

## Reaktive Programmierung Vorlesung 1 vom 19.04.2022 Was ist Reaktive Programmierung?

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

13:50:40 2022-07-05

1 [42]



## Organisatorisches

- ▶ Vorlesung: Di 12-14, MZH 1470
- ▶ Übung: Mi 12-14, MZH 1110 (nach Bedarf)
- ▶ Webseite: [www.informatik.uni-bremen.de/~cxl/lehre/rp.ss22](http://www.informatik.uni-bremen.de/~cxl/lehre/rp.ss22)
- ▶ Scheinkriterien:
  - ▶ Voraussichtlich 6 Übungsblätter
  - ▶ Alle bearbeitet, insgesamt 40% (Notenspiegel PI3)
  - ▶ Übungsgruppen 2 – 4 Mitglieder
  - ▶ **Danach:** Fachgespräch **oder** Modulprüfung

RP SS 2022

2 [42]



## Warum Reaktive Programmierung?

Herkömmliche Sprachen:

- ▶ PHP, JavaScript, Ruby, Python
- ▶ C, C++, Java

Eigenschaften:

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

Zugrundeliegendes Paradigma:



... aber die Welt ändert sich:



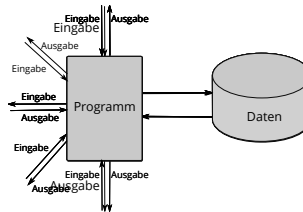
- ▶ Das **Netz** verbindet Rechner
- ▶ Selbst eingebettete Systeme sind vernetzt (Auto: ca. 130 Proz.)
- ▶ Mikroprozessoren sind **mehrkernig**
- ▶ Systeme sind **eingebettet, nebenläufig, reagieren** auf ihre Umwelt.

RP SS 2022

3 [42]



## Probleme mit dem herkömmlichen Ansatz



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

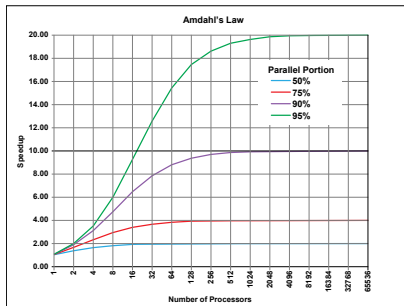
RP SS 2022

4 [42]



## 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 20x as shown in the diagram, no matter how many processors are used."



Quelle: Wikipedia

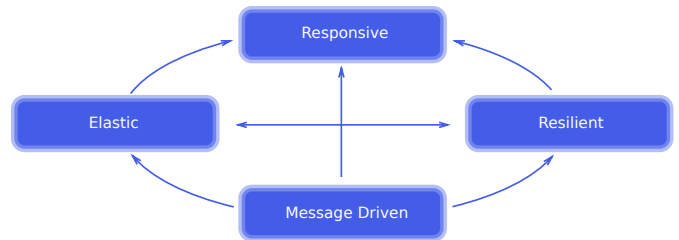
RP SS 2022

5 [42]



## The Reactive Manifesto

- ▶ <http://www.reactivemanifesto.org/>



RP SS 2022

6 [42]



## Was ist Reaktive Programmierung?

- ▶ **Imperative** Programmierung: Zustandsübergang
- ▶ **Prozedural** und **OO**: Verkapselter Zustand
- ▶ **Funktionale** Programmierung: Abbildung (mathematische Funktion)
- ▶ **Reaktive** Programmierung:
  - ④ **Datenabhängigkeit**
  - ④ **Reaktiv** = funktional + nebenläufig

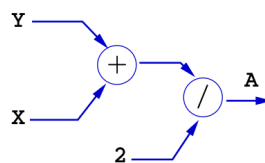
RP SS 2022

7 [42]



## 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
```

RP SS 2022

8 [42]



## 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: Grundlagen & Implementierung
- ▶ Bidirektionale Programmierung
- ▶ Meta-Programmierung
- ▶ Reaktive Ströme I
- ▶ Reaktive Ströme II
- ▶ Funktional-Reaktive Programmierung
- ▶ Software Transactional Memory
- ▶ Eventual Consistency
- ▶ CRDTs
- ▶ Robustheit, Entwurfsmuster und Theorie der Nebenläufigkeit, Abschluss
- ▶ Reaktive Programmierung in der Praxis

# I. Rückblick Haskell

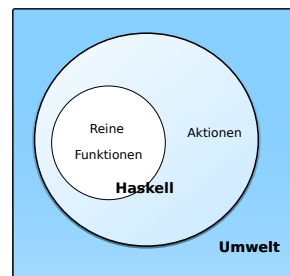
## Rückblick Haskell

- ▶ 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: `[α]`, `(α, β)`
  - ▶ Algebraische Datentypen: `data Maybe α = Just α | 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 ??`
- 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 α
(⟨=>) :: IO α -> (α -> IO β) -> IO β
return :: α -> IO α
```
- ▶ 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
  >>= \s → 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.

## II. 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  :: ((α, β) → γ) → α → β → γ
uncurry :: (α → β → γ) → (α, β) → γ
    
```

## In Haskell: Zustände explizit

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

```

type State σ α = σ → (α, σ)
    
```

- Komposition zweier solcher Berechnungen:

```

comp :: State σ α → (α → State σ β) → State σ β
comp f g = uncurry g ∘ f
    
```

- Trivialer Zustand:

```

lift :: α → State σ α
lift = curry id
    
```

- Lifting von Funktionen:

```

map :: (α → β) → State σ α → State σ β
map f g = (λ(a, s) → (f a, s)) ∘ g
    
```

## Zugriff auf den Zustand

- Zustand lesen:

```

get :: (σ → α) → State σ α
get f s = (f s, s)
    
```

- Zustand setzen:

```

set :: (σ → σ) → State σ ()
set g s = ((), g s)
    
```

## Einfaches Beispiel

- Zähler als Zustand:

```

type WithCounter α = State Int α
    
```

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

```

cntToL :: String → WithCounter String
cntToL [] = lift ""
cntToL (x:xs)
  | isUpper x = cntToL xs 'comp'
               λys → set (+1) 'comp'
               λ() → lift (toLower x: ys)
  | otherwise = cntToL xs 'comp' λys → lift (x: ys)
    
```

- Hauptfunktion (verkapselt State):

```

cntToLower :: String → (String, Int)
cntToLower s = cntToL s 0
    
```

## III. Monaden

## Monaden als Berechnungsmuster

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

State:

```

type State σ α
    
```

```

comp :: State σ α →
      (α → State σ β) →
      State σ β
    
```

```

lift :: α → State σ α
    
```

```

map :: (α → β) → State σ α →
      State σ β
    
```

Berechnungsmuster: **Monade**

Aktionen:

```

type IO α
    
```

```

(⋈) :: IO α →
      (α → IO β) →
      IO β
    
```

```

return :: α → IO α
    
```

```

fmap :: (α → β) → IO α →
      IO β
    
```

## Monaden als Berechnungsmuster

Eine Monade ist:

- ▶ **philosophisch**: metaphysisches Konzept (Leibnitz)
- ▶ **mathematisch**: durch Operationen und Gleichungen definierte, verallgemeinerte algebraische Theorie (MacLane, Kelly)
- ▶ als **Berechnungsmuster**: **verknüpfbare** Berechnungen mit einem **Ergebnis** (Moggi)
- ▶ in **Haskell**: durch mehrere Typklassen definierte Operationen mit **Eigenschaften** (Wadler)

RP SS 2022

25 [42]



## 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 o g) == fmap f o fmap g
```

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

RP SS 2022

26 [42]



## 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
```

RP SS 2022

27 [42]



## Monaden in Haskell

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

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

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

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

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

RP SS 2022

28 [42]



## Monaden mit Möglichkeiten

▶ Alternativen:

```
class Applicative f => Alternative f where
  empty :: f a
  <|> :: f a -> f a -> f a
```

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

```
class (Alternative m, Monad m) => MonadPlus m where
  mzero :: m a
  mzero = empty
  mplus :: m a -> m a -> m a
  mplus = (<|>)
```

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

RP SS 2022

29 [42]



## Beispiele für Monaden

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

RP SS 2022

30 [42]



## Die Reader-Monade

▶ Aus dem Zustand wird nur gelesen:

```
data Reader sigma alpha = R {run :: sigma -> 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 >>= g = R $ \s -> run (g (f s)) s
```

▶ Nur eine elementare Operation:

```
get :: (sigma -> alpha) -> Reader sigma alpha
get f = R $ \s -> f s
```

RP SS 2022

31 [42]



## 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)
- ▶  $f :: alpha -> Maybe beta$  ist Berechnung mit möglichem Fehler
- ▶ Fehlerfreie Berechnungen werden verkettet
- ▶ Fehler (Nothing oder Left x) werden propagiert

RP SS 2022

32 [42]



## Mehrdeutigkeit

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

```
instance Functor [α] where
  fmap = map

instance Monad [α] where
  a : as >>= g = g a ++ (as >>= g)
  [] >>= 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`)

RP SS 2022

33 [42]



## Beispiel

- ▶ Berechnung aller Permutationen einer Liste:
  - ① Ein Element überall in eine Liste einfügen:

```
ins :: α → [α] → [[α]]
ins x [] = return [x]
ins x (y:ys) = [x:y:ys] ++ do
  is ← ins x ys
  return $ y:is
```

- ② Damit Permutationen (rekursiv):

```
perms :: [α] → [[α]]
perms [] = return []
perms (x:xs) = do
  ps ← perms xs
  is ← ins x ps
  return is
```

RP SS 2022

34 [42]



## Der Listenmonade in der Listenkomprehension

- ▶ Berechnung aller Permutationen einer Liste:
  - ① Ein Element überall in eine Liste einfügen:

```
ins' :: α → [α] → [[α]]
ins' x [] = [[x]]
ins' x (y:ys) = [x:y:ys] ++ map (y :) (ins' x ys)
```

- ② Damit Permutationen (rekursiv):

```
perms' :: [α] → [[α]]
perms' [] = [[]]
perms' (x:xs) = [is | ps ← perms' xs, is ← ins' x ps]
```

- ▶ Listenkomprehension  $\cong$  Listenmonade

RP SS 2022

35 [42]



## Zum Nachdenken

Listen sind mehrdeutige Berechnungen mit einer **Reihenfolge**.

