Reaktive Programmierung
Vorlesung 1 vom 14.04.15: Was ist Reaktive Programmierung?

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2014

Organisatorisches

- Vorlesung: Donnerstags 8-10, MZH 1450
- Übung: Dienstags 16-18, MZH 1460 (nach Bedarf)
- Webseite: `www.informatik.uni-bremen.de/~cxl/lehre/rp.ss15`
- Scheinkriterien:
  - Voraussichtlich 6 Übungsblätter
  - Alle bearbeitet, insgesamt 40% (Notenspiegel PI3)
  - Übungsgruppen 2 – 4 Mitglieder
  - Fachgespräch am Ende

Warum Reaktive Programmierung?

Herkömmliche Programmiersprachen:

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

Eigenschaften:

- Imperativ und prozedural
- Sequentiell

Zugrundeliegendes Paradigma:

Warum Reaktive Programmierung?

Herkömmliche Programmiersprachen:

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

Eigenschaften:

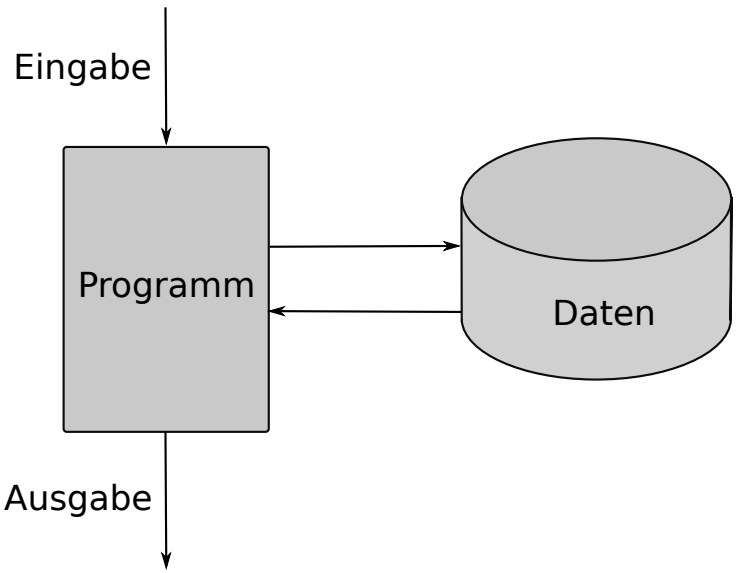
- Imperativ und prozedural
- Sequentiell

Zugrundeliegendes Paradigma:

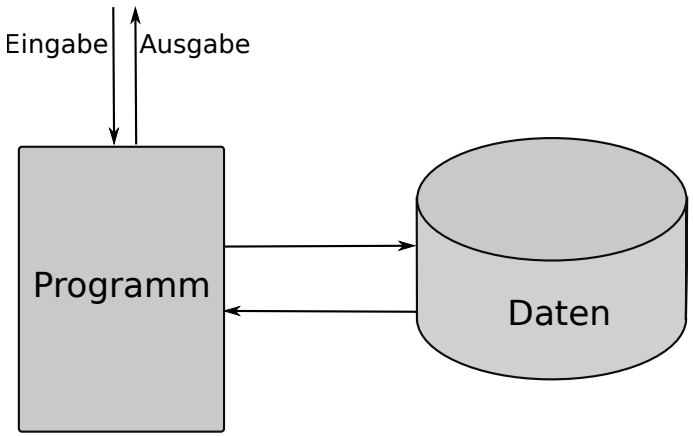
. . . aber die Welt ändert sich:

- Das Netz verbindet Rechner
- Selbst eingebettete Systeme sind vernetzt (Auto: ca. 100 Proz.)
- Mikroprozessoren sind mehrkernig
- Systeme sind eingebettet, nebenläufig, reagieren auf ihre Umwelt.

Probleme mit dem herkömmlichen Ansatz

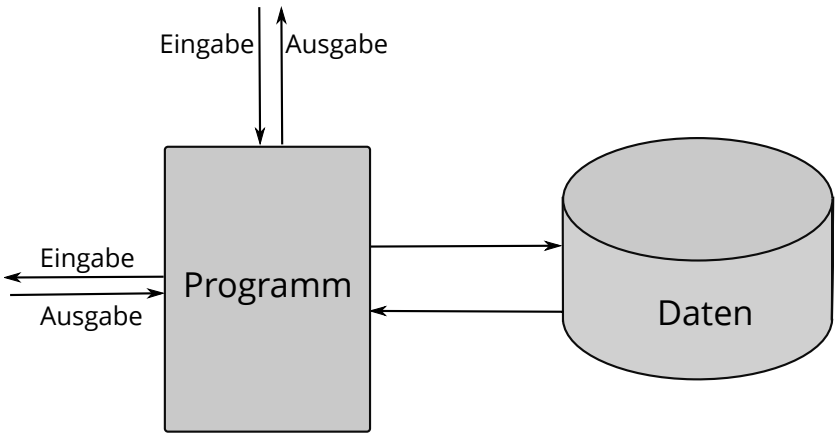


Probleme mit dem herkömmlichen Ansatz



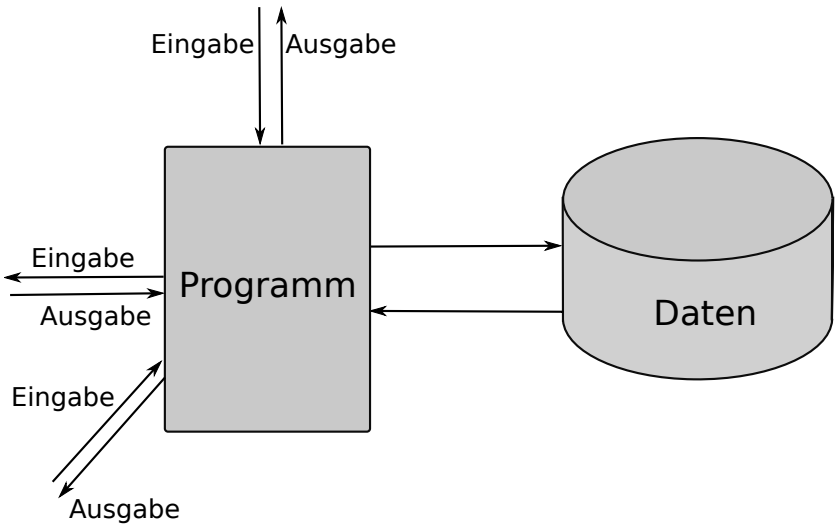
- Problem: Nebenläufigkeit

Probleme mit dem herkömmlichen Ansatz



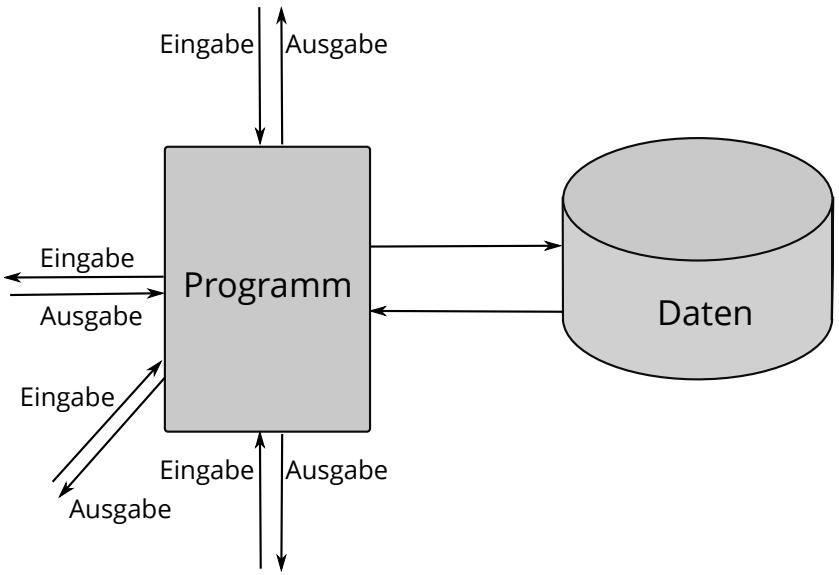
- Problem: Nebenläufigkeit

Probleme mit dem herkömmlichen Ansatz



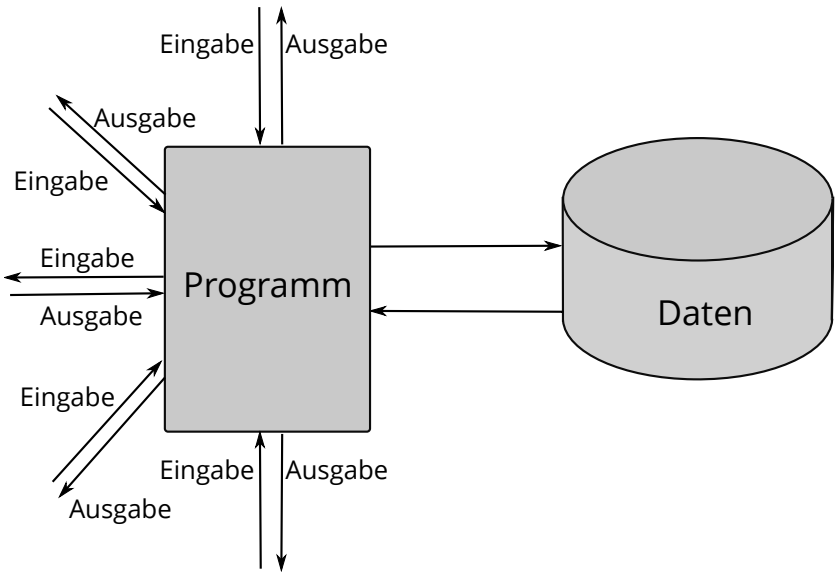
- Problem: Nebenläufigkeit
- Nebenläufigkeit verursacht Synchronisationsprobleme

Probleme mit dem herkömmlichen Ansatz



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

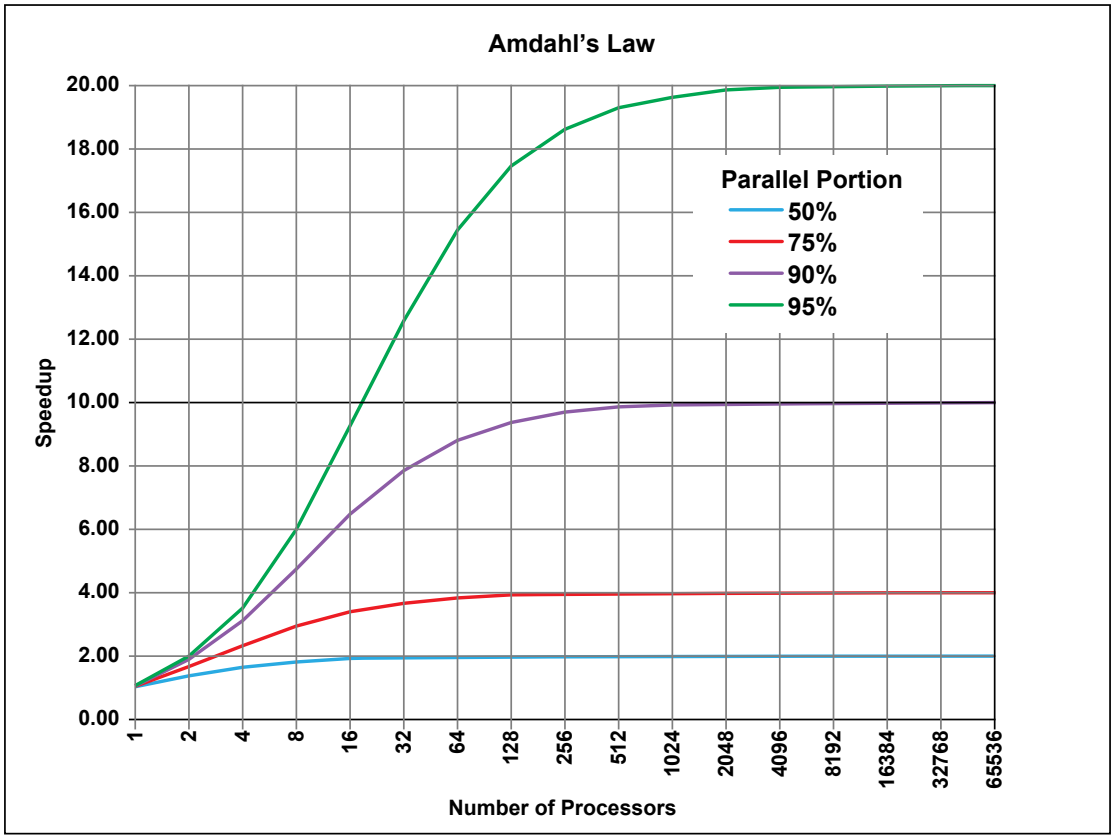
Probleme mit dem herkömmlichen Ansatz



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

Amdahl's Law

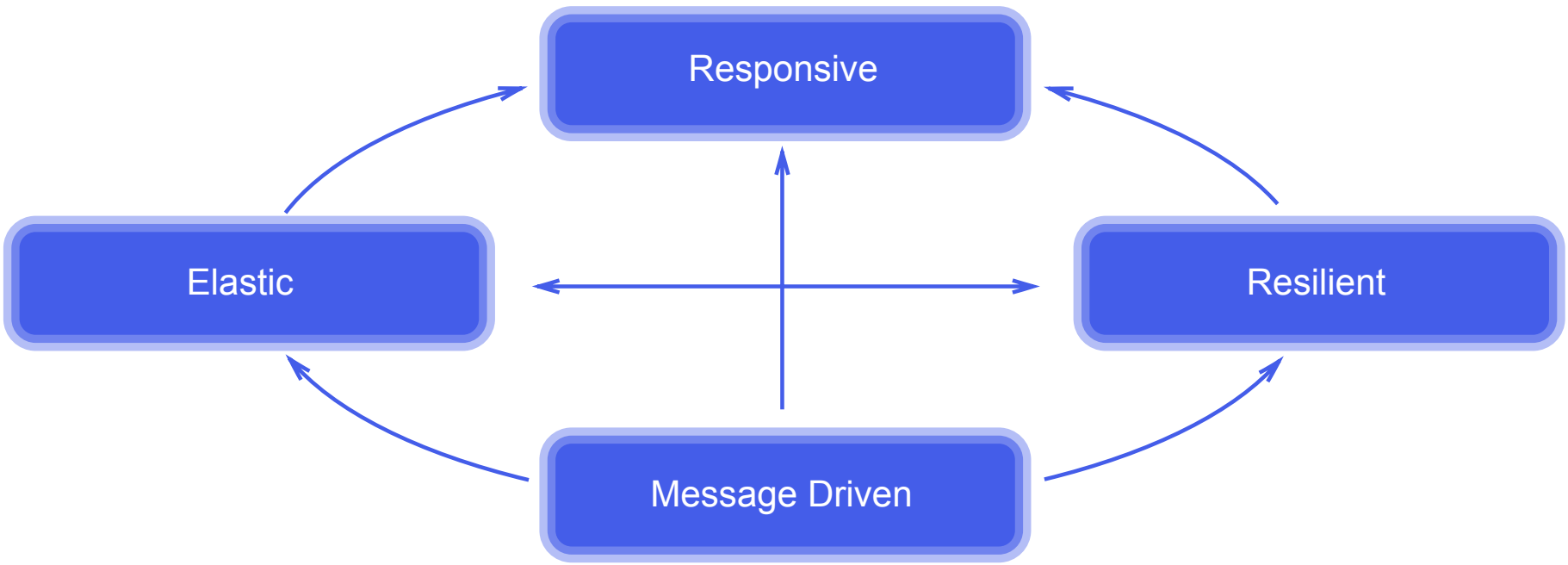
"The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors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



Quelle: Wikipedia

The Reactive Manifesto

- `http://www.reactivemanifesto.org/`



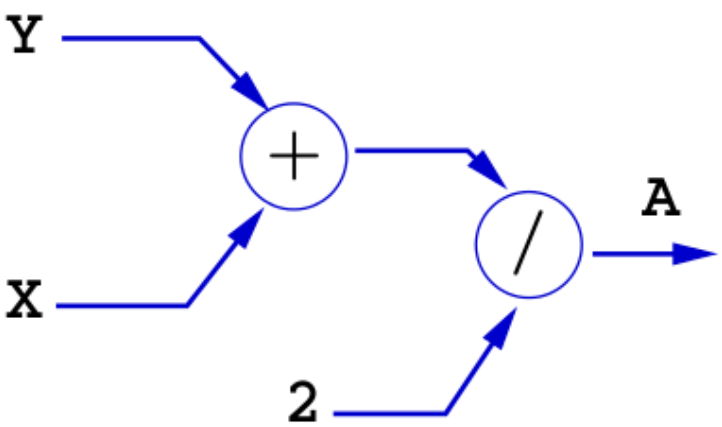
Was ist Reaktive Programmierung?

- Imperative Programmierung: Zustandsübergang
- Prozedural und OO: Verkapselter Zustand
- Funktionale Programmierung: Abbildung (mathematische Funktion)
- Reaktive Programmierung:

1. Datenabhängigkeit
2. Reaktiv = funktional + nebenläufig

Datenflusssprachen (data flow languages)

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




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

Fahrplan

- Teil I: Grundlegende Konzepte
  - Was ist Reaktive Programmierung?
  - Nebenläufigkeit und Monaden in Haskell
  - Funktional-Reaktive Programmierung
  - Einführung in Scala
  - Die Scala Collections
  - ScalaCheck
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
  - Futures and Promises
  - Reaktive Datenströme I
  - Reaktive Datenströme II
  - Das Aktorenmodell
  - Aktoren und Akka
- Teil III: Fortgeschrittene Konzepte

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte
  - Bidirektionale Programmierung: Zippers and Lenses
  - Robustheit, Entwurfsmuster
  - Theorie der Nebenläufigkeit

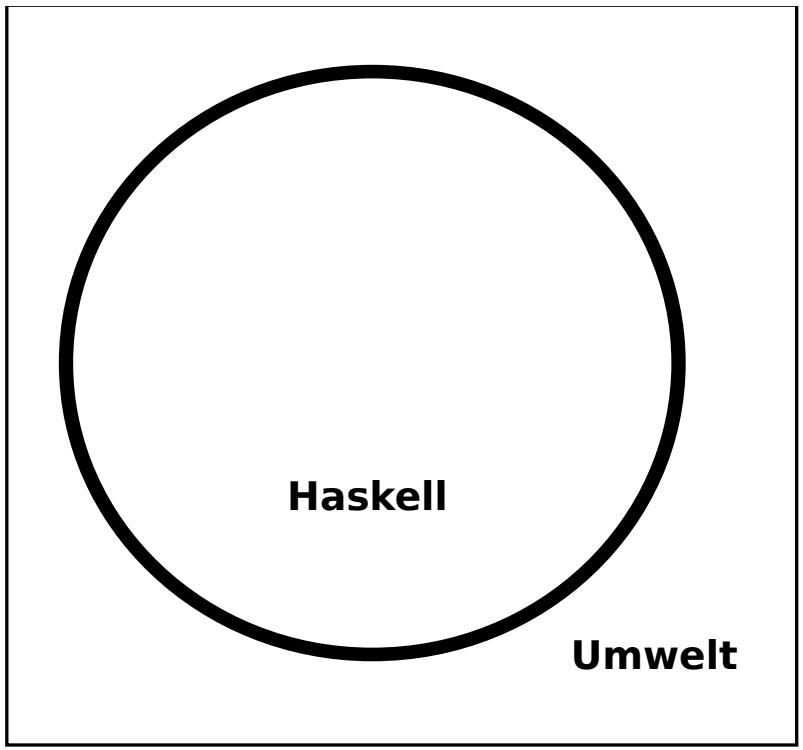
Rückblick Haskell

- 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: `[a]`, `(a, b)`
  - Algebraische Datentypen: `data Maybe a = Just a | Nothing`

Rückblick Haskell

- Abstrakte Datentypen
- Module
- Typklassen
- Verzögerte Auswertung und unendliche Datentypen

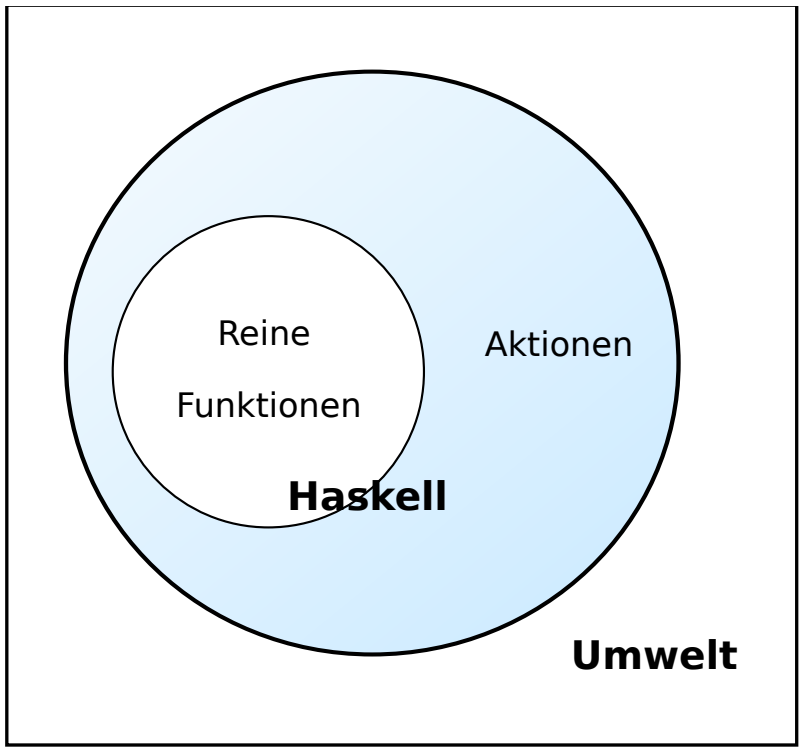
Ein- und Ausgabe in Haskell



Problem:

- Funktionen mit Seiteneffekten nicht referentiell transparent.
- readString :: ... $\rightarrow$String ??

Ein- und Ausgabe in Haskell



Problem:

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

Lösung:

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

Aktionen als abstrakter Datentyp

- ADT mit Operationen Komposition und Lifting
- Signatur:

```
type IO 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 ()
```

Einfache Beispiele

- Echo einfach

```
echo1 :: IO ()
echo1 = getLine >>= putStrLn
```

- Echo mehrfach

```
echo :: IO ()
echo = getLine >>= putStrLn >>= λ_ → echo
```

- Was passiert hier?
  - Verknüpfen von Aktionen mit $\gg=$
  - Jede Aktion gibt Wert zurück

Noch ein Beispiel

- Umgekehrtes Echo:

```
ohce :: IO ()
ohce = getLine
       >>= λs-> putStrLn (reverse s)
       >> ohce
```

- Was passiert hier?
  - Reine Funktion reverse wird innerhalb von Aktion putStrLn genutzt
  - Folgeaktion ohce benötigt Wert der vorherigen Aktion nicht
  - Abkürzung: >>

```
p >> q = p >>= λ_ ->  q
```

Die do-Notation

- Syntaktischer Zucker für IO:

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

$\Longleftrightarrow$

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

  - Rechts sind $\gg=$, $\gg$ implizit.

- Es gilt die Abseitsregel.

  - Einrückung der ersten Anweisung nach **do** bestimmt Abseits.

Drittes Beispiel

- Zählendes, endliches Echo

```
echo3 :: Int→ IO ()
echo3 cnt = do
  putStr (show cnt ++ ": ")
  s← getLine
  if s ≠ "" then do
      putStrLn $ show cnt ++ ": "++ s
      echo3 (cnt+ 1)
    else return ()
```

- Was passiert hier?
  - Kombination aus Kontrollstrukturen und Aktionen
  - Aktionen als Werte
  - Geschachtelte **do**-Notation

Module in der Standardbücherei

- Ein/Ausgabe, Fehlerbehandlung (Modul IO)
- Zufallszahlen (Modul Random)
- Kommandozeile, Umgebungsvariablen (Modul System)
- Zugriff auf das Dateisystem (Modul Directory)
- Zeit (Modul Time)

Ein/Ausgabe mit Dateien

- Im Prelude vordefiniert:
  - Dateien schreiben (überschreiben, anhängen):

```
type FilePath = String
writeFile     :: FilePath → String → IO ()
appendFile    :: FilePath → String → IO ()
```

- Datei lesen (verzögert):

```
readFile      :: FilePath → IO String
```

- Mehr Operationen im Modul IO der Standardbücherei
  - Buffered/Unbuffered, Seeking, &c.
  - Operationen auf Handle

Beispiel: Zeichen, Wörter, Zeilen zählen (wc)

```
wc :: String→ IO ()
wc file =
  do cont ← readFile file
     putStrLn $ file++ ": "++
                show (length (lines cont),
                      length (words cont),
                      length cont)
```

- Datei wird gelesen
- Anzahl Zeichen, Worte, Zeilen gezählt

Aktionen als Werte

- Aktionen sind Werte wie alle anderen.
- Dadurch Definition von Kontrollstrukturen möglich.
- Endlosschleife:

```
forever :: IO α→ IO α
forever a = a ≫ forever a
```

- Iteration (feste Anzahl):

```
forN :: Int→ IO α→ IO ()
forN n a | n == 0     = return ()
         | otherwise = a ≫ forN (n−1) a
```

- Vordefinierte Kontrollstrukturen (Control.Monad):
  - when, mapM, forM, sequence, ...

Map und Filter für Aktionen

- Listen von Aktionen sequenzieren:

```
sequence  :: [IO a] -> IO [a]

sequence_ :: [IO ()] -> IO ()
```

- Map für Aktionen:

```
mapM  :: (a -> IO b) -> [a] -> IO [b]

mapM_ :: (a -> IO ()) -> [a] -> IO ()
```

- Filter für Aktionen
  - Importieren mit `import Monad (filterM)`.

```
filterM :: (a -> IO Bool) -> [a] -> IO [a]
```

Fehlerbehandlung

- Fehler werden durch Exception repräsentiert
  - Exception ist Typklasse — kann durch eigene Instanzen erweitert werden
  - Vordefinierte Instanzen: u.a. IOError
- Fehlerbehandlung durch Ausnahmen (ähnlich Java)

```
catch :: Exception e⇒ IO α → (e→ IO α) → IO α
try   :: Exception e⇒ IO α → IO (Either e a)
```

- Faustregel: catch für unerwartete Ausnahmen, try für erwartete
- Fehlerbehandlung nur in Aktionen

Fehler fangen und behandeln

- Fehlerbehandlung für wc:

```
wc2 :: String→ IO ()
wc2 file =
  catch (wc file)
        (λe → putStrLn $ "Fehler:␣"++ show (e :: IOException))
```

- IOError kann analysiert werden (siehe System.IO.Error)

- read mit Ausnahme bei Fehler (statt Programmabbruch):

```
readIO :: Read a⇒ String→ IO a
```

Ausführbare Programme

- Eigenständiges Programm ist Aktion
- Hauptaktion: main :: IO () in Modul Main
- wc als eigenständiges Programm:

```
module Main where

import System.Environment (getArgs)
import Control.Exception
```

...

```
main :: IO ()
main = do
  args ← getArgs
  mapM_ wc2 args
```

So ein Zufall!

- Zufallswerte:

```
randomRIO :: (α, α)→ IO α
```

  - Warum ist randomIO Aktion?

So ein Zufall!

- Zufallswerte:

```
randomRIO :: (α, α) → IO α
```

  - Warum ist randomIO Aktion?

- Beispiele:
  - Aktion zufällig oft ausführen:

```
atmost :: Int → IO α → IO [α]
atmost most a =
  do l ← randomRIO (1, most)
     sequence (replicate l a)
```

  - Zufälligen String erzeugen:

```
randomStr :: IO String
randomStr = atmost 40 (randomRIO ('a','z'))
```

Funktionen mit Zustand

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

$$f : A \times S \to B \times S$$
$$\cong$$
$$f : A \to S \to B \times S$$

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

In Haskell: Zustände explizit

- Datentyp: Berechnung mit Seiteneffekt in Typ σ (polymorph über α)

```
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
```

- Lifting:

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

Beispiel: Ein Zähler

- Datentyp:

```
type WithCounter α = State Int α
```

- Zähler erhöhen:

```
tick :: WithCounter ()
tick i = ((), i+1)
```

- Zähler auslesen:

```
read :: WithCounter Int
read i = (i, i)
```

- Zähler zurücksetzen:

```
reset :: WithCounter ()
reset i = ((), 0)
```

Implizite vs. explizite Zustände

- Nachteil: Zustand ist explizit
  - Kann dupliziert werden
- Daher: Zustand implizit machen
  - Datentyp verkapseln
  - Signatur State, comp, lift , elementare Operationen
- Beispiel für eine Monade
  - Generische Datenstruktur, die Verkettung von Berechnungen erlaubt

Aktionen als Zustandstransformationen

- Idee: Aktionen sind Transformationen auf Systemzustand S
- S beinhaltet
  - Speicher als Abbildung $A \rightharpoonup V$ (Adressen A, Werte V)
  - Zustand des Dateisystems
  - Zustand des Zufallsgenerators
- In Haskell: Typ RealWorld

```
type IO α = State RealWorld α  -- ... oder so ähnlich
```

  - "Virtueller" Typ, Zugriff nur über elementare Operationen
  - Entscheidend nur Reihenfolge der Aktionen

War das jetzt reaktiv?

- Haskell ist funktional
- Für eine reaktive Sprache fehlt Nebenläufigkeit
  - Nächste Vorlesung: Concurrent Haskell
  - Damit könnten wir die Konzepte dieser VL modellieren
  - Besser: Scala = Funktional + JVM

Zusammenfassung

- Reaktive Programmierung: Beschreibung der Abhängigkeit von Daten
- Rückblick Haskell:
  - Abhängigkeit von Aussenwelt in Typ IO kenntlich
  - Benutzung von IO: vordefinierte Funktionen in der Haskell98 Bücherei
  - Werte vom Typ IO (Aktionen) können kombiniert werden wie alle anderen
- Nächstes Mal:
  - Monaden und Nebenläufigkeit in Haskell

Reaktive Programmierung
Vorlesung 2 vom 16.04.15: Monaden und Nebenlaeufigkeit in Haskell

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
  - Was ist Reaktive Programmierung?
  - Nebenläufigkeit und Monaden in Haskell
  - Funktional-Reaktive Programmierung
  - Einführung in Scala
  - Die Scala Collections
  - ScalaTest und ScalaCheck
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte

Speisekarte

- Das Geheimnis der Monade
- Concurrent Haskell

Zustandsübergangsmonaden

- Aktionen (`IO a`) sind keine schwarze Magie.
- Grundprinzip: Systemzustand Σ wird explizit behandelt.

$$f :: a \to IO\ b \ \cong\ f :: (a, \Sigma) \to (b, \Sigma)$$

  Folgende Invarianten müssen gelten:

  - Systemzustand darf nie dupliziert oder vergessen werden.
  - Auswertungsreihenfolge muss erhalten bleiben.

- Komposition muss Invarianten erhalten
  $\rightsquigarrow$ Zustandsübergangsmonaden

Komposition von Zustandsübergängen

- Im Prinzip Vorwärtskomposition:

```
(>>=) :: ST s a→ (a→ ST s b)→ ST s b
(>>=) :: (s→ (a, s))→ (a→ s→ (b, s))→ (s→ (b, s))
(>>=) :: (s→ (a, s))→ ((a, s)→ (b, s))→ (s→ (b, s))
```

- Damit f >>=g =uncurry g ∘ f
- Aber: ST kann kein Typsynonym sein
- Nötig: abstrakter Datentyp um Invarianten zu erhalten

ST als Abstrakter Datentyp

- Datentyp verkapseln:

```
newtype ST s a = ST (s → (a, s))
```

- Hilfsfunktion (Selektor)

```
unwrap :: ST s a → (s → (a, s))
unwrap (ST f) = f
```

- Damit ergibt sich

```
f >>= g = ST (uncurry (unwrap . g) ∘ unwrap f)
return a = ST (λs → (a, s))
```

Aktionen

- Aktionen: Zustandstransformationen auf der Welt
- Typ RealWorld# repräsentiert Außenwelt
  - Typ hat genau einen Wert realworld#, der nur für initialen Aufruf erzeugt wird.
  - Aktionen: **type** IO a = ST RealWorld# a
- Optimierungen:
  - ST s a durch in-place-update implementieren.
  - IO-Aktionen durch einfachen Aufruf ersetzen.
    - Compiler darf keine Redexe duplizieren!
  - Typ IO stellt lediglich Reihenfolge sicher.

Was ist eigentlich eine Monade?

- `ST` modelliert imperative Konzepte.
- Beobachtung: Andere Konzepte können ähnlich modelliert werden:
  - Ausnahmen: `f :: a-> Maybe b` mit Komposition

```
(>>=) :: Maybe a→ (a→ Maybe b)→ Maybe b
Just a >>= f = f a
Nothing >>= f = Nothing
```

Monads: The Inside Story

```
class Monad m where
    (>>=)  :: m a → (a → m b) → m b
    return :: a → m a
    (>>)   :: m a → m b → m b
    fail   :: String → m a

    p >> q = p >>= λ_ → q
    fail s = error s
```

Folgende Gleichungen müssen (sollten) gelten:

$$\begin{aligned} \text{return } a \gg\!= k &= k\ a \\ m \gg\!= \text{return} &= m \\ m \gg\!= (\lambda x \to k\ x \gg\!= h) &= (m \gg\!= k) \gg\!= h \end{aligned}$$

Beispiel: Speicher und Referenzen

- Signatur:

```
type Mem a
instance Mem Monad
```

- Referenzen sind abstrakt:

```
type Ref
newRef    :: Mem Ref
```

- Speicher liest/schreibt String:

```
readRef   :: Ref → Mem String
writeRef  :: Ref → String→ Mem ()
```

Implementation der Referenzen

Speicher: Liste von Strings, Referenzen: Index in Liste.

```
type Mem = ST [String]    -- Zustand
type Ref = Int

newRef = ST (λs→ (length s, s++[""]))
readRef r = ST (λs→ (s !! r, s))
writeRef r v = ST (λs→ ((),
                  take r s ++ [v]++ drop (r+1) s))
run :: Mem a→ a
run (ST f)= fst (f [])
```

`IORef` — Referenzen

- Datentyp der Standardbücherei (GHC)

```
import Data.IORef

data IORef a

newIORef    :: a → IO (IORef a)
readIORef   :: IORef a → IO a
writeIORef  :: IORef a → a → IO ()

modifyIORef :: IORef a → (a → a) → IO ()
atomicModifyIORef :: IORef a → (a → (a, b)) → IO b
```

- Implementation: “echte” Referenzen.

Beispiel: Referenzen

```
fac :: Int→ IO Int
fac x = do acc ← newIORef 1
           loop acc x where
             loop acc 0 = readIORef acc
             loop acc n = do t ← readIORef acc
                             writeIORef acc (t∗ n)
                             loop acc (n−1)
```

Die Identitätsmonade

- Die allereinfachste Monade:

```
type Id a = a

instance Monad Id where
  return a = a
  b >>= f  = f b
```

Die Listenmonade

- Listen sind Monaden:

```
instance Monad [] where
  m >>= f   = concatMap f m
  return x = [x]
  fail s    = []
```

- Intuition: f :: a → [b] Liste der möglichen Resultate

- Reihenfolge der Möglichkeiten relevant?

Fehlermonaden

- Erste Nährung: Maybe
- Maybe kennt nur Nothing, daher strukturierte Fehler:

```
data Either a b = Left a  | Right b
type Error  a   = Either String a

instance Monad (Either String) where
   (Right a) >>= f = f a
   (Left l)  >>= f = Left l
   return b        = Right b
```

- Nachteil: Fester Fehlertyp
- Lösung: Typklassen

Exkurs: Was *genau* ist eigentliche eine Monade?

- Monade: Konstrukt aus Kategorientheorie
- Monade $\cong$ (verallgemeinerter) Monoid
- Monade: gegeben durch algebraische Theorien
  - Operationen endlicher (beschränkter) Aritität
  - Gleichungen
- Beispiele: Maybe, List, Set, State, ...
- Monaden in Haskell: computational monads
  - Strukturierte Notation für Berechnungsparadigmen
  - Beispiel: Rechner mit Fehler, Nichtdeterminismus, Zustand, ...

