

# 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](http://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)

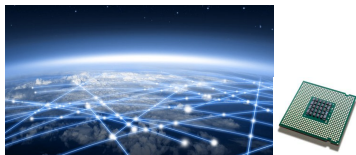
Eigenschaften:

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

Zugrundeliegendes Paradigma:

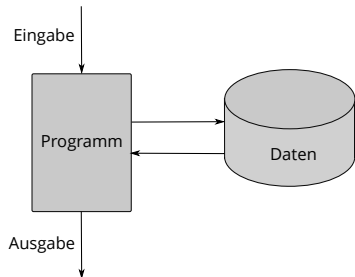


... aber die Welt ändert sich:

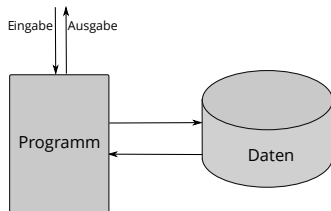


- ▶ Das **Netz** verbindet Rechner
- ▶ Selbst eingebettete Systeme sind vernetzt (Auto: ca. 100 Proz.)
- ▶ Mikroprozessoren sind **mehrkernig**
- ▶ Systeme sind **eingebettet**, **nebenläufig**, **reagieren** auf ihre Umwelt.

# Probleme mit dem herkömmlichen Ansatz

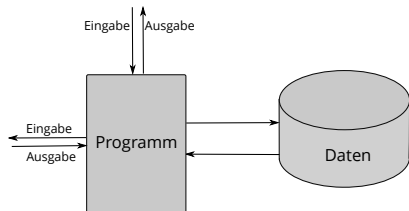


# Probleme mit dem herkömmlichen Ansatz



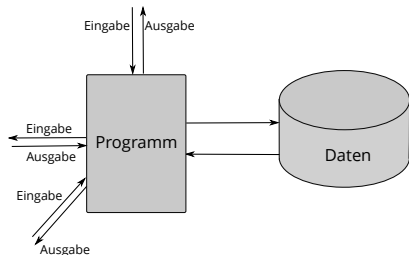
- Problem: Nebenläufigkeit

# Probleme mit dem herkömmlichen Ansatz



- Problem: Nebenläufigkeit

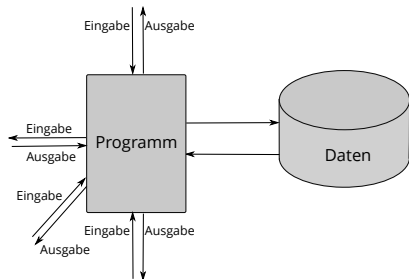
# Probleme mit dem herkömmlichen Ansatz



- ▶ Problem: Nebenläufigkeit
- ▶ Nebenläufigkeit verursacht Synchronisationsprobleme