- ① Was bedeutet das?
- ② Wie können wir das ändern?
- ③ Wie implementieren wir das?

RP SS 2022

36 [42]



# IV. IO ist keine Magie

RP SS 2022

37 [42]



## Referenzen in Haskell

- ▶ Zustand als **finite map** von Referenzen auf Werte

```
data Mem α = Mem { fresh :: Int, mem :: Map Int α }
```

```
type Stateful α β = State (Mem α) β
```

- ▶ Ungetypt (`SimpleRefs`)
- ▶ Typ der Werte ist Typparameter des Zustands

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

RP SS 2022

38 [42]



## Zum Nachdenken

- ▶ Wie können wir Referenzen **typisieren**?
  - ▶ Referenz `Ref α` verweist auf Werte vom Typ  $\alpha$
- ▶ Was ist das Problem? Der Typ von `Mem`:

```
data Mem α = Mem { fresh :: Int, mem :: Map Int α }
```

- ▶ Getypt (`Refs`): nutzt **dynamische Typen** (`Dynamic`)

```
data Mem = Mem { fresh :: Int, mem :: Map Int Dynamic }
```

```
readRef :: Typeable α => Ref α → Stateful α α
writeRef :: Typeable α => Ref α → α → Stateful α ()
```

RP SS 2022

39 [42]



## 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

RP SS 2022

40 [42]



## 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 26.04.2022  
Monaden und Monadentransformer

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

### Fahrplan

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

### Inhalt

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

## I. Monaden

### Beispiele für Monaden

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

## II. 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
```

#### ▶ 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
```

#### ▶ Mögliche Arten von Effekten:

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

### 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
```

RP SS 2022

9 [21]



## 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]
```

RP SS 2022

10 [21]



## Kombination der Effekte

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

```
data Res σ α = Res { run :: σ -> [Maybe α] }
```

- Andere Kombinationen möglich:

```
data Res σ α = Res (σ -> Maybe [α])
```

```
data Res σ α = Res (σ -> [α])
```

```
data Res σ α = Res ([σ -> α])
```

RP SS 2022

11 [21]



## Res: Monadeninstanz

- Functor durch Komposition der fmap:

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

- Monad ist Kombination

```
instance Monad (Res σ) where
  return a = Res (const [Just a])
  Res f >=> g = Res $ \s -> do ma<- f s
                    case ma of
                      Just a -> run (g a) s
                      Nothing -> return Nothing
```

RP SS 2022

12 [21]



## Res: Operationen

- Zugriff auf den Zustand:

```
get :: (σ -> α) -> Res σ α
get f = Res $ \s -> [Just $ f s]
```

- Fehler:

```
fail :: Res σ α
fail = Res $ const [Nothing]
```

- Mehrdeutige Ergebnisse:

```
join :: α -> α -> Res σ α
join a b = Res $ \s -> [Just a, Just b]
```

RP SS 2022

13 [21]



## 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

RP SS 2022

14 [21]



## III. 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.

RP SS 2022

15 [21]



RP SS 2022

16 [21]





## 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: `mt1`

- ▶ 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
  - ▶ `mt1`-Bücherei erleichtert Kombination
  - ▶ Prinzipielle Begrenzungen
- ▶ Grenze: Nebenläufigkeit → Nächste Vorlesung

Reaktive Programmierung  
Vorlesung 3 vom 03.05.2022  
Nebenläufigkeit: Futures and Promises

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

13:50:45 2022-07-05

1 [26]



## Fahrplan

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

RP SS 2022

2 [26]



## Inhalt

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

RP SS 2022

3 [26]



# I. Konzepte der Nebenläufigkeit

RP SS 2022

4 [26]



## Begrifflichkeiten

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

RP SS 2022

5 [26]



## 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`)

RP SS 2022

6 [26]



## 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)

RP SS 2022

7 [26]



## 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 Contextswitch sind **billig**
- ▶ Modul `Control.Concurrent` enthält Basisfunktionen
- ▶ Wenige Basisprimitive, darauf aufbauend Abstraktionen
- ▶ Synchronisation mit Futures

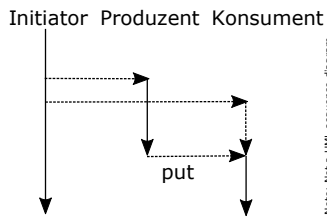
RP SS 2022

8 [26]



## Futures

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



RP SS 2022

9 [26]



## II. Futures in Scala

RP SS 2022

10 [26]



## 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]
```

```
object Future:  
  def apply[T](f: => T): Future[T] = ...
```

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

RP SS 2022

11 [26]



## Beispiel: Robot.scala

- ▶ Roboter, kann sich um `n` Positionen bewegen:

```
private def mv(n: Int): Robot =  
  if n <= 0 then this  
  else if (battery > 0) then  
    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]"
```

RP SS 2022

12 [26]



## Beispiel: Moving the robots

```
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`

RP SS 2022

13 [26]



## 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)  
  yield r3
```

- ▶ Fehler (`Failure`) werden propagiert

RP SS 2022

14 [26]



## 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**.

RP SS 2022

15 [26]



## III. Futures in Haskell

RP SS 2022

16 [26]



## 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 ()`

RP SS 2022

17 [26]



## 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*)*`

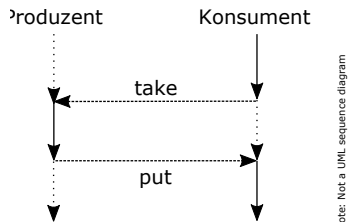
RP SS 2022

18 [26]



## Futures in Haskell: MVars

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



RP SS 2022

19 [26]



## 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	blockiert (bis gefüllt)	danach leer
Schreiben	danach gefüllt	blockiert (bis leer)

- ▶ NB. Aufwecken blockierter Prozesse einzeln in FIFO

RP SS 2022

20 [26]



## Basisfunktionen MVars

- ▶ Neue Variable erzeugen (leer oder gefüllt):

```
newEmptyMVar :: IO (MVar  $\alpha$ )
newMVar ::  $\alpha$  -> IO (MVar  $\alpha$ )
```

- ▶ Lesen:

```
takeMVar :: MVar  $\alpha$  -> IO  $\alpha$ 
```

- ▶ Schreiben:

```
putMVar :: MVar  $\alpha$  ->  $\alpha$  -> IO ()
```

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

RP SS 2022

21 [26]



## 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
  f <- newEmptyMVar; forkIO (mv f r n); return f where
```

- ▶ Bewegungsfunktion (lokal zu move):

```
mv f r n
  | n <= 0 || battery r <= 0 = putMVar f r
  | otherwise = do
    m <- randomRIO(0,10); threadDelay(m*100000)
    putStrLn $ "Bleep, bleep: "+ show r
    mv f r{pos= pos r + 1, battery= battery r - 1} (n-1)
```

RP SS 2022

22 [26]



## 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  $\alpha$ 
```

RP SS 2022

23 [26]



## Asynchrone Ausnahmen

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

```
m <- newEmptyMVar
forkIO (do {s <- takeMVar m; putStrLn s})
threadDelay (100000)
catcherr $ 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
```

RP SS 2022

24 [26]



## Explizite Fehlerbehandlung mit Try

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

```
private def mv(n: Int): Robot =
```

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

```
enum Try[+T]:  
  case Success(x: T)  
  case Failure(ex: Throwable)  
  
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
```

- ▶ 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**
- ▶ Nächste VL: das Aktorenmodell

# Reaktive Programmierung Vorlesung 4 vom 10.05.2022 Aktoren

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

## Fahrplan

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

## Das Aktorenmodell



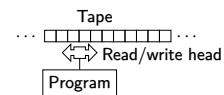
- ▶ Eingeführt von Carl Hewitt, Peter Bishop und Richard Steiger (1973)
- ▶ Grundlage für nebenläufige Programmiersprachen und Frameworks. (Unter anderem Akka)
- ▶ Theoretisches Berechnungsmodell

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

## Die Turingmaschine



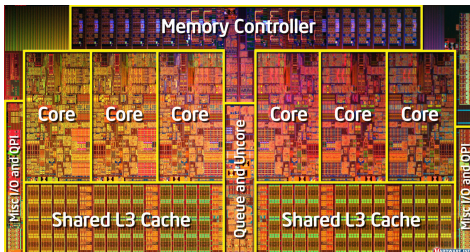
“the behavior of the computer at any moment is determined by the symbols which he [the computer] is observing, and his ‘state of mind’ at that moment”  
— Alan Turing



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

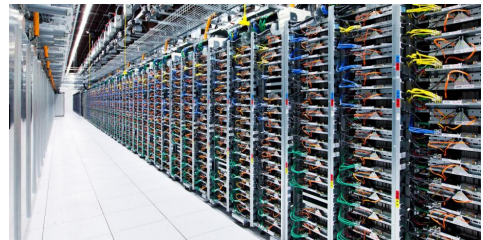


## Die Realität



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

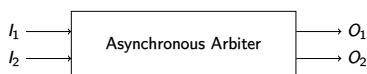
## Synchronisation



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

## Der Arbitrer

- ▶ Die Lösung: **Asynchrone Arbitrer**



- ▶ 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 Arbitrer 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 Arbitrer 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!)

RP SS 2022

9 [35]



## 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

- ▶  $\text{Aktor} \neq (\text{Thread} \mid \text{Task} \mid \text{Channel} \mid \dots)$

### Ein Aktor kann (darf) **nicht**

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

RP SS 2022

10 [35]



## 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

RP SS 2022

11 [35]



## Verhalten vs. Protokoll

### Verhalten

Das Verhalten eines Aktors ist eine seiteneffektbehaftete Funktion  
 $\text{Behavior} : \text{Msg} \rightarrow \text{IO Behavior}$

### 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
```

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

RP SS 2022

12 [35]



## 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  $\neq (\text{Queue} \mid \text{Lock} \mid \text{Channel} \mid \dots)$

RP SS 2022

13 [35]



## Kommunikation (Technisch)

- ▶ Der Versand einer Nachricht  $M$  an Aktor  $A$  bewirkt, dass zu **höchstens 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 Nachrichtenempfang gilt  $P(S)$
- ▶ Besser: Protokoll
  - ▶ z.B. auf Nachrichten des Typs  $T$  reagiert  $A$  immer mit Nachrichten des Typs  $U$

RP SS 2022

14 [35]



## Identifikation

- ▶ Aktoren werden über **Identitäten** angesprochen

### Aktoren kennen Identitäten

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

RP SS 2022

15 [35]



## 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
  - ▶ ...

RP SS 2022

16 [35]



## 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 potentielles Sicherheitsproblem
- ▶ Viele Implementierungen stellen **Authentisierungsverfahren** und **verschlüsselte Kommunikation** zur Verfügung.

