = forkIO $ forever $
  do reindeer1 buf grp id
     randomDelay
```

# Santa Claus' Arbeitsablauf

- ▶ Gruppe auswählen, Eingangsgatter öffnen, Ausgang öffnen
- ▶ Zur Erinnerung: operateGate "blockiert", bis alle Gruppenmitglieder Gatter durchschritten haben

```
santa :: Buf → Group → Group → IO ()
santa buf elves deer = do
  (name, (g1, g2)) ← atomically $
    chooseGroup "reindeer" deer 'orElse'
      chooseGroup "elves" elves
  bput buf $ "Ho, ho, my dear " ++ name
  operateGate g1
  operateGate g2
```

```
chooseGroup :: String → Group →
  STM (String, (Gate, Gate))
chooseGroup msg grp = do
  gs ← awaitGroup grp
  return (msg, gs)
```

# Hauptprogramm

- ▶ Gruppen erzeugen, Elfen und Rentiere “starten”, santa ausführen

```
main :: IO ()
main = do buf ← setupBufferListener

        elfGroup ← newGroup 3
        sequence_ [ elf buf elfGroup id |
                    id ← [1 .. 10] ]
        deerGroup ← newGroup 9
        sequence_ [ reindeer buf deerGroup id |
                    id ← [1 .. 9]]
        forever (santa buf elfGroup deerGroup)
```

# Zusammenfassung

- ▶ *The future is now, the future is concurrent*
- ▶ Lock-basierte Nebenläufigkeitsansätze skalieren schlecht
  - ▶ Korrekte Einzelteile können nicht ohne weiteres komponiert werden
- ▶ Software Transactional Memory als Lock-freie Alternative
  - ▶ **A**tomarität ( *atomically* ), **B**lockieren ( *retry* ), **C**hoice ( *orElse* ) als Fundamente kompositionaler Nebenläufigkeit
  - ▶ Faszinierend einfache Implementierungen gängiger Nebenläufigkeitsaufgaben
- ▶ Das freut auch den Weihnachtsmann:
  - ▶ Santa Claus Problem in STM Haskell

# Literatur



Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy.

Composable memory transactions.

In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.



Simon Peyton Jones.

Beautiful concurrency.

In Greg Wilson, editor, *Beautiful code*. O'Reilly, 2007.



Herb Sutter.

The free lunch is over: a fundamental turn toward concurrency in software.

*Dr. Dobbs' Journal*, 30(3), March 2005.

Reaktive Programmierung  
Vorlesung 14 vom 26.06.19  
Eventual Consistency

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Heute

- ▶ Konsistenzeigenschaften
- ▶ Eventual Consistency
- ▶ CRDTs
- ▶ Operational Transformation
  - ▶ *Das Geheimnis von Google Docs und co.*

# Was ist eigentlich Konsistenz?

- ▶ Konsistenz = **Widerspruchsfreiheit**
- ▶ In der Logik:
  - ▶ Eine Formelmenge  $\Gamma$  ist konsistent wenn:  $\exists A. \neg(\Gamma \vdash A)$
- ▶ In einem verteilten System:
  - ▶ Redundante (verteilte) Daten
  - ▶ **Globale** Widerspruchsfreiheit?

# Strikte Konsistenz

## Strikte Konsistenz

- ▶ Daten sind zu jedem Zeitpunkt global konsistent.
- ▶ Eine Leseoperation in einem beliebigen Knoten gibt den Wert der letzten globalen Schreiboperation zurück.
- ▶ In echten verteilten Systemen **nicht implementierbar**.

# Sequentielle Konsistenz

## Sequentielle Konsistenz

- ▶ Zustand nach verteilter Programmausführung = Zustand nach einer äquivalenten sequentiellen Ausführung in einem Prozess.
- ▶ Jeder Prozess sieht die selbe Folge von Operationen.

# Eventual Consistency

## Eventual Consistency

Wenn **längere Zeit** keine Änderungen stattfinden konvergieren die Daten an jedem Knoten zu einem gemeinsamen Wert.

- ▶ Beispiel: DNS

# Strong Eventual Consistency

- ▶ Eventual Consistency ist eine **informelle** Anforderung.
  - ▶ Abfragen können beliebige Werte zurückgeben bevor die Knoten konvergieren.
  - ▶ Keine Sicherheit!
- ▶ **Strong Eventual Consistency** garantiert:
  - ▶ wenn zwei Knoten die **gleiche (ungeordnete) Menge** von Operationen empfangen haben, befinden sie sich im **gleichen Zustand**.