Konzepte der Nebenläufigkeit

- Thread (lightweight process) vs. Prozess

Thread (lightweight process)	vs.	Prozess
Programmiersprache/Betriebssystem		Betriebssystem
(z.B. Java, Haskell, Linux)		
gemeinsamer Speicher		getrennter Speicher
Erzeugung billig		Erzeugung teuer
mehrere pro Programm		einer pro Programm

- Multitasking:
  - präemptiv: Kontextwechsel wird erzwungen
  - kooperativ: Kontextwechsel nur freiwillig

Zur Erinnerung: 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 `synchronize`

Threads in Haskell: Concurrent Haskell

- Sequentielles Haskell: Reduktion eines Ausdrucks
  - Auswertung
- Nebenläufiges Haskell: Reduktion eines Ausdrucks an mehreren Stellen
- `ghc` implementiert Haskell-Threads
- Modul `Control.Concurrent` enthält Basisfunktionen
- Wenige Basisprimitive, darauf aufbauend Abstraktionen

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 ()

Rahmenbedingungen

- Zeitscheiben:
  - Tick: Default $20ms$
  - Contextswitch pro Tick bei Heapallokation
  - Änderungen per Kommandozeilenoptionen: `+RTS -V<time> -C<time>`
- Blockierung:
  - Systemaufrufe blockieren alle Threads
    - Mit threaded library (`-threaded`) nicht alle
  - Aber: Haskell Standard-IO blockiert nur den aufrufenden Thread

Concurrent Haskell — erste Schritte

- Ein einfaches Beispiel:

```
write :: Char→ IO ()
write c = putChar c ≫ write c

main :: IO ()
main = forkIO (write 'X') ≫ write 'O'
```

- Ausgabe `ghc`: $(\mathtt{X}^*|\mathtt{O}^*)^*$

Synchronisation mit MVars

- Basissynchronisationmechanismus in Concurrent Haskell
  - Alles andere abgeleitet
- MVar a veränderbare Variable (vgl. IORef a)
- Entweder leer oder gefüllt mit Wert vom Typ a
- 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

Basisfunktionen MVars

- Neue Variable erzeugen (leer oder gefüllt):

```
newEmptyMVar :: IO (MVar a)
newMVar :: a → IO (MVar a)
```

- Lesen:

```
takeMVar :: MVar a → IO a
```

- Schreiben:

```
putMVar :: MVar a → a → IO ()
```

Abgeleitete Funktionen MVars

- Nicht-blockierendes Lesen/Schreiben:

```
tryTakeMVar :: MVar a → IO (Maybe a)
tryPutMVar  :: MVar a→ a→ IO Bool
```

- Änderung der MVar:

```
swapMVar    :: MVar a → a → IO a
withMVar    :: MVar a → (a → IO b) → IO b
modifyMVar  :: MVar a → (a → IO (a, b)) → IO b
```

  - Achtung: race conditions

Ein einfaches Beispiel ohne Synchronisation

- Nebenläufige Eingabe von der Tastatur

```
echo :: String→ IO ()
echo p = forever (do
  putStrLn ("***␣Please␣enter␣line␣for␣"++p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_ n (putStr (p++ ":"++ line++"␣")))

main :: IO ()
main = forkIO (echo "2") ≫ echo "1"
```

- Problem: gleichzeitige Eingabe

Ein einfaches Beispiel ohne Synchronisation

- Nebenläufige Eingabe von der Tastatur

```
echo :: String→ IO ()
echo p = forever (do
  putStrLn ("***␣Please␣enter␣line␣for␣"++p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_ n (putStr (p++ ":"++ line++"␣")))

main :: IO ()
main = forkIO (echo "2") ≫ echo "1"
```

- Problem: gleichzeitige Eingabe
- Lösung: `MVar` synchronisiert Eingabe

Ein einfaches Beispiel mit Synchronisation

- `MVar` voll $\Leftrightarrow$ Eingabe möglich
  - Also: initial voll
- Inhalt der MVar irrelevant: `MVar ()`

```
echo :: MVar ()→ String→ IO ()
echo flag p = forever (do
  takeMVar flag
  putStrLn ("***␣Please␣enter␣line␣"++ p)
  line ← getLine
  n ← randomRIO (1,100)
  replicateM_ n (putStr (p++ ":"++ line++"␣"))
  putMVar flag ())

main :: IO ()
main = do flag ← newMVar ()
          forkIO (echo flag "3") ≫ forkIO (echo flag "2") ≫
echo flag "1"
```

Das Standardbeispiel

- Speisende Philosopen
- Philosoph i:
  - vor dem Essen i-tes und $(i + 1)$ mod n-tes Stäbchen nehmen
  - nach dem Essen wieder zurücklegen
- Stäbchen modelliert als `MVar ()`

Speisende Philosophen

```
philo :: [MVar ()] → Int→ IO ()
philo chopsticks i = forever (do
   let num_phil = length (chopsticks)
   -- Thinking:
   putStrLn ("Phil␣#"++ show i ++"␣thinks.")
   randomRIO (10, 200) >>= threadDelay
   -- Get ready to eat:
   takeMVar (chopsticks !! i)
   takeMVar (chopsticks !! ((i+1) `mod` num_phil))
   -- Eat:
   putStrLn ("Phil␣#"++ show i ++"␣eats.")
   randomRIO (10, 200) >>= threadDelay
   -- Done eating:
   putMVar (chopsticks !! i) ()
   putMVar (chopsticks !! ((i+1) `mod` num_phil)) ())
```

Speisende Philosophen

- Hauptfunktion: n Stäbchen erzeugen
- Anzahl Philosophen in der Kommandozeile

```
main = do
  a:_ ← getArgs
  let num= read a
  chopsticks ← replicateM num (newMVar ())
  mapM_ (forkIO ∘ (philo chopsticks)) [0.. num−1]
  block
```

- Hilfsfunktion `block`: blockiert aufrufenden Thread

```
block :: IO ()
block = newEmptyMVar ≫= takeMVar
```

- NB: Hauptthread terminiert — Programm terminiert!

Zusammenfassung

- Monaden und andere Kuriositäten
  - Zustandsmonade - Referenzen
  - Fehlermonaden
- Concurrent Haskell bietet
  - Threads auf Quellsprachenebene
  - Synchronisierung mit `MVars`
  - Durch schlankes Design einfache Implementierung
- Funktionales Paradigma erlaubt Abstraktionen
  - Beispiel: Semaphoren
- Nächste Woche: Funktional-Reaktive Programmierung.

Reaktive Programmierung
Vorlesung 3 vom 21.04.15: Funktional-Reaktive Programmierung

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
  - Was ist Reaktive Programmierung?
  - Nebenläufigkeit und Monaden in Haskell
  - Funktional-Reaktive Programmierung
  - Einführung in Scala
  - Die Scala Collections
  - ScalaTest und ScalaCheck
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte

Das Tagemenü

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

FRP in a Nutshell

- Zwei Basiskonzepte
- Kontinuierliches, über der Zeit veränderliches Verhalten:

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

- Diskrete Ereignisse zu einem bestimmten Zeitpunkt:

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

- Obige Typdefinitionen sind Spezifikation, nicht Implementation

Verhalten: erste einfache Beispiele

- Ein kreisender und ein pulsierender Ball:

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

- Was passiert hier?
  - Basisverhalten: time :: Behaviour Time, constB :: a → Behavior a
  - Grafikbücherei: Datentyp Region, Funktion Ellipse
  - Liftings (∗, 0.5, sin, ...)

Reaktive Animationen: Verhaltensänderung

- Beispiel: auf Knopfdruck Farbe ändern:

```
color1 :: Behavior Color
color1 = red `untilB` lbp -=> blue
```

- Was passiert hier?
  - untilB kombiniert Verhalten:

```
untilB :: Behavior a -> Event (Behavior a) -> Behavior a
```

  - `=>>` ist map für Ereignisse:

```
(=>>) :: Event a -> (a->b) -> Event b
(-=>) :: Event a-> b-> Event b
e -=> v = e =>> \_ -> v
```

Reaktive Animationen: Verhaltensänderung

- Beispiel: auf Knopfdruck Farbe ändern:

```
color2r = red `untilB` ce where
          ce = (lbp -≫ blue `untilB` ce) .|.
               (key -≫ yellow `untilB` ce)
```

- Was passiert hier?
  - untilB kombiniert Verhalten:

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

  - =≫ ist map für Ereignisse:

```
(=≫) :: Event a → (a→b) → Event b
(-≫) :: Event a→ b→ Event b
e -≫ v = e =≫ λ_ → v
```

  - Kombination von Ereignissen:

Der Springende Ball

```
ball2 = paint red (translate (x,y) (ell 0.2 0.2))
        where g = −4
              x = −3 + integral 0.5
              y = 1.5 + integral v
              v = integral g `switch`
                    (hit `snapshot_` v =≫ λv'→
                     lift0 (−v') + integral g)
              hit = when (y <∗ −1.5)
```

- Nützliche Funktionen:

```
integral :: Behavior Float → Behavior Float
```

```
snapshot :: Event a → Behavior b → Event (a,b)
```

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

Implementation

- Verhalten, erste Annäherung:

```
data Beh1 a = Beh1 ([(UserAction, Time)] → Time → a)
```

- Problem: Speicherleck und Ineffizienz
- Analogie: suche in sortierten Listen

```
inList :: [Int] → Int → Bool
inList xs y = elem y xs

manyInList' :: [Int] → [Int] → [Bool]
manyInList' xs ys = map (inList xs) ys
```

- Besser Sortiertheit direkt nutzen

```
manyInList :: [Int] → [Int] → [Bool]
```

Implementation

- Verhalten werden inkrementell abgetastet:

```
data Beh2 a
    = Beh2 ([(UserAction, Time)] → [Time] → [a])
```

- Verbesserungen:
  - Zeit doppelt, nur einmal
  - Abtastung auch ohne Benutzeraktion
  - Currying

```
data Behavior a
    = Behavior (([Maybe UserAction], [Time]) → [a])
```

- Ereignisse sind im Prinzip optionales Verhalten:

```
data Event a = Event (Behaviour (Maybe a))
```

Längeres Beispiel: Paddleball

- Das Paddel:

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

- Der Ball:

```
pball vel =
 let xvel    = vel `stepAccum` xbounce −≫ negate
     xpos    = integral xvel
     xbounce = when (xpos >* 2 ||* xpos <* −2)
     yvel    = vel `stepAccum` ybounce −≫ negate
     ypos    = integral yvel
     ybounce = when (ypos >* 1.5
                ||* ypos       `between` (−2.0,−1.5) &&*
                    fst mouse `between` (xpos−0.25,xpos+0.25))
 in paint yellow (translate (xpos, ypos) (ell 0.2 0.2))
```

- Die Mauern:

```
walls :: Behavior Picture
```

- ... und alles zusammen:

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

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 nicht für diese Vorlesung
- Generelle Schwäche:
  - Fundamental nicht-kompositional — ist gibt eine Hauptfunktion
  - Fehlerbehandlung, Nebenläufigkeit?

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
- Erlaubt abstrakte Progammierung von reaktiven Animationen
- Problem ist mangelnde Kompositionalität
- Nächste Vorlesungen: Scala!

Reaktive Programmierung
Vorlesung 4 vom 23.04.15: A Practical Introduction to Scala

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
  - Was ist Reaktive Programmierung?
  - Nebenläufigkeit und Monaden in Haskell
  - Funktional-Reaktive Programmierung
  - Einführung in Scala
  - Die Scala Collections
  - ScalaCheck
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte

Heute: Scala

- A scalable language
- Rein objektorientiert
- Funktional
- Eine "JVM-Sprache"
- Seit 2004 von Martin Odersky, EPFL Lausanne (`http://www.scala-lang.org/`).
- Seit 2011 kommerziell durch Typesafe Inc.

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

- Variablen, veränderlich
- Werte, unveränderlich
- `while`-Schleifen
- Rekursion — einfache Endrekursion wird optimiert
- Typinferenz — mehr als Java, weniger als Haskell

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

- Variablen, veränderlich — *Mit Vorsicht benutzen!*
- Werte, unveränderlich
- `while`-Schleifen
- Rekursion — einfache Endrekursion wird optimiert
- Typinferenz — mehr als Java, weniger als Haskell

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

- Variablen, veränderlich — *Mit Vorsicht benutzen!*
- Werte, unveränderlich
- `while`-Schleifen — *Unnötig!*
- Rekursion — einfache Endrekursion wird optimiert
- Typinferenz — mehr als Java, weniger als Haskell

Scala am Beispiel: `02-Rational.scala`

Was sehen wir hier?

Scala am Beispiel: `02-Rational.scala`

Was sehen wir hier?

- Klassenparameter
- `this`
- Methoden, Syntax für Methodenanwendung
- `override` (nicht optional)
- private Werte und Methoden
- Klassenvorbedingunge (`require`)
- Overloading
- Operatoren

Algebraische Datentypen: `03-Expr.scala`

Was sehen wir hier?

Algebraische Datentypen: `03-Expr.scala`

Was sehen wir hier?

- `case class` erzeugt
  - Factory-Methode für Konstruktoren
  - Parameter als implizite `val`
  - abgeleitete Implementierung für `toString`, `equals`
    - strukturelle Gleichheit
  - . . . und pattern matching
- Pattern sind
  - `case 4 =>` — Literale
  - `case C(4) =>` — Konstruktoren
  - `case C(x) =>` — Variablen
  - `case C(_) =>` — Wildcards
  - `case x: C =>` — getypte pattern
  - `case C(D(x: T, y), 4) =>` — geschachtelt

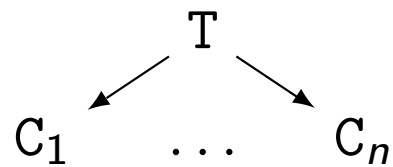
Implementierung algebraischer Datentypen

Haskell:

```
data T = C1 | ... | Cn
```

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



- Varianten als Subtypen
- Problem und Vorteil:

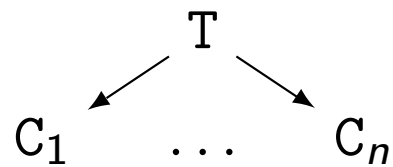
Implementierung algebraischer Datentypen

Haskell:

```
data T = C1 | ... | Cn
```

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



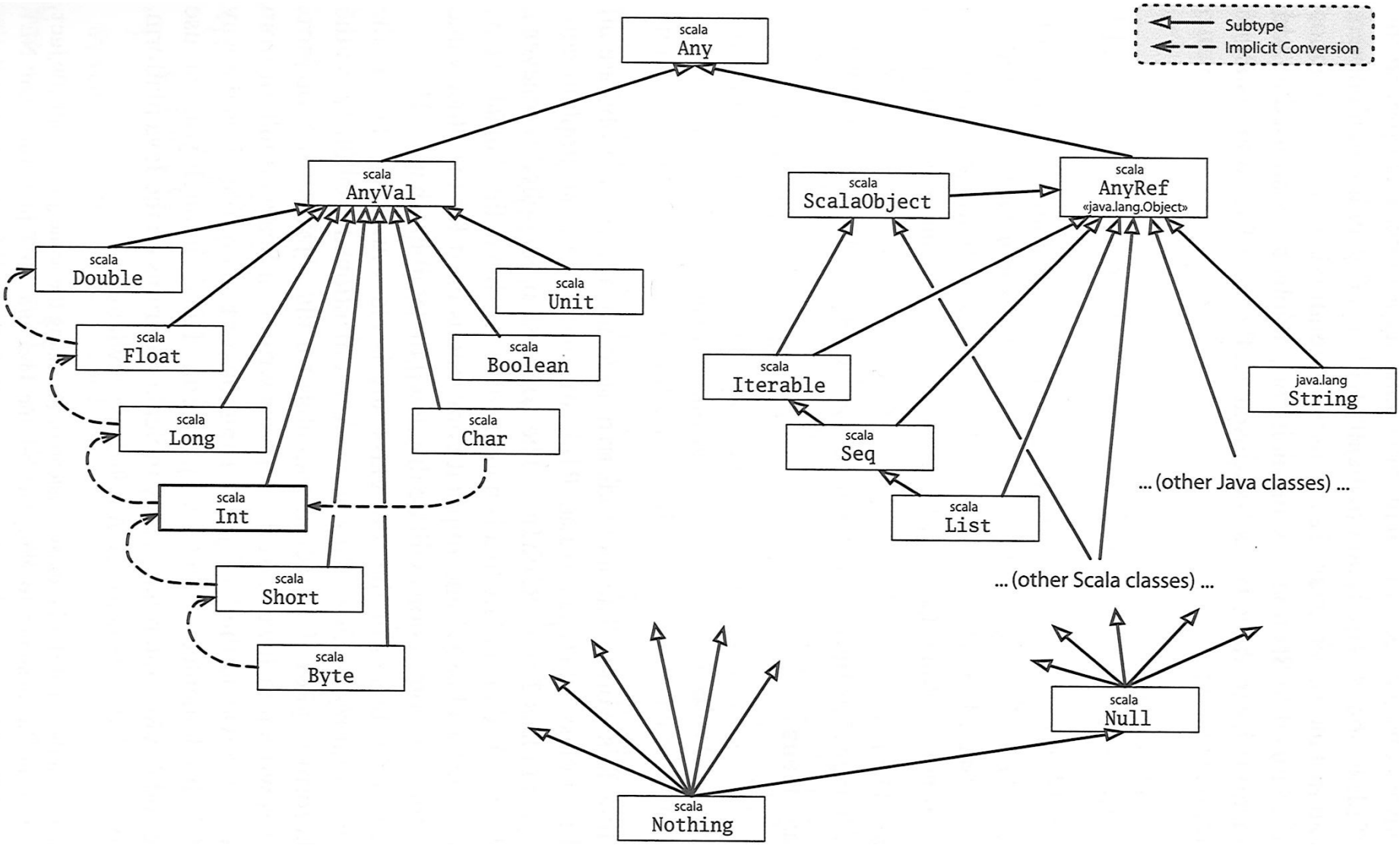
- Varianten als Subtypen
- Problem und Vorteil: Erweiterbarkeit
- `sealed` verhindert Erweiterung

Das Typsystem

Behandelt:

- Werte vs. Objekte
- Scala vs. Java
- `NULL` references

Vererbungshierarchie



Quelle: Odersky, Spoon, Venners: *Programming in Scala*

Parametrische Polymorphie

- Typparameter (wie in Java, Haskell), Bsp. `List[T]`
- Problem: Vererbung und Polymorphie
- Ziel: wenn `S` $<$ `T`, dann `List[S]` $<$ `List[T]`
- Problem: `Ref.hs`
- Warum?
  - Funktionsraum nicht monoton im ersten Argument
  - Sei $X \subseteq Y$, dann $Z \longrightarrow X \subseteq Z \longrightarrow Y$, aber $X \longrightarrow Z \not\subseteq Y \longrightarrow Z$

Typvarianz

`class C[+T]`

- Kovariant
- `S` $<$ `T`, dann `C[S]` $<$ `C[T]`
- Parameter `T` nicht in Def.bereich

`class C[T]`

- Rigide
- Kein Subtyping
- Parameter `T` kann beliebig verwendet werden

`class C[-T]`

- Kontravariant
- `S` $<$ `T`, dann `C[T]` $<$ `C[S]`
- Parameter `T` nicht in Wertebereich

Beispiel:

```
class Function[-S, +T] {
  def apply(x:S) : T
  }
```

Traits: `04-Funny.scala`

Was sehen wir hier?

- `Traits` (Mix-ins): abstrakte Klassen, Interfaces; Haskell: Typklassen
- Unterschied zu Klassen:
  - Keine Parameter
  - Keine feste Oberklasse (`super` dynamisch gebunden)
- Nützlich zur Strukturierung:

$$\textit{thin interface} + \textit{trait} = \textit{rich interface}$$

Beispiel: `04-Ordered.scala`, `04-Rational.scala`

Was sind Traits?

- Trait $\approx$ Abstrakte Klasse ohne Parameter:

```
trait Foo[T] {
  def foo: T
  def bar: String = "Hallo"
}
```

- Erlauben "Mehrfachvererbung":

```
class C extends Foo[Int] with Bar[String] { ... }
```

- Können auch als Mixins verwendet werden:

```
trait Funny {
  def laugh() = println("hahaha")
}

(new C with Funny).laugh() // hahaha
```

Implizite Parameter

- Implizite Parameter:

```
def laugh(implicit stream: PrintStream) =
  stream.println("hahaha")
```

Implizite Parameter

- Implizite Parameter:

```
def laugh(implicit stream: PrintStream) =
  stream.println("hahaha")
```

- Werden im Kontext des Aufrufs aufgelöst. (Durch den Typen)

Implizite Parameter

- Implizite Parameter:

```
def laugh(implicit stream: PrintStream) =
  stream.println("hahaha")
```

- Werden im Kontext des Aufrufs aufgelöst. (Durch den Typen)
- Implizite Parameter + Traits $\approx$ Typklassen:

```
trait Show[T] { def show(value: T): String }

def print[T](value: T)(implicit show: Show[T]) =
  println(show.show(value))

implicit object ShowInt extends Show[Int] {
  def show(value: Int) = value.toString
}

print(7)
```

Implizite Konversionen

- Implizite Konversionen:

```
implicit def stringToInt(string: String) = string.toInt

val x: Int = "3"
x * "5" == 15 // true
"5" % "4" == 1 // true
```

Implizite Konversionen

- Implizite Konversionen:

```
implicit def stringToInt(string: String) = string.toInt

val x: Int = "3"
x * "5" == 15 // true
"5" % "4" == 1 // true
```

- Mit großer Vorsicht zu genießen!
- “Extension Methods” / “Pimp-My-Library” allerdings sehr nützlich!

Implizite Konversionen

- Implizite Konversionen:

```
implicit def stringToInt(string: String) = string.toInt

val x: Int = "3"
x * "5" == 15 // true
"5" % "4" == 1 // true
```

- Mit großer Vorsicht zu genießen!
- “Extension Methods” / “Pimp-My-Library” allerdings sehr nützlich!
- Besser: Implizite Klassen

```
implicit class RichString(s: String) {
  def shuffle = Random.shuffle(s.toList)
                      .mkString
}

"Hallo".shuffle // "laoHl"
```

Scala — Die Sprache

- Objekt-orientiert:
  - Veränderlicher, gekapselter Zustand
  - Subtypen und Vererbung
  - Klassen und Objekte
- Funktional:
  - Unveränderliche Werte
  - Polymorphie
  - Funktionen höherer Ordnung

Beurteilung

- Vorteile:
  - Funktional programmieren, in der Java-Welt leben
  - Gelungene Integration funktionaler und OO-Konzepte
  - Sauberer Sprachentwurf, effiziente Implementierung, reiche Büchereien
- Nachteile:
  - Manchmal etwas zu viel
  - Entwickelt sich ständig weiter
  - One-Compiler-Language, vergleichsweise langsam

Reaktive Programmierung
Vorlesung 5 vom 30.04.15: The Scala Collection Library

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
  - Was ist Reaktive Programmierung?
  - Nebenläufigkeit und Monaden in Haskell
  - Funktional-Reaktive Programmierung
  - Einführung in Scala
  - Die Scala Collections
  - ScalaTest und ScalaCheck
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte

Heute: Scala Collections

- Sind nicht in die Sprache eingebaut!
- Trotzdem komfortabel

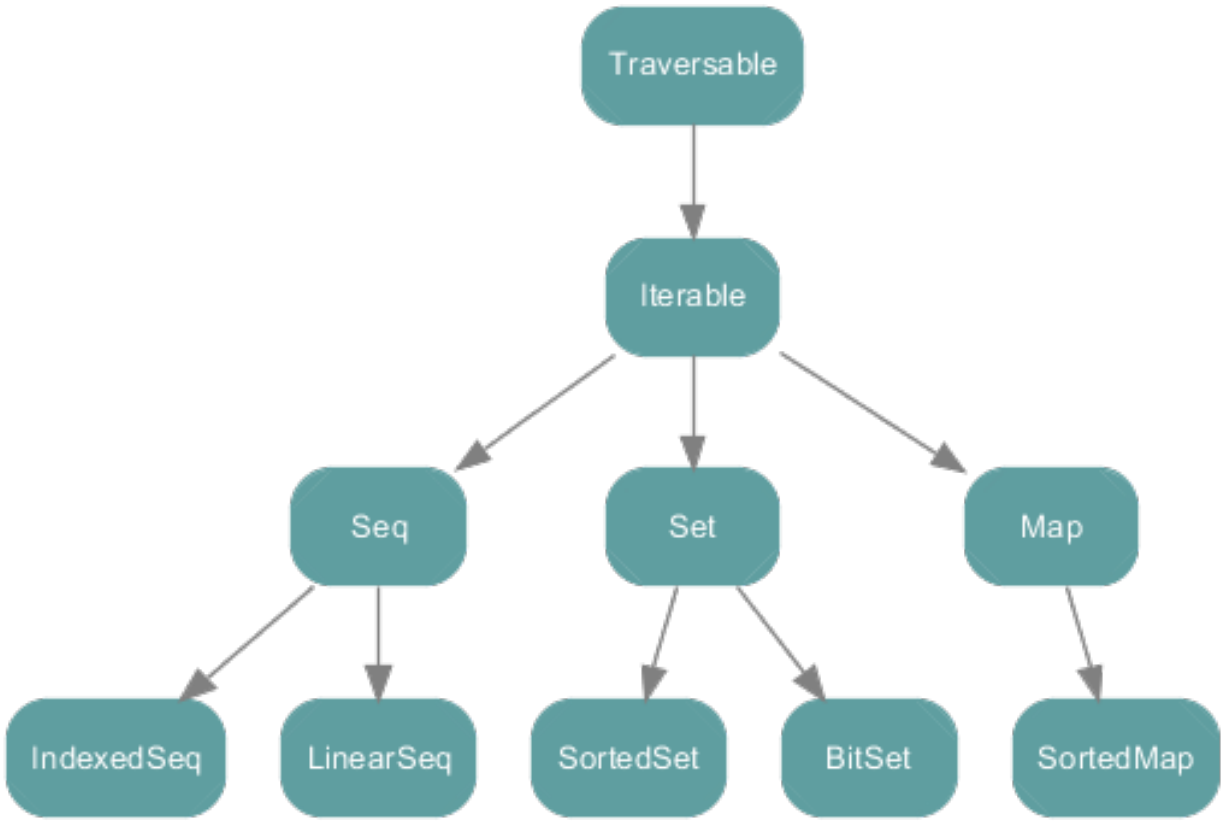
```
val ages = Map("Homer" -> 36, "Marge" -> 34)
ages("Homer") // 36
```

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

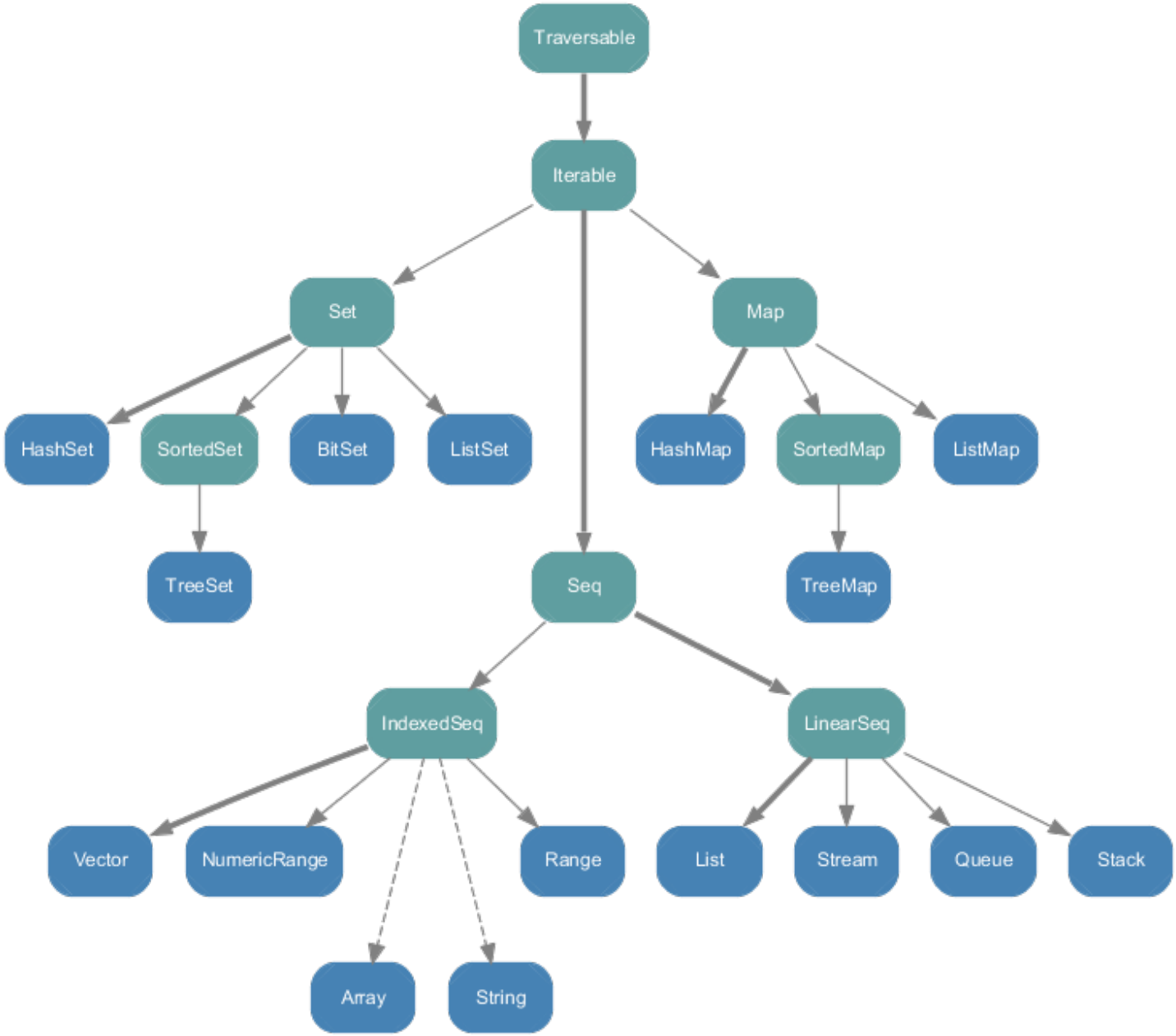
```
val a = Array(1,2,3) ++ List(1,2,3)
a.flatMap(i ⇒ Seq(i,i+1,i+2))
```

Scala Collections Bücherei

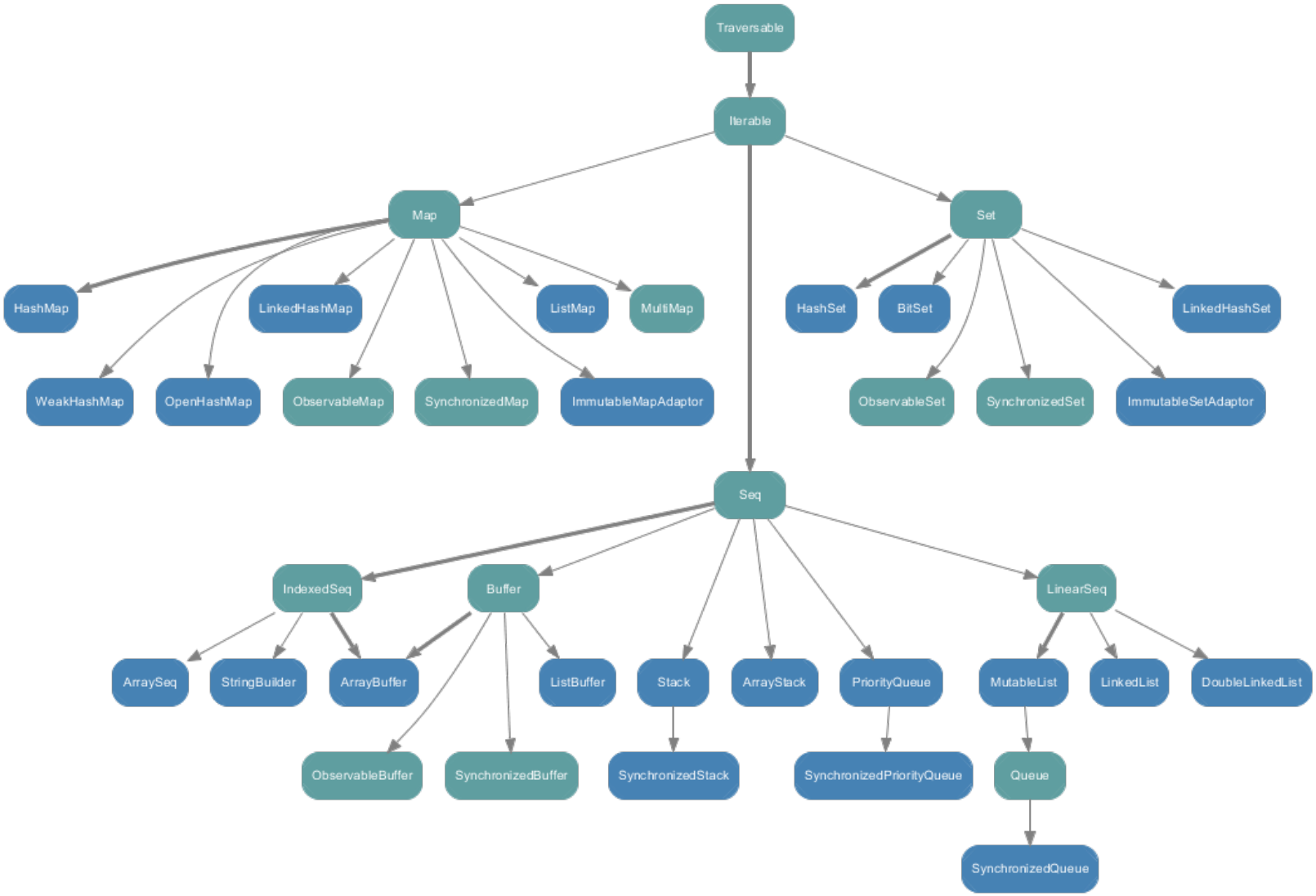
Sehr einheitliche Schnittstellen aber komplexe Bücherei:



Scala Collections Bücherei - Immutable



Scala Collections Bücherei - Mutable



Konstruktoren und Extraktoren

- Einheitliche Konstruktoren:

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

- Einheitliche Extraktoren:

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


...
```

Exkurs: Funktionen in Scala

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

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

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

Exkurs: Konstruktoren in Scala

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

```
trait Person {
  def age: Int
  def name: String
}

object Person {
  def apply(a: Int, n: String) = new Person {
    def age = a
    def name = n
  }
}

val homer = Person(36,"Homer")
```

- Vgl. Case Classes

Exkurs: Extraktoren in Scala

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

```
object Person {
  def apply(a: Int, n: String) = <...>
  def unapply(p: Person): Option[(Int,String)] =
    Some((p.age,p.name))
}

homer match {
  case Person(age, name) if age < 18 ⇒ s"hello young
      $name"
  case Person(_, name) ⇒ s"hello old $name"
}

val Person(a,n) = homer
```

scala.collection.Traversable[+A]

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

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

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

scala.collection.Traversable[+A]

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

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

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

Die wahre Signatur von `map`

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

Die wahre Signatur von `map`

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

Was machen wir damit?

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

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

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

Set[+A]

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

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

Set("apple","strawberry") -- Set("apple","peach")
> Set("strawberry")

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

- Nützliche Subtypen: `SortedSet`, `BitSet`

Map[K,V]

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

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

ages("Homer")
> 39

ages isDefinedAt "Bart" // ages contains "Bart"
> false

ages get "Marge"
> Some(34)
```

- Nützliche Subtypen: `mutable.Map`

Collections Vergleichen

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

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

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

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

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

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

Scala Collections by Example - Part I

- Problem: Namen der erwachsenen Personen in einer Liste

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

Scala Collections by Example - Part I

- Problem: Namen der erwachsenen Personen in einer Liste

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

- Lösung:

```
val adults = persons.filter(_.age >= 18).map(_.name)
```

```
> List("Homer", "Marge", "Abe")
```

Scala Collections by Example - Part II

- Problem: Fibonacci Zahlen so elegant wie in Haskell?

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

Scala Collections by Example - Part II

- Problem: Fibonacci Zahlen so elegant wie in Haskell?

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

- Lösung:

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

fibs.take(10).foreach(println)
> 0
> 1
> ...
> 21
> 34
```

Option[+A]

- Haben maximal 1 Element

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

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

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

- Hilfreich dabei:

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

Option[+A]

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

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

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

- Nützliche Operationen auf `Option`

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

x getOrElse 0

x foldLeft ("Test")(_.toString)
x exists (_ == 4)
...
```

Ranges

- Repräsentieren Zahlensequenzen

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

- `Int` ist "gepimpt" (`RichInt`):

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

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

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

For Comprehensions

- In Scala ist `for` nur syntaktischer Zucker

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

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

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

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

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

Scala Collections by Example - Part III

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

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

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

Scala Collections by Example - Part III

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

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

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

- Lösung:

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

Zusammenfassung

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

Reaktive Programmierung
Vorlesung 6 vom 05.05.15: ScalaTest and ScalaCheck

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Organisatorisches

- Zu diskutieren:
  - Vorlesung ab jetzt Dienstags von 16-18 Uhr,
  - Übung dafür Donnerstags ab 9 Uhr?