RP SS 2022

17 [35]



## 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**)

RP SS 2022

18 [35]



## 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 ...

RP SS 2022

19 [35]



## 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

RP SS 2022

20 [35]



## "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



RP SS 2022

21 [35]



## 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, ...

RP SS 2022

22 [35]



## 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

RP SS 2022

23 [35]



## Aktoren in Scala

- ▶ Eine kurze Geschichte von Akka:
  - ▶ 2006: Aktoren in der Scala Standardbücherei (Philipp Haller, `sca1a.actors`)
  - ▶ 2010: Akka 0.5 wird veröffentlicht (Jonas Bonér)
  - ▶ 2012: Scala 2.10 erscheint ohne `sca1a.actors` und Akka wird Teil der Typesafe Plattform
- ▶ Auf Akka aufbauend:
  - ▶ Apache Spark
  - ▶ Play! Framework
  - ▶ Akka HTTP (Früher Spray Framework)

RP SS 2022

24 [35]





## 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

```
abstract class Behavior[-T]:  
  def receive(ctx: ActorContext[T], msg: T): Behavior[T]  
  def receiveSignal(ctx: ActorContext[T], msg: Signal): Behavior[T]
```

## Aktoren Erzeugen

```
enum Message  
  case Count  
  
def counter(count: Int) = Behaviors.receiveMessage  
  case Message.Count => counter(count + 1)
```

Global:

```
val system = ActorSystem(counter(0), "counter")
```

In Aktoren:

```
val counter = context.spawn(counter(0), "counter")
```

## Nachrichtenversand

```
enum Message:  
  case Count  
  case Get(sender: ActorRef[Int])  
  
def counter(count: Int) = Behaviors.receive  
  case Message.Count => counter(count + 1)  
  case Message.Get(sender) =>  
    sender ! count  
    Behaviors.same  
)
```

```
val aCounter = context.spawn(counter(0), "counter")  
aCounter ! 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 Nachrichtenempfangs undefiniert.
- ▶ Nachrichten sollen unveränderlich sein. (Das kann derzeit allerdings nicht überprüft werden)

## 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.

## Supervision und Fehlerbehandlung in Akka

```
Behaviors.supervise(counter(0))  
  .onFailure[ArithmeticException]({  
    SupervisorStrategy.restart.withLimit(maxNrOfRetries = 10,  
      withinTimeRange = 10.seconds)  
  })
```

## Aktorsysteme Testen

### ▶ Tests mit TestKit

```
"A counter" must {
  "be able to count to three" in {
    val c = testKit.spawn(counter(0))
    val p = testKit.createTestProbe[Int]()
    c ! Count
    c ! Count
    c ! Count
    c ! Get(p)
    createTestProbe[Echo.Pong](3)
  }
}
```

## Bewertung

### ▶ Vorteile:

- ▶ Nah am Aktorenmodell (Carl-Hewitt-approved)
- ▶ Aber nicht zu nahe (Referenzen sind typisiert)
- ▶ keine Race Conditions
- ▶ Effizient
- ▶ Stabil und ausgereift
- ▶ Umfangreiche Konfigurationsmöglichkeiten

### ▶ Nachteile:

- ▶ 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 5 vom 17.05.2022  
Bidirektionale Programmierung: Zippers and Lenses

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

13:50:51 2022-07-05

1 [35]



## Fahrplan

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

RP SS 2022

2 [35]



## 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

RP SS 2022

3 [35]



## 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}
```

RP SS 2022

4 [35]



## Aufwand

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

RP SS 2022

5 [35]



## 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 c) 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 []
```

RP SS 2022

6 [35]



## 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
```

RP SS 2022

7 [35]



## 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

RP SS 2022

8 [35]



## 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)
```

RP SS 2022

9 [35]



## Zipping 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"
```

RP SS 2022

10 [35]



## Zipping 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)
```

RP SS 2022

11 [35]



## 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))
```

RP SS 2022

12 [35]



## Ersetzen und Löschen

Unterbaum im Fokus löschen: wo ist der neue Fokus?

- Rechter Baum, wenn vorhanden
- Linker Baum, wenn vorhanden
- 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."

RP SS 2022

13 [35]



## 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

RP SS 2022

14 [35]



## Zipper für andere Datenstrukturen

Binäre Bäume:

```
enum Tree[+A]:
  case Leaf(value: A)
  case Node(left: Tree[A],
            right: Tree[A])
```

Kontext:

```
enum Context[+A]:
  case object Empty
  case Left(up: Context[A],
            right: Tree[A])
  case Right(left: Tree[A],
             up: Context[A])
```

```
case Loc(tree: Tree[A], context: Context[A])
```

RP SS 2022

15 [35]



## Tree-Zipper: Navigation

Fokus nach **links**

```
def goLeft: Loc[A] = context match
  case Empty => sys.error("goLeft at empty")
  case Left(Left(c, r) => 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(Left(c, r) => sys.error("goRight of right")
```

RP SS 2022

16 [35]



## 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))
```

RP SS 2022

17 [35]



## 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

RP SS 2022

18 [35]



## Zippping Lists

- Listen:

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

- Damit:

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

- Listen sind ihr 'eigener Kontext':

$List\ a \cong Ctxt\ a$

RP SS 2022

19 [35]



## Zippping 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

RP SS 2022

20 [35]



## Bidirektionale Programmierung

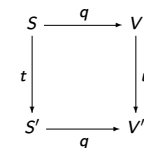
- Verallgemeinerung der Idee des Kontext
- Motivierendes Beispiel: Update in einer Datenbank
- Weitere Anwendungsfelder:
  - Benutzerschnittstellen (MVC)
  - Datensynchronisation

RP SS 2022

21 [35]



## 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?

RP SS 2022

22 [35]



## Lösung

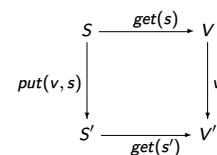
- Eine Operation  $get$  für den View
- Inverse Operation  $put$  wird automatisch erzeugt (wo möglich)
- Beide müssen invers sein — deshalb **bidirektionale Programmierung**

RP SS 2022

23 [35]



## Putting and Getting



- Signatur der Operationen:

```
get  : S -> V
put  : V x S -> S
```

- Es müssen die **Linsengesetze** gelten:

```
get(put(v, s)) = v
put(get(s), s) = s
put(v, put(w, s)) = put(v, s)
```

RP SS 2022

24 [35]



## Erweiterung: Erzeugung

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

- Signatur:

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

- Weitere Gesetze:

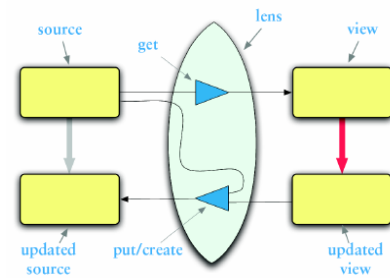
$$\begin{aligned} \text{get}(\text{create}(v)) &= v \\ \text{put}(v, \text{create}(w)) &= \text{create}(w) \end{aligned}$$

RP SS 2022

25 [35]



## Die Linse im Überblick



RP SS 2022

26 [35]



## 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)
```

RP SS 2022

27 [35]



## 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.

RP SS 2022

28 [35]



## Abhilfe mit Linsen

- Zuerst einmal: die **Linse**.

```
object Lenses {
  case class Lens[A, B](
    get: A => B,
    set: (A, B) => A
  )
}
```

- 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))
```

RP SS 2022

29 [35]



## 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

RP SS 2022

30 [35]



## 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

RP SS 2022

31 [35]



## Linsen konstruieren

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

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

- Die **Identitätslinse**:

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

RP SS 2022

32 [35]



## Linsen komponieren

▶ Gegeben Linsen  $L_1 : S_1 \leftrightarrow S_2, L_2 : S_2 \leftrightarrow S_3$

▶ Die Komposition ist definiert als:

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

▶ Beispiel hier:

TurtleX = TurtlePosition · 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: Meta-Programmierung

Reaktive Programmierung  
Vorlesung 6 vom 24.05.2022  
Meta-Programmierung

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

### Fahrplan

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

### Was ist Meta-Programmierung?

"Programme höherer Ordnung" / Makros



### Was sehen wir heute?

- ▶ Anwendungsbeispiel: JSON Serialisierung
- ▶ Meta-Programmierung in Scala:
  - ▶ Inlining
  - ▶ Compile-Time Operations
  - ▶ Typeclass Derivation
- ▶ 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  $\xleftarrow{\text{JSON}}$  Haskell

### JSON: Erster Versuch

JSON1.scala

- ▶ Unpraktisch: Für jeden Typ muss manuell eine Instanz erzeugt werden
- ▶ Idee: Typklassenableitung 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

# I. Metaprogrammierung in Haskell



## Was brauchen wir?

- ▶ Idee: Funktion  $AST \rightarrow AST$  zur Compilezeit ausführen
- ① Repräsentation des AST in der Sprache  $\rightarrow$  Template Haskell
- ② Konversion von und in diese Repräsentation  $\rightarrow$  Template Haskell, Reification und Splicing
- ③ Büchereien für häufige Use-Cases (e.g. JSON)  $\rightarrow$  Generics

## Template Haskell I

- ▶ **Gettype** Repräsentation des AST in Haskell
  - ▶ Vgl. Reflektion in Java
  - ▶ Datentyp `Exp` für Ausdrücke, `Dec` für Deklarationen, ...
  - ▶ <https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH-Syntax.html>
  - ▶ Beispiel: Repräsentation von `("foo", x)`:

```
r = TupE [Just (LitE (StringL "foo")), Just (VarE (mkName "x"))]
```

## Template Haskell II

- ▶ Reifikation: vom Code zur Repräsentation (Quotation via "Oxford brackets")

```
r2 = runQ [| ("foo", x) |]
```

- ▶ Splicing: von der Repräsentation zum Code

```
$(return r)$
```

- ▶ Q: The "quotation monad"
  - ▶ Erlaubt e.g. auch IO-Aktionen einzubetten (`runIO`).
- ▶ Beispiele: generische Selektion aus Tupeln, generisches Currying

## Was ist mit Jason?

- ▶ Wir könnten Serialisierung JSON mit Template Haskell schreiben.
- ▶ Aber: müsste für jeden Datentyp spezifisch sein.
- ▶ Besser: **generisch** für alle Datentypen
  - ▶ Konstante Konstruktoren werden zu Konstanten
  - ▶ Mehrstellige Konstruktoren werden zu Produkten
  - ▶ Namen durch Konstruktornamen gegeben

## Generische Programmierung: Polynomiale Datentypen

- ▶ **Polynomiale** Datentypen sind gegeben durch
  - ▶ Produkte (d.h. konstante oder mehrstellige Konstruktoren)
  - ▶ Summen (mehrere Konstruktoren)
- ▶ Beispiele: Listen, Maybe, Bäume, ...
- ▶ Gegenbeispiel: alles mit Funktionsräumen, Sequenzen, Arrays
- ▶ Polynomiale Datentypen können **generisch** repräsentiert werden: **Generics**

## Generische Programmierung: Generics

- ▶ Generische Repräsentation von Datentypen:  
[https://wiki.haskell.org/GHC.Generics#Representation\\_types](https://wiki.haskell.org/GHC.Generics#Representation_types)
- ▶ Konversion von und nach **Generic**:  
<https://hackage.haskell.org/package/base-4.16.1.0/docs/GHC-Generics.html#g:24>
- ▶ Beispiel:

```
data T = Null | Two T T deriving Generic
t = Two Null (Two Null Null)

from t
```
- ▶ Damit: **generische** Konversion von und nach JSON

## II. Metaprogrammierung in Scala

## Metaprogrammierung in Scala: Scalameta

- ▶ Idee: Typsichere Operationen zur Compilezeit

```
import scala.compiletime.constValue
import scala.compiletime.ops.int.*

type A = 3
type B = 4
type C = A + B
def sevenToFourteen(c: C) = constValue[C * 2]
```

## Inlining

```
inline def assert(inline debug: Boolean, assertion: => Boolean): Unit =
  if debug then
  else if !debug then
  else error("debug mode must be a known value")
```

## Heterogene Listen

- Generische Tupel

```
import compiletime._

type Concat[L <: Tuple, +R <: Tuple] <: Tuple = L match
  case EmptyTuple => R
  case h *: t => h *: Concat[t, R]

def concat[L <: Tuple, R <: Tuple](l: L, r: R): Concat[L,R] =
  runtime.Tuples.concat(l, r).asInstanceOf[Concat[L,R]]

val example: (Int, String, Boolean, Double) = concat((1, "Hallo"), (false, 4.2))
```

- Viele Operationen normaler Listen vorhanden:
- Was ist der parameter für map? => Polymorphe Funktionen

## Polymorphe Funktionen und Match Types

- Match Types werden zur Compilezeit reduziert

```
type F[X] = X match
  case Int => Boolean
  case String => Int
  case Boolean => String
  case Double => Double
```

- Damit können Polymorphe Funktionen implementiert werden:

```
def f[X](x: X): F[X] = x match
  case i: Int => i > 0
  case s: String => s.length
  case b: Boolean => b.toString
  case d: Double => -d
```

```
val example2: (Boolean, Int, String, Double) = example.map(f)
```

## Typklassenableitung

- Typklassen können abgeleitet werden:

```
enum Tree[T] derives Eq, Ordering, Show:
  case Branch(left: Tree[T], right: Tree[T])
  case Leaf(elem: T)
```

- Wird übersetzt zu

```
given [T: Eq] : Eq[Tree[T]] = Eq.derived
given [T: Ordering] : Ordering[Tree] = Ordering.derived
given [T: Show] : Show[Tree] = Show.derived
```

- Was macht Eq.derived?

## Typklassenableitung

- Implementierung der Ableitung:

```
import scala.deriving.Mirror

object Eq:
  def derived[T](using Mirror.Of[T]): Eq[T] = ...
```

- Damit können wir für Alle Algebraischen Datentypen Typklassen ableiten, aber...
- Wie Sieht der Mirror für Tree aus?

## Typklassenableitung: Mirrors

- Mirror für Tree

```
new Mirror.Sum:
  type MirroredType = Tree
  type MirroredElemTypes[T] = (Branch[T], Leaf[T])
  type MirroredMonoType = Tree[_]
  type MirroredLabel = "Tree"
  type MirroredElemLabels = ("Branch", "Leaf")

  def ordinal(x: MirroredMonoType): Int = x match
    case _: Branch[_] => 0
    case _: Leaf[_] => 1
```

## Typklassenableitung: Mirrors

- Mirror für Branch

```
new Mirror.Product:
  type MirroredType = Branch
  type MirroredElemTypes[T] = (Tree[T], Tree[T])
  type MirroredMonoType = Branch[_]
  type MirroredLabel = "Branch"
  type MirroredElemLabels = ("left", "right")

  def fromProduct(p: Product): MirroredMonoType =
    new Branch(...)
```

## Typklassenableitung: Mirrors

- Mirror für Leaf

```
new Mirror.Product:
  type MirroredType = Leaf
  type MirroredElemTypes[T] = Tuple1[T]
  type MirroredMonoType = Leaf[_]
  type MirroredLabel = "Leaf"
  type MirroredElemLabels = Tuple1["elem"]

  def fromProduct(p: Product): MirroredMonoType =
    new Leaf(...)
```

## Json Zweiter Versuch

JSON2.scala

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

## Zusammenfassung

- ▶ **Meta-Programmierung:** Programme, die Programme erzeugen
- ▶ Typsichere Manipulation des AST:
  - ▶ Template Haskell, Scalameta
- ▶ Damit **generische Programmierung**
  - ▶ Generic Haskell, Scala

Reaktive Programmierung  
Vorlesung 7 vom 31.05.2022  
Reaktive Ströme (Observables)

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

### Fahrplan

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

### 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
```

- ▶ (Try[T] =>Unit)=>Unit
- ▶ Umgedreht:  
Unit =>(Unit =>Try[T])
- ▶ () =>( () =>Try[T])
- ▶ ≈ Try[T]

### Try vs Future

- ▶ Try[T]: Blockieren → Try[T]
- ▶ Future[T]: Callback → Try[T] (**Reaktiv**)

