

# 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/~cx1/lehre/rp.ss15](http://www.informatik.uni-bremen.de/~cx1/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:

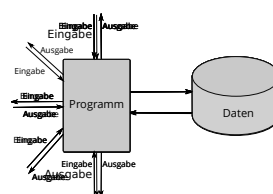


... aber die Welt ändert sich:



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

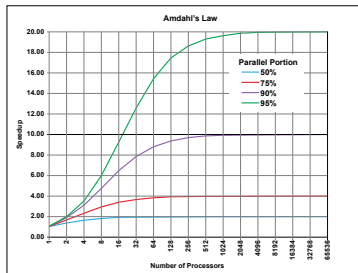
## Probleme mit dem herkömmlichen Ansatz



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

## Amdahl's Law

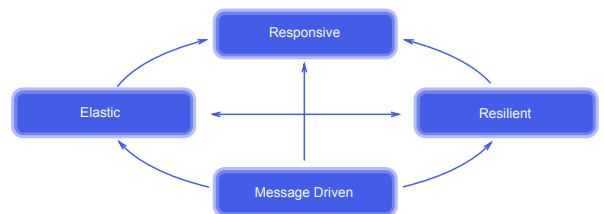
"The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20x as shown in the diagram, no matter how many processors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



Quelle: Wikipedia

## The Reactive Manifesto

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

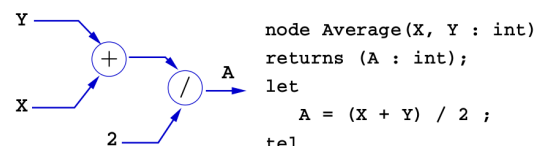


## Was ist Reaktive Programmierung?

- ▶ 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 Zeitverzög
  - ▶ Verwendung in der Luftfahrtindustrie (Airbus)



## 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

9 [36]

## 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

10 [36]

## Fahrplan

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

11 [36]

## 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`

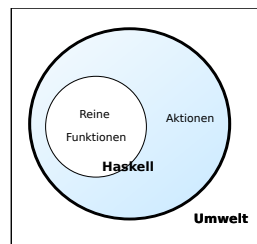
12 [36]

## Rückblick Haskell

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

13 [36]

## 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“

14 [36]

## Aktionen als abstrakter Datentyp

- ▶ ADT mit Operationen `Komposition` und `Lifting`

### Signatur:

```
type IO α
```

```
( $\gg$ ) :: IO α -> (α -> IO β) -> IO β
```

```
return :: α -> IO α
```

- ▶ Plus elementare Operationen (lesen, schreiben etc)

15 [36]

## Elementare Aktionen

- ▶ Zeile von `stdin` lesen:

```
getLine :: IO String
```

- ▶ Zeichenkette auf `stdout` ausgeben:

```
putStr :: String -> IO ()
```

- ▶ Zeichenkette mit Zeilenvorschub ausgeben:

```
putStrLn :: String -> IO ()
```

16 [36]

## Einfache Beispiele

- ▶ Echo einfach

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

- ▶ Echo mehrfach

```
echo :: IO ()
echo = getLine >>= putStrLn >>= \_ → echo
```

- ▶ Was passiert hier?

- ▶ Verknüpfen von Aktionen mit `>>=`
- ▶ Jede Aktion gibt **Wert** zurück

17 [36]

## Noch ein Beispiel

- ▶ Umgekehrtes Echo:

```
ohce :: IO ()
ohce = getLine
      >>= \s → 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 >>= \_ → q
```

18 [36]

## Die do-Notation

- ▶ Syntaktischer Zucker für IO:

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

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

- ▶ Rechts sind `>>=`, `>>` implizit.

- ▶ Es gilt die **Abseitsregel**.

- ▶ Einrückung der ersten Anweisung nach `do` bestimmt Abseits.

19 [36]

## 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

20 [36]

## Module in der Standardbibliothek

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

21 [36]

## Ein/Ausgabe mit Dateien

- ▶ Im Prelude vordefiniert:

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

- ▶ Datei lesen (verzögert):

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

- ▶ Mehr Operationen im Modul IO der Standardbibliothek

- ▶ Buffered/Unbuffered, Seeking, &c.
- ▶ Operationen auf Handle

22 [36]

## 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

23 [36]

## 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`, ...

24 [36]

## 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]
```

25 [36]

## 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**

26 [36]

## Fehler fangen und behandeln

- ▶ Fehlerbehandlung für wc:

```
wc2 :: String → IO ()
wc2 file =
  catch (wc file)
    (\e → 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
```

27 [36]

## 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
```

28 [36]

## 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'))
```

29 [36]

## Funktionen mit Zustand

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

$$f : A \times S \rightarrow B \times S$$

$$\cong$$

$$f : A \rightarrow S \rightarrow B \times S$$

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

30 [36]

## In Haskell: Zustände **explizit**

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

```
type State σ α = σ → (α, σ)
```

- ▶ Komposition zweier solcher Berechnungen:

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

- ▶ Lifting:

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

31 [36]

## 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)
```

32 [36]

## 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

33 [36]

## Aktionen als Zustandstransformationen

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

34 [36]

## 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**

35 [36]

## 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

36 [36]