

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 12 vom 01.02.2021: Anwendungsbeispiel: Funktionale Webanwendungen

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität
Bremen

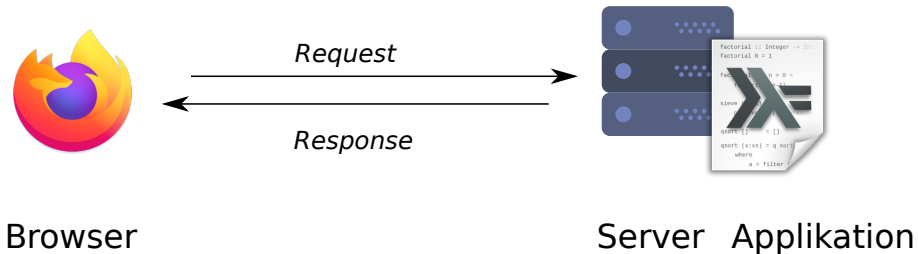
Wintersemester 2020/21

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ **Teil III: Funktionale Programmierung im richtigen Leben**
 - ▶ Aktionen und Zustände
 - ▶ Monaden als Berechnungsmuster
 - ▶ Funktionale Webanwendungen
 - ▶ Scala — Eine praktische Einführung
 - ▶ Rückblick & Ausblick

I. Eine kurze Einführung in die Webentwicklung

Wie funktioniert das Web?



Kennzeichen einer Webanwendung

- ▶ **Zustandsfreiheit**: jeder Request ist ein neuer
- ▶ **Nebenläufigkeit**: ein Server, viele Browser (gleichzeitig)
- ▶ **Entkoppelung**: Serveranwendung und Browser weit entfernt

Grober Ablauf

① Browser stellt **Anfrage**

▶ *Gib mir Seite `/home/cxl/`*

② Server nimmt Anfrage entgegen, **löst** Anfrage auf

▶ *`/home/cxl/`, das muss die Datei `/var/www/cxl/index.html` sein.*

③ Server sendet Antwort

▶ *Hier ist die Seite: `<h1>Hallo</h1><p>Foo ba...`*

Verfeinerter Ablauf

- ▶ Das **Protokoll** ist HTTP (RFC 2068, 7540). HTTP kennt vier Arten von Requests: GET, POST, PUT, DELETE.
- ▶ Von der URL `http://www.foo.de/baz/bar/` löst der Server den **Pfad** (`/bar/bar/` zu einer Resource auf (**Routing**). Das kann eine Datei sein (static routing), oder es wird eine Funktion aufgerufen, die ein Ergebnis erzeugt.
- ▶ HTTP kennt verschiedene Arten von **response codes** (100, 404, ...). Der Inhalt der Antwort ist **beliebig**, und nicht notwendigerweise HTML.

Architekturermägungen

- ▶ Webanwendungen müssen **zustandsfrei** sein und **skalieren**
- ▶ Übertragung ist **unzuverlässig**.
- ▶ Dazu: **REST** (Representational State Transfer)
 - ▶ Sammlung von **Architekturprinzipien**
- ▶ Dazu: CRUD (create, read, update, delete)

Merkmale von REST-Architekturen

- ① Zustandslosigkeit — jede Nachricht in sich vollständig
- ② Caching
- ③ Einheitliche Schnittstelle:
 - ▶ Adressierbare Ressourcen — als URL
 - ▶ Repräsentation zur Veränderungen von Ressourcen
 - ▶ Selbstbeschreibende Nachrichten
 - ▶ *Hypermedia as the engine of the application state* (HATEOAS)
- ④ Architektur: Client-Server, mehrschichtig

Anatomie einer Web-Applikation

- ▶ **Routing:** Auflösen der Pfade zu Aktionen
- ▶ Eigentliche Aktion
- ▶ **Persistentes** Backend
- ▶ Erzeugung von HTML (meistens), JSON (manchmal)

Hands on!

Übung 12.1: HTTP handgemacht

Wenn nötig, installieren Sie das Programm `telnet` (oder schalten es, in Windows, frei). Starten Sie das Programm und verbinden Sie sich mit dem Web-Server der Uni Bremen:

```
telnet> open www.uni-bremen.de 80
```

Jetzt können Sie ein HTTP-Request von Hand eingeben (danach eine Leerzeile eingeben):

```
GET / HTTP/1.1
```

```
Host: www.uni-bremen.de
```

Was passiert?