### Was ist dual zu Iterable?

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

- ▶ () =>( () =>Try[Option[T]])
- ▶ Umgedreht:  
(Try[Option[T]] =>Unit)=>Unit
- ▶ ( T =>Unit, Throwable =>Unit, ()=>Unit )=>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 then  
        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 do
      if battery > 0 then
        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)
  .map(_ + _).takeUntil(_ > 5)
```

## Observables Intern

DEMO

## Observable Contract

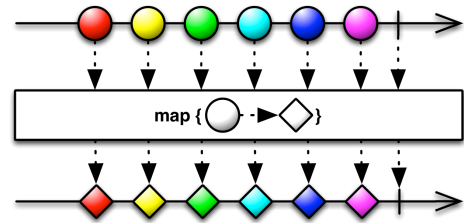
- ▶ die `onNext` Methode eines Observers wird beliebig oft aufgerufen.
- ▶ `onCompleted` oder `onError` werden nur einmal aufgerufen und schließen sich gegenseitig aus.
- ▶ Nachdem `onCompleted` oder `onError` aufgerufen wurde wird `onNext` nicht mehr aufgerufen.

`onNext*(onCompleted|onError)?`

- ▶ Diese Spezifikation wird durch die Konstruktoren erzungen.

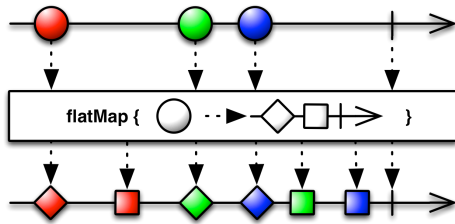
## map

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



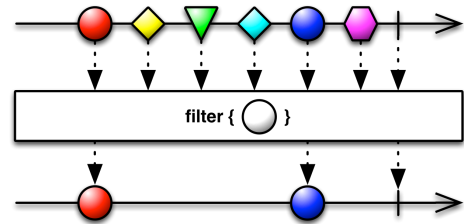
## flatMap

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



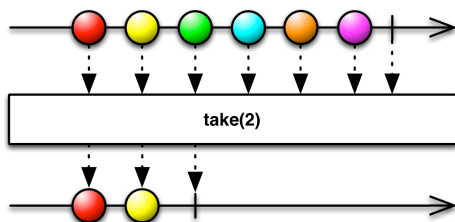
## filter

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



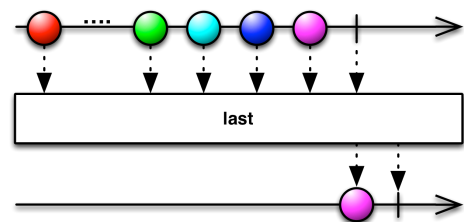
## take

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



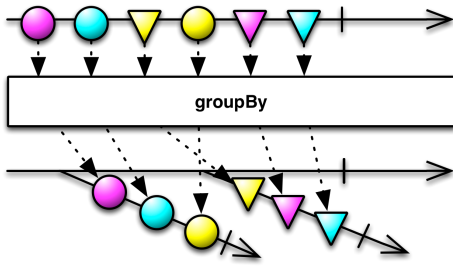
## last

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



## groupBy

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



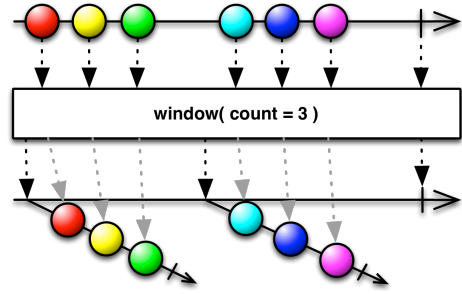
RP SS 2022

17 [26]



## window

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



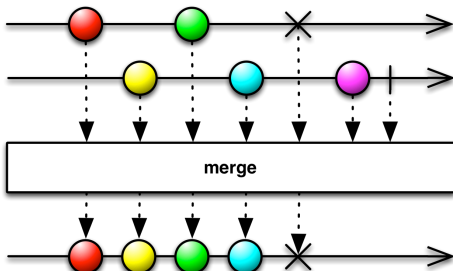
RP SS 2022

18 [26]



## merge

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



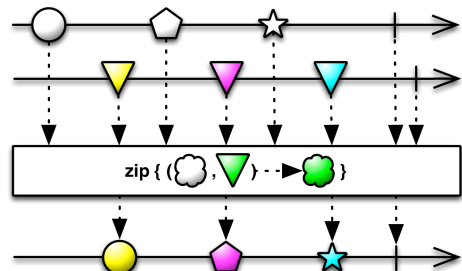
RP SS 2022

19 [26]



## zip

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



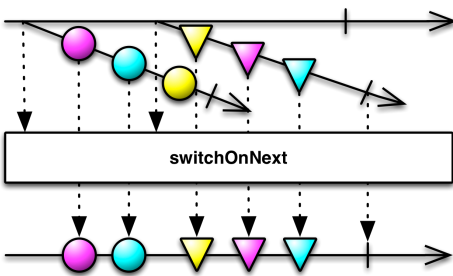
RP SS 2022

20 [26]



## switch

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



RP SS 2022

21 [26]



## Subscriptions

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

```
trait Subscription:
  def cancel(): Unit
```

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

```
trait MultiAssignmentSubscription extends Subscription:
  def subscription_=(s: Subscription)
  def subscription: Subscription
```

RP SS 2022

22 [26]



## Schedulers

- Nebenläufigkeit über Scheduler

```
trait Scheduler:
  def schedule(work: => Unit): Subscription
```

```
trait Observable[T]:
  ...
  def observeOn(scheduler: Scheduler): Observable[T]
```

- Subscription.cancel() muss synchronisiert sein.

RP SS 2022

23 [26]



## 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))

RP SS 2022

24 [26]



## 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 8 vom 07.06.2022  
Reaktive Ströme II

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

Fahrplan

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

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

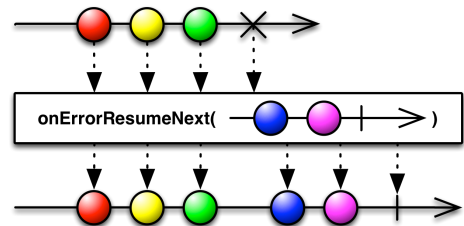
- ▶ Wenn Datenströme Fehler produzieren, können wir diese möglicherweise behandeln.
- ▶ Aber: `Observer.onError` beendet den Strom.

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

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

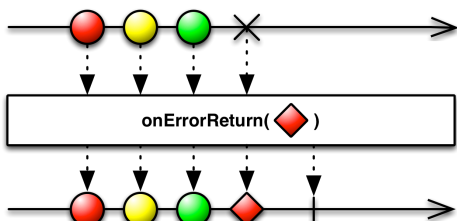
`onErrorResumeNext`

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



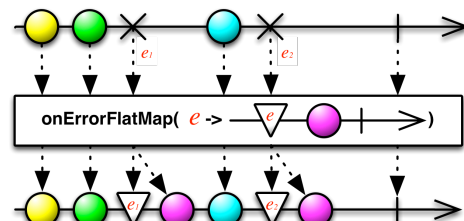
`onErrorReturn`

```
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(scheduler: Scheduler): Observable[T]
```

- ▶ CODE DEMO

## Little's Gesetz

- ▶ In einer stabilen Warteschlange gilt:

$$L = \lambda \times W$$

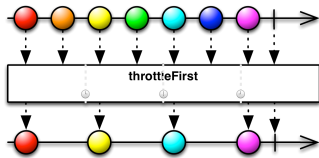
- ▶ Länge der Warteschlange = Ankunftsrate × Durchschnittliche Wartezeit
- ▶ Ankunftsrate =  $\frac{\text{Länge der Warteschlange}}{\text{Durchschnittliche Wartezeit}}$
- ▶ Wenn ein Datenstrom über einen längeren Zeitraum mit einer Frequenz  $> \lambda$  Daten produziert, haben wir ein Problem!

## 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)
```

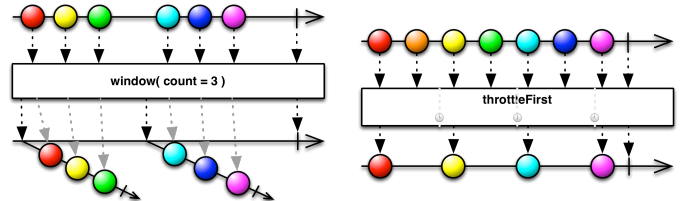


- ▶ Problem: Kurzzeitige Überschreitungen von  $\lambda$  sollen nicht zu Throttling führen.

## Throttling / Debouncing

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

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



- ▶ Was ist wenn wir selbst die Daten Produzieren?

## Back Pressure

- ▶ Wenn wir Kontrolle über die Produktion der Daten haben, ist es unsinnig, sie wegzuerwerfen!
- ▶ 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)
  - ▶ ...

## 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 **Publishers**
- ▶ **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.

RP SS 2022

17 [50]



## 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.

RP SS 2022

18 [50]



## 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 responsivity, 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`.

RP SS 2022

19 [50]



## 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.

RP SS 2022

20 [50]



## Reactive Streams: 2. Subscriber [T]

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

- 11 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.
- 12 `Subscriber.onSubscribe` MUST be called at most once for a given `Subscriber` (based on object equality).
- 13 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.

RP SS 2022

21 [50]



## Reactive Streams: 3. Subscription

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

- 14 `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`.
- 15 The `Subscription` MUST allow the `Subscriber` to call `Subscription.request` synchronously from within `onNext` or `onSubscribe`.
- 16 `Subscription.request` MUST place an upper bound on possible synchronous recursion between `Publisher` and `Subscriber`.
- 17 `Subscription.request` SHOULD respect the responsivity of its caller by returning in a timely manner.
- 18 `Subscription.cancel` MUST respect the responsivity of its caller by returning in a timely manner. MUST be idempotent and MUST be thread-safe.
- 19 After the `Subscription` is cancelled, additional `Subscription.request(long n)` MUST be NOPs.

RP SS 2022

22 [50]



## Reactive Streams: 3. Subscription

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

- 20 After the `Subscription` is cancelled, additional `Subscription.cancel()` MUST be NOPs.
- 21 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.
- 22 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.
- 23 While the `Subscription` is not cancelled, `Subscription.request(long n)` MAY synchronously call `onNext` on this (or other) subscriber(s).
- 24 While the `Subscription` is not cancelled, `Subscription.request(long n)` MAY synchronously call `onComplete` or `onError` on this (or other) subscriber(s).

RP SS 2022

23 [50]



## Reactive Streams: 3. Subscription

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

- 25 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.
- 26 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.
- 27 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.

RP SS 2022

24 [50]



## Reactive Streams: 3. Subscription

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

- Calling `Subscription.cancel` MUST return normally. The only legal way to signal failure to a Subscriber is via the `onError` method.
- Calling `Subscription.request` MUST return normally. The only legal way to signal failure to a Subscriber is via the `onError` method.
- 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".

RP SS 2022

25 [50]



## Reactive Streams: 4. Processor [I,0]

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

- A `Processor` represents a processing stage — which is both a `Subscriber` and a `Publisher` and MUST obey the contracts of both.
- 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.