Fahrplan

- Teil I: Grundlegende Konzepte
  - Was ist Reaktive Programmierung?
  - Nebenläufigkeit und Monaden in Haskell
  - Funktional-Reaktive Programmierung
  - Einführung in Scala
  - Die Scala Collections
  - ScalaTest und ScalaCheck
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte

Was ist eigentlich Testen?

Myers, 1979

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

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

E. W. Dijkstra, 1972

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

Testmethoden

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

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

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

ScalaTest

- Test Framework für Scala

```
import org.scalatest.FlatSpec

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

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

ScalaTest Assertions 1

- ScalaTest Assertions sind Makros:

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

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

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

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

ScalaTest Assertions 2

- Fehler manuell werfen:

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

- Erwartete Exeptions:

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

- Assumptions

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

ScalaTest Matchers

- Gleichheit überprüfen:

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

- Länge prüfen:

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

- Und so weiter...

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

- Siehe `http://www.scalatest.org/user_guide/using_matchers`

ScalaTest Styles

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

```
class SetSpec extends FunSpec {
  describe("A Set") {
    describe("when empty") {
      it("should have size 0") {
        assert(Set.empty.size == 0)
      }

      it("should produce NoSuchElementException when
          head is invoked") {
        intercept[NoSuchElementException] {
          Set.empty.head
} } } } }
```

- Übersicht unter
  http://www.scalatest.org/user_guide/selecting_a_style

Blackbox Test

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

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

- z.B.

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

- Was ist mit allen anderen Eingaben?

Property based Testing

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

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

- Wir würden gerne so was schreiben:

```
forall x >= 1 -> primeFactors(x).product = x
                 && primeFactors(x).forall(isPrime)
```

Property based Testing

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

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

- Wir würden gerne so was schreiben:

```
forall x >= 1 -> primeFactors(x).product = x
                 && primeFactors(x).forall(isPrime)
```

- Aber wo kommen die Eingaben her?

Testen mit Zufallswerten

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

- Zufallszahlen sind doch einfach!

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

Testen mit Zufallswerten

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

- Zufallszahlen sind doch einfach!

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

- Was ist mit dieser Funktion?

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

ScalaCheck

- ScalaCheck nutzt Generatoren um Testwerte für Properties zu generieren

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

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

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

Zufallsgeneratoren

- Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }

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

Zufallsgeneratoren

- Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }

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

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

Zufallsgeneratoren

- Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }

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

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

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

Zufallsgeneratoren

- Ein generischer Zufallsgenerator:

```
trait Generator[+T] { def generate: T }

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

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

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

- ```
  val pairs =
      Generator((integers.generate,integers.generate))
  ```
  ```

Zufallsgeneratoren Kombinieren

- Ein generischer, kombinierbarer Zufallsgenerator:

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

Einfache Zufallsgeneratoren

- Einelementige Wertemenge:

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

- Eingeschränkter Wertebereich:

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

- Aufzählbare Wertemenge:

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

Beispiel: Listen Generieren

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

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

Beispiel: Listen Generieren

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

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

- Die Menge der leeren Listen enthält genau ein Element:

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

Beispiel: Listen Generieren

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

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

- Die Menge der leeren Listen enthält genau ein Element:

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

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

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

ScalaCheck

- ScalaCheck nutzt Generatoren um Testwerte für Properties zu generieren

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

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

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

Kombinatoren in ScalaCheck

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

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

Wertemenge einschränken

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

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

scala> prop.check
Gave up after only 4 passed tests. 500 tests were
    discarded.
```

- Besser:

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

Kombinatoren für Properties

- Properties können miteinander kombiniert werden:

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

ScalaCheck in ScalaTest

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

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

Zusammenfassung

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

Reaktive Programmierung
Vorlesung 7 vom 12.05.14: Futures and Promises

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
  - Futures and Promises
  - Das Aktorenmodell
  - Aktoren und Akka
  - Reaktive Datenströme I
  - Reaktive Datenströme II
- Teil III: Fortgeschrittene Konzepte

Implizite Fehlerbehandlung mit Ausnahmen

- Die Signatur einer Methode verrät nichts über mögliche Fehler

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Robot = {
    if (n <= 0) this
    else if (battery > 0) {
      Thread.sleep(1000);
      Robot(pos + 1, battery - 1).move(n-1)
    } else throw new LowBatteryException
} }
```

- Problem bei der Kombination:
  - Wir müssen `try` und `catch` benutzen
  - Kombination wird umständlich, und kombiniert mit Seiteneffekten unmöglich
  - Beispiel: `Robot` als veränderliche Klasse

Der Datentyp `Try`

- Macht Fehler explizit (Materialisierung):

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

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

object Try {
  def apply[T](expr: ⇒ T): Try[T] =
  try Success(expr)
  catch { case NonFatal(ex) ⇒ Failure(ex) } }
```

- Ist `Try` eine Monade?

Der Datentyp `Try`

- Macht Fehler explizit (Materialisierung):

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

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

object Try {
  def apply[T](expr: ⇒ T): Try[T] =
  try Success(expr)
  catch { case NonFatal(ex) ⇒ Failure(ex) } }
```

- Ist `Try` eine Monade? Nein, `Try(e) flatMap f != f e`

Explizite Fehlerbehandlung

- `Try` macht Fehler explizit:

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Try[Robot] = Try {
    def mv(r: Robot, n: Int): Robot = {
      if (n <= 0) this
      else if (battery > 0) {
          Thread.sleep(1000);
          mv(Robot(pos+1, battery- 1), n-1)
        } else throw new LowBatteryException }
    mv(this, n)
}   }

for { atCheckpoint ← robot.move(3)
      atGoal ← atCheckpoint.move(2) } yield atGoal
```

Explizite Fehlerbehandlung

- `Try` macht Fehler explizit:

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Try[Robot] = Try {
    def mv(r: Robot, n: Int): Robot = {
      if (n <= 0) this
      else if (battery > 0) {
          Thread.sleep(1000);
          mv(Robot(pos+1, battery- 1), n-1)
        } else throw new LowBatteryException }
    mv(this, n)
}   }


for { atCheckpoint ← robot.move(3)
      atGoal ← atCheckpoint.move(2) } yield atGoal
```

- Aber gibt es hier noch mehr unsichtbare Besonderheiten?

Explizite Fehlerbehandlung

- `Try` macht Fehler explizit:

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Try[Robot] = Try {
    def mv(r: Robot, n: Int): Robot = {
      if (n <= 0) this
      else if (battery > 0) {
          Thread.sleep(1000);
          mv(Robot(pos+1, battery- 1), n-1)
        } else throw new LowBatteryException }
    mv(this, n)
}   }


for { atCheckpoint ← robot.move(3)
      atGoal ← atCheckpoint.move(2) } yield atGoal
```

- Aber gibt es hier noch mehr unsichtbare Besonderheiten?
- Die Methode gibt das Ergebnis `n` Sekunden verzögert zurück!

Blockierende Methoden

- Was ist das Problem an Verzögerungen?

```
import scala.util.Random
val robotSwarm =
    List.fill(6)(Robot(0,Random.nextInt(10)))
val survivors = robotSwarm.map(_.move(5)).collect {
  case Success(survivor) ⇒ survivor }
```

- Wie lange dauert das?

Blockierende Methoden

- Was ist das Problem an Verzögerungen?

```
import scala.util.Random
val robotSwarm =
    List.fill(6)(Robot(0,Random.nextInt(10)))
val survivors = robotSwarm.map(_.move(5)).collect {
  case Success(survivor) ⇒ survivor }
```

- Wie lange dauert das?

- Bis zu 30*s*, weil die Methode `move` blockiert!

Typische Verzögerungen

Verzögerung	Zeit
execute typical instruction	1 ns
fetch from L1 cache memory	0.5 ns
branch misprediction	5 ns
fetch from L2 cache memory	7 ns
Mutex lock/unlock	25 ns
fetch from main memory	100 ns
send 2K bytes over 1Gbps network	20,000 ns
read 1MB sequentially from memory	250,000 ns
fetch from new disk location (seek)	8,000,000 ns
read 1MB sequentially from disk	20,000,000 ns
send packet US to Europe and back	150,000,000 ns

$1ns = 10^{-9}s$

`http://norvig.com/21-days.html#answers`

Nebenläufigkeit in Scala

- Scala hat kein sprachspezifisches Thread-Modell, sondern nutzt das Threadmodell der JVM.
- Daher sind Threads vergleichsweise teuer.
- Synchronisation auf unterster Ebene durch Monitore (`synchronized`)
- Bevorzugtes Abstraktionsmodell: Aktoren (dazu später mehr)

Futures

- Futures machen Fehler und Verzögerungen explizit!

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Future[Robot] = Future {
    def mv(r: Robot, n: Int): Robot = {
      if (n <= 0) this
      else if (battery > 0) {
          Thread.sleep(1000);
          mv(Robot(pos+1, battery- 1), n-1)
        } else throw new LowBatteryException }
    mv(this, n)
}   }
val robotSwarm =
    List.fill(6)(Robot(0,Random.nextInt(10)))
val moved = robotSwarm.map(_.move(5))
```

- Wie lange dauert das?

Futures

- Futures machen Fehler und Verzögerungen explizit!

```
case class Robot(pos: Int, battery: Int) {
  def move(n: Int): Future[Robot] = Future {
    def mv(r: Robot, n: Int): Robot = {
      if (n <= 0) this
      else if (battery > 0) {
          Thread.sleep(1000);
          mv(Robot(pos+1, battery- 1), n-1)
        } else throw new LowBatteryException }
    mv(this, n)
}   }
val robotSwarm =
    List.fill(6)(Robot(0,Random.nextInt(10)))
val moved = robotSwarm.map(_.move(5))
```

- Wie lange dauert das?
- 0 Sekunden! Nach spätestens 5 Sekunden sind alle Futures erfüllt:

Wie funktioniert das?

- Futures haben ein einfaches Interface

```
trait Future[+T] {
  def isCompleted: Boolean
  def onComplete(f: Try[T] ⇒ Unit): Unit
  def value: Option[Try[T]]
  def map[U](f: T ⇒ U): Future[U]
  def flatMap[U](f: T ⇒ Future[U]): Future[U]
  def filter(p: T ⇒ Boolean): Future[T]
}
```

- Und können einfach erzeugt werden

```
object Future {
  def apply[T](f: ⇒ T): Future[T] = ...
}
```

- Siehe `Future.scala`

Promises

- Promises sind das Gegenstück zu Futures

```
trait Promise {
  def complete(result: Try[T])
  def future: Future[T]
}

object Promise {
  def apply[T]: Promise[T] = ...
}
```

- Das Future eines Promises wird durch die `complete` Methode erfüllt.
- Siehe `Promise.scala`

Execution Contexts

- Wir haben etwas verschwiegen:

```
trait Future[T] {
  def onComplete(cb: Try[T] ⇒ Unit)
    (implicit ec: ExecutionContext): Unit
}
```

- Die meisten Methoden auf Futures erwarten implizit einen `ExecutionContext`

```
trait ExecutionContext {
  def execute(runnable: Runnable): Unit
  def reportFailure(cause: Throwable): Unit
  def prepare(): ExecutionContext
}
```

- Darüber kann kontrolliert werden wo der Code ausgeführt wird.

Await und Duration

- Mit Await können Futures in den Klassischen Kontrollfluss eingebunden werden:

```
import scala.language.postfixOps
import scala.concurrent.util.duration._

val answer = for {
  _ ← sendRequest()
  a ← awaitResponse()
} yield a

Await.result(answer, atMost = 1 hour)
```

Blocking vs. Non-blocking IO

- Blockierende Futures verbrauchen einen ganzen Thread.

```
def nextRequest: Future[String] = Future {
    Stream.readLine() }
```

- Threads sind teuer! (Limit typischerweise $<$ 100000 Threads)
- Wenn möglich: nichtblockierende IO

```
def nextRequest: Future[String] = {
  val p = Promise[String]
  Stream.onNextLine(p.success)
  p.future
}
```

Nebenläufigkeit in anderen Sprachen

- Andere funktionale Sprachen (Haskell, Erlang) haben leicht-gewichtige Threads
  - Laufzeitsystem handelt Threads, Erzeugung und Synchronisation billig
- Haskell hat `MVar` (ähnlich `Futures`, aber ohne Callback)
- Erlang hat Aktoren

Zusammenfassung

- 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
- Nächste Vorlesung: Das Aktorenmodell

Reaktive Programmierung
Vorlesung 8 vom 19.05.15: The Actor Model

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Organisatorisches

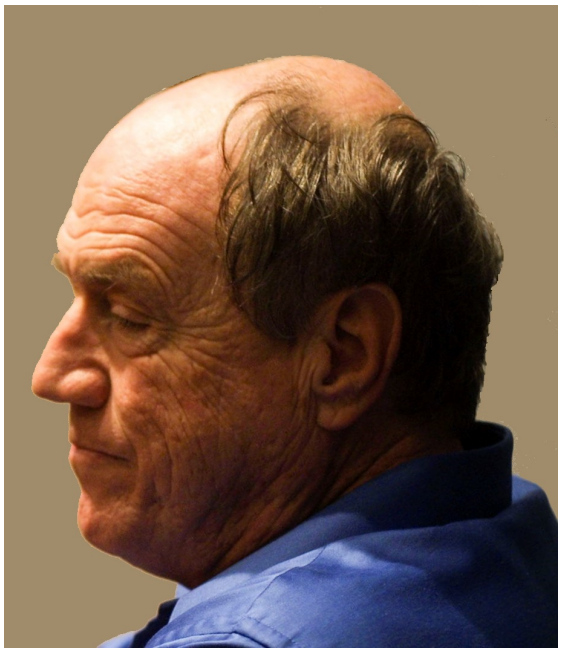
Wir sind umgezogen!

- Christoph: Cartesium 2.046
- Martin: Cartesium 2.051

Fahrplan

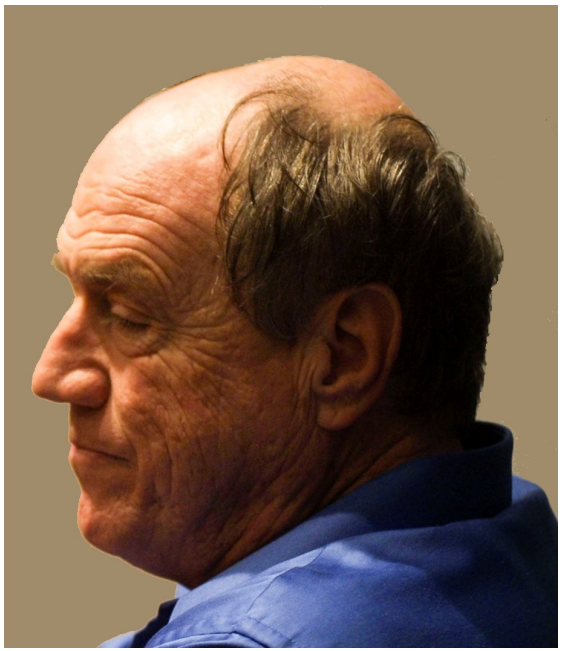
- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
  - Futures and Promises
  - Das Aktorenmodell
  - Aktoren und Akka
  - Reaktive Datenströme - Observables
  - Reaktive Datenströme - Back Pressure und Spezifikation
  - Reaktive Datenströme - Akka Streams
- Teil III: Fortgeschrittene Konzepte

Das Aktorenmodell



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

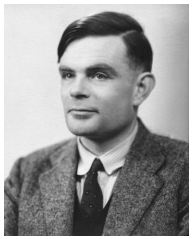
Das Aktorenmodell



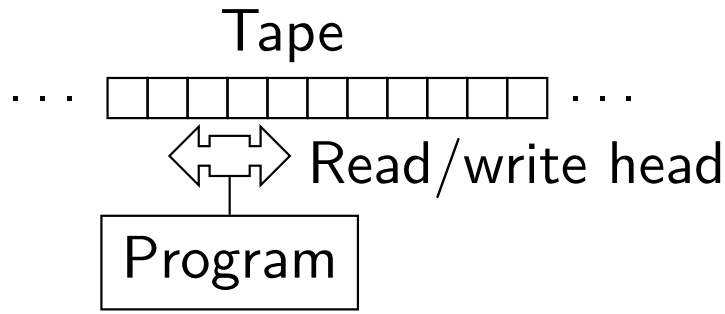
- Eingeführt von Carl Hewitt, Peter Bishop und Richard Steiger (1973)
- Grundlage für nebenläufige Programmiersprachen und Frameworks. (Unter anderem Akka)
- Theoretisches Berechnungsmodell

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

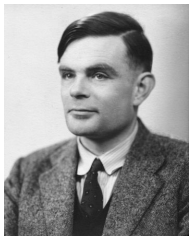
Die Turingmaschine



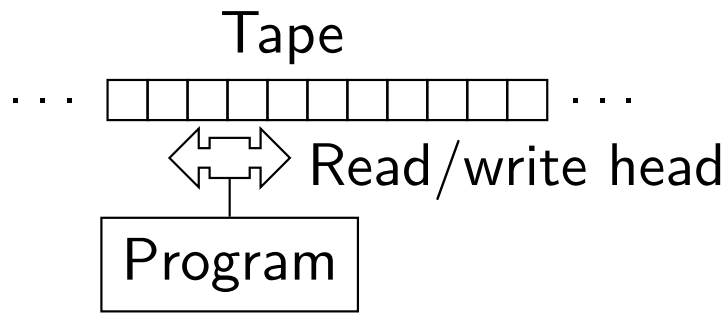
> "the behavior of the computer at any moment is determined by the symbols which he [the computer] is observing, and his 'state of mind' at that moment" — Alan Turing



Die Turingmaschine

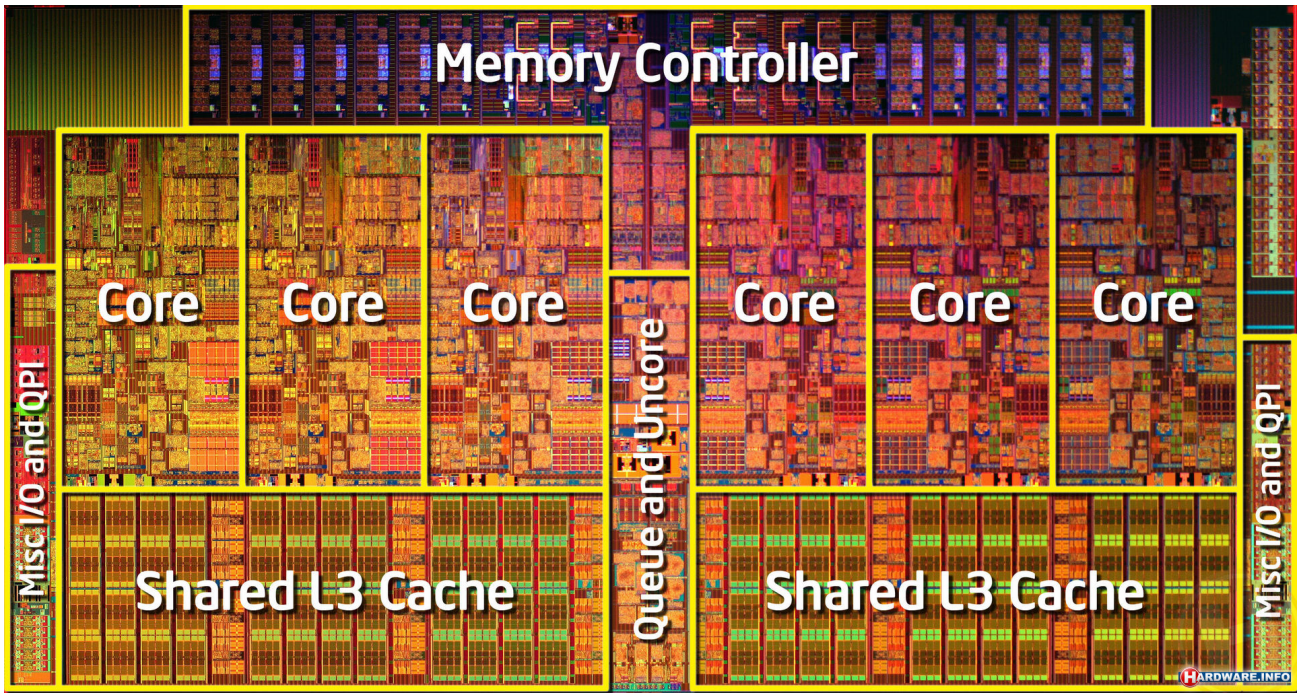


"the behavior of the computer at any moment is determined by the symbols which he [the computer] is observing, and his 'state of mind' at that moment" — Alan Turing



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

Die Realität



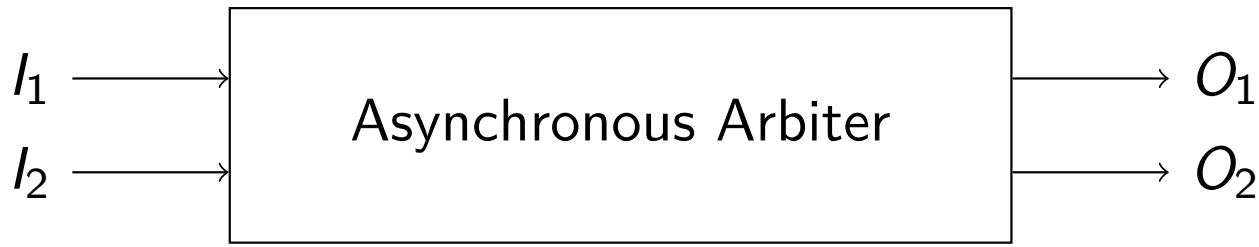
- $3GHz = 3'000'000'000Hz \Longrightarrow$ Ein Takt $= 3,333 * 10^{-10}s$
- $c = 299'792'458\frac{m}{s}$
- Maximaler Weg in einem Takt $< 0,1m$ (Physikalische Grenze)

Synchronisation

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

Der Arbiter

- Die Lösung: Asynchrone Arbiter



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

Unbounded Nondeterminism

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

Beispiel

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

Das Aktorenmodell

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

- Frei nach der relationalen Quantenphysik

Drei Grundlagen

- Verarbeitung
- Speicher
- Kommunikation

- Die (nichtdeterministische) Turingmaschine ist ein Spezialfall des Aktorenmodells
- Ein Aktorensystem besteht aus Aktoren (Alles ist ein Aktor!)

Aktoren

- Ein Aktor verarbeitet Nachrichten

Während ein Aktor eine Nachricht verarbeitet kann er

- neue Aktoren erzeugen
- Nachrichten an bekannte Aktor-Referenzen versenden
- festlegen wie die nächste Nachricht verarbeitet werden soll

- Aktor $\neq$ (Thread | Task | Channel | ...)

Aktoren

- Ein Aktor verarbeitet Nachrichten

Während ein Aktor eine Nachricht verarbeitet kann er

- neue Aktoren erzeugen
- Nachrichten an bekannte Aktor-Referenzen versenden
- festlegen wie die nächste Nachricht verarbeitet werden soll

- Aktor $\neq$ (Thread | Task | Channel | ...)

Ein Aktor kann (darf) nicht

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

Aktoren (Technisch)

- Aktor $\approx$ Schleife über unendliche Nachrichtenliste + Zustand (Verhalten)

Aktoren (Technisch)

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

Aktoren (Technisch)

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

Verhalten vs. Protokoll

Verhalten

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

Protokoll

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

- Beispiel:

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

$$\exists a(b, Ping)\ \mathcal{U}\ \Diamond b(Pong)$$

Kommunikation

- Nachrichten sind unveränderliche Daten, reine Funktionen oder Futures
- Die Zustellung von Nachrichten passiert höchstens einmal (Best-effort)

- Wenn z.B. die Netzwerkverbindung abbricht, wird gewartet, bis der Versand wieder möglich ist
- Wenn aber z.B. der Computer direkt nach Versand der Nachricht explodiert (oder der Speicher voll läuft), kommt die Nachricht möglicherweise niemals an

- Über den Zeitpunkt des Empfangs kann keine Aussage getroffen werden (Unbounded indeterminacy)
- Über die Reihenfolge der Empfangenen Nachrichten wird im Aktorenmodell keine Aussage gemacht (In vielen Implementierungen allerdings schon)
- Nachrichtenversand $\neq$ (Queue | Lock | Channel | ...)

Kommunikation (Technisch)

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

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

Location Transparency

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

Sicherheit in Aktorsystemen

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

Inkonsistenz in Aktorsystemen

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

Eventual Consistency

Definition

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

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

Fehlerbehandlung in Aktorsystemen

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

"Let it Crash!"(Nach Joe Armstrong)

- Unbegrenzter Nichtdeterminismus ist statisch kaum analysierbar
- Unschärfe beim Testen von verteilten Systemen
- Selbst wenn ein Programm fehlerfrei ist kann Hardware ausfallen
- Je verteilter ein System umso wahrscheinlicher geht etwas schief

  - Deswegen:
    - Offensives Programmieren
    - Statt Fehler zu vermeiden, Fehler behandeln!
    - Teile des Programms kontrolliert abstürzen lassen und bei Bedarf neu starten

Das Aktorenmodell in der Praxis

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

Zusammenfassung

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

Reaktive Programmierung
Vorlesung 9 vom 26.05.15: Actors in Akka

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
  - Futures and Promises
  - Das Aktorenmodell
  - Aktoren und Akka
  - Reaktive Datenströme I
  - Reaktive Datenströme II
- Teil III: Fortgeschrittene Konzepte

Aktoren in Scala

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

Akka

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

Rückblick

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

Aktoren in Akka

```
trait Actor {
  type Receive = PartialFunction[Any,Unit]

  def receive: Receive

  implicit val context: ActorContext
  implicit final val self: ActorRef
  final def sender: ActorRef

  def preStart()
  def postStop()
  def preRestart(reason: Throwable, message: Option[Any])
  def postRestart(reason: Throwable)

  def supervisorStrategy: SupervisorStrategy
  def unhandled(message: Any)
}
```

Aktoren Erzeugen

```
object Count

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

Aktoren Erzeugen

```
object Count

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

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

Aktoren Erzeugen

```
object Count

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

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

Global:

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

Aktoren Erzeugen

```
object Count

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

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

Global:

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

In Aktoren:

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

Nachrichtenversand

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

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

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

counter ! Count
```

"!" ist asynchron – Der Kontrollfluss wird sofort an den Aufrufer zurückggegeben.

Eigenschaften der Kommunikation

- Nachrichten die aus dem selben Aktor versendet werden 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)

Verhalten

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

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

Nachrichten werden sequenziell abgearbeitet.

Modellieren mit Aktoren

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

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

Beispiel

Aktorpfade

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

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

- `ActorSelection` $\neq$ `ActorRef`

Location Transparency und Akka Remoting

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

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

remoteCounter ! Count
```

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

src/resource/application.conf:

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

Supervision und Fehlerbehandlung in Akka

- OneForOneStrategy vs. AllForOneStrategy

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

Aktorsysteme Testen

- Um Aktorsyteme zu testen müssen wir eventuell die Regeln brechen:

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

- Oder: Integrationstests mit `TestKit`

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

Event-Sourcing (Akka Persistence)

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

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

akka-http (ehemals Spray)

- Aktoren sind ein hervorragendes Modell für Webserver
- akka-http ist ein minimales HTTP interface für Akka

```
val serverBinding = Http(system).bind(
    interface = "localhost", port = 80)

...

val requestHandler: HttpRequest ⇒ HttpResponse = {
  case HttpRequest(GET, Uri.Path("/ping"), _, _, _) ⇒
      HttpResponse(entity = "PONG!")
  ...
}
```

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

Bewertung

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

Zusammenfassung

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

Reaktive Programmierung
Vorlesung 10 vom 02.06.15: Reactive Streams (Observables)

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
  - Futures and Promises
  - Das Aktorenmodell
  - Aktoren und Akka
  - Reaktive Datenströme - Observables
  - Reaktive Datenströme - Back Pressure und Spezifikation
  - Reaktive Datenströme - Akka Streams
- Teil III: Fortgeschrittene Konzepte

Klassifikation von Effekten

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

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

Future[T] ist dual zu Try[T]

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

Future[T] ist dual zu Try[T]

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

- (Try[T] ⇒ Unit)⇒ Unit

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])

`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])`

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 $\longrightarrow$ `Try[T]`
- `Future[T]`: Callback $\longrightarrow$ `Try[T]` (Reaktiv)

Was ist dual zu `Iterable`?

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

Was ist dual zu `Iterable`?

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

- $() \Rightarrow$

Was ist dual zu `Iterable`?

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

- `() ⇒ () ⇒ Try[Option[T]]`

Was ist dual zu `Iterable`?

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

- `() ⇒ () ⇒ Try[Option[T]]`
- Umgedreht:
  `(Try[Option[T]] ⇒ Unit)⇒ Unit`

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 onCompleted(): Unit
}

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

Warum Observables?

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

(robotA.goldAmounts zip robotB.goldAmounts)
  .map(_ + _).takeUntil(_ > 5)
```

Observable Robots

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

(robotA.goldAmounts zip robotB.goldAmounts)
  .map(_ + _).takeUntil(_ > 5)
```

Observables Intern

DEMO

Observable Contract

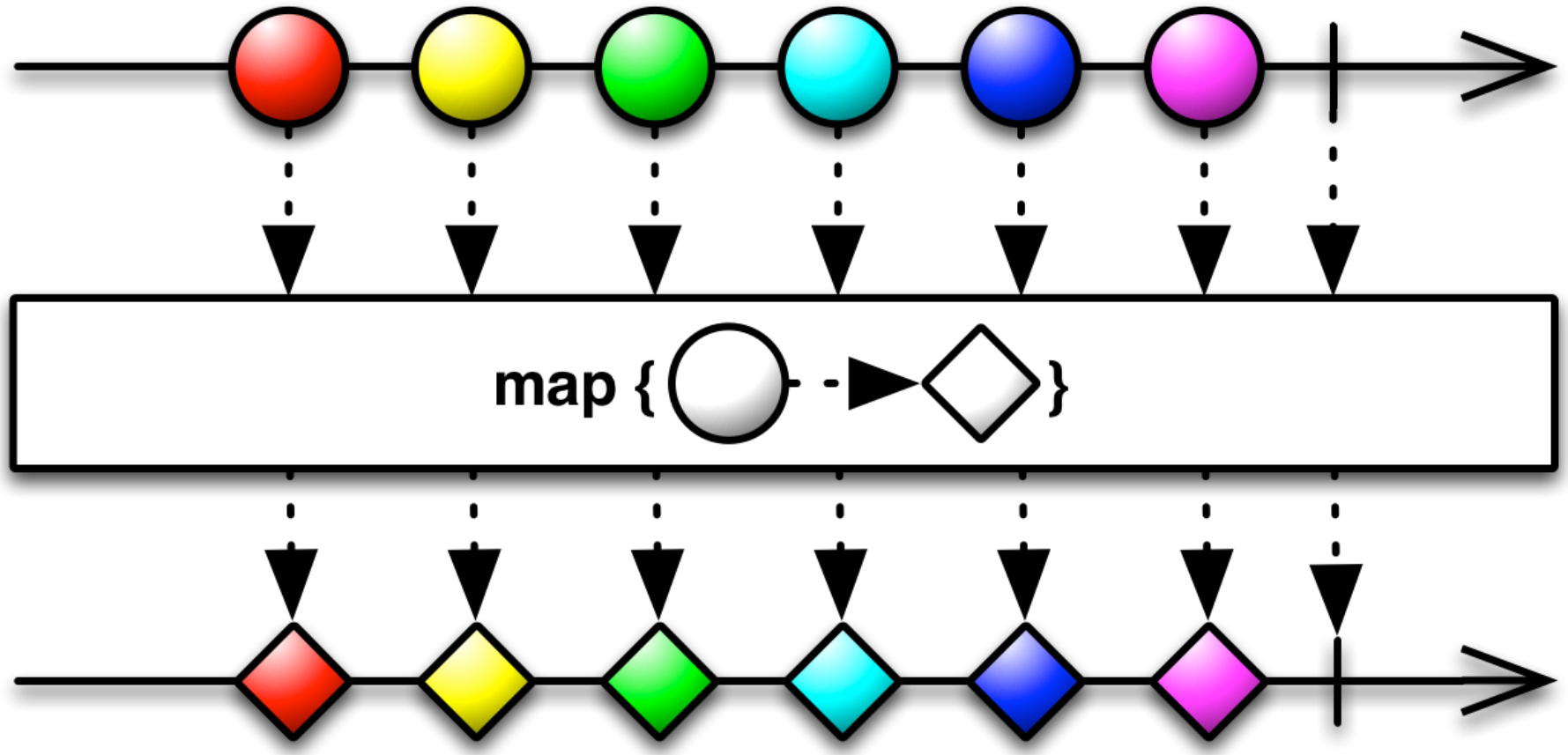
- die `onNext` Methode eines Observers wird beliebig oft aufgerufen.
- `onCompleted` oder `onError` werden nur einmal aufgerufen und schließen sich gegenseitig aus.
- Nachdem `onCompleted` oder `onError` aufgerufen wurde wird `onNext` nicht mehr aufgerufen.