Der Server antwortet

Der Server antwortet

Lösung: Wahrscheinlich kommt die Antwort:

```
HTTP/1.1 301 Moved Permanently
Date: Sun, 31 Jan 2021 01:52:40 GMT
Server: Apache/2.4.29 (Ubuntu)
Location: https://www.uni-bremen.de/
Content-Length: 317
Content-Type: text/html; charset=iso-8859-1
```

gefolgt von etwas HTML. Damit teilt uns der Server mit, dass er nur noch https-Anfragen annimmt (und sagt uns, wo). Eventuell kommt auch

```
HTTP/1.1 400 Bad Request
...
```

dann war die Anfrage nicht HTTP-konform (wahrscheinlich vertippt?).

II. Web Development in Haskell

Scotty: ein einfaches Web-Framework

From the web-page <https://hackage.haskell.org/package/scotty>:

Scotty is the cheap and cheerful way to write RESTful, declarative web applications.

- ▶ A page is as simple as defining the verb, url pattern, and Text content.
- ▶ It is template-language agnostic. Anything that returns a Text value will do.
- ▶ Conforms to WAI Application interface.
- ▶ Uses very fast Warp webserver by default.

Ein erster Eindruck

```
{-# LANGUAGE OverloadedStrings #-}  
  
import Web.Scotty  
  
import Data.Monoid (mconcat)  
  
main = scotty 3000 $  
  get "/:word" $ do  
    beam ← param "word"  
    html $ mconcat ["<h1>Scotty, ", beam, " me up!</h1>"]
```

(Auch von der Webseite.)

Ein erstes Problem

- ▶ Repräsentation von Zeichenketten als `type String=[Char]` ist elegant, aber benötigt **Platz** und ist **langsam**.
- ▶ Daher gibt es **mehrere** Alternativen:
 - ▶ `Data.Text` Unicode-Text, strikt und schnell
 - ▶ `Data.Text.Lazy`, Unicode-Text, String kann größer sein als der Speicher
 - ▶ `Data.ByteString` Sequenzen von Bytes, kein Unicode, kompakt
- ▶ Deshalb `mconcat [...]` oben (`class Monoid`)
- ▶ String-Literale können **überladen** werden (`LANGUAGE OverloadedStrings`)
- ▶ Mit `pack` und `unpack` Konversion von Strings in oder von `Text`.
- ▶ Potenzielle Quelle der Verwirrung: Scotty nutzt `Text.Lazy`, Blaze nutzt `Text`.

HTML

- ▶ Scotty gibt nur den Inhalt zurück, aber wir wollen HTML erzeugen.
- ▶ Drei Möglichkeiten:
 - 1 Text selber zusammensetzen: `"<h1>Willkommen!</h1>\n`
 - 2 Templating: HTML-Dokumente durch Haskell anreichern lassen (Hamlet, Heist)
 - 3 Zugrundeliegende Struktur (DOM) in Haskell erzeugen, und in Text konvertieren.

Erzeugung von HTML: Blaze

Selbstbeschreibung: <https://jaspervdj.be/blaze/>

BlazeHtml is a blazingly fast HTML combinator library for the Haskell programming language. It embeds HTML templates in Haskell code for optimal efficiency and composability.

- ▶ Kann (X)HTML4 und HTML5 erzeugen.
- ▶ Dokument wird als Monade repräsentiert und wird durch Kombinatoren erzeugt:

```
numbers :: Int → Html
numbers n = docTypeHtml $ do
    H.head $ do
        H.title "Natural numbers"
    body $ do
        p "A list of natural numbers:"
        ul $ forM_ [1 .. n] (li ∘ toHtml)
```

```
image = img ! src "foo.png" ! alt "A foo image."
```

- ▶ Siehe Tutorial.