- ▶ Beispiel: Versionskontrollsystem *git*
  - ▶ Wenn jeder Nutzer seine lokalen Änderungen eingechekct hat, dann haben alle Nutzer die gleiche Sicht auf den *head*.

# Monotonie

- ▶ Strong Eventual Consistency kann einfach erreicht werden:
  - ▶ Nach jedem empfangenen Update alle Daten zurücksetzen.
- ▶ Für sinnvolle Anwendungen brauchen wir eine weitere Garantie:

## Monotonie

Ein verteiltes System ist monoton, wenn der Effekt jeder Operation erhalten bleibt (keine Rollbacks).

# Beispiel: Texteditor

- ▶ Szenario: Webinterface mit Texteditor
- ▶ Mehrere Nutzer können den Text verändern und sollen **immer die neueste Version** sehen.
- ▶ Siehe Google Docs, Etherpad und co.

# Naive Methoden

- ▶ Ownership
  - ▶ Vor Änderungen: Lock-Anfrage an Server
  - ▶ Nur ein Nutzer kann gleichzeitig das Dokument ändern
  - ▶ Nachteile: Verzögerungen, Änderungen nur mit Netzverbindung
- ▶ Three-Way-Merge
  - ▶ Server führt nebenläufige Änderungen auf Grundlage eines **gemeinsamen Ursprungs** zusammen.
  - ▶ Requirement: *the chickens must stop moving so we can count them*

# Conflict-Free Replicated Data Types

- ▶ Konfliktfreie replizierte Datentypen
- ▶ Garantieren
  - ▶ Strong Eventual Consistency
  - ▶ Monotonie
  - ▶ Konfliktfreiheit
- ▶ Zwei Klassen:
  - ▶ Zustandsbasierte CRDTs
  - ▶ Operationsbasierte CRDTs

# Zustandsbasierte CRDTs

- ▶ Konvergente replizierte Datentypen (CvRDTs)
- ▶ Knoten senden ihren gesamten Zustand an andere Knoten.
- ▶ Nur bestimmte Operationen auf dem Datentypen erlaubt (*update*).
- ▶ Eine **kommutative**, **assoziative**, **idempotente** *merge*-Funktion
  - ▶ Funktioniert gut mit Gossiping-Protokollen
  - ▶ Nachrichtenverlust unkritisch

# CvRDT: Zähler

- ▶ Einfacher CvRDT

- ▶ Zustand:  $P \in \mathbb{N}$ , Datentyp:  $\mathbb{N}$

$$\text{query}(P) = P$$

$$\text{update}(P, +, m) = P + m$$

$$\text{merge}(P_1, P_2) = \max(P_1, P_2)$$

- ▶ Wert kann nur größer werden.

# CvRDT: PN-Zähler

- ▶ Gängiges Konzept bei CRDTs: Komposition
- ▶ Aus zwei Zählern kann ein komplexerer Typ **zusammengesetzt** werden:
  - ▶ Zähler P (Positive) und Zähler N (Negative)
  - ▶ Zustand:  $(P, N) \in \mathbb{N} \times \mathbb{N}$ , Datentyp:  $\mathbb{Z}$

$$\text{query}((P, N)) = \text{query}(P) - \text{query}(N)$$

$$\text{update}((P, N), +, m) = (\text{update}(P, +, m), N)$$

$$\text{update}((P, N), -, m) = (P, \text{update}(N, +, m))$$

$$\text{merge}((P_1, N_1), (P_2, N_2)) = (\text{merge}(P_1, P_2), \text{merge}(N_1, N_2))$$

# CvRDT: Mengen

- ▶ Ein weiterer einfacher CRDT:
  - ▶ Zustand:  $P \in \mathcal{P}(A)$ , Datentyp:  $\mathcal{P}(A)$

$$\text{query}(P) = P$$

$$\text{update}(P, +, a) = P \cup \{a\}$$

$$\text{merge}(P_1, P_2) = P_1 \cup P_2$$

- ▶ Die Menge kann nur wachsen.

# CvRDT: Zwei-Phasen-Mengen

- ▶ Durch Komposition kann wieder ein komplexerer Typ entstehen.
- ▶ Menge  $P$  (Hinzugefügte Elemente) und Menge  $N$  (Gelöschte Elemente)
- ▶ Zustand:  $(P, N) \in \mathcal{P}(A) \times \mathcal{P}(A)$ , Datentyp:  $\mathcal{P}(A)$

$$\text{query}((P, N)) = \text{query}(P) \setminus \text{query}(N)$$

$$\text{update}((P, N), +, m) = (\text{update}(P, +, m), N)$$

$$\text{update}((P, N), -, m) = (P, \text{update}(N, +, m))$$

$$\text{merge}((P_1, N_1), (P_2, N_2)) = (\text{merge}(P_1, P_2), \text{merge}(N_1, N_2))$$

# Operationsbasierte CRDTs

- ▶ Kommutative replizierte Datentypen (CmRDTs)
- ▶ Knoten senden nur **Operationen** an andere Knoten
- ▶ *update* unterscheidet zwischen lokalem und externem Effekt.
- ▶ Netzwerkprotokoll wichtig
- ▶ Nachrichtenverlust führt zu Inkonsistenzen
- ▶ Kein *merge* nötig
- ▶ Kann die übertragenen **Datenmengen** erheblich **reduzieren**

# CmRDT: Zähler

- ▶ Zustand:  $P \in \mathbb{N}$ , Typ:  $\mathbb{N}$