RP SS 2022

26 [50]



## Akka Streams

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

RP SS 2022

27 [50]



## 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

RP SS 2022

28 [50]



## 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
```

RP SS 2022

29 [50]



## 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
}
```

RP SS 2022

30 [50]



## 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]` – Vereint  $n$  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

RP SS 2022

31 [50]



## 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

RP SS 2022

32 [50]



## 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.

- ▶ Dannach: 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))

- ▶ Fehlerbehandlung über Flow interface:

```
val source = Source(0 to 5).map(100 / _).recover {
  case _: ArithmeticException => 0
}
val result = source.runWith(Sink.fold(0)(_ + _))
```

- ▶ result =Future(Success(228))

## Integration mit Aktoren?

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

- ▶ Deprecated. Jetzt: GraphStage

```
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[MyActorSubscriber])
```

## 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

RP SS 2022

41 [50]



## 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)
```

RP SS 2022

42 [50]



## 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)
```

RP SS 2022

43 [50]



## 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
  - ▶ ...

RP SS 2022

44 [50]



## 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

RP SS 2022

45 [50]



## 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.

RP SS 2022

46 [50]



## WebSockets in akka-http

- ▶ WebSockets ist ein `Flow[Message,Message,Unit]`
- ▶ Können über bidirektionale 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  
    })
```

RP SS 2022

47 [50]



## 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  $\Rightarrow$  akka streams

RP SS 2022

48 [50]



## 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 9 vom 14.06.2022  
Funktional-Reaktive Programmierung

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

### Fahrplan

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

### Das Tagemenü

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

### Dialektik des Programmierens

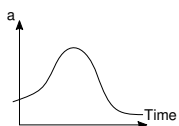
|           | Einer     | Viele         |
|-----------|-----------|---------------|
| Synchron  | Try[T]    | Iterable[T]   |
| Asynchron | Future[T] | Observable[T] |
| Reaktiv   | Event a   | Behaviour a   |

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

### FRP in a Nutshell: Zwei Basiskonzepte

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

```
type Time = Float
type Behaviour α = Time → α
```



- ▶ Beispiel: Position eines Objektes

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

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

```
type Event α = [(Time, α)]
```



- ▶ Beispiel: Benutzereingabe

### Verhalten: erste einfache Beispiele

- ▶ Ein kreisender und ein pulsierender Ball:

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

- ▶ Was passiert hier?

- ▶ Basisverhalten: `time :: Behavior Time, constB :: a → Behavior a`
- ▶ Grafikbücherei: Datentyp `Region`, Funktion `Ellipse`
- ▶ Liftings `(*, 0.5, sin, ...)`

### Lifting

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

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

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

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

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

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

### Reaktive Animationen: Verhaltensänderung

- ▶ Beispiel: auf Knopfdruck Farbe ändern:

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

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

- ▶ Was passiert hier?

- ▶ `untilB` und `switch` kombinieren Verhalten

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

- ▶ Event ist ein Functor:

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

- ▶ Kombination von Ereignissen:

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

## Der Springende Ball

```
ball2 = paint red (translate (x,y) (e11 0.2 0.2)) where
  g = -4
  x = -3 + integral 0.5
  y = 1.5 + integral vy
  vy = integral g 'switch'
      (hity 'snapshot_' vy =>> λv' → lift0 (-v') + integral g)
  hity = when (y <= -1.5)
```

```
ball2x = paint red (translate (x,y) (e11 0.2 0.2)) where
  g = -4
  x = -3 + integral vx
  vx = 1 'switch' (hitx 'snapshot_' vx =>> λv' → lift0 (-v'))
  hitx = when (x <= -3 || x >= 3)
  y = 1.5 + integral vy
  vy = integral g 'switch'
      (hity 'snapshot_' vy =>> λv' → lift0 (-v') + integral g)
  hity = when (y <= -1.5)
```

RP SS 2022 9 [15]

### Nützliche Funktionen:

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

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

## Implementation

### Verhalten 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))
```

RP SS 2022 11 [15]

## 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]
```

RP SS 2022 10 [15]

## Längeres Beispiel: Pong!

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

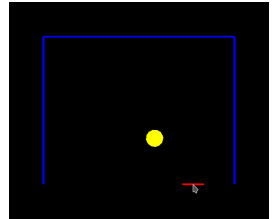
```
paddle = paint red (translate (fst mouse, -1.7) (rec 0.5 0.05))
```

### Die Mauern:

```
walls :: Behavior Picture
```

### ... und alles zusammen:

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



RP SS 2022 12 [15]

## Pong: der Ball

### Der Ball:

```
pball vel =
  let xvel = vel 'stepAccum' xbnc → negate
      xpos = integral xvel
      xbnc = when (xpos >= 2 || xpos <= -2)
      yvel = vel 'stepAccum' ybnc → negate
      ypos = integral yvel
      ybnc = 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) (e11 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
```

RP SS 2022 13 [15]

## 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

RP SS 2022 14 [15]

## 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)

RP SS 2022 15 [15]



Reaktive Programmierung  
Vorlesung 10 vom 21.06.2022  
Software Transactional Memory

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

12:35:49 2022-07-12

1 [37]



## Fahrplan

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

RP SS 2022

2 [37]



## Heute: Software Transactional Memory (STM)

- ▶ Einen fundamental anderen Ansatz nebenläufiger Datenmodifikation
  - ▶ Kein **pessimistischer Ansatz** (wie Locks, conditional variables)
  - ▶ sondern **optimistisch: Transaktionen!**
  - ▶ "Ask forgiveness, not permission"
- ▶ Implementierung in Haskell
- ▶ Fallbeispiele:
  - ▶ Puffer: Reader-/Writer
  - ▶ Speisende Philosophen
  - ▶ Weihnachtlich: das Santa Claus Problem

RP SS 2022

3 [37]



## 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 ()
```

RP SS 2022

4 [37]



## Stand der Technik: Locks und Conditional variables

- ▶ Grundlegende Idee: Zugriff auf gemeinsame Ressourcen nur in **kritischen Abschnitten**
  - 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

RP SS 2022

5 [37]



## 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

RP SS 2022

6 [37]



## 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?

RP SS 2022

7 [37]



## 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

RP SS 2022

8 [37]



## STM: software transactional memory

### Grundidee: Drei Eigenschaften

- 1 Transaktionen sind **atomar**
  - 2 Transaktionen sind **bedingt**
  - 3 Transaktionen sind **komponierbar**
- ▶ Transaktionen werden entweder **ganz** oder (bei Konflikten) **gar nicht** ausgeführt.
  - ▶ Eigenschaften entsprechen Operationen:
    - ▶ Atomare Transaktion
    - ▶ Bedingte Transaktion
    - ▶ Komposition von Transaktionen
  - ▶ Typsystem stellt sicher, dass Transaktionen reversibel sind.

RP SS 2022

9 [37]



## Transaktionen sind atomar

- ▶ 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 }
```

RP SS 2022

10 [37]



## Blockieren und 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

RP SS 2022

11 [37]



## 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?)
  - ▶ Auf Änderung an in Transaktion **gelesenen** Variablen!
  - ▶ Genial: System verantwortlich für Verwaltung der Aufwackbedingung
- ▶ Keine lost wakeups, keine händische Verwaltung von conditional variables

RP SS 2022

12 [37]



## 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

RP SS 2022

13 [37]



## 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)

RP SS 2022

14 [37]



## 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!

RP SS 2022

15 [37]



## Gedankenmodell für atomare Speicheränderungen

### Mögliche Implementierung

- ▶ Thread  $T_1$  im `atomically`-Block nimmt keine Speicheränderungen vor, sondern speichert Schreiboperationen in **Transaktions-Log**
- ▶ Leseoperationen konsultieren zunächst Log
- ▶ Beim Verlassen des `atomically`-Blocks:
  1. **globales Lock** greifen
  2. konsistenter Speicher gelesen?
  - Ja:** 3. Änderungen einpflegen    **Nein:** 3. Änderungen verwerfen
  4. Lock freigeben                    4. Lock freigeben, Block wiederholen

### Konsistenter Speicher

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

RP SS 2022

16 [37]



## 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])
```

RP SS 2022

17 [37]



## 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
```

RP SS 2022

18 [37]



## Puffer mit STM: Anwendungsbeispiel

```
useBuffer :: IO ()
useBuffer = do
  b ← atomically $ new
  forkIO $ forever $ do
    n ← randomRIO(1,5)
    threadDelay (n*10^6)
    t ← getCurrentTime
    mapM_ (\x → atomically $ put b $ show x) (replicate n t)
  forever $ do x ← atomically $ get b
              putStrLn $ x
```

RP SS 2022

19 [37]



## Anwendungsbeispiel Philosophers.hs

- ▶ Gesetzlich vorgeschrieben als Beispiel für nebenläufige Programmierung
- ▶ 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)

RP SS 2022

20 [37]



## 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
```

RP SS 2022

21 [37]



## 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

RP SS 2022

22 [37]



## 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:

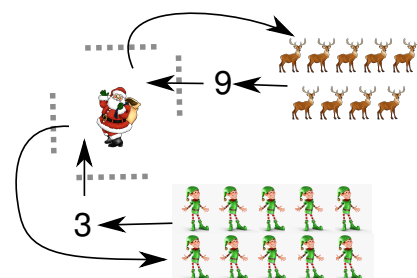
J. A. Trono, *A new exercise in concurrency*, SIGCSE Bulletin, 26:8–10, 1994.