```
onNext*(onCompleted|onError)?
```

- Diese Spezifikation wird durch die Konstruktoren erzwungen.

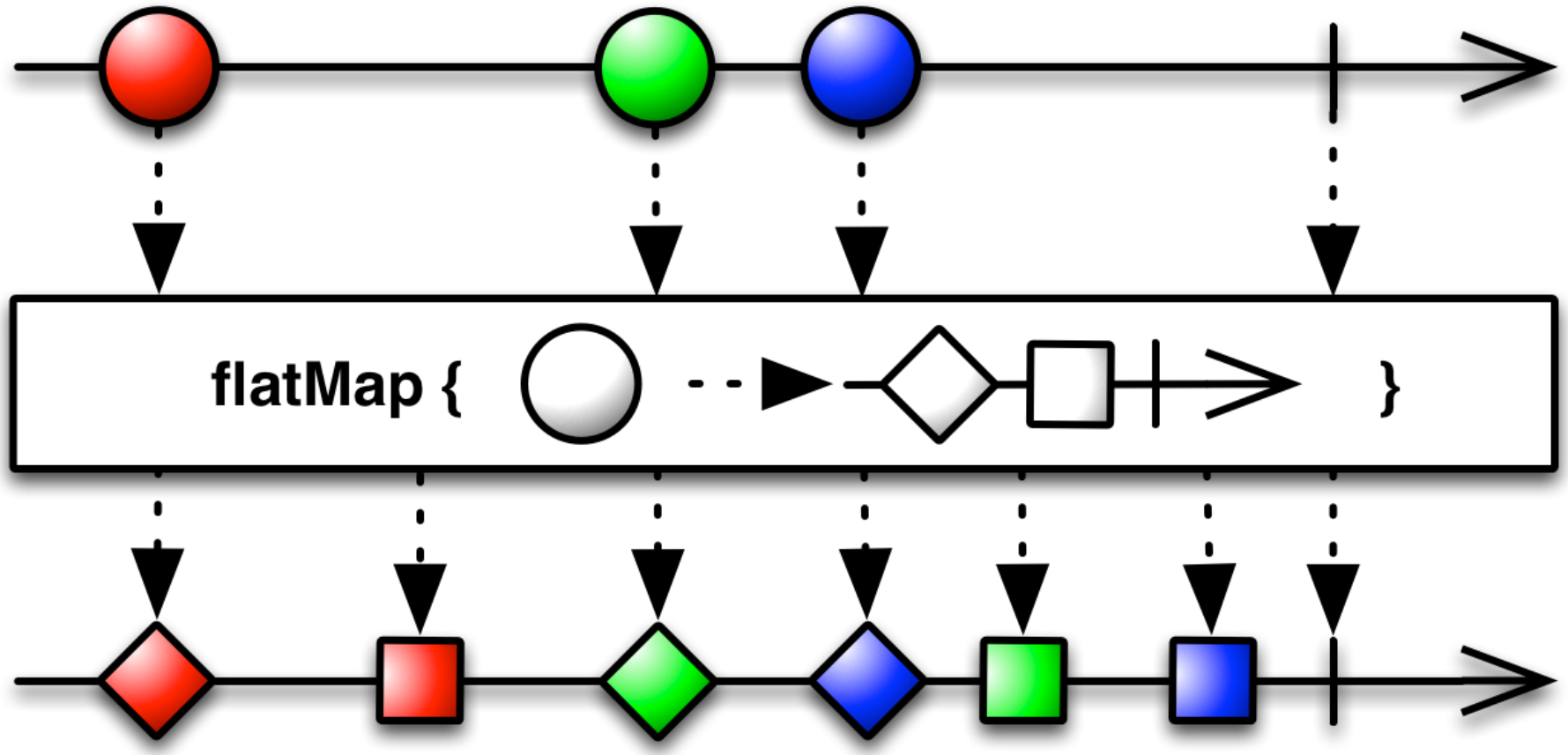
map

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



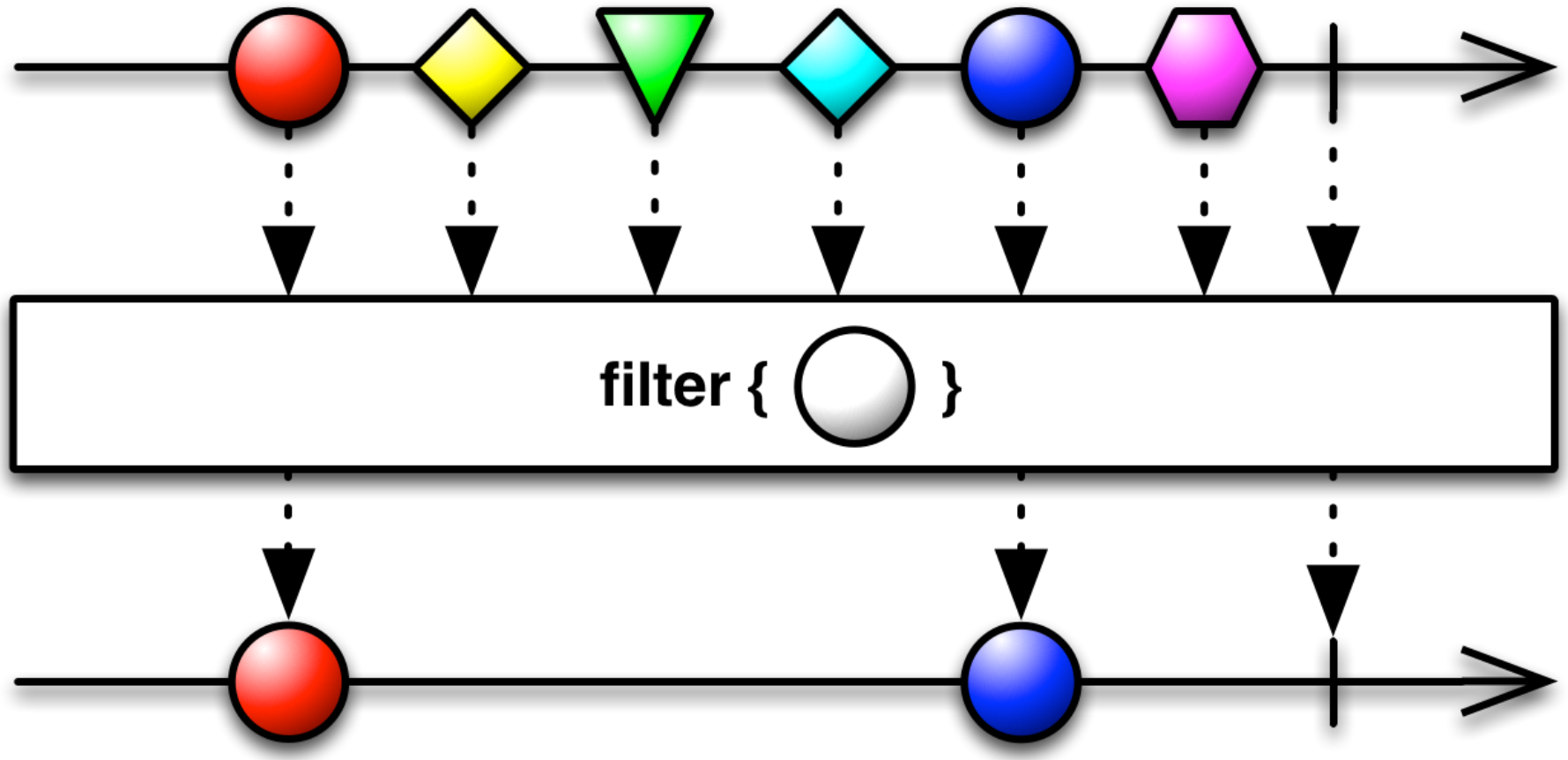
flatMap

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



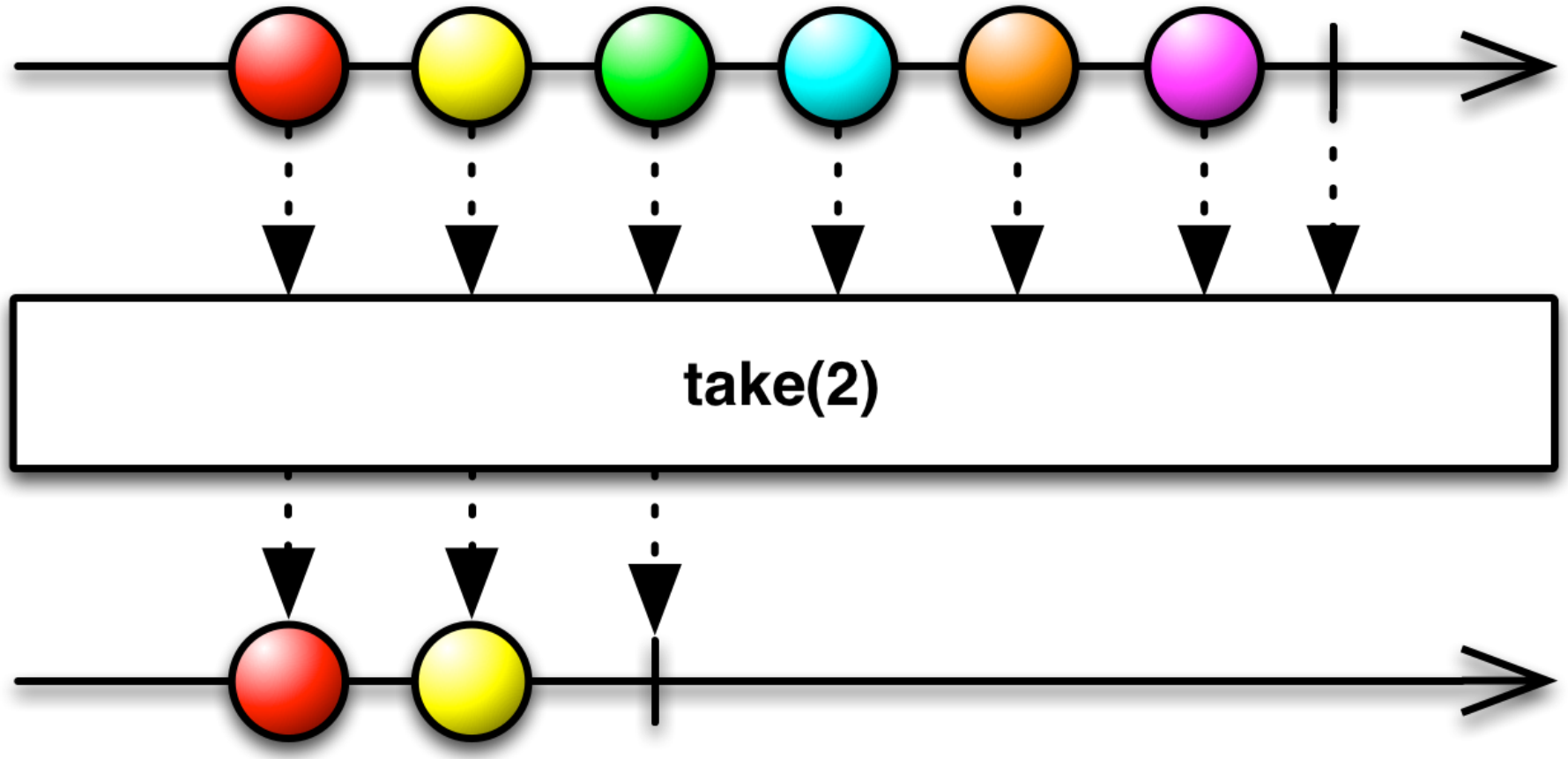
filter

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



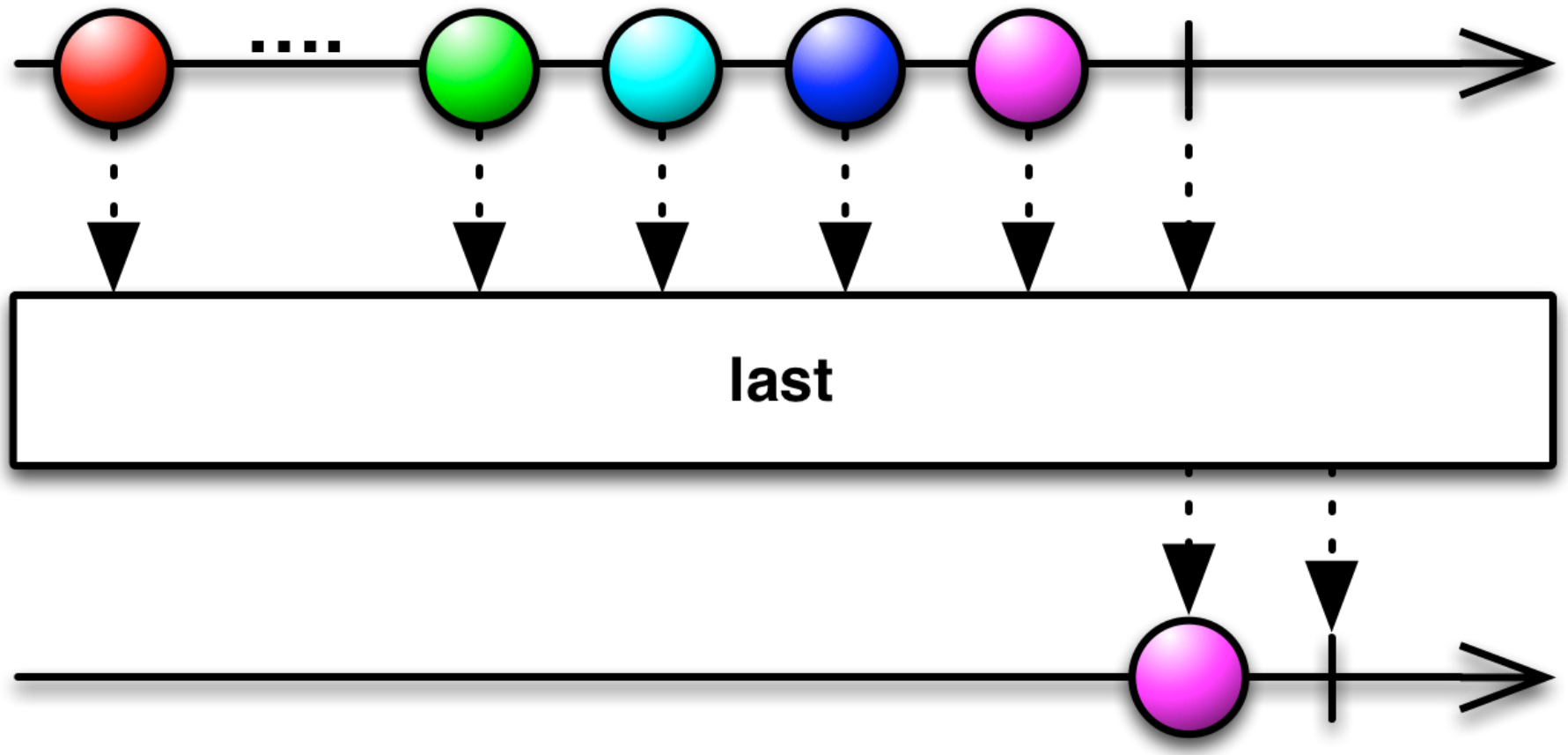
take

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



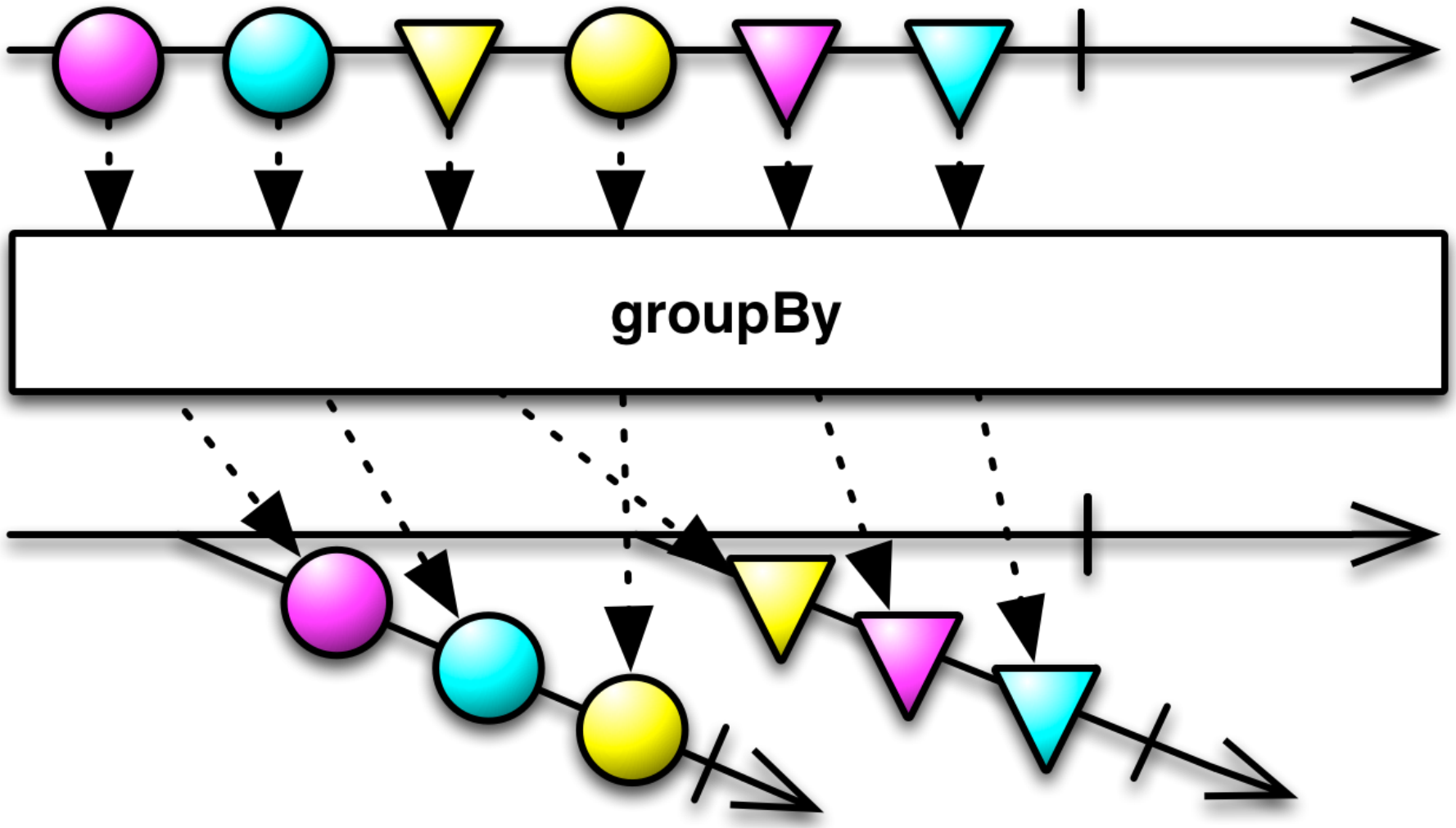
last

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



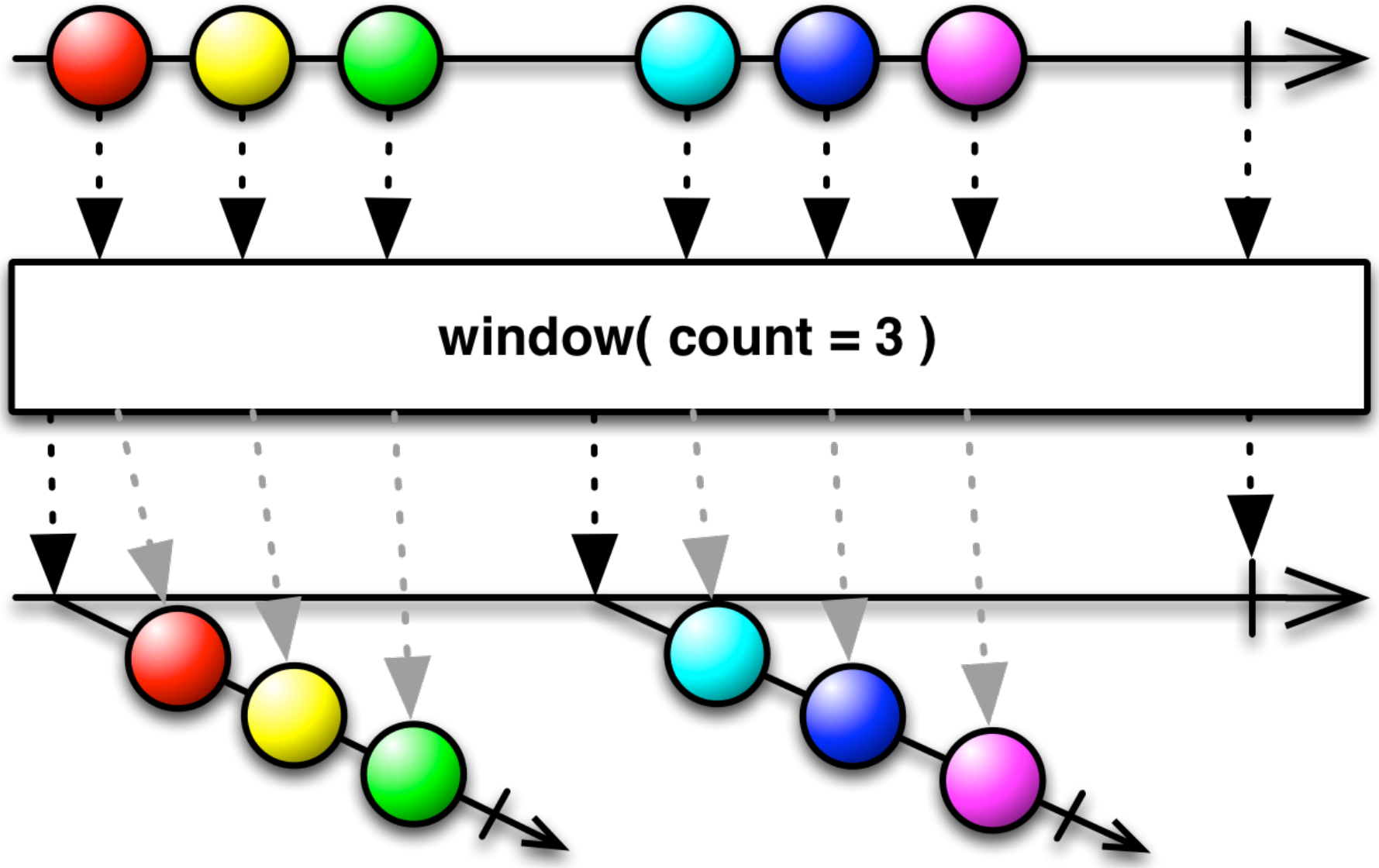
groupBy

```
def groupBy[U](T ⇒ U): Observable[Observable[T]]
```



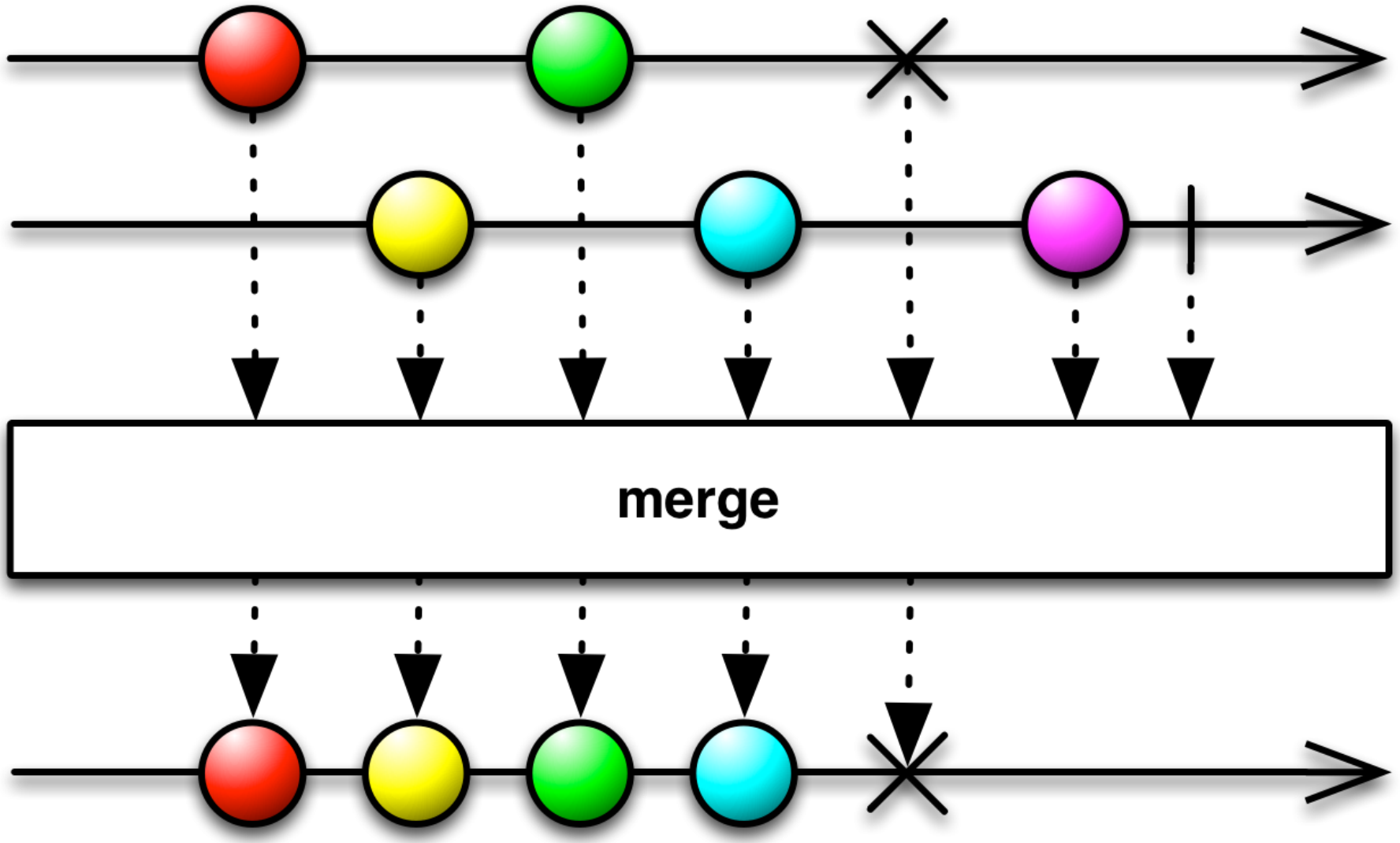
window

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



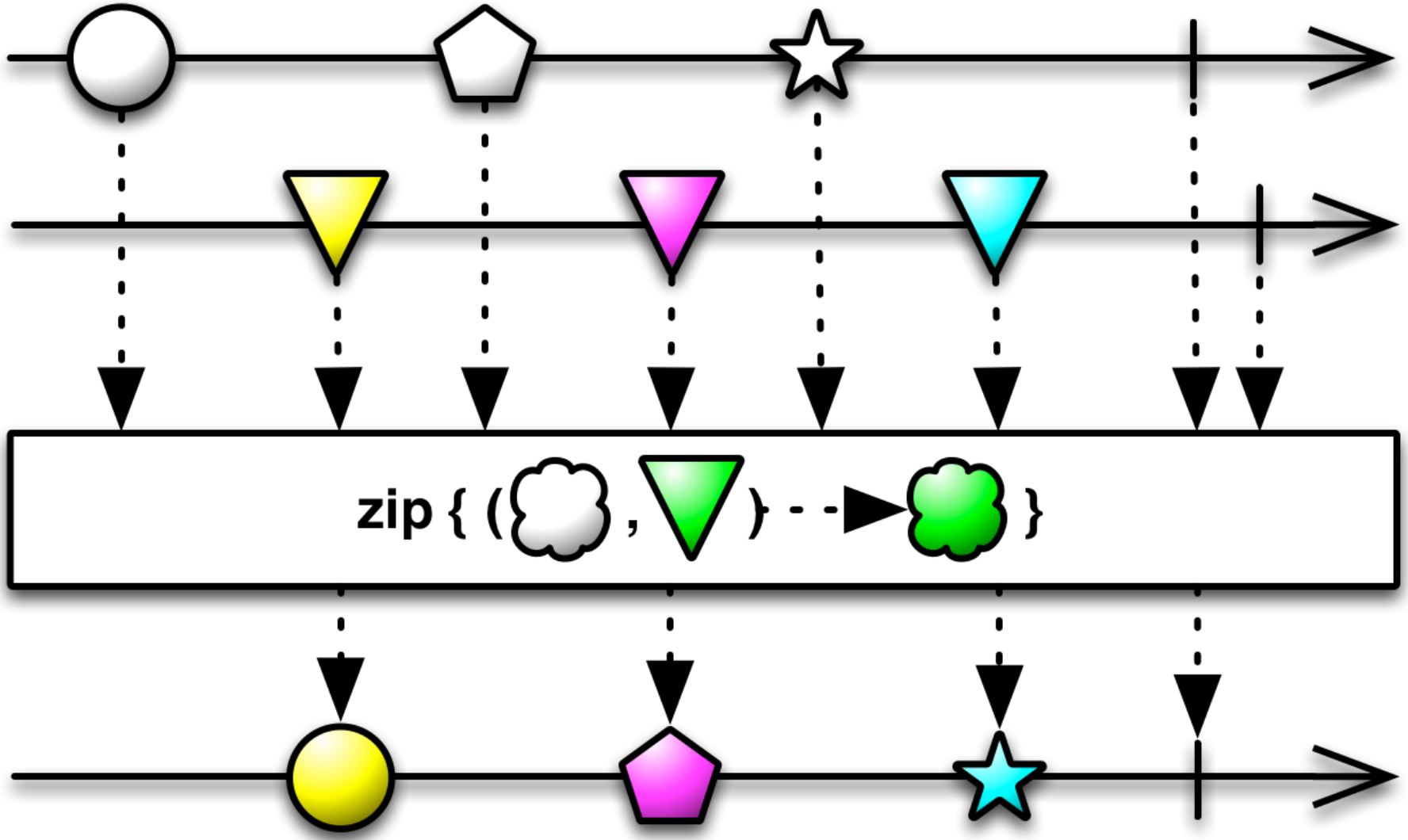
merge

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



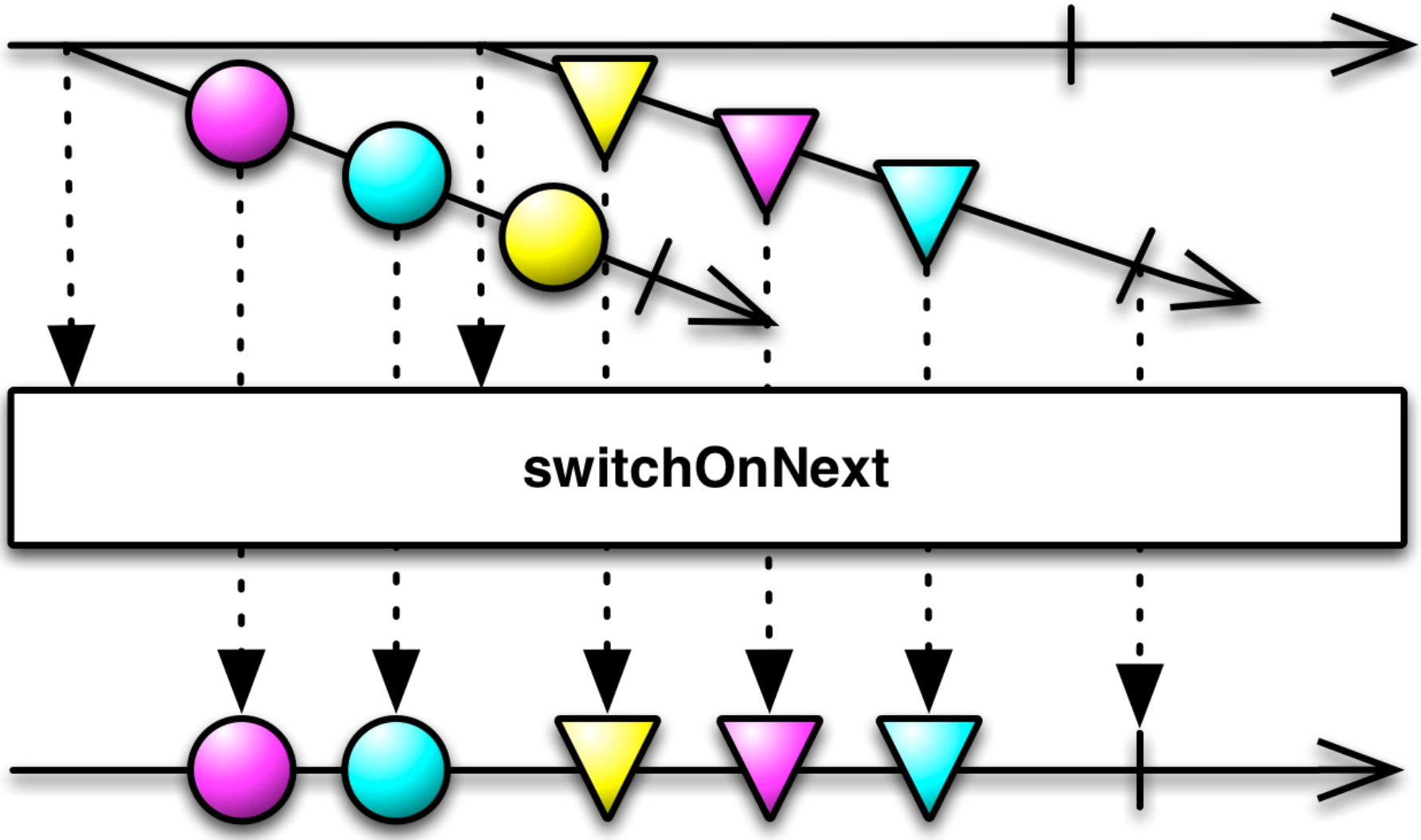
zip

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



switch

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



Subscriptions

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

```
Subscription(cancel: ⇒ Unit)

BooleanSubscription(cancel: ⇒ Unit)

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

CompositeSubscription(subscriptions: Subscription*)
```

Schedulers

- Nebenläufigkeit über `Scheduler`

```
trait Scheduler {
  def schedule(work: ⇒ Unit): Subscription
}

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

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

Hot vs. Cold Streams

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

  z.B. Maus Klicks

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

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

Observables Bibliotheken

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

Zusammenfassung

- Futures sind dual zu Try
- Observables sind dual zu Iterable
- Observables abstrahieren viele Nebenläufigkeitsprobleme weg:

  Außen funktional (Hui) - Innen imperativ (Pfui)
- Nächstes mal: Back Pressure und noch mehr reaktive Ströme

Reaktive Programmierung
Vorlesung 11 vom 09.06.15: Reactive Streams II

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
  - Futures and Promises
  - Das Aktorenmodell
  - Aktoren und Akka
  - Reaktive Datenströme - Observables
  - Reaktive Datenströme - Back Pressure und Spezifikation
  - Reaktive Datenströme - Akka Streams
- Teil III: Fortgeschrittene Konzepte

Rückblick: Observables

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

Datenstromgesetze

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

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

- Wir können die Gesetze erzwingen: `CODE DEMO`

Fehlerbehandlung

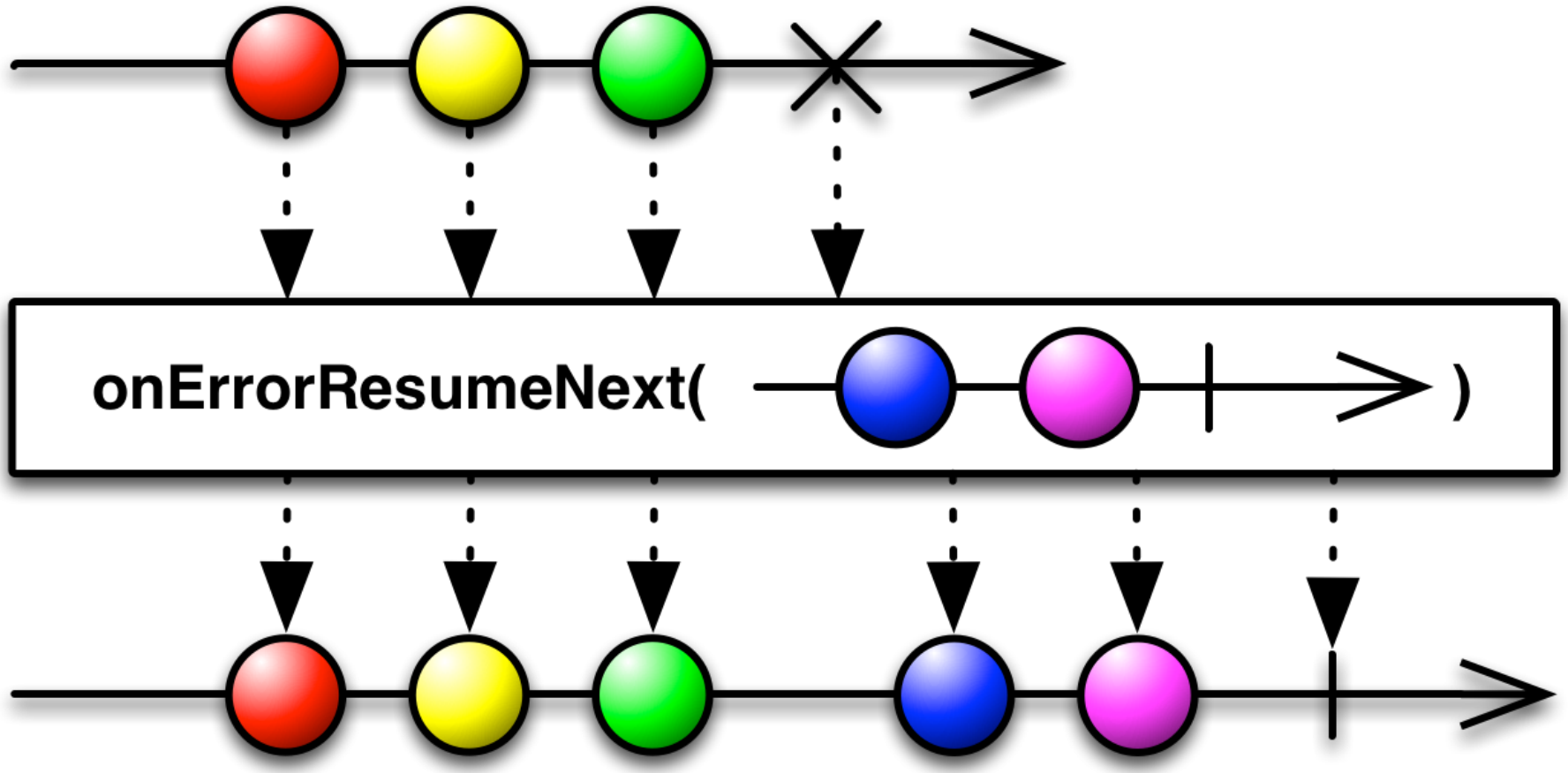
- Wenn Datenströme Fehler produzieren, können wir diese möglicherweise behandeln.
- Aber: *Observer.onError* beendet den Strom.

