Reaktive Programmierung
Vorlesung 1 vom 05.04.17: Was ist Reaktive Programmierung?

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2017

Organisatorisches

- Vorlesung: Mittwochs 14-16, MZH 1110
- Übung: Donnerstags 12-14, MZH 1450 (nach Bedarf)
- Webseite: `www.informatik.uni-bremen.de/~cxl/lehre/rp.ss17`
- Scheinkriterien:
  - Voraussichtlich 6 Übungsblätter
  - Alle bearbeitet, insgesamt 40% (Notenspiegel PI3)
  - Übungsgruppen 2 – 4 Mitglieder
  - Danach: Fachgespräch oder Modulprüfung

Warum Reaktive Programmierung?

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

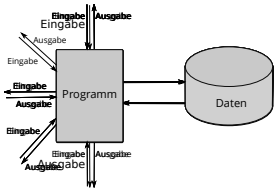
Eigenschaften:
- Imperativ und prozedural
- Sequentiell

Zugrundeliegendes Paradigma:

. . . aber die Welt ändert sich:

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

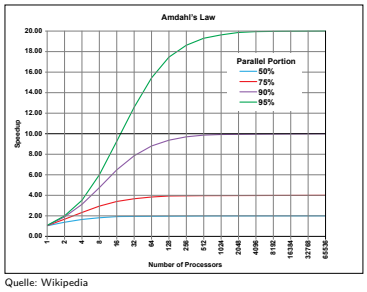
Probleme mit dem herkömmlichen Ansatz



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

Amdahl's Law

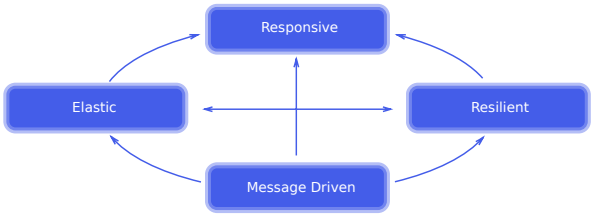
"The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors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



Quelle: Wikipedia

The Reactive Manifesto

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

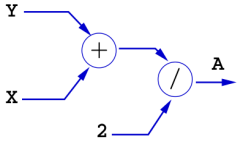


Was ist Reaktive Programmierung?

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

Datenflusssprachen (data flow languages)

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




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

Struktur der VL

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

Fahrplan

- Einführung
- Monaden als Berechnungsmuster
- Nebenläufigkeit: Futures and Promises
- Aktoren I: Grundlagen
- Aktoren II: Implementation
- Bidirektionale Programmierung
- Meta-Programmierung
- Reaktive Ströme I
- Reaktive Ströme II
- Functional Reactive Programming
- Software Transactional Memory
- Eventual Consistency
- Robustheit und Entwurfsmuster
- Theorie der Nebenläufigkeit, Abschluss

Rückblick Haskell

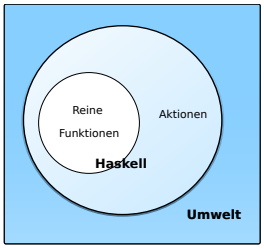
Rückblick Haskell

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

Rückblick Haskell

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

Ein- und Ausgabe in Haskell



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

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

Aktionen als abstrakter Datentyp

- ADT mit Operationen Komposition und Lifting
- Signatur:

```
type IO α
(>>=)  :: IO α → (α→ IO β) → IO β
return :: α→ IO α
```

- Plus elementare Operationen (lesen, schreiben etc)

Elementare Aktionen

- Zeile von stdin lesen:

```
getLine :: IO String
```

- Zeichenkette auf stdout ausgeben:

```
putStr  :: String→ IO ()
```

- Zeichenkette mit Zeilenvorschub ausgeben:

```
putStrLn :: String→ IO ()
```

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 $\gg=$, >> 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

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 System.IO der Standardbücherei
  - Buffered/Unbuffered, Seeking, &c.
  - Operationen auf Handle
- Noch mehr Operationen in System.Posix
  - Filedeskriptoren, Permissions, special devices, etc.

