

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 1 vom 02.11.2020: Einführung

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Was ist Funktionale Programmierung?

- ▶ Programme als Funktionen — Funktionen als Programme
 - ▶ **Keine** veränderlichen Variablen
 - ▶ **Rekursion** statt while-Schleifen
- ▶ Funktionen als Daten — Daten als Funktionen
 - ▶ Erlaubt **Abstraktionsbildung**
- ▶ Denken in Algorithmen, nicht in Zustandsveränderung

Lernziele

- ▶ **Konzepte** und **typische Merkmale** des funktionalen Programmierens kennen, verstehen und anwenden können:
 - ▶ Modellierung mit **algebraischen Datentypen**
 - ▶ **Rekursion**
 - ▶ Starke **Typisierung**
 - ▶ **Funktionen höher Ordnung** (map, filter, fold)
- ▶ Datenstrukturen und Algorithmen in einer funktionalen Programmiersprache **umsetzen** und auf einfachere praktische Probleme **anwenden** können.

Modulhandbuch Informatik (Bachelor)

Die Vorlesung *Praktische Informatik 3* vermittelt essenzielles Grundwissen und Basisfähigkeiten, deren Beherrschung für nahezu jede vertiefte Beschäftigung mit Informatik Voraussetzung ist.

I. Organisatorisches

Personal

► **Vorlesung:**

Christoph Lüth <clueth@uni-bremen.de>

www.informatik.uni-bremen.de/~clueth/ (MZH 4186, Tel. 59830)

► **Tutoren:**

Thomas Barkowsky <barkowsky@informatik.uni-bremen.de>

Tobias Brandt <Tobias.Brandt@dfki.de>

Alexander Krug <krug@uni-bremen.de>

Robert Sachtleben <rob_sac@uni-bremen.de>

Muhammad Tarek Soliman <soliman@uni-bremen.de>

► **Webseite:** www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws20

Corona-Edition

- ▶ Vorlesungen sind **asynchron**
 - ▶ Videos werden Montags zur Verfügung gestellt
 - ▶ Vorlesungen in mehreren Teilen mit Kurzübungen
- ▶ Übungen: Präsenz/Online
 - ▶ Präsenzbetrieb für 56 Stud./Woche
 - ▶ 3 Tutorien mit Präsenzbetrieb
 - ▶ Präsenztutorium ist **optional!**
 - ▶ Präsenztermine gekoppelt an TI2 (gleiche Kohorte)
 - ▶ 3 Online-Tutorien

Termine

- ▶ **Vorlesung:** Online
- ▶ **Tutorien:**

Di	12– 14	MZH 1470	Robert	Online	Tobias
Do	10– 12	MZH 1470	Thomas	Online	Robert
	10– 12	MZH 1090	Tarek	Online	Alexander
- ▶ Alle Tutorien haben einen Zoom-Raum (für Präsenztutorien als Backup) — siehe Webseite
- ▶ Diese Woche **alle** Tutorien online — Präsenzbetrieb startet **nächste Woche**
- ▶ **Anmeldung** zu den Übungsgruppen über stud.ip (ab 18:00)
- ▶ **Sprechstunde:** Donnerstags 14-16 (via Zoom, bei Bedarf)

Scheinkriterien

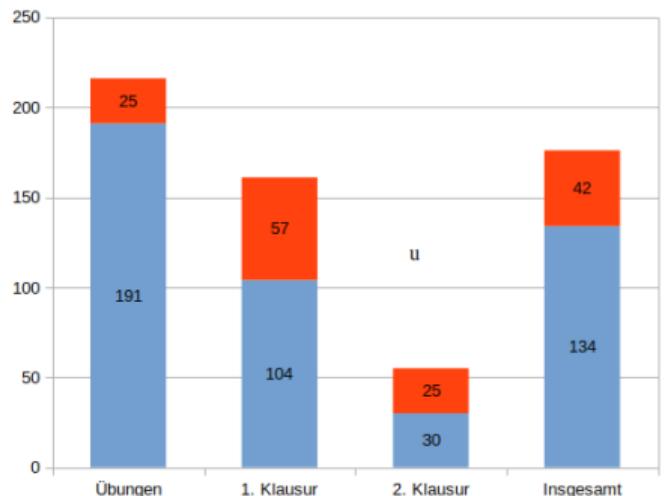
- ▶ Übungsblätter:
 - ▶ 6 Einzelübungsblätter (fünf beste werden gewertet)
 - ▶ 3 Gruppenübungsblätter (doppelt gewichtet)
- ▶ Übungsblätter der letzten Semester können **nicht** berücksichtigt werden
- ▶ Elektronische Klausur am Ende (Individualität der Leistung)
- ▶ Mind. 50% in den Einzelübungsblättern, in allen Übungsblättern und mind. 50% in der E-Klausur
- ▶ Note: 25% Übungsblätter und 75% E-Klausur
- ▶ **Notenspiegel** (in Prozent aller Punkte):

Pkt.%	Note	Pkt.%	Note	Pkt.%	Note	Pkt.%	Note
≥ 95	1.0	89.5-85	1.7	74.5-70	2.7	59.5-55	3.7
94.5-90	1.3	84.5-80	2.0	69.5-65	3.0	54.5-50	4.0
		79.5-75	2.3	64.5-60	3.3	49.5-0	n/b

Spielregeln

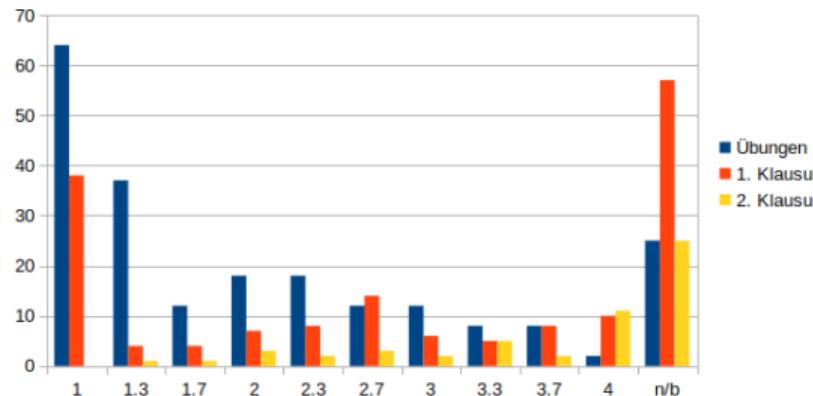
- ▶ **Quellen angeben** bei
 - ▶ Gruppenübergreifender Zusammenarbeit
 - ▶ Internetrecherche, Literatur, etc.
- ▶ **Täuschungsversuch:**
 - ▶ Null Punkte, **kein** Schein, **Meldung** an das **Prüfungsamt**
- ▶ **Deadline verpaßt?**
 - ▶ Triftiger Grund (z.B. Krankheit)
 - ▶ **Vorher** ankündigen, sonst **null** Punkte.

Statistik von PI3 im Wintersemester 19/20



Nicht bestanden
Bestanden

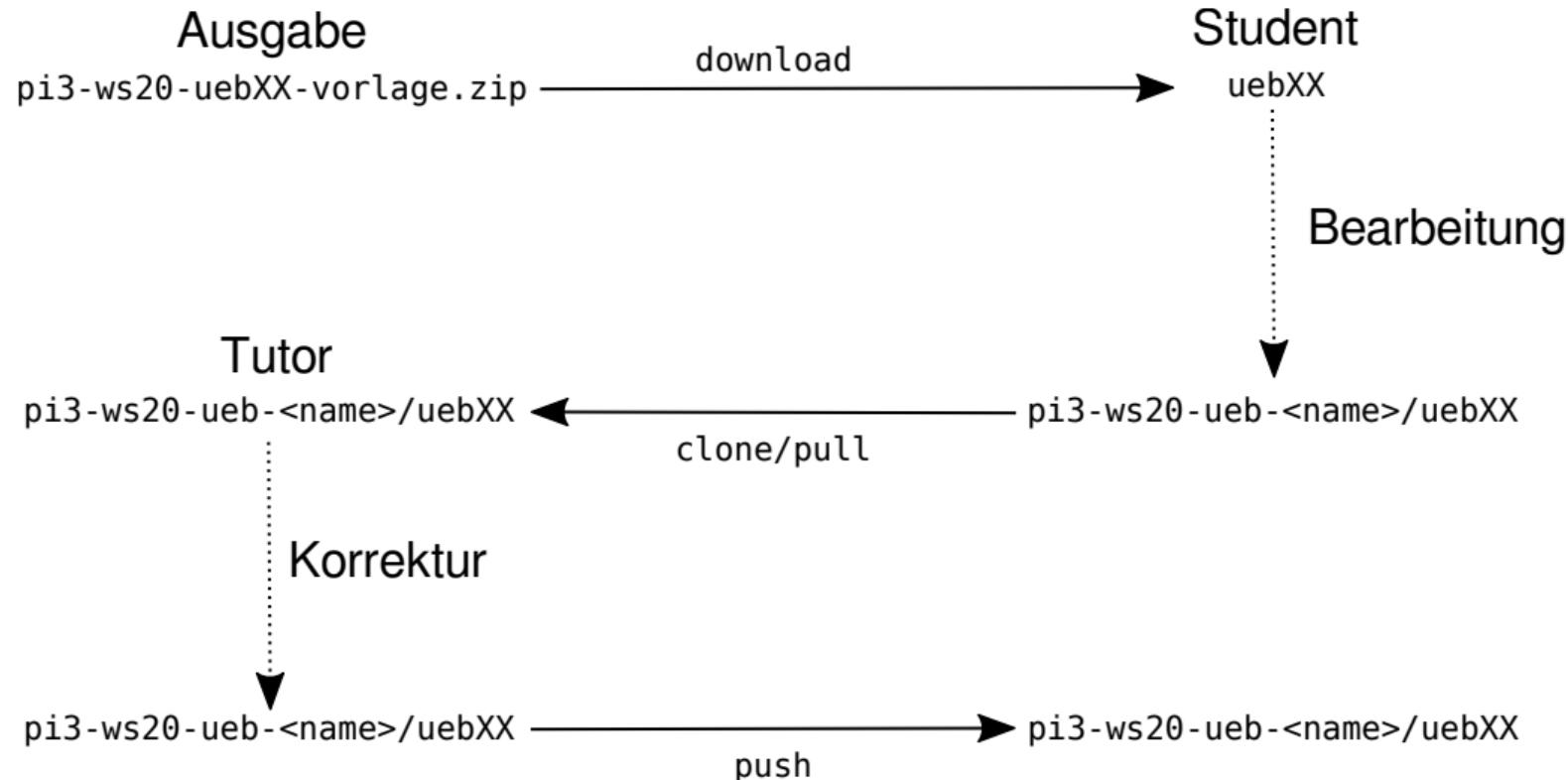
Notenverteilung



Übungsbetrieb

- ▶ Ausgabe der Übungsblätter über die Webseite **Montag mittag**
- ▶ Besprechung der Übungsblätter in den Tutorien
- ▶ 6 Einzelübungsblätter:
 - ▶ Bearbeitungszeit bis **Montag folgender Woche 12:00**
 - ▶ Die fünf besten werden gewertet
- ▶ 3 Gruppenübungsblätter (doppelt gewichtet):
 - ▶ Bearbeitungszeit bis **Montag übernächster Woche 12:00**
 - ▶ Übungsgruppen: max. **drei Teilnehmer**
- ▶ **Abgabe** elektronisch
- ▶ **Bewertung**: Korrektheit, Angemessenheit ("Stil"), Dokumentation

Ablauf des Übungsbetriebs



II. Einführung

Fahrplan

► Teil I: Funktionale Programmierung im Kleinen

- Einführung
 - Funktionen
 - Algebraische Datentypen
 - Typvariablen und Polymorphie
 - Funktionen höherer Ordnung I
 - Rekursive und zyklische Datenstrukturen
 - Funktionen höherer Ordnung II
-
- Teil II: Funktionale Programmierung im Großen
 - Teil III: Funktionale Programmierung im richtigen Leben

Warum funktionale Programmierung lernen?

- ▶ Funktionale Programmierung macht aus Programmierern Informatiker
- ▶ Blick über den Tellerrand — was kommt in 10 Jahren?
- ▶ **Herausforderungen** der Zukunft:
 - ▶ Nebenläufige und **reaktive** Systeme (Mehrkernarchitekturen, serverless computing)
 - ▶ Massiv **verteilte** Systeme („Internet der Dinge“)
 - ▶ Große **Datenmengen** („Big Data“)

The Future is Bright — The Future is Functional

- ▶ Funktionale Programmierung enthält die **wesentlichen** Elemente moderner Programmierung:
 - ▶ Datenabstraktion und **Funktionale Abstraktion**
 - ▶ Modularisierung
 - ▶ Typisierung und **Spezifikation**
- ▶ Funktionale Ideen jetzt im Mainstream:
 - ▶ Reflektion — LISP
 - ▶ Generics in Java — Polymorphie
 - ▶ Lambda-Fkt. in Java, C++ — Funktionen höherer Ordnung

Geschichtliches: Die Anfänge

► **Grundlagen** 1920/30

- Kombinatorlogik und λ -Kalkül (Schönfinkel, Curry, Church)

► Erste funktionale **Programmiersprachen** 1960

- LISP (McCarthy), ISWIM (Landin)

► **Weitere** Programmiersprachen 1970– 80

- FP (Backus); ML (Milner, Gordon); Hope (Burstall); Miranda (Turner)



Moses Schönfinkel



Haskell B. Curry



Alonzo Church



John McCarthy



John Backus



Robin Milner



Mike Gordon

Geschichtliches: Die Gegenwart

► **Konsolidierung** 1990

- CAML, Formale Semantik für Standard ML
- Haskell als Standardsprache

► **Kommerzialisierung** 2010

- OCaml
- Scala, Clojure (JVM)
- F# (.NET)

Warum Haskell?

- ▶ **Moderne** Sprache
- ▶ Standardisiert, mehrere **Implementationen**
 - ▶ Interpreter: ghci, hugs
 - ▶ Compiler: ghc, nhc98
 - ▶ Build: stack
- ▶ **Rein** funktional
- ▶ **Essenz** der funktionalen Programmierung



Programme als Funktionen

- ▶ Programme als Funktionen:

$$P : \text{Eingabe} \rightarrow \text{Ausgabe}$$

- ▶ Keine veränderlichen Variablen — kein versteckter Zustand
- ▶ Rückgabewert hängt ausschließlich von Werten der Argumente ab, nicht vom Aufrufkontext (**referentielle Transparenz**)
- ▶ Alle **Abhängigkeiten explizit**

Beispiel: Programmieren mit Funktionen

- ▶ Programme werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2
```

Beispiel: Programmieren mit Funktionen

- ▶ Programme werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
```

Beispiel: Programmieren mit Funktionen

- ▶ Programme werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)  
→ if False then 1 else 2* fac 1
```

Beispiel: Programmieren mit Funktionen

- ▶ Programme werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
```

Beispiel: Programmieren mit Funktionen

- ▶ Programme werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
```

Beispiel: Programmieren mit Funktionen

- ▶ Programme werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
      → 2* if False then 1 else 1* fac (1-1)
```

Beispiel: Programmieren mit Funktionen

- ▶ Programme werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
      → 2* if False then 1 else 1* fac (1-1)
      → 2* 1* fac 0
```

Beispiel: Programmieren mit Funktionen

- ▶ Programme werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
      → 2* if False then 1 else 1* fac (1-1)
      → 2* 1* fac 0
      → 2* 1* if 0 == 0 then 1 else 0* fac (0-1)
```

Beispiel: Programmieren mit Funktionen

- ▶ Programme werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
      → 2* if False then 1 else 1* fac (1-1)
      → 2* 1* fac 0
      → 2* 1* if 0 == 0 then 1 else 0* fac (0-1)
      → 2* 1* if True then 1 else 0* fac (0-1)
```

Beispiel: Programmieren mit Funktionen

- ▶ Programme werden durch **Gleichungen** definiert:

```
fac n = if n == 0 then 1 else n* fac(n-1)
```

- ▶ Auswertung durch **Reduktion** von **Ausdrücken**:

```
fac 2 → if 2 == 0 then 1 else 2* fac (2-1)
      → if False then 1 else 2* fac 1
      → 2* fac 1
      → 2* if 1 == 0 then 1 else 1* fac (1-1)
      → 2* if False then 1 else 1* fac (1-1)
      → 2* 1* fac 0
      → 2* 1* if 0 == 0 then 1 else 0* fac (0-1)
      → 2* 1* if True then 1 else 0* fac (0-1)
      → 2* 1* 1 → 2
```

Beispiel: Nichtnumerische Werte

- ▶ Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s + repeat (n-1) s
```

- ▶ Auswertung:

```
repeat 2 "hallo"
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s + repeat (n-1) s
```

- Auswertung:

```
repeat 2 "hallo\u25b2"
```

```
→ if 2 == 0 then "" else "hallo\u25b2" + repeat (2-1) "hallo\u25b2"
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s + repeat (n-1) s
```

- Auswertung:

```
repeat 2 "hallo\u20ac"  
→ if 2 == 0 then "" else "hallo\u20ac" + repeat (2-1) "hallo\u20ac"  
→ if False then "" else "hallo\u20ac" + repeat 1 "hallo\u20ac"
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- Auswertung:

```
repeat 2 "hallo\u20ac"  
→ if 2 == 0 then "" else "hallo\u20ac" ++ repeat (2-1) "hallo\u20ac"  
→ if False then "" else "hallo\u20ac" ++ repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" ++ repeat 1 "hallo\u20ac"
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s + repeat (n-1) s
```

- Auswertung:

```
repeat 2 "hallo\u20ac"  
→ if 2 == 0 then "" else "hallo\u20ac" + repeat (2-1) "hallo\u20ac"  
→ if False then "" else "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + if 1 == 0 then "" else "hallo\u20ac" + repeat (1-1) "hallo\u20ac"
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- Auswertung:

```
repeat 2 "hallo\u20ac"  
→ if 2 == 0 then "" else "hallo\u20ac" ++ repeat (2-1) "hallo\u20ac"  
→ if False then "" else "hallo\u20ac" ++ repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" ++ repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" ++ if 1 == 0 then "" else "hallo\u20ac" ++ repeat (1-1) "hallo\u20ac"  
→ "hallo\u20ac" ++ if False then "" else "hallo\u20ac" ++ repeat 1 "hallo\u20ac"
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s + repeat (n-1) s
```

- Auswertung:

```
repeat 2 "hallo\u20ac"  
→ if 2 == 0 then "" else "hallo\u20ac" + repeat (2-1) "hallo\u20ac"  
→ if False then "" else "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + if 1 == 0 then "" else "hallo\u20ac" + repeat (1-1) "hallo\u20ac"  
→ "hallo\u20ac" + if False then "" else "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + ("hallo\u20ac" + repeat 0 "hallo\u20ac")
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s ++ repeat (n-1) s
```

- Auswertung:

```
repeat 2 "hallo\u20ac"  
→ if 2 == 0 then "" else "hallo\u20ac" ++ repeat (2-1) "hallo\u20ac"  
→ if False then "" else "hallo\u20ac" ++ repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" ++ repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" ++ if 1 == 0 then "" else "hallo\u20ac" ++ repeat (1-1) "hallo\u20ac"  
→ "hallo\u20ac" ++ if False then "" else "hallo\u20ac" ++ repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" ++ ("hallo\u20ac" ++ repeat 0 "hallo\u20ac")  
→ "hallo\u20ac" ++ ("hallo\u20ac" ++ if 0 == 0 then "" else "hallo\u20ac" ++ repeat (0-1) "hallo\u20ac")
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s + repeat (n-1) s
```

- Auswertung:

```
repeat 2 "hallo\u20ac"  
→ if 2 == 0 then "" else "hallo\u20ac" + repeat (2-1) "hallo\u20ac"  
→ if False then "" else "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + if 1 == 0 then "" else "hallo\u20ac" + repeat (1-1) "hallo\u20ac"  
→ "hallo\u20ac" + if False then "" else "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + ("hallo\u20ac" + repeat 0 "hallo\u20ac")  
→ "hallo\u20ac" + ("hallo\u20ac" + if 0 == 0 then "" else "hallo\u20ac" + repeat (0-1) "hallo\u20ac")  
→ "hallo\u20ac" + ("hallo\u20ac" + if True then "" else "hallo\u20ac" + repeat (-1) "hallo\u20ac")
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s + repeat (n-1) s
```

- Auswertung:

```
repeat 2 "hallo\u20ac"  
→ if 2 == 0 then "" else "hallo\u20ac" + repeat (2-1) "hallo\u20ac"  
→ if False then "" else "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + if 1 == 0 then "" else "hallo\u20ac" + repeat (1-1) "hallo\u20ac"  
→ "hallo\u20ac" + if False then "" else "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + ("hallo\u20ac" + repeat 0 "hallo\u20ac")  
→ "hallo\u20ac" + ("hallo\u20ac" + if 0 == 0 then "" else "hallo\u20ac" + repeat (0-1) "hallo\u20ac")  
→ "hallo\u20ac" + ("hallo\u20ac" + if True then "" else "hallo\u20ac" + repeat (-1) "hallo\u20ac")  
→ "hallo\u20ac" + ("hallo\u20ac" + "")
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then "" else s + repeat (n-1) s
```

- Auswertung:

```
repeat 2 "hallo\u20ac"  
→ if 2 == 0 then "" else "hallo\u20ac" + repeat (2-1) "hallo\u20ac"  
→ if False then "" else "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + if 1 == 0 then "" else "hallo\u20ac" + repeat (1-1) "hallo\u20ac"  
→ "hallo\u20ac" + if False then "" else "hallo\u20ac" + repeat 1 "hallo\u20ac"  
→ "hallo\u20ac" + ("hallo\u20ac" + repeat 0 "hallo\u20ac")  
→ "hallo\u20ac" + ("hallo\u20ac" + if 0 == 0 then "" else "hallo\u20ac" + repeat (0-1) "hallo\u20ac")  
→ "hallo\u20ac" + ("hallo\u20ac" + if True then "" else "hallo\u20ac" + repeat (-1) "hallo\u20ac")  
→ "hallo\u20ac" + ("hallo\u20ac" + "")  
→ "hallo\u20achallo\u20ac"
```

Auswertung als Ausführungsbegri

- ▶ Programme werden durch **Gleichungen** definiert:

$$f(x) = E$$

- ▶ **Auswertung** durch **Anwenden** der Gleichungen:

- ▶ Suchen nach **Vorkommen** von f , e.g. $f(t)$

- ▶ $f(t)$ wird durch $E \begin{bmatrix} t \\ x \end{bmatrix}$ ersetzt

- ▶ Auswertung kann **divergieren**!

Ausdrücke und Werte

- ▶ Nichtreduzierbare Ausdrücke sind **Werte**
- ▶ Vorgebenere Basiswerte: Zahlen, Zeichen
 - ▶ Durch **Implementation** gegeben
- ▶ Definierte **Datentypen**: Wahrheitswerte, Listen, ...
 - ▶ Modellierung von Daten

Jetzt seit ihr dran!

Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) + "*" else ""
```

div n m ist die ganzzahlige Division: div 7 2→3

Berechnet wie oben die Reduktion von stars 5

Jetzt seit ihr dran!

Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) ++ "*" else ""
```

div n m ist die ganzzahlige Division: div 7 2 → 3

Berechnet wie oben die Reduktion von stars 5

Lösung:

stars 5 →

Jetzt seit ihr dran!

Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) + "*" else ""
```

div n m ist die ganzzahlige Division: div 7 2→3

Berechnet wie oben die Reduktion von stars 5

Lösung:

```
stars 5 → if 5 > 1 then stars (div 5 2) + "*" else ""  
→
```

Jetzt seit ihr dran!

Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) ++ "*" else ""
```

div n m ist die ganzzahlige Division: div 7 2→3

Berechnet wie oben die Reduktion von stars 5

Lösung:

```
stars 5 → if 5 > 1 then stars (div 5 2) ++ "*" else ""  
→ stars 2 ++ "*"  
→
```

Jetzt seit ihr dran!

Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) + "*" else ""
```

div n m ist die ganzzahlige Division: div 7 2→3

Berechnet wie oben die Reduktion von stars 5

Lösung:

```
stars 5 → if 5 > 1 then stars (div 5 2) + "*" else ""  
→ stars 2 + "*"  
→ (if 2 > 1 then stars (div 2 2) + "*" else "") + "*"  
→
```

Jetzt seit ihr dran!

Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) + "*" else ""
```

div n m ist die ganzzahlige Division: div 7 2→3

Berechnet wie oben die Reduktion von stars 5

Lösung:

```
stars 5 → if 5 > 1 then stars (div 5 2) + "*" else ""  
→ stars 2 + "*"  
→ (if 2 > 1 then stars (div 2 2) + "*" else "") + "*"  
→ (stars 1 + "*") + "*"  
→
```

Jetzt seit ihr dran!

Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) + "*" else ""
```

div n m ist die ganzzahlige Division: div 7 2→3

Berechnet wie oben die Reduktion von stars 5

Lösung:

```
stars 5 → if 5 > 1 then stars (div 5 2) + "*" else ""  
→ stars 2 + "*"  
→ (if 2 > 1 then stars (div 2 2) + "*" else "") + "*"  
→ (stars 1 + "*") + "*"  
→ ((if 1 > 1 then stars (div 1 2) + "*" else "") + "*") + "*"  
→
```

Jetzt seit ihr dran!

Übung 1.1: Auswertung

Hier ist eine weitere Beispiel-Funktion:

```
stars n = if n > 1 then stars (div n 2) + "*" else ""
```

div n m ist die ganzzahlige Division: div 7 2 → 3

Berechnet wie oben die Reduktion von stars 5

Lösung:

```
stars 5 → if 5 > 1 then stars (div 5 2) + "*" else ""  
→ stars 2 + "*"  
→ (if 2 > 1 then stars (div 2 2) + "*" else "") + "*"  
→ (stars 1 + "*") + "*"  
→ ((if 1 > 1 then stars (div 1 2) + "*" else "") + "*") + "*"  
→ ("" + "*") + "*" → "**"
```

III. Typen

Typisierung

- ▶ **Typen** unterscheiden Arten von Ausdrücken und Werten:

repeat *n* *s* = ... *n* Zahl
 s Zeichenkette

- ▶ **Wozu** Typen?

- ▶ Frühzeitiges Aufdecken “offensichtlicher” Fehler
- ▶ Erhöhte Programmsicherheit
- ▶ Hilfestellung bei Änderungen

Slogan

“Well-typed programs can’t go wrong.”

— Robin Milner

Signaturen

- ▶ Jede Funktion hat eine **Signatur**

```
fac      :: Int → Int
```

```
repeat  :: Int → String → String
```

- ▶ **Typüberprüfung**

- ▶ `fac` nur auf `Int` anwendbar, Resultat ist `Int`
- ▶ `repeat` nur auf `Int` und `String` anwendbar, Resultat ist `String`

Übersicht: Typen in Haskell

Typ	Bezeichner	Beispiel		
Ganze Zahlen	Int	0	94	-45
Fließkomma	Double	3.0	3.141592	
Zeichen	Char	'a'	'x'	'\034'
Zeichenketten	String	"yuck"	"hi\nho\"\\n"	
Wahrheitswerte	Bool	True	False	
Funktionen		$a \rightarrow b$		

► Später mehr. **Viel** mehr.

Das Rechnen mit Zahlen

Beschränkte Genauigkeit,
konstanter Aufwand

beliebige Genauigkeit,
wachsender Aufwand



Das Rechnen mit Zahlen

Beschränkte Genauigkeit,
konstanter Aufwand

beliebige Genauigkeit,
wachsender Aufwand

Haskell bietet die Auswahl:

- ▶ **Int** - ganze Zahlen als Maschinenworte (≥ 31 Bit)
- ▶ **Integer** - beliebig große ganze Zahlen
- ▶ **Rational** - beliebig genaue rationale Zahlen
- ▶ **Float, Double** - Fließkommazahlen (reelle Zahlen)

Ganze Zahlen: Int und Integer

- Nützliche Funktionen (**überladen**, auch für Integer):

```
+, *, ^, - :: Int → Int → Int
abs           :: Int → Int — Betrag
div, quot    :: Int → Int → Int
mod, rem     :: Int → Int → Int
```

Es gilt: $(\text{div } x \text{ } y) * y + \text{mod } x \text{ } y = x$

- Vergleich durch $=, \neq, \leq, <, \dots$

- **Achtung:** Unäres Minus

- Unterschied zum Infix-Operator $-$
- Im Zweifelsfall klammern: `abs (-34)`

Fließkommazahlen: Double

- ▶ Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
 - ▶ Logarithmen, Wurzel, Exponentiation, π und e , trigonometrische Funktionen
- ▶ Konversion in ganze Zahlen:
 - ▶ `fromIntegral :: Int, Integer → Double`
 - ▶ `fromInteger :: Integer → Double`
 - ▶ `round, truncate :: Double → Int, Integer`
- ▶ Überladungen mit Typannotation auflösen:

```
round (fromInt 10) :: Int
```
- ▶ **Rundungsfehler!**

Alphanumerische Basisdatentypen: Char

- ▶ Notation für einzelne **Zeichen**: 'a',...
- ▶ Nützliche **Funktionen**:

```
ord :: Char → Int
```

```
chr :: Int → Char
```

```
toLowerCase :: Char → Char
```

```
toUpperCase :: Char → Char
```

```
isDigit :: Char → Bool
```

```
isAlpha :: Char → Bool
```

- ▶ Zeichenketten: String



Jetzt seit ihr noch mal dran.

- ▶ ZIP-Datei mit den Quellen auf der Webseite verlinkt (Rubrik *Vorlesung*)
- ▶ Für diese Vorlesung: eine Datei Examples.hs mit den Quellen der Funktionen `fac`, `repeat` und `start`.
- ▶ Unter der Rubrik *Übung*: Kurzanleitung PI3-Übungsbetrieb
- ▶ Durchlesen und Haskell Tool Stack installieren, Experimente ausprobieren, 0. übungsblatt angehen.

Übung 1.2: Mehr Sterne

Ändert die Funktion `stars` so ab, dass sie eine Zeichenkette aus `n` Sternchen zurückgibt.

Zusammenfassung

- ▶ Programme sind **Funktionen**, definiert durch **Gleichungen**
 - ▶ Referentielle Transparenz
 - ▶ kein impliziter Zustand, keine veränderlichen Variablen
- ▶ **Ausführung** durch **Reduktion** von Ausdrücken
- ▶ Typisierung:
 - ▶ Basistypen: Zahlen, Zeichen(ketten), Wahrheitswerte
 - ▶ Jede Funktion f hat eine Signatur $f :: a \rightarrow b$

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 2 vom 09.11.2020: Funktionen

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

- ▶ **Teil I: Funktionale Programmierung im Kleinen**
 - ▶ Einführung
 - ▶ **Funktionen**
 - ▶ Algebraische Datentypen
 - ▶ Typvariablen und Polymorphie
 - ▶ Funktionen höherer Ordnung I
 - ▶ Rekursive und zyklische Datenstrukturen
 - ▶ Funktionen höherer Ordnung II
- ▶ **Teil II: Funktionale Programmierung im Großen**
- ▶ **Teil III: Funktionale Programmierung im richtigen Leben**

Inhalt und Lernziele

- ▶ Definition von **Funktionen**
 - ▶ Syntaktische Feinheiten
- ▶ Bedeutung von Haskell-Programmen
 - ▶ Striktheit
- ▶ Leben ohne Variablen
 - ▶ Funktionen statt Schleifen
 - ▶ Zahllose Beispiele

Lernziele

Wir wollen einfache Haskell-Programme schreiben können, eine Idee von ihrer Bedeutung bekommen, und ein Leben ohne veränderliche Variablen führen.

I. Definition von Funktionen

Definition von Funktionen

- ▶ Zwei wesentliche Konstrukte:
 - ▶ Fallunterscheidung
 - ▶ Rekursion

Definition von Funktionen

- ▶ Zwei wesentliche Konstrukte:
 - ▶ Fallunterscheidung
 - ▶ Rekursion

Satz

Fallunterscheidung und Rekursion auf natürlichen Zahlen sind **Turing-mächtig**.

- ▶ Funktionen müssen **partiell** sein können.
 - ▶ Insbesondere nicht-terminierende Rekursion
- ▶ Fragen: wie schreiben Funktionen in Haskell auf (**Syntax**), und was bedeutet das (**Semantik**)?

Haskell-Syntax: Charakteristika

► Leichtgewichtig

- Wichtigstes Zeichen:
- Funktionsapplikation: `f a`
- Klammern sind optional
- **Höchste** Priorität (engste Bindung)
- Abseitsregel: Gültigkeitsbereich durch Einrückung
 - Keine Klammern (`{ ... }`) (optional)
 - Auch in anderen Sprachen (Python, Ruby)

Haskell-Syntax: Funktionsdefinition

Generelle Form:

► **Signatur:**

```
max :: Int → Int → Int
```

► **Definition:**

```
max x y = if x < y then y else x
```

- Kopf, mit Parametern
- Rumpf (evtl. länger, mehrere Zeilen)
- Typisches Muster: Fallunterscheidung, dann rekursiver Aufruf
- Was gehört zum Rumpf (Geltungsbereich)?

Haskell-Syntax I: Die Abseitsregel

Funktionsdefinition:

```
f x1 x2 x3...xn = e
```

- ▶ **Gültigkeitsbereich** der Definition von `f`:
alles, was gegenüber `f` eingerückt ist.
- ▶ Beispiel:

```
f x = hier faengts an
      und hier gehts weiter
          immer weiter
g y z = und hier faengt was neues an
```

- ▶ Gilt auch verschachtelt.
- ▶ Kommentare sind passiv (heben das Abseits nicht auf).

Haskell-Syntax II: Kommentare

- ▶ Pro Zeile: Ab `---` bis Ende der Zeile

```
f x y = irgendwas  — und hier der Kommentar!
```

- ▶ Über mehrere Zeilen: Anfang `{—`, Ende `—}`

```
{—  
    Hier faengt der Kommentar an  
    erstreckt sich ueber mehrere Zeilen  
    bis hier  
    —}  
f x y = irgendwas
```

- ▶ Kann geschachtelt werden.

Haskell-Syntax III: Bedingte Definitionen

- ▶ Statt verschachtelter Fallunterscheidungen . . .