- ▶  $query(P) = P$
- ▶  $update(+, n)$ 
  - ▶ lokal:  $P := P + n$
  - ▶ extern:  $P := P + n$

# CmRDT: Last-Writer-Wins-Register

- ▶ Zustand:  $(x, t) \in X \times \text{timestamp}$
- ▶  $query((x, t)) = x$
- ▶  $update(=, x')$ 
  - ▶ lokal:  $(x, t) := (x', \text{now}())$
  - ▶ extern: *if*  $t < t'$  *then*  $(x, t) := (x', t')$

# Vektor-Uhren

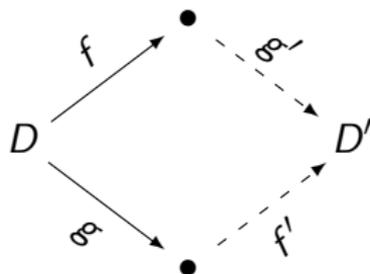
- ▶ Im LWW Register benötigen wir Timestamps
  - ▶ Kausalität muss erhalten bleiben
  - ▶ Timestamps müssen eine total Ordnung haben
- ▶ Datum und Uhrzeit ungeeignet
- ▶ Lösung: Vektor-Uhren
  - ▶ Jeder Knoten hat einen Zähler, der bei Operationen hochgesetzt wird
  - ▶ Zusätzlich merkt sich jeder Knoten den aktuellsten Zählerwert, den er bei den anderen Knoten beobachtet hat.

# Operational Transformation

- ▶ Die CRDTs die wir bis jetzt kennengelernt haben sind recht einfach
- ▶ Das Texteditor Beispiel ist damit noch nicht umsetzbar
- ▶ Kommutative Operationen auf einer Sequenz von Buchstaben?
  - ▶ Einfügen möglich (totale Ordnung durch Vektoruhren)
  - ▶ Wie Löschen?

# Operational Transformation

- ▶ Idee: Nicht-kommutative Operationen transformieren



- ▶ Für *transform* muss gelten:

$$\text{transform } f \ g = \langle f', g' \rangle \implies g' \circ f = f' \circ g \quad (1)$$

$$\text{applyOp } (g \circ f) \ D = \text{applyOp } g \ (\text{applyOp } f \ D) \quad (2)$$

# Operationen für Text

Operationen bestehen aus **drei**

Arten von Aktionen:

- ▶ *Retain*— Buchstaben beibehalten
- ▶ *Delete*— Buchstaben löschen
- ▶ *Insert c* — Buchstaben *c* einfügen

Eine **Operation** ist eine Sequenz von Aktionen

# Operationen für Text

Operationen bestehen aus **drei**

Arten von Aktionen:

- ▶ *Retain*— Buchstaben beibehalten
- ▶ *Delete*— Buchstaben löschen
- ▶ *Insert c* — Buchstaben *c* einfügen

Eine **Operation** ist eine Sequenz von Aktionen

Ein **Beispiel**:

Eingabe: R 1 P 7

Ausgabe:

Aktionen:

# Operationen für Text

Operationen bestehen aus **drei**

Arten von Aktionen:

- ▶ *Retain*— Buchstaben beibehalten
- ▶ *Delete*— Buchstaben löschen
- ▶ *Insert c* — Buchstaben *c* einfügen

Eine **Operation** ist eine Sequenz von Aktionen

Ein **Beispiel**:

Eingabe: 1 P 7

Ausgabe: R

Aktionen: *Retain*,

# Operationen für Text

Operationen bestehen aus **drei**

Arten von Aktionen:

- ▶ *Retain*— Buchstaben beibehalten
- ▶ *Delete*— Buchstaben löschen
- ▶ *Insert c* — Buchstaben *c* einfügen

Eine **Operation** ist eine Sequenz von Aktionen

Ein **Beispiel**:

Eingabe: P 7

Ausgabe: R

Aktionen: *Retain*,  
*Delete*,

# Operationen für Text

Operationen bestehen aus **drei**

Arten von Aktionen:

- ▶ *Retain*— Buchstaben beibehalten
- ▶ *Delete*— Buchstaben löschen
- ▶ *Insert c* — Buchstaben *c* einfügen

Eine **Operation** ist eine Sequenz von Aktionen

Ein **Beispiel**:

Eingabe: 7  
Ausgabe: R P  
Aktionen: *Retain*,  
*Delete*,  
*Retain*,

# Operationen für Text

Operationen bestehen aus **drei**

Arten von Aktionen:

- ▶ *Retain*— Buchstaben beibehalten
- ▶ *Delete*— Buchstaben löschen
- ▶ *Insert c* — Buchstaben *c* einfügen

Eine **Operation** ist eine Sequenz von Aktionen

Ein **Beispiel**:

Eingabe: 7  
Ausgabe: R P 1  
Aktionen: *Retain*,  
*Delete*,  
*Retain*,  
*Insert 1*,

# Operationen für Text

Operationen bestehen aus **drei**

Arten von Aktionen:

- ▶ *Retain*— Buchstaben beibehalten
- ▶ *Delete*— Buchstaben löschen
- ▶ *Insert c* — Buchstaben *c* einfügen

Eine **Operation** ist eine Sequenz von Aktionen

Ein **Beispiel**:

Eingabe:

Ausgabe: R P 1 7

Aktionen: *Retain,*  
*Delete,*  
*Retain,*  
*Insert 1,*  
*Retain.*

# Operationen für Text

Operationen bestehen aus **drei**

