Reaktive Programmierung
Vorlesung 1 vom 02.04.19
Was ist Reaktive Programmierung?

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2019

Organisatorisches

- Vorlesung: Mittwochs 14-16, MZH 1110
- Übung: Donnerstags 16-18, MZH 1450 (nach Bedarf)
- Webseite: `www.informatik.uni-bremen.de/~cxl/lehre/rp.ss19`
- 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)

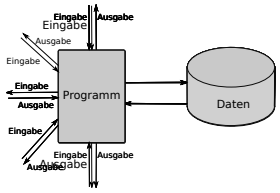
Eigenschaften:
- **Imperativ** und **prozedural**
- **Sequentiell**

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.

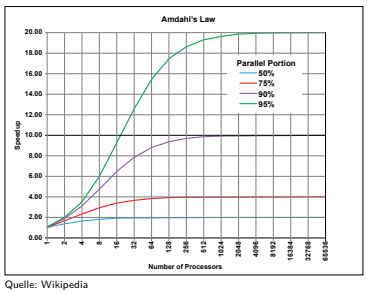
Probleme mit dem herkömmlichen Ansatz



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

Amdahl's Law

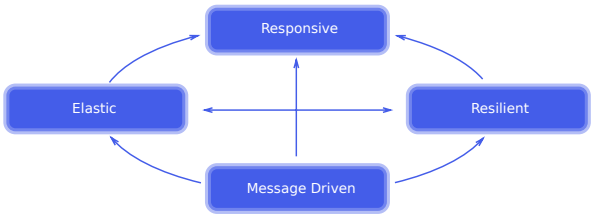
"The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. For example, if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20× as shown in the diagram, no matter how many processors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



Quelle: Wikipedia

The Reactive Manifesto

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

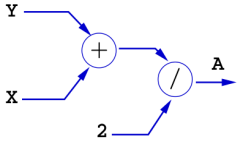


Was ist Reaktive Programmierung?

- 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, sexy types
  - Robustheit: Eventual Consistency, Entwurfsmuster
- Bilingualer **Übungsbetrieb** und **Vorlesung**
  - Kein Scala-Programmierkurs
  - Erlernen von Scala ist nützlicher Seiteneffekt

Fahrplan

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

Rückblick Haskell

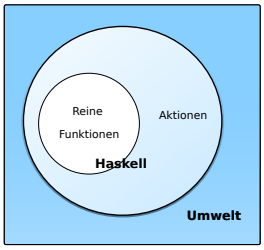
Rückblick Haskell

- 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 ()
```

Die do-Notation

- Syntaktischer Zucker für IO:

```
echo =                          echo = do
  getLine                         s← getLine
  ≫= λs→ putStrLn s      ⟺       putStrLn s
  ≫ echo                          echo
```

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

Zustandsabhängige Berechnungen

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

```
curry   :: ((α, β) → γ)→ α→ β→ γ
uncurry :: (α→ β→ γ)→ (α, β) → γ
```


In Haskell: Zustände explizit

- **Zustandstransformer:** 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
```

- Trivialer Zustand:

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

- Lifting von Funktionen:

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


Zugriff auf den Zustand

- Zustand lesen:

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

- Zustand setzen:

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


Einfaches Beispiel

- Zähler als Zustand:

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

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

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

- Hauptfunktion (verkapselt State):

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


Monaden

Monaden als Berechnungsmuster

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

State:

```
type State σ α
```

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

```
lift :: α→ State σ α
```

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

Aktionen:

```
type IO α
```

```
(≫=)   :: IO α →
          (α→ IO β) →
          IO β
```

```
return :: α→ IO α
```

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

Berechnungsmuster: **Monade**

Monaden als Berechngsmuster

Eine Monade ist:

- **mathematisch**: durch Operationen und Gleichungen definiert (verallgemeinerte algebraische Theorie)
- als **Berechnungsmuster**: **verknüpfbare** Berechnungen mit einem **Ergebnis**
- in **Haskell**: durch mehrere Typklassen definierte Operationen mit **Eigenschaften**