```
f x y = if B1 then P else  
         if B2 then Q else R
```

... **bedingte Gleichungen**:

```
f x y  
| B1 = P  
| B2 = Q
```

- ▶ Auswertung der Bedingungen von oben nach unten
- ▶ Wenn keine Bedingung wahr ist: **Laufzeitfehler**! Deshalb:
| `otherwise = R`

Haskell-Syntax IV: Lokale Definitionen

- Lokale Definitionen mit `where` oder `let`:

```
f x y
| g = P y
| otherwise = f x where
|   y = M
|   f x = N x
```

```
f x y =
let y = M
  f x = N x
in  if g then P y
      else f x
```

- `f`, `y`, ... werden **gleichzeitig** definiert (Rekursion!)
- Namen `f`, `y` und Parameter (`x`) **überlagern** andere
- Es gilt die **Abseitsregel**
 - Deshalb: Auf gleiche Einrückung der lokalen Definition achten!

Jetzt seit ihr dran!

Übung 2.1: Syntax

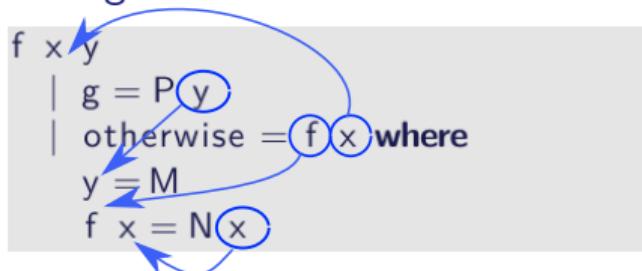
In dem Beispielprogramm auf der vorherigen Folie, welche der Variablen `f`, `x` und `y` auf den rechten Seiten wird wo gebunden?

Jetzt seit ihr dran!

Übung 2.1: Syntax

In dem Beispielprogramm auf der vorherigen Folie, welche der Variablen `f`, `x` und `y` auf den rechten Seiten wird wo gebunden?

Lösung:



II. Auswertung von Funktionen

Auswertung von Funktionen

- ▶ Auswertung durch **Anwendung** von Gleichungen
- ▶ **Auswertungsrelation** $s \rightarrow t$:
 - ▶ Anwendung einer Funktionsdefinition
 - ▶ Anwendung von elementaren Operationen (arithmetisch, Zeichenketten)
- ▶ Frage: spielt die **Reihenfolge** eine Rolle?

Auswertung von Ausdrücken

```
inc :: Int → Int
```

```
inc x = x+ 1
```

```
dbl :: Int → Int
```

```
dbl x = 2*x
```

- ▶ Reduktion von `inc (dbl (inc 3))`

Auswertung von Ausdrücken

```
inc :: Int → Int
```

```
inc x = x+ 1
```

```
dbl :: Int → Int
```

```
dbl x = 2*x
```

- ▶ Reduktion von `inc (dbl (inc 3))`
- ▶ Von **außen** nach **innen** (outermost-first):
`inc (dbl (inc 3))`

Auswertung von Ausdrücken

```
inc :: Int → Int
```

```
inc x = x+ 1
```

```
dbl :: Int → Int
```

```
dbl x = 2*x
```

- ▶ Reduktion von `inc (dbl (inc 3))`
- ▶ Von **außen** nach **innen** (outermost-first):

$$\text{inc} (\text{dbl} (\text{inc} 3)) \rightarrow \text{dbl} (\text{inc} 3) + 1$$

Auswertung von Ausdrücken

inc :: Int → Int

inc x = x+ 1

dbl :: Int → Int

dbl x = 2*x

- ▶ Reduktion von inc (dbl (inc 3))
- ▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{dbl (inc 3)+ 1} \\ &\rightarrow 2*(\text{inc 3})+ 1 \end{aligned}$$

Auswertung von Ausdrücken

inc :: Int → Int

inc x = x+ 1

dbl :: Int → Int

dbl x = 2*x

- ▶ Reduktion von inc (dbl (inc 3))
- ▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{dbl (inc 3)+ 1} \\ &\rightarrow 2*(\text{inc 3})+ 1 \\ &\rightarrow 2*(3+ 1)+ 1 \rightarrow 2*4+1 \rightarrow 8+1 \rightarrow 9 \end{aligned}$$

Auswertung von Ausdrücken

```
inc :: Int → Int  
inc x = x+ 1
```

```
dbl :: Int → Int  
dbl x = 2*x
```

- ▶ Reduktion von `inc (dbl (inc 3))`
- ▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{dbl (inc 3)+ 1} \\ &\rightarrow 2*(\text{inc 3})+ 1 \\ &\rightarrow 2*(3+ 1)+ 1 \rightarrow 2*4+1 \rightarrow 8+1 \rightarrow 9 \end{aligned}$$

- ▶ Von **innen** nach **außen** (innermost-first):
- `inc (dbl (inc 3))`

Auswertung von Ausdrücken

```
inc :: Int → Int  
inc x = x+ 1
```

```
dbl :: Int → Int  
dbl x = 2*x
```

- ▶ Reduktion von `inc (dbl (inc 3))`
- ▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{dbl (inc 3)+ 1} \\ &\rightarrow 2*(\text{inc 3})+ 1 \\ &\rightarrow 2*(3+ 1)+ 1 \rightarrow 2*4+1 \rightarrow 8+1 \rightarrow 9 \end{aligned}$$

- ▶ Von **innen** nach **außen** (innermost-first):
- $$\text{inc (dbl (inc 3))} \rightarrow \text{inc (dbl (3+1))}$$

Auswertung von Ausdrücken

```
inc :: Int → Int
```

```
inc x = x+ 1
```

```
dbl :: Int → Int
```

```
dbl x = 2*x
```

► Reduktion von `inc (dbl (inc 3))`

► Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{dbl (inc 3)+ 1} \\ &\rightarrow 2*(\text{inc 3})+ 1 \\ &\rightarrow 2*(3+ 1)+ 1 \rightarrow 2*4+1 \rightarrow 8+1 \rightarrow 9 \end{aligned}$$

► Von **innen** nach **außen** (innermost-first):

$$\text{inc (dbl (inc 3))} \rightarrow \text{inc (dbl (3+1))} \rightarrow \text{inc (dbl 4)}$$

Auswertung von Ausdrücken

```
inc :: Int → Int
```

```
inc x = x+ 1
```

```
dbl :: Int → Int
```

```
dbl x = 2*x
```

► Reduktion von `inc (dbl (inc 3))`

► Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{dbl (inc 3)+ 1} \\ &\rightarrow 2*(\text{inc 3})+ 1 \\ &\rightarrow 2*(3+ 1)+ 1 \rightarrow 2*4+1 \rightarrow 8+1 \rightarrow 9 \end{aligned}$$

► Von **innen** nach **außen** (innermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{inc (dbl (3+1))} \rightarrow \text{inc (dbl 4)} \\ &\rightarrow \text{inc (2*4)} \end{aligned}$$

Auswertung von Ausdrücken

```
inc :: Int → Int  
inc x = x+ 1
```

```
dbl :: Int → Int  
dbl x = 2*x
```

- ▶ Reduktion von `inc (dbl (inc 3))`
- ▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{dbl (inc 3)+ 1} \\ &\rightarrow 2*(\text{inc 3})+ 1 \\ &\rightarrow 2*(3+ 1)+ 1 \rightarrow 2*4+1 \rightarrow 8+1 \rightarrow 9 \end{aligned}$$

- ▶ Von **innen** nach **außen** (innermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{inc (dbl (3+1))} \rightarrow \text{inc (dbl 4)} \\ &\rightarrow \text{inc (2*4)} \rightarrow \text{inc 8} \end{aligned}$$

Auswertung von Ausdrücken

```
inc :: Int → Int
```

```
inc x = x+ 1
```

```
dbl :: Int → Int
```

```
dbl x = 2*x
```

- ▶ Reduktion von `inc (dbl (inc 3))`

- ▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{dbl (inc 3)+ 1} \\ &\rightarrow 2*(\text{inc 3})+ 1 \\ &\rightarrow 2*(3+ 1)+ 1 \rightarrow 2*4+1 \rightarrow 8+1 \rightarrow 9 \end{aligned}$$

- ▶ Von **innen** nach **außen** (innermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{inc (dbl (3+1))} \rightarrow \text{inc (dbl 4)} \\ &\rightarrow \text{inc (2*4)} \rightarrow \text{inc 8} \\ &\rightarrow 8+1 \end{aligned}$$

Auswertung von Ausdrücken

```
inc :: Int → Int  
inc x = x+ 1
```

```
dbl :: Int → Int  
dbl x = 2*x
```

- ▶ Reduktion von `inc (dbl (inc 3))`
- ▶ Von **außen** nach **innen** (outermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{dbl (inc 3)+ 1} \\ &\rightarrow 2*(\text{inc 3})+ 1 \\ &\rightarrow 2*(3+ 1)+ 1 \rightarrow 2*4+1 \rightarrow 8+1 \rightarrow 9 \end{aligned}$$

- ▶ Von **innen** nach **außen** (innermost-first):

$$\begin{aligned} \text{inc (dbl (inc 3))} &\rightarrow \text{inc (dbl (3+1))} \rightarrow \text{inc (dbl 4)} \\ &\rightarrow \text{inc (2*4)} \rightarrow \text{inc 8} \\ &\rightarrow 8+1 \rightarrow 9 \end{aligned}$$

Auswertung von Ausdrücken

```
inc :: Int → Int
inc x = x+ 1
```

```
dbl :: Int → Int
dbl x = 2*x
```

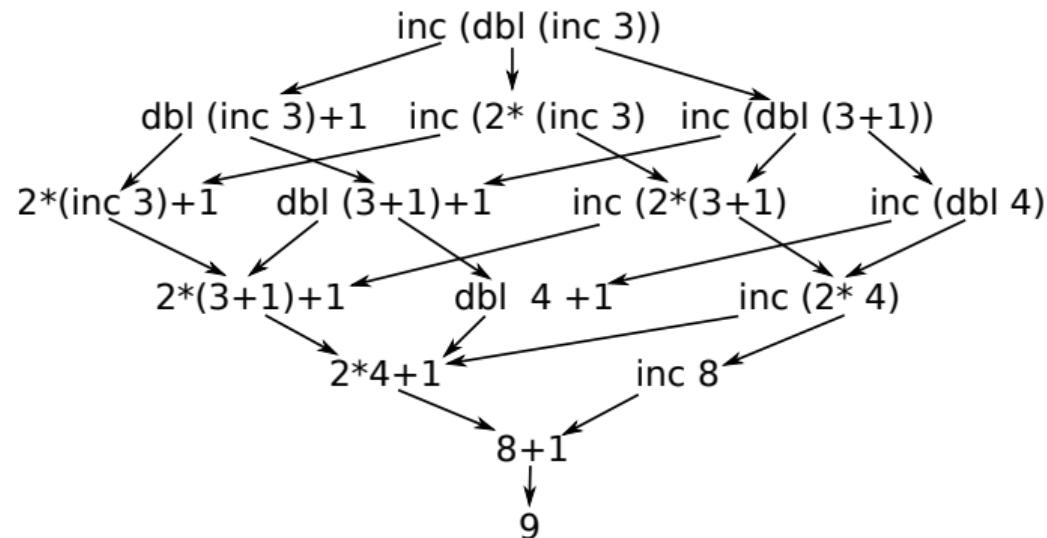
- ▶ Volle Reduktion von `inc (dbl (inc 3))`:

Auswertung von Ausdrücken

```
inc :: Int → Int  
inc x = x+ 1
```

```
dbl :: Int → Int  
dbl x = 2*x
```

- Volle Reduktion von `inc (dbl (inc 3))`:



Konfluenz

- ▶ Es kommt immer das gleiche heraus?
- ▶ Sei $\xrightarrow{*}$ die Reduktion in null oder mehr Schritten.

Definition (Konfluenz)

$\xrightarrow{*}$ ist **konfluent** gdw:

Für alle r, s, t mit $s \xleftarrow{*} r \xrightarrow{*} t$ gibt es u so dass $s \xrightarrow{*} u \xleftarrow{*} t$.

Konfluenz

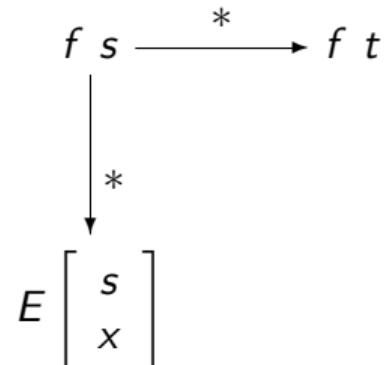
- Wenn wir von Laufzeitfehlern abstrahieren, gilt:

Theorem (Konfluenz)

Die Auswertungsrelation $\xrightarrow{*}$ für funktionale Programme ist **konfluent**.

- Beweisskizze:

Sei $f \ x = E$ und $s \xrightarrow{*} t$:



Konfluenz

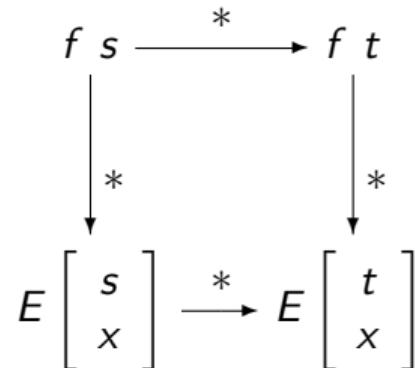
- Wenn wir von Laufzeitfehlern abstrahieren, gilt:

Theorem (Konfluenz)

Die Auswertungsrelation $\xrightarrow{*}$ für funktionale Programme ist **konfluent**.

- Beweisskizze:

Sei $f \ x = E$ und $s \xrightarrow{*} t$:



Auswirkung der Auswertungsstrategie

- Auswertungsstrategie ist also egal?

Auswirkung der Auswertungsstrategie

- ▶ Auswertungsstrategie ist also egal?
- ▶ Beispiel:

```
repeat :: Int → String → String
repeat n s = if n == 0 then ""
             else s ++ repeat (n-1) s
```

```
undef :: String
undef = undef
```

- ▶ Auswertung von `repeat 0 undef`:

Auswirkung der Auswertungsstrategie

- ▶ Auswertungsstrategie ist also egal?
- ▶ Beispiel:

```
repeat :: Int → String → String
repeat n s = if n == 0 then ""
             else s ++ repeat (n-1) s
```

```
undef :: String
undef = undef
```

- ▶ Auswertung von `repeat 0 undef`:

```
repeat 0 undef
```

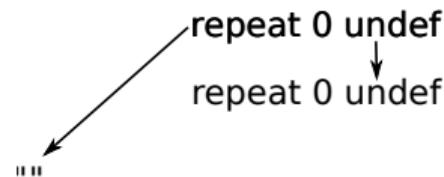
Auswirkung der Auswertungsstrategie

- ▶ Auswertungsstrategie ist also egal?
- ▶ Beispiel:

```
repeat :: Int → String → String
repeat n s = if n == 0 then ""
             else s ++ repeat (n-1) s
```

```
undef :: String
undef = undef
```

- ▶ Auswertung von `repeat 0 undef`:



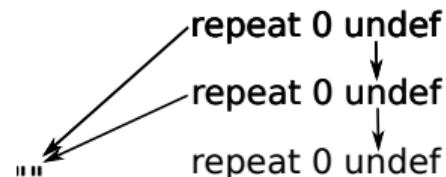
Auswirkung der Auswertungsstrategie

- ▶ Auswertungsstrategie ist also egal?
- ▶ Beispiel:

```
repeat :: Int → String → String
repeat n s = if n == 0 then ""
             else s ++ repeat (n-1) s
```

```
undef :: String
undef = undef
```

- ▶ Auswertung von `repeat 0 undef`:



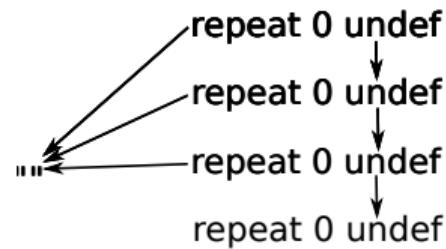
Auswirkung der Auswertungsstrategie

- ▶ Auswertungsstrategie ist also egal?
- ▶ Beispiel:

```
repeat :: Int → String → String
repeat n s = if n == 0 then ""
             else s ++ repeat (n-1) s
```

```
undef :: String
undef = undef
```

- ▶ Auswertung von `repeat 0 undef`:



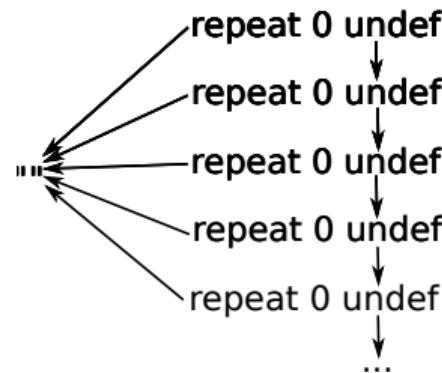
Auswirkung der Auswertungsstrategie

- ▶ Auswertungsstrategie ist also egal?
- ▶ Beispiel:

```
repeat :: Int → String → String
repeat n s = if n == 0 then ""
             else s ++ repeat (n-1) s
```

```
undef :: String
undef = undef
```

- ▶ Auswertung von `repeat 0 undef`:



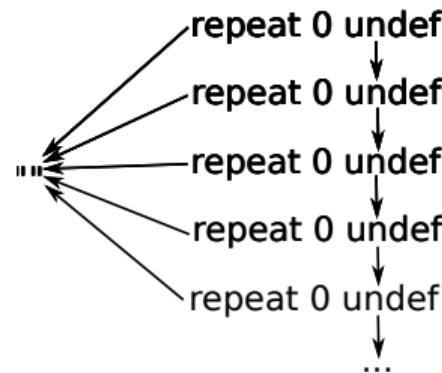
Auswirkung der Auswertungsstrategie

- ▶ Auswertungsstrategie ist also egal?
- ▶ Beispiel:

```
repeat :: Int → String → String
repeat n s = if n == 0 then ""
             else s ++ repeat (n-1) s
```

```
undef :: String
undef = undef
```

- ▶ Auswertung von `repeat 0 undef`:



- ▶ outermost-first **terminiert**
- ▶ innermost-first terminiert **nicht**

Termination und Normalform

Definition (Termination)

→ ist **terminierend** gdw. es **keine unendlichen** Ketten gibt:

$$t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \dots t_n \rightarrow \dots$$

Theorem (Normalform)

Sei \rightarrow^* konfluent und terminierend, dann wertet jeder Term zu genau einer **Normalform** aus, die nicht weiter ausgewertet werden kann.

- Daraus folgt: **terminierende** funktionale Programme werten unter jeder Auswertungsstrategie jeden Ausdruck zum gleichen Wert aus (der Normalform).

Auswirkung der Auswertungsstrategie

- ▶ Auswertungsstrategie nur für **nicht-terminierende** Programme relevant.
- ▶ Leider ist nicht-Termination **nötig** (Turing-Mächtigkeit)
- ▶ Gibt es eine **semantische** Charakterisierung?
- ▶ Auswertungsstrategie und Parameterübergabe:
 - ▶ Outermost-first entspricht **call-by-need**, verzögerte Auswertung.
 - ▶ Innermost-first entspricht **call-by-value**, strikte Auswertung

Zum Mitdenken...

Übung 2.2:

Warum entspricht outermost-first call-ny-need und innermost-first call-by-value?

Zum Mitdenken...

Übung 2.2:

Warum entspricht outermost-first call-ny-need und innermost-first call-by-value?

Lösung: Der Aufruf einer Funktion $f\ x = E$ entspricht hier der Ersetzung der linken Seite f durch die rechte Seite E , mit den Parametern x entsprechend ersetzt.

Wenn wir beispielsweise Auswertung des Ausdrucks $dbl\ (dbl\ (dbl\ (7+3)))$ betrachten, dann wird innermost-first zuerst $7+3$ reduziert, dann $dbl\ 10$ etc, d.h. jeweils die **Argumente** der Funktion — Funktionen bekommen nur Werte übergeben.

Bei outermost-first wird zuerst das äußerste dbl reduziert, was dem Aufruf der Funktion dbl mit dem nicht ausgewerteten Argument $dbl\ (dbl\ (7+3))$ entspricht (verzögerte Auswertung).

III. Semantik und Striktheit

Bedeutung (Semantik) von Programmen

► **Operationale** Semantik:

- Durch den **Ausführungs**begriff
- Ein Programm **ist**, was es **tut**.
- In diesem Fall: →

► **Denotationelle** Semantik:

- Programme werden auf **mathematische Objekte** abgebildet (Denotat).
- Für funktionale Programme: **rekursiv** definierte Funktionen

Äquivalenz von operationaler und denotationaler Semantik

Sei P ein funktionales Programm, $\xrightarrow{*}$ die dadurch definierte Reduktion, und $\llbracket P \rrbracket$ das Denotat. Dann gilt für alle Ausdrücke t und Werte v

$$t \xrightarrow{*} v \iff \llbracket P \rrbracket(t) = v$$

Definition (Striktheit)

Funktion f ist **strikt** \iff Ergebnis ist undefiniert, sobald ein Argument undefiniert ist.

- ▶ **Denotationelle** Eigenschaft (nicht operational)
- ▶ Haskell ist nach **Sprachdefinition nicht-strikt**
 - ▶ `repeat 0 undef` **muss** "" ergeben.
 - ▶ Meisten **Implementationen** nutzen **verzögerte Auswertung**
- ▶ Andere Programmiersprachen:
 - ▶ Java, C, etc. sind **call-by-value** (nach Sprachdefinition) und damit **strikt**
 - ▶ Fallunterscheidung ist **immer** nicht-strikt, Konjunktion und Disjunktion meist auch.

Jetzt seit ihr dran!

Übung 2.3: Strikte Fallunterscheidung

Warum ist Fallunterscheidung immer nicht-strikt, auch in Java?

Jetzt seit ihr dran!

Übung 2.3: Strikte Fallunterscheidung

Warum ist Fallunterscheidung immer nicht-strikt, auch in Java?

Lösung: Betrachte

```
y = x == 0 ? -1 : 100/x;
```

```
if (x == 0) {  
    y= -1;  
} else {  
    y= 100/x;  
}
```

Wäre die Fallunterscheidung strikt, würden erst **beide** Fälle ausgewertet; es wäre nicht mehr möglich, die Auswertung undefinierter Ausdrücke abzufangen. Das gleich gilt für das Programm rechts.

IV. Leben ohne Variablen

Rekursion statt Schleifen

Fakultät imperativ:

```
r= 1;  
while (n > 0) {  
    r= n* r;  
    n= n- 1;  
}
```

- ▶ Veränderliche Variablen werden zu Funktionsparametern
- ▶ Iteration (while-Schleifen) werden zu Rekursion
- ▶ Endrekursion verbraucht keinen Speicherplatz

Rekursion statt Schleifen

Fakultät imperativ:

```
r = 1;  
while (n > 0) {  
    r = n * r;  
    n = n - 1;  
}
```

Fakultät rekursiv:

```
fac' n r =  
    if n ≤ 0 then r  
    else fac' (n-1) (n*r)  
  
fac n = fac' n 1
```

- ▶ Veränderliche Variablen werden zu Funktionsparametern
- ▶ Iteration (while-Schleifen) werden zu Rekursion
- ▶ Endrekursion verbraucht keinen Speicherplatz

Rekursive Funktionen auf Zeichenketten

- ▶ Test auf die leere Zeichenkette:

```
null :: String → Bool  
null xs = xs == ""
```

- ▶ Kopf und Rest einer nicht-leeren Zeichenkette (vordefiniert):

```
head :: String → Char  
tail :: String → String
```



Suche in einer Zeichenkette

- ▶ Suche nach einem Zeichen in einer Zeichenkette:

```
count1 :: Char → String → Int
```

- ▶ In einem leeren String: kein Zeichen kommt vor
- ▶ Ansonsten: Kopf vergleichen, zum Vorkommen im Rest addieren

```
count1 c s =  
  if null s then 0  
  else if head s == c then 1 + count1 c (tail s)  
                        else count1 c (tail s)
```

- ▶ Übung: wie formuliere ich `count` mit Guards? (Lösung in den Quellen)

Suche in einer Zeichenkette

- Endrekursiv:

```
count3 c s = count3' c s 0
count3' c s r =
  if null s then r
  else count3' c (tail s) (if head s == c then 1+r else r)
```

Suche in einer Zeichenkette

- Endrekursiv:

```
count3 c s = count3' c s 0
count3' c s r =
  if null s then r
  else count3' c (tail s) (if head s == c then 1+r else r)
```

- Endrekursiv mit lokaler Definition

```
count4 c s = count4' s 0 where
count4' s r =
  if null s then r
  else count4' (tail s) (if head s == c then 1+r else r)
```



Strings konstruieren

- `:` hängt Zeichen vorne an Zeichenkette an (vordefiniert)

```
(::) :: Char → String → String
```

- Es gilt: Wenn `not (null s)`, dann `head s : tail s == s`

- Mit `(::)` wird `(++)` definiert:

```
(++) :: String → String → String
```

```
xs ++ ys
```

```
| null xs = ys
```

```
| otherwise = head xs : (tail xs ++ ys)
```

- `quadrat` konstruiert ein Quadrat aus Zeichen:

```
quadrat :: Int → Char → String
```

```
quadrat n c = repeat n (repeat n (c: "") ++ "\n")
```



Strings analysieren

- ▶ Warum immer nur Kopf/Rest?
- ▶ Letztes Zeichen (dual zu `head`):

```
last1 :: String → Char
last1 s = if null s then last1 s
          else if null (tail s) then head s
          else last1 (tail s)
```

- ▶ Besser: mit Fehlermeldung

```
last :: String → Char
last s
| null s = error "last: empty string"
| null (tail s) = head s
| otherwise      = last (tail s)
```

Strings analysieren

- ▶ Anfang der Zeichenkette (dual zu `tail`):

```
init :: String → String
init s
| null s = error "init: empty string"      — nicht s
| null (tail s) = ""
| otherwise      = head s : init (tail s)
```

- ▶ Damit: Wenn `not (null s)`, dann `init s ++ (last s: "") = s`

Strings analysieren: das Palindrom

- ▶ Palindrom: vorwärts und rückwärts gelesen gleich.
- ▶ Rekursiv:
 - ▶ Alle Wörter der Länge 1 oder kleiner sind Palindrome
 - ▶ Für längere Wörter: wenn erstes und letztes Zeichen gleich sind und der Rest ein Palindrom.
- ▶ Erste Variante:

```
palin1 :: String → Bool
palin1 s
| length s ≤ 1      = True
| head s == last s = palin1 (init (tail s))
| otherwise          = False
```



Strings analysieren: das Palindrom

- ▶ Problem: Groß/Kleinschreibung, Leerzeichen, Satzzeichen irrelevant.
- ▶ Daher: nicht-alphanumerische Zeichen entfernen, alles Kleinschrift:

```
clean :: String → String
clean s
| null s = ""
| isAlphaNum (head s) = toLower (head s) : clean (tail s)
| otherwise = clean (tail s)
```

- ▶ Erweiterte Version:

```
palin2 s = palin1 (clean s)
```



Fortgeschritten: Vereinfachung von palin1

- Das hier ist nicht so schön:

```
palin1 s
  | length s ≤ 1      = True
  | head s == last s = palin1 (init (tail s))
  | otherwise          = False
```

- Was steht da eigentlich:

```
palin1' s = if length s ≤ 1 then True
            else if head s == last s then palin1' (init (tail s))
                                         else False
```

- Damit:

```
palin3 s = length s ≤ 1 || head s == last s && palin3 (init (tail s))
```

- Terminiert nur wegen Nicht-Striktheit von ||

Zusammenfassung

- ▶ **Bedeutung** von Haskell-Programmen:
 - ▶ Auswertungsrelation →
 - ▶ Auswertungsstrategien: innermost-first, outermost-first
 - ▶ Auswertungsstrategie für terminierende Programme irrelevant
- ▶ **Striktheit**
 - ▶ Haskell ist **spezifiziert** als nicht-strikt
 - ▶ Meist implementiert durch verzögerte Auswertung
- ▶ Leben **ohne Variablen**:
 - ▶ Rekursion statt Schleifen
 - ▶ Funktionsparameter statt Variablen
- ▶ Nächste Vorlesung: Datentypen

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 3 vom 16.11.2020: Algebraische Datentypen

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Fahrplan

- ▶ **Teil I: Funktionale Programmierung im Kleinen**
 - ▶ Einführung
 - ▶ Funktionen
 - ▶ Algebraische Datentypen
 - ▶ Typvariablen und Polymorphie
 - ▶ Funktionen höherer Ordnung I
 - ▶ Rekursive und zyklische Datenstrukturen
 - ▶ Funktionen höherer Ordnung II
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Inhalt und Lernziele

► Algebraische Datentypen:

- Aufzählungen
- Produkte
- Rekursive Datentypen

Lernziel

Wir wissen, was algebraische Datentypen sind. Wir können mit ihnen modellieren, wir kennen ihre Eigenschaften, und können auf ihnen Funktionen definieren.

I. Datentypen

Warum Datentypen?

- ▶ Immer nur `Int` ist auch langweilig ...

Warum Datentypen?

- ▶ Immer nur `Int` ist auch langweilig ...
- ▶ **Abstraktion:**
- ▶ *Bool* statt `Int`, Namen statt RGB-Codes, ...

Warum Datentypen?

- ▶ Immer nur `Int` ist auch langweilig ...
- ▶ **Abstraktion:**
- ▶ *Bool* statt `Int`, Namen statt RGB-Codes, ...
- ▶ **Bessere** Programme (verständlicher und wartbarer)

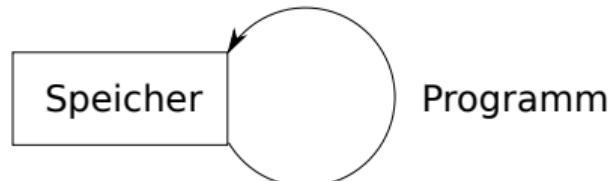
Warum Datentypen?

- ▶ Immer nur `Int` ist auch langweilig ...
- ▶ **Abstraktion:**
- ▶ *Bool* statt `Int`, Namen statt RGB-Codes, ...
- ▶ **Bessere** Programme (verständlicher und wartbarer)
- ▶ Datentypen haben **wohlverstandene algebraische Eigenschaften**

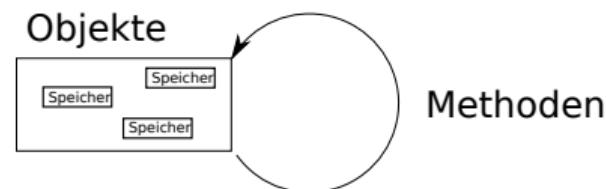
Datentypen als Modellierungskonstrukt

Programme **manipulieren** ein **Modell** der Umwelt:

- ▶ Imperative Sicht:



- ▶ Objektorientierte Sicht:



- ▶ Funktionale Sicht:



Das Modell besteht aus Datentypen.

Beispiel: Uncle Bob's Auld-Time Grocery Shoppe



Ein Tante-Emma Laden wie in früheren Zeiten.

Beispiel: Uncle Bob's Auld-Time Grocery Shoppe

Äpfel	Boskoop	55	ct/Stk
	Cox Orange	60	ct/Stk
	Granny Smith	50	ct/Stk
Eier		20	ct/Stk
Käse	Gouda	14,50	€/kg
	Appenzeller	22.70	€/kg
Schinken		1.99	€/100 g
Salami		1.59	€/100 g
Milch		0.69	€/l
	Bio	1.19	€/l

Aufzählungen

- ▶ Aufzählungen: Menge von **disjunkten** Konstanten

$$Apfel = \{Boskoop, Cox, Smith\}$$

$$Boskoop \neq Cox, Cox \neq Smith, Boskoop \neq Smith$$

- ▶ Genau drei **unterschiedliche** Konstanten
- ▶ Funktion mit **Definitionsbereich** $Apfel$ muss drei Fälle unterscheiden
- ▶ Beispiel: $preis : Apfel \rightarrow \mathbb{N}$ mit

$$preis(a) = \begin{cases} 55 & a = Boskoop \\ 60 & a = Cox \\ 50 & a = Smith \end{cases}$$

Aufzählung und Fallunterscheidung in Haskell

► Definition

```
data Apfelsorte = Boskoop | CoxOrange | GrannySmith
```

- Implizite Deklaration der **Konstruktoren** Boskoop :: Apfelsorte als Konstanten
 - **Großschreibung** der Konstruktoren und Typen
- ## ► Fallunterscheidung:

```
apreis :: Apfelsorte → Int
apreis a = case a of
  Boskoop → 55
  CoxOrange → 60
  GrannySmith → 50
```

Aufzählung und Fallunterscheidung in Haskell

► Definition