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

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

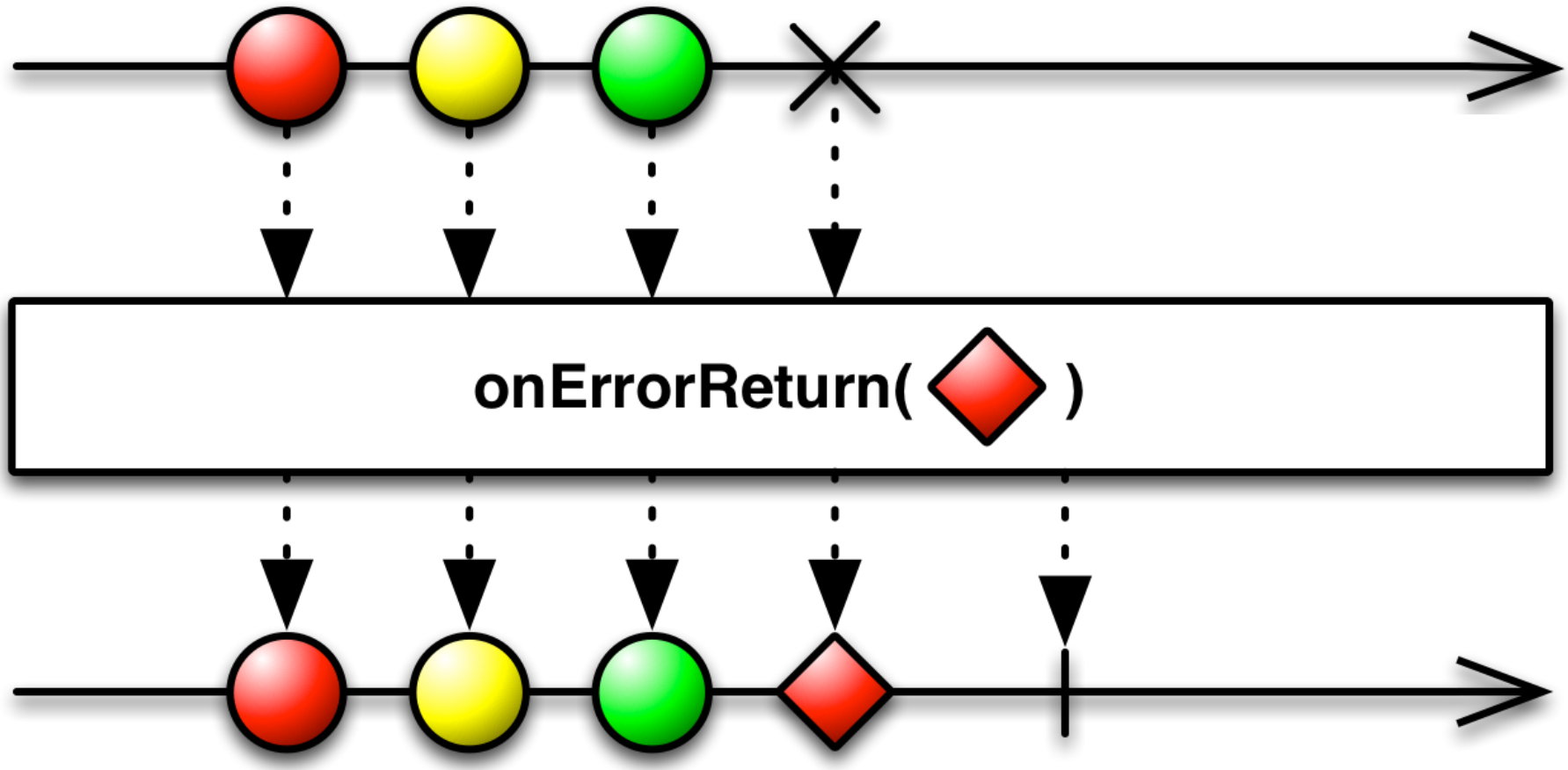
onErrorResumeNext

```
def onErrorResumeNext(f: ⇒ Observable[T]): Observable[T]
```



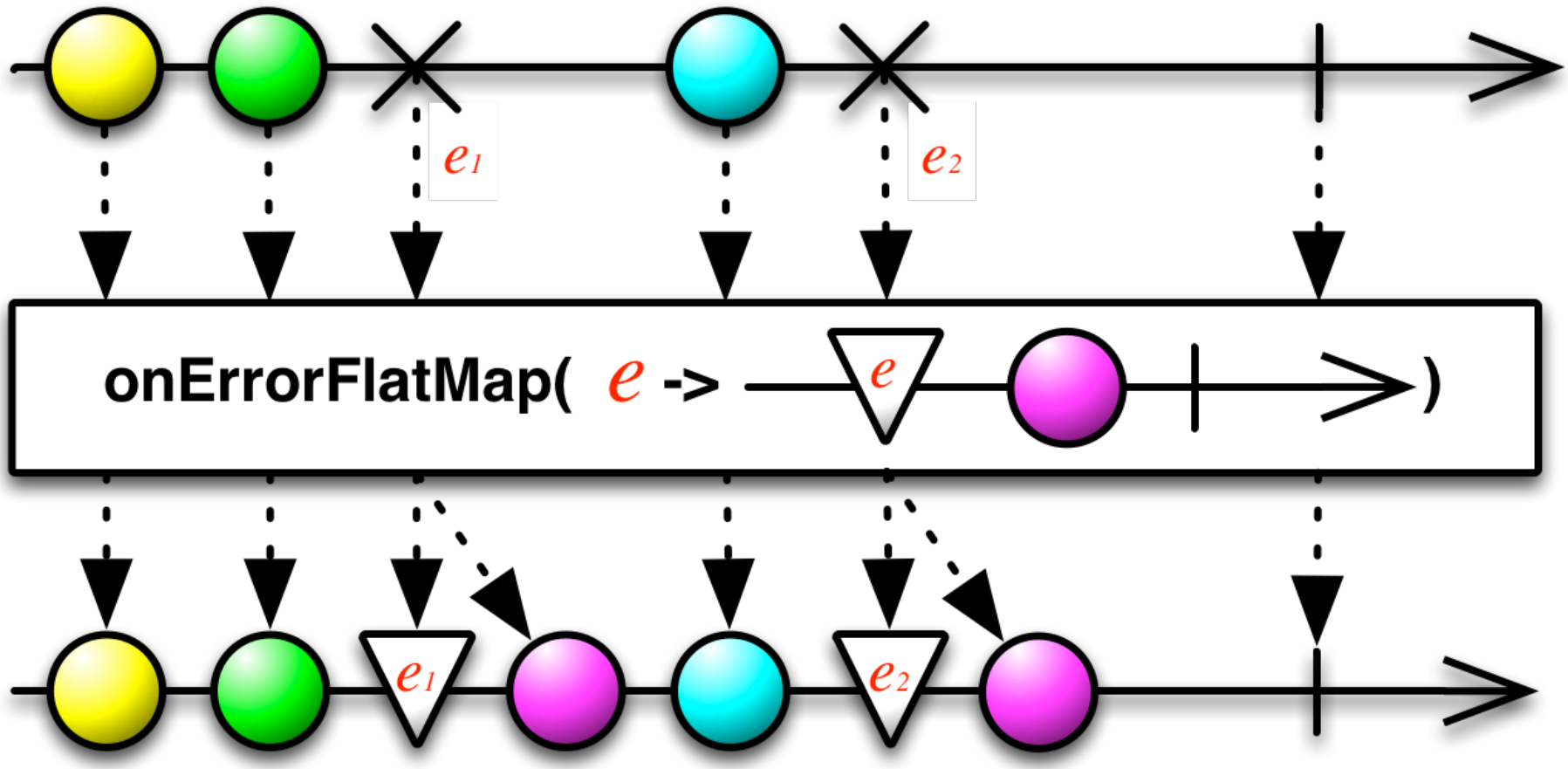
onErrorReturn

```
def onErrorReturn(f: ⇒ T): Observable[T]
```



onErrorFlatMap

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



Schedulers

- Nebenläufigkeit über `Scheduler`

```
trait Scheduler {
  def schedule(work: ⇒ Unit): Subscription
}

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

- CODE DEMO

Littles Gesetz

- In einer stabilen Warteschlange gilt:

$$L = \lambda \times W$$

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

Littles Gesetz

- In einer stabilen Warteschlange gilt:

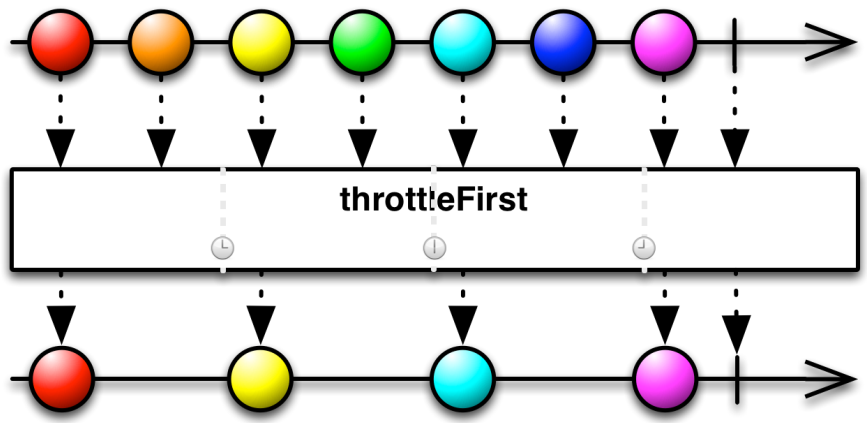
$$L = \lambda \times W$$

- Länge der Warteschlange = Ankunftsrate $\times$ Durschnittliche 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 λ bestimmen. Wenn λ überschritten wird, müssen wir etwas unternehmen.
- Idee: Throttling

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

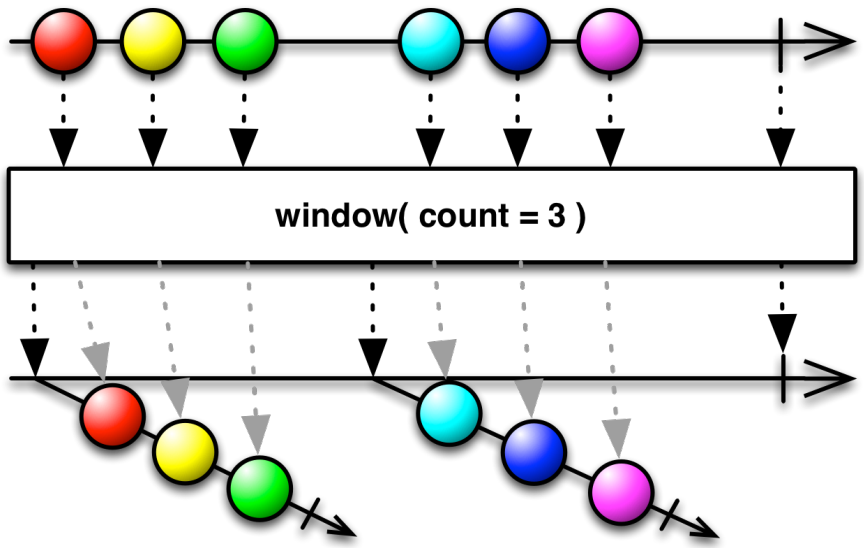


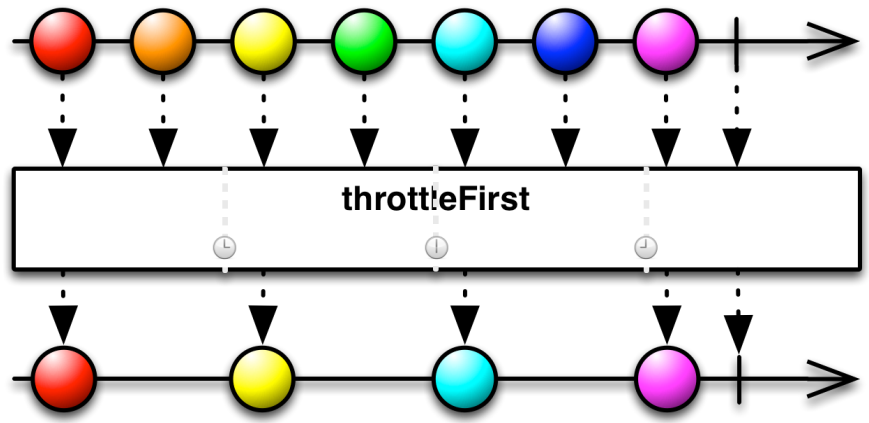
- Problem: Kurzzeitige Überschreigungen von λ sollen nicht zu Throttling führen.

Throttling / Debouncing

- Besser: Throttling erst bei längerer Überschreitung der Kapazität:

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

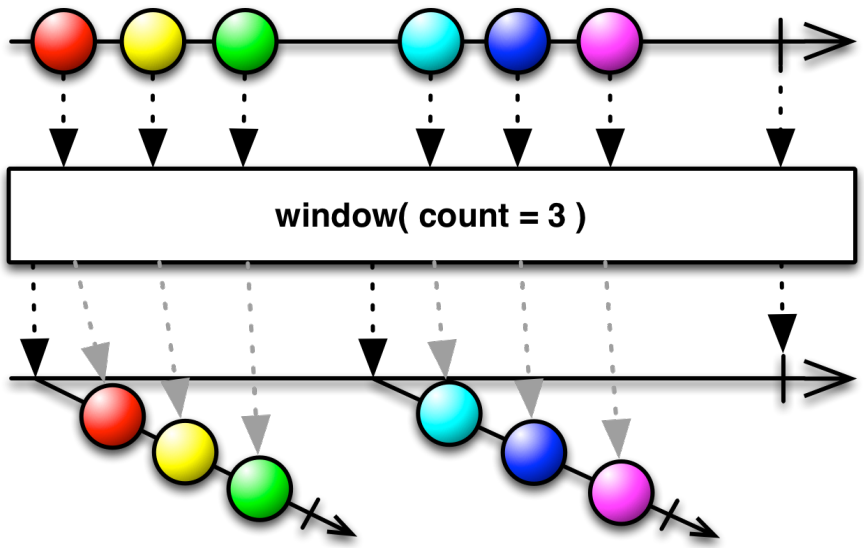


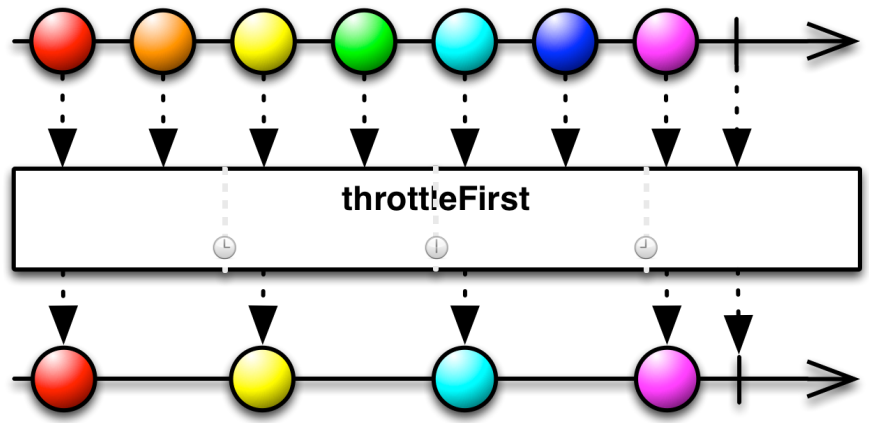


Throttling / Debouncing

- Besser: Throttling erst bei längerer Überschreitung der Kapazität:

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





- Was ist wenn wir selbst die Daten Produzieren?

Back Pressure

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

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

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

Back Pressure

- Bessere Idee: der Konsument muss mehr Kontrolle bekommen!

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

- Aufwändig in Observables zu implementieren!

- Siehe `http://www.reactive-streams.org/`

Reactive Streams Initiative

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

Reactive Streams: Interfaces

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

Reactive Streams: 1. `Publisher[T]`

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

1. The total number of `onNext` signals sent by a `Publisher` to a `Subscriber` MUST be less than or equal to the total number of elements requested by that `Subscriber`'s `Subscription` at all times.

2. A `Publisher` MAY signal less `onNext` than requested and terminate the `Subscription` by calling `onComplete` or `onError`.

3. `onSubscribe`, `onNext`, `onError` and `onComplete` signaled to a `Subscriber` MUST be signaled sequentially (no concurrent notifications).

4. If a `Publisher` fails it MUST signal an `onError`.

5. If a `Publisher` terminates successfully (finite stream) it MUST signal an `onComplete`.

6. If a `Publisher` signals either `onError` or `onComplete` on a `Subscriber`, that `Subscriber`'s `Subscription` MUST be considered cancelled.

Reactive Streams: 1. `Publisher[T]`

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

7. Once a terminal state has been signaled (`onError`, `onComplete`) it is REQUIRED that no further signals occur.

8. If a `Subscription` is cancelled its `Subscriber` MUST eventually stop being signaled.

9. `Publisher.subscribe` MUST call `onSubscribe` on the provided `Subscriber` prior to any other signals to that `Subscriber` and MUST return normally, except when the provided `Subscriber` is null in which case it MUST throw a java.lang.NullPointerException to the caller, for all other situations the only legal way to signal failure (or reject the `Subscriber`) is by calling `onError` (after calling `onSubscribe`).

10. `Publisher.subscribe` MAY be called as many times as wanted but MUST be with a different `Subscriber` each time.

11. A `Publisher` MAY support multiple Subscribers and decides whether each `Subscription` is unicast or multicast.

Reactive Streams: 2. `Subscriber[T]`

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

1. A `Subscriber` MUST signal demand via `Subscription.request(long n)` to receive `onNext` signals.
2. If a `Subscriber` suspects that its processing of signals will negatively impact its `Publisher`'s 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`.

Reactive Streams: 2. `Subscriber[T]`

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

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

Reactive Streams: 2. `Subscriber[T]`

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

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.

Reactive Streams: 3. `Subscription`

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

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

Reactive Streams: 3. `Subscription`

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

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

Reactive Streams: 3. `Subscription`

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

12. While the `Subscription` is not cancelled, `Subscription.cancel()` MUST request the Publisher to eventually stop signaling its Subscriber. The operation is NOT REQUIRED to affect the `Subscription` immediately.

13. While the `Subscription` is not cancelled, `Subscription.cancel()` MUST request the Publisher to eventually drop any references to the corresponding subscriber. Re-subscribing with the same Subscriber object is discouraged, but this specification does not mandate that it is disallowed since that would mean having to store previously cancelled subscriptions indefinitely.

14. While the `Subscription` is not cancelled, calling `Subscription.cancel` MAY cause the `Publisher`, if stateful, to transition into the shut-down state if no other `Subscription` exists at this point.

Reactive Streams: 3. `Subscription`

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

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

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

18. A `Subscription` MUST support an unbounded number of calls to request and MUST support a demand (sum requested - sum delivered) up to $2^{63} - 1$ (`java.lang.Long.MAX_VALUE`). A demand equal or greater than $2^{63} - 1$ (`java.lang.Long.MAX_VALUE`) MAY be considered by the Publisher as "effectively unbounded".

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

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

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

2. A `Processor` MAY choose to recover an `onError` signal. If it chooses to do so, it MUST consider the `Subscription` cancelled, otherwise it MUST propagate the `onError` signal to its `Subscribers` immediately.

Akka Streams

- Vollständige Implementierung der Reactive Streams Spezifikation
- Basiert auf Datenflussgraphen und Materialisierern
- Datenflussgraphen werden als Aktornetzwerk materialisiert
- Fast final (aktuelle Version 1.0-RC3)

Akka Streams - Grundkonzepte

Datenstrom (Stream) – Ein Prozess der Daten überträgt und transformiert
Element – Recheneinheit eines Datenstroms
Back-Presure – 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
Abfulss (Sink) – Verarbeitungsschritt mit genau einem Eingang
Datenfluss (Flow) – Verarbeitungsschritt mit jeweils genau einem Ein- und Ausgang
Ausführbarer Datenfluss (RunnableFlow) – Datenfluss der an eine Quelle und einen Abfluss angeschlossen ist

Akka Streams - Beispiel

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

val source = Source(1 to 10)
val sink = Sink.fold[Int,Int](0)(_ + _)
val sum: Future[Int] = source runWith sink
```

Datenflussgraphen

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

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

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

Zusammenfassung

- Die Konstruktoren in der Rx Bibliothek wenden viel Magie an um Gesetze einzuhalten
- Fehlerbehandlung durch Kombinatoren ist einfach zu implementieren
- Observables eigenen 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
- Nächstes mal: Mehr Akka Streams und Integration mit Aktoren

Reaktive Programmierung
Vorlesung 12 vom 16.06.15: Reactive Streams III

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
  - Futures and Promises
  - Das Aktorenmodell
  - Aktoren und Akka
  - Reaktive Datenströme - Observables
  - Reaktive Datenströme - Back Pressure und Spezifikation
  - Reaktive Datenströme - Akka Streams
- Teil III: Fortgeschrittene Konzepte

Rückblick: Akka Streams

- Vollständige Implementierung der Reactive Streams Spezifikation
- Basiert auf Datenflussgraphen und Materialisierern
- Datenflussgraphen werden als Aktornetzwerk materialisiert
- Fast final (aktuelle Version 1.0-RC3)

Heute

- Datenflussgraphen
  - geschlossen
  - partiell
  - zyklisch
- Puffer und Back-Pressure
- Fehlerbehandlung
- Integration mit Aktoren
- Anwendungsbeispiel: akka-http
  - Routen
  - HTTP
  - WebSockets

Akka Streams - Grundkonzepte

Datenstrom (Stream) – Ein Prozess, der Daten überträgt und transformiert
Element – Recheneinheit eines Datenstroms
Back-Presure – 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
Senke (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 Senke angeschlossen ist

Akka Streams - Beispiel

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

val source = Source(1 to 10)
val sink = Sink.fold[Int,Int](0)(_ + _)
val sum: Future[Int] = source runWith sink
```

Datenflussgraphen

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

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

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

Operatoren in Datenflussgraphen

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

Partielle Datenflussgraphen

- Datenflussgraphen können partiell sein:

```
val pickMaxOfThree = FlowGraph.partial() {
  implicit builder ⇒

  val zip1 = builder.add(ZipWith[Int,Int,Int](math.max))
  val zip2 = builder.add(ZipWith[Int,Int,Int](math.max))

  zip1.out ~> zip2.in0

  UniformFanInShape(zip2.out, zip1.in0, zip1.in1,
      zip2.in1)
}
```

- Offene Anschlüsse werden später belegt

Sources, Sinks und Flows als Datenflussgraphen

- Source — Graph mit genau einem offenen Ausgang

```
Source(){ implicit builder ⇒
  outlet
}
```

- Sink — Graph mit genau einem offenen Eingang

```
Sink() { implicit builder ⇒
  inlet
}
```

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

```
Flow() { implicit builder ⇒
  (inlet,outlet)
}
```

Zyklische Datenflussgraphen

- Zyklen in Datenflussgraphen sind erlaubt:

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

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

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

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

Zyklische Datenflussgraphen

- Besser:

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

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

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

Pufferung

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

```
Source(1 to 3)
  .map( i ⇒ println(s"A: $i"); i)
  .map( i ⇒ println(s"B: $i"); i)
  .map( i ⇒ println(s"C: $i"); i)
  .map( i ⇒ println(s"D: $i"); i)
  .runWith(Sink.ignore)
```

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

Fehlerbehandlung

- Standardmäßig führen Fehler zum Abbruch:

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

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

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

- `result = Future(Success(228))`

Integration mit Aktoren - ActorPublisher

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

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

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

Integration mit Aktoren - ActorSubscriber

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

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

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

Integration für einfache Fälle

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

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

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

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

Anwendung: akka-http

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

Low-Level Server API

- HTTP-Server wartet auf Anfragen:
  `Source[IncomingConnection, Future[ServerBinding]]`

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

- Zu jeder Anfrage gibt es eine Antwort:

```
val requestHandler: HttpRequest ⇒ HttpResponse = {
  case HttpRequest(GET,Uri.Path("/ping"), _, _, _) ⇒
    HttpResponse(entity = "PONG!")
}

val serverSink =
  Sink.foreach(_.handleWithSyncHandler(requestHandler))

serverSource.to(serverSink)
```

High-Level Server API

- Minimalbeispiel:

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

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

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

HTTP

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

Das Server-Push Problem

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

WebSockets

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

WebSockets in akka-http

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

```
def routes = get {
  path("ping")(handleWebsocketMessages(wsFlow))
}

def wsFlow: Flow[Message,Message,Unit] =
  BidiFlow.fromFunctions(serialize,deserialize)
          .join(Flow.collect {
            case Ping ⇒ Pong
          })
```

Zusammenfassung

- 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` $\Rightarrow$ `Response`
  - HTTP ist pull basiert
  - WebSockets sind bidirektional $\rightarrow$ 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 WebWorker(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 13 vom 23.06.14: Bidirektionale Programmierung: Zippers and Lenses

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte
  - Bidirektionale Programmierung: Zippers and Lenses
  - Eventual Consistency
  - Robustheit, Entwurfsmuster
  - Theorie der Nebenläufigkeit

Was gibt es heute?

- Motivation: funktionale Updates
  - Akka ist *stateful*, aber im allgemeinen ist funktional besser
  - Globalen Zustand vermeiden hilft der Skalierbarkeit und der Robustheit
- Der Zipper
  - Manipulation innerhalb einer Datenstruktur
- Linsen
  - Bidirektionale Programmierung

Ein einfacher Editor

- Datenstrukturen:

```
type Text  = List[String]
case class Pos(line: Int, col: Int)
case class Editor(text: Text, cursor: Pos)
```

- Operationen: Cursor bewegen (links)

```
def goLeft: Editor =
  if (cursor.col == 0) sys.error("At start of line")
  else Editor(text, cursor.copy(col = cursor.col - 1))
```

Beispieloperationen

- Text rechts einfügen:

```
def insertRight(s: String): Editor = {
  val (befor,after) =
      text(cursor.line).splitAt(cursor.col)
  val newLine = before + s + after
  val newText = text.take(cursor.line) ++
    (newLine :: text.drop(cursor.line + 1))
  Editor(newText,cursor)
}
```

- Problem: Aufwand für Manipulation

Manipulation strukturierter Datentypen

- Anderes Beispiel: *n*-äre Bäume (rose trees)

```
sealed trait Tree[A]
case class Leaf[A](a: A) extends Tree[A]
case class Node[A](children: Tree[A]*) extends Tree[A]
```

  - Bsp: Abstrakte Syntax von einfachen Ausdrücken

- Update auf Beispielterm $t = a * b - c * d$: ersetze b durch $x + y$

```
val t = Node(Leaf("-"),
  Node(Leaf("*"), Leaf("a"), Leaf ("b")),
  Node(Leaf("*"), Leaf("c"), Leaf ("d"))
)
```

Der Zipper

- Idee: Kontext nicht wegwerfen!
- Nicht: `case class Path(i: Int*)`
- Sondern:

```
sealed trait Context[+A]
case object Empty extends Context[Nothing]
case class Cons[A](
  left: List[Tree[A]],
  up: Context[A],
  right: List[Tree[A]]) extends Context[A]
```

  - Kontext ist 'inverse Umgebung' (*"Like a glove turned inside out"*)
- `Location[A]` ist Baum mit Fokus

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

Zipping Trees: Navigation

- Fokus nach links

```
def goLeft: Location[A] = context match {
  case Cons(l::le,up,ri) ⇒
    Location(l, Cons(le,up,(t::ri)))
  case _ ⇒ sys.error("goLeft of first")
}
```

- Fokus nach rechts

```
def goRight: Location[A] = context match {
  case Cons(le,up,r::ri) ⇒
    Location(r,Cons(t::le,up,ri))
  case _ ⇒ sys.error("goRight of last")
}
```

Zipping Trees: Navigation

- Fokus nach oben

```
def goUp: Location[A] = context match {
  case Empty ⇒ sys.error("goUp of empty")
  case Cons(le,up,ri) ⇒
    Location(Node((le.reverse ++ t::ri) :_*), up)
}
```

- Fokus nach unten

```
def goDown: Location[A] = tree match {
  case Leaf(_) ⇒ sys.error("goDown at leaf")
  case Node() ⇒ sys.error("goDown at empty")
  case Node(t,ts@ _*) ⇒
    Loaction(t,Cons(Seq.empty,context,ts))
}
```

Zipping Trees: Navigation

- Hilfsfunktion (auf `Tree[A]`):

```
def top: Location[A] =
  Location(this,Empty)
```

- Damit andere Navigationsfunktionen:

```
def path(ps: List[Int]): Loaction[A] = ps match {
  case Nil  ⇒ this
  case i::ps if i == 0 ⇒ goDown.path(ps)
  case i::ps if i > 0 ⇒ goLeft.path((i-1)::ps)
}
```

Einfügen

- Einfügen: Wo?
- Links des Fokus einfügen

```
def insertLeft(t: Tree[A]): Loaction[A] = context match {
  case Empty ⇒ sys.error("insertLeft at empty")
  case Cons(le,up,ri) ⇒ Location(tree,Cons(t::le,up,ri))
}
```

- Rechts des Fokus einfügen

```
def insertRight(t: Tree[A]): Location[A] = context match {
  case Empty ⇒ sys.error("insertRight at empty")
  case Cons(le,up,ri) ⇒ Location(tree,Cons(le,up,t::ri))
}
```

Einfügen

- Unterhalb des Fokus einfügen

```
def insertDown(t: Tree[A]): Location[A] = tree match {
  case Leaf(_) ⇒ sys.error("insertDown at leaf")
  case Node(ts @_*) ⇒ Location(t,Cons(Nil,context,ts))
}
```

Ersetzen und Löschen

- Unterbaum im Fokus ersetzen:

```
def update(t: Tree): Location[A] =
  Location(t,context)
```

- Unterbaum im Fokus löschen: wo ist der neue Fokus?
  1. Rechter Baum, wenn vorhanden
  2. Linker Baum, wenn vorhanden
  3. Elternknoten

```
def delete: Location[A] = context match {
  case Empty ⇒ Location(Node(),Empty)
  case Cons(le,up,r::ri) ⇒ Location(r,Cons(le,up,ri))
  case Cons(l:le,up,Nil) ⇒ Location(l,Cons(le,up,Nil))
  case Cons(Nil,up,Nil) ⇒ Location(Node(),up)
}
```

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

Schnelligkeit

- Wie schnell sind Operationen?

Schnelligkeit

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

Schnelligkeit

- Wie schnell sind Operationen?
  - Aufwand: `goLeft` $O(left(n))$, alle anderen $O(1)$.
- Warum sind Operationen so schnell?
  - Kontext bleibt erhalten
  - Manipulation: reine Zeiger-Manipulation

Zipper für andere Datenstrukturen

- Binäre Bäume:

```
sealed trait Tree[+A]
case class Leaf(value: A) extends Tree[A]
case class Node(left: Tree[A],
                right: Tree[A]) extends Tree[A]
```

- Kontext:

```
sealed trait Context[+A]
case object Empty extends Context[Nothing]
case class Left[A](up: Context[A],
                   right: Tree[A]) extends Context[A]
case class Right[A](left: Tree[A],
                    up: Context[A]) extends Context[A]


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

Tree-Zipper: Navigation

- Fokus nach links

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

- Fokus nach rechts

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

Tree-Zipper: Navigation

- Fokus nach oben

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

- Fokus nach unten links

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

- Fokus nach unten rechts

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

Tree-Zipper: Einfügen und Löschen

- Einfügen links

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

- Einfügen rechts

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

- Löschen

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

  - Neuer Fokus: anderer Teilbaum

Zipping Lists

- Listen:

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class ::[A](head: A, tail: List[A])
  extends List[A]
```

- Damit:

```
sealed trait Context[+A]
case object Empty extends Context[Nothing]
case class Snoc[A](init: Context[A], last: A)
  extends Context[A]
```

- Listen sind ihr ‘eigener Kontext’ :

$$\mathsf{List}[A] \cong \mathsf{Context}[A]$$

Zipping Lists: Fast Reverse

- Listenumkehr schnell:

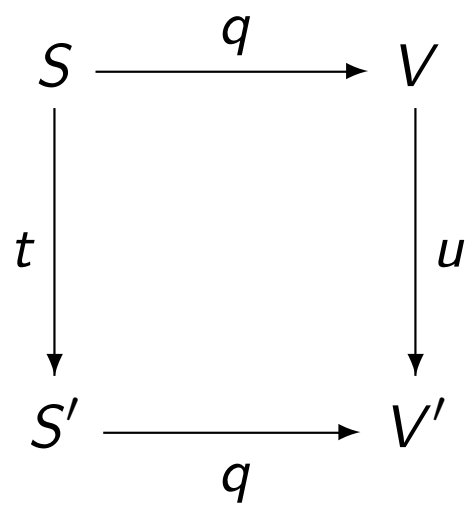
```
def reverse(init: List[A] = Nil) = this match {
  case Nil  ⇒ init
  case x::xs ⇒ xs.reverse(x::init)
}
```

- Argument von `reverse`: Kontext
  - Liste der Elemente davor in umgekehrter Reihenfolge

Bidirektionale Programmierung

- Motivierendes Beispiel: Update in einer Datenbank
- Weitere Anwendungsfelder:
  - Software Engineering (round-trip)
  - Benutzerschnittstellen (MVC)
  - Datensynchronisation

View Updates

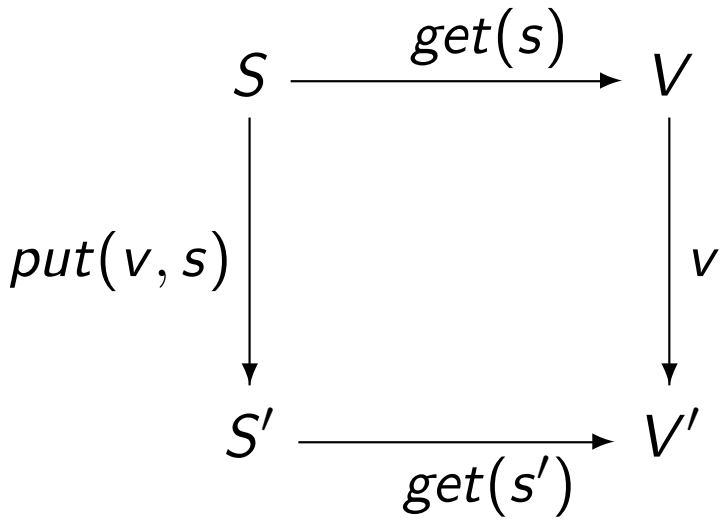


- View v durch Anfrage q (Bsp: Anfrage auf Datenbank)
- View wird verändert (Update u)
- Quelle S soll entsprechend angepasst werden (Propagation der Änderung)
- Problem: q soll beliebig sein
  - Nicht-injektiv? Nicht-surjektiv?

Lösung

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

Putting and Getting



- Signatur der Operationen:

$$\begin{aligned} get &: S \longrightarrow V \\ put &: V \times S \longrightarrow S \end{aligned}$$

- Es müssen die Linsengesetze gelten:

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

Erweiterung: Erzeugung

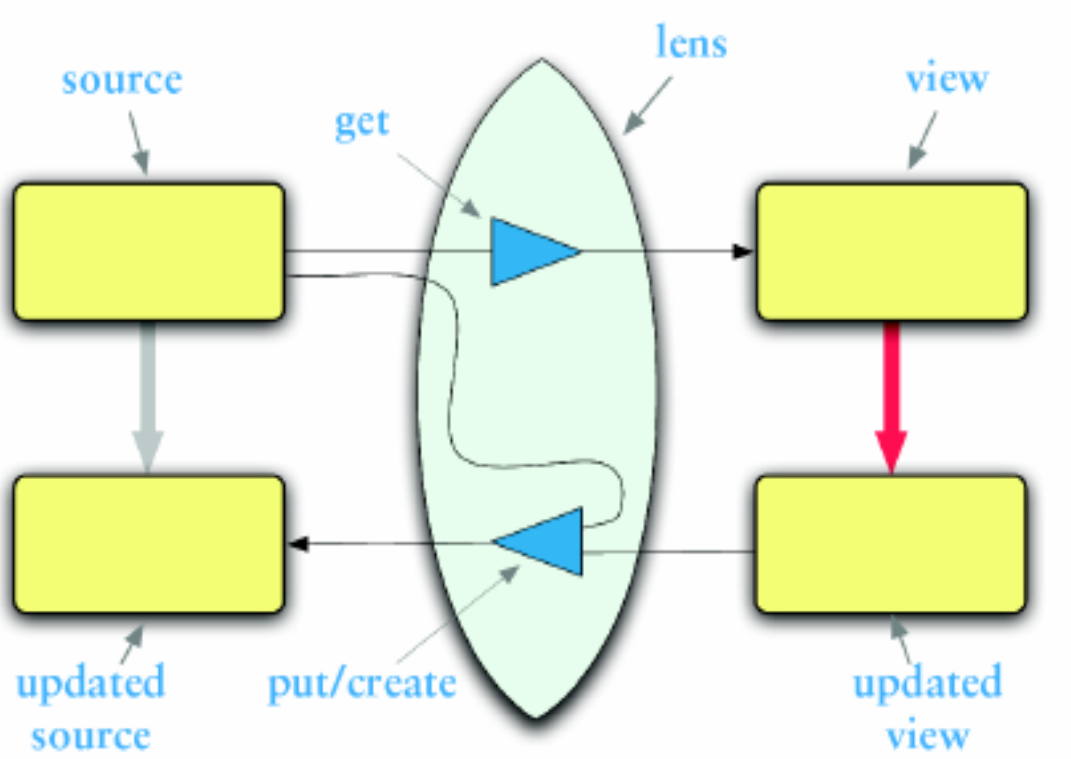
- Wir wollen auch Elemente (im Ziel) erzeugen können.
- Signatur:

$$create : V \longrightarrow S$$

- Weitere Gesetze:

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

Die Linse im Überblick



Linsen im Beispiel

- Updates auf strukturierten Datenstrukturen:

```
case class Turtle(
  position: Point =
      Point(),
  color: Color = Color(),
  heading: Double = 0.0,
  penDown: Boolean = false)
```

```
case class Point(
  x: Double = 0.0,
  y: Double = 0.0)

case class Color(
  r: Int = 0,
  g: Int = 0,
  b: Int = 0)
```

- Ohne Linsen: functional record update

```
scala> val t = new Turtle();
t: Turtle = Turtle(Point(0.0,0.0),Color(0,0,0),0.0,false)

scala> t.copy(penDown = ! t.penDown);
res5: Turtle = Turtle(Point(0.0,0.0),Color(0,0,0),0.0,true)
```

Linsen im Beispiel

- Das wird sehr schnell sehr aufwändig:

```
scala> def forward(t:Turtle) : Turtle =
    t.copy(position= t.position.copy(x= t.position.x+
        1));

forward: (t: Turtle)Turtle
scala> forward(t);
res6: Turtle =
    Turtle(Point(1.0,0.0),Color(0,0,0),0.0,false)
```

- Linsen helfen, das besser zu organisieren.

Abhilfe mit Linsen

- Zuerst einmal: die Linse.