Arten von Aktionen:

- ▶ *Retain*— Buchstaben beibehalten
- ▶ *Delete*— Buchstaben löschen
- ▶ *Insert c* — Buchstaben *c* einfügen

Eine **Operation** ist eine Sequenz von Aktionen

- ▶ Operationen sind **partiell**.

Ein **Beispiel**:

Eingabe:

Ausgabe: R P 1 7

Aktionen: *Retain,*  
*Delete,*  
*Retain,*  
*Insert 1,*  
*Retain.*

# Operationen Komponieren

- ▶ Komposition: Fallunterscheidung auf der **Aktion**

- ▶ Keine einfache Konkatenation!

- ▶ Beispiel:

$$p = [Delete, Insert X, Retain]$$

$$q = [Retain, Insert Y, Delete]$$

$$compose\ p\ q =$$

- ▶  $compose$  ist partiell.

- ▶ **Äquivalenz** von Operationen:

$$compose\ p\ q \cong [Delete, Delete, Insert X, Insert Y]$$

# Operationen Komponieren

- ▶ Komposition: Fallunterscheidung auf der **Aktion**

- ▶ Keine einfache Konkatenation!

- ▶ Beispiel:

$$p = [\textit{Insert X}, \textit{Retain}]$$

$$q = [\textit{Retain}, \textit{Insert Y}, \textit{Delete}]$$

$$\textit{compose } p \ q = [\textit{Delete},$$

- ▶ *compose* ist partiell.

- ▶ **Äquivalenz** von Operationen:

$$\textit{compose } p \ q \cong [\textit{Delete}, \textit{Delete}, \textit{Insert X}, \textit{Insert Y}]$$

# Operationen Komponieren

- ▶ Komposition: Fallunterscheidung auf der **Aktion**

- ▶ Keine einfache Konkatenation!

- ▶ Beispiel:

$$p = [\textit{Retain}]$$

$$q = [\textit{Insert Y}, \textit{Delete}]$$

$$\textit{compose } p \ q = [\textit{Delete}, \textit{Insert X},$$

- ▶ *compose* ist partiell.

- ▶ **Äquivalenz** von Operationen:

$$\textit{compose } p \ q \cong [\textit{Delete}, \textit{Delete}, \textit{Insert X}, \textit{Insert Y}]$$

# Operationen Komponieren

- ▶ Komposition: Fallunterscheidung auf der **Aktion**

- ▶ Keine einfache Konkatenation!

- ▶ Beispiel:

$$p = [\textit{Retain}]$$

$$q = [\textit{Delete}]$$

$$\textit{compose } p \ q = [\textit{Delete}, \textit{Insert } X, \textit{Insert } Y,$$

- ▶ *compose* ist partiell.

- ▶ **Äquivalenz** von Operationen:

$$\textit{compose } p \ q \cong [\textit{Delete}, \textit{Delete}, \textit{Insert } X, \textit{Insert } Y]$$

# Operationen Komponieren

- ▶ Komposition: Fallunterscheidung auf der **Aktion**

- ▶ Keine einfache Konkatination!

- ▶ Beispiel:

$$p = []$$

$$q = []$$

$$\text{compose } p \ q = [\text{Delete}, \text{Insert } X, \text{Insert } Y, \text{Delete}]$$

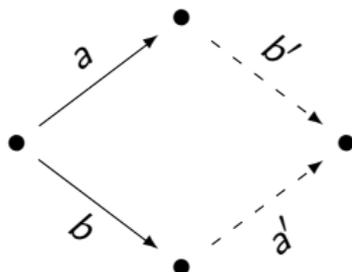
- ▶ *compose* ist partiell.

- ▶ **Äquivalenz** von Operationen:

$$\text{compose } p \ q \cong [\text{Delete}, \text{Delete}, \text{Insert } X, \text{Insert } Y]$$

# Operationen Transformieren

## ► Transformation



## ► Beispiel:

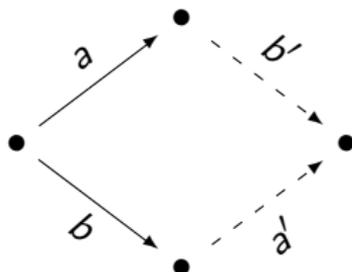
$a = [Insert\ X, Retain, Delete]$

$b = [Delete, Retain, Insert\ Y]$

$transform\ a\ b = ([$   
 $\quad , [$   
 $\quad \quad )$

# Operationen Transformieren

## ► Transformation



## ► Beispiel:

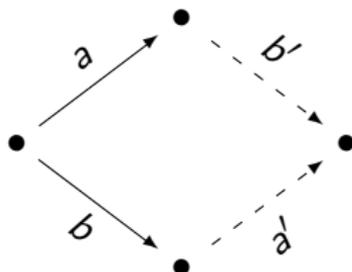
$a = [Retain, Delete]$

$b = [Delete, Retain, Insert Y]$

$transform\ a\ b = ([Insert\ X,$   
 $\quad\quad\quad , [Retain,$   
 $\quad\quad\quad )$

# Operationen Transformieren

## ► Transformation



## ► Beispiel:

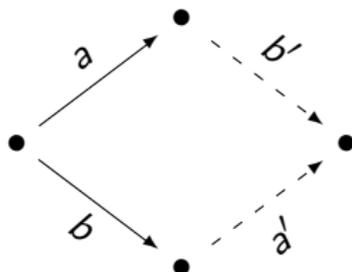
$a = [Delete]$

$b = [Retain, Insert Y]$

$transform\ a\ b = ([Insert\ X, Delete,$   
 $\quad\quad\quad , [Retain,$   
 $\quad\quad\quad )$

# Operationen Transformieren

## ► Transformation



## ► Beispiel:

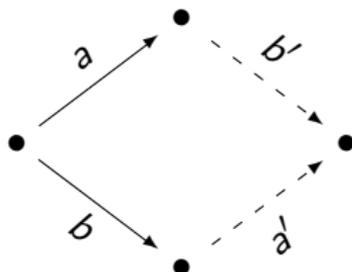
$a = []$

$b = [Insert\ Y]$

$transform\ a\ b = ([Insert\ X, Delete,$   
 $, [Retain, Delete,$   
 $])$

# Operationen Transformieren

## ► Transformation



## ► Beispiel:

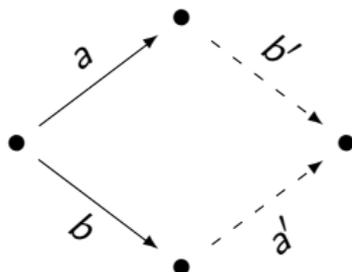
$$a = []$$

$$b = []$$