```
data Apfelsorte = Boskoop | CoxOrange | GrannySmith
```

- Implizite Deklaration der **Konstruktoren** Boskoop :: Apfelsorte als Konstanten
 - **Großschreibung** der Konstruktoren und Typen
- **Fallunterscheidung:**

```
apreis :: Apfelsorte → Int
apreis a = case a of
  Boskoop → 55
  CoxOrange → 60
  GrannySmith → 50
```

```
data Farbe = Rot | Gruen
farbe :: Apfelsorte → Farbe
farbe d =
  case d of
    GrannySmith → Gruen
    _ → Rot
```

Fallunterscheidung in der Funktionsdefinition

- Abkürzende Schreibweisen (**syntaktischer Zucker**):

$$\begin{array}{ll} f\ c_1 = e_1 & f\ x = \text{case } x \text{ of } c_1 \rightarrow e_1 \\ \dots & \dots \\ f\ c_n = e_n & c_n \rightarrow e_n \end{array}$$

- Damit:

```
apreis :: Apfelsorte → Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50
```

Der einfachste Aufzählungstyp

- ▶ **Einfachste** Aufzählung: Wahrheitswerte

$$\text{Bool} = \{\text{False}, \text{True}\}$$

- ▶ Genau zwei unterschiedliche Werte
- ▶ **Definition** von Funktionen:
 - ▶ Wertetabellen sind explizite Fallunterscheidungen

\wedge	<i>true</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>

$$\begin{array}{lllll} \text{true} & \wedge & \text{true} & = & \text{true} \\ \text{true} & \wedge & \text{false} & = & \text{false} \\ \text{false} & \wedge & \text{true} & = & \text{false} \\ \text{false} & \wedge & \text{false} & = & \text{false} \end{array}$$

Wahrheitswerte: Bool

- **Vordefiniert** als

```
data Bool = False | True
```

- **Vordefinierte Funktionen:**

not	:: Bool → Bool	— Negation
(&&)	:: Bool → Bool → Bool	— Konjunktion
()	:: Bool → Bool → Bool	— Disjunktion

- **if _ then _ else _** als syntaktischer Zucker:

$$\text{if } b \text{ then } p \text{ else } q \longrightarrow \text{case } b \text{ of } \begin{array}{l} \text{True} \rightarrow p \\ \text{False} \rightarrow q \end{array}$$

Striktheit Revisited

- **Konjunktion** definiert als

```
a && b = case a of False → False  
                  True → b
```

- Alternative Definition als Wahrheitstabelle:

```
and :: Bool → Bool → Bool  
and False True = False  
and False False = False  
and True True = True  
and True False = False
```

Übung 3.1: Kurze Frage: Gibt es einen Unterschied zwischen den beiden?



Striktheit Revisited

- **Konjunktion** definiert als

```
a && b = case a of False → False  
           True → b
```

- Alternative Definition als Wahrheitstabelle:

```
and :: Bool → Bool → Bool  
and False True = False  
and False False = False  
and True True = True  
and True False = False
```

Übung 3.1: Kurze Frage: Gibt es einen Unterschied zwischen den beiden?

Lösung:



- Erste Definition ist **nicht-strikt** im zweiten Argument.
- Merke: wir können Striktheit von Funktionen (ungewollt) **erzwingen**

II. Produkte

Produkte

- ▶ Konstruktoren können **Argumente** haben
- ▶ Beispiel: Ein **RGB-Wert** besteht aus drei Werten

- ▶ Mathematisch: Produkt (Tripel)

$$\text{Colour} = \{(r, g, b) \mid r \in \mathbb{N}, g \in \mathbb{N}, b \in \mathbb{N}\}$$

- ▶ In Haskell: Konstruktoren mit **Argumenten**

```
data Colour = RGB Int Int Int
```

- ▶ Beispielwerte:

```
yellow :: Colour
yellow = RGB 255 255 0      — 0xFFFF00
```

```
violet :: Colour
violet = RGB 238 130 238    — 0xEE82EE
```

Funktionsdefinition auf Produkten

► **Funktionsdefinition:**

- Konstruktorargumente sind **gebundene** Variablen
- Wird bei der **Auswertung** durch konkretes Argument ersetzt
- Kann mit Fallunterscheidung kombiniert werden

► Beispiele:

```
red :: Colour → Int
red (RGB r _ _) = r
```

Funktionsdefinition auf Produkten

► **Funktionsdefinition:**

- Konstruktorargumente sind **gebundene** Variablen
- Wird bei der **Auswertung** durch konkretes Argument ersetzt
- Kann mit Fallunterscheidung kombiniert werden

► Beispiele:

```
red :: Colour → Int
red (RGB r _ _) = r
```

```
adjust :: Colour → Float → Colour
adjust (RGB r g b) f = RGB (conv r) (conv g) (conv b) where
  conv colour = min (round (fromIntegral colour * f)) 255
```



Beispiel: Bob's Auld-Time Grocery Shoppe

- ▶ Käsesorten und deren Preise:

```
data Kaesesorte = Gouda | Appenzeller
```

```
kpreis :: Käsesorte → Int
```

```
kpreis Gouda = 1450
```

```
kpreis Appenzeller = 2270
```

Beispiel: Bob's Auld-Time Grocery Shoppe

- ▶ Käsesorten und deren Preise:

```
data Kaesesorte = Gouda | Appenzeller
```

```
kpreis :: Käsesorte → Int
```

```
kpreis Gouda = 1450
```

```
kpreis Appenzeller = 2270
```

- ▶ Alle Artikel:

```
data Artikel =  
  Apfel Apfelsorte | Eier  
  | Käse Käsesorte | Schinken  
  | Salami | Milch Bio
```

```
data Bio = Bio | Chemie
```

Beispiel: Bob's Auld-Time Grocery Shoppe

- ▶ Berechnung des Preises für eine bestimmte **Menge** eines **Produktes**
- ▶ Mengenangaben:

```
data Menge = Stueck Int | Gramm Int | Liter Double
```

- ▶ Preisberechnung

```
preis :: Artikel → Menge → Int
```

- ▶ Aber was ist mit ungültigen Kombinationen (3 Liter Äpfel)?
- ▶ Könnten Laufzeitfehler erzeugen (**error** ..)

Beispiel: Bob's Auld-Time Grocery Shoppe

- ▶ Berechnung des Preises für eine bestimmte **Menge** eines **Produktes**
- ▶ Mengenangaben:

```
data Menge = Stueck Int | Gramm Int | Liter Double
```

- ▶ Preisberechnung

```
preis :: Artikel → Menge → Int
```

- ▶ Aber was ist mit ungültigen Kombinationen (3 Liter Äpfel)?
- ▶ Könnten Laufzeitfehler erzeugen (`error ..`) aber nicht wieder fangen.
- ▶ Ausnahmebehandlung **nicht referentiell transparent**

Beispiel: Bob's Auld-Time Grocery Shoppe

- ▶ Berechnung des Preises für eine bestimmte **Menge** eines **Produktes**
- ▶ Mengenangaben:

```
data Menge = Stueck Int | Gramm Int | Liter Double
```

- ▶ Preisberechnung

```
preis :: Artikel → Menge → Int
```

- ▶ Aber was ist mit ungültigen Kombinationen (3 Liter Äpfel)?
- ▶ Könnten Laufzeitfehler erzeugen (`error ..`) aber nicht wieder fangen.
- ▶ Ausnahmebehandlung **nicht referentiell transparent**
- ▶ Könnten spezielle Werte (`0` oder `-1`) zurückgeben

Beispiel: Bob's Auld-Time Grocery Shoppe

- ▶ Berechnung des Preises für eine bestimmte **Menge** eines **Produktes**
- ▶ Mengenangaben:

```
data Menge = Stueck Int | Gramm Int | Liter Double
```

- ▶ Preisberechnung

```
preis :: Artikel → Menge → Int
```

- ▶ Aber was ist mit ungültigen Kombinationen (3 Liter Äpfel)?
- ▶ Könnten Laufzeitfehler erzeugen (`error ..`) aber nicht wieder fangen.
- ▶ Ausnahmebehandlung **nicht referentiell transparent**
- ▶ Könnten spezielle Werte (`0` oder `-1`) zurückgeben
- ▶ Besser: Ergebnis als Datentyp mit explizitem Fehler (**Reifikation**):

```
data Preis = Cent Int | Ungueltig
```

Beispiel: Bob's Auld-Time Grocery Shoppe

- Der Preis und seine Berechnung:

```
data Preis = Cent Int | Ungueltig
```

```
preis :: Artikel → Menge → Preis
```

```
preis (Apfel a) (Stueck n) = Cent (n* apreis a)
```

```
preis Eier (Stueck n) = Cent (n* 20)
```

```
preis (Kaese k)(Gramm g) = Cent (div (g* kpreis k) 1000)
```

```
preis Schinken (Gramm g) = Cent (div (g* 199) 100)
```

```
preis Salami (Gramm g) = Cent (div (g* 159) 100)
```

```
preis (Milch bio) (Liter l) =
```

```
    Cent (round (l* case bio of Bio → 119; Chemie → 69))
```

```
preis _ _ = Ungueltig
```



Übung 3.1: Refaktorierungen

Was passiert bei folgenden Änderungen an `preis` :

- 1 Vorletzte Zeile zu `Cent (round (l* case bio of Chemie → 69; Bio→ 119`
- 2 Vorletzte Zeile zu `Cent (round (l* case bio of Bio→ 119; _ → 69`
- 3 Vertauschung der zwei vorletzten und letzten Zeile.

Jetzt seit ihr dran

Übung 3.1: Refaktorierungen

Was passiert bei folgenden Änderungen an `preis` :

- ① Vorletzte Zeile zu `Cent (round (l* case bio of Chemie → 69; Bio→ 119`
- ② Vorletzte Zeile zu `Cent (round (l* case bio of Bio→ 119; _ → 69`
- ③ Vertauschung der zwei vorletzten und letzten Zeile.

Lösung:

- ① Nichts, unterschiedliche Fälle können getauscht werden.
- ② Nichts, da `_` nur `Chemie` sein kann
- ③ Der letzte Fall wird nie aufgerufen — der Milchpreis wäre `Ungueltig`

III. Algebraische Datentypen

Der Allgemeine Fall: Algebraische Datentypen

```
data T = C1
        | C2
        :
        | Cn
```

► Aufzählungen

Der Allgemeine Fall: Algebraische Datentypen

```
data T = C1 t1,1 ... t1,k1
```

- ▶ **Aufzählungen**
- ▶ Konstrukturen mit **einem** oder **mehreren** Argumenten (Produkte)

Der Allgemeine Fall: Algebraische Datentypen

```
data T =  C1 t1,1 ... t1,k1
         |  C2 t2,1 ... t2,k2
         :
         |  Cn tn,1 ... tn,kn
```

- ▶ **Aufzählungen**
- ▶ Konstrukturen mit **einem** oder **mehreren** Argumenten (Produkte)
- ▶ Der allgemeine Fall: **mehrere** Konstrukturen

Eigenschaften algebraischer Datentypen

```
data T =  C1 t1,1 ... t1,k1
         |  C2 t2,1 ... t2,k2
         :
         |  Cn tn,1 ... tn,kn
```

Drei Eigenschaften eines algebraischen Datentypen

- ① Konstruktoren C₁, ..., C_n sind **disjunkt**:

$$C_i \ x_1 \dots x_n = C_j \ y_1 \dots y_m \implies i = j$$

Eigenschaften algebraischer Datentypen

```
data T = C1 t1,1 ... t1,k1
        | C2 t2,1 ... t2,k2
        |
        |
        | Cn tn,1 ... tn,kn
```

Drei Eigenschaften eines algebraischen Datentypen

- 1 Konstruktoren C₁, ..., C_n sind **disjunkt**:

$$C_i x_1 \dots x_n = C_j y_1 \dots y_m \implies i = j$$

- 2 Konstruktoren sind **injektiv**:

$$C x_1 \dots x_n = C y_1 \dots y_n \implies x_i = y_i$$

Eigenschaften algebraischer Datentypen

```
data T = C1 t1,1 ... t1,k1
       | C2 t2,1 ... t2,k2
       |
       | Cn tn,1 ... tn,kn
```

Drei Eigenschaften eines algebraischen Datentypen

① Konstruktoren C₁, ..., C_n sind **disjunkt**:

$$C_i x_1 \dots x_n = C_j y_1 \dots y_m \implies i = j$$

② Konstruktoren sind **injektiv**:

$$C x_1 \dots x_n = C y_1 \dots y_m \implies x_i = y_i$$

③ Konstruktoren **erzeugen** den Datentyp:

$$\forall x \in T. x = C_i y_1 \dots y_m$$

Eigenschaften algebraischer Datentypen

```
data T = C1 t1,1 ... t1,k1
       | C2 t2,1 ... t2,k2
       |
       | Cn tn,1 ... tn,kn
```

Drei Eigenschaften eines algebraischen Datentypen

① Konstruktoren C₁, ..., C_n sind **disjunkt**:

$$C_i x_1 \dots x_n = C_j y_1 \dots y_m \implies i = j$$

② Konstruktoren sind **injektiv**:

$$C x_1 \dots x_n = C y_1 \dots y_m \implies x_i = y_i$$

③ Konstruktoren **erzeugen** den Datentyp:

$$\forall x \in T. x = C_i y_1 \dots y_m$$

Diese Eigenschaften machen **Fallunterscheidung** wohldefiniert.

Algebraische Datentypen: Nomenklatur

data $T = C_1 t_{1,1} \dots t_{1,k_1} \mid \dots \mid C_n t_{n,1} \dots t_{n,k_n}$

► C_i sind **Konstruktoren**

► **Immer** implizit definiert und deklariert

► **Selektoren** sind Funktionen $sel_{i,j}$:

$sel_{i,j} \quad :: T \rightarrow t_{i,k_i}$

$sel_{i,j} (C_i t_{i,1} \dots t_{i,k_i}) = t_{i,j}$

► Partiell, linksinvers zu Konstruktor C_i

► **Können** implizit definiert und deklariert werden

► **Diskriminatoren** sind Funktionen dis_i :

$dis_i \quad :: T \rightarrow \text{Bool}$

$dis_i (C_i \dots) = \text{True}$

$dis_i _ = \text{False}$

► Definitionsbereich des Selektors sel_i , **nie** implizit

Auswertung der Fallunterscheidung

- Argument der Fallunterscheidung wird **nur soweit nötig** ausgewertet
- Beispiel:

```
f :: Preis → Int
f p = case p of Cent i → i; Ungueltig → 0
```

```
g :: Preis → Int
g p = case p of Cent i → 99; Ungueltig → 0
```

```
add :: Preis → Preis → Preis
add (Cent i) (Cent j) = Cent (i + j)
add _ _ = Ungueltig
```

Auswertung der Fallunterscheidung

- ▶ Argument der Fallunterscheidung wird **nur soweit nötig** ausgewertet
- ▶ Beispiel:

```
f :: Preis → Int
f p = case p of Cent i → i; Ungueltig → 0
```

```
g :: Preis → Int
g p = case p of Cent i → 99; Ungueltig → 0
```

```
add :: Preis → Preis → Preis
add (Cent i) (Cent j) = Cent (i + j)
add _ _ = Ungueltig
```

- ▶ Argument von **Cent** wird in **f** ausgewertet, in **g** nicht
- ▶ Zweites Argument von **add** wird nicht immer ausgewertet

Rekursive Algebraische Datentypen

```
data T =  C1 t1,1 ... t1,k1  
         :  
         |  Cn tn,1 ... tn,kn
```

- ▶ Der definierte Typ **T** kann **rechts** benutzt werden.
- ▶ Rekursive Datentypen definieren **unendlich große** Wertemengen.
- ▶ Modelliert **Aggregation** (Sammlung von Objekten).
- ▶ Funktionen werden durch **Rekursion** definiert.

Uncle Bob's Auld-Time Grocery Shoppe Revisited

- ▶ Das **Lager** für Bob's Shoppe:

- ▶ ist entweder leer,
- ▶ oder es enthält einen Artikel und Menge, und noch mehr

```
data Lager = LeeresLager
          | Lager Artikel Menge Lager
```

Suchen im Lager

- Rekursive Suche (erste Version):

```
suche :: Artikel → Lager → Menge
suche art LeeresLager = ???
```

Suchen im Lager

- Rekursive Suche (erste Version):

```
suche :: Artikel → Lager → Menge
suche art LeeresLager = ???
```

- Modellierung des **Resultats**:

```
data Resultat = Gefunden Menge | NichtGefunden
```

- Damit rekursive **Suche**:

```
suche :: Artikel → Lager → Resultat
suche art (Lager lart m 1)
  | art == lart = Gefunden m
  | otherwise    = suchen art l
suche art LeeresLager = NichtGefunden
```

Einlagern

- Signatur:

```
einlagern :: Artikel → Menge → Lager → Lager
```

- Erste Version:

```
einlagern a m l = Lager a m l
```

- Mengen sollen **aggregiert** werden ($35\text{ l Milch} + 20\text{ l Milch} = 55\text{ l Milch}$)
- Dazu Hilfsfunktion:

```
addiere (Stueck i) (Stueck j) = Stueck (i + j)
addiere (Gramm g) (Gramm h) = Gramm (g + h)
addiere (Liter l) (Liter m) = Liter (l + m)
addiere m n = error ("addiere:" ++ show m ++ " und " ++ show n)
```

Einlagern

- Damit einlagern:

```
einlagern :: Artikel → Menge → Lager → Lager
einlagern a m LeeresLager = Lager a m LeeresLager
einlagern a m (Lager al ml 1)
| a = al = Lager a (addiere m ml) 1
| otherwise = Lager al ml (einlagern a m 1)
```

- Problem:

Einlagern

- ▶ Damit einlagern:

```
einlagern :: Artikel → Menge → Lager → Lager
einlagern a m LeeresLager = Lager a m LeeresLager
einlagern a m (Lager al ml 1)
  | a = al  = Lager a (addiere m ml) 1
  | otherwise = Lager al ml (einlagern a m 1)
```

- ▶ Problem: **Falsche Mengenangaben**

- ▶ Bspw. `einlagern Eier (Liter 3.0) 1`
- ▶ Erzeugen Laufzeitfehler in `addiere`
- ▶ Lösung: eigentliche Funktion `einlagern` wird als **lokale Funktion** versteckt, und nur mit gültiger Mengenangabe aufgerufen.

Einlagern

- Lösung: eigentliche Funktion `einlagern` wird als **lokale Funktion** versteckt, und nur mit gültiger Mengenangabe aufgerufen.

```
einlagern :: Artikel → Menge → Lager → Lager
einlagern a m l =
  let einlagern' a m LeeresLager = Lager a m LeeresLager
    einlagern' a m (Lager al ml l)
      | a == al  = Lager a (addiere m ml) l
      | otherwise = Lager al ml (einlagern' a m l)
  in case preis a m of
    Ungueltig → l
    _ → einlagern' a m l
```

Einkaufen und bezahlen

- Wir brauchen einen **Einkaufskorb**:

```
data Einkaufskorb = LeererKorb
                  | Einkauf Artikel Menge Einkaufskorb
```

- Artikel einkaufen:

```
einkauf :: Artikel → Menge → Einkaufskorb → Einkaufskorb
einkauf a m e =
  case preis a m of
    Ungueltig → e
    _ → Einkauf a m e
```

- Auch hier: ungültige Mengenangaben erkennen
- Es wird **nicht** aggregiert

Beispiel: Kassenbon

```
kassenbon :: Einkaufskorb → String
```

Ausgabe:

** Bob's Aulde-Time Grocery Shoppe **

Unveränderlicher Kopf

Artikel	Menge	Preis
Kaese Appenzeller	378 g.	8.58 EU
Schinken	50 g.	0.99 EU
Milch Bio	1.0 l.	1.19 EU
Schinken	50 g.	0.99 EU
Apfel Boskoop	3 St	1.65 EU
Summe:		13.40 EU

Ausgabe von Artikel und
Menge (rekursiv)

Ausgabe von `kasse`

Kassenbon: Implementation

► Kernfunktion:

```
artikel :: Einkaufskorb → String
artikel LeererKorb = ""
artikel (Einkauf a m e) =
  formatL 20 (show a) ++
  formatR 7 (menge m) ++
  formatR 10 (showEuro (cent a m)) ++ "\n" ++ artikel e
```

► Hilfsfunktionen:

```
formatL :: Int → String → String
```

```
formatR :: Int → String → String
```

```
showEuro :: Int → String
```



Kurz zum Nachdenken

Übung 3.2: Zeichenketten

Wie könnten wohl Zeichenketten (**String**) definiert sein?

IV. Rekursive Datentypen

Beispiel: Zeichenketten selbstgemacht

- ▶ Eine **Zeichenkette** ist
 - ▶ entweder **leer** (das leere Wort ϵ)
 - ▶ oder ein **Zeichen** c und eine weitere **Zeichenkette** xs

```
data MyString = Empty
              | Char :+ MyString
```

- ▶ **Lineare** Rekursion
 - ▶ Genau ein rekursiver Aufruf
- ▶ Haskell-Merkwürdigkeit #237:
 - ▶ Die Namen von Operator-Konstruktoren müssen mit einem : beginnen.

Rekursiver Typ, rekursive Definition

- ▶ Typisches Muster: **Fallunterscheidung**

- ▶ Ein Fall pro Konstruktor

- ▶ Hier:

- ▶ Leere Zeichenkette

- ▶ Nichtleere Zeichenkette

Funktionen auf Zeichenketten

► Länge:

```
length :: MyString → Int
length Empty      = 0
length (c :+ s) = 1 + length s
```

Funktionen auf Zeichenketten

► Länge:

```
length :: MyString → Int
length Empty    = 0
length (c :+ s) = 1 + length s
```

► Verkettung:

```
(++) :: MyString → MyString → MyString
Empty ++ t = t
(c :+ s) ++ t = c :+ (s ++ t)
```

Funktionen auf Zeichenketten

► Länge:

```
length :: MyString → Int
length Empty    = 0
length (c :+ s) = 1 + length s
```

► Verkettung:

```
(++) :: MyString → MyString → MyString
Empty ++ t = t
(c :+ s) ++ t = c :+ (s ++ t)
```

► Umdrehen:

```
rev :: MyString → MyString
rev Empty    = Empty
rev (c :+ t) = rev t ++ (c :+ Empty)
```



Zusammenfassung

- ▶ Algebraische Datentypen: Aufzählungen, Produkte, rekursive Datentypen
- ▶ Drei Schlüsseleigenschaften der Konstruktoren: **disjunkt**, **injektiv**, **erzeugend**
- ▶ Rekursive Datentypen sind **unendlich** (induktiv)
- ▶ Funktionen werden durch **Fallunterscheidung** und **Rekursion** definiert
- ▶ Fallbeispiele: Bob's Shoppe, Zeichenketten

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 4 vom 23.11.2020: Typvariablen und Polymorphie

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Fahrplan

- ▶ **Teil I: Funktionale Programmierung im Kleinen**
 - ▶ Einführung
 - ▶ Funktionen
 - ▶ Algebraische Datentypen
 - ▶ Typvariablen und Polymorphie
 - ▶ Funktionen höherer Ordnung I
 - ▶ Rekursive und zyklische Datenstrukturen
 - ▶ Funktionen höherer Ordnung II
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- ▶ Letzte Vorlesungen: algebraische Datentypen
- ▶ Diese Vorlesung:
 - ▶ **Abstraktion** über Typen: Typvariablen und Polymorphie
 - ▶ Arten der Polymorphie:
 - ▶ Parametrische Polymorphie
 - ▶ Ad-hoc Polymorphie
 - ▶ Typableitung in Haskell

Lernziele

Wir verstehen, wie in Haskell die Typableitung funktioniert, und was Signaturen wie `head :: [α] → α` und `elem :: Eq α ⇒ α → [α] → Bool` bedeuten.

Ähnliche Datentypen der letzten Vorlesung

```
data Lager = LeeresLager
          | Lager Artikel Menge Lager
```

```
data Einkaufskorb = LeererKorb
                    | Einkauf Artikel Menge Einkaufskorb
```

```
data MyString = Empty
              | Char :+: MyString
```

- ▶ ein **konstanter** Konstruktor
- ▶ ein **linear rekursiver** Konstruktor

Ähnliche Funktionen der letzten Vorlesung

```
kasse  :: Einkaufskorb → Int
kasse LeererKorb = 0
kasse (Einkauf a m e) = cent a m + kasse e
```

```
inventur :: Lager → Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m + inventur l
```

```
length :: MyString → Int
length Empty      = 0
length (c :+ s) = 1 + length s
```

- ▶ ein Fall pro Konstruktor
- ▶ **linearer** rekursiver Aufruf

Die Lösung: Polymorphie

Definition (Polymorphie)

Polymorphie ist **Abstraktion über Typen**

Arten der Polymorphie

- ▶ **Parametrische** Polymorphie (Typvariablen):
Generisch über **alle** Typen
- ▶ **Ad-Hoc** Polymorphie (Überladung):
Nur für **bestimmte** Typen

Anders als in Java (mehr dazu später).

I. Parametrische Polymorphie

Parametrische Polymorphie: Typvariablen

- ▶ Typvariablen abstrahieren über Typen

```
data List α = Empty
           | Cons α (List α)
```

- ▶ α ist eine Typvariable
- ▶ List α ist ein polymorpher Datentyp
- ▶ Signatur der Konstruktoren

```
Empty :: List α
Cons  :: α → List α → List α
```

- ▶ Typvariable α wird bei Anwendung instantiiert

Polymorphe Ausdrücke

- ▶ **Typkorrekte** Terme:
Empty

Polymorphe Ausdrücke

► Typkorrekte Terme:

Empty

Typ

List α

Polymorphe Ausdrücke

► Typkorrekte Terme:

Empty

Typ

Cons 57 Empty

List α

Polymorphe Ausdrücke

► Typkorrekte Terme:

Empty

Typ

Cons 57 Empty

List α

List Int

Polymorphe Ausdrücke

► Typkorrekte Terme:

Empty

Typ

Cons 57 Empty

List α

Cons 7 (Cons 8 Empty)

List Int

Polymorphe Ausdrücke

► Typkorrekte Terme:

Empty

Typ

Cons 57 Empty

List α

Cons 7 (Cons 8 Empty)

List Int

List Int

Polymorphe Ausdrücke

► Typkorrekte Terme:	Typ
Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	List Int
Cons 'p' (Cons 'i' (Cons '3' Empty))	

Polymorphe Ausdrücke

► Typkorrekte Terme:	Typ
Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	List Int
Cons 'p' (Cons 'i' (Cons '3' Empty))	List Char

Polymorphe Ausdrücke

► Typkorrekte Terme:	Typ
Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	List Int
Cons 'p' (Cons 'i' (Cons '3' Empty))	List Char
Cons True Empty	

Polymorphe Ausdrücke

► Typkorrekte Terme:

	Typ
Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	List Int
Cons 'p' (Cons 'i' (Cons '3' Empty))	List Char
Cons True Empty	List Bool

► Nicht typ-korrekt:

Cons 'a' (Cons 0 Empty)
Cons True (Cons 'x' Empty)

wegen Signatur des Konstruktors:

Cons :: $\alpha \rightarrow \text{List } \alpha \rightarrow \text{List } \alpha$

Polymorphe Funktionen

- Parametrische Polymorphie für **Funktionen**:

```
(++) :: List α → List α → List α
```

```
Empty ++ t = t
```

```
(Cons c s) ++ t = Cons c (s ++ t)
```

- Typvariable vergleichbar mit Funktionsparameter
- Typvariable α wird bei Anwendung instantiiert:

```
Cons 'p' (Cons 'i' Empty) ++ Cons '3' Empty
```

```
Cons 3 Empty ++ Cons 5 (Cons 57 Empty)
```

aber **nicht**

```
Cons True Empty ++ Cons 'a' (Cons 'b' Empty)
```

Beispiel: Der Shop (refaktoriert)

- ▶ Einkaufswagen und Lager als Listen?
- ▶ Problem: zwei Typen als Argument

```
type Lager = List (Artikel Menge)
```

- ▶ Geht so **nicht!**
- ▶ Lösung: zu einem Typ zusammenfassen

```
data Posten = Posten Artikel Menge
```

- ▶ Damit:

```
type Lager = List Posten
type Einkaufskorb = List Posten
```

- ▶ **Gleicher** Typ!

Tupel

- Mehr als **eine** Typvariable möglich
- Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair α β = Pair { left :: α, right :: β }
```

- Signatur Konstruktor und Selektoren:

```
Pair :: α → β → Pair α β
```

```
left :: Pair α β → α
```

```
right :: Pair α β → β
```

Tupel

- Mehr als **eine** Typvariable möglich
- Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair α β = Pair { left :: α, right :: β }
```

- Signatur Konstruktor und Selektoren:

```
Pair :: α → β → Pair α β
```

```
left :: Pair α β → α
```

```
right :: Pair α β → β
```

- Beispielterm

Typ

```
Pair 4 'x'
```

Tupel

- ▶ Mehr als **eine** Typvariable möglich
 - ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair α β = Pair { left :: α, right :: β }
```

- ## ► Signatur Konstruktor und Selektoren:

Pair :: $\alpha \rightarrow \beta \rightarrow \text{Pair } \alpha \beta$

left :: **Pair** α β \rightarrow α

right :: Pair α β \rightarrow β

- Beispielterm Typ

Pair 4 'x'

Pair Int Char

Tupel

- Mehr als **eine** Typvariable möglich
- Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair α β = Pair { left :: α, right :: β }
```

- Signatur Konstruktor und Selektoren:

```
Pair :: α → β → Pair α β
```

```
left :: Pair α β → α
```

```
right :: Pair α β → β
```

- Beispielterm

	Typ
Pair 4 'x'	Pair Int Char

Pair (Cons True Empty) 'a'	
----------------------------	--

Tupel

- ▶ Mehr als **eine** Typvariable möglich
 - ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair α β = Pair { left :: α, right :: β }
```

- ## ► Signatur Konstruktor und Selektoren:

Pair :: $\alpha \rightarrow \beta \rightarrow \text{Pair } \alpha \beta$

left :: **Pair** α β \rightarrow α

right :: **Pair** α β \rightarrow β

- ## ► Beispielterm Typ

Pair 4 'x' Pair Int Char

Pair (Cons True Empty) 'a' Pair (List Bool) Char

Tupel

- ▶ Mehr als **eine** Typvariable möglich
 - ▶ Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair α β = Pair { left :: α, right :: β }
```

- ## ► Signatur Konstruktor und Selektoren:

Pair :: $\alpha \rightarrow \beta \rightarrow \text{Pair } \alpha \beta$

left :: **Pair** α β \rightarrow α

right :: **Pair** α β \rightarrow β

- ## ► Beispielterm Typ

Pair 4 'x' Pair Int Char

Pair (Cons True Empty) 'a' Pair (List Bool) Char

Pair (3+ 4) Empty

Tupel

- Mehr als **eine** Typvariable möglich
- Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair α β = Pair { left :: α, right :: β }
```

- Signatur Konstruktor und Selektoren:

```
Pair :: α → β → Pair α β
```

```
left :: Pair α β → α
```

```
right :: Pair α β → β
```

- Beispielterm
- | | Typ |
|----------------------------|-----------------------|
| Pair 4 'x' | Pair Int Char |
| Pair (Cons True Empty) 'a' | Pair (List Bool) Char |
| Pair (3+ 4) Empty | Pair Int (List α) |

Tupel

- Mehr als **eine** Typvariable möglich
- Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair α β = Pair { left :: α, right :: β }
```

- Signatur Konstruktor und Selektoren:

```
Pair :: α → β → Pair α β
```

```
left :: Pair α β → α
```

```
right :: Pair α β → β
```

- Beispielterm
- | | Typ |
|----------------------------|-----------------------|
| Pair 4 'x' | Pair Int Char |
| Pair (Cons True Empty) 'a' | Pair (List Bool) Char |
| Pair (3+ 4) Empty | Pair Int (List α) |
| Cons (Pair 7 'x') Empty | |

Tupel

- Mehr als **eine** Typvariable möglich
- Beispiel: **Tupel** (kartesisches Produkt, Paare)

```
data Pair α β = Pair { left :: α, right :: β }
```

- Signatur Konstruktor und Selektoren:

```
Pair :: α → β → Pair α β
```

```
left :: Pair α β → α
```

```
right :: Pair α β → β
```

► Beispielterm	Typ
Pair 4 'x'	Pair Int Char
Pair (Cons True Empty) 'a'	Pair (List Bool) Char
Pair (3+ 4) Empty	Pair Int (List α)
Cons (Pair 7 'x') Empty	List (Pair Int Char)

Jetzt seit ihr dran!

Übung 4.1: Neue Typen

Sind folgende Ausdrücke typkorrekt, und wenn ja welchen Typ haben sie?

- ① right (Pair (3 + 4) Empty)
- ② head (Pair (Cons 'x' Empty) True)
- ③ right (head (Cons (Pair 'x' 3) Empty))
- ④ head (tail (Cons 3 (Cons 4 Empty)))

Jetzt seit ihr dran!

Übung 4.1: Neue Typen

Sind folgende Ausdrücke typkorrekt, und wenn ja welchen Typ haben sie?

- ① right (Pair (3 + 4) Empty)
- ② head (Pair (Cons 'x' Empty) True)
- ③ right (head (Cons (Pair 'x' 3) Empty))
- ④ head (tail (Cons 3 (Cons 4 Empty)))

Lösung:

- ① Typ: List α
- ② Typfehler
- ③ Typ: Integer
- ④ Typ: Integer

II. Vordefinierte Datentypen

Vordefinierte Datentypen: Tupel und Listen

- ▶ Eingebauter **syntaktischer Zucker**
- ▶ **Listen**