```
object Lenses {
  case class Lens[O, V](
    get: O ⇒ V,
    set: (O, V) ⇒ O
  ) }
```

- Linsen für die Schildkröte:

```
  val TurtlePosition =
    Lens[Turtle, Point](_.position,
      (t, p) ⇒ t.copy(position = p))

  val PointX =
    Lens[Point, Double](_.x,
     (p, x) ⇒ p.copy(x = x))
```

Benutzung

- Längliche Definition, aber einfache Benutzung:

```
scala> StandaloneTurtleLenses.TurtleX.get(t);
res12: Double = 0.0

scala> StandaloneTurtleLenses.TurtleX.set(t, 4.3);
res13: Turtles.Turtle =
    Turtle(Point(4.3,0.0),Color(0,0,0),0.0,false)
```

- Viel *boilerplate*, aber:

- Definition kann abgeleitet werden

Abgeleitete Linsen

- Aus der Shapeless-Bücherei:

```
object ShapelessTurtleLenses {

  import Turtles._
  import shapeless._, Lens._, Nat._

  val TurtleX = Lens[Turtle] >> _0 >> _0
  val TurtleHeading = Lens[Turtle] >> _2

  def right(t: Turtle, delta: Double) =
    TurtleHeading.modify(t)(_ + delta)
```

- Neue Linsen aus vorhandenen konstruieren

Linsen konstruieren

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

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

- Die Identitätslinse:

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

Linsen komponieren

- Gegeben Linsen $L_1 : S_1 \longleftrightarrow S_2, L_2 : S_2 \longleftrightarrow S_3$

- Die Komposition ist definiert als:

$$\begin{aligned} L_2 \cdot L_1 &: S_1 \longleftrightarrow S_3 \\ get &= get_2 \cdot get_1 \\ put(v, s) &= put_1(put_2(v, get_1(s)), s) \\ create &= create_1 \cdot create_2 \end{aligned}$$

Mehr Linsen und Bidirektionale Progammierung

- 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: Eventual Consistency

Reaktive Programmierung
Vorlesung 14 vom 30.06.15: Eventual Consistency

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte
  - Bidirektionale Programmierung: Zippers and Lenses
  - Eventual Consistency
  - Robustheit, Entwurfsmuster
  - Theorie der Nebenläufigkeit

Heute

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

Was ist eigentlich Konsistenz?

- Konsistenz = Widerspruchsfreiheit
- In der Logik:
  - Eine Formelmenge Γ 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 Zeitpunk 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 eingecheckt 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
- Meherere Nutzer können den Text verändern und sollen immer die neueste Version sehen.
- Siehe Google Docs, Etherpad und co.

Naive Methoden

- Ownership
  - Vor Änderungen: Lock-Anfrage an Server
  - Nur ein Nutzer kann gleichzeitig das Dokument ändern
  - Nachteile: Verzögerungen, Änderungen nur mit Netzverbindung
- Three-Way-Merge
  - Server führt nebenläufige Änderungen auf Grundlage eines gemeinsamen Ursprungs zusammen.
  - Requirement: *the chickens must stop moving so we can count them*

Conflict-Free Replicated Data Types

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

Zustandsbasierte CRDTs

- Konvergente replizierte Datentypen (CvRDTs)
- Knoten senden ihren gesamten Zustand an andere Knoten.
- Nur bestimmte Operationen auf dem Datentypen erlaubt (*update*).
- Eine kommutative, assoziative, idempotente *merge*-Funktion
  - Funktioniert gut mit Gossiping-Protokollen
  - Nachrichtenverlust unkritisch

CvRDT: Zähler

- Einfacher CvRDT
  - Zustand: $P \in \mathbb{N}$, Datentyp: $\mathbb{N}$

    $query(P) = P$

    $update(P, +, m) = P + m$

    $merge(P_1, P_2) = max(P_1, P_2)$

- Wert kann nur größer werden.

CvRDT: PN-Zähler

- Gängiges Konzept bei CRDTs: Komposition
- Aus zwei Zählern kann ein komplexerer Typ zusammengesetzt werden:
  - Zähler P (Positive) und Zähler N (Negative)
  - Zustand: $(P, N) \in \mathbb{N} \times \mathbb{N}$, Datentyp: $\mathbb{Z}$

    $query((P, N)) = query(P) - query(N)$

    $update((P, N), +, m) = (update(P, +, m), N)$

    $update((P, N), -, m) = (P, update(N, +, m))$

    $merge((P_1, N_1), (P_2, N_2)) = (merge(P_1, P_2), merge(N_1, N_2))$

CvRDT: Mengen

- Ein weiterer 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 wieder ein komplexerer Typ entstehen.
  - Menge P (Hinzugefügte Elemente) und Menge N (Gelöschte Elemente)
  - Zustand: $(P, N) \in \mathcal{P}(A) \times \mathcal{P}(A)$, Datentyp: $\mathcal{P}(A)$

    $query((P, N)) = query(P) \setminus query(N)$

    $update((P, N), +, m) = (update(P, +, m), N)$

    $update((P, N), -, m) = (P, update(N, +, m))$

    $merge((P_1, N_1), (P_2, N_2)) = (merge(P_1, P_2), merge(N_1, N_2))$

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: $(x, t) \in X \times \mathit{timestamp}$
- $\mathit{query}((x, t)) = x$
- $\mathit{update}(=, x')$
  - lokal: $(x, t) := (x', \mathit{now}())$
  - extern: $\mathit{if}\ \ t < t'\ \ \mathit{then}\ (x, t) := (x', t')$

Vektor-Uhren

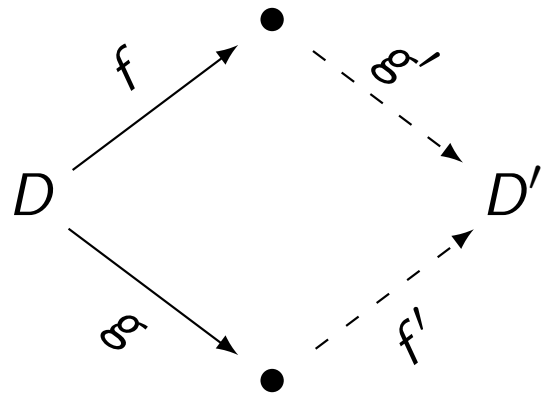
- Im LWW Register benötigen wir Timestamps
  - Kausalität muss erhalten bleiben
  - Timestamps müssen eine total Ordnung haben
- Datum und Uhrzeit ungeeignet
- Lösung: Vektor-Uhren
  - Jeder Knoten hat einen Zähler, der bei Operationen hochgesetzt wird
  - Zusätzlich merkt sich jeder Knoten den aktuellsten Zählerwert, den er bei den anderen Knoten beobachtet hat.

Operational Transformation

- Die CRDTs die wir bis jetzt kennengelernt haben sind recht einfach
- Das Texteditor Beispiel ist damit noch nicht umsetzbar
- Kommutative Operationen auf einer Sequenz von Buchstaben?
  - Einfügen möglich (totale Ordnung durch Vektoruhren)
  - Wie Löschen?

Operational Transformation

- Idee: Nicht-kommutative Operationen transformieren



- Für *transform* muss gelten:

$$transform\ f\ g = \langle f', g' \rangle \Longrightarrow g' \circ f = f' \circ g$$
$$applyOp\ (g \circ f)\ D = applyOp\ g\ (applyOp\ f\ D)$$

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen:

- *Retain*— Buchstaben beibehalten
- *Delete*— Buchstaben löschen
- *Insert c* — Buchstaben *c* einfügen

Eine Operation ist eine Sequenz von Aktionen

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen:

- *Retain*— Buchstaben beibehalten
- *Delete*— Buchstaben löschen
- *Insert c* — Buchstaben *c* einfügen

Eine Operation ist eine Sequenz von Aktionen

Ein Beispiel:

Eingabe: `R 1 P 5`
Ausgabe:
Aktionen:

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen:

- *Retain*— Buchstaben beibehalten
- *Delete*— Buchstaben löschen
- *Insert c* — Buchstaben *c* einfügen

Eine Operation ist eine Sequenz von Aktionen

Ein Beispiel:

Eingabe: `1 P 5`
Ausgabe: `R`
Aktionen: *Retain*,

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen:

- *Retain*— Buchstaben beibehalten
- *Delete*— Buchstaben löschen
- *Insert c* — Buchstaben *c* einfügen

Eine Operation ist eine Sequenz von Aktionen

Ein Beispiel:

Eingabe:	`P 5`
Ausgabe:	`R`
Aktionen:	*Retain*, *Delete*,

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen:

- *Retain*— Buchstaben beibehalten
- *Delete*— Buchstaben löschen
- *Insert c* — Buchstaben *c* einfügen

Eine Operation ist eine Sequenz von Aktionen

Ein Beispiel:

Eingabe:	`5`
Ausgabe:	`R P`
Aktionen:	*Retain*, *Delete*, *Retain*,

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen:

- *Retain*— Buchstaben beibehalten
- *Delete*— Buchstaben löschen
- *Insert* c — Buchstaben c einfügen

Eine Operation ist eine Sequenz von Aktionen

Ein Beispiel:

Eingabe:	`5`
Ausgabe:	`R P 1`
Aktionen:	*Retain*, *Delete*, *Retain*, *Insert* `1`,

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen:

- *Retain*— Buchstaben beibehalten
- *Delete*— Buchstaben löschen
- *Insert* c — Buchstaben c einfügen

Eine Operation ist eine Sequenz von Aktionen

Ein Beispiel:

Eingabe:	
Ausgabe:	`R P 1 5`
Aktionen:	*Retain*, *Delete*, *Retain*, *Insert* `1`, *Retain*.

Operationen für Text

Operationen bestehen aus drei Arten von Aktionen:

- *Retain*— Buchstaben beibehalten
- *Delete*— Buchstaben löschen
- *Insert* c — Buchstaben c einfügen

Eine Operation ist eine Sequenz von Aktionen

Ein Beispiel:

Eingabe:
Ausgabe: `R P 1 5`
Aktionen: *Retain*, *Delete*, *Retain*, *Insert* `1`, *Retain*.

- Operationen sind partiell.

Operationen Komponieren

- Komposition: Fallunterscheidung auf der Aktion
  - Keine einfache Konkatenation!
- Beispiel:

$$\begin{aligned} p &= [\mathit{Delete}, \mathit{Insert}\ \texttt{X}, \mathit{Retain}] \\ q &= [\mathit{Retain}, \mathit{Insert}\ \texttt{Y}, \mathit{Delete}] \\ \mathit{compose}\ p\ q &= \end{aligned}$$

- *compose* ist partiell.
- Äquivalenz von Operationen:

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

Operationen Komponieren

- Komposition: Fallunterscheidung auf der Aktion
  - Keine einfache Konkatenation!
- Beispiel:

$$\begin{aligned} p &= [\mathit{Insert}\ \mathtt{X}, \mathit{Retain}] \\ q &= [\mathit{Retain}, \mathit{Insert}\ \mathtt{Y}, \mathit{Delete}] \\ \mathit{compose}\ p\ q &= [\mathit{Delete}, \end{aligned}$$

- *compose* ist partiell.
- Äquivalenz von Operationen:

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

Operationen Komponieren

- Komposition: Fallunterscheidung auf der Aktion
  - Keine einfache Konkatenation!
- Beispiel:

$$\begin{aligned} p &= [\mathit{Retain}] \\ q &= [\mathit{Insert}\ \mathtt{Y}, \mathit{Delete}] \\ \mathit{compose}\ p\ q &= [\mathit{Delete}, \mathit{Insert}\ \mathtt{X},] \end{aligned}$$

- *compose* ist partiell.
- Äquivalenz von Operationen:

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

Operationen Komponieren

- Komposition: Fallunterscheidung auf der Aktion
  - Keine einfache Konkatenation!
- Beispiel:

$$\begin{aligned} p &= [\mathit{Retain}] \\ q &= [\mathit{Delete}] \\ \mathit{compose}\ p\ q &= [\mathit{Delete}, \mathit{Insert}\ \texttt{X}, \mathit{Insert}\ \texttt{Y}, \end{aligned}$$

- *compose* ist partiell.
- Äquivalenz von Operationen:

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

Operationen Komponieren

- Komposition: Fallunterscheidung auf der Aktion
  - Keine einfache Konkatenation!
- Beispiel:

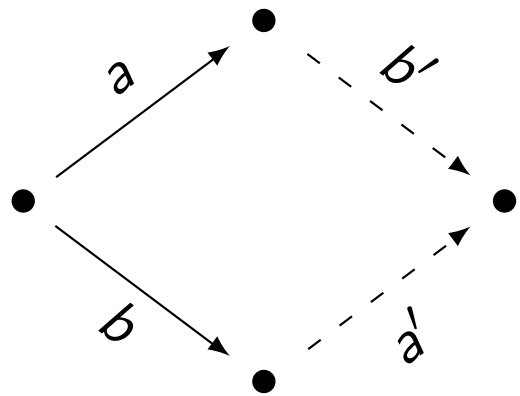
$$\begin{aligned} p &= [] \\ q &= [] \\ \mathit{compose}\ p\ q &= [\mathit{Delete}, \mathit{Insert}\ \texttt{X}, \mathit{Insert}\ \texttt{Y}, \mathit{Delete}] \end{aligned}$$

- *compose* ist partiell.
- Äquivalenz von Operationen:

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

Operationen Transformieren

- Transformation

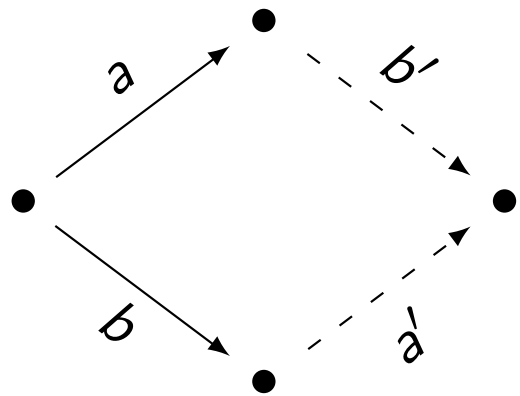


- Beispiel:

$$
\begin{aligned}
a &= [\mathit{Insert}\ \mathtt{X}, \mathit{Retain}, \mathit{Delete}] \\
b &= [\mathit{Delete}, \mathit{Retain}, \mathit{Insert}\ \mathtt{Y}] \\
\mathit{transform}\ a\ b &= ([\\
&\quad\ , [\\
&\quad\ )]
\end{aligned}
$$

Operationen Transformieren

- Transformation

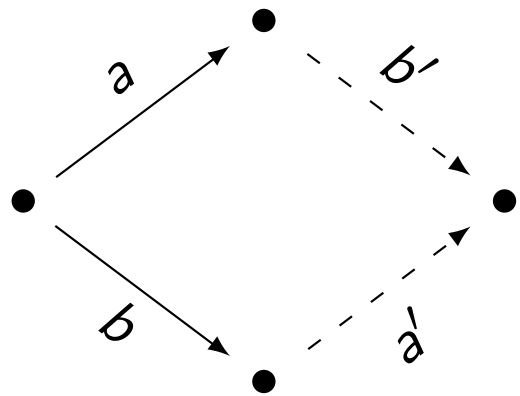


- Beispiel:

$$
\begin{aligned}
a &= [\mathit{Retain}, \mathit{Delete}] \\
b &= [\mathit{Delete}, \mathit{Retain}, \mathit{Insert}\ \mathtt{Y}] \\
\mathit{transform}\ a\ b &= ([\mathit{Insert}\ \mathtt{X}, \\
&\quad , [\mathit{Retain}, \\
&\quad)
\end{aligned}
$$

Operationen Transformieren

- Transformation

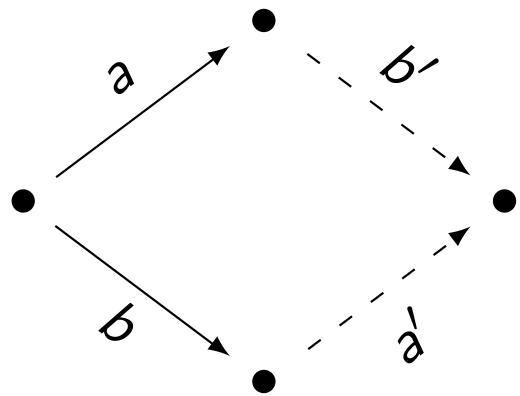


- Beispiel:

$$
\begin{aligned}
a &= [\mathit{Delete}] \\
b &= [\mathit{Retain}, \mathit{Insert}\ \mathtt{Y}] \\
\mathit{transform}\ a\ b &= ([\mathit{Insert}\ \mathtt{X}, \mathit{Delete}, \\
&\quad , [\mathit{Retain}, \\
&\quad)
\end{aligned}
$$

Operationen Transformieren

- Transformation

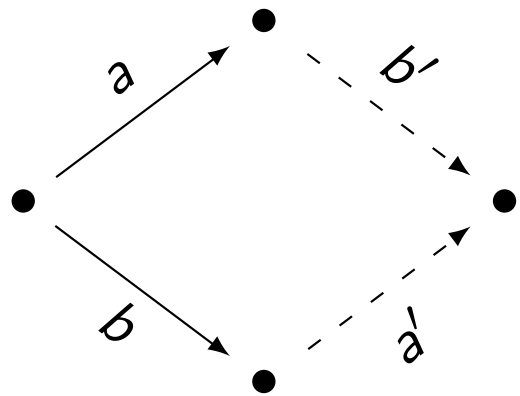


- Beispiel:

$$
\begin{aligned}
a &= [] \\
b &= [\mathit{Insert}\ \mathtt{Y}] \\
\mathit{transform}\ a\ b &= ([\mathit{Insert}\ \mathtt{X}, \mathit{Delete}, \\
&\quad , [\mathit{Retain}, \mathit{Delete}, \\
&\quad)
\end{aligned}
$$

Operationen Transformieren

- Transformation

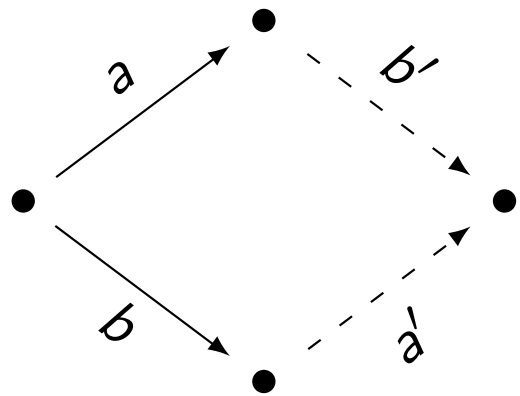


- Beispiel:

$$
\begin{aligned}
a &= [] \\
b &= [] \\
transform\ a\ b &= ([Insert\ \texttt{X}, Delete, Retain \\
&\quad , [Retain, Delete, Insert\ \texttt{Y} \\
&\quad)
\end{aligned}
$$

Operationen Transformieren

- Transformation

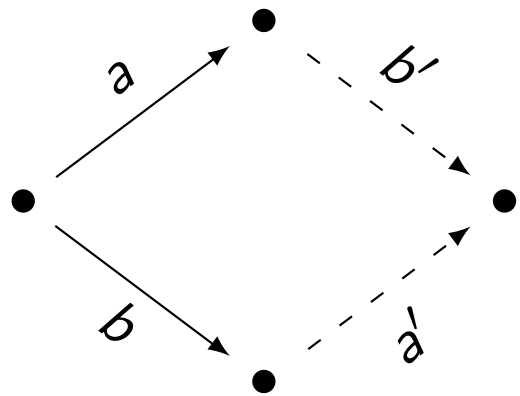


- Beispiel:

$$
\begin{aligned}
a &= [] \\
b &= [] \\
\mathit{transform}\ a\ b &= ([\mathit{Insert}\ \mathtt{X}, \mathit{Delete}, \mathit{Retain}] \\
&\quad , [\mathit{Retain}, \mathit{Delete}, \mathit{Insert}\ \mathtt{Y}] \\
&\quad)
\end{aligned}
$$

Operationen Transformieren

- Transformation



- Beispiel:

$$
\begin{aligned}
a &= [\mathit{Insert}\ \texttt{X}, \mathit{Retain}, \mathit{Delete}] \\
b &= [\mathit{Delete}, \mathit{Retain}, \mathit{Insert}\ \texttt{Y}] \\
\mathit{transform}\ a\ b &= ([\mathit{Insert}\ \texttt{X}, \mathit{Delete}, \mathit{Retain}] \\
&\quad , [\mathit{Retain}, \mathit{Delete}, \mathit{Insert}\ \texttt{Y}] \\
&\quad)
\end{aligned}
$$

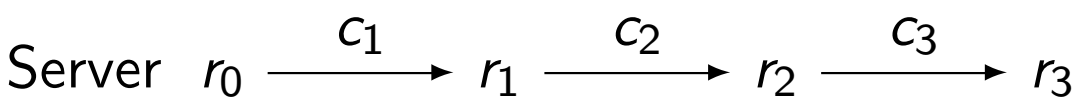
Operationen Verteilen

- 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 sequentialisieren
  - Transformierte Operationen verteilen

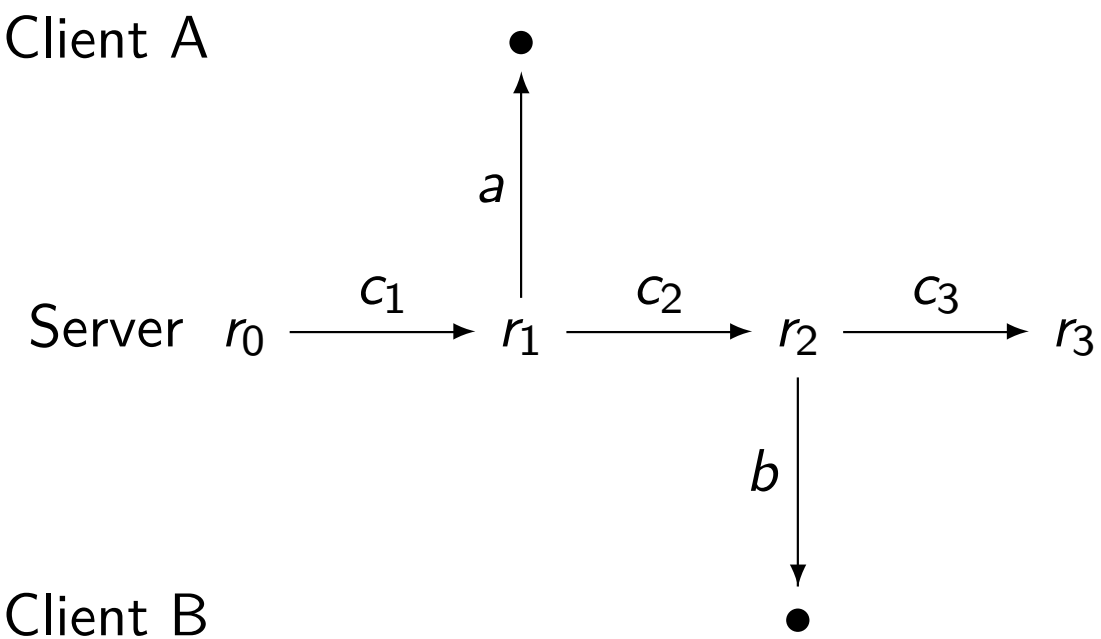
Client A



Client B

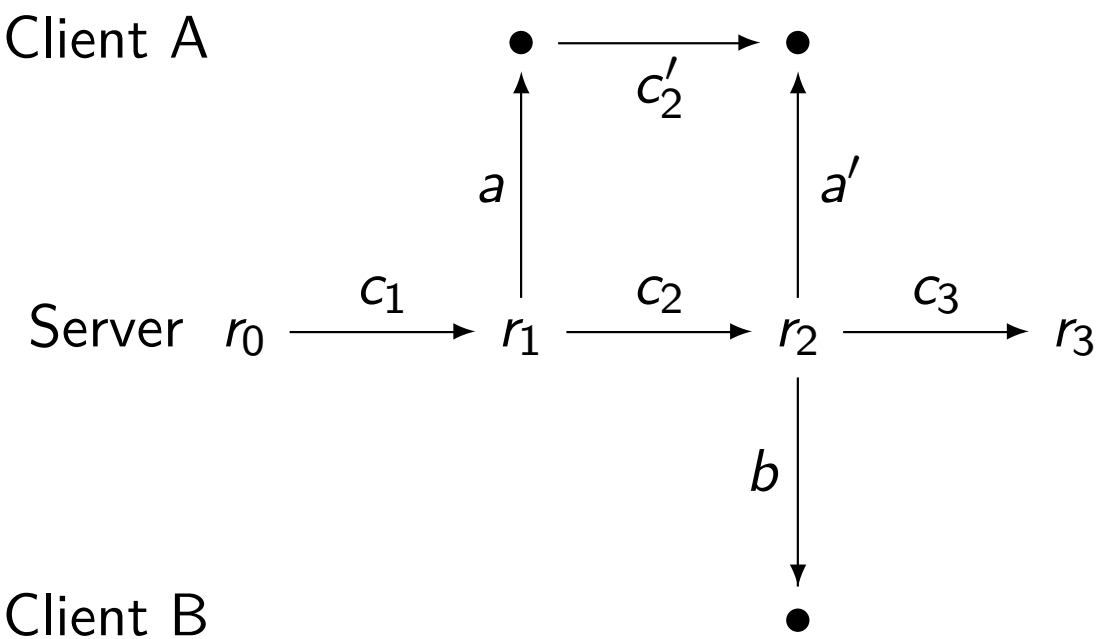
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



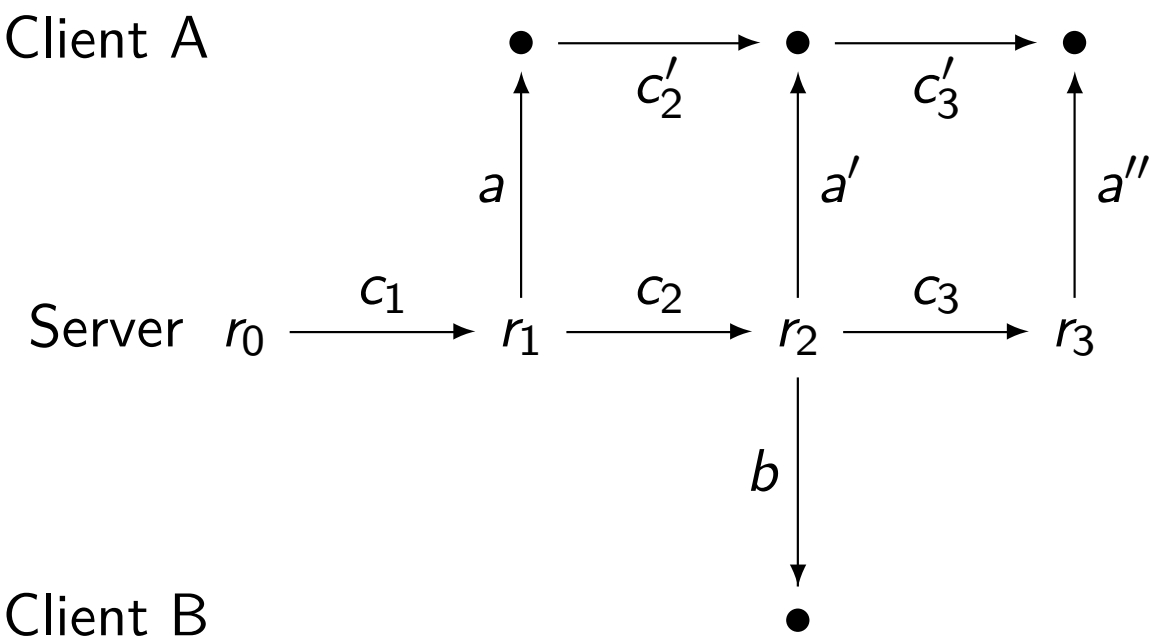
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



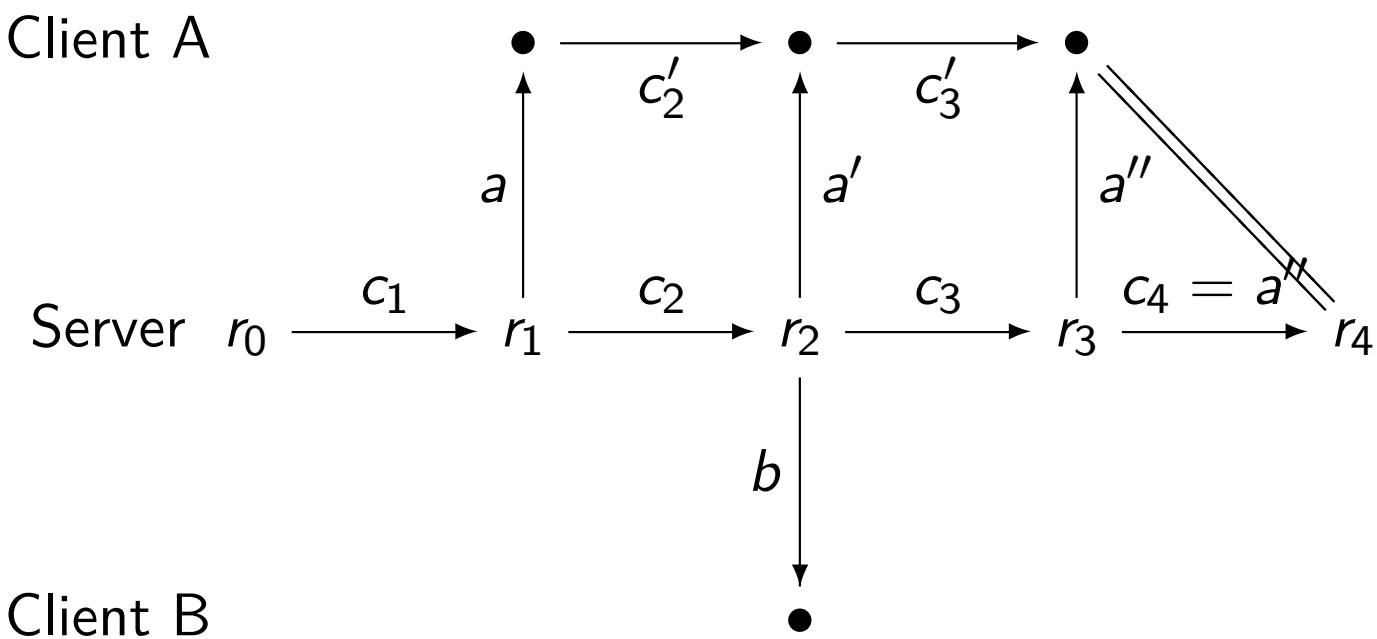
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



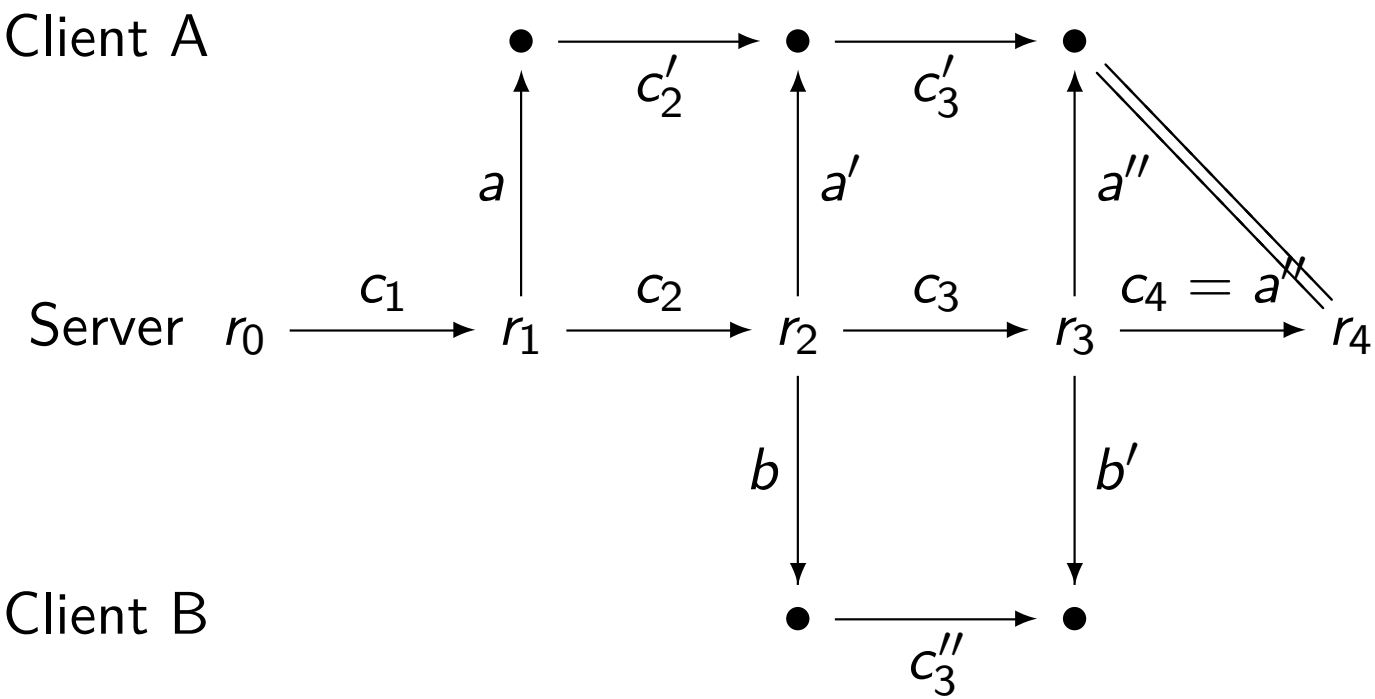
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



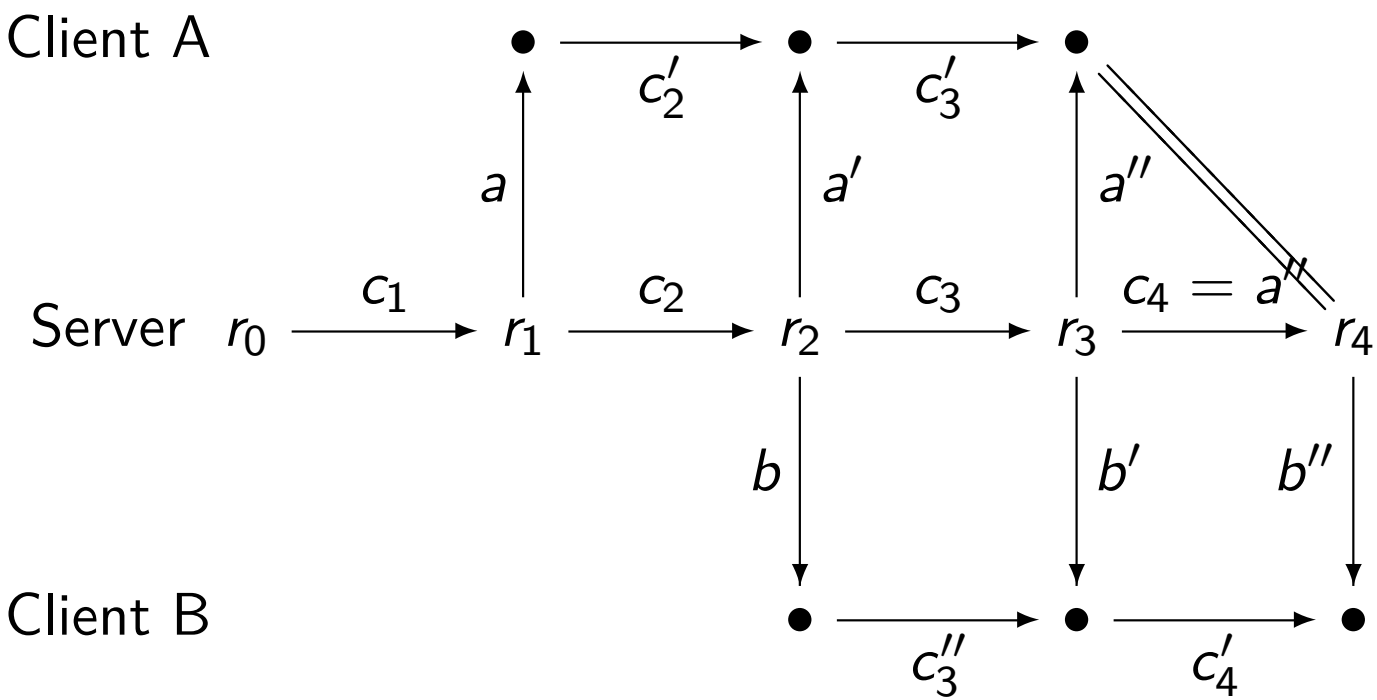
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



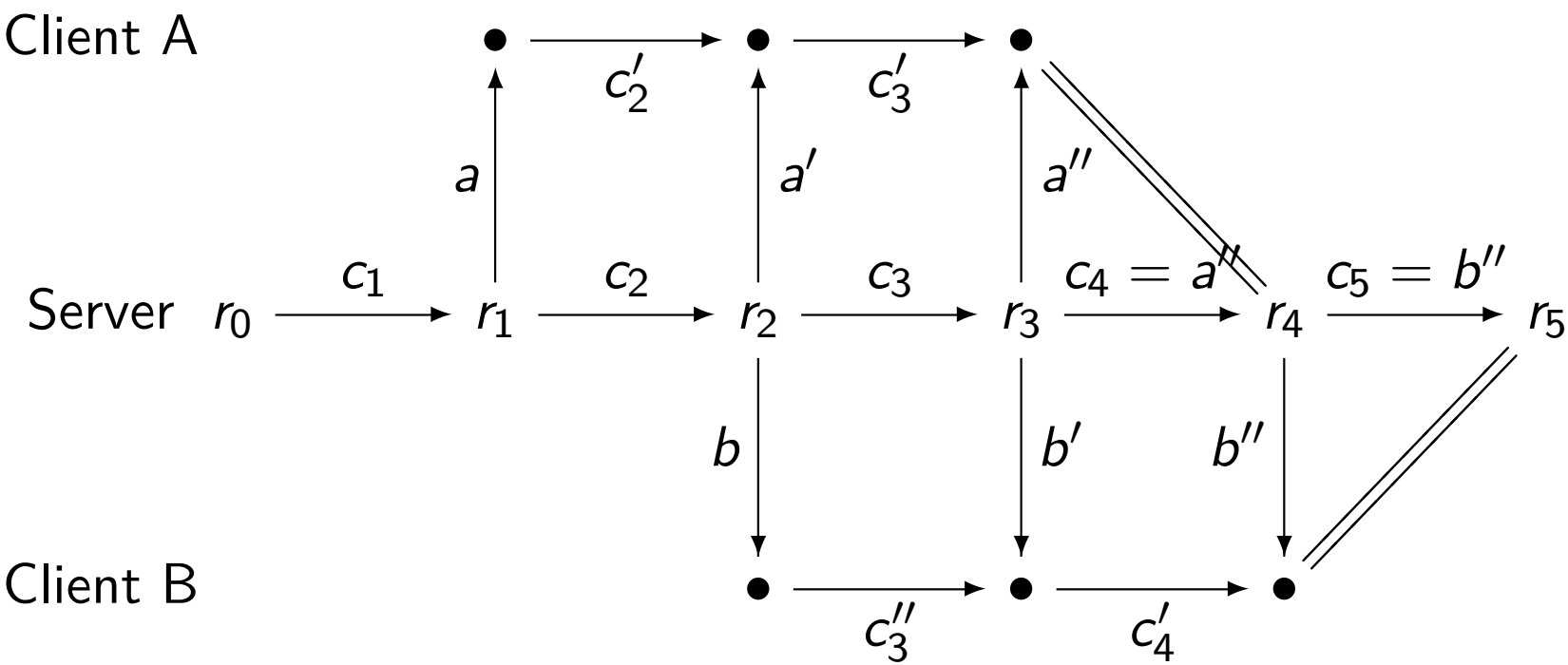
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen

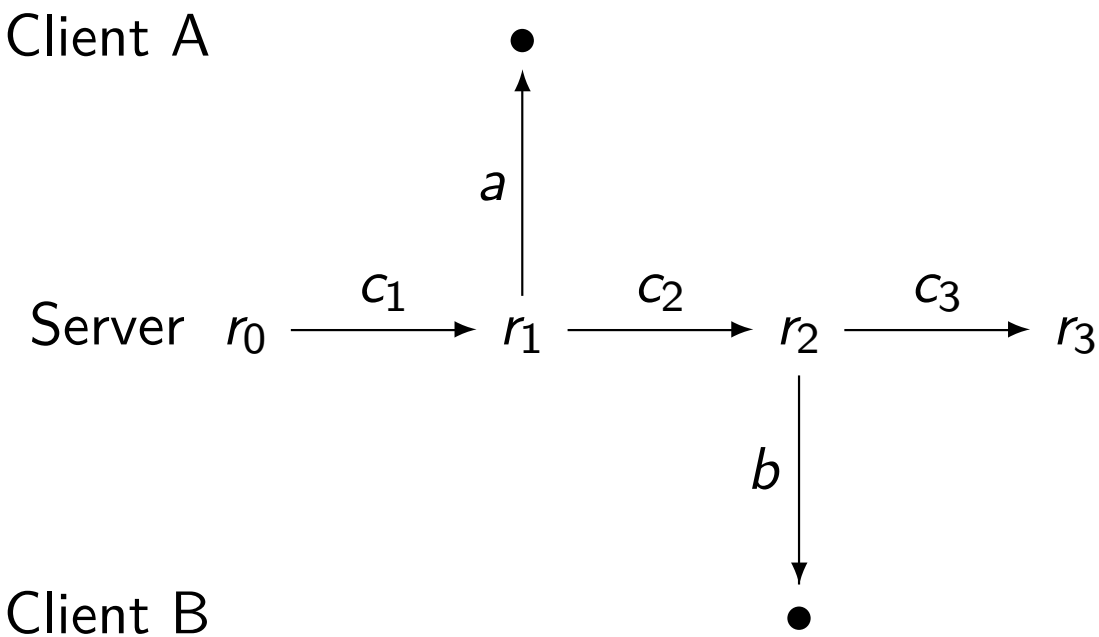
Client A

Server $r_0 \xrightarrow{c_1} r_1 \xrightarrow{c_2} r_2 \xrightarrow{c_3} r_3$

Client B

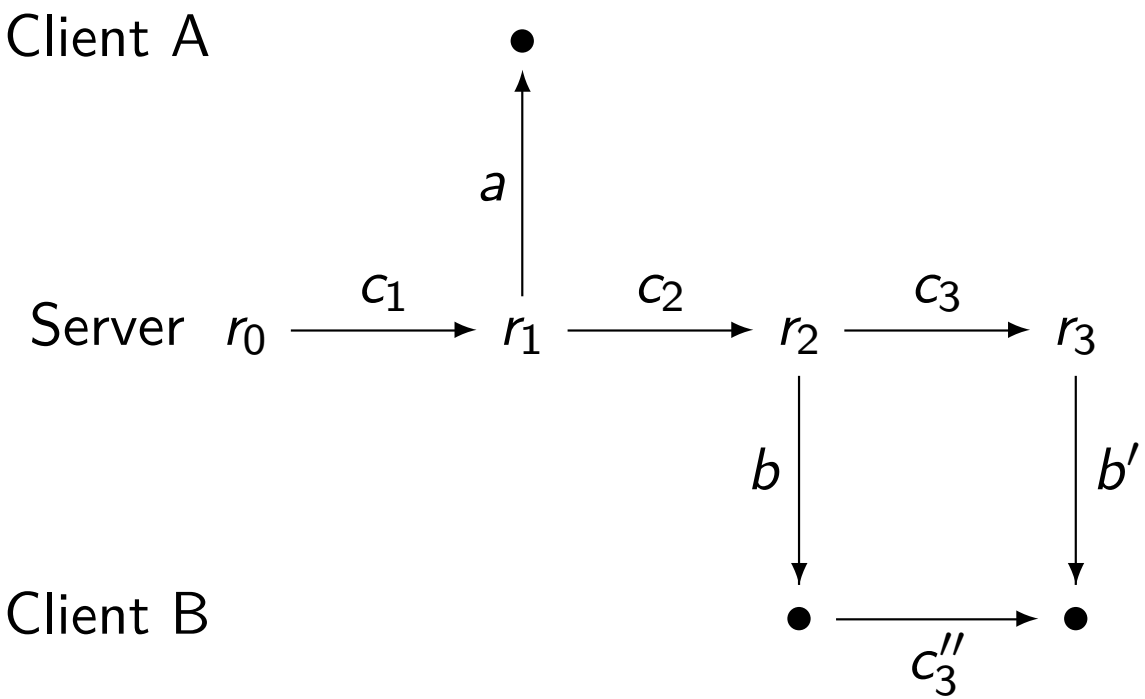
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



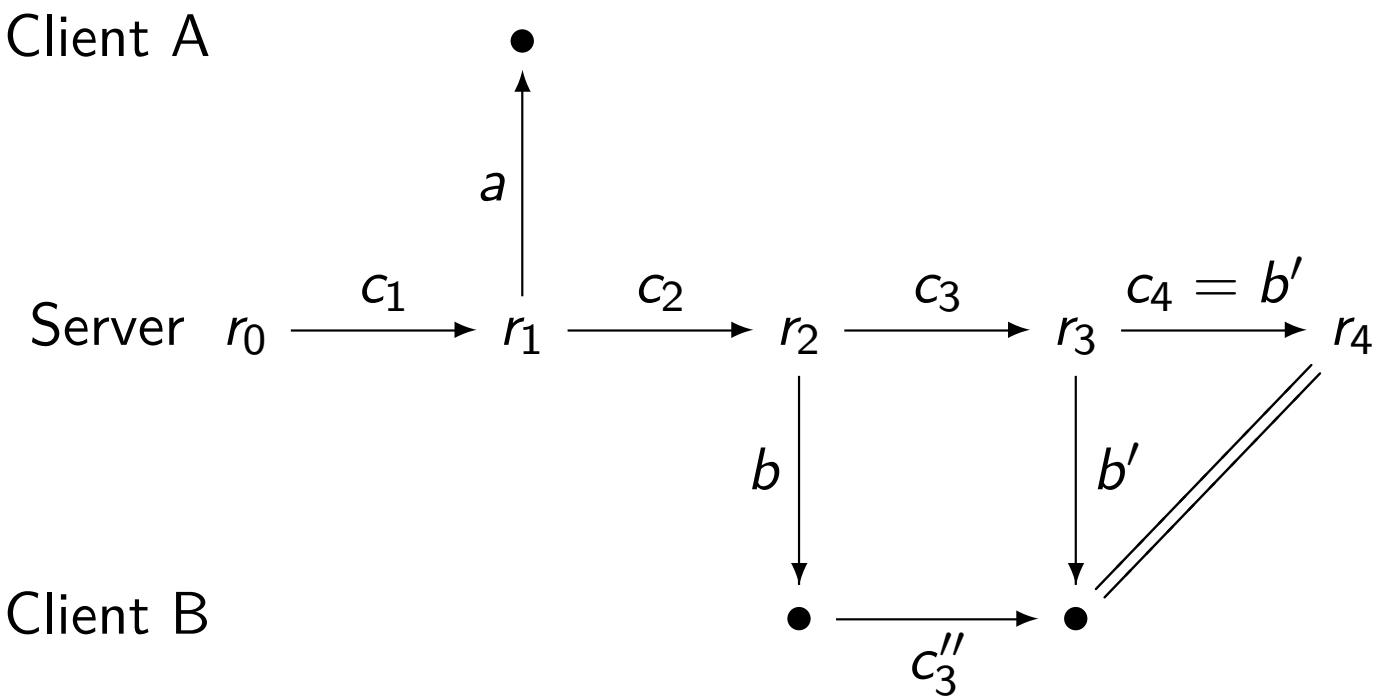
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



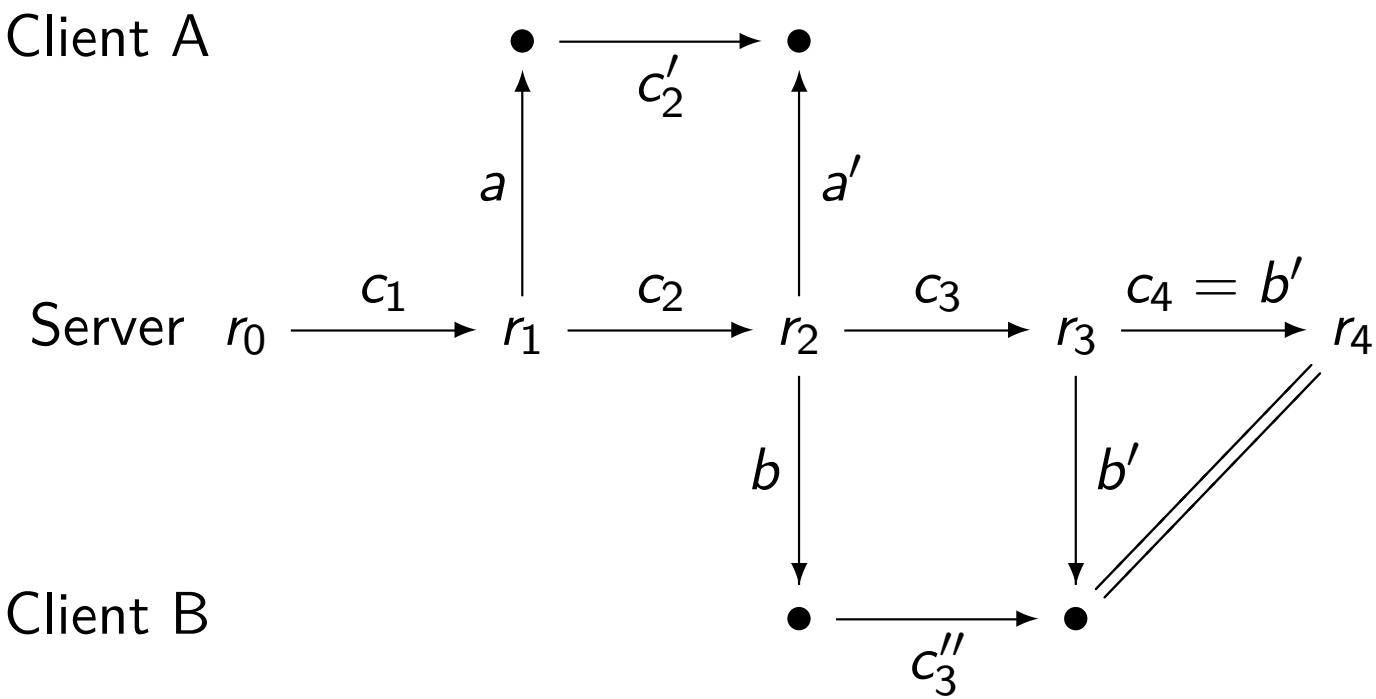
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



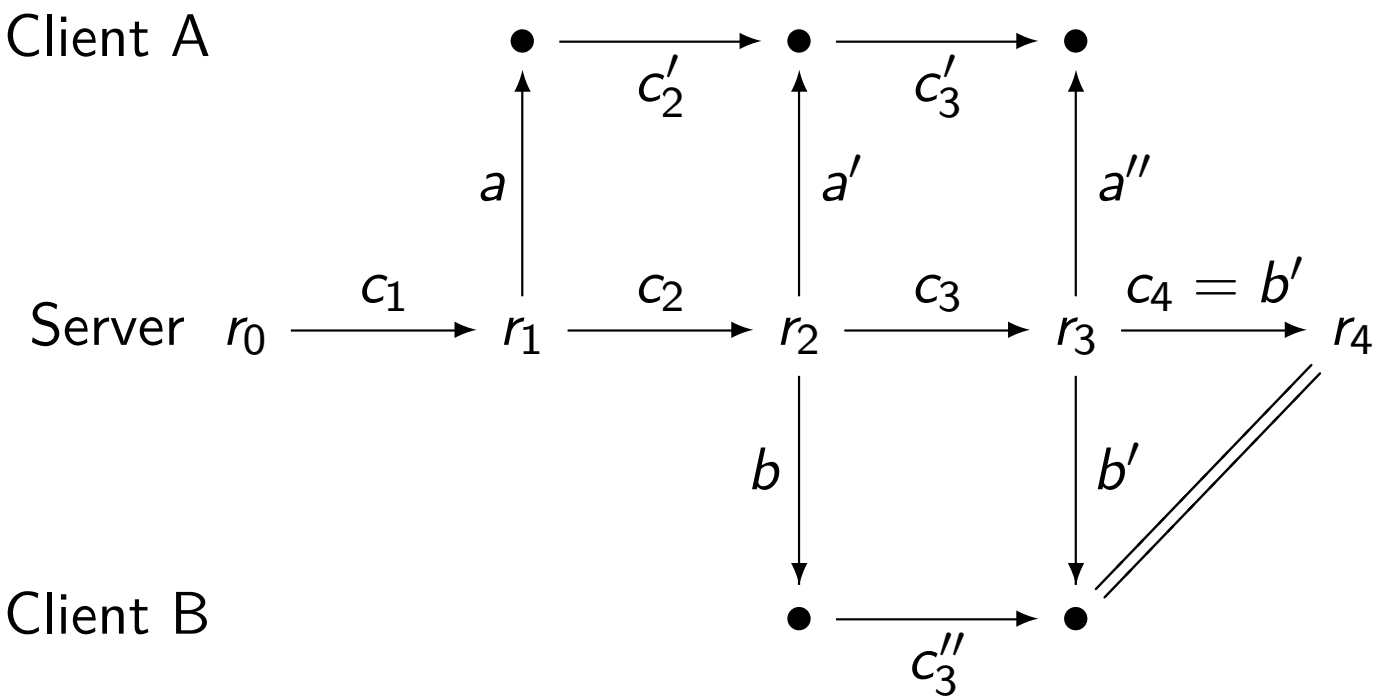
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



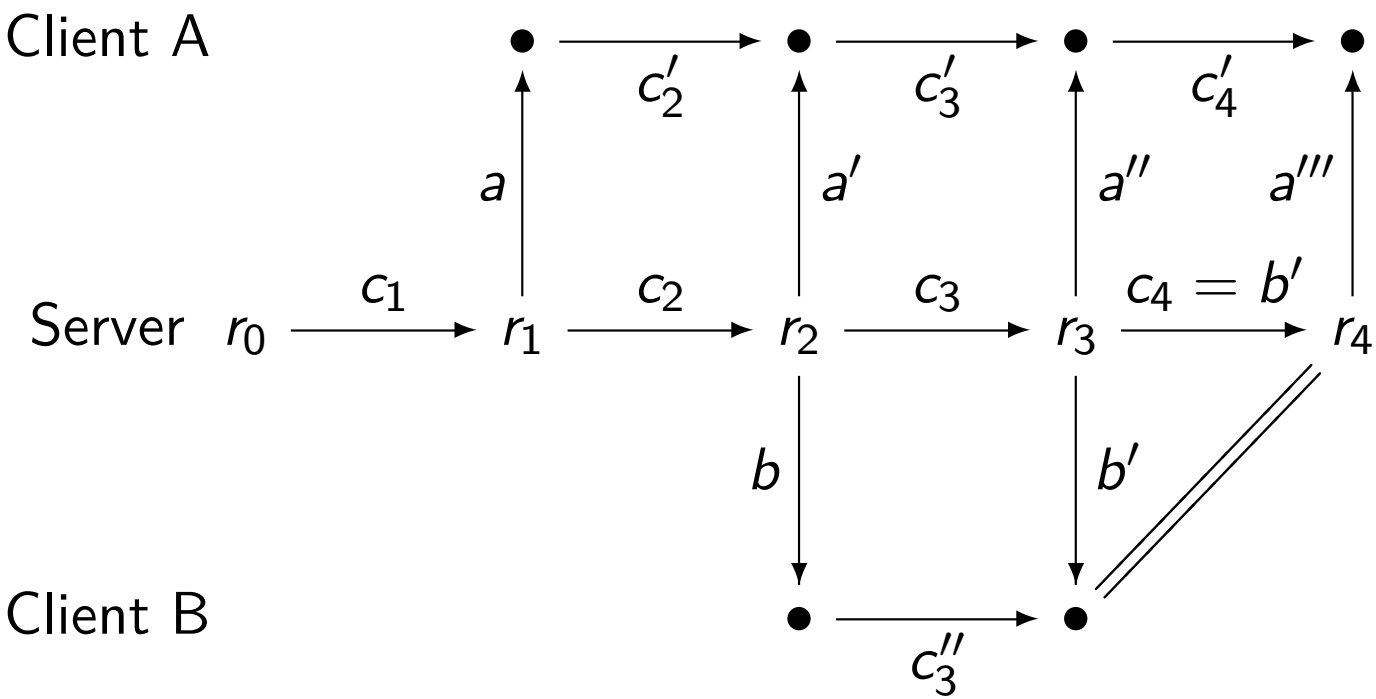
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



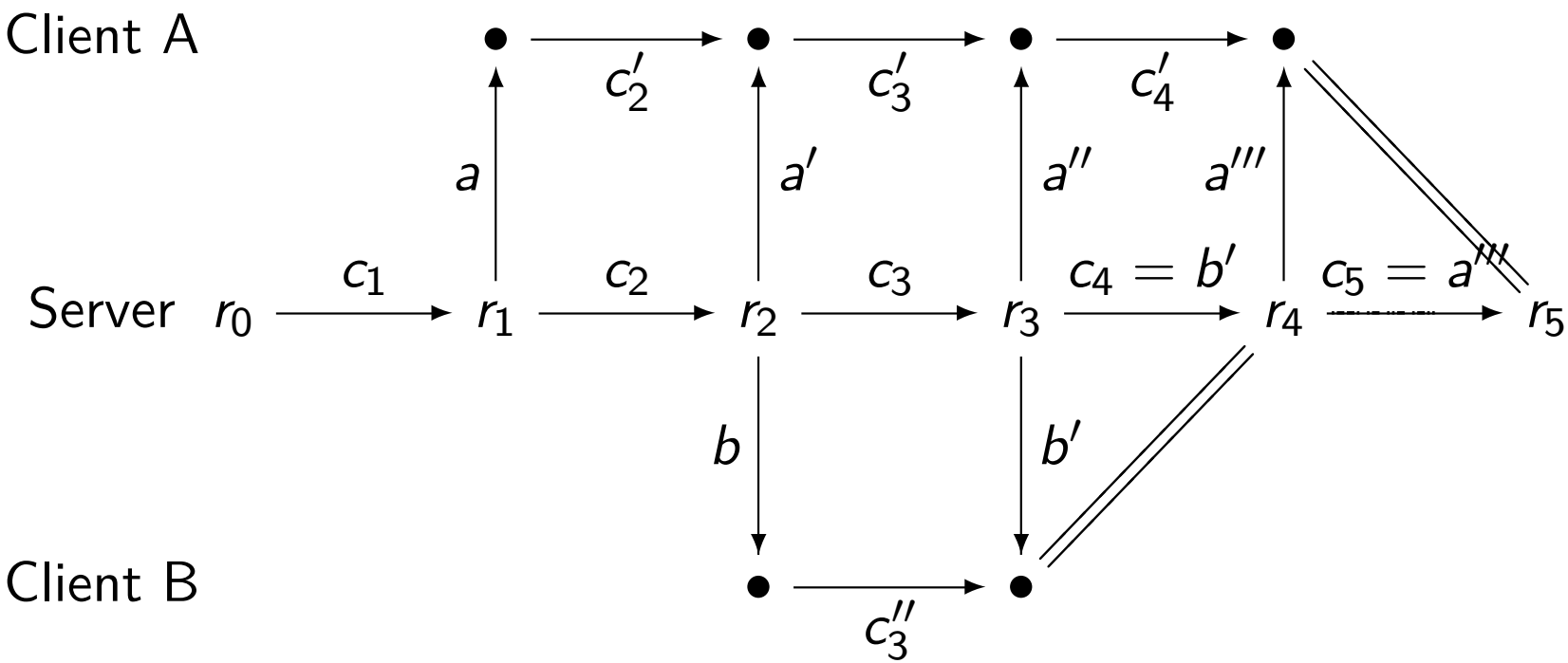
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



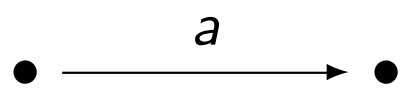
Der Server

- Zweck:
  - Nebenläufige Operationen sequentialisieren
  - Transformierte Operationen verteilen



Der Client

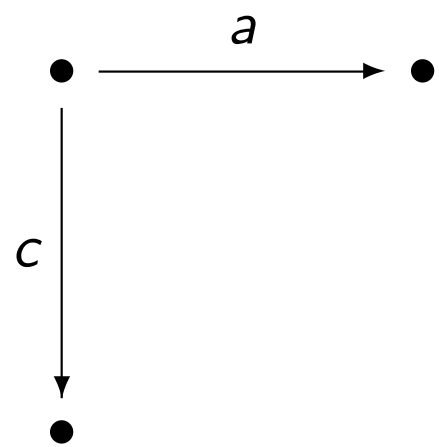
- Zweck: Operationen Puffern während eine Bestätigung aussteht



Revision r

Der Client

- Zweck: Operationen Puffern während eine Bestätigung aussteht

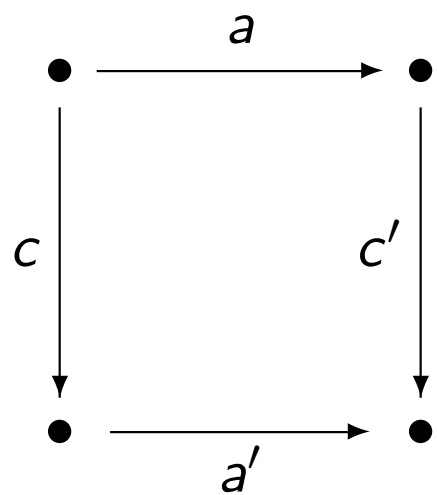


Revision r

Revision $r + 1$

Der Client

- Zweck: Operationen Puffern während eine Bestätigung aussteht

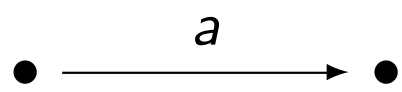


Revision r

Revision $r + 1$

Der Client

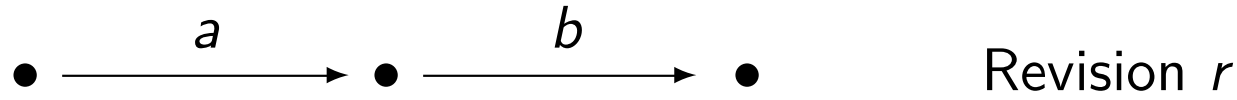
- Zweck: Operationen Puffern während eine Bestätigung aussteht



Revision r

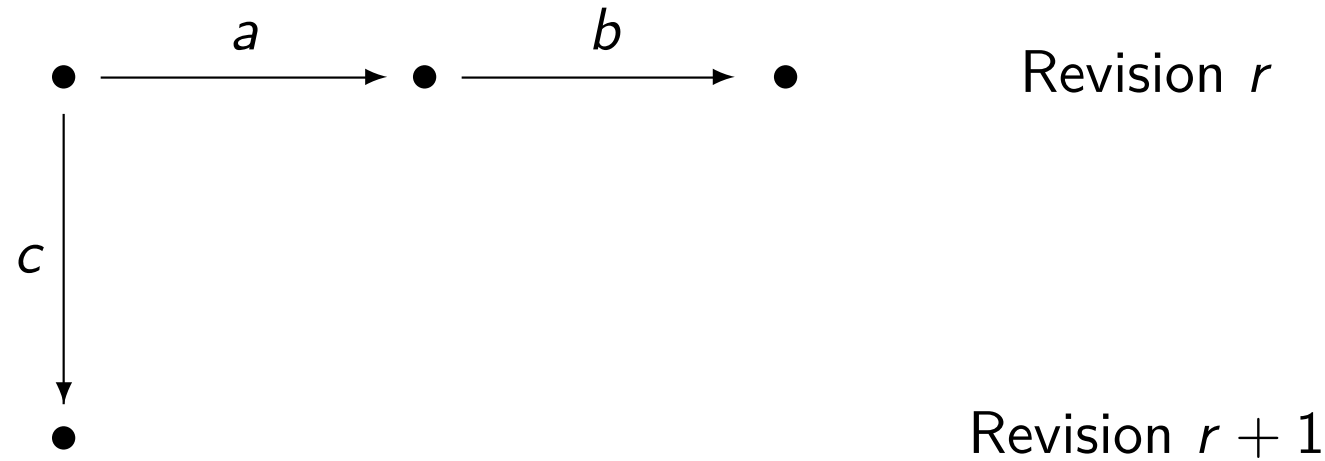
Der Client

- Zweck: Operationen Puffern während eine Bestätigung aussteht



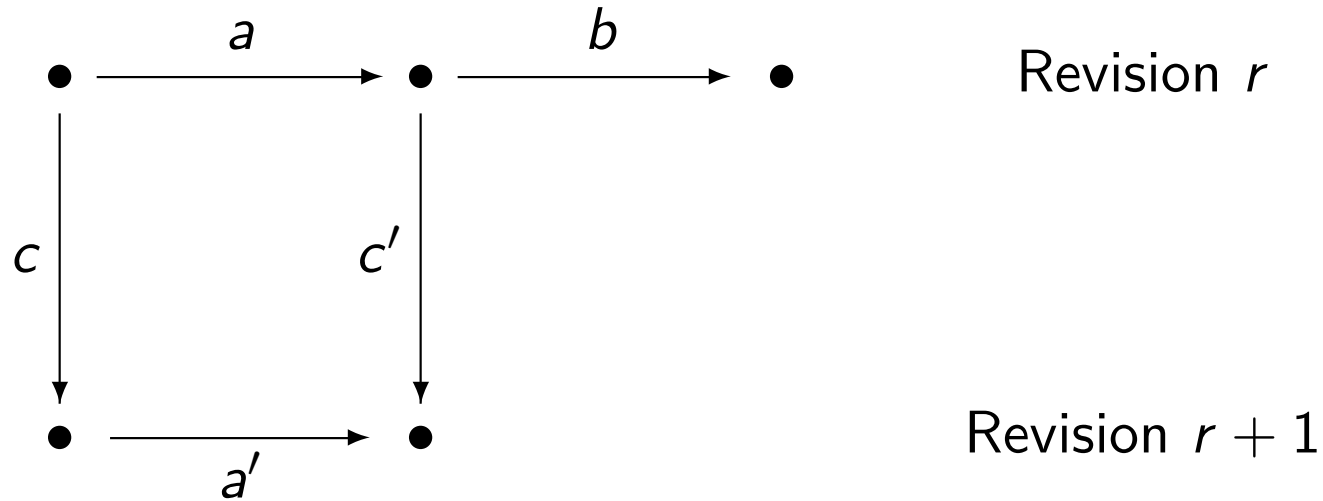
Der Client

- Zweck: Operationen Puffern während eine Bestätigung aussteht



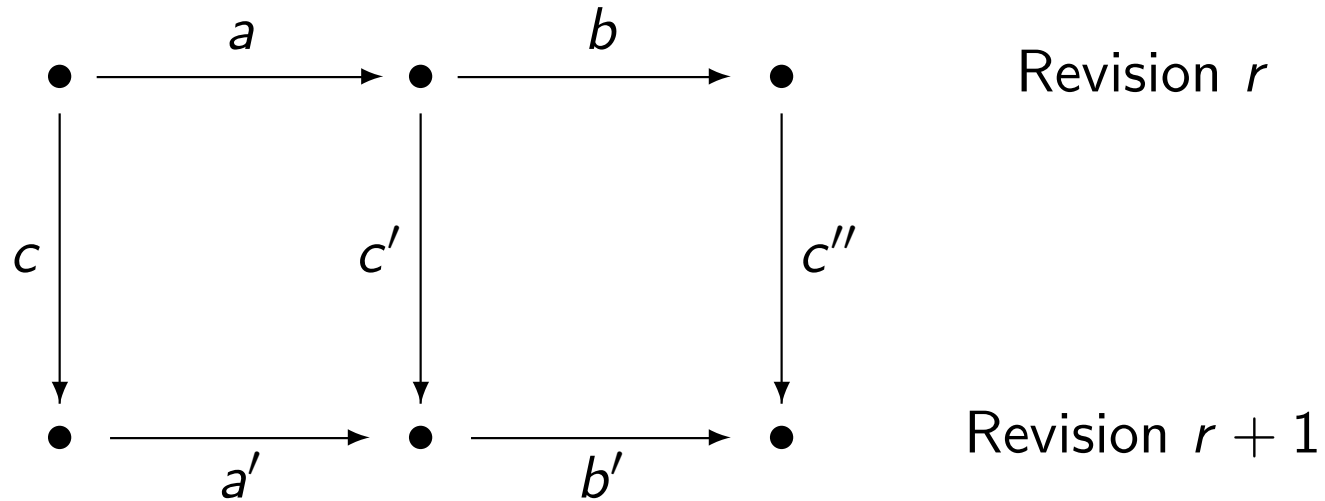
Der Client

- Zweck: Operationen Puffern während eine Bestätigung aussteht



Der Client

- Zweck: Operationen Puffern während eine Bestätigung aussteht



Zusammenfassung

- Strikte Konsistenz in verteilten Systemen nicht erreichbar
- Strong Eventual Consistency
  - Wenn längere Zeit keine Änderungen stattgefunden haben befinden sich schließlich alle Knoten im gleichen Zustand.
  - Wenn zwei Knoten die gleiche Menge Updates beobachten befinden sie sich im gleichen Zustand.
- Conflict-Free replicated Data Types:
  - Zustandsbasiert: CvRDTs
  - Operationsbasiert: CmRDTs
- Operational Transformation
  - Strong Eventual Consistency auch ohne kommutative Operationen

Reaktive Programmierung
Vorlesung 15 vom 06.07.15: Robustheit und Entwurfsmuster

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte
  - Bidirektionale Programmierung: Zippers and Lenses
  - Eventual Consistency
  - Robustheit, Entwurfsmuster
  - Theorie der Nebenläufigkeit

Organisatorisches

Die nächste Übung (9. Juli) fällt aus

Fragen zum Übungblatt gerne während und nach der Vorlesung

Rückblick: Konsistenz

- Strikte Konsistenz in verteilten Systemen nicht erreichbar
- Strong Eventual Consistency
  - Wenn längere Zeit keine Änderungen stattgefunden haben befinden sich schließlich alle Knoten im gleichen Zustand.
  - Wenn zwei Knoten die gleiche Menge Updates beobachten befinden sie sich im gleichen Zustand.
- Conflict-Free replicated Data Types:
  - Zustandsbasiert: CvRDTs
  - Operationsbasiert: CmRDTs
- Operational Transformation
  - Strong Eventual Consistency auch ohne kommutative Operationen

Robustheit in verteilten Systemen

Lokal:

- Nachrichten gehen nicht verloren
- Aktoren können abstürzen - Lösung: Supervisor

Verteilt:

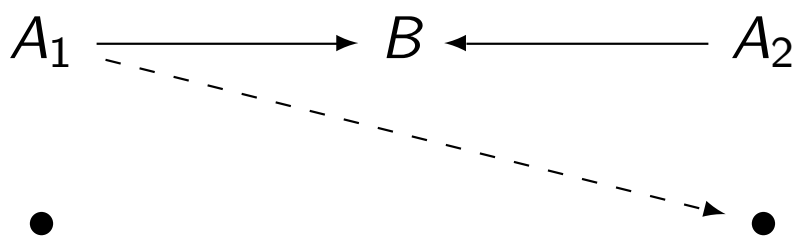
- Nachrichten können verloren gehen
- Teilsysteme können abstürzen
  - Hardware-Fehler
  - Stromausfall
  - Geplanter Reboot (Updates)
  - Naturkatastrophen / Höhere Gewalt
  - Software-Fehler

Zwei-Armeen-Problem

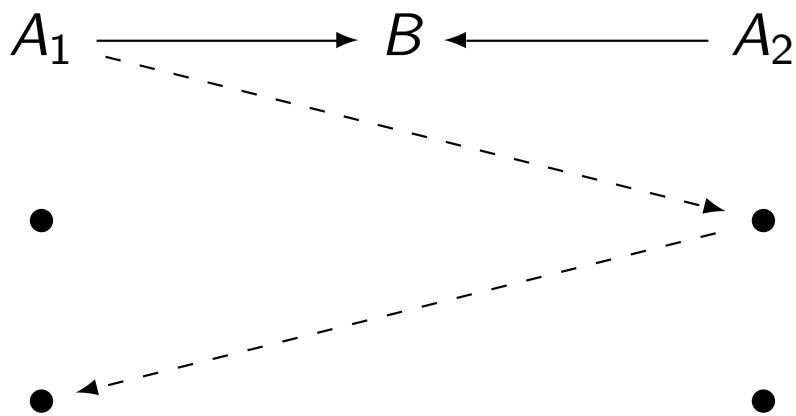
- Zwei Armeen A_1 und A_2 sind jeweils zu klein um gegen den Feind B zu gewinnen.
- Daher wollen sie sich über einen Angriffszeitpunkt absprechen.

Zwei-Armeen-Problem

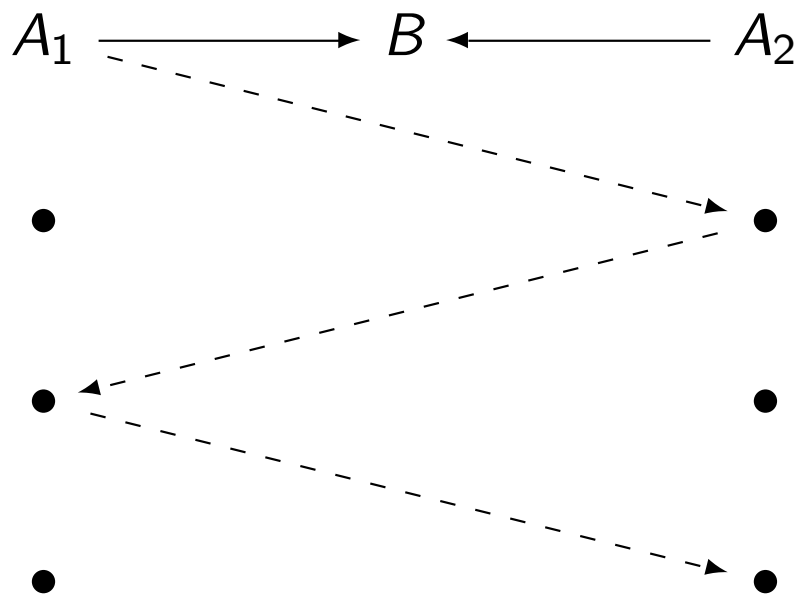
Zwei-Armeen-Problem



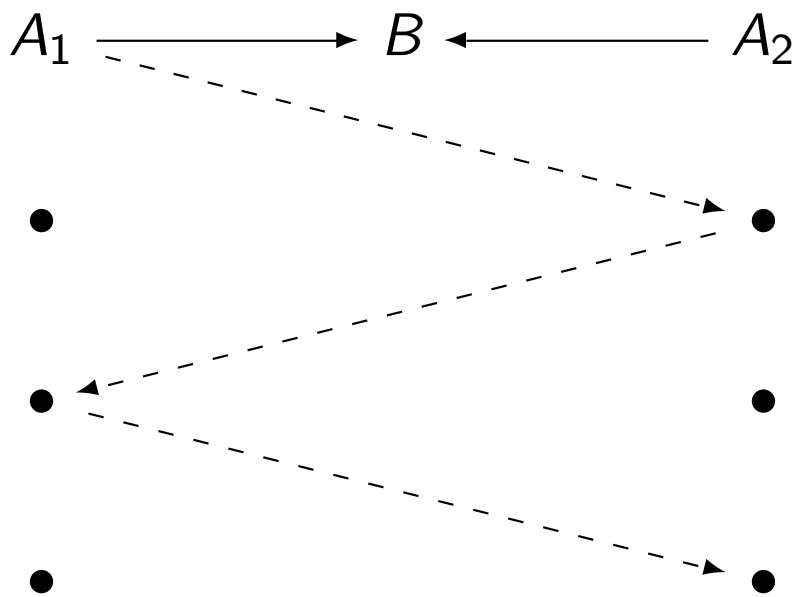
Zwei-Armeen-Problem



Zwei-Armeen-Problem



Zwei-Armeen-Problem



- Unlösbar – Wir müssen damit leben!

Unsichere Kanäle

- Unsichere Kanäle sind ein generelles Problem der Netzwerktechnik
- Lösungsstrategien:
  - Redundanz – Nachrichten mehrfach schicken
  - Indizierung – Nachrichten numerieren
  - Timeouts – Nicht ewig auf Antwort warten
  - Heartbeats – Regelmäßige „Lebenszeichen“
- Beispiel: TCP
  - Drei-Wege Handschlag
  - Indizierte Pakete

Gossipping

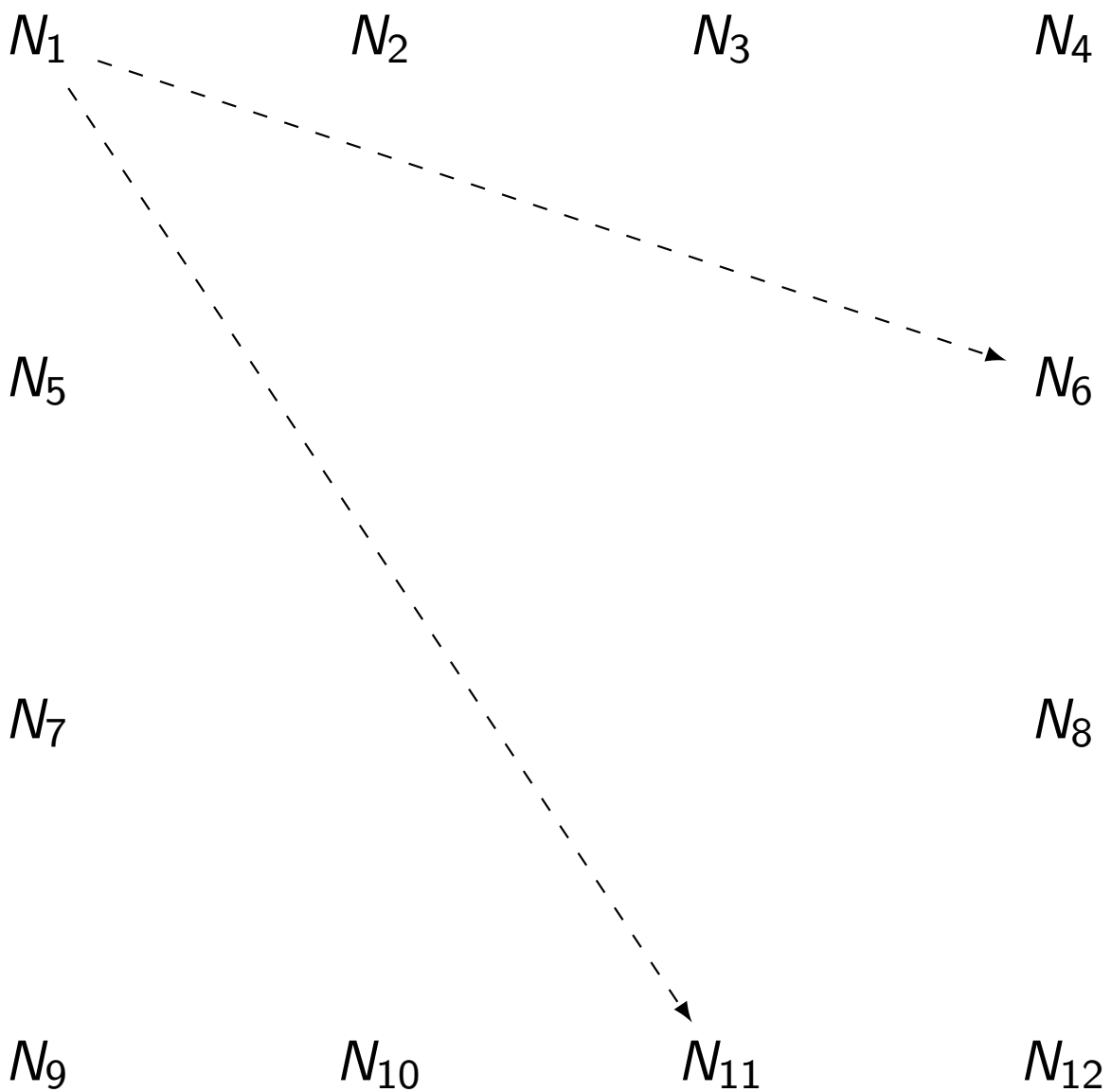
N_1 N_2 N_3 N_4

N_5 N_6

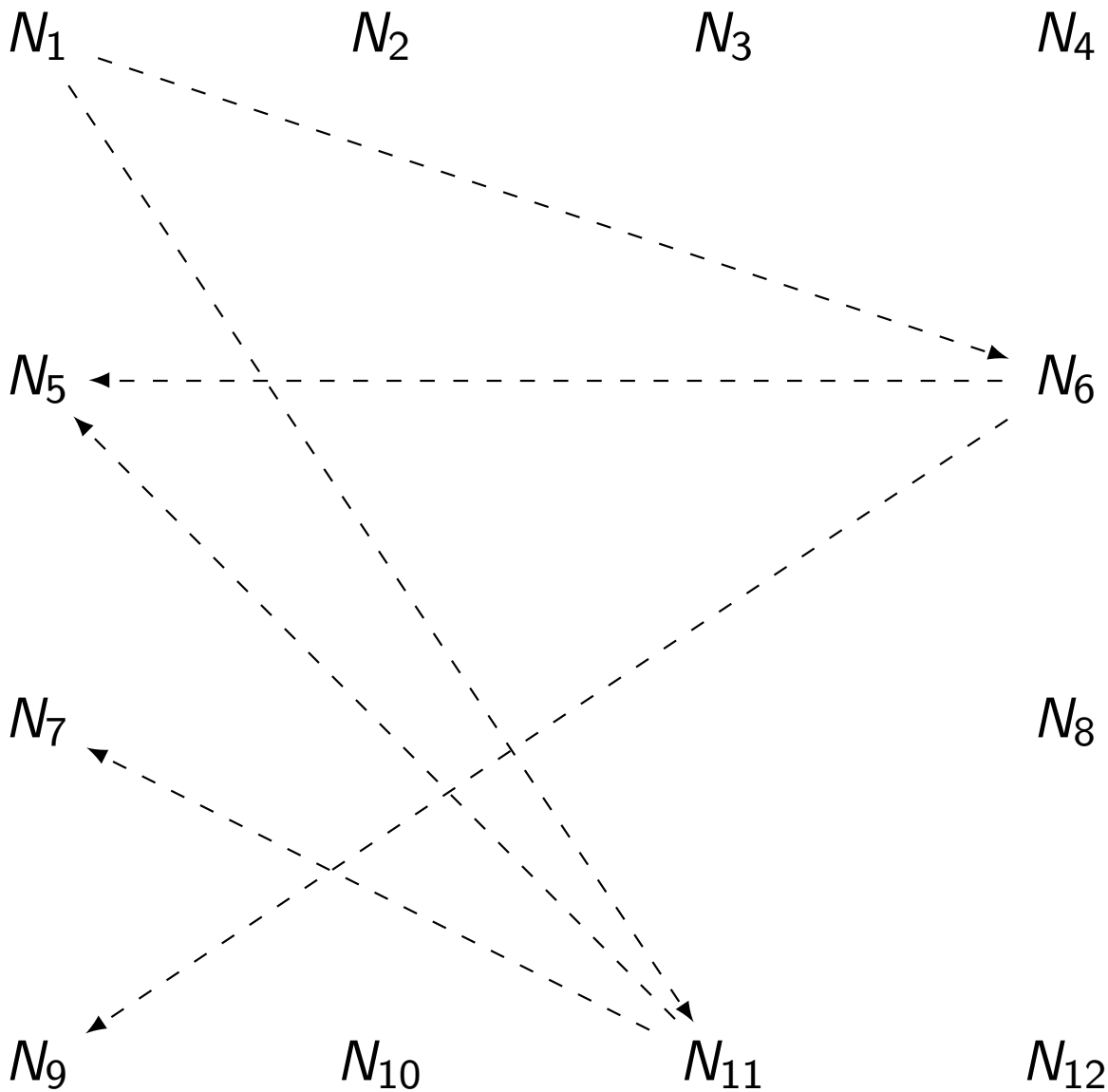
N_7 N_8

N_9 N_{10} N_{11} N_{12}

Gossipping



Gossipping

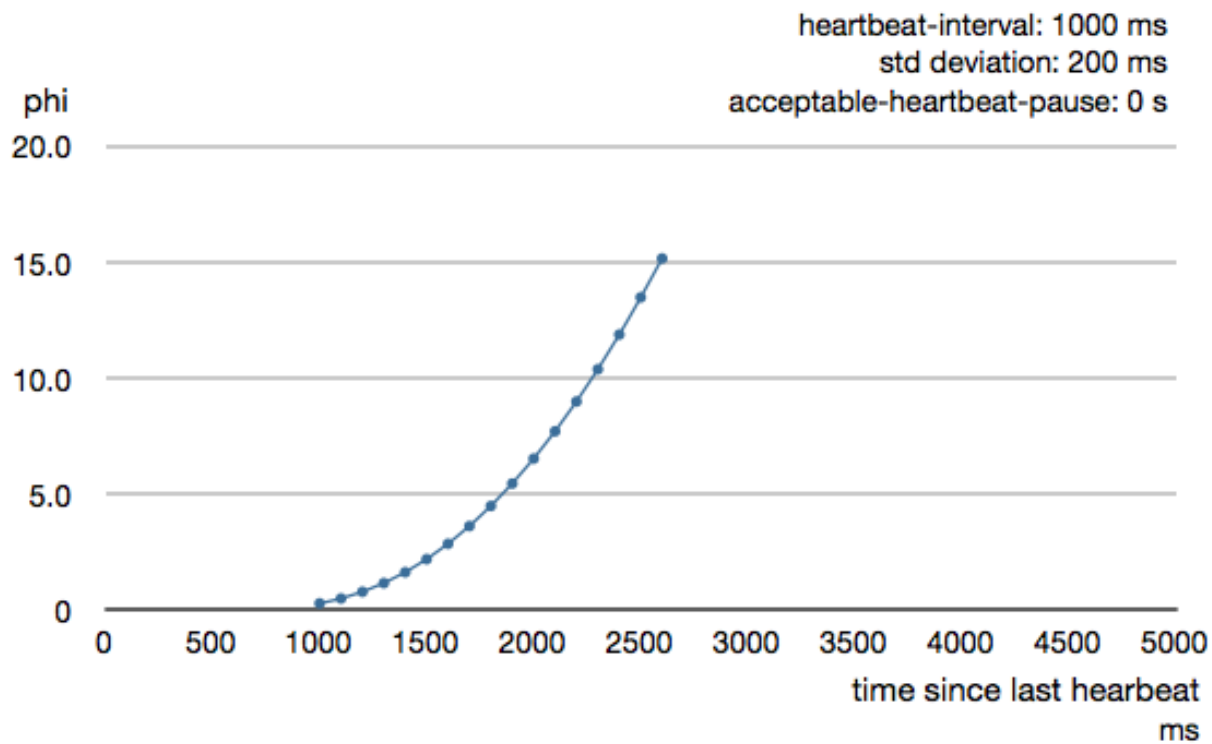


Gossipping

- Jeder Knoten verbreitet Informationen periodisch weiter an zufällige weitere Knoten
- Funktioniert besonders gut mit CvRDTs
  - Nachrichtenverlust unkritisch
- Anwendungen
  - Ereignis-Verteilung
  - Datenabgleich
  - Anti-entropy Protokolle
  - Aggregate, Suche

Heartbeats

- Kleine Nachrichten in regelmäßigen Abständen
- Standardabweichung kann dynamisch berechnet werden
- $\Phi = -log_{10}(1 - F(timeSinceLastHeartbeat))$



Akka Clustering

- Verteiltes Aktorsystem
  - Infrastruktur wird über gossipping Protokoll geteilt
  - Ausfälle werden über Heartbeats erkannt
- Sharding: Horizontale Verteilung der Resourcen
  - In Verbindung mit Gossipping mächtig

(Anti-)Patterns: Request/Response

- Problem: Warten auf eine Antwort — Benötigt einen Kontext der die Antwort versteht
- Pragmatische Lösung: Ask-Pattern