$$\text{transform } a \ b = ([\text{Insert X}, \text{Delete}, \text{Retain}], [\text{Retain}, \text{Delete}, \text{Insert Y}])$$

# Operationen Transformieren

## ► Transformation



## ► Beispiel:

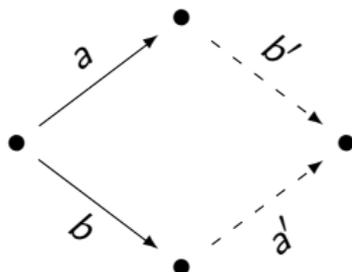
$$a = []$$

$$b = []$$

$$\text{transform } a \ b = ([\text{Insert X}, \text{Delete}, \text{Retain}], [\text{Retain}, \text{Delete}, \text{Insert Y}])$$

# Operationen Transformieren

## ► Transformation



## ► Beispiel:

$a = [\textit{Insert X}, \textit{Retain}, \textit{Delete}]$

$b = [\textit{Delete}, \textit{Retain}, \textit{Insert Y}]$

$\textit{transform } a \textit{ } b = ([\textit{Insert X}, \textit{Delete}, \textit{Retain}]$   
 $\quad , [\textit{Retain}, \textit{Delete}, \textit{Insert Y}]$   
 $\quad )$

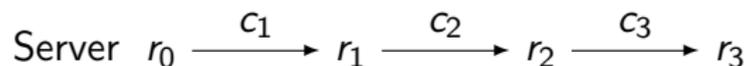
# Operationen Verteilen

- ▶ Wir haben die Funktion *transform* die zwei nicht-kommutativen Operationen  $a$  und  $b$  zu kommutierenden Gegenständen  $a'$  und  $b'$  transformiert.
- ▶ Was machen wir jetzt damit?
- ▶ Kontrollalgorithmus nötig

# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen

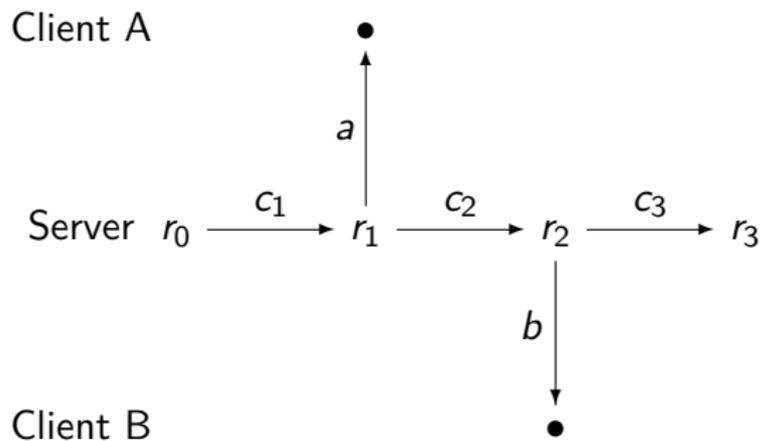
Client A



Client B

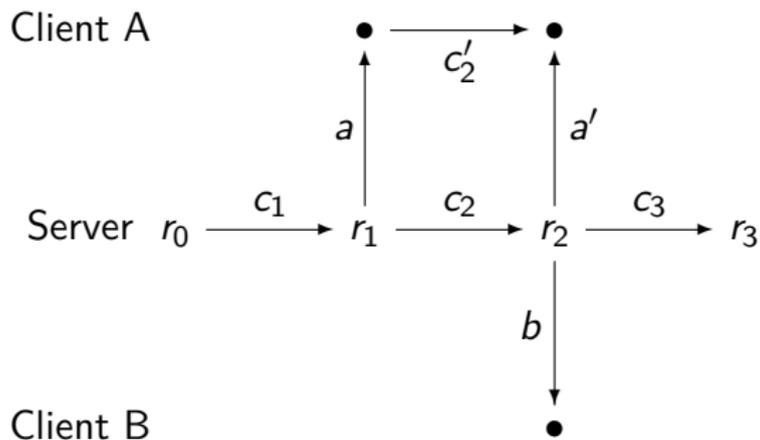
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



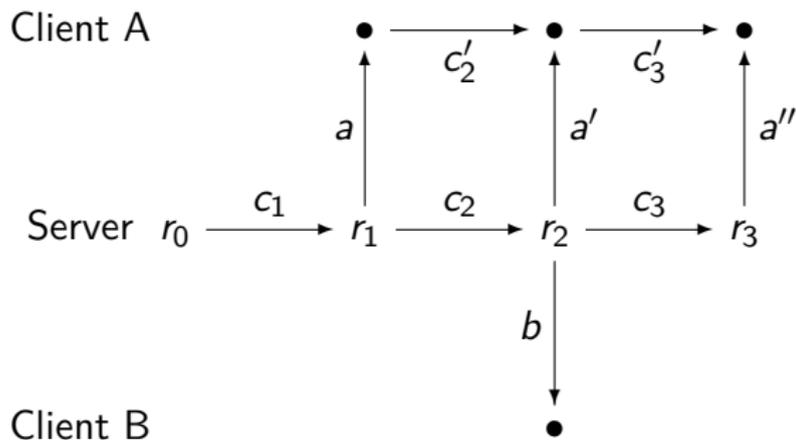
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



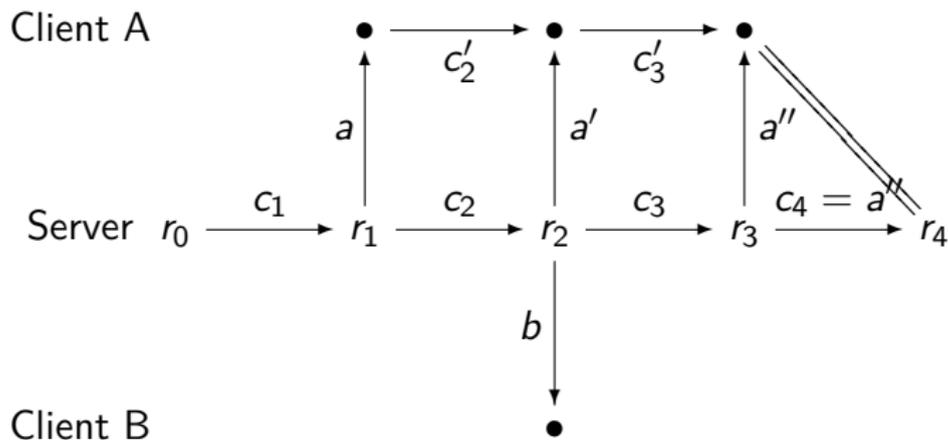
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



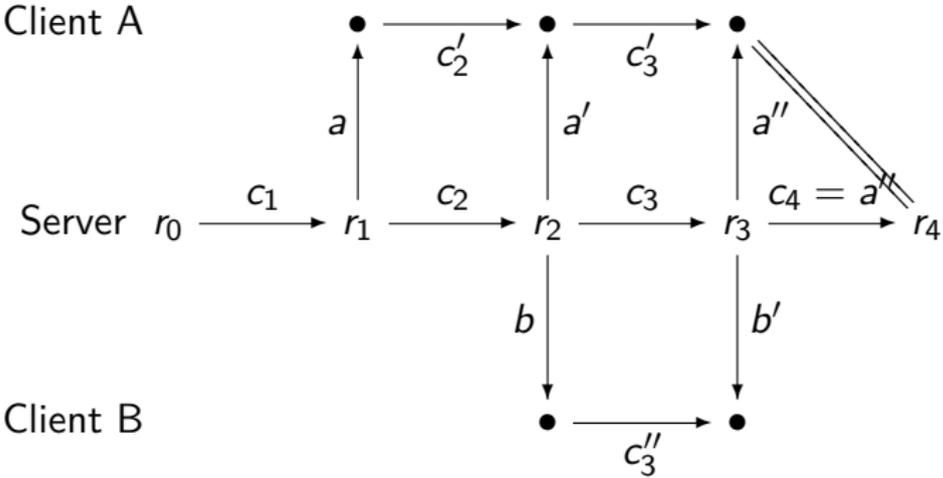
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



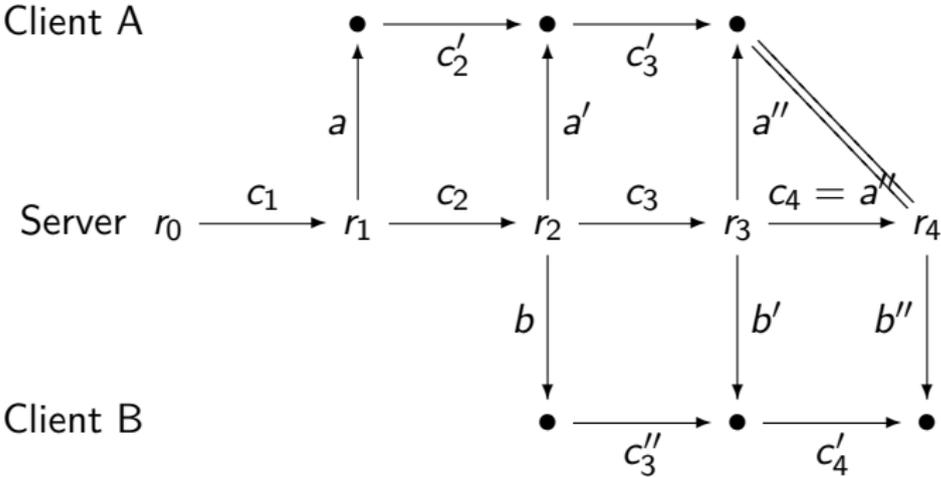
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequentialisieren
  - ▶ Transformierte Operationen verteilen



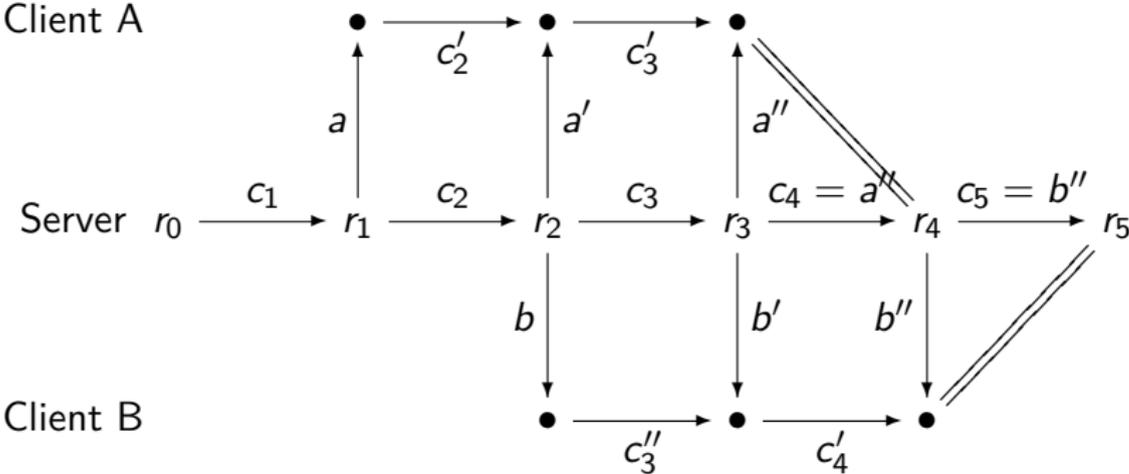
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequentialisieren
  - ▶ Transformierte Operationen verteilen



# Der Server

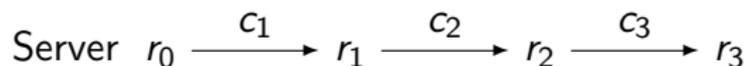
- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequentialisieren
  - ▶ Transformierte Operationen verteilen



# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen

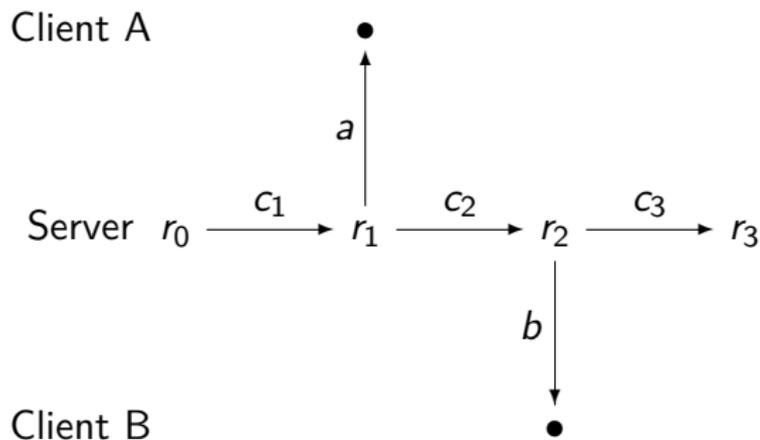
Client A



Client B

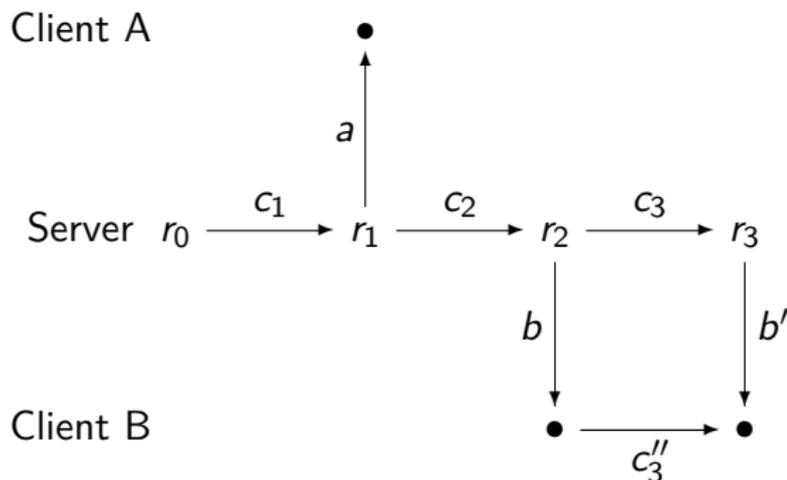
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



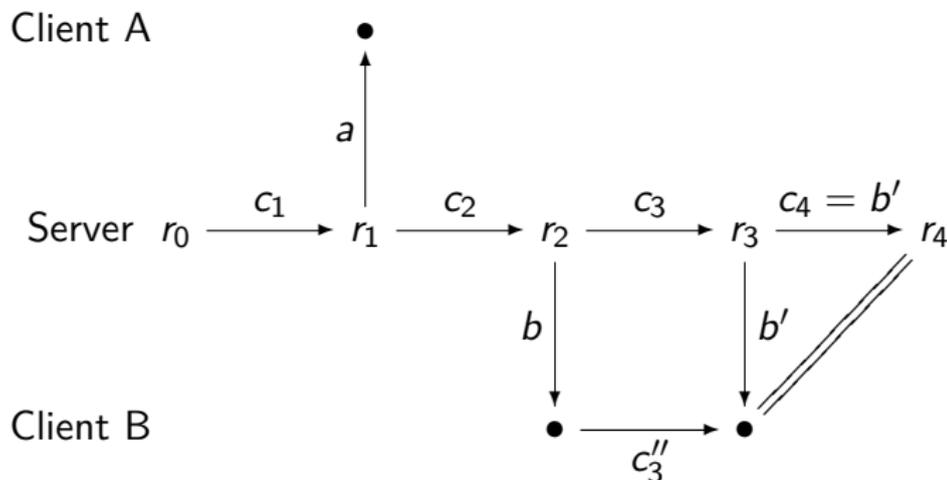
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



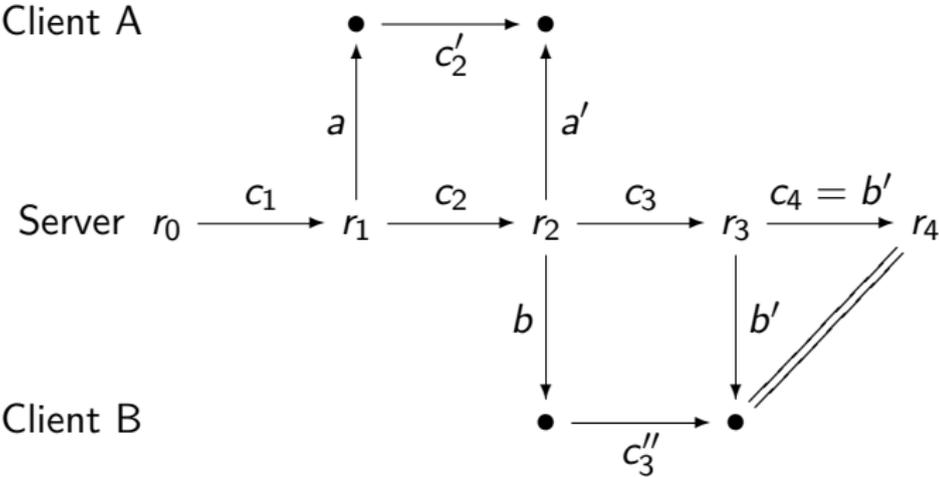
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



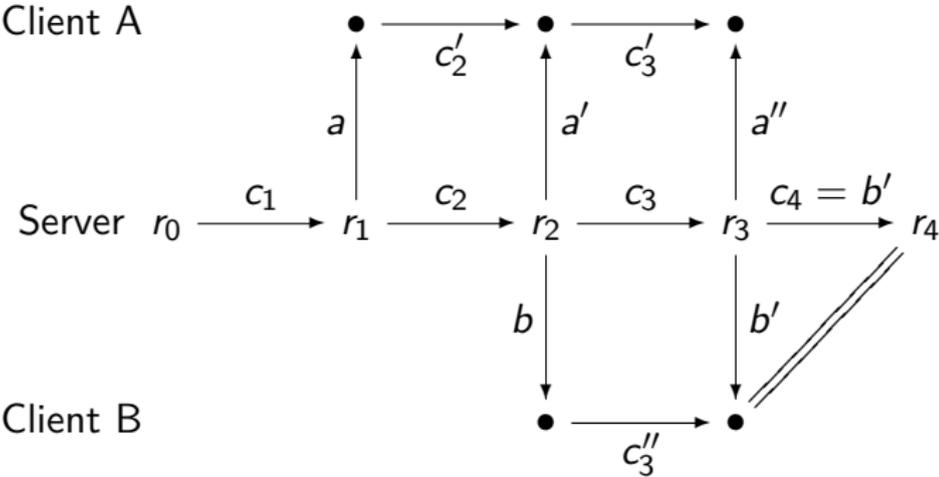
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



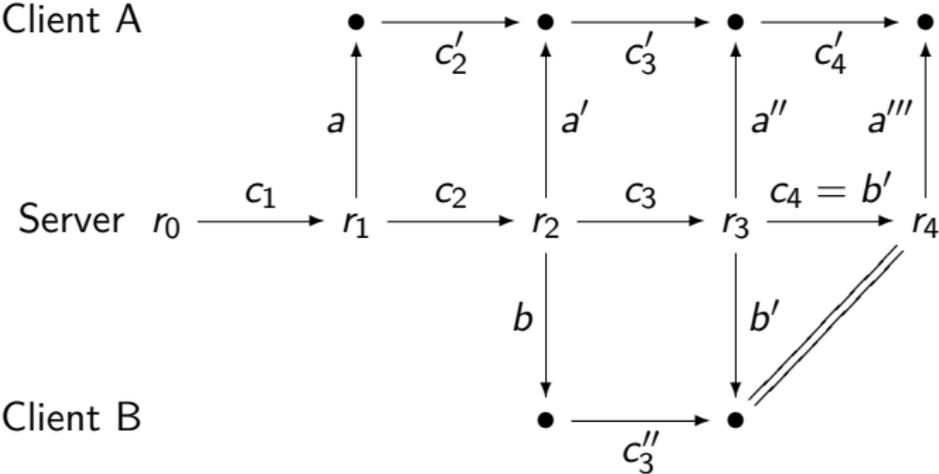
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



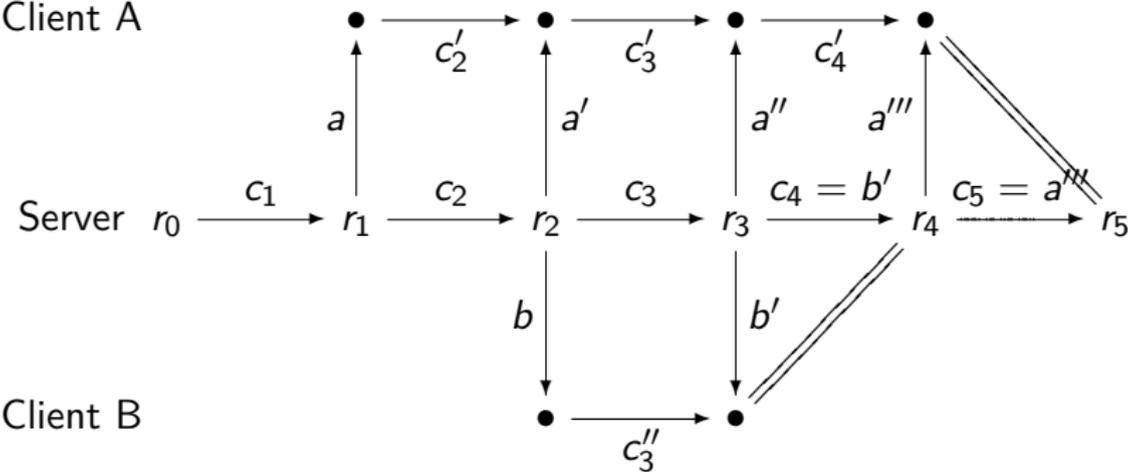
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequentialisieren
  - ▶ Transformierte Operationen verteilen



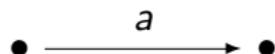
# Der Server

- ▶ Zweck:
  - ▶ Nebenläufige Operationen sequenzialisieren
  - ▶ Transformierte Operationen verteilen



# Der Client

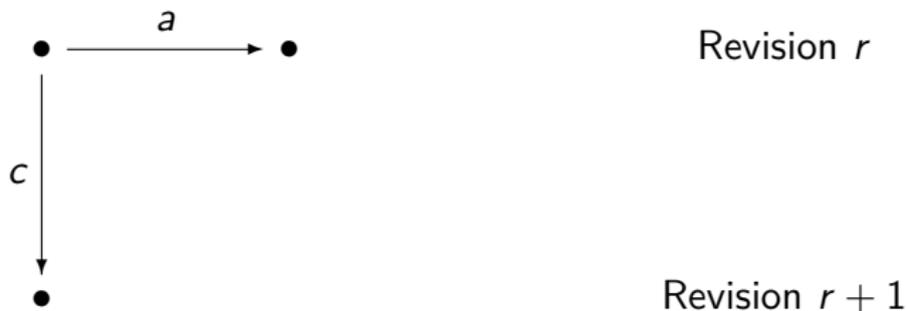
- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



Revision *r*

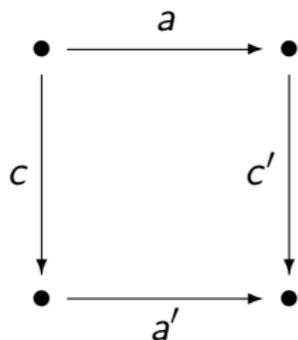
# Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



# Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht

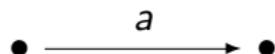


Revision  $r$

Revision  $r + 1$

# Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



Revision *r*

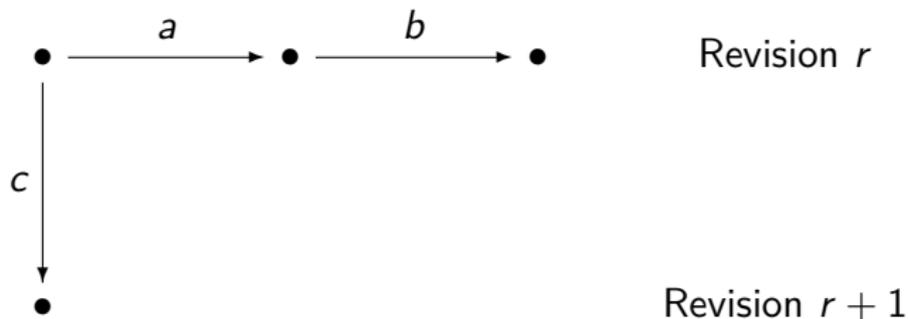
# Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



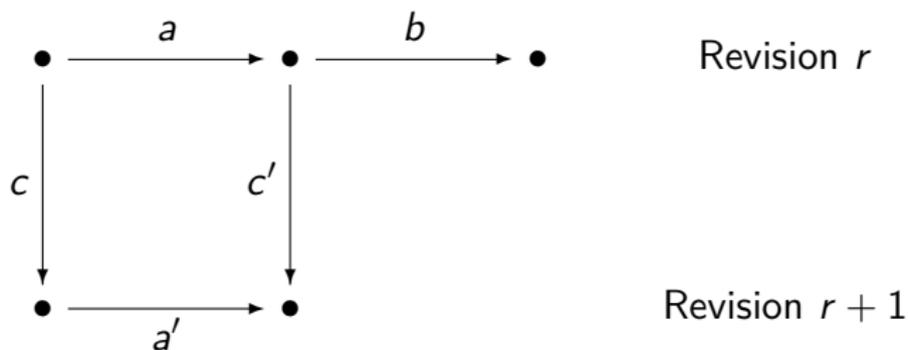
# Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



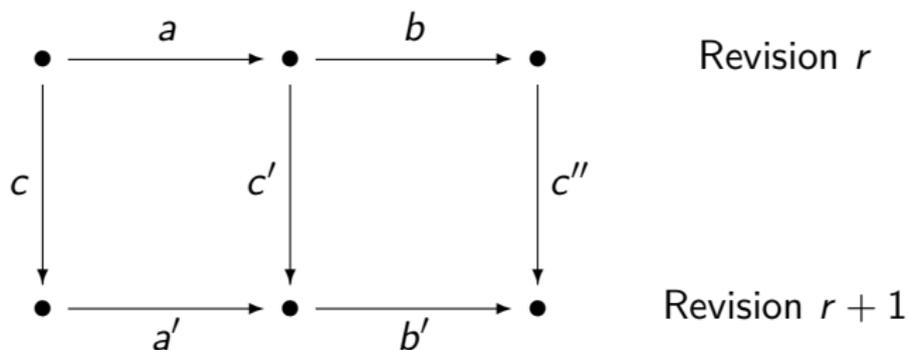
# Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



# Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