```
data [α] = [] | α : [α]
```

- ▶ Weitere Abkürzungen:
Listenliterale: [x] für x:[], [x,y] für x:y:[] etc.
Aufzählungen: [n .. m] und [n, m .. k] für **aufzählbare** Typen

- ▶ **Tupel** sind das kartesische Produkt

```
data (α, β) = ( fst :: α, snd :: β)
```

- ▶ (a, b) = alle Kombinationen von Werten aus a und b
- ▶ Auch n-Tupel: (a,b,c) etc. (aber ohne Selektoren)
- ▶ 0-Tupel: () (*unit type*, Typ mit genau einem Element)

Vordefinierte Datentypen: Optionen

- Existierende Typen:

```
data Preis = Cent Int | Ungueltig
```

```
data Resultat = Gefunden Menge | NichtGefunden
```

- Instanzen eines **vordefinierten** Typen:

```
data Maybe α = Nothing | Just α
```

- Vordefinierten Funktionen (`import Data.Maybe`):

```
fromJust      :: Maybe α → α      — partiell
```

```
fromMaybe     :: α → Maybe α → α
```

```
listToMaybe   :: [α] → Maybe α      — totale Variante von head
```

```
maybeToList   :: Maybe α → [α]      — rechtsinvers zu listToMaybe
```

- Es gilt: $\text{listToMaybe}(\text{maybeToList } m) = m$
 $\text{length } l \leq 1 \implies \text{maybeToList}(\text{listToMaybe } l) = l$

Übersicht: vordefinierte Funktionen auf Listen I

(+)	:: $[\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$	— Verkettet zwei Listen
(!!)	:: $[\alpha] \rightarrow \text{Int} \rightarrow \alpha$	— n -tes Element selektieren, gezählt ab 0
concat	:: $[[\alpha]] \rightarrow [\alpha]$	— “flachklopfen”
length	:: $[\alpha] \rightarrow \text{Int}$	— Länge
head, last	:: $[\alpha] \rightarrow \alpha$	— Erstes bzw. letztes Element
tail, init	:: $[\alpha] \rightarrow [\alpha]$	— Hinterer bzw. vorderer Rest
replicate	:: $\text{Int} \rightarrow \alpha \rightarrow [\alpha]$	— Erzeuge n Kopien
repeat	:: $\alpha \rightarrow [\alpha]$	— Erzeugt zyklische Liste
take, drop	:: $\text{Int} \rightarrow [\alpha] \rightarrow [\alpha]$	— Erste bzw. letzte n Elemente
splitAt	:: $\text{Int} \rightarrow [\alpha] \rightarrow ([\alpha], [\alpha])$	— Spaltet an Index n , gezählt ab 0
reverse	:: $[\alpha] \rightarrow [\alpha]$	— Dreht Liste um
zip	:: $[\alpha] \rightarrow [\beta] \rightarrow [(\alpha, \beta)]$	— Erzeugt Liste von Paaren
unzip	:: $[(\alpha, \beta)] \rightarrow ([\alpha], [\beta])$	— Spaltet Liste von Paaren
and, or	:: $[\text{Bool}] \rightarrow \text{Bool}$	— Konjunktion/Disjunktion
sum, product	:: $[\text{Int}] \rightarrow \text{Int}$	— Summe und Produkt (überladen)

Vordefinierte Datentypen: Zeichenketten

- ▶ `String` sind Listen von Zeichen:

```
type String = [Char]
```

- ▶ Alle vordefinierten Funktionen auf `Listen` verfügbar.
- ▶ **Syntaktischer Zucker** für Stringliterale:

```
"yoho" == ['y', 'o', 'h', 'o'] == 'y': 'o': 'h': 'o': []
```

- ▶ Beispiele:

```
"abc" !! 1 ~> 'b'  
reverse "oof" ~> "foo"  
['a', 'c'.. 'z'] ~> "acegikmoqsuwy"  
splitAt 10 "Praktische \ Informatik" ~> ("Praktische", "\ Informatik")
```



Jetzt seit ihr dran!

Übung 4.2: Vordefinierte Typen

Sind folgende Ausdrücke typkorrekt, wenn ja welchen Typ haben sie, und was ist ihr Wert?

- ① `take 4 (replicate 3 (3, 4))`
- ② `snd (unzip (zip [1..10] "foo"))`
- ③ `"a"++ [('a')]`
- ④ `head [("foo", []), ("baz", 4 :: Integer)]`

Jetzt seit ihr dran!

Übung 4.2: Vordefinierte Typen

Sind folgende Ausdrücke typkorrekt, wenn ja welchen Typ haben sie, und was ist ihr Wert?

- ① `take 4 (replicate 3 (3, 4))`
- ② `snd (unzip (zip [1..10] "foo"))`
- ③ `"a"++ [('a')]`
- ④ `head [("foo", []), ("baz", 4 :: Integer)]`

Lösung:

- ① Typ: `[(Integer, Integer)]`, Wert: `[(3,4),(3,4),(3,4)]`
- ② Typ: `String`, Wert: `"foo"`
- ③ Typ: `String`, Wert: `"aa"`
- ④ Typfehler

III. Ad-Hoc Polymorphie

Parametrische Polymorphie: Grenzen

► Eine Funktion $f: \alpha \rightarrow \beta$ funktioniert auf **allen** Typen **gleich**.

► Nicht immer der Fall:

► Gleichheit: $(==) :: \alpha \rightarrow \alpha \rightarrow \text{Bool}$

Nicht auf allen Typen ist Gleichheit entscheidbar (besonders **Funktionen**)

► Ordnung: $(\triangleleft) :: \alpha \rightarrow \alpha \rightarrow \text{Bool}$

Nicht auf allen Typen definiert

► Anzeige: $\text{show} :: \alpha \rightarrow \text{String}$

Konversion in Zeichenketten höchst divers (Zeichenketten, Listen, Zahlen...)

Ad-Hoc Polymorphie und Overloading

Definition (Überladung)

Funktion $f :: \alpha \rightarrow \beta$ existiert für **mehr als einen**, aber **nicht** für **alle** Typen

- ▶ Lösung: **Typklassen**
- ▶ Typklassen bestehen aus:
 - ▶ **Deklaration** der Typklasse
 - ▶ **Instantiierung** für bestimmte Typen
- ▶ **Achtung:** hat wenig mit Klassen in Java zu tun

Typklassen: Syntax

► Deklaration:

```
class Show α where
  show :: α → String
```

► Instantiierung:

```
instance Show Bool where
  show True  = "Wahr"
  show False = "Falsch"
```

Prominente vordefinierte Typklassen

- ▶ Gleichheit: `Eq` für `(==)`
- ▶ Ordnung: `Ord` für `(≤)` (und andere Vergleiche)
- ▶ Anzeigen: `Show` für `show`
- ▶ Lesen: `Read` für `read :: String → α` (Achtung: Laufzeitefehler!)
- ▶ Numerische Typklassen:
 - ▶ `Num` für `0, 1, +, -`
 - ▶ `Integral` für `quot, rem, div, mod`
 - ▶ `Fractional` für `/`
 - ▶ `Floating` für `exp, log, sin, cos`

Typklassen in polymorphen Funktionen

- ▶ Element einer Liste (vordefiniert):

```
elem :: Eq α ⇒ α → [α] → Bool
elem e []      = False
elem e (x:xs) = e == x || elem e xs
```

- ▶ Sortierung einer List: `qsort`

```
qsort :: Ord α ⇒ [α] → [α]
```

- ▶ Liste ordnen und anzeigen:

```
showsorrted :: (Ord α, Show α) ⇒ [α] → String
showsorrted x = show (qsort x)
```

Hierarchien von Typklassen

- ▶ Typklassen können andere **voraussetzen**:

```
class Eq α⇒ Ord α where
  (⟨) :: α→ α→ Bool
  (≤) :: α→ α→ Bool
  a < b = a ≤ b && a ≠ b
```

- ▶ **Default**-Definition von `(⟨)`
- ▶ Kann bei Instantiierung überschrieben werden

Jetzt wieder ihr!

Übung 4.2: Meine Paare

Erinnert auch an die selbstgemachten Paare?

```
data Pair α β = Pair { left :: α, right :: β }
```

Schreibt eine **Show**-Instanz, welches ein Tupel als (a, b) anzeigt!

Jetzt wieder ihr!

Übung 4.2: Meine Paare

Erinnert auch an die selbstgemachten Paare?

```
data Pair α β = Pair { left :: α, right :: β }
```

Schreibt eine **Show**-Instanz, welches ein Tupel als (a, b) anzeigt!

Lösung:

- ▶ Voraussetzung: **Show** a, **Show** b
- ▶ Klammersetzung beachten

```
instance (Show a, Show b) ⇒ Show (Pair a b) where
  show (Pair a b) = "(" ++ show a ++ ", " ++ show b ++ ")"
```

IV. Typherleitung

Typen in Haskell (The Story So Far)

- ▶ Primitive Basisdatentypen: $\text{Bool}, \text{Double}$
- ▶ Funktionstypen $\text{Double} \rightarrow \text{Int} \rightarrow \text{Int}, [\text{Char}] \rightarrow \text{Double}$
- ▶ Typkonstruktoren: $[], (\dots), \text{Foo}$
- ▶ Typvariablen $\text{fst} :: (\alpha, \beta) \rightarrow \alpha$
 $\text{length} :: [\alpha] \rightarrow \text{Int}$
 $(++) :: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$
- ▶ Typklassen : $\text{elem} :: \text{Eq } \alpha \Rightarrow \alpha \rightarrow [\alpha] \rightarrow \text{Bool}$
 $\text{max} :: \text{Ord } \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$

Typinferenz: Das Problem

- Gegeben Definition von `f`:

```
f m xs = m + length xs
```

- Frage: welchen Typ hat `f`?
 - Unterfrage: ist die angegebene Typsignatur korrekt?
- **Informelle** Ableitung

$$f \ m \ xs \ = \ m \quad + \quad \text{length} \quad xs$$

Typinferenz: Das Problem

- Gegeben Definition von `f`:

```
f m xs = m + length xs
```

- Frage: welchen Typ hat `f`?
 - Unterfrage: ist die angegebene Typsignatur korrekt?
- **Informelle** Ableitung

$$f \ m \ xs \ = \ m \quad + \quad \text{length} \quad xs$$
$$[\alpha] \rightarrow \text{Int}$$

Typinferenz: Das Problem

- Gegeben Definition von `f`:

```
f m xs = m + length xs
```

- Frage: welchen Typ hat `f`?
 - Unterfrage: ist die angegebene Typsignatur korrekt?
- **Informelle** Ableitung

$$f \ m \ xs \ = \ m \quad + \quad \text{length} \quad xs$$
$$[\alpha] \rightarrow \text{Int}$$
$$[\alpha]$$

Typinferenz: Das Problem

- Gegeben Definition von `f`:

```
f m xs = m + length xs
```

- Frage: welchen Typ hat `f`?
 - Unterfrage: ist die angegebene Typsignatur korrekt?
- **Informelle** Ableitung

$$f \ m \ xs \ = \ m \quad + \quad \text{length} \quad xs$$
$$[\alpha] \rightarrow \text{Int}$$
$$[\alpha]$$
$$\text{Int}$$

Typinferenz: Das Problem

- Gegeben Definition von `f`:

```
f m xs = m + length xs
```

- Frage: welchen Typ hat `f`?
 - Unterfrage: ist die angegebene Typsignatur korrekt?
- **Informelle** Ableitung

$$f \ m \ xs \ = \ m \quad + \quad \text{length} \quad xs$$
$$[\alpha] \rightarrow \text{Int}$$
$$[\alpha]$$
$$\text{Int}$$
$$\text{Int}$$

Typinferenz: Das Problem

- Gegeben Definition von `f`:

```
f m xs = m + length xs
```

- Frage: welchen Typ hat `f`?
 - Unterfrage: ist die angegebene Typsignatur korrekt?
- **Informelle** Ableitung

$$f \ m \ xs \ = \ m \quad + \quad \text{length} \quad xs$$

$[\alpha] \rightarrow \text{Int}$

$[\alpha]$

Int

Int

Int

$f :: \text{Int} \rightarrow [\alpha] \rightarrow \text{Int}$

Typinferenz (nach Hindley-Milner)

- ▶ Typinferenz: **Herleitung** des Typen eines Ausdrucks
- ▶ Für bekannte Bezeichner wird Typ eingesetzt
- ▶ Für Variablen wird allgemeinster Typ angenommen
- ▶ Bei der Funktionsanwendung wird **unifiziert**:

$f \ m \ xs = m + \text{length} \ xs$

Typinferenz (nach Hindley-Milner)

- ▶ Typinferenz: **Herleitung** des Typen eines Ausdrucks
- ▶ Für bekannte Bezeichner wird Typ eingesetzt
- ▶ Für Variablen wird allgemeinster Typ angenommen
- ▶ Bei der Funktionsanwendung wird **unifiziert**:

$$\begin{array}{ccccccc} f & m & xs & = & m & + & \text{length} & xs \\ & \alpha & & & & & [\beta] \rightarrow \text{Int} & \gamma \end{array}$$

Typinferenz (nach Hindley-Milner)

- ▶ Typinferenz: **Herleitung** des Typen eines Ausdrucks
 - ▶ Für bekannte Bezeichner wird Typ eingesetzt
 - ▶ Für Variablen wird allgemeinster Typ angenommen
 - ▶ Bei der Funktionsanwendung wird **unifiziert**:

`f m xs = m + length xs`

α [math]\beta \rightarrow Int γ
[math]\beta $\gamma \mapsto$ [math]\beta

Typinferenz (nach Hindley-Milner)

- ▶ Typinferenz: **Herleitung** des Typen eines Ausdrucks
- ▶ Für bekannte Bezeichner wird Typ eingesetzt
- ▶ Für Variablen wird allgemeinster Typ angenommen
- ▶ Bei der Funktionsanwendung wird **unifiziert**:

$f \ m \ xs = m + \text{length} \ xs$

$$\frac{\alpha \quad [\beta] \rightarrow \text{Int} \quad \gamma}{[\beta] \quad \gamma \mapsto [\beta]}$$

Int

Typinferenz (nach Hindley-Milner)

- ▶ Typinferenz: **Herleitung** des Typen eines Ausdrucks
 - ▶ Für bekannte Bezeichner wird Typ eingesetzt
 - ▶ Für Variablen wird allgemeinster Typ angenommen
 - ▶ Bei der Funktionsanwendung wird **unifiziert**:

`f m xs = m + length xs`

$$\begin{array}{c}
 \alpha \qquad \qquad \qquad [\beta] \rightarrow \text{Int} \qquad \gamma \\
 \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad [\beta] \qquad \gamma \mapsto [\beta] \\
 \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \text{Int} \\
 \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
 \end{array}$$

Typinferenz (nach Hindley-Milner)

- ▶ Typinferenz: **Herleitung** des Typen eines Ausdrucks
 - ▶ Für bekannte Bezeichner wird Typ eingesetzt
 - ▶ Für Variablen wird allgemeinster Typ angenommen
 - ▶ Bei der Funktionsanwendung wird **unifiziert**:

Typinferenz (nach Hindley-Milner)

- ▶ Typinferenz: **Herleitung** des Typen eines Ausdrucks
 - ▶ Für bekannte Bezeichner wird Typ eingesetzt
 - ▶ Für Variablen wird allgemeinster Typ angenommen
 - ▶ Bei der Funktionsanwendung wird **unifiziert**:

`f m xs = m + length xs`

α	$[\beta] \rightarrow \text{Int}$	γ
		$[\beta]$
		$\gamma \mapsto [\beta]$
	Int	
$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$		
Int		$\alpha \mapsto \text{In}$
	$\text{Int} \rightarrow \text{Int}$	

Typinferenz (nach Hindley-Milner)

- ▶ Typinferenz: **Herleitung** des Typen eines Ausdrucks
 - ▶ Für bekannte Bezeichner wird Typ eingesetzt
 - ▶ Für Variablen wird allgemeinster Typ angenommen
 - ▶ Bei der Funktionsanwendung wird **unifiziert**:

`f m xs = m + length xs`

α [β] \rightarrow Int γ
[β] $\gamma \mapsto$ [β]

Tnt

Int → Int → Int

Int

$\text{Int} \rightarrow \text{Int}$

$\alpha \mapsto \text{Int}$

Int

$f :: \text{Int} \rightarrow [\beta] \rightarrow \text{Int}$

Typinferenz

Theorem (Entscheidbarkeit der Typinferenz)

Die Typinferenz ist **entscheidbar**, und findet immer den **allgemeinsten** Typ, wenn er existiert.

- ▶ Entscheidbarkeit ist nicht alles.
- ▶ Grundsätzliche Komplexität ist $DEXPTIME(n)$ (deterministisch exponentiell), aber in der Praxis ist das **nie** ein Problem.



Typinferenz

- Unifikation kann **mehrere Substitutionen** beinhalten:

f x y = (x, 3) : ('f', y) : []

Typinferenz

- Unifikation kann **mehrere Substitutionen** beinhalten:

$$\begin{array}{lllll} f \ x \ y = & (x, \ 3) & : & ('f', \ y) & : \quad [] \\ & \alpha \text{ Int} & & \text{Char} \ \beta & [\gamma] \end{array}$$

Typinferenz

- Unifikation kann **mehrere Substitutionen** beinhalten:

$$\begin{array}{lllll} f \ x \ y = & (x, \ 3) & : & ('f', \ y) & : \quad [] \\ & \alpha \text{ Int} & & \text{Char} \ \beta & [\gamma] \\ & (\alpha, \ \text{Int}) & & (\text{Char}, \ \beta) & \end{array}$$

Typinferenz

- Unifikation kann **mehrere Substitutionen** beinhalten:

$$\begin{array}{llll} f \ x \ y = & (x, \ 3) & : & ('f', \ y) \quad : \quad [] \\ & \alpha \text{ Int} & & \text{Char} \ \beta \quad [\gamma] \\ & (\alpha, \ \text{Int}) & & (\text{Char}, \ \beta) \\ & & & [(\text{Char}, \ \beta)] \quad \gamma \mapsto (\text{Char}, \ \beta) \end{array}$$

Typinferenz

- Unifikation kann **mehrere Substitutionen** beinhalten:

$$\begin{array}{llll} f \ x \ y = & (x, \ 3) & : & ('f', \ y) \quad : \quad [] \\ & \alpha \text{ Int} & & \text{Char} \ \beta \quad [\gamma] \\ & (\alpha, \ \text{Int}) & & (\text{Char}, \ \beta) \\ & & & [(\text{Char}, \ \beta)] \quad \gamma \mapsto (\text{Char}, \ \beta) \\ & & & [(\text{Char}, \ \text{Int})] \quad \beta \mapsto \text{Int}, \ \alpha \mapsto \text{Char} \end{array}$$

Typinferenz

- Unifikation kann **mehrere Substitutionen** beinhalten:

$$\begin{array}{llll} f \ x \ y = & (x, \ 3) & : & ('f', \ y) \quad : \quad [] \\ & \alpha \text{ Int} & & \text{Char} \ \beta \quad [\gamma] \\ & (\alpha, \ \text{Int}) & & (\text{Char}, \ \beta) \\ & & & \text{[(Char, } \beta)] \quad \gamma \mapsto (\text{Char, } \beta) \\ & & & \text{[(Char, Int)]} \quad \beta \mapsto \text{Int, } \alpha \mapsto \text{Char} \\ f \ :: \ \text{Char} \rightarrow \text{Int} \rightarrow \text{[(Char, Int)]} & & & \end{array}$$

- Allgemeinster Typ **muss nicht** existieren (Typfehler!)

Und was ist mit Typklassen?

- ▶ Typklassen schränken den Typ ein
- ▶ Typklassen werden bei der Unifikation **vereinigt**:

```
elem 3
Eq α :: α → [α] → Bool      Num β :: β
                           elem 3
(Eq α, Num α) :: [α] → Bool
```

- ▶ Instantiierung muss Typklassen berücksichtigen:

elem 3	"abc"	
(Eq α, Num α) :: [α] → Bool	[Char]	α → Char

- ▶ Char muss Instanz von Eq und Num sein.

Typfehler

- ▶ Typfehler treten auf, wenn zwei Typen t_1, t_2 nicht **unifiziert** werden können.
- ▶ Es gibt drei Arten von Typfehlern:
 - ➊ Typkonstanten nicht unifizierbar: `[True] ++ "a"`
 - ➋ Typ nicht Instanz der geforderten Klasse: `3 + 'a'`
 - ➌ Unifikation gibt **unendlichen** Typ: `x : x`



V. Abschließende Bemerkungen

Polymorphie: the missing link

	Parametrisch	Ad-Hoc
Funktionen	$f :: \alpha \rightarrow \text{Int}$	<code>class F α where</code> $f :: \alpha \rightarrow \text{Int}$
Typen	<code>data Maybe α =</code> $\text{Just } \alpha \mid \text{Nothing}$	

Polymorphie: the missing link

	Parametrisch	Ad-Hoc
Funktionen	$f :: \alpha \rightarrow \text{Int}$	<code>class F \alpha where</code> $f :: \alpha \rightarrow \text{Int}$
Typen	<code>data Maybe \alpha =</code> $\text{Just } \alpha \mid \text{Nothing}$	Konstruktorklassen

- Kann Entscheidbarkeit der Typherleitung gefährden

Zusammenfassung

- ▶ **Abstraktion** über Typen
 - ▶ Uniforme Abstraktion: Typvariable, parametrische Polymorphie
 - ▶ Fallbasierte Abstraktion: Überladung, ad-hoc-Polymorphie
- ▶ In der Sprache Haskell: **Typvariablen** und **Typklassen**
- ▶ Wichtige **vordefinierte** Typen:
 - ▶ Listen $[\alpha]$
 - ▶ Optionen $\text{Maybe } \alpha$
 - ▶ Tupel (α, β)

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 5 vom 30.11.2020: Funktionen Höherer Ordnung I

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Fahrplan

- ▶ **Teil I: Funktionale Programmierung im Kleinen**
 - ▶ Einführung
 - ▶ Funktionen
 - ▶ Algebraische Datentypen
 - ▶ Typvariablen und Polymorphie
 - ▶ **Funktionen höherer Ordnung I**
 - ▶ Rekursive und zyklische Datenstrukturen
 - ▶ Funktionen höherer Ordnung II
- ▶ **Teil II: Funktionale Programmierung im Großen**
- ▶ **Teil III: Funktionale Programmierung im richtigen Leben**

Inhalt

- ▶ Funktionen **höherer Ordnung**:
 - ▶ Funktionen als **gleichberechtigte Objekte**
 - ▶ Funktionen als **Argumente**
- ▶ Spezielle Funktionen: **map**, **filter**, **fold** und Freunde

Lernziel

Wir verstehen, wie wir mit **map**, **filter** und **fold** wiederkehrende Funktionsmuster kürzer und verständlicher aufschreiben können, und wir verstehen, warum der Funktionstyp in $\alpha \rightarrow \beta$ ein Typ wie jeder andere ist.

I. Funktionen als Werte

Funktionen Höherer Ordnung

Slogan

“Functions are first-class citizens.”

- ▶ Funktionen sind **gleichberechtigt**: Ausdrücke wie **alle anderen**
- ▶ **Grundprinzip** der funktionalen Programmierung
- ▶ Modellierung **allgemeiner Berechnungsmuster**
- ▶ Kontrollabstraktion

Ähnliche Datentypen der letzten Vorlesung

```
data Lager = LeeresLager
          | Lager Artikel Menge Lager
```

```
data Einkaufskorb = LeererKorb
                    | Einkauf Artikel Menge Einkaufskorb
```

```
data MyString = Empty
              | Char ::+ MyString
```

- ▶ ein **konstanter** Konstruktor
- ▶ ein **linear rekursiver** Konstruktor

Ähnliche Datentypen der letzten Vorlesung

```
data Lager = LeeresLager
          | Lager Artikel Menge Lager
```

```
data Einkaufskorb = LeererKorb
                    | Einkauf Artikel Menge Einkaufskorb
```

```
data MyString = Empty
              | Char ::+ MyString
```

- ▶ ein **konstanter** Konstruktor
- ▶ ein **linear rekursiver** Konstruktor

Gelöst durch Polymorphie

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufskorb → Int
kasse LeererKorb = 0
kasse (Einkauf a m e) = cent a m + kasse e
```

```
inventur :: Lager → Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m + inventur l
```

```
length :: MyString → Int
length Empty = 0
length (c :+ s) = 1 + length s
```

Gemeinsamkeiten:

- ▶ ein Fall pro Konstruktor
- ▶ **linearer** rekursiver Aufruf

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufskorb → Int
kasse LeererKorb = 0
kasse (Einkauf a m e) = cent a m + kasse e
```

```
inventur :: Lager → Int
inventur LeeresLager = 0
inventur (Lager a m l) = cent a m + inventur l
```

```
length :: MyString → Int
length Empty = 0
length (c :+ s) = 1 + length s
```

Gemeinsamkeiten:

- ▶ ein Fall pro Konstruktor
- ▶ **linearer** rekursiver Aufruf

Nicht durch Polymorphie gelöst

Ein einheitlicher Rahmen

- Zwei ähnliche Funktionen:

```
toL :: String → String
toL []     = []
toL (c:cs) = toLower c : toL cs
```

```
toU :: String → String
toU []     = []
toU (c:cs) = toUpper c : toU cs
```

- Warum nicht **eine** Funktion ...

Ein einheitlicher Rahmen

- Zwei ähnliche Funktionen:

```
toL :: String → String
toL []     = []
toL (c:cs) = toLower c : toL cs
```

```
toU :: String → String
toU []     = []
toU (c:cs) = toUpper c : toU cs
```

- Warum nicht **eine** Funktion ...

```
map f []     = []
map f (c:cs) = f c : map f cs
```

Ein einheitlicher Rahmen

- Zwei ähnliche Funktionen:

```
toL :: String → String
toL []     = []
toL (c:cs) = toLower c : toL cs
```

```
toU :: String → String
toU []     = []
toU (c:cs) = toUpper c : toU cs
```

- Warum nicht **eine** Funktion und **zwei** Instanzen?

```
map f []     = []
map f (c:cs) = f c : map f cs
```

```
toL cs = map toLower cs
toU cs = map toUpper cs
```

- **Funktion f als Argument**

- Was hätte **map** für einen **Typ**?

Funktionen als Werte: Funktionstypen

- Was hätte `map` für einen **Typ**?

```
map f []      = []
```

```
map f (c:cs) = f c : map f cs
```

- Was ist der Typ des **ersten** Arguments?

Funktionen als Werte: Funktionstypen

- Was hätte `map` für einen **Typ**?

```
map f []      = []
```

```
map f (c:cs) = f c : map f cs
```

- Was ist der Typ des **ersten Arguments**?

- Eine Funktion mit beliebigen Definitions- und Wertebereich: $\alpha \rightarrow \beta$

- Was ist der Typ des **zweiten Arguments**?

Funktionen als Werte: Funktionstypen

- Was hätte `map` für einen **Typ**?

```
map f []      = []
map f (c:cs) = f c : map f cs
```

- Was ist der Typ des **ersten Arguments**?
 - Eine Funktion mit beliebigen Definitions- und Wertebereich: $\alpha \rightarrow \beta$
- Was ist der Typ des **zweiten Arguments**?
 - Eine Liste, auf deren Elemente die Funktion f angewandt wird: $[\alpha]$
- Was ist der **Ergebnistyp**?

Funktionen als Werte: Funktionstypen

- ▶ Was hätte `map` für einen **Typ**?

```
map f []      = []
```

```
map f (c:cs) = f c : map f cs
```

- ▶ Was ist der Typ des **ersten Arguments**?

- ▶ Eine Funktion mit beliebigen Definitions- und Wertebereich: $\alpha \rightarrow \beta$

- ▶ Was ist der Typ des **zweiten Arguments**?

- ▶ Eine Liste, auf deren Elemente die Funktion f angewandt wird: $[\alpha]$

- ▶ Was ist der **Ergebnistyp**?

- ▶ Eine Liste von Elementen aus dem Wertebereich von f : $[\beta]$

- ▶ Alles **zusammengesetzt**:

Funktionen als Werte: Funktionstypen

- ▶ Was hätte `map` für einen **Typ**?

```
map f []      = []
map f (c:cs) = f c : map f cs
```

- ▶ Was ist der Typ des **ersten Arguments**?
 - ▶ Eine Funktion mit beliebigen Definitions- und Wertebereich: $\alpha \rightarrow \beta$
- ▶ Was ist der Typ des **zweiten Arguments**?
 - ▶ Eine Liste, auf deren Elemente die Funktion f angewandt wird: $[\alpha]$
- ▶ Was ist der **Ergebnistyp**?
 - ▶ Eine Liste von Elementen aus dem Wertebereich von f : $[\beta]$
- ▶ Alles **zusammengesetzt**:

```
map :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]
```

Was zum Selberdenken.

Die **konstante** Funktion ist

```
const :: α → β → α
const c _ = c
```

Übung 5.1: Was macht diese Funktion?

```
mystery xs = sum (map (const 1) xs)
```

Was zum Selberdenken.

Die **konstante** Funktion ist

```
const :: α → β → α
const c _ = c
```

Übung 5.1: Was macht diese Funktion?

```
mystery xs = sum (map (const 1) xs)
```

Lösung: Betrachten wir eine Beispiel-Auswertung:

```
sum (map (const 1) []) ~> sum [] ~> 0
sum (map (const 1) [True, False, True]) ~> sum [1,1,1] ~> 3
sum (map (const 1) "foobaz") ~> sum ([1,1,1,1,1,1]) ~> 6
```

Was zum Selberdenken.

Die **konstante** Funktion ist

```
const :: α → β → α
const c _ = c
```

Übung 5.1: Was macht diese Funktion?