Monaden in Haskell

- Aktion auf Funktionen:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

fmap bewahrt Identität und Komposition:

```
    fmap id  ==  id
fmap (f ∘ g) ==  fmap f ∘ fmap g
```

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

Monaden in Haskell

- Applicative:

```
class Functor f⇒ Applicative f where
  pure :: a→ f a
  <*> :: f (a → b) → f a → f b
```

Eigenschaften: links-neutralität, bewahrt Komposition, Homomorphismus:

```
                    pure id <*> v == v
  pure (∘) <*> u <*> v <*> w == u <*> (v <*> w)
             pure f <*> pure x == pure (f x)
                  u <*> pure y == pure ($ y) <*> u
```


Monaden in Haskell

- Verkettung (>>=) und Lifting (return):

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

>>=ist assoziativ und return das neutrale Element:

```
      return a >>= k  ==  k a
        m >>= return  ==  m
m >>= (λx → k x >>= h) ==  (m >>= k) >>= h
```

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

Monaden mit Möglichkeiten

- Alternativen:

```
class Applicative f ⇒ Alternative f where
  empty :: f a
  <|>   :: f a→ f a→ f a
```

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

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

- Gleichungen: mzero Identität für mplus und>>=, mplus assoziativ.

Beispiele für Monaden

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

Die Reader-Monade

- Aus dem Zustand wird nur gelesen:

```
data Reader σ α = R {run :: σ→ α}
```

- Instanzen:

```
instance Functor (Reader σ) where
  fmap f (R g) = R (f. g)

instance Monad (Reader σ) where
  return a = R (const a)
  R f >>= g = R $ λs→ run (g (f s)) s
```

- Nur eine elementare Operation:

```
get :: (σ→ α)→ Reader σ α
get f = R $ λs→ f s
```


Fehler und Ausnahmen

- Maybe als Monade:

```
instance Functor Maybe where
  fmap f (Just a) = Just (f a)
  fmap f Nothing  = Nothing

instance Monad Maybe where
  Just a >>= g  = g a
  Nothing >>= g = Nothing
  return = Just
```

  - Ähnlich mit Either
- Berechnungsmodell: **Ausnahmen** (Fehler)
  - f :: α→ Maybe β ist Berechnung mit möglichem Fehler
  - Fehlerfreie Berechnungen werden verkettet
  - Fehler (Nothing oder Left x) werden propagiert

Mehrdeutigkeit

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

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

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

- Berechnungsmodell: Mehrdeutigkeit
  - `f :: α→ [β]` ist Berechnung mit **mehreren** möglichen Ergebnissen
  - Verkettung: Anwendung der folgenden Funktion auf **jedes** Ergebnis (concatMap)

Beispiel

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

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

  2. Damit Permutationen (rekursiv):

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


Der Listenmonade in der Listenkomprehension

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

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

  2. Damit Permutationen (rekursiv):

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

- Listenkomprehension ≅ Listenmonade

IO ist keine Magie

Referenzen in Haskell

- Zustand als **finite map** von Referenzen auf Werte
- Ungetypt: `SimpleRefs`
- Typ der Werte ist Typparameter des Zustands

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

- Getypt: `Refs`
  - Typ der Werte durch Typparameter der Referenz
  - Nutzt **dynamische Typen**:

```
readRef :: Typeable a⇒ Ref a→ Stateful a
writeRef :: Typeable a⇒ Ref a→ a→ Stateful ()
```


Implizite vs. explizite Zustände

- Wie funktioniert jetzt IO?
- Nachteil von `State`: Zustand ist **explizit**
  - Kann dupliziert werden
- Daher: Zustand **implizit** machen
  - Datentyp verkapseln (kein `run`)
  - Zugriff auf `State` nur über elementare Operationen

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

Zusammenfassung

- War das jetzt **reaktiv?**
  - Haskell ist **funktional**
  - Für eine reaktive Sprache fehlt **Nebenläufigkeit**
- Nächstes Mal:
  - Monaden **komponieren** — Monadentransformer
- Danach: Nebenläufigkeit in Haskell und Scala