# Zusammenfassung

- ▶ Strikte Konsistenz in verteilten Systemen nicht erreichbar
- ▶ Strong Eventual Consistency
  - ▶ Wenn **längere Zeit** keine Änderungen stattgefunden haben befinden sich schließlich alle Knoten im **gleichen Zustand**.
  - ▶ Wenn zwei Knoten die **gleiche Menge** Updates beobachten befinden sie sich im **gleichen Zustand**.
- ▶ Conflict-Free replicated Data Types:
  - ▶ Zustandsbasiert: CvRDTs
  - ▶ Operationsbasiert: CmRDTs
- ▶ Operational Transformation
  - ▶ Strong Eventual Consistency auch ohne kommutative Operationen

Reaktive Programmierung  
Vorlesung 15 vom 03.07.19  
Robustheit und Entwurfsmuster

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Rückblick: Konsistenz

- ▶ Strikte Konsistenz in verteilten Systemen nicht erreichbar
- ▶ Strong Eventual Consistency
  - ▶ Wenn **längere Zeit** keine Änderungen stattgefunden haben befinden sich schließlich alle Knoten im **gleichen Zustand**.
  - ▶ Wenn zwei Knoten die **gleiche Menge** Updates beobachten befinden sie sich im **gleichen Zustand**.
- ▶ Conflict-Free replicated Data Types:
  - ▶ Zustandsbasiert: CvRDTs
  - ▶ Operationsbasiert: CmRDTs
- ▶ Operational Transformation
  - ▶ Strong Eventual Consistency auch ohne kommutative Operationen

# Robustheit in verteilten Systemen

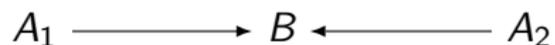
Lokal:

- ▶ Nachrichten gehen nicht verloren
- ▶ Aktoren können abstürzen - Lösung: Supervisor

Verteilt:

- ▶ Nachrichten können verloren gehen
- ▶ Teilsysteme können abstürzen
  - ▶ Hardware-Fehler
  - ▶ Stromausfall
  - ▶ Geplanter Reboot (Updates)
  - ▶ Naturkatastrophen / Höhere Gewalt
  - ▶ Software-Fehler

# Zwei-Armeen-Problem

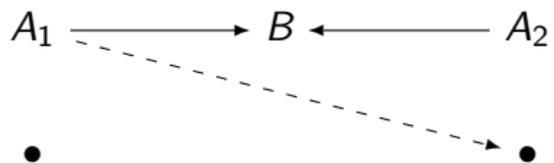


- ▶ Zwei Armeen  $A_1$  und  $A_2$  sind jeweils zu klein um gegen den Feind  $B$  zu gewinnen.
- ▶ Daher wollen sie sich über einen Angriffszeitpunkt absprechen.

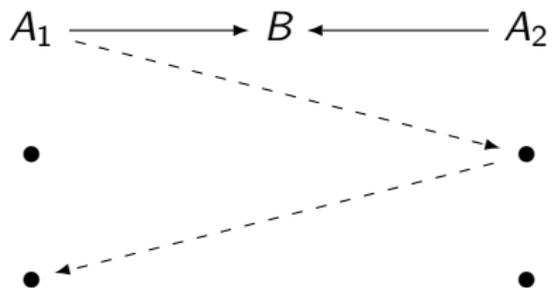
# Zwei-Armeen-Problem



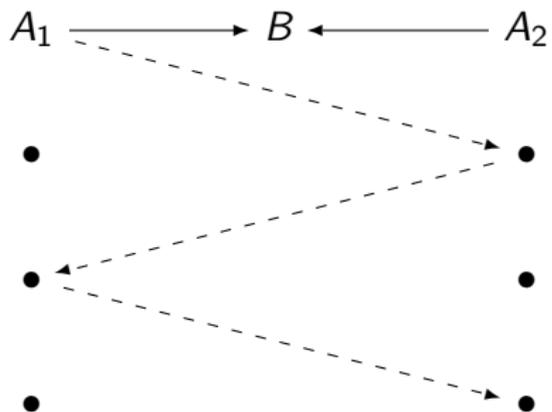
# Zwei-Armeen-Problem



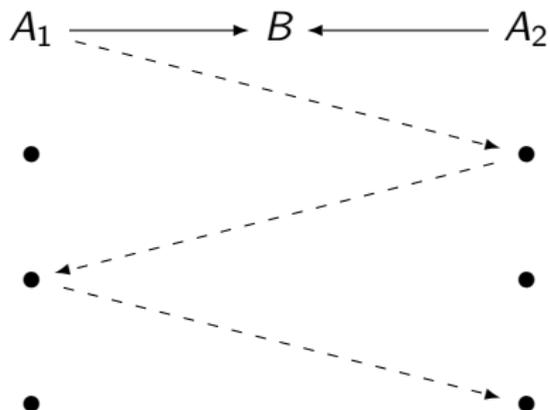
# Zwei-Armeen-Problem



# Zwei-Armeen-Problem



# Zwei-Armeen-Problem



► Unlösbar – Wir müssen damit leben!

# Unsichere Kanäle

- ▶ Unsichere Kanäle sind ein generelles Problem der Netzwerktechnik
- ▶ Lösungsstrategien:
  - ▶ Redundanz – Nachrichten mehrfach schicken
  - ▶ Indizierung – Nachrichten numerieren
  - ▶ Timeouts – Nicht ewig auf Antwort warten
  - ▶ Heartbeats – Regelmäßige „Lebenszeichen“
- ▶ Beispiel: TCP
  - ▶ Drei-Wege Handschlag
  - ▶ Indizierte Pakete

# Gossiping

$N_1$

$N_2$

$N_3$

$N_4$

$N_5$

$N_6$

$N_7$

$N_8$

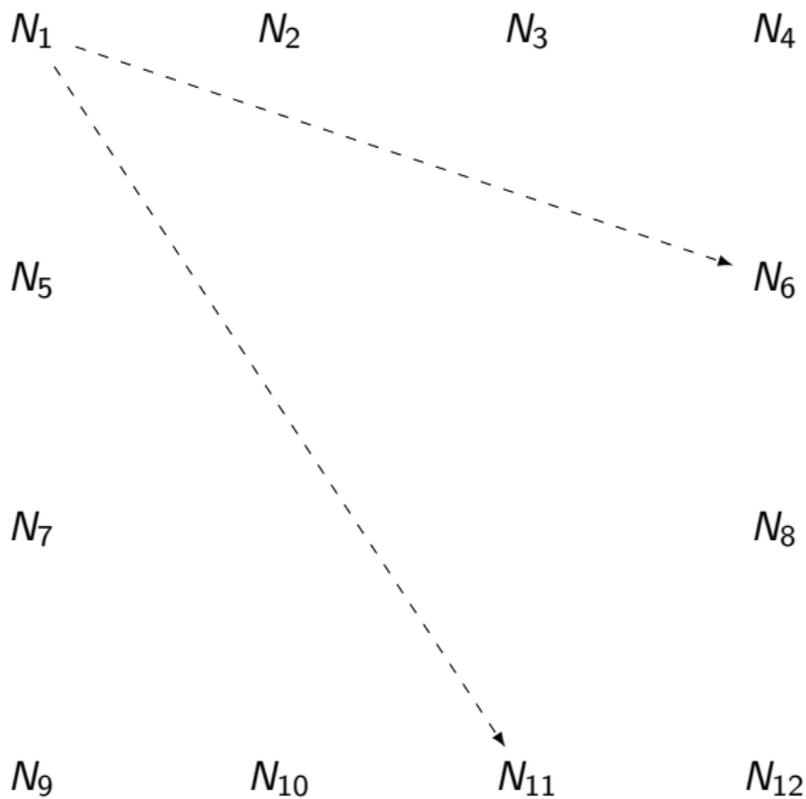
$N_9$

$N_{10}$

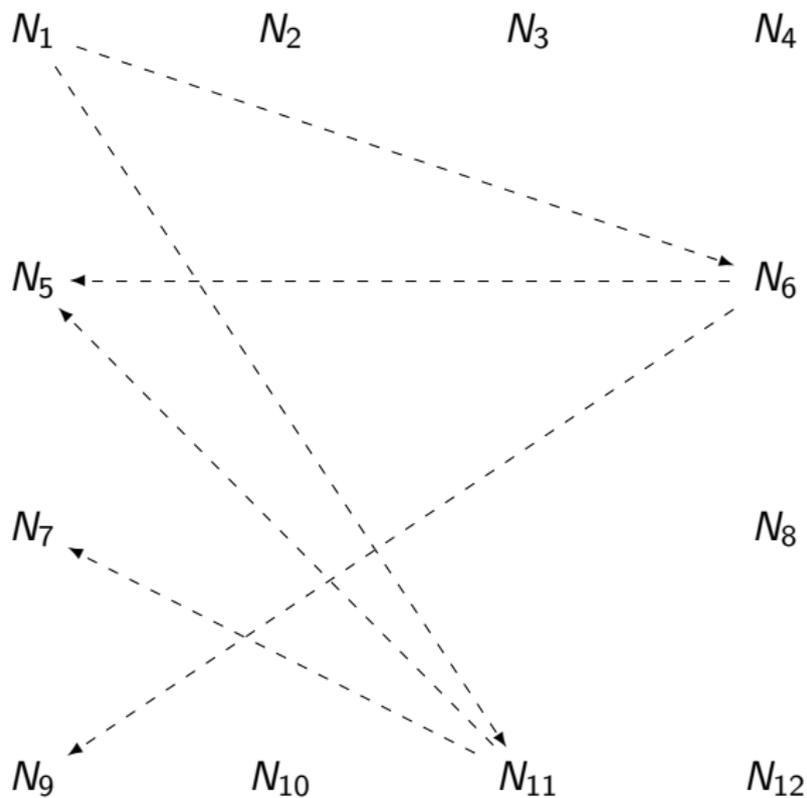
$N_{11}$

$N_{12}$

# Gossiping



# Gossiping

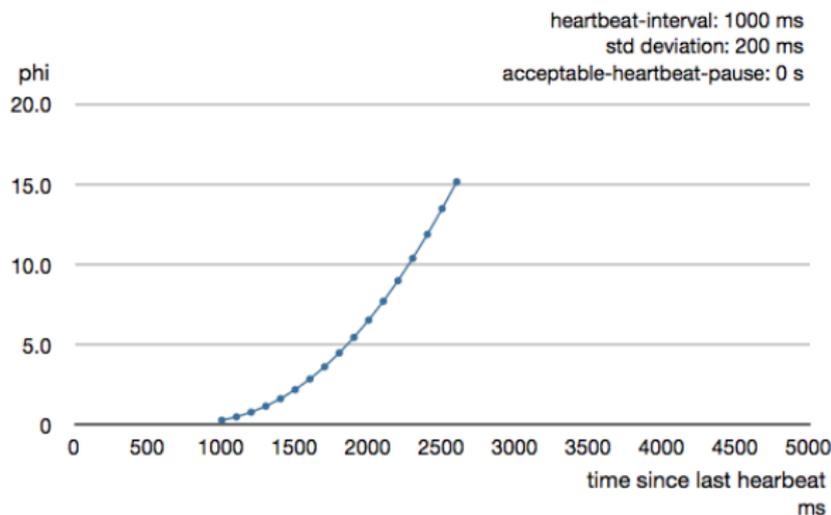


# Gossiping

- ▶ Jeder Knoten verbreitet Informationen periodisch weiter an **zufällige** weitere Knoten
- ▶ Funktioniert besonders gut mit CvRDTs
  - ▶ Nachrichtenverlust unkritisch
- ▶ Anwendungen
  - ▶ Ereignis-Verteilung
  - ▶ Datenabgleich
  - ▶ Anti-entropy Protokolle
  - ▶ Aggregate, Suche

# Heartbeats

- ▶ Kleine Nachrichten in regelmäßigen Abständen
- ▶ Standardabweichung kann dynamisch berechnet werden
- ▶  $\Phi = -\log_{10}(1 - F(\text{timeSinceLastHeartbeat}))$



# Akka Clustering

- ▶ Verteiltes Aktorsystem
  - ▶ Infrastruktur wird über gossiping Protokoll geteilt
  - ▶ Ausfälle werden über Heartbeats erkannt
- ▶ **Sharding**: Horizontale Verteilung der Ressourcen
  - ▶ In Verbindung mit Gossiping mächtig

## (Anti-)Patterns: Request/Response

- ▶ Problem: Warten auf eine Antwort — Benötigt einen Kontext der die Antwort versteht
- ▶ Pragmatische Lösung: Ask-Pattern