```
mystery xs = sum (map (const 1) xs)
```

Lösung: Betrachten wir eine Beispiel-Auswertung:

```
sum (map (const 1) []) ~> sum [] ~> 0
sum (map (const 1) [True, False, True]) ~> sum [1,1,1] ~> 3
sum (map (const 1) "foobaz") ~> sum ([1,1,1,1,1,1]) ~> 6
```

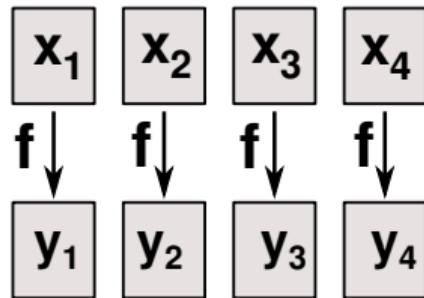
Die mysteriöse Funktion berechnet die **Länge** der Liste **xs**!

II. Map und Filter

Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



- ▶ Auswertung:
`toL "AB"`

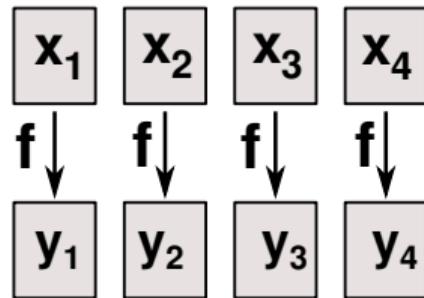
Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
```

```
map f [] = []
```

```
map f (c:cs) = f c : map f cs
```



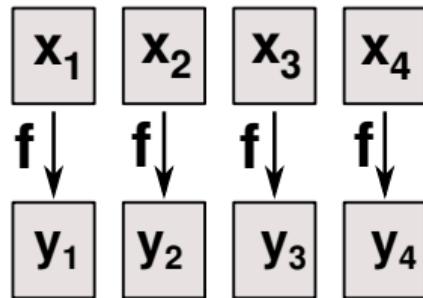
- ▶ Auswertung:

```
toL "AB"  $\rightarrow$  map toLower ('A':'B':[ ])
```

Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



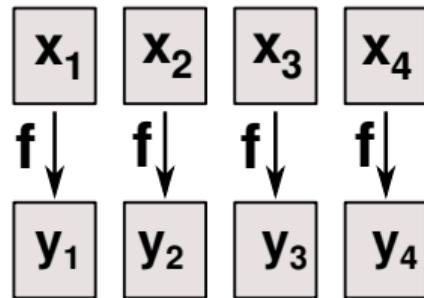
- ▶ Auswertung:

```
toL "AB"   $\rightarrow$  map toLower ('A':'B':[ ])
           $\rightarrow$  toLower 'A': map toLower ('B':[ ])
```

Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



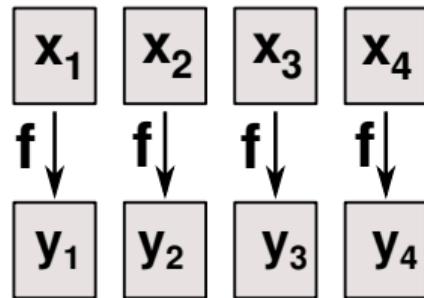
- ▶ Auswertung:

```
toL "AB"   $\rightarrow$  map toLower ('A':'B':[ ])
           $\rightarrow$  toLower 'A': map toLower ('B':[ ])
           $\rightarrow$  'a':map toLower ('B':[ ])
```

Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



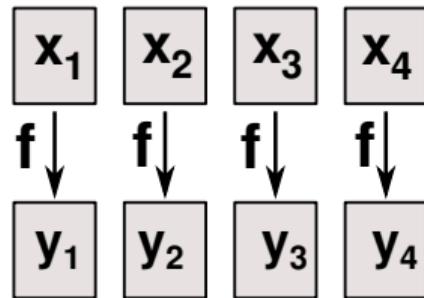
- ▶ Auswertung:

```
toL "AB"   $\rightarrow$  map toLower ('A':'B':[ ])
           $\rightarrow$  toLower 'A': map toLower ('B':[ ])
           $\rightarrow$  'a':map toLower ('B':[ ])
           $\rightarrow$  'a':toLower 'B':map toLower []
```

Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



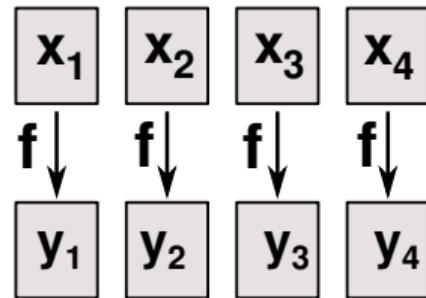
- ▶ Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
                → toLower 'A': map toLower ('B':[])
                → 'a':map toLower ('B':[])
                → 'a':toLower 'B':map toLower []
                → 'a':'b':map toLower []
```

Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



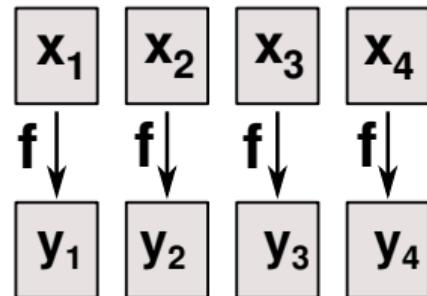
- ▶ Auswertung:

```
toL "AB"   $\rightarrow$  map toLower ('A':'B':[ ])
           $\rightarrow$  toLower 'A': map toLower ('B':[ ])
           $\rightarrow$  'a':map toLower ('B':[ ])
           $\rightarrow$  'a':toLower 'B':map toLower []
           $\rightarrow$  'a':'b':map toLower []
           $\rightarrow$  'a':'b':[]
```

Funktionen als Argumente: map

- ▶ `map` wendet Funktion auf alle Elemente an
- ▶ Signatur:

```
map :: ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $[\alpha] \rightarrow [\beta]$ 
map f []      = []
map f (c:cs) = f c : map f cs
```



- ▶ Auswertung:

```
toL "AB"  → map toLower ('A':'B':[])
                  → toLower 'A': map toLower ('B':[])
                  → 'a':map toLower ('B':[])
                  → 'a':toLower 'B':map toLower []
                  → 'a':'b':map toLower []
                  → 'a':'b':[] ≡ "ab"
```

- ▶ **Funktionsausdrücke** werden symbolisch reduziert

- ▶ Keine Änderung

Funktionen als Argumente: filter

- ▶ Elemente **filtern**: filter
- ▶ Signatur:

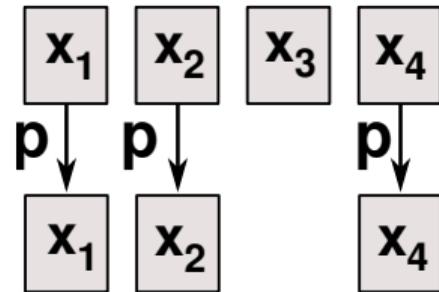
```
filter :: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$   $[\alpha] \rightarrow [\alpha]$ 
```

- ▶ Definition

```
filter p []  = []
filter p (x:xs)
  | p x       = x: filter p xs
  | otherwise = filter p xs
```

- ▶ Beispiel:

```
digits :: String  $\rightarrow$  String
digits = filter isDigit
```



Beispiel filter: Sieb des Erathostenes

- ▶ Für jede gefundene Primzahl p alle Vielfachen heraussieben:

```
sieve' :: [Integer] → [Integer]
sieve' (p:ps) = p: sieve' (filterPs ps) where
  filterPs (q: qs)
    | q `mod` p ≠ 0 = q: filterPs qs
    | otherwise       = filterPs qs
```

- ▶ „Sieb“: es werden alle q gefiltert mit $\text{mod } q \text{ } p \neq 0$

Beispiel filter: Sieb des Erathostenes

- ▶ Es werden alle `q` gefiltert mit `mod q p ≠ 0`
- ▶ **Namenlose** (anonyme) Funktion $\lambda q \rightarrow \text{mod } q \ p \neq 0$

```
sieve :: [Integer] → [Integer]
sieve (p:ps) = p: sieve (filter (λq → q `mod` p ≠ 0) ps)
```

- ▶ Damit (NB: kleinste Primzahl ist `2`):

```
primes :: [Integer]
primes = sieve [2..]
```

Beispiel filter: Sieb des Erathostenes

- ▶ Es werden alle `q` gefiltert mit `mod q p ≠ 0`
- ▶ **Namenlose** (anonyme) Funktion $\lambda q \rightarrow \text{mod } q \text{ } p \neq 0$

```
sieve :: [Integer] → [Integer]
sieve (p:ps) = p: sieve (filter (\q → q `mod` p ≠ 0) ps)
```

- ▶ Damit (NB: kleinste Primzahl ist `2`):

```
primes :: [Integer]
primes = sieve [2..]
```

- ▶ Primzahlzählfunktion $\pi(n)$:

```
pcf :: Integer → Int
pcf n = length (takeWhile (\m → m < n) primes)
```

Primzahltheorem:

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n / \log n} = 1$$



Jetzt seit ihr dran...

Für das Palindrom hatten wir eine Funktion `clean` definiert:

```
clean :: String → String
clean [] = []
clean (s:xs) | isAlphaNum s = toLower s : clean xs
            | otherwise      = clean xs
```

Übung 5.2: Clean refactored

Wie sieht `clean` mit `map` und `filter` (und **ohne Rekursion**) aus?

Jetzt seit ihr dran...

Für das Palindrom hatten wir eine Funktion `clean` definiert:

```
clean :: String → String
clean [] = []
clean (s:xs) | isAlphaNum s = toLower s : clean xs
            | otherwise      = clean xs
```

Übung 5.2: Clean refactored

Wie sieht `clean` mit `map` und `filter` (und **ohne Rekursion**) aus?

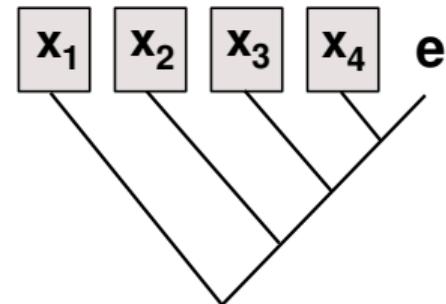
Lösung:

```
clean' :: String → String
clean' xs = map toLower (filter isAlphaNum xs)
```

III. Strukturelle Rekursion

Strukturelle Rekursion

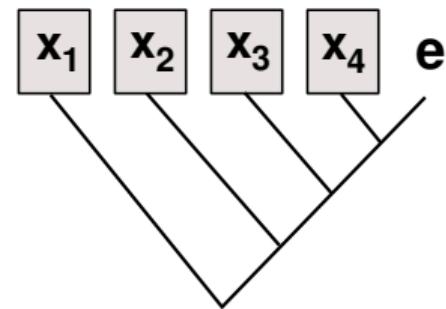
- ▶ **Strukturelle Rekursion:** gegeben durch
 - ▶ eine Gleichung für die leere Liste
 - ▶ eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)
- ▶ Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- ▶ Auswertung:



`sum [4,7,3]` →
`concat [A, B, C]` →
`length [4, 5, 6]` →

Strukturelle Rekursion

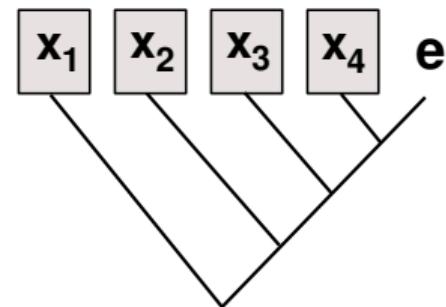
- ▶ **Strukturelle Rekursion:** gegeben durch
 - ▶ eine Gleichung für die leere Liste
 - ▶ eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)
- ▶ Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- ▶ Auswertung:



`sum [4,7,3]` → $4 + 7 + 3 + 0$
`concat [A, B, C]` →
`length [4, 5, 6]` →

Strukturelle Rekursion

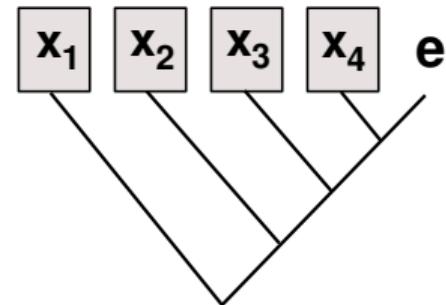
- ▶ **Strukturelle Rekursion:** gegeben durch
 - ▶ eine Gleichung für die leere Liste
 - ▶ eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)
- ▶ Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- ▶ Auswertung:



`sum [4,7,3]` → `4 + 7 + 3 + 0`
`concat [A, B, C]` → `A ++ B ++ C ++ []`
`length [4, 5, 6]` →

Strukturelle Rekursion

- ▶ **Strukturelle Rekursion:** gegeben durch
 - ▶ eine Gleichung für die leere Liste
 - ▶ eine Gleichung für die nicht-leere Liste (mit **einem** rekursiven Aufruf)
- ▶ Beispiel: `kasse`, `inventur`, `sum`, `concat`, `length`, `(++)`, ...
- ▶ Auswertung:



`sum [4,7,3]` → `4 + 7 + 3 + 0`
`concat [A, B, C]` → `A ++ B ++ C ++ []`
`length [4, 5, 6]` → `1+ 1+ 1+ 0`

Strukturelle Rekursion

► Allgemeines Muster:

$$\begin{aligned} f [] &= e \\ f (x:xs) &= x \otimes f \ xs \end{aligned}$$

► Parameter der Definition:

- Startwert (für die leere Liste) $e :: \beta$
- Rekursionsfunktion $\otimes :: \alpha \rightarrow \beta \rightarrow \beta$

► Auswertung:

$$f [x_1, \dots, x_n] = x_1 \otimes x_2 \otimes \dots \otimes x_n \otimes e$$

► **Terminiert** immer (wenn Liste endlich und \otimes, e terminieren)

Strukturelle Rekursion durch foldr

- ▶ **Strukturelle** Rekursion

- ▶ Basisfall: leere Liste
- ▶ Rekursionsfall: Kombination aus Listenkopf und Rekursionswert

- ▶ Signatur

```
foldr :: ( $\alpha \rightarrow \beta \rightarrow \beta$ )  $\rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$ 
```

- ▶ Definition

```
foldr f e []      = e
foldr f e (x:xs) = f x (foldr f e xs)
```

Beispiele: foldr

- ▶ **Summieren** von Listenelementen.

```
sum  ::  [Int] → Int
sum xs = foldr (+) 0 xs
```

- ▶ **Flachklopfen** von Listen.

```
concat  ::  [[a]] → [a]
concat xs = foldr (++) [] xs
```

- ▶ **Länge** einer Liste

```
length  ::  [a] → Int
length xs = foldr (λx n → n + 1) 0 xs
```

Beispiele: foldr

- ▶ **Konjunktion** einer Liste

```
and  ::  [Bool] → Bool
and xs = foldr (&&) True xs
```

- ▶ **Konjunktion** von Prädikaten

```
all  ::  ( $\alpha \rightarrow \text{Bool}$ ) →  $[\alpha]$  → Bool
all p xs = and (map p xs)
```

Der Shoppe, revisited.

► Kasse alt:

```
kasse :: Einkaufskorb → Int
kasse (Ekwg ps) = kasse' ps where
  kasse' [] = 0
  kasse' (p: ps) = cent p + kasse' ps
```

► Kasse neu:

```
kasse' :: Einkaufskorb → Int
kasse' (Ek ps) = foldr (λp ps → cent p + ps) 0 ps
```

Besser:

```
kasse :: Einkaufskorb → Int
kasse (Ek ps) = sum (map cent ps)
```

Der Shoppe, revisited.

- ▶ Inventur alt:

```
inventur :: Lager → Int
inventur (Lager ps) = inventur' ps where
  inventur' [] = 0
  inventur' (p: ps) = cent p + inventur' ps
```

- ▶ Suche nach einem Artikel neu:

```
inventur :: Lager → Int
inventur (Lager l) = sum (map cent l)
```

Der Shoppe, revisited.

- ▶ Suche nach einem Artikel alt:

```
suche :: Artikel → Lager → Maybe Menge
suche art (Lager ps) = suche' art ps where
  suche' art (Posten lart  m: l)
    | art == lart = Just m
    | otherwise    = suche' art l
  suche' art []    = Nothing
```

- ▶ Suche nach einem Artikel neu:

```
suche :: Artikel → Lager → Maybe Menge
suche a (Lager ps) =
  listToMaybe (map (λ(Posten _ m) → m)
    (filter (λ(Posten la _) → la == a) ps))
```

Der Shoppe, revisited.

- ## ► Kassenbon formatieren neu:

```
kassenbon :: Einkaufskorb → String
kassenbon ek@(Ek ps) =
  "Bob's Aulde Grocery Shoppe\n\n" ++
  "Artikel" ++ concatMap artikel ps ++
  "-----\n" ++
  "Menge" ++
  "Preis\n" ++
  "\n" ++
  "Summe:" ++ formatR 31 (showEuro (kasse ek))
```

artikel :: Posten → String

Noch ein Beispiel: rev

- ▶ Listen **umdrehen**:

```
rev1 :: [α] → [α]
rev1 []      = []
rev1 (x:xs) = rev1 xs ++ [x]
```

- ▶ Mit **foldr**:

```
rev2 :: [α] → [α]
rev2 = foldr (λx xs → xs ++ [x]) []
```

- ▶ Unbefriedigend: doppelte Rekursion $O(n^2)$!

Iteration mit foldl

- foldr faltet von rechts:

$$\text{foldr } \otimes [x_1, \dots, x_n] \ e = x_1 \otimes x_2 \ (x_2 \otimes (\dots (x_n \otimes e)))$$

Iteration mit foldl

- foldr faltet von rechts:

$$\text{foldr } \otimes [x_1, \dots, x_n] \ e = x_1 \otimes x_2 \ (x_2 \otimes (\dots (x_n \otimes e)))$$

- Warum nicht **andersherum**?

$$\text{foldl } \otimes [x_1, \dots, x_n] \ e = (((e \otimes x_1) \otimes x_2) \dots) \otimes x_n$$

Iteration mit foldl

- foldr faltet von rechts:

$$\text{foldr } \otimes [x_1, \dots, x_n] \ e = x_1 \otimes x_2 \ (x_2 \otimes (\dots (x_n \otimes e)))$$

- Warum nicht **andersherum**?

$$\text{foldl } \otimes [x_1, \dots, x_n] \ e = (((e \otimes x_1) \otimes x_2) \dots) \otimes x_n$$

- Definition von foldl:

```
foldl :: ( $\alpha \rightarrow \beta \rightarrow \alpha$ )  $\rightarrow \alpha \rightarrow [\beta] \rightarrow \alpha$ 
```

```
foldl f a [] = a
```

```
foldl f a (x:xs) = foldl f (f a x) xs
```

- foldl ist ein **Iterator** mit Anfangszustand **e**, Iterationsfunktion \otimes
- Entspricht einfacher Iteration (**for**-Schleife)

Beispiel: rev revisited

- ▶ Listenumkehr **endrekursiv**:

```
rev3 :: [α] → [α]
rev3 xs = rev0 xs [] where
  rev0 []      ys = ys
  rev0 (x:xs) ys = rev0 xs (x:ys)
```

- ▶ Listenumkehr durch falten **von links**:

```
rev4 :: [α] → [α]
rev4 = foldl (λxs x → x: xs) []
```

```
rev5 :: [α] → [α]
rev5 = foldl (flip (:)) []
```

- ▶ Nur noch **eine** Rekursion $O(n)!$

foldr vs. foldl

- ▶ $f = \text{foldr} \otimes e$ entspricht

$$f [] = e$$

$$f (x:xs) = x \otimes f \ xs$$

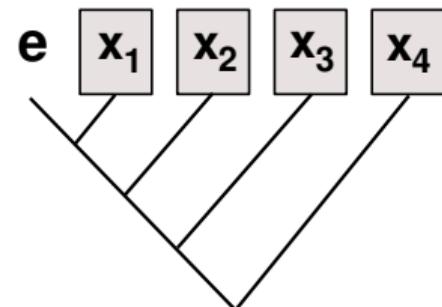
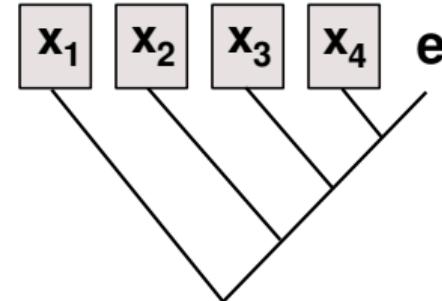
- ▶ **Nicht-strikt** in xs , z.B. `and`, `or`
- ▶ Konsumiert nicht immer die ganze Liste
- ▶ Auch für unendliche Listen anwendbar
- ▶ $f = \text{foldl} \otimes e$ entspricht

$$f \ xs = g \ e \ xs \text{ where}$$

$$g \ a [] = a$$

$$g \ a \ (x:xs) = g \ (a \otimes x) \ xs$$

- ▶ Effizient (endrekursiv) und **strikt** in xs
- ▶ Konsumiert immer die ganze Liste
- ▶ Divergiert immer für unendliche Listen



Wann ist foldl = foldr?

Definition (Monoid)

(\otimes, e) ist ein **Monoid** wenn

$$e \otimes x = x$$

(Neutrales Element links)

$$x \otimes e = x$$

(Neutrales Element rechts)

$$(x \otimes y) \otimes z = x \otimes (y \otimes z)$$

(Assoziativit)

Theorem

Wenn (\otimes, e) **Monoid** und \otimes strikt, dann gilt fr alle e, xs

$$foldl \otimes e xs = foldr \otimes e xs$$

- ▶ Beispiele: concat, sum, product, length, reverse
- ▶ Gegenbeispiel: all, any (nicht-strikt)

Übersicht: vordefinierte Funktionen auf Listen II

map	:: $(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$	— Auf alle Elemente anwenden
filter	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$	— Elemente filtern
foldr	:: $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$	— Falten von rechts
foldl	:: $(\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$	— Falten von links
mapConcat	:: $(\alpha \rightarrow [\beta]) \rightarrow [\alpha] \rightarrow [\beta]$	— map und concat
takeWhile	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$	— längster Prefix mit p
dropWhile	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$	— Rest von takeWhile
span	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow ([\alpha], [\alpha])$	— takeWhile und dropWhile
all	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow \text{Bool}$	— Argument gilt für alle
any	:: $(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow \text{Bool}$	— Argument gilt mind. einmal
elem	:: $(\text{Eq } \alpha) \Rightarrow \alpha \rightarrow [\alpha] \rightarrow \text{Bool}$	— Ist Element enthalten?
zipWith	:: $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow [\gamma]$	— verallgemeinertes zip

► Mehr: siehe `Data.List`

Jetzt seit ihr dran!

Übung 5.3: elem selbstgemacht

Wie könnte die vordefinierte Funktion

`elem :: (Eq α) → α → [α] → Bool`

definiert sein?

Jetzt seit ihr dran!

Übung 5.3: elem selbstgemacht

Wie könnte die vordefinierte Funktion

`elem :: (Eq α) → α → [α] → Bool`

definiert sein?

Lösung: Eine Möglichkeit:

```
elem x xs = not (null (filter (λy → x == y) xs))
```

Jetzt seit ihr dran!

Übung 5.3: elem selbstgemacht

Wie könnte die vordefinierte Funktion

`elem :: (Eq α) → α → [α] → Bool`

definiert sein?

Lösung: Eine Möglichkeit:

`elem x xs = not (null (filter (λy → x == y) xs))`

oder auch

`elem x = not ∘ null ∘ filter (x ==)`

IV. Funktionen Höherer Ordnung

Funktionen als Argumente: Funktionskomposition

► Funktionskomposition (mathematisch)

$(\circ) :: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$
 $(f \circ g) \ x = f \ (g \ x)$

- Vordefiniert
- Lies: f nach g
- Funktionskomposition **vorwärts**:

$(>.) :: (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$
 $(f >.) g \ x = g \ (f \ x)$

- **Nicht** vordefiniert

η -Kontraktion

- ▶ “ $>.>$ ist dasselbe wie
 - nur mit vertauschten Argumenten”
- ▶ Vertauschen der **Argumente** (vordefiniert):

```
flip :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow \beta \rightarrow \alpha \rightarrow \gamma$ 
```

```
flip f b a = f a b
```

η -Kontraktion

- ▶ “ $>.$ ” ist dasselbe wie \circ nur mit vertauschten Argumenten”
- ▶ Vertauschen der **Argumente** (vordefiniert):

```
flip :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow \beta \rightarrow \alpha \rightarrow \gamma$ 
flip f b a = f a b
```

- ▶ Damit Funktionskomposition vorwärts:

```
( $>.$ ) :: ( $\alpha \rightarrow \beta$ )  $\rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$ 
( $>.$ ) = flip ( $\circ$ )
```

- ▶ **Da fehlt doch was?!**

η -Kontraktion

- " $>.>$ ist dasselbe wie \circ nur mit vertauschten Argumenten"
- Vertauschen der **Argumente** (vordefiniert):

```
flip :: ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow \beta \rightarrow \alpha \rightarrow \gamma$ 
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
( $>.>$ ) :: ( $\alpha \rightarrow \beta$ )  $\rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$ 
( $>.>$ ) = flip ( $\circ$ )
```

- **Da fehlt doch was?!** Nein:

```
( $>.>$ ) f g a = flip ( $\circ$ ) f g a     $\equiv$     ( $>.>$ ) = flip ( $\circ$ )
```

- Warum?

η -Äquivalenz und η -Kontraktion

η -Äquivalenz

Sei f eine Funktion $f : A \rightarrow B$, dann gilt $f = \lambda x. f x$

► In Haskell: η -Kontraktion

- Bedingung: Ausdruck $E :: \alpha \rightarrow \beta$, Variable $x :: \alpha$, E darf x nicht enthalten

$$\lambda x \rightarrow E x \equiv E$$

► Spezialfall Funktionsdefinition (punktfreie Notation)

$$f x = E x \equiv f = E$$

► Hier:

$$(>.) f g a = \text{flip } (\circ) f g a \equiv (>.) f g a = \text{flip } (\circ) f g a$$

η -Äquivalenz und η -Kontraktion

η -Äquivalenz

Sei f eine Funktion $f : A \rightarrow B$, dann gilt $f = \lambda x. f x$

► In Haskell: η -Kontraktion

- Bedingung: Ausdruck $E :: \alpha \rightarrow \beta$, Variable $x :: \alpha$, E darf x nicht enthalten

$$\lambda x \rightarrow E x \equiv E$$

► Spezialfall Funktionsdefinition (punktfreie Notation)

$$f x = E x \equiv f = E$$

► Hier:

$$(>.) f g a = \text{flip } (\circ) f g a \equiv (>.) f g = \text{flip } (\circ) f g$$

η -Äquivalenz und η -Kontraktion

η -Äquivalenz

Sei f eine Funktion $f : A \rightarrow B$, dann gilt $f = \lambda x. f x$

► In Haskell: η -Kontraktion

- Bedingung: Ausdruck $E :: \alpha \rightarrow \beta$, Variable $x :: \alpha$, E darf x nicht enthalten

$$\lambda x \rightarrow E x \equiv E$$

► Spezialfall Funktionsdefinition (punktfreie Notation)

$$f x = E x \equiv f = E$$

► Hier:

$$(>.) f g a = \text{flip } (\circ) f g a \equiv (>.) f = \text{flip } (\circ) f$$

η -Äquivalenz und η -Kontraktion

η -Äquivalenz

Sei f eine Funktion $f : A \rightarrow B$, dann gilt $f = \lambda x. f x$

► In Haskell: η -Kontraktion

- Bedingung: Ausdruck $E :: \alpha \rightarrow \beta$, Variable $x :: \alpha$, E darf x nicht enthalten

$$\lambda x \rightarrow E x \equiv E$$

► Spezialfall Funktionsdefinition (punktfreie Notation)

$$f x = E x \equiv f = E$$

► Hier:

$$(>.) f g a = \text{flip } (\circ) f g a \equiv (>.) = \text{flip } (\circ)$$

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$\alpha \rightarrow \beta \rightarrow \gamma \equiv \alpha \rightarrow (\beta \rightarrow \gamma)$$

- **Inbesondere**: $(\alpha \rightarrow \beta) \rightarrow \gamma \neq \alpha \rightarrow (\beta \rightarrow \gamma)$
- Funktionsanwendung ist linksassoziativ:

$$f \ a \ b \equiv (f \ a) \ b$$

- **Inbesondere**: $f \ (a \ b) \neq (f \ a) \ b$

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$\alpha \rightarrow \beta \rightarrow \gamma \equiv \alpha \rightarrow (\beta \rightarrow \gamma)$$

- **Inbesondere**: $(\alpha \rightarrow \beta) \rightarrow \gamma \neq \alpha \rightarrow (\beta \rightarrow \gamma)$

- Funktionsanwendung ist linksassoziativ:

$$f \ a \ b \equiv (f \ a) \ b$$

- **Inbesondere**: $f (a \ b) \neq (f \ a) \ b$
- **Partielle** Anwendung von Funktionen:

- Für $f :: \alpha \rightarrow \beta \rightarrow \gamma$, $x :: \alpha$ ist $f \ x :: \beta \rightarrow \gamma$

- Beispiele:

- `map toLower :: String → String`
- `(3 ==) :: Int → Bool`
- `concat ∘ map (replicate 2) :: String → String`

Zusammenfassung

- ▶ Funktionen **höherer Ordnung**
 - ▶ Funktionen als **gleichberechtigte Objekte** und Argumente
 - ▶ Spezielle Funktionen höherer Ordnung: `map`, `filter`, `fold` und Freunde
- ▶ Formen der **Rekursion**:
 - ▶ Strukturelle Rekursion entspricht `foldr`
 - ▶ Iteration entspricht `foldl`
- ▶ Partielle Applikation, η -Äquivalenz, namenlose Funktionen
- ▶ Nächste Woche: Rekursive und zyklische Datenstrukturen

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 6 vom 07.12.2020: Rekursive und zyklische Datenstrukturen

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Fahrplan

- ▶ **Teil I: Funktionale Programmierung im Kleinen**
 - ▶ Einführung
 - ▶ Funktionen
 - ▶ Algebraische Datentypen
 - ▶ Typvariablen und Polymorphie
 - ▶ Funktionen höherer Ordnung I
 - ▶ Rekursive und zyklische Datenstrukturen
 - ▶ Funktionen höherer Ordnung II
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

- ▶ **Rekursive** Datentypen und **zyklische** Daten
 - ▶ ... und wozu sie nützlich sind
 - ▶ Fallbeispiel: Labyrinth
- ▶ Performance-Aspekte

Lernziele

- 1 Wir verstehen, wie in Haskell „unendliche“ Datenstrukturen modelliert werden. Warum sind unendliche Listen nicht wirklich unendlich?
- 2 Wir wissen, worauf wir achten müssen, wenn uns die Geschwindigkeit unserer Haskell-Programme wichtig ist.

I. Rekursive und Zyklische Datenstrukturen

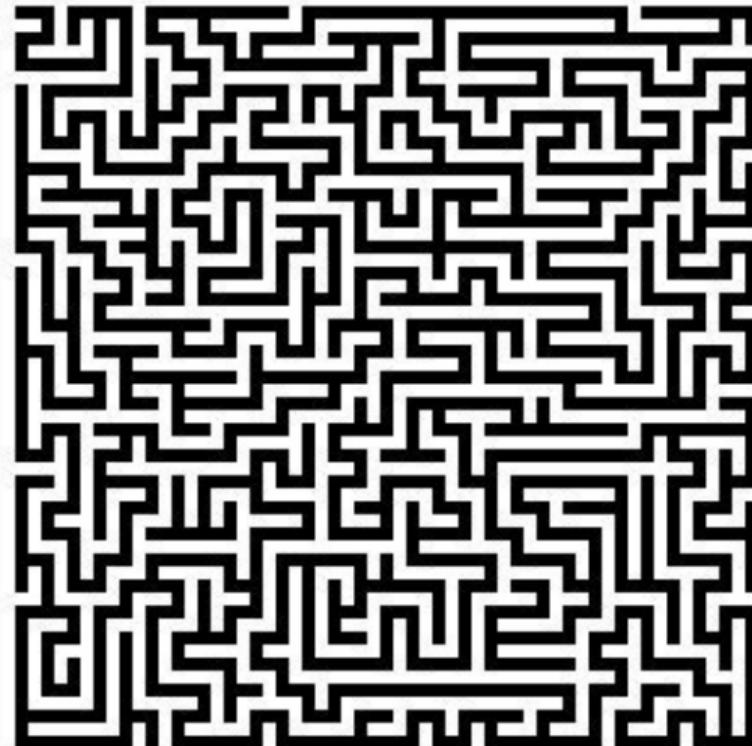
Konstruktion zyklischer Datenstrukturen

- ▶ **Zyklische** Datenstrukturen haben keine **endliche freie** Repräsentation
 - ▶ Nicht durch endlich viele Konstruktoren darstellbar
 - ▶ Sondern durch Konstruktoren und **Gleichungen**
- ▶ Einfaches Beispiel:

```
ones = 1 : ones
```

- ▶ Nicht-Striktheit erlaubt einfache Definition von Funktionen auf zyklische Datenstrukturen
- ▶ Aber: Funktionen können **divergieren**

Fallbeispiel: Zyklische Datenstrukturen



Quelle: docs.gimp.org

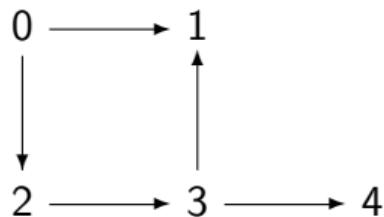
Modellierung eines Labyrinths

- ▶ Ein **gerichtetes** Labyrinth ist entweder
 - ▶ eine Sackgasse,
 - ▶ ein Weg, oder
 - ▶ eine Abzweigung in zwei Richtungen.
- ▶ Jeder Knoten im Labyrinth hat ein Label α .