Persistenz

- ▶ Eine Web-Applikation muss **Zustände** verwalten können
 - ▶ Nutzerdaten, Warenbestand, Einkauf, ...
- ▶ Üblicher Ansatz: **Datenbank**
 - ▶ ACID-Eigenschaften garantiert, insbesondere Nebenläufigkeit
 - ▶ Aber: externe Anbindung nötig
- ▶ Hier: **Mutable Variables** `MVar` `a` (nicht durable, aber schnell und einfach)

Nebenläufige Zustände

- ▶ Haskell ist **nebenläufig** (hier ein Thread pro Verbindung)
- ▶ `MVar α` sind synchronisierte veränderlich Variablen.
- ▶ Kann **leer** oder **gefüllt** sein.

```
newMVar ::  $\alpha \rightarrow \text{IO (MVar } \alpha)$   
readMVar :: MVar  $\alpha \rightarrow \text{IO } \alpha$  — MVar bleibt gefüllt  
takeMVar :: MVar  $\alpha \rightarrow \text{IO } \alpha$  — MVar danach leer  
putMVar  :: MVar  $\alpha \rightarrow \alpha \rightarrow \text{IO ()}$  — Füllt MVar
```

- ▶ `readMVar` und `takeMVar` **blockieren**, wenn Variable leer ist
- ▶ Erlaubt einfache Synchronisation (vgl. `synchronized` in Java)

Zustand

- ▶ Wie können wir den Benutzer **identifizieren**?
- ▶ Ein Ansatz: **Cookies**
 - ▶ Widerspricht dem REST-Ansatz.
- ▶ Hier: über die URL — jeder Benutzer bekommt eine Resource

Erste Schritte.

Übung 12.2: Lecker Kekse!

Laden Sie die Quellen für die Vorlesung von heute herunter, und betreten Sie das Verzeichhnis `simple-cookies`. Dort finden Sie ein einfaches Programm, welches mit Cookies zählt, wie oft eine Seite aufgerufen wurde, und das obligatorische Hello World, welches ein einfaches HTML-Dokument mittels Blaze erzeugt.

Erweitern Sie das Programm `SimpleCookies.hs` so dass es ein HTML-Dokument zurückgibt. Schreiben Sie dazu eine Funktion

```
hitPage :: String → Html
```

welche über die Anzahl der bisherigen Aufrufe (`hits'`) parametrisiert ist.

Eine mögliche Lösung

Eine mögliche Lösung

Lösung:

```
hitPage :: String → Html
hitPage hits =
  docTypeHtml $ do
    H.head $ do
      H.title "Simple_Cookies"
    H.body $ do
      H.h1 "Hello_World."
      H.span $ do
        H.text "You_have_been_here_"
        H.text (T.pack hits)
        H.text "_times."
```

III. Ein Web-Shop für Onkel Bob

Architektur des Web-Shop

Model-View-Controller-Paradigma (Entwurfsmuster):

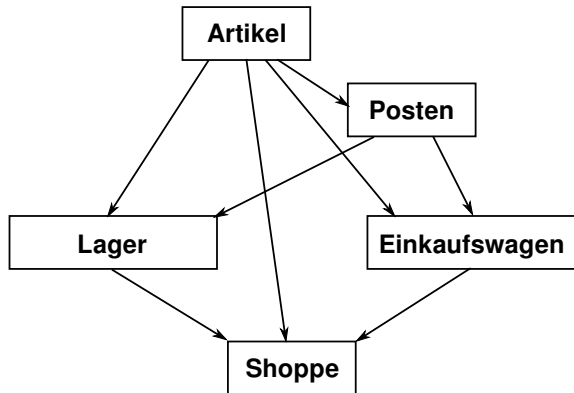
- ▶ Das **Model** ist der eigentliche (und persistente) Teil der Anwendung, bestehend aus den Datentypen samt der Funktionen darauf.
- ▶ Die **Views** sind Funktionen, die Webseiten aufbauen.
- ▶ Der **Controller** übersetzt Anfragen von außen in die Aufrufe der Model-Funktionen, erzeugt aus den Ergebnissen mit den Views Webseiten und schickt diese wieder zurück.

Entwurf der Anwendung

Resource	Methode	Daten
/	GET	Home-Page: Angebote anzeigen. Link zu neuem Einkauf
/einkauf/neu	GET	Neuen Einkauf beginnt, Einkaufswagen wird zugeteilt. Dann Weiterleitung zu folgender:
/einkauf/:id	GET	Einkaufswagen darstellen Link zur Bezahlseite
/einkauf/:id	POST	Angegebene Produkte in den Einkaufswagen
/einkauf/:id/kasse	GET	Bezahlseite mit Rechnung. Link zur Home-Page
/einkauf/:id/kaufen	GET	Bezahlt, Einkaufswagen löschen
/einkauf/:id/abbruch	GET	Abgebrochen, Produkte zurück
/einkauf/lieferung	POST	Anlieferung von Artikeln
/einkauf/lager	GET	Lagerbestand als JSON-Objekt