```
import akka.patterns.ask

(otherActor ? Request) map {
  case Response => //
}
```

- ▶ Eignet sich nur für sehr einfache Szenarien
- ▶ Lösung: Neuer Aktor für jeden Response Kontext

## (Anti-)Patterns: Nachrichten

- ▶ Nachrichten sollten **typisiert** sein

```
otherActor ! "add 5 to your local state" // NO  
otherActor ! Modify(_ + 5) // YES
```

- ▶ Nachrichten dürfen **nicht** veränderlich sein!

```
val state: scala.collection.mutable.Buffer  
otherActor ! Include(state) // NO  
otherActor ! Include(state.toList) // YES
```

- ▶ Nachrichten dürfen **keine Referenzen** auf veränderlichen Zustand enthalten

```
var state = 7  
otherActor ! Modify(_ + state) // NO  
val stateCopy = state  
otherActor ! Modify(_ + stateCopy) // YES
```

## (Anti-)Patterns: State-Leaks

- ▶ Lokaler Zustand darf auf keinen Fall “auslaufen”!

```
var state = 0
(otherActor ? Request) map { case Response => sender !
  RequestComplete }
```

## (Anti-)Patterns: State-Leaks

- ▶ Lokaler Zustand darf auf keinen Fall “auslaufen”!

```
var state = 0
(otherActor ? Request) map { case Response => sender !
  RequestComplete }
```

- ▶ Besser?

```
(otherActor ? Request) map { case Response =>
  state += 1; RequestComplete
} pipeTo sender
```

## (Anti-)Patterns: State-Leaks

- ▶ Lokaler Zustand darf auf keinen Fall “auslaufen”!

```
var state = 0
(otherActor ? Request) map { case Response => sender !
  RequestComplete }
```

- ▶ Besser?

```
(otherActor ? Request) map { case Response =>
  state += 1; RequestComplete
} pipeTo sender
```

- ▶ So geht's!

```
(otherActor ? Request) map { case Response =>
  self ! IncState
  RequestComplete
} pipeTo sender
```

## (Anti-)Patterns: Single-Responsibility

- ▶ Problem: Fehler in Komplexen Aktoren sind kaum behandelbar

```
var interestDivisor = initial

def receive = {
  case Divide(dividend, divisor) =>
    sender ! Quotient(dividend / divisor)
  case CalculateInterest(amount) =>
    sender ! Interest(amount / interestDivisor)
  case AlterInterest(by) =>
    interestDivisor ← by
}
```

## (Anti-)Patterns: Single-Responsibility

- ▶ Problem: Fehler in Komplexen Aktoren sind kaum behandelbar

```
var interestDivisor = initial

def receive = {
  case Divide(dividend, divisor) =>
    sender ! Quotient(dividend / divisor)
  case CalculateInterest(amount) =>
    sender ! Interest(amount / interestDivisor)
  case AlterInterest(by) =>
    interestDivisor ← by
}
```

- ▶ Welche Strategie bei DivByZeroException?

## (Anti-)Patterns: Single-Responsibility

- ▶ Problem: Fehler in Komplexen Aktoren sind kaum behandelbar

```
var interestDivisor = initial

def receive = {
  case Divide(dividend, divisor) =>
    sender ! Quotient(dividend / divisor)
  case CalculateInterest(amount) =>
    sender ! Interest(amount / interestDivisor)
  case AlterInterest(by) =>
    interestDivisor ← by
}
```

- ▶ Welche Strategie bei DivByZeroException?
- ▶ Ein Akteur sollte immer nur **eine** Aufgabe haben!

# (Anti-)Patterns: Akteur-Beziehungen

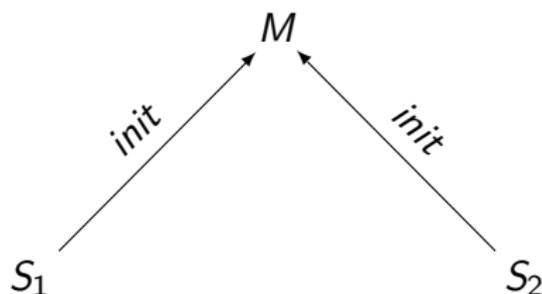
*M*

*S<sub>1</sub>*

*S<sub>2</sub>*

- ▶ Problem: Wer registriert sich bei wem in einer Master-Slave-Hierarchie?

# (Anti-)Patterns: Akteur-Beziehungen



- ▶ Problem: Wer registriert sich bei wem in einer Master-Slave-Hierarchie?
- ▶ Slaves sollten sich beim Master registrieren!
  - ▶ Flexibel / Dynamisch
  - ▶ Einfachere Konfiguration in verteilten Systemen

# (Anti-)Patterns: Aufgabenverteilung

- ▶ Problem: Nach welchen Regeln soll die Aktorhierarchie aufgebaut werden?

# (Anti-)Patterns: Aufgabenverteilung

- ▶ Problem: Nach welchen Regeln soll die Aktorhierarchie aufgebaut werden?
- ▶ **Wichtige** Informationen und zentrale Aufgaben sollten möglichst nah an der Wurzel sein.
- ▶ **Gefährliche** bzw. unsichere Aufgaben sollten immer Kindern übertragen werden.

## (Anti-)Patterns: Zustandsfreie Aktoren

- ▶ Ein Aktor ohne Zustand

```
class Calculator extends Actor {  
  def receive = {  
    case Divide(x,y) => sender ! Result(x / y)  
  }  
}
```

## (Anti-)Patterns: Zustandsfreie Aktoren

- ▶ Ein Aktor ohne Zustand

```
class Calculator extends Actor {  
  def receive = {  
    case Divide(x,y) => sender ! Result(x / y)  
  }  
}
```

- ▶ Ein Fall für Käpt'n Future!

```
class UsesCalculator extends Actor {  
  def receive = {  
    case Calculate(Divide(x,y)) =>  
      Future(x/y) pipeTo self  
    case Result⊗ =>  
      println("Got it: " + x)  
  }  
}
```

## (Anti-)Pattern: Initialisierung

- ▶ Problem: Akteur benötigt Informationen bevor er mit der eigentlichen Arbeit loslegen kann

## (Anti-)Pattern: Initialisierung

- ▶ Problem: Akteur benötigt Informationen bevor er mit der eigentlichen Arbeit loslegen kann
- ▶ Lösung: Parametrisierter Zustand

```
class Robot extends Actor {  
  def receive = uninitialized  
  def uninitialized: Receive = {  
    case Init(pos,power) =>  
      context.become(initialized(pos,power))  
  }  
  def initialized(pos: Point, power: Int): Receive = {  
    case Move(North) =>  
      context.become(initialized(pos + (0,1), power - 1))  
  }  
}
```

# (Anti-)Patterns: Kontrollnachrichten

- ▶ Problem: Akteur mit mehreren Zuständen behandelt bestimmte Nachrichten in jedem Zustand gleich

# (Anti-)Patterns: Kontrollnachrichten

- ▶ Problem: Akteur mit mehreren Zuständen behandelt bestimmte Nachrichten in jedem Zustand gleich
- ▶ Lösung: Verkettete partielle Funktionen

```
class Obstacle extends Actor {  
  def rejectMoveTo: Receive = {  
    case MoveTo => sender ! Reject  
  }  
  def receive = uninitialized orElse rejectMoveTo  
  def uninitialized: Receive = ...  
  def initialized: Receive = ...  
}
```

## (Anti-)Patterns: Circuit Breaker

- ▶ Problem: Wir haben eine elastische, reaktive Anwendung aber nicht genug Geld um eine unbegrenzt große Server Farm zu betreiben.
- ▶ Lösung: Bei Überlastung sollten Anfragen nicht mehr verarbeitet werden.

```
class DangerousActor extends Actor with ActorLogging {  
  val breaker =  
    new CircuitBreaker(context.system.scheduler,  
      maxFailures = 5,  
      callTimeout = 10.seconds,  
      resetTimeout = 1.minute).onOpen(notifyMeOnOpen())  
  
  def notifyMeOnOpen(): Unit =  
    log.warning("My CircuitBreaker is now open, and will  
      not close for one minute")  
}
```

# (Anti)-Patterns: Message Transformer