```
data Lab α = Dead α
          | Pass α (Lab α)
          | TJnc α (Lab α) (Lab α)
```

Definition von Labyrinthen

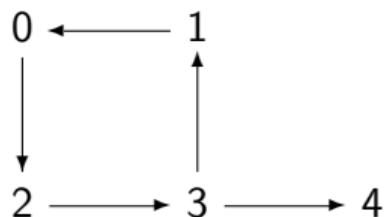
Ein einfaches Labyrinth ohne Zyklen:



Definition in Haskell:

```
s0 = TJnc 0 s1 s2
s1 = Dead 1
s2 = Pass 2 s3
s3 = TJnc 3 s1 s4
s4 = Dead 4
```

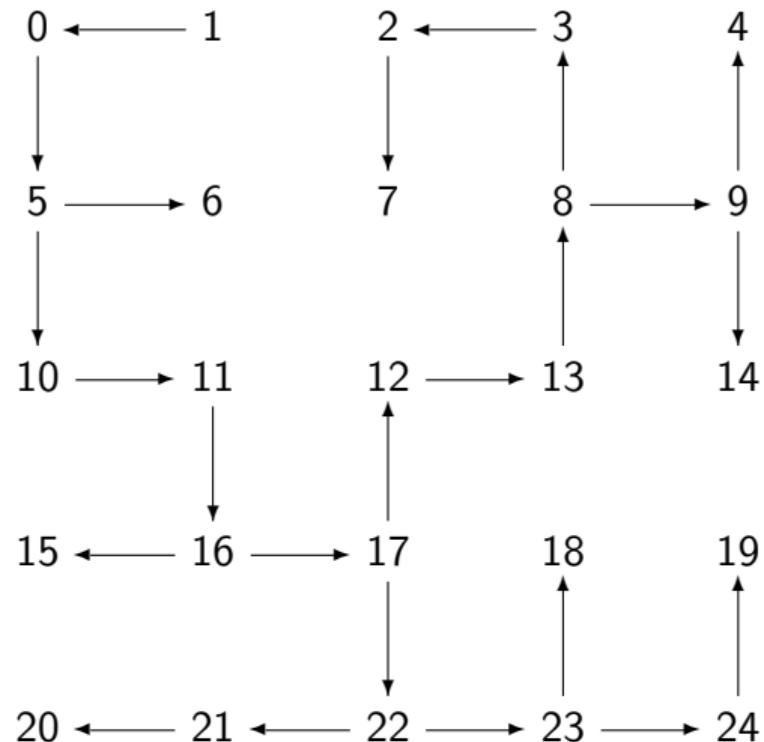
Ein einfaches Labyrinth mit Zyklen:



Definition in Haskell:

```
t0 = Pass 0 t2
t1 = Pass 1 t0
t2 = Pass 2 t3
t3 = TJnc 3 t1 t4
t4 = Dead 4
```

Ein Labyrinth (zyklenfrei)



Traversierung des Labyrinths

- Ziel: **Pfad** zu einem gegebenen **Ziel** finden

- Benötigt **Pfade** und **Traversierung**

- Pfade: Liste von Knoten

```
type Path α = [α]
```

- Traversierung: erfolgreich (Pfad) oder nicht erfolgreich

```
type Trav α = Maybe [α]
```

Traversionsstrategie

- ▶ Geht erstmal von **zyklenfreien** Labyrinth aus
- ▶ An jedem Knoten prüfen, ob Ziel erreicht, ansonsten
 - ▶ an Sackgasse: Fehlschlag (**Nothing**)
 - ▶ an Passagen: Weiterlaufen

```
cons :: α → Trav α → Trav α
cons _ Nothing      = Nothing
cons i (Just is) = Just (i: is)
```

- ▶ an Kreuzungen: Auswahl treffen

```
select :: Trav α → Trav α → Trav α
select Nothing t = t
select t      _ = t
```

- ▶ Erfordert Propagation von Fehlschlägen (in **cons** und **select**)

Zyklenfreie Traversion

- Zusammengesetzt:

```
traverse_1 :: (Show α, Eq α) ⇒ α → Lab α → Trav α
traverse_1 t l
| nid l == t = Just [nid l]
| otherwise = case l of
  Dead _ → Nothing
  Pass i n → cons i (traverse_1 t n)
  TJnc i n m → cons i (select (traverse_1 t n)
                                (traverse_1 t m))
```



Zyklenfreie Traversal

- Zusammengesetzt:

```
traverse_1 :: (Show α, Eq α) ⇒ α → Lab α → Trav α
traverse_1 t l
| nid l == t = Just [nid l]
| otherwise = case l of
  Dead _ → Nothing
  Pass i n → cons i (traverse_1 t n)
  TJnc i n m → cons i (select (traverse_1 t n)
                                (traverse_1 t m))
```



- Wie mit Zyklen umgehen?
- An jedem Knoten prüfen ob schon im Pfad enthalten.

Traversierung mit Zyklen

- ▶ Veränderte **Strategie**: Pfad bis hierher übergeben
- ▶ Pfad muss hinten erweitert werden ($O(n)$)
- ▶ Besser: Pfad **vorne** erweitern ($O(1)$), am Ende umdrehen
- ▶ Wenn **aktueller** Knoten in bisherigen Pfad **enthalten** ist, Fehlschlag
- ▶ Ansonsten wie oben

Traversierung mit Zyklen

```
traverse_2 :: Eq α ⇒ α → Lab α → Trav α
traverse_2 t l = trav_2 l [] where
  trav_2 l p
    | nid l == t = Just (reverse (nid l: p))
    | elem (nid l) p = Nothing
    | otherwise = case l of
        Dead _ → Nothing
        Pass i n → trav_2 n (i: p)
        TJnc i n m → select (trav_2 n (i: p)) (trav_2 m (i: p))
```

► Kritik:

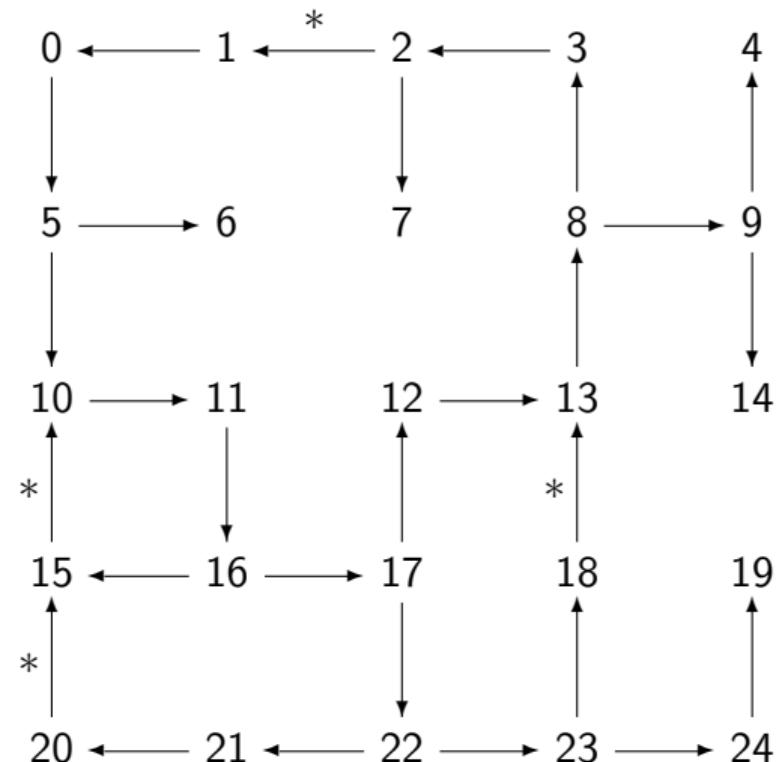
Traversierung mit Zyklen

```
traverse_2 :: Eq α ⇒ α → Lab α → Trav α
traverse_2 t l = trav_2 l [] where
  trav_2 l p
    | nid l == t = Just (reverse (nid l: p))
    | elem (nid l) p = Nothing
    | otherwise = case l of
        Dead _ → Nothing
        Pass i n → trav_2 n (i: p)
        TJnc i n m → select (trav_2 n (i: p)) (trav_2 m (i: p))
```

► Kritik:

- Prüfung `elem` immer noch $O(n)$
- Abhilfe: **Menge** der besuchten Knoten getrennt von aufgebautem **Pfad**
- Erfordert effiziente Datenstrukturen für Mengen (`Data.Set`, `Data.IntSet`) → später

Ein Labyrinth (mit Zyklen)



Der allgemeine Fall: variadische Bäume

- ▶ Labyrinth → **Graph** oder **Baum**
- ▶ Labyrinth mit mehr als 2 Nachfolgern: **variadischer Baum**

```
data VTree α = NT α [VTree α]
```

- ▶ Kürzere Definition erlaubt einfachere Funktionen:

```
traverse :: Eq α ⇒ α → VTree α → Maybe [α]
traverse t vt = trav [] vt where
  trav p (NT l vs)
  | l == t = Just (reverse (l: p))
  | elem l p = Nothing
  | otherwise = select (map (trav (l: p)) vs)
```



Traversierung verallgemeinert

- Änderung der Parameter der Traversionsfunktion `trav`:

```
trav :: Eq α ⇒ [(VTree α, [α])] → Maybe [α]
```

- Liste der nächsten **Kandidaten** mit **Pfad** der dorthin führt.
- Algorithmus:
 - 1 Wenn Liste leer, Fehlschlag
 - 2 Wenn Liste nicht leer, ist der aktuelle Knoten der Kopf der Liste.
 - 3 Prüfe, ob aktueller Knoten das Ziel ist.
 - 4 Wenn nicht am Ziel und aktueller Knoten schon besucht, nächsten Kandidaten traversieren
 - 5 Ansonsten füge Kinder des aktuellem Knotens mit aktuellem Pfad zu Kandidaten hinzu und traversiere weiter

Traversierung verallgemeinert

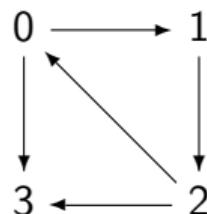
- Änderung der Parameter der Traversionsfunktion `trav`:

```
trav :: Eq α ⇒ [(VTree α, [α])] → Maybe [α]
```

- Liste der nächsten **Kandidaten** mit **Pfad** der dorthin führt.
- Algorithmus:
 - ① Wenn Liste leer, Fehlschlag
 - ② Wenn Liste nicht leer, ist der aktuelle Knoten der Kopf der Liste.
 - ③ Prüfe, ob aktueller Knoten das Ziel ist.
 - ④ Wenn nicht am Ziel und aktueller Knoten schon besucht, nächsten Kandidaten traversieren
 - ⑤ Ansonsten füge Kinder des aktuellem Knotens mit aktuellem Pfad zu Kandidaten hinzu und traversiere weiter
- Tiefensuche: Kinder **vorne** anfügen (Kandidatenliste ist ein **Stack**)
- Breitensuche: Kinder **hinten** anhängen (Kandidatenliste ist eine **Queue**)
- Andere Bewertungen möglich

Ein einfaches Beispiel

Ein einfaches Labyrinth mit Zyklen:



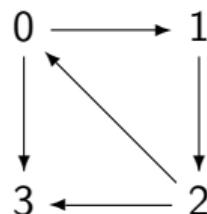
Definition in Haskell:

```
100 = NT 0 [101, 103]  
101 = NT 1 [102]  
102 = NT 2 [100, 103]  
103 = NT 3 [100]
```

- ▶ Gesucht: Pfad von 0 zu 3

Ein einfaches Beispiel

Ein einfaches Labyrinth mit Zyklen:



Definition in Haskell:

```
100 = NT 0 [101, 103]
101 = NT 1 [102]
102 = NT 2 [100, 103]
103 = NT 3 [100]
```

- ▶ Gesucht: Pfad von 0 zu 3
- ▶ Tiefensuche: [0, 1, 2, 3]
- ▶ Breitensuche: [0, 3]

Tiefensuche

```
depth_first_search :: Eq α⇒ α→ VTree α→ Maybe [α]
depth_first_search t vt = trav [(vt, [])] where
  trav [] = Nothing
  trav ((NT l ch, p):rest)
    | l == t    = Just (reverse (l:p))
    | elem l p  = trav rest
    | otherwise = trav (more++ rest) where
      more = map (λc→ (c, l:p)) ch
```

Breitensuche

```
breadth_first_search :: Eq α⇒ α→ VTree α→ Maybe [α]
breadth_first_search t vt = trav [(vt, [])] where
  trav [] = Nothing
  trav ((NT l ch, p):rest)
    | l == t    = Just (reverse (l:p))
    | elem l p  = trav rest
    | otherwise = trav (rest ++ more) where
      more = map (λc→ (c, l:p)) ch
```

Was zum Nachdenken

Übung 6.1: Wo ist der Stack?

Wo ist der Stack bei `traverse`, und warum lässt sich `traverse` nicht zu Breitensuche verallgemeinern?

Was zum Nachdenken

Übung 6.1: Wo ist der Stack?

Wo ist der Stack bei `traverse`, und warum lässt sich `traverse` nicht zu Breitensuche verallgemeinern?

Lösung: Der Stack ist bei `traverse` der Aufruf-Stack, implizit in dieser Zeile:

```
select (map (trav (l: p)) vs)
```

Hier werden die Kinder in Stack-Order aufgerufen (Kinder der Kinder vor Geschwistern). Die Traversionsfunktion `trav` der Tiefen/Breitensuche hat dagegen keinen Aufruf-Stack; sie ist **endrekursiv** (und damit potenziell effizienter).

II. Vorteile der Nicht-Strikten Auswertung

Zyklische Listen

- ▶ Durch Gleichungen können wir **zyklische** Listen definieren.

```
nats :: [Integer]
nats = natsfrom 0 where
  natsfrom i = i: natsfrom (i+1)
```

- ▶ Repräsentation durch endliche, zyklische Datenstruktur

- ▶ Kopf wird nur **einmal** ausgewertet.

```
fives :: [Integer]
fives = trace "***\u21d3Foo!\u21d3***" 5 : fives
```



- ▶ Es gibt keine **unendlichen** Listen, es gibt nur Berechnungen von Listen, die nicht terminieren.

Unendliche Weiten?

- ▶ Verschiedene Ebenen:
 - ▶ Mathematisch — unendliche Strukturen (natürliche Zahlen, Listen)
 - ▶ Implementierung — immer endlich (kann unendliche Strukturen **repräsentieren**)
- ▶ Berechnung auf unendlichen Strukturen: Vereinigung der Berechnungen auf allen **endlichen** Teilstrukturen
- ▶ Jede Berechnung hat **endlich** viele Parameter.
- ▶ Daher nicht entscheidbar, ob Liste „unendlich“ (zyklisch) ist:

```
isCyclic :: [a] → Bool
```

Unendliche Listen und Nicht-Striktheit

- ▶ Nicht-Striktheit macht den Umgang mit zyklischen Datenstrukturen einfacher
- ▶ Beispiel: Sieb des Eratosthenes:
 - ▶ Ab wo muss ich sieben, um die n -Primzahl zu berechnen?
 - ▶ Einfacher: Liste **aller** Primzahlen berechnen, davon n -te selektieren.

Fibonacci-Zahlen

- ▶ Aus der Kaninchenzucht.
- ▶ Sollte jeder Informatiker kennen.

```
fib1 :: Integer → Integer
fib1 0 = 1
fib1 1 = 1
fib1 n = fib1 (n-1)+ fib1 (n-2)
```

- ▶ Problem: **exponentieller Aufwand**.

Fibonacci-Zahlen

- Lösung: zuvor berechnete **Teilergebnisse wiederverwenden**.
- Sei `fibs :: [Integer]` Strom aller Fibonaccizahlen:

```
    fibs  ~> [1, 1, 2, 3, 5, 8, 13, 21, 34, 55 .. ]  
tail fibs ~> [1, 2, 3, 5, 8, 13, 21, 34, 55 .. ]  
tail (tail fibs) ~> [2, 3, 5, 8, 13, 21, 34, 55...]
```

Fibonacci-Zahlen

- Lösung: zuvor berechnete **Teilergebnisse wiederverwenden**.
- Sei `fibs :: [Integer]` Strom aller Fibonaccizahlen:

```
fibs  ~> [1, 1, 2, 3, 5, 8, 13, 21, 34, 55 .. ]  
tail fibs ~> [1, 2, 3, 5, 8, 13, 21, 34, 55 .. ]  
tail (tail fibs) ~> [2, 3, 5, 8, 13, 21, 34, 55...]
```

- Damit ergibt sich:

```
fibs :: [Integer]  
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- n -te Fibonaccizahl mit `fibs !! n`:

```
fib2 :: Integer → Integer  
fib2 n = genericIndex fibs n
```

- **Aufwand: linear**, da `fibs` nur einmal ausgewertet wird.

Was zum Nachdenken.

Übung 6.1: Fibonacci

Es gibt eine geschlossene Formel für die Fibonacci-Zahlen:

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right)$$

In Haskell (zählt ab 0):

```
fib3 :: Integer → Integer
fib3 n = round ((1/sqrt 5)*(((1+ sqrt 5)/2)^(n+1)-((1- sqrt 5)/2)^(n+1)))
```

Was ist hier das Problem?

Was zum Nachdenken.

Übung 6.1: Fibonacci

Es gibt eine geschlossene Formel für die Fibonacci-Zahlen:

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right)$$

In Haskell (zählt ab 0):

```
fib3 :: Integer → Integer
fib3 n = round ((1/sqrt 5)*(((1+ sqrt 5)/2)^(n+1)-((1- sqrt 5)/2)^(n+1)))
```

Was ist hier das Problem?

Lösung: Die Fließkommaarithmetik wird irgendwann (ab 74) ungenau.

III. Effizienzerwägungen

Beispiel: Listen umdrehen

- ▶ Liste umdrehen, **nicht** endrekursiv:

```
rev' :: [a] → [a]
rev' []      = []
rev' (x:xs) = rev' xs ++ [x]
```

- ▶ Hängt auch noch hinten an — $O(n^2)$!

Beispiel: Listen umdrehen

- ▶ Liste umdrehen, **nicht** endrekursiv:

```
rev' :: [a] → [a]
rev' []      = []
rev' (x:xs) = rev' xs ++ [x]
```

- ▶ Hängt auch noch hinten an — $O(n^2)$!
- ▶ Liste umdrehen, **endrekursiv** und $O(n)$:

```
rev :: [a] → [a]
rev xs = rev0 xs [] where
  rev0 []      ys = ys
  rev0 (x:xs) ys = rev0 xs (x:ys)
```

- ▶ Schneller weil geringere Aufwandsklasse, nicht nur wg. Endrekursion
- ▶ Frage: ist Endrekursion immer schneller?

Beispiel: Fakultät

- Fakultät **nicht** endrekursiv:

```
fac1 :: Integer → Integer
fac1 n = if n == 0 then 1 else n * fac1 (n-1)
```

Beispiel: Fakultät

- Fakultät **nicht** endrekursiv:

```
fac1 :: Integer → Integer
fac1 n  = if n == 0 then 1 else n * fac1 (n-1)
```

- Fakultät endrekursiv:

```
fac2 :: Integer → Integer
fac2 n      = fac' n 1 where
  fac' :: Integer → Integer → Integer
  fac' n acc = if n == 0 then acc
                else fac' (n-1) (n*acc)
```

- `fac1` verbraucht Stack, `fac2` nicht.

Beispiel: Fakultät

- Fakultät **nicht** endrekursiv:

```
fac1 :: Integer → Integer
fac1 n  = if n == 0 then 1 else n * fac1 (n-1)
```

- Fakultät endrekursiv:

```
fac2 :: Integer → Integer
fac2 n      = fac' n 1 where
  fac' :: Integer → Integer → Integer
  fac' n acc = if n == 0 then acc
                else fac' (n-1) (n*acc)
```

- `fac1` verbraucht Stack, `fac2` nicht.
- Ist **nicht** merklich schneller?!

Verzögerte Auswertung und Speicherlecks

- ▶ **Garbage collection** gibt unbenutzten Speicher wieder frei.
 - ▶ **Unbenutzt**: Bezeichner nicht mehr Speicher im **erreichbar**
- ▶ Verzögerte Auswertung **effizient**, weil nur bei **Bedarf** ausgewertet wird
 - ▶ Aber Achtung: **Speicherleck!**

Verzögerte Auswertung und Speicherlecks

- ▶ **Garbage collection** gibt unbenutzten Speicher wieder frei.
 - ▶ **Unbenutzt**: Bezeichner nicht mehr Speicher im **erreichbar**
- ▶ Verzögerte Auswertung **effizient**, weil nur bei **Bedarf** ausgewertet wird
 - ▶ Aber Achtung: **Speicherleck!**
- ▶ Eine Funktion hat ein **Speicherleck**, wenn Speicher **unnötig** lange im Zugriff bleibt.
 - ▶ “Echte” Speicherlecks wie in C/C++ nicht möglich.
- ▶ Beispiel: `fac2`
 - ▶ Zwischenergebnisse werden **nicht** auswertet.
 - ▶ Insbesondere ärgerlich bei **nicht-terminierenden** Funktionen.

Striktheit

- ▶ **Strikte Argumente** erlauben Auswertung **vor** Aufruf
 - ▶ Dadurch **konstanter** Platz bei **Endrekursion**.
- ▶ Erzwungene Striktheit: `seq` :: $\alpha \rightarrow \beta \rightarrow \beta$

$\perp \text{ 'seq' } b = \perp$

$a \text{ 'seq' } b = b$

- ▶ `seq` vordefiniert (nicht **in** Haskell definierbar)
- ▶ $(\$!)$:: $(a \rightarrow b) \rightarrow a \rightarrow b$ strikte Funktionsanwendung

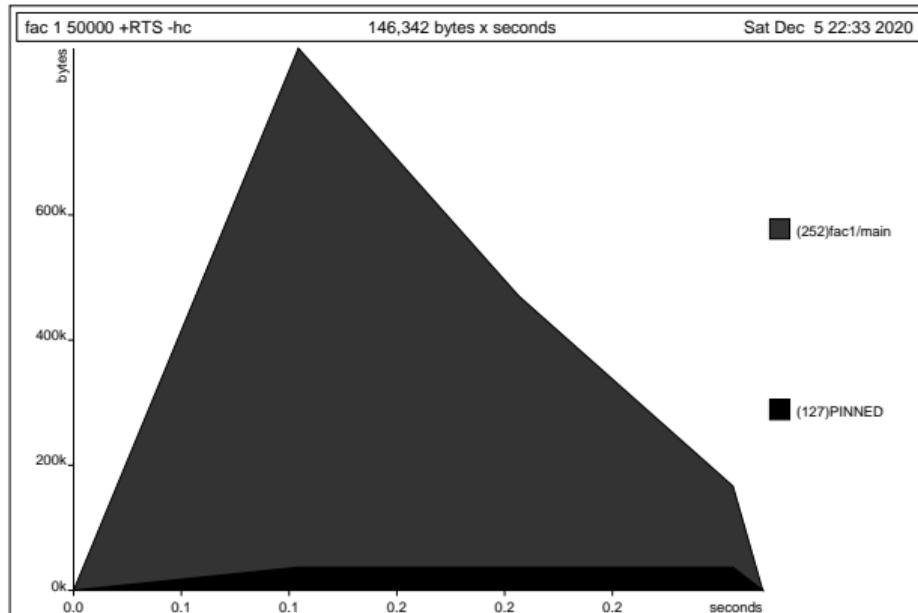
`f \$! x = x 'seq' f x`

- ▶ `ghc` macht Striktheitsanalyse

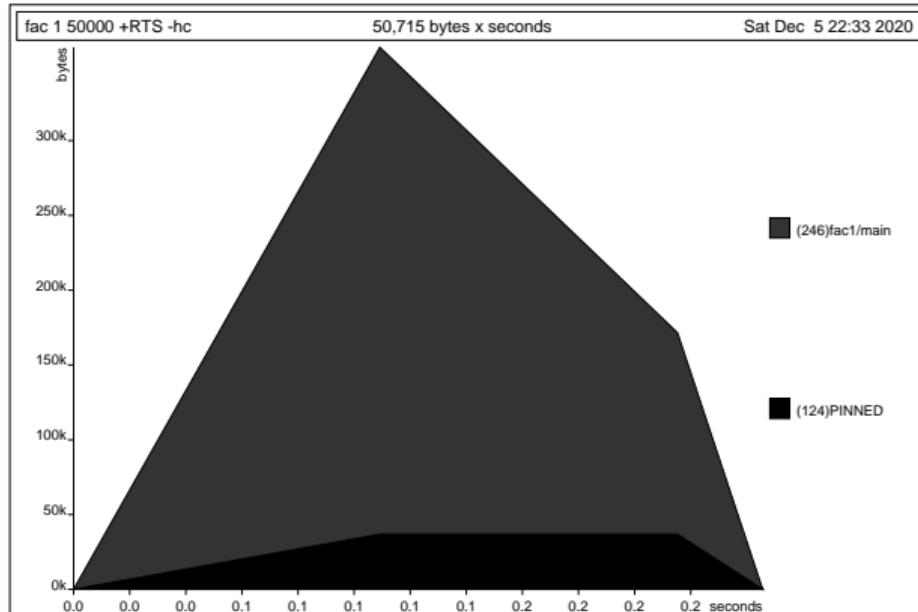
- ▶ Fakultät in konstantem Platzaufwand

```
fac3 :: Integer → Integer
fac3 n = fac' n 1 where
  fac' n acc = seq acc (if n == 0 then acc
                        else fac' (n-1) (n*acc))
```

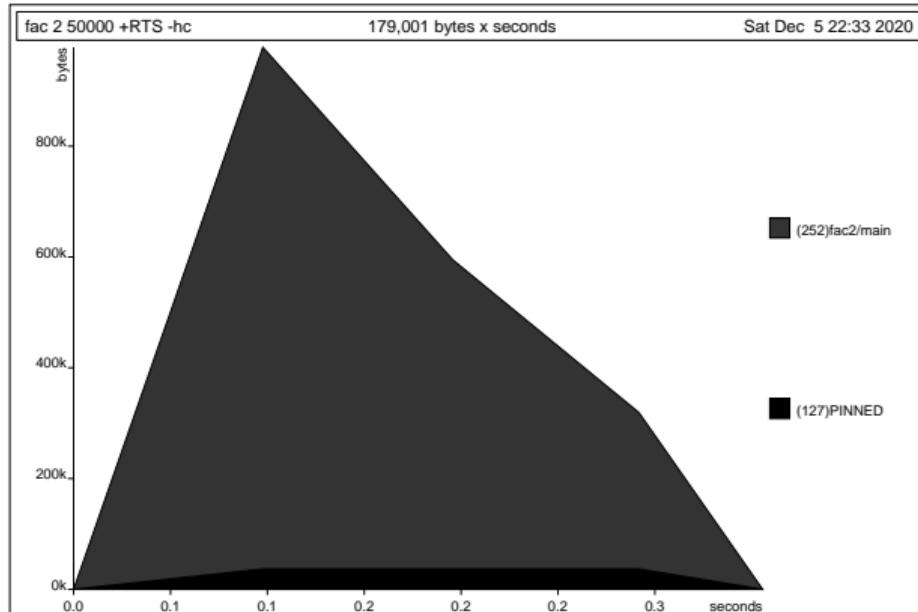
Speicherprofil: fac1 50000, nicht optimiert



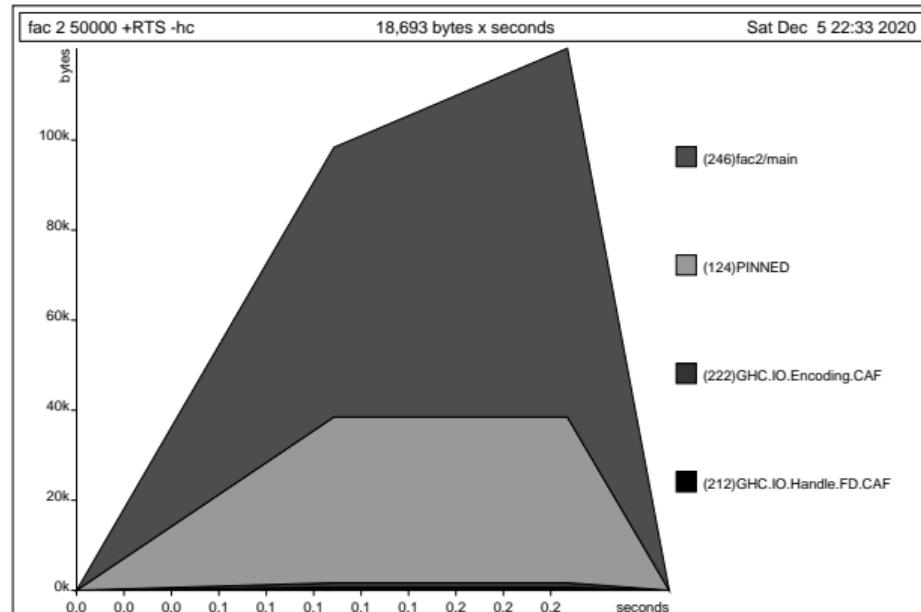
Speicherprofil: fac1 50000, optimiert



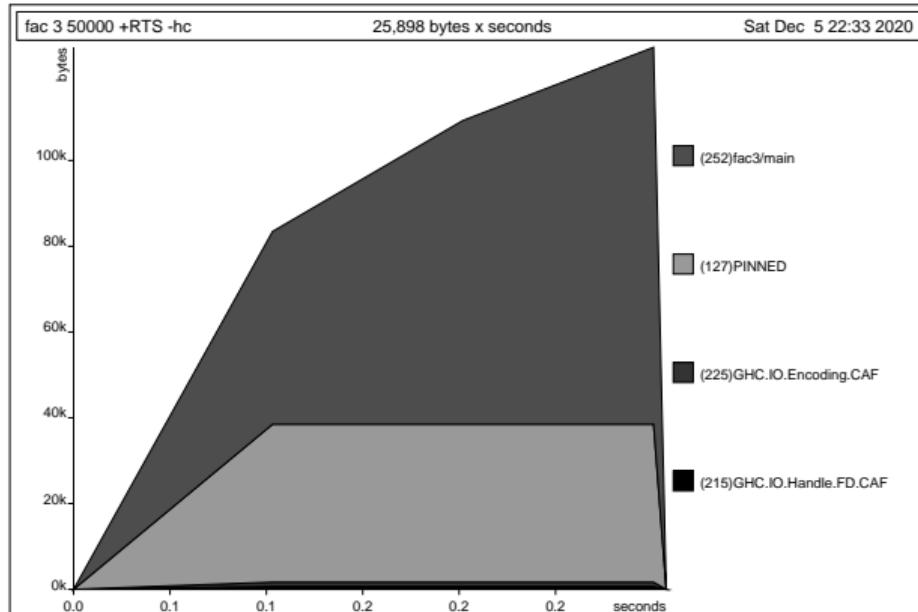
Speicherprofil: fac2 50000, nicht optimiert



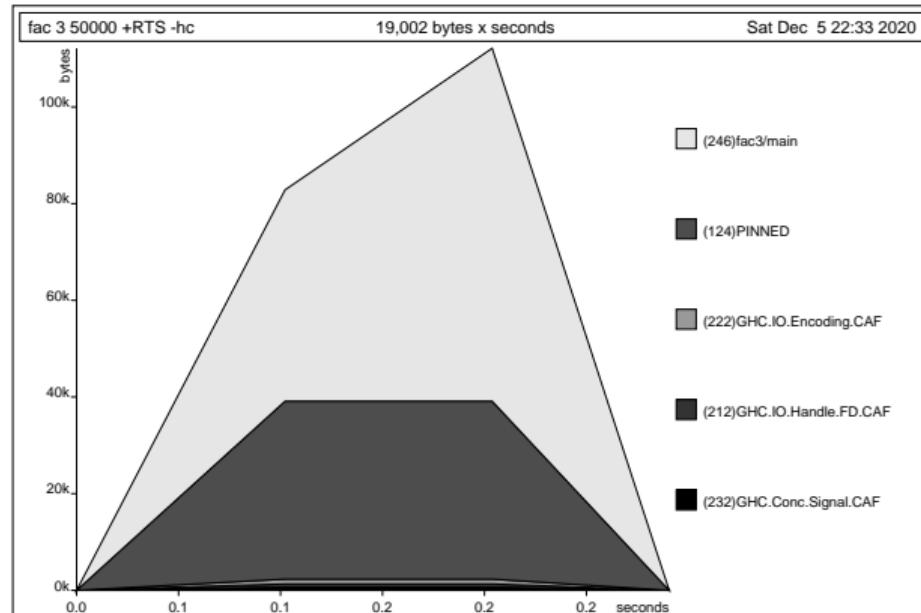
Speicherprofil: fac2 50000, optimiert



Speicherprofil: fac3 50000, nicht optimiert



Speicherprofil: fac3 50000, optimiert



Fakultät als Funktion höherer Ordnung

- ▶ Nicht end-rekursiv mit `foldr`:

```
fac_foldr :: Integer → Integer
fac_foldr i = foldr (*) 1 [1.. i]
```

- ▶ End-rekursiv mit `foldl`:

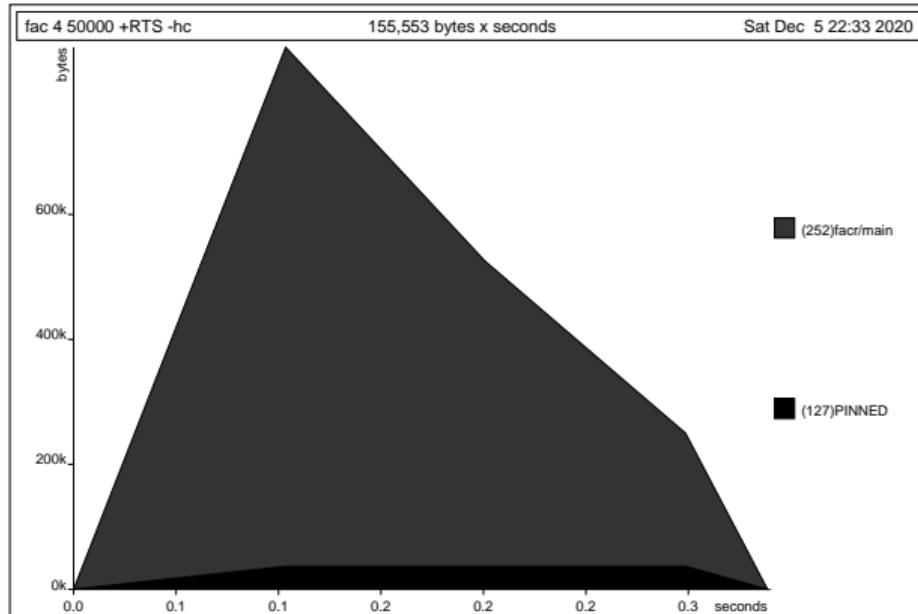
```
fac_foldl :: Integer → Integer
fac_foldl i = foldl (*) 1 [1.. i]
```

- ▶ End-rekursiv und strikt mit `foldl'`:

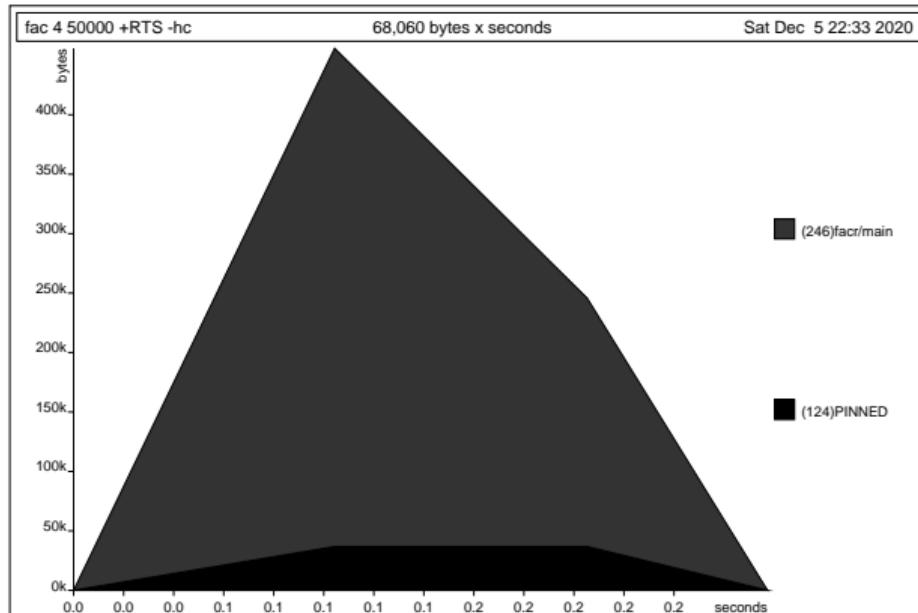
```
fac_foldl' :: Integer → Integer
fac_foldl' i = foldl' (*) 1 [1.. i]
```

- ▶ **Exakt** die gleichen Ergebnisse!