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
- Erstaunlich (hinreichend) effizient

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
```


Kontrollstrukturen

- Vordefinierte Kontrollstrukturen (Control.Monad):

```
when :: Bool→ IO ()→ IO ()
```

- Sequenzierung:

```
sequence :: [IO α]→ IO [α]
```

  - Sonderfall: [()] als ()

```
sequence_ :: [IO ()]→ IO ()
```

- Map und Filter für Aktionen:

```
mapM     :: (α→ IO β)→ [α]→ IO [β]
mapM_    :: (α→ IO ())→ [α]→ IO ()
filterM  :: (α→ IO Bool) → [α] → IO [α]
```

Fehlerbehandlung

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

```
throw :: Exception γ⇒ γ→ α
catch :: Exception γ⇒ IO α → (γ→ IO α) → IO α
try   :: Exception γ⇒ IO α → IO (Either γ α)
```

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

Fehler fangen und behandeln

- Fehlerbehandlung für wc:

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

- 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
```


Beispiel: Traversion eines Verzeichnisbaums

- Verzeichnisbaum traversieren, und für jede Datei eine Aktion ausführen:

```
travFS :: (FilePath→ IO ())→ FilePath→ IO ()
```

- Nutzt Funktionalität aus System.Directory, System.FilePath

```
travFS action p = do
  res ← try (getDirectoryContents p)
  case res of
    Left  e → putStrLn $ "ERROR: "++ show (e :: IOError)
    Right cs → do let cp = map (p </>) (cs \\ [".", ".."])
                  dirs  ← filterM doesDirectoryExist cp
                  files ← filterM doesFileExist cp
                  mapM_ action files
                  mapM_ (travFS action) dirs
```


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'))
```


Module in der Standardbücherei

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

Fallbeispiel: Wörter raten

- Unterhaltungsprogramm: der Benutzer rät Wörter
- Benutzer kann einzelne Buchstaben eingeben oder das ganze Wort
- Wort wird maskiert ausgegeben, nur geratene Buchstaben angezeigt

Wörter raten: Programmstruktur

- Trennung zwischen Spiel-Logik und Nutzerschnittstelle
- Spiel-Logik (GuessGame):
  - Programmzustand:

```
data State = St { word   :: String -- Zu ratendes Wort
                , hits   :: String -- Schon geratene Buchstaben
                , miss   :: String -- Falsch geratene Buchstaben
                }
```

  - Initialen Zustand (Wort auswählen):

```
initialState :: [String]→ IO State
```

  - Nächsten Zustand berechnen (Char ist Eingabe des Benutzers):

```
data Result = Miss | Hit | Repetition | GuessedIt |
    TooManyTries

processGuess :: Char→ State→ (Result, State)
```

Wörter raten: Nutzerschnittstelle

- Hauptschleife (play)
  - Zustand anzeigen
  - Benutzereingabe abwarten
  - Neuen Zustand berechnen
  - Rekursiver Aufruf mit neuem Zustand
- Programmanfang (main)
  - Lexikon lesen
  - Initialen Zustand berechnen
  - Hauptschleife aufrufen

```
play :: State→ IO ()
play st = do
  putStrLn (render st)
  c ← getGuess st
  case (processGuess c st) of
    (Hit, st)  → play st
    (Miss, st) → do putStrLn "Sorry, no."; play st
    (Repetition, st)→ do putStrLn "You already tried that."; play
        st
    (GuessedIt, st)→ putStrLn "Congratulations, you guessed it."
    (TooManyTries, st) →
      putStrLn $ "The word was "++ word st ++ " -- you lose."
```


Kontrollumkehr

- Trennung von Logik (State, processGuess) und Nutzerinteraktion nützlich und sinnvoll
- Wird durch Haskell Tysystem unterstützt (keine UI ohne IO)
- Nützlich für andere UI mit Kontrollumkehr
- Beispiel: ein GUI für das Wörterratespiel (mit Gtk2hs)
  - GUI ruft Handler-Funktionen des Nutzerprogramms auf
  - Spielzustand in Referenz (IORef) speichern
- Vgl. MVC-Pattern (Model-View-Controller)

Eine GUI für das Ratespiel

- Binden von Funktionen an Signale

```
-- Process key presses
onKeyPress window $ λe→ case eventKeyChar e of
  Just c→ do handleKeyPress window l1 l2 gs c; return
      True

-- Process quit button
```

- Eventloop von Gtk2Hs aufrufen (Kontrollumkehr):

```
-- Run it!
onDestroy window mainQuit
widgetShowAll window
render st l1 l2
```


Zusammenfassung

- War das jetzt reaktiv?
  - Haskell ist funktional
  - Für eine reaktive Sprache fehlt Nebenläufigkeit
- Nächstes Mal:
  - Monaden, Ausnahmen, Referenzen in Haskell und Scala
- Danach: Nebenläufigkeit in Haskell und Scala