RP SS 2022

23 [37]



## Santa Claus Problem, veranschaulicht



RP SS 2022

24 [37]



## Lösungsstrategie

- ▶ Modellieren jede Elfe, jedes Rentier, und den Weihnachtsmann als **Thread**
  - ▶ Santa wartet und koordiniert, sobald genügend "Teilnehmer" vorhanden
  - ▶ Elfen und Rentiere tun fortwährend dasselbe: Sammeln, arbeiten, herumstehen
- ▶ Verwenden **Gruppen** (`Group`) als Sammelpplätze 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)

RP SS 2022

25 [37]



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

- ▶ Puffer wichtig, da `println` nicht thread-sicher!

```
type Buf = TChan String
out :: Buf -> String -> IO ()
out buf = atomically o writeTChan buf
```

- ▶ Lese-Thread liest Daten aus `Buf` und gibt sie sequentiell an `stdout` aus
- ▶ Aktion der Elfen:

```
meetInStudy :: Buf -> Int -> IO ()
meetInStudy buf id = out buf $ "Elf " ++ show id ++ " meeting in the study"
```

- ▶ Aktion der Rentiere:

```
deliverToys :: Buf -> Int -> IO ()
deliverToys buf id = out buf $ "Reindeer " ++ show id ++ " delivering toys"
```

RP SS 2022

26 [37]



## Arbeitsablauf von Elfen und Rentieren

- ▶ Generisch: Tun im Grunde dasselbe, parametrisiert über `task`

```
simpleJob :: Group -> IO () -> IO ()
simpleJob grp task = do
  (inGate, outGate) <- joinGroup grp
  passGate inGate
  task
  passGate outGate
```

```
elf1, reindeer1 :: Buf -> Group -> Int -> IO ()
elf1 buf grp elfId =
  simpleJob grp (meetInStudy buf elfId)
reindeer1 buf grp reinId =
  simpleJob grp (deliverToys buf reinId)
```

RP SS 2022

27 [37]



## Gatter: Erzeugung, Durchgang

- ▶ Gatter haben eine aktuelle und eine Gesamtkapazität

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

- ▶ Anfänglich leere Aktualkapazität (Santa kontrolliert Durchgang)

```
newGate :: Int -> STM Gate
newGate n = do tv <- newTVar 0
  return $ Gate n tv
```

- ▶ Um Gatter zu passieren muss noch Kapazität vorhanden sein

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

RP SS 2022

28 [37]



## 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!

RP SS 2022

29 [37]



## 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)

RP SS 2022

30 [37]



## Gruppen: Erzeugung, Beitritt

- ▶ Gruppen haben Kapazität (total und aktuell) und zwei Gatter

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

- ▶ Am Anfang ist die aktuelle gleich der totalen Kapazität

```
newGroup :: Int -> IO Group
```

- ▶ Um einer Gruppe beizutreten, muss noch Platz vorhanden sein:

```
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)
```

RP SS 2022

31 [37]



## 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)
```

RP SS 2022

32 [37]



## Elfen und Rentiere

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

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

```
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 -> IO ()
santa buf elves deer = do
  (name, (g1, g2)) <- atomically $
    chooseGroup "reindeer" deer 'orElse' chooseGroup "elves" elves
  out buf $ "Ho, ho, my dear " ++ name
  operateGate g1
  operateGate g2
  where chooseGroup :: String -> Group -> STM (String, (Gate, Gate))
        chooseGroup msg grp = do
          gs <- awaitGroup grp
          return (msg, gs)
```

## Hauptprogramm

- ▶ Ausgabepuffer erzeugen und Ausgabe starten, Gruppen erzeugen, Elfen und Rentiere "starten", `santa` ausführen

```
main :: IO ()
main = do
  outbuf <- newTChanIO
  forkIO $ forever $ do s <- atomically $ readTChan outbuf; putStrLn s

  elfGroup <- newGroup 3
  sequence_ [ elf outbuf elfGroup id | id <- [1..10] ]
  deerGroup <- newGroup 9
  sequence_ [ reindeer outbuf deerGroup id | id <- [1..9] ]