```
import akka.patterns.ask

(otherActor ? Request) map {
  case Response ⇒ //
}
```

- Eignet sich nur für sehr einfache Szenarien
- Lösung: Neuer Aktor für jeden Response Kontext

(Anti-)Patterns: Nachrichten

- Nachrichten sollten typisiert sein

```
otherActor ! "add 5 to your local state" // NO
otherActor ! Modify(_ + 5) // YES
```

- Nachrichten dürfen nicht veränderlich sein!

```
val state: scala.collection.mutable.Buffer
otherActor ! Include(state) // NO
otherActor ! Include(state.toList) // YES
```

- Nachrichten dürfen keine Referenzen auf veränderlichen Zustand enthalten

```
var state = 7
otherActor ! Modify(_ + state) // NO
val stateCopy = state
otherActor ! Modify(_ + stateCopy) // YES
```

(Anti-)Patterns: State-Leaks

- Lokaler Zustand darf auf keinen Fall “auslaufen”!

```
var state = 0
(otherActor ? Request) map { case Response ⇒ sender !
    RequestComplete }
```

(Anti-)Patterns: State-Leaks

- Lokaler Zustand darf auf keinen Fall “auslaufen”!

```
var state = 0
(otherActor ? Request) map { case Response ⇒ sender !
    RequestComplete }
```

- Besser?

```
(otherActor ? Request) map { case Response ⇒
  state += 1; RequestComplete
} pipeTo sender
```

(Anti-)Patterns: State-Leaks

- Lokaler Zustand darf auf keinen Fall “auslaufen”!

```
var state = 0
(otherActor ? Request) map { case Response ⇒ sender !
    RequestComplete }
```

- Besser?

```
(otherActor ? Request) map { case Response ⇒
  state += 1; RequestComplete
} pipeTo sender
```

- So geht's!

```
(otherActor ? Request) map { case Response ⇒
  self ! IncState
  RequestComplete
} pipeTo sender
```

(Anti-)Patterns: Single-Responsibility

- Problem: Fehler in Komplexen Aktoren sind kaum behandelbar

```
var interestDivisor = initial

def receive = {
  case Divide(dividend, divisor) ⇒
    sender ! Quotient(dividend / divisor)
  case CalculateInterest(amount) ⇒
    sender ! Interest(amount / interestDivisor)
  case AlterInterest(by) ⇒
    interestDivisor += by
}
```

(Anti-)Patterns: Single-Responsibility

- Problem: Fehler in Komplexen Aktoren sind kaum behandelbar

```
var interestDivisor = initial

def receive = {
  case Divide(dividend, divisor) ⇒
    sender ! Quotient(dividend / divisor)
  case CalculateInterest(amount) ⇒
    sender ! Interest(amount / interestDivisor)
  case AlterInterest(by) ⇒
    interestDivisor += by
}
```

- Welche Strategie bei `DivByZeroException`?

(Anti-)Patterns: Single-Responsibility

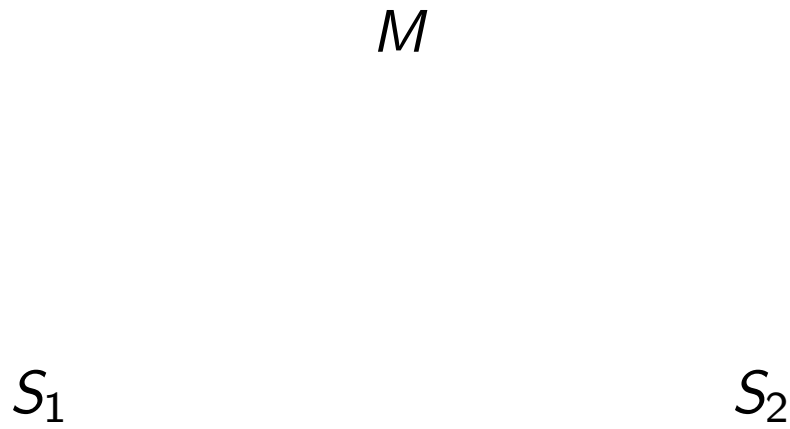
- Problem: Fehler in Komplexen Aktoren sind kaum behandelbar

```
var interestDivisor = initial

def receive = {
  case Divide(dividend, divisor) ⇒
    sender ! Quotient(dividend / divisor)
  case CalculateInterest(amount) ⇒
    sender ! Interest(amount / interestDivisor)
  case AlterInterest(by) ⇒
    interestDivisor += by
}
```

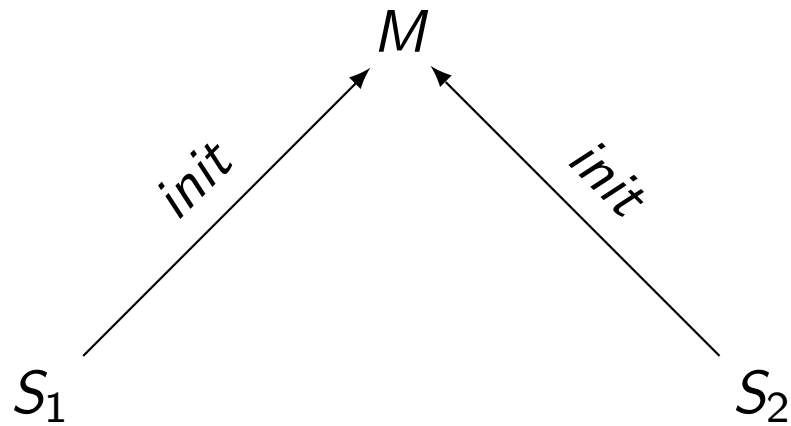
- Welche Strategie bei `DivByZeroException`?
- Ein Aktor sollte immer nur eine Aufgabe haben!

(Anti-)Patterns: Aktor-Beziehungen



- Problem: Wer registriert sich bei wem in einer Master-Slave-Hierarchie?

(Anti-)Patterns: Aktor-Beziehungen



- Problem: Wer registriert sich bei wem in einer Master-Slave-Hierarchie?
- Slaves sollten sich beim Master registrieren!
  - Flexibel / Dynamisch
  - Einfachere Konfiguration in verteilten Systemen

(Anti-)Patterns: Aufgabenverteilung

- Problem: Nach welchen Regeln soll die Aktorhierarchie aufgebaut werden?

(Anti-)Patterns: Aufgabenverteilung

- Problem: Nach welchen Regeln soll die Aktorhierarchie aufgebaut werden?
- Wichtige Informationen und zentrale Aufgaben sollten möglichst nah an der Wurzel sein.
- Gefährliche bzw. unsichere Aufgaben sollten immer Kindern übertragen werden.

(Anti-)Patterns: Zustandsfreie Aktoren

- Ein Aktor ohne Zustand

```
class Calculator extends Actor {
  def receive = {
    case Divide(x,y) ⇒ sender ! Result(x / y)
  }
}
```

(Anti-)Patterns: Zustandsfreie Aktoren

- Ein Aktor ohne Zustand

```
class Calculator extends Actor {
  def receive = {
    case Divide(x,y) ⇒ sender ! Result(x / y)
  }
}
```

- Ein Fall für Käpt'n Future!

```
class UsesCalculator extends Actor {
  def receive = {
    case Calculate(Divide(x,y)) ⇒
      Future(x/y) pipeTo self
    case Result(x) ⇒
      println("Got it: " + x)
  }
}
```

(Anti-)Pattern: Initialisierung

- Problem: Aktor benötigt Informationen bevor er mit der eigentlichen Arbeit loslegen kann

(Anti-)Pattern: Initialisierung

- Problem: Aktor benötigt Informationen bevor er mit der eigentlichen Arbeit loslegen kann
- Lösung: Parametrisierter Zustand

```
class Robot extends Actor {
  def receive = uninitialized
  def uninitialized: Receive = {
    case Init(pos,power) ⇒
      context.become(initialized(pos,power))
  }
  def initialized(pos: Point, power: Int): Receive = {
    case Move(North) ⇒
      context.become(initialized(pos + (0,1), power - 1))
  }
}
```

(Anti-)Patterns: Kontrollnachrichten

- Problem: Aktor mit mehreren Zuständen behandelt bestimmte Nachrichten in jedem Zustand gleich

(Anti-)Patterns: Kontrollnachrichten

- Problem: Aktor mit mehreren Zuständen behandelt bestimmte Nachrichten in jedem Zustand gleich
- Lösung: Verkettete partielle Funktionen

```
class Obstacle extends Actor {
  def rejectMoveTo: Receive = {
    case MoveTo ⇒ sender ! Reject
  }
  def receive = uninitialized orElse rejectMoveTo
  def uninitialized: Receive = ...
  def initialized: Receive = ...
}
```

(Anti-)Patterns: Circuit Breaker

- Problem: Wir haben eine elastische, reaktive Anwendung aber nicht genug Geld um eine unbegrenzt große Server Farm zu betreiben.
- Lösung: Bei Überlastung sollten Anfragen nicht mehr verarbeitet werden.

```
class DangerousActor extends Actor with ActorLogging {
  val breaker =
    new CircuitBreaker(context.system.scheduler,
      maxFailures = 5,
      callTimeout = 10.seconds,
      resetTimeout = 1.minute).onOpen(notifyMeOnOpen())

  def notifyMeOnOpen(): Unit =
    log.warning("My CircuitBreaker is now open, and
        will not close for one minute")
```

(Anti)-Patterns: Message Transformer

```
class MessageTransformer(from: ActorRef, to: ActorRef,
  transform: PartialFunction[Any,Any]) extends Actor {

  def receive = {
    case m ⇒ to forward transform(m)
  }
}
```

Weitere Patterns

- Lange Aufgaben unterteilen
- Aktor Systeme sparsam erstellen
- Futures sparsam einsetzen
- `Await.result()` nur bei Interaktion mit Nicht-Aktor-Code
- Dokumentation Lesen!

Zusammenfassung

- Nachrichtenaustausch in verteilten Systemen ist unzuverlässig
- Zwei Armeen Problem
- Lösungsansätze
  - Drei-Wege Handschlag
  - Nachrichtennummerierung
  - Heartbeats
  - Gossipping Protokolle
- Patterns und Anti-Patterns
- Nächstes mal: Theorie der Nebenläufigkeit

Reaktive Programmierung
Vorlesung 16 vom 14.07.2015: Theorie der Nebenläufigkeit

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Organisatorisches

Wir sind umgezogen!

- Martin Ring: MZH 1362

- Christoph Lüth: MZH 1361

Fahrplan

- Teil I: Grundlegende Konzepte
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte
  - Bidirektionale Programmierung: Zippers and Lenses
  - Eventual Consistency
  - Robustheit, Entwurfsmuster
  - Theorie der Nebenläufigkeit

Theorie der Nebenläufigkeit

- Nebenläufige Systeme sind kompliziert
  - Nicht-deterministisches Verhalten
  - Neue Fehlerquellen wie Deadlocks
  - Schwer zu testen
- Reaktive Programmierung kann diese Fehlerquellen einhegen
- Theoretische Grundlagen zur Modellierung nebenläufiger Systeme
  - zur Spezifikation (CSP)
  - aber auch als Berechnungsmodell (π-Kalkül)

Temporale Logik, Prozessalgebren und Modelchecking

- Prozessalgebren und temporale Logik beschreiben Systeme anhand ihrer Zustandsübergänge
- Ein System ist dabei im wesentlichen eine endliche Zustandsmaschine $\mathcal{M} = \langle S, \Sigma, \rightarrow \rangle$ mit Zustandsübergang $\rightarrow \subseteq S \times \Sigma \times S$
- Temporale Logiken reden über eine Zustandsmaschine
- Prozessalgebren erlauben mehrere Zustandsmaschinen und ihre Synchronisation
- Der Trick ist Abstraktion: mehrere interne Zustandsübergänge werden zu einem Zustandsübergang zusammengefaßt

Einfache Beispiele

- Einfacher Kaffee-Automat:

$$P = 10c \rightarrow coffee \rightarrow P$$

- Kaffee-Automat mit Auswahl:

$$P = 10c \rightarrow coffee \rightarrow P \;\Box\; 20c \rightarrow latte \rightarrow P$$

- Pufferprozess:

$$COPY = left?x \rightarrow right!x \rightarrow COPY$$

NB. Eingabe ($c?x$) und Ausgabe ($c!x$) sind reine Konvention.

CSP: Syntax

Gegeben Prozeßalphabet Σ, besondere Ereignisse $\checkmark, \tau$

$P ::= Stop \mid a \to P \mid \mu\, P.F(P)$	fundamentale Operationen
$\mid P \,\Box\, Q \mid P \sqcap Q$	externe und interne Auswahl
$\mid P \parallel Q \mid P \underset{X}{\parallel} Q$	synchronisiert parallel
$\mid P \,\vert\vert\vert\, Q$	unsynchronisiert parallel
$\mid P \setminus X$	hiding
$\mid Skip \mid P;\ Q$	sequentielle Komposition

Externe vs. interne Auswahl

- Interne Zustandsübergänge (τ) sind nicht beobachtbar, aber können Effekte haben.
- Vergleiche:

$$a \to b \to \mathit{Stop} \,\Box\, a \to c \to \mathit{Stop}$$

$$a \to b \to \mathit{Stop} \,\sqcap\, a \to c \to \mathit{Stop}$$

$$a \to (b \to \mathit{Stop} \,\Box\, c \to \mathit{Stop})$$

$$a \to (b \to \mathit{Stop} \,\sqcap\, c \to \mathit{Stop})$$

Beispiel: ein Flugbuchungssystem

- Operationen des Servers:
  - Nimmt Anfragen an, schickt Resultate (mit flid)
  - Nimmt Buchungsanfragen an, schickt Bestätigung (ok) oder Fehler (fail)
  - Nimmt Stornierung an, schickt Bestätigung
- Unterschied zwischen interner Auswahl $\sqcap$ (Server trifft Entscheidung), und externer Auswahl $\Box$ (Server reagiert)

$$
\begin{aligned}
SERVER = \; & query?(from, to) \rightarrow result!flid \rightarrow SERVER \\
& \Box\, booking?flid \rightarrow (ok \rightarrow SERVER \sqcap fail \rightarrow SERVER) \\
& \Box\, cancel?flid \rightarrow ok \rightarrow SERVER
\end{aligned}
$$

Eingabe $(c?x)$ und Ausgabe $(c!a)$ sind reine Konvention

Beispiel: ein Flugbuchungssystem

- Operationen des Servers:
  - Nimmt Anfragen an, schickt Resultate (mit flid)
  - Nimmt Buchungsanfragen an, schickt Bestätigung (ok) oder Fehler (fail)
  - Nimmt Stornierung an, schickt Bestätigung
- Unterschied zwischen interner Auswahl $\sqcap$ (Server trifft Entscheidung), und externer Auswahl $\Box$ (Server reagiert)

$$\begin{aligned} SERVER = \; & query \to result \to SERVER \\ & \Box\, booking \to (ok \to SERVER \sqcap fail \to SERVER) \\ & \Box\, cancel \to ok \to SERVER \end{aligned}$$

Eingabe $(c?x)$ und Ausgabe $(c!a)$ sind reine Konvention

Beispiel: ein Flugbuchungssystem

- Der Client:
  - Stellt Anfrage
  - wenn der Flug richtig ist, wird er gebucht;
  - oder es wird eine neue Anfrage gestellt.

$$CLIENT = query \rightarrow result \rightarrow (booking \rightarrow ok \rightarrow CLIENT \sqcap CLIENT)$$

- Das Gesamtsystem — Client und Server synchronisiert:

$$SYSTEM = CLIENT \parallel SERVER$$

Beispiel: ein Flugbuchungssystem

- Der Client:
  - Stellt Anfrage
  - wenn der Flug richtig ist, wird er gebucht;
  - oder es wird eine neue Anfrage gestellt.

$$CLIENT = query \rightarrow result \rightarrow (booking \rightarrow ok \rightarrow CLIENT \sqcap CLIENT)$$

- Das Gesamtsystem — Client und Server synchronisiert:

$$SYSTEM = CLIENT \parallel SERVER$$

- Problem: Deadlock
  - Es gibt Werkzeuge (Modelchecker, z.B. FDR), um solche Deadlocks in Spezifikationen zu finden

Beispiel: ein Flugbuchungssystem

- Der Client:
  - Stellt Anfrage
  - wenn der Flug richtig ist, wird er gebucht;
  - oder es wird eine neue Anfrage gestellt.

$$\begin{aligned} CLIENT \quad = query \rightarrow result \rightarrow \\ (booking \rightarrow \quad (ok \rightarrow CLIENT \\ \Box\ fail \rightarrow CLIENT) \\ \sqcap\ CLIENT) \end{aligned}$$

- Das Gesamtsystem — Client und Server synchronisiert:

$$SYSTEM = CLIENT \parallel SERVER$$

- Problem: Deadlock
  - Es gibt Werkzeuge (Modelchecker, z.B. FDR), um solche Deadlocks in Spezifikationen zu finden

Ziele der Semantik von Prozesskalkülen

- Reasoning about processes by their external behaviour
- Untersuchung von
  - Verfeinerung (Implementation)
  - deadlock: Keine Transition möglich
  - livelock: Divergenz
- Grundlegender Begriff: Äquivalenz (Gleichheit) von Prozessen

Operationale Semantik für CSP (I)

Definition: Labelled Transition System (LTS)

Ein labelled transition system (LTS) ist $L = (N, A, \rightarrow)$ mit Menge N der Knoten (Zustände), Menge A von Labels und Relation $\{\xrightarrow{a} \subseteq N \times N\}_{a \in A}$ von Kanten (Zustandsübergänge).

Hier: $N = P$,$A = \Sigma \bigcup \{\checkmark, \tau\}$, $\rightarrow$ definiert wie folgt:

$$\frac{}{e \rightarrow P \xrightarrow{a} P[a/e]} \; a \in comms(e)$$

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

Operationale Semantik für CSP (II)

$$\frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \,\Box\, Q \xrightarrow{\tau} P \,\Box\, Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \,\Box\, Q \xrightarrow{a} P'} \; a \neq \tau \qquad \frac{Q \xrightarrow{a} Q'}{P \,\Box\, Q \xrightarrow{a} Q'} \; a \neq \tau$$

$$\frac{P \xrightarrow{x} P'}{P \setminus B \xrightarrow{\tau} P'} \; x \in B \qquad \frac{P \xrightarrow{x} P'}{P \setminus B \xrightarrow{x} P' \setminus B} \; x \notin B$$

Operationale Semantik für CSP (III)

$$\frac{P \xrightarrow{\tau} P'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P' \underset{X}{\parallel} Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} P \underset{X}{\parallel} Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \underset{X}{\parallel} Q \xrightarrow{a} P' \underset{X}{\parallel} Q} \; a \notin X \qquad \frac{Q \xrightarrow{a} Q'}{P \underset{X}{\parallel} Q \xrightarrow{a} P \underset{X}{\parallel} Q'} \; a \notin X$$

$$\frac{P \xrightarrow{a} P' \; Q \xrightarrow{a} Q'}{P \underset{X}{\parallel} Q \xrightarrow{a} P' \underset{X}{\parallel} Q'} \; a \in X$$

Denotationale Semantik für CSP

- Operationale Semantik erklärt das Verhalten, erlaubt kein Reasoning
- Denotationale Semantik erlaubt Abstraktion über dem Verhalten
- Für CSP: Denotat eines Prozesses ist:
  - die Menge aller seiner Traces
  - die Menge seiner Traces und Acceptance-Mengen
  - die Menge seiner Traces und seiner Failure/Divergence-Mengen

Anwendungsgebiete für CSP

- Modellierung nebenläufiger Systeme (Bsp: ISS)
- Verteilte Systeme und verteilte Daten
- Analyse von Krypto-Protokollen
- Hautpwerkzeug: der Modellchecker FDR
  - `http://www.cs.ox.ac.uk/projects/fdr/`