```
class MessageTransformer(from: ActorRef, to: ActorRef,  
  transform: PartialFunction[Any,Any]) extends Actor {  
  
  def receive = {  
    case m => to forward transform(m)  
  }  
}
```

# Weitere Patterns

- ▶ Lange Aufgaben unterteilen
- ▶ Aktor Systeme sparsam erstellen
- ▶ Futures sparsam einsetzen
- ▶ `Await.result()` **nur** bei Interaktion mit Nicht-Aktor-Code
- ▶ Dokumentation Lesen!

# Zusammenfassung

- ▶ Nachrichtenaustausch in verteilten Systemen ist unzuverlässig
- ▶ Zwei Armeen Problem
- ▶ Lösungsansätze
  - ▶ Drei-Wege Handschlag
  - ▶ Nachrichtennummerierung
  - ▶ Heartbeats
  - ▶ Gossiping Protokolle
- ▶ Patterns und Anti-Patterns
- ▶ Nächstes mal: Theorie der Nebenläufigkeit

Reaktive Programmierung  
Vorlesung 16 vom 10.07.2019  
Theorie der Nebenläufigkeit

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

# Fahrplan

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

# Theorie der Nebenläufigkeit

- ▶ Nebenläufige Systeme sind **kompliziert**
  - ▶ Nicht-deterministisches Verhalten
  - ▶ Neue Fehlerquellen wie **Deadlocks**
  - ▶ Schwer zu testen
- ▶ Reaktive Programmierung kann diese Fehlerquellen **einhegen**
- ▶ **Theoretische Grundlagen** zur Modellierung nebenläufiger Systeme
  - ▶ zur **Spezifikation** (CSP)
  - ▶ aber auch als **Berechnungsmodell** ( $\pi$ -Kalkül)

# Temporale Logik, Prozessalgebren und Modelchecking

- ▶ Prozessalgebren und temporale Logik beschreiben **Systeme** anhand ihrer **Zustandsübergänge**
- ▶ Ein System ist dabei im wesentlichen eine **endliche Zustandsmaschine**  $\mathcal{M} = \langle S, \Sigma, \rightarrow \rangle$  mit Zustandsübergang  $\rightarrow \subseteq S \times \Sigma \times S$
- ▶ Temporale Logiken reden über **eine** Zustandsmaschine
- ▶ Prozessalgebren erlauben **mehrere** Zustandsmaschinen und ihre **Synchronisation**
- ▶ Der Trick ist **Abstraktion**: mehrere interne Zustandsübergänge werden zu einem Zustandsübergang zusammengefaßt

# Einfache Beispiele

- ▶ Einfacher Kaffee-Automat:

$$P = 10c \rightarrow \text{coffee} \rightarrow P$$

- ▶ Kaffee-Automat mit Auswahl:

$$P = 10c \rightarrow \text{coffee} \rightarrow P \sqcap 20c \rightarrow \text{latte} \rightarrow P$$

- ▶ Pufferprozess:

$$COPY = \text{left}?x \rightarrow \text{right}!x \rightarrow COPY$$

NB. Eingabe ( $c?x$ ) und Ausgabe ( $c!x$ ) sind **reine Konvention**.

# CSP: Syntax

Gegeben Prozeßalphabet  $\Sigma$ , besondere Ereignisse  $\checkmark, \tau$

$P ::= Stop$		$a \rightarrow P$		$\mu P.F(P)$	fundamentale Operationen
	$P \square Q$		$P \sqcap Q$		externe und interne Auswahl
	$P \parallel Q$		$P \parallel_X Q$		synchronisiert parallel
	$P \parallel\parallel Q$				unsynchronisiert parallel
	$P \setminus X$				hiding
	$Skip$		$P; Q$		sequentielle Komposition

# Externe vs. interne Auswahl

- ▶ Interne Zustandsübergänge ( $\tau$ ) sind **nicht beobachtbar**, aber können Effekte haben.
- ▶ Vergleiche:

$$a \rightarrow b \rightarrow Stop \sqcap a \rightarrow c \rightarrow Stop$$

$$a \rightarrow b \rightarrow Stop \sqcap a \rightarrow c \rightarrow Stop$$

$$a \rightarrow (b \rightarrow Stop \sqcap c \rightarrow Stop)$$

$$a \rightarrow (b \rightarrow Stop \sqcap c \rightarrow Stop)$$

# Beispiel: ein Flugbuchungssystem

- ▶ Operationen des Servers:
  - ▶ Nimmt Anfragen an, schickt Resultate (mit flid)
  - ▶ Nimmt Buchungsanfragen an, schickt Bestätigung (ok) oder Fehler (fail)
  - ▶ Nimmt Stornierung an, schickt Bestätigung
- ▶ Unterschied zwischen **interner** Auswahl  $\sqcap$  (Server trifft Entscheidung), und **externer** Auswahl  $\square$  (Server reagiert)

$SERVER = query?(from, to) \rightarrow result!flid \rightarrow SERVER$   
 $\square booking?flid \rightarrow (ok \rightarrow SERVER \sqcap fail \rightarrow SERVER)$   
 $\square cancel?flid \rightarrow ok \rightarrow SERVER$

Eingabe ( $c?x$ ) und Ausgabe ( $c!a$ ) sind reine **Konvention**

# Beispiel: ein Flugbuchungssystem

- ▶ Operationen des Servers:
  - ▶ Nimmt Anfragen an, schickt Resultate (mit flid)
  - ▶ Nimmt Buchungsanfragen an, schickt Bestätigung (ok) oder Fehler (fail)
  - ▶ Nimmt Stornierung an, schickt Bestätigung
- ▶ Unterschied zwischen **interner** Auswahl  $\sqcap$  (Server trifft Entscheidung), und **externer** Auswahl  $\square$  (Server reagiert)

$SERVER = query \rightarrow result \rightarrow SERVER$   
 $\square booking \rightarrow (ok \rightarrow SERVER \sqcap fail \rightarrow SERVER)$   
 $\square cancel \rightarrow ok \rightarrow SERVER$

Eingabe ( $c?x$ ) und Ausgabe ( $c!a$ ) sind reine **Konvention**

# Beispiel: ein Flugbuchungssystem

- ▶ Der Client:

- ▶ Stellt Anfrage
- ▶ wenn der Flug richtig ist, wird er gebucht;
- ▶ oder es wird eine neue Anfrage gestellt.

$$CLIENT = query \rightarrow result \rightarrow$$
$$(booking \rightarrow ok \rightarrow CLIENT$$
$$\sqcap CLIENT)$$

- ▶ Das Gesamtsystem — Client und Server **synchronisiert**:

$$SYSTEM = CLIENT \parallel SERVER$$

# Beispiel: ein Flugbuchungssystem

- ▶ Der Client:

- ▶ Stellt Anfrage
- ▶ wenn der Flug richtig ist, wird er gebucht;
- ▶ oder es wird eine neue Anfrage gestellt.

$$CLIENT = query \rightarrow result \rightarrow$$
$$(booking \rightarrow ok \rightarrow CLIENT$$
$$\sqcap CLIENT)$$

- ▶ Das Gesamtsystem — Client und Server **synchronisiert**:

$$SYSTEM = CLIENT \parallel SERVER$$

- ▶ Problem: **Deadlock**

- ▶ Es gibt **Werkzeuge** (Modelchecker, z.B. FDR), um solche Deadlocks in Spezifikationen zu finden

# Beispiel: ein Flugbuchungssystem

- ▶ Der Client:

- ▶ Stellt Anfrage
- ▶ wenn der Flug richtig ist, wird er gebucht;
- ▶ oder es wird eine neue Anfrage gestellt.

$$\begin{aligned} CLIENT &= query \rightarrow result \rightarrow \\ &\quad (booking \rightarrow (ok \rightarrow CLIENT \\ &\quad \quad \square fail \rightarrow CLIENT)) \\ &\quad \square CLIENT) \end{aligned}$$

- ▶ Das Gesamtsystem — Client und Server **synchronisiert**:

$$SYSTEM = CLIENT \parallel SERVER$$

- ▶ Problem: **Deadlock**

- ▶ Es gibt **Werkzeuge** (Modelchecker, z.B. FDR), um solche Deadlocks in Spezifikationen zu finden

# Ziele der Semantik von Prozesskalkülen

- ▶ Reasoning about processes by their external behaviour
- ▶ Untersuchung von
  - ▶ Verfeinerung (Implementation)
  - ▶ **deadlock**: Keine Transition möglich
  - ▶ **livelock**: Divergenz
- ▶ Grundlegender Begriff: **Äquivalenz (Gleichheit) von Prozessen**

# Operationale Semantik für CSP (I)

## Definition: Labelled Transition System (LTS)

Ein **labelled transition system (LTS)** ist  $L = (N, A, \rightarrow)$  mit Menge  $N$  der Knoten (Zustände), Menge  $A$  von Labels und Relation  $\{\overset{a}{\rightarrow} \subseteq N \times N\}_{a \in A}$  von Kanten (Zustandsübergänge).

Hier:  $N = P, A = \Sigma \cup \{\surd, \tau\}$ ,  $\rightarrow$  definiert wie folgt:

$$\frac{}{e \rightarrow P \xrightarrow{a} P[a/e]} \quad a \in \text{comms}(e)$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P}$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

# Operationale Semantik für CSP (II)

$$\frac{P \xrightarrow{\tau} P'}{P \square Q \xrightarrow{\tau} P' \square Q}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \square Q \xrightarrow{\tau} P \square Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \square Q \xrightarrow{a} P'} \quad a \neq \tau$$

$$\frac{Q \xrightarrow{a} Q'}{P \square Q \xrightarrow{a} Q'} \quad a \neq \tau$$

$$\frac{P \xrightarrow{x} P'}{P \setminus B \xrightarrow{\tau} P'} \quad x \in B$$

$$\frac{P \xrightarrow{x} P'}{P \setminus B \xrightarrow{x} P' \setminus B} \quad x \notin B$$

# Operationale Semantik für CSP (III)

$$\frac{P \xrightarrow{\tau} P'}{P \parallel_X Q \xrightarrow{\tau} P' \parallel_X Q}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \parallel_X Q \xrightarrow{\tau} P \parallel_X Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \parallel_X Q \xrightarrow{a} P' \parallel_X Q} \quad a \notin X$$

$$\frac{Q \xrightarrow{a} Q'}{P \parallel_X Q \xrightarrow{a} P \parallel_X Q'} \quad a \notin X$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \parallel_X Q \xrightarrow{a} P' \parallel_X Q'} \quad a \in X$$

# Denotationale Semantik für CSP

- ▶ **Operationale** Semantik erklärt das **Verhalten**, erlaubt kein **Reasoning**
- ▶ **Denotationale** Semantik erlaubt **Abstraktion** über dem Verhalten
- ▶ Für CSP: Denotat eines Prozesses ist:
  - ▶ die Menge aller seiner **Traces**
  - ▶ die Menge seiner **Traces** und **Acceptance-Mengen**
  - ▶ die Menge seiner **Traces** und seiner **Failure/Divergence-Mengen**

# Anwendungsgebiete für CSP

- ▶ Modellierung nebenläufiger Systeme (Bsp: ISS)
- ▶ Verteilte Systeme und verteilte Daten
- ▶ Analyse von Krypto-Protokollen
- ▶ Hauptwerkzeug: der Modellchecker **FDR**
  - ▶ <http://www.cs.ox.ac.uk/projects/fdr/>