  forever (santa outbuf elfGroup deerGroup)
```

## Zusammenfassung

- ▶ Lock-basierte Nebenläufigkeitsansätze skalieren schlecht
- ▶ Software Transactional Memory als Lock-freie Alternative
  - ▶ Atomarität (`atomically`), Blockieren (`retry`), Choice (`orElse`) als Fundamente kompositionaler Nebenläufigkeit
  - ▶ Faszinierend einfache Implementierungen gängiger Nebenläufigkeitsaufgaben
- ▶ Das freut auch den Weihnachtsmann:
  - ▶ Santa Claus Problem in STM Haskell
- ▶ Geht das auch in Scala? **Im Prinzip ja, aber:**
  - ① Typsichere Reversibilität der Transaktionen nicht möglich
  - ② Muss auf Java-Threadmodell aufbauen

## 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.
- Nathan G. Bronson, Hassan Chafi, and Kunle Olukotun. CCSTM: A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days*, 2010. Available at <http://ppl.stanford.edu/papers/scaladays2010bronson.pdf>

Reaktive Programmierung  
Vorlesung 11 vom 28.06.2022  
Eventual Consistency

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

## Fahrplan

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

## Heute

- ▶ Konsistenzeigenschaften
- ▶ Eventual Consistency
- ▶ Operational Transformation
  - ▶ *Das Geheimnis von Google Docs und co.*

## Was ist eigentlich Konsistenz?

- ▶ Konsistenz = **Widerspruchsfreiheit**
- ▶ Logische Konsistenz:
  - ▶ Eine Formelmengemenge  $\Gamma$  ist konsistent im Kalkül wenn:  $\exists A. \neg(\Gamma \vdash A)$
  - ▶ Ein Kalkül ist konsistent wenn  $\exists A. \neg(\emptyset \vdash A)$
- ▶ In einem verteilten (Speicher)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.
- ▶ Beispiel: Lokaler Speicherzugriff
- ▶ In verteilten Systemen **nicht praktikabel**.

## Eventual Consistency

### Eventual Consistency

Wenn **längere Zeit** keine Änderungen oder Fehler stattfinden konvergieren die Daten an jedem Knoten zu einem gemeinsamen Wert.

- ▶ Beispiel: DNS
- ▶ **Informelle** Anforderung:
  - ▶ Abfragen können beliebige Werte zurückgeben bevor die Knoten konvergieren.
  - ▶ Keine Sicherheitseigenschaft!

## Zwischen Strikter und Eventual Consistency

- ▶ Strikte Konsistenz ist zu **stark**, Eventual Consistency häufig zu **schwach**.
- ▶ Dazwischen gibt es ein breites Spektrum an Konsistenzeigenschaften.
- ▶ Wir unterscheiden
  - ▶ Data-Centric Consistency
  - ▶ Client-Centric Consistency

## 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.
- ▶ Nur so schnell wie das schwächste Glied

## Kausale Konsistenz

### Kausale Konsistenz

- ▶ Abschwächung von sequentieller Konsistenz mit selbem Ergebnis
- ▶ Jeder Prozess sieht Operationen die **in Abhängigkeit stehen** in der selben Reihenfolge.

## Client-Centric Consistency

- ▶ Monotonic Read
- ▶ Monotonic Write
- ▶ Read your Writes
- ▶ Writes follow Reads

## Vektor-Uhren

- ▶ Um Operationen zu ordnen benötigen wir Timestamps
  - ▶ Kausalität muss erhalten bleiben
  - ▶ Timestamps müssen eine totale 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.

## Strong Eventual Consistency

- ▶ **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 an alle verteilt 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

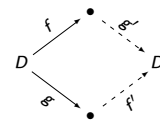
- ▶ Szenario: Webinterface mit Texteditor
- ▶ Mehrere Nutzer können den Text verändern und sollen **immer die neuest mögliche Version** sehen.
- ▶ Siehe Google Docs, Etherpad und co.
- ▶ Der Effekt jeder Operation soll erhalten bleiben
- ▶ Es soll niemals Konflikte geben

## 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*
  - ▶ Nicht konfliktfrei möglich.

## 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 sind **partiell**.

Ein **Beispiel**:

Eingabe: R P P 2  
 Ausgabe: R P 2 2  
 Aktionen: Retain,  
 Delete,  
 Retain,  
 Insert 2,  
 Retain.

## Operationen Komponieren

- ▶ Komposition: Fallunterscheidung auf der **Aktion**

- ▶ Keine einfache Konkatenation!

- ▶ Beispiel:

$p = [\text{Delete}, \text{Insert X}, \text{Retain}]$   
 $q = [\text{Retain}, \text{Insert Y}, \text{Delete}]$   
 $\text{compose } p \ q = [\text{Delete}, \text{Insert X}, \text{Insert Y}, \text{Delete}]$

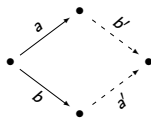
- ▶  $\text{compose}$  ist partiell.

- ▶ **Äquivalenz** von Operationen:

$\text{compose } p \ q \cong [\text{Delete}, \text{Delete}, \text{Insert X}, \text{Insert Y}]$

## Operationen Transformieren

- ▶ Transformation



- ▶ Beispiel:

$a = [\text{Insert X}, \text{Retain}, \text{Delete}]$   
 $b = [\text{Delete}, \text{Retain}, \text{Insert Y}]$   
 $\text{transform } a \ b = ([\text{Insert X}, \text{Delete}, \text{Retain}], [\text{Retain}, \text{Delete}, \text{Insert Y}])$

## Operationen Verteilen

- ▶ Wir haben die Funktion *transform* die zwei nicht-kommutativen Operationen  $a$  und  $b$  zu kommutierenden Gegenstücken  $a'$  und  $b'$  transformiert.

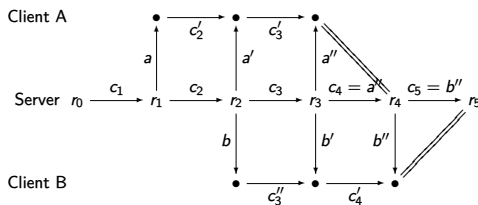
- ▶ Was machen wir jetzt damit?

- ▶ Kontrollalgorithmus nötig

## Der Server

- ▶ Zweck:

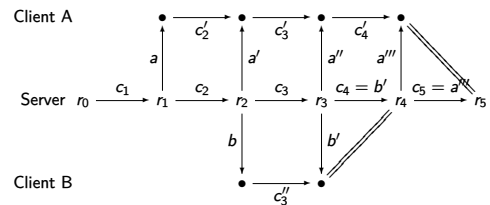
- ▶ Nebenläufige Operationen sequenzialisieren
- ▶ Transformierte Operationen verteilen



## Der Server

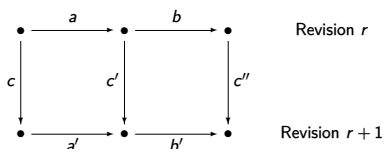
- ▶ Zweck:

- ▶ Nebenläufige Operationen sequenzialisieren
- ▶ Transformierte Operationen verteilen



## Der Client

- ▶ Zweck: Operationen Puffern während eine Bestätigung aussteht



## Zusammenfassung

- ▶ Strikte Konsistenz in verteilten Systemen nicht erreichbar
- ▶ Eventual Consistency
  - ▶ Wenn **längere Zeit** keine Änderungen stattgefunden haben befinden sich schließlich alle Knoten im **gleichen Zustand**.
- ▶ Strong Eventual Consistency
  - ▶ Wenn zwei Knoten die **gleiche Menge** Updates beobachten befinden sie sich im **gleichen Zustand**.
- ▶ Operational Transformation
  - ▶ Strong Eventual Consistency auch ohne kommutative Operationen
- ▶ Nächste Woche: Kommutative Replizierte Datentypen (CRDTs)



Reaktive Programmierung  
Vorlesung 12 vom 05.07.2022  
Konfliktfreie Replizierte Datentypen

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

## Fahrplan

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

## Rückblick: Was ist eigentlich Konsistenz?

- ▶ Konsistenz = **Widerspruchsfreiheit**
- ▶ In der Logik:
  - ▶ Eine Formelmengende  $\Gamma$  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 eingeecheckt 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).

## Partielle Ordnungen

- ▶ Was bedeutet, dass ein Effekt erhalten bleibt?
- ▶ Unsere Daten müssen **größer** werden
- ▶ Semantik definierbar

### Partielle Ordnung

Eine Partielle Ordnung ist eine Relation die **reflexiv**, **anti-symmetrisch** und **transitiv** ist.

## Conflict-Free Replicated Data Types

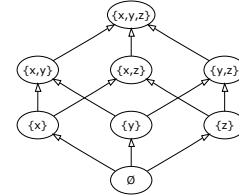
- ▶ Konfliktfreie replizierte Datentypen
- ▶ Garantieren
  - ▶ Strong Eventual Consistency
  - ▶ Monotonie
  - ▶ Konfliktfreiheit
- ▶ Zwei Klassen:
  - ▶ Zustandsbasierte CRDTs
  - ▶ Operationsbasierte CRDTs

## 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

## Merge

- ▶ Die Merge-Funktion ist **kommutativ**, **assoziativ** und **idempotent**
- ▶ Die Merge-Funktion bildet einen Verband (Lattice)



## CvRDT: Mengen

- ▶ Ein einfacher CRDT:
  - ▶ Zustand:  $P \in \mathcal{P}(A)$ , Datentyp:  $\mathcal{P}(A)$
  - $query(P) = P$
  - $update(P, +, a) = P \cup \{a\}$
  - $merge(P_1, P_2) = P_1 \cup P_2$
- ▶ Die Menge kann nur wachsen.

## CvRDT: Zwei-Phasen-Mengen

- ▶ Durch Komposition kann 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:  $\mathcal{P}(A)$
  - $query((P, N)) = query(P) \setminus query(N)$
  - $update((P, N), +, m) = (update(P, +, m), N)$
  - $update((P, N), -, m) = (P, update(N, +, m))$
  - $merge((P_1, N_1), (P_2, N_2)) = (merge(P_1, P_2), merge(N_1, N_2))$

## CvRDT: Zähler

- ▶ Ein weiterer (vermeintlich) einfacher CvRDT
- ▶ Zustand:  $P \in \mathbb{N}$ , Datentyp:  $\mathbb{N}$ 
  - $query(P) = P$
  - $update(P, +, m) = P + m$
  - $merge(P_1, P_2) = \max(P_1, P_2)$
- ▶ Wert kann nur größer werden.
- ▶ Aber: Semantik eines Zählers leider nicht eingehalten

## CvRDT: Verteilter Zähler

- ▶ Jeder Knoten hat seinen Eigenen Zähler
- ▶ Zustand:  $DP \in (id \times \mathbb{N})$ , Datentyp:  $\mathbb{N}$ 
  - $query(DP) = \sum_{(id, P)} query(P)$
  - $update(DP, +, m) = DP \cup update(DP_{id}, +, m)$
  - $merge(DP^1, DP^2) = \{(id, merge(DP_{id}^1, DP_{id}^2)) \mid (id, \_) \in DP^1 \cup DP^2\}$
- ▶ Effekt bleibt erhalten!

## 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:  $\mathbb{Z}$   
 $query((P, N)) = query(P) - query(N)$   
 $update((P, N), +, m) = (update(P, +, m), N)$   
 $update((P, N), -, m) = (P, update(N, +, m))$   
 $merge((P_1, N_1), (P_2, N_2)) = (merge(P_1, P_2), merge(N_1, N_2))$

## CmRDT: Last-Writer-Wins-Register

- ▶ Gegeben eine eindeutigen und total geordneten Timestamp  $T$ :
- ▶ Zustand:  $(X, T) \in X \times timestamp$
- ▶  $query((X, T)) = X$
- ▶  $update((X, T), write, Y) = (Y, T_{now})$
- ▶  $merge((X_1, T_1), (X_2, T_2)) = if T_1 > T_2 then (X_1, T_{now}) else (X_2, T_{now})$

## Vektor-Uhren

- ▶ Im LWW Register benötigen wir Timestamps
  - ▶ Kausalität muss erhalten bleiben
  - ▶ Timestamps müssen eine 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.
- ▶ Die Ordnung ist aber partiell!

## Operationsbasierte CRDTs

- ▶ Kommutative replizierte Datentypen (CmRDTs)
- ▶ Knoten senden nur **Operationen** an andere Knoten
- ▶  $update$  unterscheidete 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:  $(x, t) \in X \times timestamp$
- ▶  $query((x, t)) = x$
- ▶  $update(=, x')$ 
  - ▶ lokal:  $(x, t) := (x', now())$
  - ▶ extern:  $if t < t' then (x, t) := (x', t')$

## 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

Reaktive Programmierung  
Vorlesung 14 vom 12.07.2022  
Theorie der Nebenläufigkeit

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

Fahrplan

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

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
  - ▶ Eigenschaften und ihre Prüfung (**model checking**)

Temporale Logik, Prozessalgebren und Modelchecking

- ▶ Temporale (und modale) Logik beschreiben **Systeme** anhand ihrer **Zustandsübergänge**
- ▶ Ein System ist dabei im wesentlichen eine **endliche Zustandsmaschine**.

Finite State Machine (FSM)

Eine endliche Zustandsmaschine  $\mathcal{M} = \langle S, \Sigma, \rightarrow \rangle$  ist gegeben durch

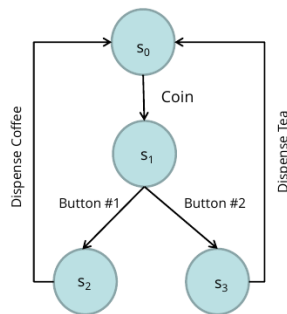
- ▶ eine Menge  $\Sigma$  von **Zuständen**,
- ▶ eine Menge  $I \subseteq \Sigma$  von **initialen** Zuständen, und
- ▶ eine **Zustandsübergangsrelation**  $\rightarrow \subseteq \Sigma \times \Sigma$

so dass

$$\forall s \in \Sigma \exists s' \in \Sigma. s \rightarrow s'$$

Einfache Beispiele

- 1 Münze einwerfen
- 2 Knopf drücken: Tee oder Kaffee
- 3 Tee oder Kaffee ausschenken
- 4 Zurück zu (1)



Model Checking

Das Model-Checking Problem

Gegeben ein Modell  $\mathcal{M}$  und eine Eigenschaft  $\phi$ , gilt

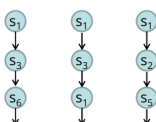
$$\mathcal{M} \models \phi?$$

- ▶ System  $\mathcal{M}$  wird als FSM modelliert
- ▶ Wie beschreiben wir  $\phi$ ?
  - ▶ Temporale Logik
- ▶ Wie beweisen wir das?
  - ▶ Indem wir die Zustände aufzählen und das Modell **prüfen**

Temporal-Logiken

Linear Time

- ▶ Jeder Zeitpunkt hat **genaue einen** Nachfolger
- ▶ Systemzustand: Menge von Zustandssequenzen



Branching Time

- ▶ Jeder Zeitpunkt hat **mehrere** Nachfolger
- ▶ Systemzustand: unendlicher **Baum**



Formeln der LTL

$$\phi ::= p \mid \neg\phi \mid \underbrace{\phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2}_{\text{Aussagenlogik}} \mid X\phi \mid \Box\phi \mid \Diamond\phi \mid \phi_1 U \phi_2$$

- ▶  $X\phi$ :  $\phi$  gilt im **nächsten** Zustand
- ▶  $\Box\phi$ :  $\phi$  gilt in **allen** Zuständen
- ▶  $\Diamond\phi$ :  $\phi$  gilt in **mindestens einem** Zustand
- ▶  $\phi U \psi$ :  $\phi$  gilt in allen Zuständen, bis  $\psi$  gilt.

## Einfache Aussagen

- ▶ Irgendwann gibt es Kaffee:

$\diamond \text{Kaffee}$

- ▶ Wenn ich Knopf-1 drücke, gibt es danach Kaffee:

$\square (\text{Knopf-1} \longrightarrow X \text{Kaffee})$

- ▶ Wenn ich eine Münze einwerfe, gibt es irgendwann entweder Kaffee oder Tee:

$\square (\text{Coin} \longrightarrow (\diamond \text{Kaffee} \vee (\diamond \text{Tee})))$

## Entscheidbarkeit und Zustandsexplosion

### Entscheidbarkeit

- ▶ LTL ohne U ist NP-vollständig;
  - ▶ LTL ist PSPACE-vollständig;
  - ▶ CTL ist EXPTIME-vollständig.
- ▶ Schlüssel zur **praktischen** Handhabbarkeit: **Zustandsabstraktion**
- ▶ Werkzeuge: Spin, nuSMV/nuXMV, UPPAAL, ...
- ▶ Erweiterungen: **hybrid state machines** — Zustände sind **kontinuierlich** und werden durch **Differentialgleichungen** beschrieben.