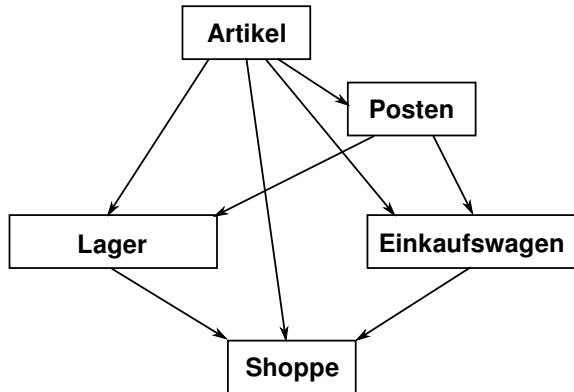
Model: der Shop

- ▶ Einheitliches Interface des Shop.
- ▶ Verwaltet Menge von **Einkäufen** (Einkaufswagen),
- ▶ Funktionen (Auszug):
 - ▶ Neuer Einkaufswagen
 - ▶ Produkt in Einkaufswagen
 - ▶ Einkauf abschließen/abbrechen
- ▶ Rein funktional, ADT **Shop** α



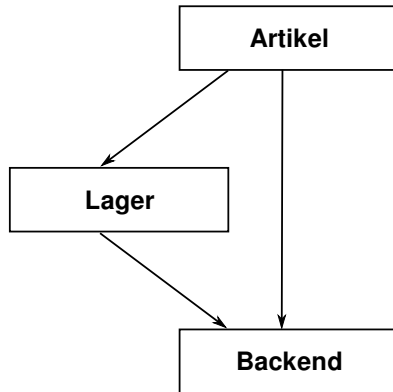
Model: der Shop

- ▶ Einheitliches Interface des Shop.
- ▶ Verwaltet Menge von **Einkäufen** (Einkaufswagen),
- ▶ Funktionen (Auszug):
 - ▶ Neuer Einkaufswagen
 - ▶ Produkt in Einkaufswagen
 - ▶ Einkauf abschließen/abbrechen
- ▶ Rein funktional, ADT **Shop** α
- ▶ Änderungen:
 - ▶ Einheitliche Mengen
 - ▶ Posten nicht mehr als ADT
 - ▶ Einkaufswagen nicht mehr als Modul



Model: der Shop

- ▶ Einheitliches Interface des Shop.
- ▶ Verwaltet Menge von **Einkäufen** (Einkaufswagen),
- ▶ Funktionen (Auszug):
 - ▶ Neuer Einkaufswagen
 - ▶ Produkt in Einkaufswagen
 - ▶ Einkauf abschließen/abbrechen
- ▶ Rein funktional, ADT **Shop** α
- ▶ Änderungen:
 - ▶ Einheitliche Mengen
 - ▶ Posten nicht mehr als ADT
 - ▶ Einkaufswagen nicht mehr als Modul



Controller

- ▶ Persistiert den **Zustand** des Shop (nur für Laufzeit des Servers)
- ▶ Nutzt **UUID** zur Zuordnung des Einkauf (garantiert eindeutige Bezeichner)
- ▶ **Zugriff** auf den Shop:
 - ▶ **Ändernd** (muss synchronisieren)
 - ▶ **Lesen** (ohne Synchronisation)

View

- ▶ Erzeugt Seiten (Templates):

```
homePage :: Text → [(Posten, Int)] → Html
```

```
shoppingPage :: String → String → [Text] → [(Posten, Int)]  
              → Int → [Posten] → Html
```

```
checkoutPage :: String → String → [(Posten, Int)] → Int → Html
```

```
thankYouPage :: Text → Html
```

- ▶ Weitere Funktionen: Artikelname, Mengeneinheiten, Euros etc.
- ▶ Artikel werden über eine eindeutige Kennung (`articleId`) identifiziert.

Zusammenfassung

- ▶ Wichtige Prinzipien für Web-Anwendungen:
 - ▶ Nebenläufigkeit, Zustandsfreiheit, REST
- ▶ Haskell ist für Web-Development gut geeignet:
 - ▶ Zustandsfreiheit macht Nebenläufigkeit einfach
 - ▶ Bequeme Manipulation von Bäumen
 - ▶ Abstraktionsbildung
- ▶ Web-Programmierung ist **umständlich**.