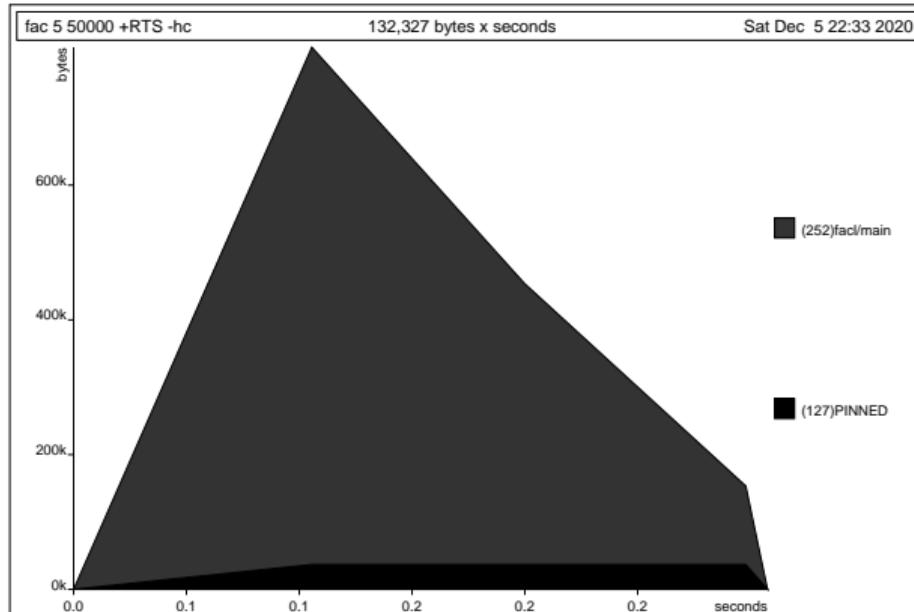
Speicherprofil: foldr 50000, nicht optimiert



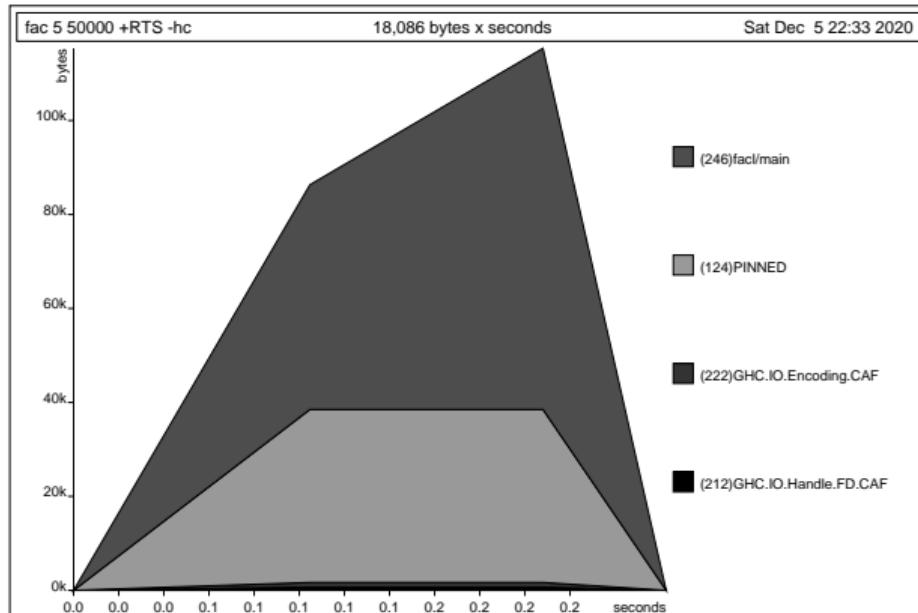
Speicherprofil: foldr 50000, optimiert



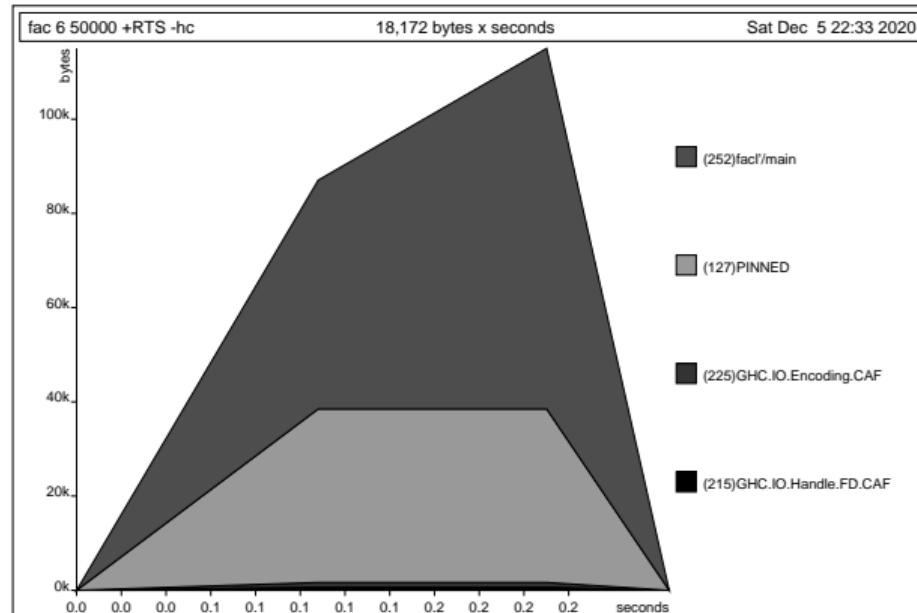
Speicherprofil: foldl 50000, nicht optimiert



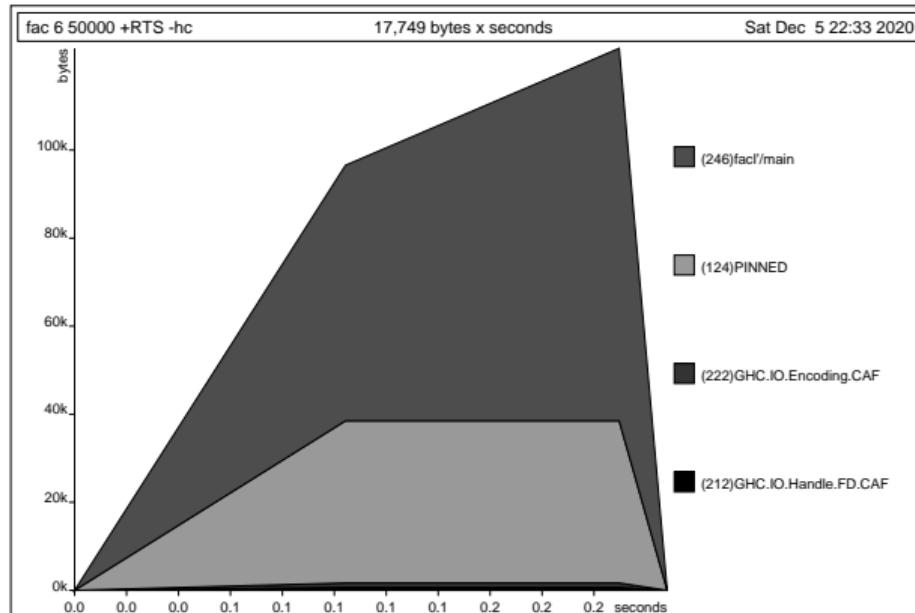
Speicherprofil: foldl 50000, optimiert



Speicherprofil: foldl' 50000, nicht optimiert



Speicherprofil: foldl' 50000, optimiert



Fazit Speicherprofile

- ▶ Endrekursion **nur** bei **strikten Funktionen** schneller
- ▶ Optimierung des *ghc*
 - ▶ Meist **ausreichend** für **Striktheitsanalyse**
 - ▶ Aber **nicht** für Endrekursion
- ▶ Deshalb:
 - ▶ **Manuelle** Überführung in Endrekursion **sinnvoll**
 - ▶ **Compiler-Optimierung** für Striktheit nutzen

Zusammenfassung

- ▶ Rekursive Datentypen können **zyklische Datenstrukturen** modellieren
- ▶ Das Labyrinth — Sonderfall eines **variadischen Baums**
- ▶ Unendliche Listen — nützlich wenn Länge der Liste nicht im voraus bekannt
- ▶ Effizienzerwägungen:
- ▶ Überführung in Endrekursion sinnvoll, Striktheit durch Compiler

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 7 vom 14.12.2020: Funktionen Höherer Ordnung II — Jenseits der Liste

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Fahrplan

► Teil I: Funktionale Programmierung im Kleinen

- Einführung
- Funktionen
- Algebraische Datentypen
- Typvariablen und Polymorphie
- Funktionen höherer Ordnung I
- Rekursive und zyklische Datenstrukturen
- Funktionen höherer Ordnung II

► Teil II: Funktionale Programmierung im Großen

► Teil III: Funktionale Programmierung im richtigen Leben

- ▶ Mehr über `map` und `fold`
- ▶ `map` und `fold` sind nicht nur für Listen
- ▶ Funktionen höherer Ordnung in anderen Programmiersprachen

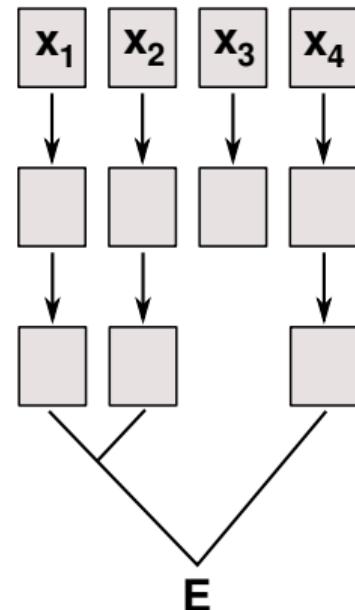
Lernziel

Wir verstehen, warum `map` und `fold` besonders sind, wie sie für andere Datentypen aussehen, und wann wir sie benutzen können.

I. Berechnungsmuster

map und filter als Berechnungsmuster

- ▶ map, filter, fold als Berechnungsmuster:
 - ① Anwenden einer Funktion auf **jedes** Element der Liste
 - ② möglicherweise **Filtern** bestimmter Elemente
 - ③ **Kombination** der Ergebnisse zu Endergebnis **E**
- ▶ Gut parallelisierbar, skalierbar
- ▶ Berechnungsmuster für große Datenmengen
 - ▶ Map/Reduce (Google), Hadoop



Listenkomprehension

- ▶ Besondere Notation: Listenkomprehension

$[f x \mid x \leftarrow as, g x] \equiv \text{map } f (\text{filter } g as)$

- ▶ Beispiel:

- ▶ Remember this?

```
suche :: Artikel → Lager → Maybe Menge
suche a (Lager ps) =
    listToMaybe (map (λ(Posten _ m) → m)
                    (filter (λ(Posten la _) → la == a) ps))
```

- ▶ Sieht so besser aus:

```
suche :: Artikel → Lager → Maybe Menge
suche a (Lager ps) = listToMaybe [ m | Posten la m ← ps, la == a ]
```

Listenkomprehension mit mehreren Generatoren

- Anderes Beispiel: Primzahlzwillinge

```
twin_primes :: [(Integer, Integer)]  
twin_primes = [(x, y) | (x, y) ← zip primes (tail primes), x+2 == y]
```

- Mit mehreren Generatoren werden **alle Kombinationen** generiert:

```
idx :: [String]  
idx = [ a: show i | a← ['a'.. 'z'], i← [0.. 9]]
```

Beispiel I: Quicksort

- ▶ Quicksort per Listenkomprehension:

```
qsort1 :: Ord α⇒ [α]→ [α]
qsort1 [] = []
qsort1 xs@(x:_)= qsort1 [y | y← xs, y< x ] ++
                  [x0 | x0← xs, x0 == x ] ++
                  qsort1 [z | z← xs, z> x ]
```

- ▶ Erstaunlich effizient



- ▶ Einfache Rekursion mit 3-Weg-Split nicht wesentlich effizienter, aber wesentlich länger

Beispiel I: Quicksort

- ▶ Quicksort per Listenkomprehension:

```
qsort1 :: Ord α⇒ [α] → [α]
qsort1 [] = []
qsort1 xs@(x:_)= qsort1 [y | y← xs, y< x ] ++
                  [x0 | x0← xs, x0 == x ] ++
                  qsort1 [z | z← xs, z> x ]
```

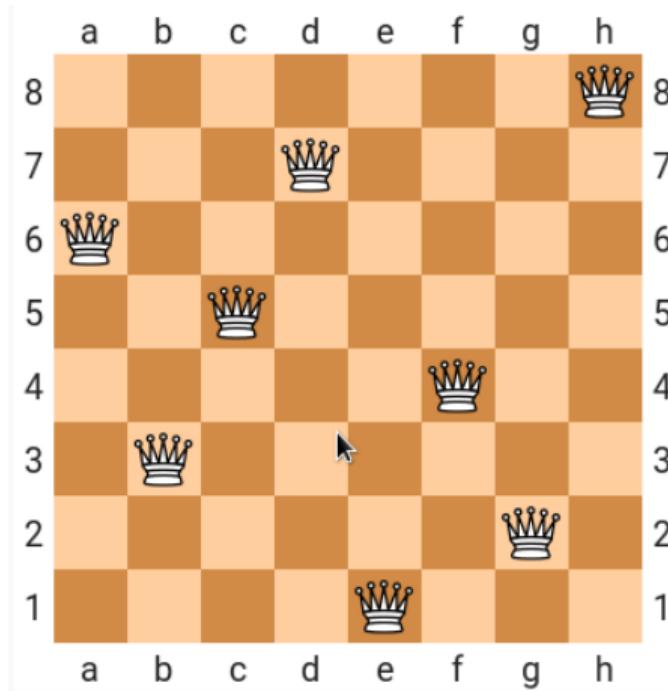
- ▶ Erstaunlich effizient



- ▶ Einfache Rekursion mit 3-Weg-Split nicht wesentlich effizienter, aber wesentlich länger
- ▶ Grund: Sortierte Liste wird nicht im ganzen aufgebaut

Beispiel II: 8-Damen-Problem

- Problem: Plaizeire 8 Damen sicher auf einem Schachbrett



Source: wikipedia

Beispiel II: n-Damen-Problem

- Position der Königinnen:

```
type Pos = (Int, Int)
type Board = [Pos]
```

- Rekursiv: Lösung für $n - 1$ Königinnen, n -te sicher dazu positionieren

```
queens :: Int → [Board]
queens n = qu n where
  qu :: Int → [Board]
  qu i | i == 0 = [[]] — Nicht []
        | otherwise = [ p++ [(i, j)] | p ← qu (i-1), j ← [1.. n],
                                         safe p (i, j)]
```

- Invariante: n -te Königin in n -ter Spalte

Beispiel II: n-Damen-Problem

- Wann ist eine Königin sicher?

```
safe :: Board → Pos → Bool
safe others nu = and [ not (threatens other nu) | other ← others ]
```

- Bedrohung: gleiche Zeile oder Diagonale

```
threatens :: Pos → Pos → Bool
threatens (i, j) (m, n) = (j == n) || (i+j == m+n) || (i-j == m-n)
```

- Diagonalen charakterisiert durch $y = a + x$ bzw. $y = a - x$ für konstantes a
- Gleiche Spalte ($i = m$) durch Konstruktion ausgeschlossen



Was zum Nachdenken

```
queens :: Int → [Board]
queens n = qu n where
  qu :: Int → [Board]
  qu i | i == 0    = [[]] — Nicht []
        | otherwise = [ p++ [(i, j)] | p ← qu (i-1), j ← [1.. n],
                                         safe p (i, j)]
```

Übung 7.1: Warum?

Wieso ist dort [[]] so wichtig? Was passiert, wenn wir [] zurückgeben?

Was zum Nachdenken

```
queens :: Int → [Board]
queens n = qu n where
  qu :: Int → [Board]
  qu i | i == 0    = [[]] — Nicht []
        | otherwise = [ p++ [(i, j)] | p ← qu (i-1), j ← [1.. n],
                                         safe p (i, j)]
```

Übung 7.1: Warum?

Wieso ist dort `[]` so wichtig? Was passiert, wenn wir `[]` zurückgeben?

Lösung:

- ▶ Mit `[]` gibt es **keine** Lösung, mit `[][]` gibt es **eine, leere** Lösung für $i = 0$.
- ▶ Mit `[]` gäbe es **nie** eine Lösung für **alle** i .

II. Map und Fold: Jenseits der Listen

map als strukturerhaltende Abbildung

map ist die kanonische **strukturerhaltende Abbildung**

- ▶ Für map gelten folgende Aussagen:

$$\text{map id} = \text{id}$$

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g)$$

$$\text{length} \circ \text{map } f = \text{length}$$

- ▶ Was davon ist spezifisch für Listen?
- ▶ Wie können wir das verallgemeinern?

map als strukturerhaltende Abbildung

map ist die kanonische **strukturerhaltende Abbildung**

- ▶ Für map gelten folgende Aussagen:

$$\text{map id} = \text{id}$$

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g)$$

$$\text{length} \circ \text{map } f = \text{length}$$

- ▶ Was davon ist spezifisch für Listen?
- ▶ Wie können wir das verallgemeinern?

→ Typklassen?

map als strukturerhaltende Abbildung

map ist die kanonische **strukturerhaltende Abbildung**

- ▶ Für map gelten folgende Aussagen:

$$\text{map id} = \text{id}$$

$$\text{map } f \circ \text{map } g = \text{map } (f \circ g)$$

$$\text{length} \circ \text{map } f = \text{length}$$

- ▶ Was davon ist spezifisch für Listen?
- ▶ Wie können wir das verallgemeinern?

→ Konstruktorklassen!

Funktoren

- ▶ **Konstruktorklassen** sind Typklassen für Typkonstruktoren.
- ▶ Die Konstruktorklasse **Functor** für alle Typen mit einer stukturierhaltenden Abbildung:

```
class Functor f where
  fmap :: (α → β) → f α → f β
```

- ▶ Es sollte gelten (kann nicht geprüft werden):

$$\text{fmap id} = \text{id}$$

$$\text{fmap f} \circ \text{fmap g} = \text{fmap (f} \circ \text{g)}$$

- ▶ Infix-Synomym `<$>` für `fmap`

Instanzen von Functor

- ▶ Listen sind eine Instanz von Functor, aber es gibt `map` und `fmap`

Instanzen von Functor

- ▶ Listen sind eine Instanz von Functor, aber es gibt `map` und `fmap`
- ▶ Maybe ist eine Instanz von Functor:

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

- ▶ Propagiert `Nothing` — oft sehr nützlich

Instanzen von Functor

- ▶ Listen sind eine Instanz von Functor, aber es gibt `map` und `fmap`
- ▶ Maybe ist eine Instanz von Functor:

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

- ▶ Propagiert `Nothing` — oft sehr nützlich
- ▶ Tupel sind Instanzen von Functor im **zweiten** Argument, bspw:

```
instance Functor (a, ) where
  fmap f (a, b) = (a, f b)
```

foldr ist kanonisch

foldr ist die **kanonische strukturell rekursive** Funktion.

- ▶ Alle strukturell rekursiven Funktionen sind als Instanz von `foldr` darstellbar
- ▶ Insbesondere auch `map` und `filter`:

```
map f = foldr ((:). f) []
```

```
filter p = foldr (λa as → if p a then a:as else as) []
```

- ▶ Jeder algebraischer Datentyp hat ein `foldr`
- ▶ Nicht als Konstrukturklasse darstellbar (wie `Functor` und `fmap`)
- ▶ Anmerkung: Typklasse `Foldable` schränkt Signatur von `foldr` ein

fold für andere Datentypen

fold ist universell

Jeder algebraische Datentyp T hat genau ein `foldr`.

► Kanonische Signatur für T :

- Pro Konstruktor C ein Funktionsargument f_C
- Freie Typvariable β für T

► Kanonische Definition:

- Pro Konstruktor C eine Gleichung
- Gleichung wendet f_C auf Argumente an (und `fold` rekursiv auf Argumente vom Typ T)

fold für andere Datentypen

- ▶ Beispiel:

```
data IL = Cons Int IL | Err String | Mt
```

- ▶ Das Fold dazu:

fold für andere Datentypen

- Beispiel:

```
data IL = Cons Int IL | Err String | Mt
```

- Das Fold dazu:

```
foldIL :: (Int → β → β) → (String → β → β) → IL → β
foldIL f e a (Cons i il) = f i (foldIL f e a il)
foldIL f e a (Err str)   = e str
foldIL f e a Mt         = a
```

- Was ist das?

- Eine Art Listen von `Int` mit Fehlern („Ausnahmen“)
- Das zweite Argument von `foldIL` fängt aufgetretene Ausnahmen

fold für bekannte Datentypen

- Bool:

```
data Bool = False | True
```

```
foldBool ::  $\beta \rightarrow \beta \rightarrow \text{Bool} \rightarrow \beta$ 
```

```
foldBool a1 a2 False = a1
```

```
foldBool a1 a2 True = a2
```

fold für bekannte Datentypen

- Bool: Fallunterscheidung:

```
data Bool = False | True
```

```
foldBool ::  $\beta \rightarrow \beta \rightarrow \text{Bool} \rightarrow \beta$ 
```

```
foldBool a1 a2 False = a1
```

```
foldBool a1 a2 True = a2
```

- Maybe α :

```
data Maybe  $\alpha$  = Nothing | Just  $\alpha$ 
```

```
foldMaybe ::  $\beta \rightarrow (\alpha \rightarrow \beta) \rightarrow \text{Maybe } \alpha \rightarrow \beta$ 
```

```
foldMaybe b f Nothing = b
```

```
foldMaybe b f (Just a) = f a
```

fold für bekannte Datentypen

- Bool: Fallunterscheidung:

```
data Bool = False | True
```

```
foldBool ::  $\beta \rightarrow \beta \rightarrow \text{Bool} \rightarrow \beta$ 
```

```
foldBool a1 a2 False = a1
```

```
foldBool a1 a2 True = a2
```

- Maybe α : Auswertung

```
data Maybe  $\alpha$  = Nothing | Just  $\alpha$ 
```

```
foldMaybe ::  $\beta \rightarrow (\alpha \rightarrow \beta) \rightarrow \text{Maybe } \alpha \rightarrow \beta$ 
```

```
foldMaybe b f Nothing = b
```

```
foldMaybe b f (Just a) = f a
```

- Als `maybe` vordefiniert

fold für bekannte Datentypen

- Tupel:

```
data (α, β) = (α, β)
```

```
foldPair :: (α → β → γ) → (α, β) → γ
foldPair f (a, b) = f a b
```

fold für bekannte Datentypen

- ▶ Tupel: die `uncurry`-Funktion

```
data (α, β) = (α, β)
```

```
foldPair :: (α → β → γ) → (α, β) → γ
foldPair f (a, b) = f a b
```

- ▶ Dazu gehört die Funktion `curry` (beide vordefiniert):

```
curry :: ((α, β) → γ) → α → β → γ
curry f a b = f (a, b)
```

- ▶ Die beiden sind **invers**:

$$\text{uncurry} \circ \text{curry} = \text{id} \quad \text{curry} \circ \text{uncurry} = \text{id}$$

fold für bekannte Datentypen

- Natürliche Zahlen:

```
data Nat = Zero | Succ Nat

foldNat ::  $\beta \rightarrow (\beta \rightarrow \beta) \rightarrow \text{Nat} \rightarrow \beta$ 
foldNat e f Zero = e
foldNat e f (Succ n) = f (foldNat e f n)
```

fold für bekannte Datentypen

- Natürliche Zahlen: Iterator

```
data Nat = Zero | Succ Nat

foldNat ::  $\beta \rightarrow (\beta \rightarrow \beta) \rightarrow \text{Nat} \rightarrow \beta$ 
foldNat e f Zero = e
foldNat e f (Succ n) = f (foldNat e f n)
```

- Wendet Funktion f n -mal auf Startwert e an:

$$\text{foldNat } e \ f \ n = f^n(e)$$

- Konversion nach `Int`:

```
natToInt :: Nat  $\rightarrow$  Int
natToInt = foldNat 0 (1+)
```

Kurze Denkpause

Übung 7.2: Merkwürdige Zahlen

Wenn wir die natürlichen Zahlen mit einem Typ-Parameter versehen:

```
data FNat α = FZero | FSucc α (FNat α)
```

Was ist die kanonische Funktion `foldFNat`, und welcher Datentyp ist das?

Kurze Denkpause

Übung 7.2: Merkwürdige Zahlen

Wenn wir die natürlichen Zahlen mit einem Typ-Parameter versehen:

```
data FNat α = FZero | FSucc α (FNat α)
```

Was ist die kanonische Funktion `foldFNat`, und welcher Datentyp ist das?

Lösung:

```
foldFNat :: β → (α → β → β) → FNat α → β
foldFNat e f FZero = e
foldFNat e f (FSucc a n) = f a (foldFNat e f n)
```

Das sind natürlich Listen, mit `foldr`:

```
foldr :: (α → β → β) → β → [α] → β
```

fold für binäre Bäume

- Binäre Bäume:

```
data Tree α = Mt | Node α (Tree α) (Tree α)
```

- Label **nur** in den Knoten

fold für binäre Bäume

- Binäre Bäume:

```
data Tree α = Mt | Node α (Tree α) (Tree α)
```

- Label **nur** in den Knoten
- Instanz von **fold**:

```
foldT :: β → (α → β → β → β) → Tree α → β
foldT e f Mt = e
foldT e f (Node a l r) = f a (foldT e f l) (foldT e f r)
```

- Instanz von **Functor**, kein (offensichtliches) Filter

```
instance Functor Tree where
  fmap f Mt = Mt
  fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)
```

Funktionen mit foldT

- ▶ Höhe des Baumes berechnen:

```
height :: Tree α → Int
height = foldT 0 (λ_ l r → 1 + max l r)
```

- ▶ Inorder-Traversierung der Knoten:

```
inorder :: Tree α → [α]
inorder = foldT [] (λa l r → l ++ [a] ++ r)
```

- ▶ Enthält der Baum dieses Element?

```
isElem :: Eq α ⇒ α → Tree α → Bool
isElem a = foldT False (λb l r → a = b || l || r)
```

- ▶ Nich-Striktheit von `||` begrenzt Traversierung

Kanonische Eigenschaften von foldT und fmap

- Auch hier gilt:

$$\text{foldT } \text{Mt Node} = \text{id}$$

$$\text{fmap id} = \text{id}$$

$$\text{fmap f} \circ \text{fmap g} = \text{fmap (f} \circ \text{g)}$$

- Gilt für **alle** Datentypen. Insbesondere gilt:

$$\text{fold } C_1 \ C_2 \dots C_n = \text{id}$$

Falten mit den Konstruktoren ergibt die Identität.

Variadische Bäume

- Das Labyrinth ist ein variadischer Baum:

```
data VTree α = NT α [VTree α]
```

- Auch hierfür `fold` und `map`:

Variadische Bäume

- Das Labyrinth ist ein variadischer Baum:

```
data VTree α = NT α [VTree α]
```

- Auch hierfür `fold` und `map`:

```
foldT :: (α → [β] → β) → VTree α → β
foldT f (NT a ns) = f a (map (foldT f) ns)
```

```
instance Functor VTree where
  fmap f (NT a ns) = NT (f a) (map (fmap f) ns)
```

Suche im Labyrinth

- ▶ Tiefensuche via `foldT`

```
dfs1 :: VTree α → [Path α]
dfs1 = foldT add where
  add a [] = [[a]]
  add a ps = [ a:p | p ← concat ps]
```

- ▶ Problem:



- ▶ `foldT` terminiert **nicht** für **zyklische** Strukturen

Suche im Labyrinth

- ▶ Tiefensuche via `foldT`

```
dfs2 :: Eq α ⇒ VTree α → [Path α]
dfs2 = foldT add where
  add a [] = [[a]]
  add a ps = [a:p | p ← concat ps, not (a `elem` p) ]
```

- ▶ Problem:



- ▶ `foldT` terminiert **nicht** für **zyklische** Strukturen
- ▶ Auch nicht, wenn `add` prüft ob `a` schon enthalten ist
- ▶ Pfade werden vom Ende konstruiert

Grenzen von foldr

- ▶ `foldr` traversiert die gesamte Struktur, konstruiert Ergebnis von nicht-rekursiven Konstruktoren her
- ▶ Nicht-Striktheit erlaubt zyklische Strukturen, wenn **lokal** Abbruch der Rekursion möglich
 - ▶ Beispiel: `all = foldr (&&) True`
 - ▶ Gegenbeispiel: Tiefensuche in zyklischen Strukturen, Breitensuche
- ▶ `foldl` ist **nicht** generalisierbar
 - ▶ Warum?

Grenzen von foldr

- ▶ `foldr` traversiert die gesamte Struktur, konstruiert Ergebnis von nicht-rekursiven Konstruktoren her
- ▶ Nicht-Striktheit erlaubt zyklische Strukturen, wenn **lokal** Abbruch der Rekursion möglich
 - ▶ Beispiel: `all = foldr (&&) True`
 - ▶ Gegenbeispiel: Tiefensuche in zyklischen Strukturen, Breitensuche
- ▶ `foldl` ist **nicht** generalisierbar
 - ▶ Warum? Nur für **linear rekursive** Typen

Andere Arten der Rekursion

- ▶ Andere rekursive Struktur über Listen
 - ▶ Quicksort: baumartige Rekursion
- ▶ Rekursion nicht (nur) über Listenstruktur:
 - ▶ `take`: Begrenzung der Rekursion

```
take :: Int → [α] → [α]
take n _           | n ≤ 0 = []
take _ []          = []
take n (x:xs)      = x : take (n-1) xs
```

- ▶ Version mit `fold` divergiert für nicht-endliche Listen

Kurzes Gehirnjogging

Übung 7.3:

Wie sieht die Version von `take` mit `fold` aus (`foldl` oder `foldr`)?

Kurzes Gehirnjogging

Übung 7.3:

Wie sieht die Version von `take` mit `fold` aus (`foldl` oder `foldr`)?

Lösung:

► Mit `foldl`:

```
take :: Int → [α] → [α]
take i = foldl (λp a → if length p < i then (p ++ [a]) else p) []
```

► Mit `foldr` und `zip`:

```
takez' i = map snd ∘ zip [1.. i]
```

Geschummelt weil `zip` nicht mit `fold` implementiert werden kann

III. Anhang: Datentypen in anderen Programmiersprachen

Andere Programmiersprachen

- ▶ C — systemnah, schnell
- ▶ Java — objektorientiert, Systemsprache
- ▶ Python — Skriptsprache

Datentypen in C

- ▶ **C**: Produkte, Aufzählungen, keine rekursiven Typen
- ▶ Rekursion **nur** durch **Zeiger**
- ▶ Konstruktoren **nutzerimplementiert**
- ▶ Manuelle Speicherverwaltung (**malloc/free**)

Datentypen in Java

- ▶ Nachbildung durch Klassen
- ▶ Datentyp ist abstrakte Klasse, Konstruktoren sind Unterklassen dieser Klasse
- ▶ Volle Speicherverwaltung (mit garbage collection)

Datentypen in Python

- ▶ Listen und **Tupel** fest eingebaut
- ▶ Diverse Funktionen auf Listen
 - ▶ Methoden (**stateful**) vs. Funktionen
 - ▶ Bsp. `sort` vs. `sorted`
- ▶ Definition eigener Typen über Klassen
- ▶ Volle Speicherverwaltung (mit garbage collection)

Polymorphie in C

- ▶ Polymorphie in C: `void *`
- ▶ Pointer-to-void ist kompatibel mit allen anderen Pointer-Typen.
- ▶ Manueller Typ-Cast nötig
 - ▶ Vergl. `Object` in Java
- ▶ Extrem Fehleranfällig

Polymorphie in Java

- ▶ Polymorphie in **Java**: Methode auf alle Subklassen anwendbar
 - ▶ Manuelle **Typkonversion** nötig, fehleranfällig
- ▶ Neu ab Java 1.5: **Generics**
 - ▶ Damit **parametrische Polymorphie** möglich
 - ▶ **Nachteil:** Benutzung umständlich, weil keine Typherleitung
 - ▶ **Vorteil:** Typkorrektheit sichergestellt:
 - ▶ Allerdings: Typ-Parameter nur für Klassen.