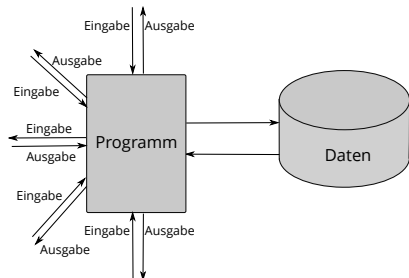


# Probleme mit dem herkömmlichen Ansatz



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

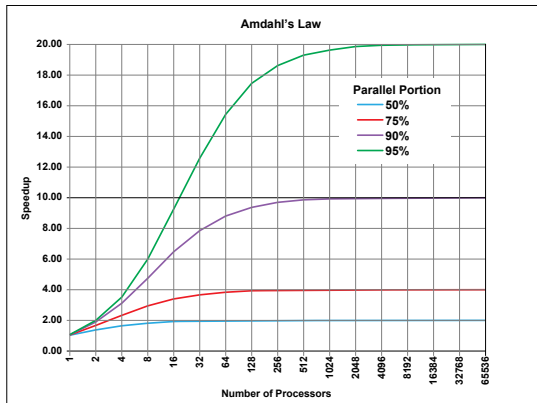
# Probleme mit dem herkömmlichen Ansatz



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

# Amdahl's Law

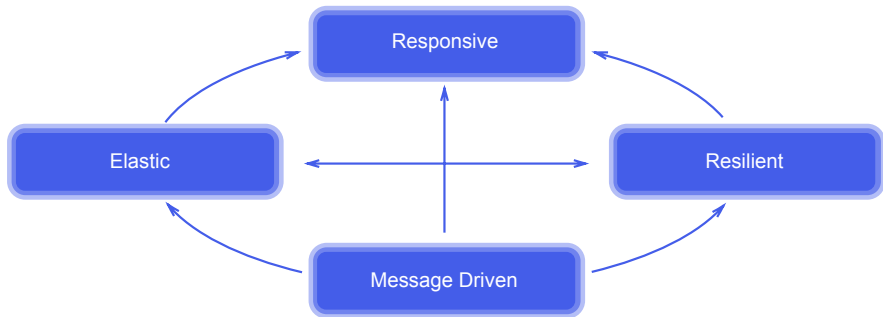
“The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors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



Quelle: Wikipedia

# The Reactive Manifesto

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

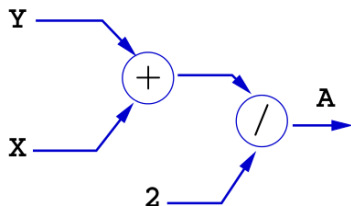


# Was ist Reaktive Programmierung?

- ▶ 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

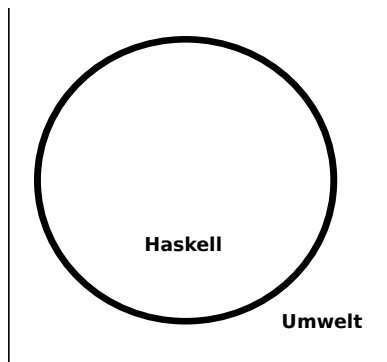
# Rückblick Haskell

- ▶ 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

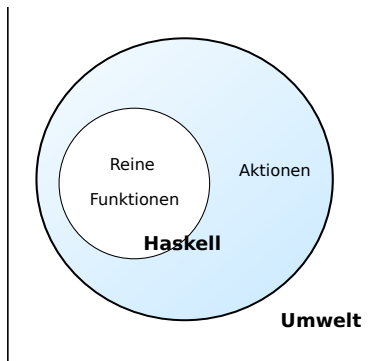
# Ein- und Ausgabe in Haskell



## Problem:

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

# Ein- und Ausgabe in Haskell



## Problem:

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

## Lösung:

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

# Aktionen als abstrakter Datentyp

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

- ▶ Signatur:

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

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

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

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

# Elementare Aktionen

- ▶ Zeile von `stdin` lesen:

```
getLine  :: IO String
```

- ▶ Zeichenkette auf `stdout` **ausgeben**:

```
putStr   :: String → IO ()
```

- ▶ Zeichenkette mit Zeilenvorschub **ausgeben**:

```
putStrLn :: String → IO ()
```

# Einfache Beispiele

- ▶ Echo einfach

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

- ▶ Echo mehrfach

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

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



## 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
```

# Die do-Notation

- ▶ Syntaktischer Zucker für IO:

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



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

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

## 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 Standardbibliothek
  - ▶ `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  $\alpha$   $\rightarrow$  IO  $\alpha$   
forever a = a  $\gg$  forever a
```

- ▶ Iteration (feste Anzahl):

```
forN :: Int  $\rightarrow$  IO  $\alpha$   $\rightarrow$  IO ()  
forN n a | n == 0    = return ()  
         | otherwise = a  $\gg$  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 :: ( $\alpha$ ,  $\alpha$ )  $\rightarrow$  IO  $\alpha$ 
```

- ▶ Warum ist randomIO **Aktion**?

- ▶ **Beispiele:**

- ▶ Aktion zufällig oft ausführen:

```
atmost :: Int  $\rightarrow$  IO  $\alpha$   $\rightarrow$  IO [ $\alpha$ ]  
atmost most a =  
  do l  $\leftarrow$  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  $f : A \rightarrow B$  mit Seiteneffekt in **Zustand**  $S$ :

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

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

# In Haskell: Zustände **explizit**

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

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

- ▶ Komposition zweier solcher Berechnungen:

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

- ▶ Lifting:

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

# Beispiel: Ein Zähler

- ▶ Datentyp:

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

- ▶ 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  $A \rightarrow V$  (Adressen  $A$ , Werte  $V$ )
  - ▶ Zustand des Dateisystems
  - ▶ Zustand des Zufallsgenerators
- ▶ In Haskell: Typ `RealWorld`

```
type IO  $\alpha$  = State RealWorld  $\alpha$  — ... 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