Ad-Hoc Polymorphie in Java

- ▶ `interface` und `abstract class`
- ▶ Flexible in Java: beliebig viele Parameter etc.
- ▶ Eingeschränkt durch Vererbungshierarchie
- ▶ Ähnliche Standardklassen
 - ▶ `toString`
 - ▶ `equals` und `==`, keine abgeleitete strukturelle Gleichheit

Polymorphie in Python

- ▶ In Python werden Typen zur **Laufzeit** geprüft (**dynamic typing**)
- ▶ **duck typing**: strukturell gleiche Typen sind gleich
- ▶ Polymorphie durch Klassen
- ▶ Statt Interfaces kennt Python **Mixins**
- ▶ Abstrakte Klassen ohne Oberklasse

Funktionen höherer Ordnung in C

- Implizit vorhanden: Funktionen = Zeiger auf Funktionen

```
extern list map1(void *(*f)(void *x), list l);
```

```
extern list filter(int(*f)(void *x), list l);
```

- Keine direkte Syntax (e.g. namenlose Funktionen)
- Typsystem zu schwach (keine Polymorphie)
- Benutzung: `qsort` (C-Standard 7.20.5.2)

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Funktionen höherer Ordnung in Java

- **Java**: keine direkte Syntax für Funktionen höherer Ordnung
- Folgendes ist **nicht** möglich:

```
interface Collection {  
    Object fold(Object f(Object a, Collection c), Object a); }
```

- Aber folgendes:

```
interface Foldable { Object f (Object a); }  
  
interface Collection { Object fold(Foldable f, Object a); }
```

- Vergleiche **Iterator** aus Collections Framework (Java SE 6):

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next(); }
```

- Seit Java SE 8 (März 2014): Anonyme Funktionen (Lambda-Ausdrücke)

Funktionen höherer Ordnung in Python

- ▶ Python kennt map, filter, fold:

```
letters = map(chr, range(97, 123))
```

- ▶ Map auf Iteratoren definiert, nicht auf Listen
- ▶ Python kennt Listenkomprehension:

```
idx = [ x+ str(i) for x in letters for i in range(10) ]
```

- ▶ Python kennt Lambda-Ausdrücke:

```
num = map (lambda x: 3*x+1, range (1,10))
```

Zusammenfassung

- ▶ Einge Funktionen höherer Ordnung sind speziell:
 - ▶ `map` ist die strukturerhaltende Funktion
 - ▶ `fold` ist die strukturelle Rekursion über dem Typen
- ▶ Jeder Datentyp hat `map` und `fold`
- ▶ Konstruktorklassen sind Klassen für Typkonstruktoren
 - ▶ Beispiel `Functor`
- ▶ Listenkomprehension ist ein nützlicher, leichtgewichtiger syntaktischer Zucker für `map` und `filter`

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 8 vom 21.12.2020: Abstrakte Datentypen

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Organisatorisches

- ▶ Abgabe des 7. Übungsblattes in Gruppen zu **drei** Studenten.
- ▶ Bitte **jetzt** eine Gruppe suchen!
- ▶ Klausurtermine:
 - ▶ Klausur: 03.02.2020, 10:00/11:30/15:00
 - ▶ Wiederholungstermin: 21.04.2020, 10:00/11:30/15:00

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ **Teil II: Funktionale Programmierung im Großen**
- ▶ Abstrakte Datentypen
- ▶ Signaturen und Eigenschaften
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

► **Abstrakte Datentypen**

- Allgemeine Einführung
- Realisierung in Haskell
- Beispiele

I. Modularisierung und Abstrakte Datentypen

Warum Modularisierung?

- ▶ Übersichtlichkeit der Module **Lesbarkeit**
- ▶ Getrennte Übersetzung **technische** Handhabbarkeit
- ▶ Verkapselung **konzeptionelle** Handhabbarkeit

Abstrakte Datentypen

Definition (Abstrakter Datentyp)

Ein **abstrakter Datentyp** (ADT) besteht aus einem (oder mehreren) **Typen** und **Operationen** darauf, mit folgenden Eigenschaften:

- 1 Werte des Typen können nur über die Operationen **erzeugt** werden
- 2 Eigenschaften von Werten des Typen werden nur über die Operationen **beobachtet**
- 3 Einhaltung von **Invarianten** über dem Typ kann garantiert werden

Implementation von ADTs in einer Programmiersprache:

- ▶ benötigt Möglichkeit der **Kapselung** (Einschränkung der Sichtbarkeit)
- ▶ bspw. durch **Module** oder **Objekte**

ADTs vs. algebraische Datentypen

- ▶ Algebraische Datentypen
 - ▶ **Frei erzeugt** durch **Konstruktoren**
 - ▶ Keine Einschränkungen
 - ▶ Insbesondere keine Gleichheiten der Konstruktoren ($[] \neq x:xs$, $x:ls \neq y:ls$ etc.)
- ▶ ADTs:
 - ▶ Keine ausgezeichneten Konstruktoren
 - ▶ Einschränkungen und Invarianten möglich
 - ▶ Gleichheiten möglich

ADTs vs. Objekte

- ▶ ADTs (z.B. Haskell): **Typ** plus **Operationen**
- ▶ Objekte (z.B. Java): **Interface**, **Methoden**.
- ▶ **Gemeinsamkeiten:**
 - ▶ Verkapselung (information hiding) der Implementation
- ▶ **Unterschiede:**
 - ▶ Objekte haben **internen Zustand**, ADTs sind **referentiell transparent**;
 - ▶ Objekte haben **Konstruktoren**, ADTs nicht
 - ▶ Vererbungsstruktur auf Objekten (**Verfeinerung** für ADTs)
 - ▶ Java: interface eigenes Sprachkonstrukt
 - ▶ Java: packages für Sichtbarkeit

ADTs in Haskell: Module

- ▶ Einschränkung der Sichtbarkeit durch **Verkapselung**
- ▶ **Modul**: Kleinste verkapselbare Einheit
- ▶ Ein **Modul** umfaßt:
 - ▶ **Definitionen** von Typen, Funktionen, Klassen
 - ▶ **Deklaration** der nach außen **sichtbaren** Definitionen
- ▶ Gleichzeitig: Modul $\hat{=}$ Übersetzungseinheit (getrennte Übersetzung)

Module: Syntax

- ▶ Syntax:

```
module Name(Bezeichner) where Rumpf
```

- ▶ Bezeichner können leer sein (dann wird alles exportiert)
- ▶ Bezeichner sind:
 - ▶ **Typen**: T, T(c₁, ..., c_n), T(..)
 - ▶ **Klassen**: C, C(f₁, ..., f_n), C(..)
 - ▶ Andere Bezeichner: **Werte, Felder, Klassenmethoden**
 - ▶ Importierte **Module**: module M
- ▶ Typsynonyme und Klasseninstanzen bleiben sichtbar
- ▶ Module können **rekursiv** sein (*don't try at home*)

Refakturierung im Einkaufsparadies

```

module Shoppi where

import Data.Maybe
import Data.List

-- Modellierung der Artikel.

data Apfelsorte = Boskoop | CoxOrange | GrannySmith
    deriving (Eq, Show)

apreis :: Apfelsorte → Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50

data Käsesorte = Gouda | Appenzeller
    deriving (Eq, Show)

kpreis :: Käsesorte → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

data Bio = Bio | Konv
    deriving (Eq, Show)

data Artikel =
    Apfelsorte | Eier
    | Käse Käsesorte | Schinken
    | Salami | Milch Bio
    deriving (Eq, Show)

data Menge = Stück Int | Gramm Int | Liter Double
    deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel → Menge → Preis
preis (Apfel a) (Stück n) = Just (n * apreis a)
preis (Eier (Stück n)) = Just (n * 20)
preis (Käse Käse (Gramm g)) = Just (round (fromIntegral g * 1000 * kpreis Käse))
preis Schinken (Gramm g) = Just (div (g * 100) 100)
preis Salami (Gramm g) = Just (div (g * 159) 100)
preis (Milch Bio (Liter l)) = Just (l * 69)
preis (Milch Bio (Liter l)) = Just (round (l * case bio of Bio → 119; Konv → 69))
preis _ = Nothing

-- Addition von Mengen
addiere :: Menge → Menge → Menge
addiere (Stück n) (Stück m) = Stück (n + m)
addiere (Gramm n) (Gramm m) = Gramm (n + m)
addiere (Liter n) (Liter m) = Liter (n + m)
addiere m n = error ("addiere: " ++ show m ++ " und " ++ show n)

-- Posten:
data Posten = Posten Artikel Menge
    deriving (Eq, Show)

cent :: Posten → Int
cent (Posten a m) = fromMaybe 0 (preis a m) -- gibt keinen Laufzeitfehler

-- Lagerhaltung:
data Lager = Lager [Posten]
    deriving (Eq, Show)

leeresLager :: Lager
leeresLager = Lager []

suche :: Artikel → Lager → Maybe Menge
suche a (Lager ps) =
    listToMaybe [Posten a m | Posten a m ← ps, a == a]

einlagern :: Artikel → Menge → Lager → Lager
einlagern a m (Lager ps) =
    let hinein a m [] = [Posten a m]
        hinein a m (l : ps) =
            a == a → (Posten a (addiere m m)) : l
            | otherwise = (Posten a m) : hinein a m l
    in case Preis a of
        Nothing → Lager ps
        _ → Lager (hinein a m ps)

data Einkaufswagen = Ewig [Posten]
    deriving (Eq, Show)

leererWagen :: Einkaufswagen
leererWagen = Ewig []

einkauf :: Artikel → Menge → Einkaufswagen → Einkaufswagen
einkauf a m (Ewig ps) =
    | isUst (preis a m) = Ewig (Posten a m : ps)
    | otherwise = Ewig ps

kasse :: Einkaufswagen → Int
kasse (Ewig ps) = sum (map cent ps)

kassenbon = Einkaufswagen → String
kassenbon ew@Ewig ps) =
    "BdB s_AudeGrocery.Shoppi(n,n" ++
    "Artikel" "Menge" "Preis)n" ++
    "concatMap artikel ps" ++
    "Summe" ++ formatR 31 (showEuro (kasse ew))

artikel :: Posten → String
artikel (Posten a m) =
    formatR 20 (show a) ++
    formatR 7 (menger) ++
    formatR 10 (showEuro (cent p)) ++ "\n"

merge :: Menge → String
merge (Stück n) = show n ++ ".St"
merge (Gramm g) = show g ++ ".g."
merge (Liter l) = show l ++ ".l."

formatL :: Int → String → String
formatL n str = take n (str ++ replicate n ' ')

formatR :: Int → String → String
formatR n str =
    take n (replicate (n - length str) ' ' ++ str)

showEuro :: Int → String
showEuro i =
    show (div i 100) ++ "." ++
    show (mod (div i 10) 10) ++
    show (mod i 10) ++ ".EU"

inventur :: Lager → Int
inventur (Lager l) = sum (map cent l)

```

Refakturierung im Einkaufsparadies

```

module Shoppe where
  import Data.Maybe

  -- Modellierung der Artikel.

  data Apfelsorte = Boskoop | CoxOrange | GrannySmith
    deriving (Eq, Show)

  apreis :: Apfelsorte → Int
  apreis Boskoop = 55
  apreis CoxOrange = 60
  apreis GrannySmith = 50

  data Käsesorte = Gouda | Appenzeller
    deriving (Eq, Show)

  kpreis :: Käsesorte → Double
  kpreis Gouda = 1450
  kpreis Appenzeller = 2270

  data Bio = Bio | Konv
    deriving (Eq, Show)

  data Artikel =
    Apf Apfelsorte | Eier
    | Käse Käsesorte | Schinken
    | Salami | Milch Bio
    deriving (Eq, Show)

  data Menge = Stück Int | Gamm Int | Liter Double
    deriving (Eq, Show)

  type Preis = Maybe Int

  preis :: Artikel → Menge → Preis
  preis (Apf a) (Stück n) = Just (n * apreis a)
  preis Eier (Stück n) = Just (n * 20)
  preis (Käse k) (Gamm g) = Just (round (fromIntegral g * 1000 * kpreis k))
  preis Schinken (Gamm g) = Just (div (g * 199) 100)
  preis Salami (Gamm g) = Just (div (g * 159) 100)
  preis (Milch Bio) (Liter l) = Just (l * 69)
  preis _ = Nothing

  -- Addition von Mengen
  addiere :: Menge → Menge → Menge
  addiere (Stück 1) (Stück 1) = Stück (1 + 1)
  addiere (Gamm g) (Gamm h) = Gamm (g + h)
  addiere (Liter l) (Liter m) = Liter (l + m)
  addiere m n = error ("addiere: " ++ show m ++ " und " ++ show n)

  -- Posten:
  data Posten = Posten Artikel Menge
    deriving (Eq, Show)

  cent :: Posten → Int
  cent (Posten a m) = fromMaybe 0 (preis a m) -- gibt keinen Laufzeitfehler

  -- Lagerhaltung:
  data Lager = Lager [Posten]
    deriving (Eq, Show)

  leerLager :: Lager
  leerLager = Lager []

```

```

suche :: Artikel → Lager → Maybe Posten
suche a (Lager ps) =
  listToMaybe [ m | Posten la m ← ps, la == a ]

einlagern :: Artikel → Menge → Lager → Lager
einlagern a m (Lager ps) =
  let hinein a m l = Posten a m : l
  hinein a m (Posten al ml : l)
  | a == al => (Posten a (addiere ml m) : l)
  | otherwise => (Posten al ml : hinein a m l)
  in case Preis a of
    Nothing => Lager ps
    _ => Lager (hinein a m ps)

data Einkaufswagen = Ewig [Posten]
  deriving (Eq, Show)

leererWagen :: Einkaufswagen
leererWagen = Ewig []

einkauf :: Artikel → Menge → Einkaufswagen → Einkaufswagen
einkauf a m (Ewig ps) =
  | isJust (preis a m) => Ewig (Posten a m : ps)
  | otherwise => Ewig ps

kasse :: Einkaufswagen → Int
kasse (Ewig ps) = sum (map cent ps)

kassieren :: Einkaufswagen → String
kassieren ew@Ewig ps) =
  "Bü's_Auflde_Grocery_Shoppe(n,n" ++
  "Artikel" "Menge" "Preis)n" ++
  "concatMap artikel ps" ++
  "Summe" ++ formatR 31 (showEuro (kasse ew))

artikel :: Posten → String
artikel (Posten a m) =
  formatR 20 (show a) ++
  formatR 7 (show m) ++
  formatR 10 (showEuro (cent a)) ++ "\n"

merge :: Menge → String
merge (Stück n) = show n ++ ".St"
merge (Gamm g) = show g ++ ".g."
merge (Liter l) = show l ++ ".L"

format :: Int → String → String
format n str = take n (str ++ replicate n ' ')

formatR :: Int → String → String
formatR n str =
  take n (replicate (n - length str) ' ' ++ str)

showEuro :: Int → String
showEuro i =
  show (div i 100) ++ "."
  show (mod (div i 10) 10) ++
  show (mod i 10) ++ ".EU"

inventur :: Lager → Int
inventur (Lager l) = sum (map cent l)

```

Refakturierung im Einkaufsparadies

```
module Shopper where
```

```
import Data.Maybe
```

— Modellierung der Artikel.

```
data Apfelsorte = Boskoop | CoxOrange | GrannySmith
    deriving (Eq, Show)
```

```
apreis :: Apfelsorte -> Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50
```

```
data Käsesorten = Gouda | Appenzeller
```

```
    deriving (Eq, Show)
```

```
kpries :: Käsesorten -> Double
kpries Gouda = 1450
kpries Appenzeller = 2270
```

```
data Bio = Bio | Konv
    deriving (Eq, Show)
```

```
data Artikel =
    Apf1 Apfelsorte | Eier
    | Käse Käsesorten | Schinken
    | Salami | Milch Bio
    deriving (Eq, Show)
```

```
data Merge = Stuck Int | Gamm Int | Liter Double
```

```
type Preis = Maybe Int
```

```
preis :: Artikel -> Merge#(Preis)
preis (Apf1 a) = Stuck a; // just (ne a)
preis Eier (Stuck n) = Just (n*20)
preis (Käse k)Gamm g = Just (round(frontIntegral g*3000*k) kpries k)
preis Schinken (Gamm g) = Just (div (g*199) 100)
preis Salami (Gamm g) = Just (div (g*199) 100)
preis Milch Bio (Litter n) = Just (n*60)
preis Bio (Litter n) = Just (round (1*case bio of Bio => 119; Konv => 69))
preis _ = Nothing
```

— Addition von Mengen

```
addiere :: Merge#(Preis) -> Merge#(Preis)
addiere (Stuck n) (Stuck m) = Stuck (n+m)
addiere (Gamm g) (Gamm h) = Gamm (g+h)
addiere (Litter l) (Litter m) = Litter (l+m)
addiere m n = error ("addiere: " + show me " und " + show n)
```

— Posten:

```
data Posten = Posten Artikel Merge
    deriving (Eq, Show)
```

```
cent :: Posten#(Int)
```

```
cent (Posten a m) = fromMaybe 0 (preis a m) — gibt keinen Laufzeitfehler!
```

— Lagerhaltung:

```
data Lager = Lager [Posten]
    deriving (Eq, Show)
```

```
leeresLager :: Lager
```

```
leeresLager = Lager []
```

Artikel

```
suche :: Artikel -> Lager -> Maybe#(Posten)
```

```
suche a (Lager ps) = listToMaybe (map (Posten a m+) ps, a == a)
```

```
einlagern :: Artikel -> Merge#(Lager) -> Lager
```

```
einlagern a m (Lager ps) =  
    let hinein a m1 = (Posten a m1) : ps  
    hinein a m = (Posten a m1 : hinein a m1)  
    a == a -> (Posten a (addiere m m1)) : 1  
    otherwise -> (Posten a m1 : hinein a m1)  
in case Preis a of  
    Nothing -> Lager ps  
    _ -> Lager (hinein a m ps)
```

```
data Einkaufswagen = Ebag [Posten]
    deriving (Eq, Show)
```

```
leererWagen :: Einkaufswagen
```

```
leererWagen = Ebag []
```

```
einkauf :: Artikel -> Merge#(Einkaufswagen) -> Einkaufswagen
```

```
einkauf a m (Ebag ps) =  
    | is unit (preis a m) = Ebag (Posten a m: ps)  
    | otherwise = Ebag ps
```

```
kasse :: Einkaufswagen -> Int
```

```
kasse (Ebag ps) = sum (map cent ps)
```

```
kaessbon = Einkaufswagen -> String
```

```
kaessbon ps@Ebag ps =  
    "Bd's_Aufde_Grocery_Shoppin' n" +  
    "Artikel.....Merge.....Preis n" +  
    "....." +  
    concatMap artikel ps +  
    "Summe" + formatR 31 (showEuro (kasse ps))
```

```
artikel :: Posten -> String
```

```
artikel (Posten a m) =  
    formatR 20 (show a) +  
    formatR 7 (merge) +  
    formatR 10 (showEuro (cent p)) + "n"
```

```
merge :: Merge -> String
```

```
merge (Stuck n) = show n + "St"  
merge (Gamm g) = show g + "g"  
merge (Litter l) = show l + "L"
```

```
formatR :: Int -> String -> String
```

```
formatR n str =  
    take n (replicate (n - length str) ' ' + str)
```

```
showEuro :: Int -> String
```

```
showEuro i =  
    show (div i 100) + "." +  
    show (mod (div i 10) 10) +  
    show (mod i 10) + "EUR"
```

```
inventur :: Lager -> Int
```

```
inventur (Lager l) = sum (map cent l)
```

Refakturierung im Einkaufsparadies

```

module Shopper where
import Data.Maybe

-- Modellierung der Artikel.
data Apfelsorte = Boskoop | CoxOrange | GrannySmith
    deriving (Eq, Show)

apreis :: Apfelsorte -> Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50

data Kassenzorte = Gouda | Appenzeller
    deriving (Eq, Show)

kpreis :: Kassenzorte -> Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

data Bio = Bio | Konv
    deriving (Eq, Show)

data Artikel =
    Apf Apfelsorte | Eier
    | Kass Kassenzorte | Schinken
    | Salat
    | Milch Bio
    deriving (Eq, Show)

data Merge = Stuck Int | Gamm Int | Liter Double
    deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel -> Merge#(Preis)
preis (Apf a) = Stuck a; // just (ne aplais a)
preis Eier (Stuck n) = Just (n*20)
preis (Kass k)Gamm g = Just (round(frontIntegral g*3000*kpreis k))
preis Schinken (Gamm g) = Just (div (g*100) 300)
preis Salat (Gamm g) = Just (div (g*150) 300)
preis Milch (Litter l) = Just (round (l*case Bio of Bio => 110; Konv => 60))
preis _ = Nothing

-- Addition von Mengen
addiere :: Merge -> Merge
addiere (Stuck n) (Stuck l) = Stuck (l+n)
addiere (Gamm g) (Gamm h) = Gamm (g+h)
addiere (Litter l) (Litter m) = Litter (l+m)
addiere m n = error ("addiere: "++show m++" und "++show n)

-- Posten
data Posten = Posten Artikel Merge
    deriving (Eq, Show)

cent :: Posten -> Int
cent (Posten a m) = fromMaybe 0 (preis a m) -- gibt keinen Laufzeitfehler

-- Lagerhaltung:
data Lager = Lager [Posten]
    deriving (Eq, Show)

lernerLager :: Lager
lernerLager = Lager []

```

Artikel

Posten
Lager

```

suche :: Artikel -> Lager -> Maybe Posten
suche a (Lager ps) =
    listToMaybe (filter (\p -> Posten.a == p.a, ps, a == a))

einlagen :: Artikel -> Merge -> Lager -> Lager
einlagen a m (Lager ps) =
    let hinein a m l = (Posten a m): l
    in hinein a m (Posten a (addiere m m): l)
        | a == al = (Posten a (addiere m m): l)
        | otherwise = (Posten a m: hinein a m: l)
    in case Preis of
        Nothing => Lager ps
        _ => Lager (hinein a m ps)

data Einkaufswagen = Eweg [Posten]
    deriving (Eq, Show)

leererWagen :: Einkaufswagen
leererWagen = Eweg []

einkauf :: Artikel -> Merge -> Einkaufswagen -> Einkaufswagen
einkauf a m (Eweg ps) =
    | IsJust (preis a m) = Eweg (Posten a m: ps)
    | otherwise = Eweg ps

kasse :: Einkaufswagen -> Int
kasse (Eweg ps) = sum (map cent ps)

kassenzon = Einkaufswagen -> String
kassenzon m@Eweg ps) =
    "Bö's. Aufde.Grocery.Shoppin'n'"+  

    "Artikel.....Merge.....Preis....."+  

    "-----"+  

    concatMap artikel ps +  

    "Summe"++formatR 31 (showEuro (kasse m))

artikel :: Posten -> String
artikel (Posten a m) =
    formatR 20 (show a) ++
    formatR 7 (show m) ++
    formatR 10 (showEuro (cent p)) ++ "\n"

merge :: Merge -> String
merge (Stuck n) = show n++ ".St"
merge (Gamm g) = show g++ ".g."
merge (Litter l) = show l++ ".L"

formatR :: Int -> String -> String
formatR n str =
    take n (replicate (n - length str) ' ' ++ str)

showEuro :: Int -> String
showEuro i =
    show (div i 100) ++ "."
    show (mod (div i 10) 10) ++
    show (mod i 10) ++ ".EU"

inventur :: Lager -> Int
inventur (Lager l) = sum (map cent l)

```

Lager

Refakturierung im Einkaufsparadies

module Shopper where

import Data.Maybe

— Modellierung der Artikel.

```
data Apfelsorte = Boskoop | CoxOrange | GrannySmith
    deriving (Eq, Show)

apreis :: Apfelsorte -> Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50
```

```
data Kassensorte = Gouda | Appenzeller
```

deriving (Eq, Show)

```
ipreis :: Kassensorte -> Double
```

ipreis Gouda = 1400

ipreis Appenzeller = 2200

```
data Bio = Bio | Konv
```

deriving (Eq, Show)

```
data Artikel =
```

Apfelsorte | Eier

| Kassensorte | Schinken

| Salami | Milch Bio

deriving (Eq, Show)

```
data Merge = Stuck Int | Gamm Int | Liter Double
```

deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel -> Merge# Preis

preis (Apfelsort a) (Stuck n) = Just (n*apreis a)

preis Eier (Stuck n) = Just (n*20)

preis (Kassensort k) (Gamm g) = Just (round (frontintegral g* 3000* ipreis k))

preis Schinken (Gamm g) = Just (div (g* 100 300))

preis Salami (Gamm g) = Just (div (g* 150 300))

preis MilchBio (Litter l) =

Just (round (l* case Bio of Bio a => 110; Konv a => 60))

preis _ = Nothing

— Addition von Mengen

addiere :: Merge -> Merge

addiere (Stuck n) (Stuck m) = Stuck (1+n)

addiere (Gamm n) (Gamm m) = Gamm (n+m)

addiere (Litter n) (Litter m) = Litter (1+n)

addiere m n = error ("addiere: "++ show m ++ " und "++ show n)

— Posten:

```
data Posten = Posten Artikel Merge
```

deriving (Eq, Show)

cent :: Posten -> Int

cent (Posten a m) = fromMaybe 0 (preis a m) — gibt keinen Laufzeitfehler!

— Lagerhaltung:

```
data Lager = Lager [Posten]
```

deriving (Eq, Show)

leeresLager :: Lager

leeresLager = Lager []

Artikel

suche :: Artikel -> Lager -> Maybe Maybe

suche a (Lager ps) =

listToMaybe [m | Posten la m =: ps, la == a]

einlagen :: Artikel -> Merge -> Lager

einlagen a m (Lager ps) =

let hinein a m l = [Posten a m |

hinein a m l) = (Posten a m : addiere m m l): l

| otherwise = (Posten a m l: hinein a m l)

in case preis a m of

Nothing => Lager ps

_ => Lager (hinein a m ps)

Lager

data Einkaufswagen = Eweg [Posten]

deriving (Eq, Show)

leererWagen :: Einkaufswagen

leererWagen = Eweg []

einkauf :: Artikel -> Merge -> Einkaufswagen

einkauf a m (Eweg ps) =

| lg us = (preis a m) = Eweg (Posten a m: ps)

| otherwise = Eweg ps

kasse :: Einkaufswagen -> Int

kasse (Eweg ps) = sum (map cent ps)

kaessbon = Einkaufswagen -> String

kaessbon ps@Eweg ps) =

"Börs's, Aufde,Grocery,Shoppe,n,n"++

"Artikel,Menge,Preis,n"++

concatMap artikel ps ++

"Summe"++ formatR 31 (showEuro (kasse ps))

artikel :: Posten -> String

artikel (Posten a m) =

formatR 20 (show a) ++

formatR 7 (merge) ++

formatR 10 (showEuro (cent p)) ++ "\n"

merge :: Merge -> String

merge (Stuck n) = show n ++ ".St"

merge (Gamm g) = show g ++ ".g."

merge (Litter l) = show l ++ ".L."

formatR :: Int -> String -> String

formatR n str =

take n (replicate (n - length str) ' ' ++ str)

showEuro :: Int -> String

showEuro i =

show (div i 100) ++ "," ++

show (mod (div i 10) 10) ++

show (mod i 10) ++ ".EU"

Inventur :: Lager -> Int

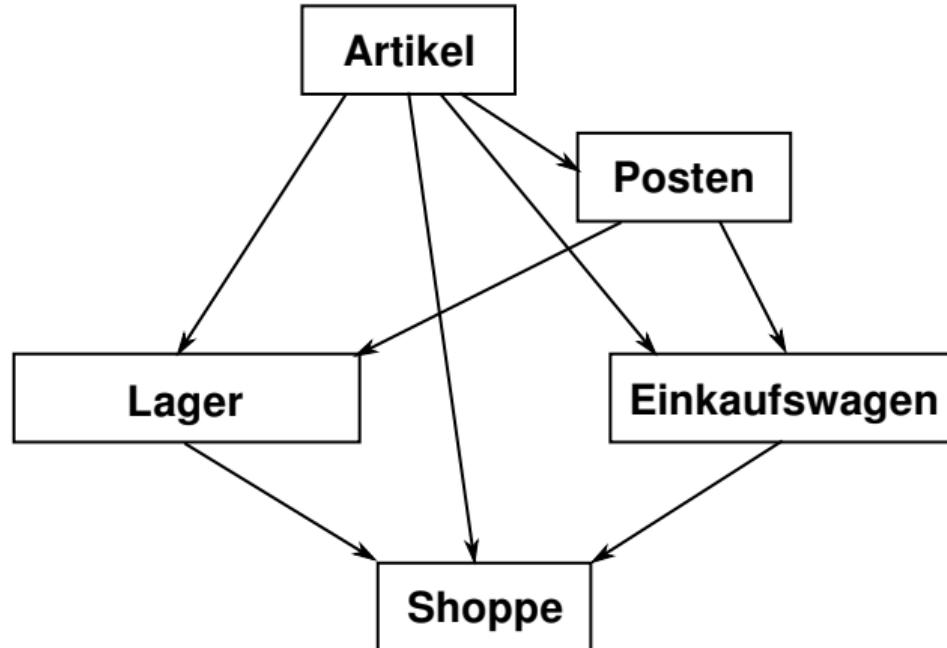
Inventur (Lager l) = sum (map cent l)

Einkaufswagen

Posten

Lager

Refakturierung im Einkaufsparadies: Modulararchitektur



Refakturierung im Einkaufsparadies I: Artikel

- ▶ Es wird **alles** exportiert
- ▶ Reine Datenmodellierung

```
module Artikel where
```

```
data Apfelsorte = Boskoop | CoxOrange | GrannySmith
apreis :: Apfelsorte → Int
```

```
data Kaesesorte = Gouda | Appenzeller
kpreis :: Kaesesorte → Double
```

```
data Menge = Stueck Int | Gramm Int | Liter Double
addiere :: Menge → Menge → Menge
```

Refakturierung im Einkaufsparadies II: Posten

- ▶ Implementiert ADT Posten:

```
data Posten = Posten Artikel Menge
             deriving (Eq, Show)
```

```
module Posten(
  Posten,
  artikel,
  menge,
  posten,
  cent,
  hinzu) where
```

```
artikel :: Posten → Artikel
artikel (Posten a _) = a
```

- ▶ Konstruktor wird **nicht** exportiert
- ▶ Invariante: Posten hat immer die korrekte Menge zu Artikel

```
posten :: Artikel → Menge → Maybe Posten
posten a m =
  case preis a m of
    Just _ → Just (Posten a m)
    Nothing → Nothing
```

Refakturierung im Einkaufsparadies III: Lager

```
module Lager(  
    Lager,  
    leeresLager,  
    einlagern,  
    suche,  
    liste,  
    inventur  
) where  
import Artikel  
import Posten
```

- ▶ Implementiert ADT Lager

```
data Lager
```

- ▶ Signatur der exportierten Funktionen:

```
leeresLager :: Lager
```

```
einlagern :: Artikel → Menge → Lager → Lager
```

```
suche a (Lager l) = M.lookup a l
```

```
liste (Lager m) = M.toList m
```

```
inventur = sum ∘ map (fromJust ∘ uncurry preis) ∘ liste
```

- ▶ **Invariante:** Lager enthält keine doppelten Artikel

Refakturierung im Einkaufsparadies IV: Einkaufswagen

```
module Einkaufswagen(
    Einkaufswagen,
    leererWagen,
    einkauf,
    kasse,
    kassenbon
) where
```

- ▶ ADT durch **Verkapselung**:

```
data Einkaufswagen = Ekwg [Posten]
    deriving (Eq, Show)
```

- ▶ Ein Typsynonym würde exportiert
- ▶ **Invariante**: Korrekte Menge zu Artikel im Einkaufswagen
 - einkauf :: Artikel → Menge → Einkaufswagen
→ Einkaufswagen
 - einkauf a m (Ekwg ps) = **case** posten a m **of**
 - Just p → Ekwg (p: ps)
 - Nothing → Ekwg ps
- ▶ Nutzt dazu ADT Posten

Refakturierung im Einkaufsparadies V: Hauptmodul

```
module Shoppe where

import Artikel
import Lager
import Einkaufswagen
```

- ▶ Nutzt andere Module

```
w0= leererWagen
w1= einkauf (Apfel Boskoop) (Stueck 3) w0
w2= einkauf Schinken (Gramm 50) w1
w3= einkauf (Milch Bio) (Liter 1) w2
w4= einkauf Schinken (Gramm 50) w3
```

Benutzung von ADTs

- ▶ **Operationen** und **Typen** müssen **importiert** werden
- ▶ Möglichkeiten des Imports:
 - ▶ **Alles** importieren
 - ▶ **Nur bestimmte** Operationen und **Typen** importieren
 - ▶ Bestimmte Typen und Operationen **nicht** importieren

Importe in Haskell

- ▶ Syntax:

```
import [qualified] M [as N] [hiding] [(Bezeichner)]
```

- ▶ *Bezeichner* geben an, **was** importiert werden soll:
 - ▶ Ohne Bezeichner wird **alles** importiert
 - ▶ Mit **hiding** werden Bezeichner **nicht** importiert
- ▶ Für jeden exportierten Bezeichner **f** aus **M** wird importiert
 - ▶ **f** und **qualifizierter** Bezeichner **M.f**
 - ▶ **qualified**: **nur qualifizierter** Bezeichner **M.f**
 - ▶ Umbenennung bei Import mit **as** (dann **N.f**)
 - ▶ Klasseninstanzen und Typsynonyme werden immer importiert
- ▶ Alle Importe stehen immer am **Anfang** des Moduls

Beispiel

```
module M(a,b) where  
...  
...
```

Import(e)	Bekannte Bezeichner
import M	

Beispiel

```
module M(a,b) where  
...  
...
```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	

Beispiel

```
module M(a,b) where
...

```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(nothing)
import M(a)	

Beispiel

```
module M(a,b) where
...
...
```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(nothing)
import M(a)	a, M.a
import qualified M	

Beispiel

```
module M(a,b) where
...

```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(nothing)
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	

Beispiel

```
module M(a,b) where  
...  
  
```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(<i>nothing</i>)
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	(<i>nothing</i>)
import qualified M(a)	

Beispiel

```
module M(a,b) where  
...  
  
```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(<i>nothing</i>)
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	(<i>nothing</i>)
import qualified M(a)	M.a
import M hiding ()	

Beispiel

```
module M(a,b) where
...

```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(nothing)
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	(nothing)
import qualified M(a)	M.a
import M hiding ()	a, b, M.a, M.b
import M hiding (a)	

Beispiel

```
module M(a,b) where
...

```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(nothing)
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	(nothing)
import qualified M(a)	M.a
import M hiding ()	a, b, M.a, M.b
import M hiding (a)	b, M.b
import qualified M hiding ()	

Beispiel

```
module M(a,b) where
...

```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(<i>nothing</i>)
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	(<i>nothing</i>)
import qualified M(a)	M.a
import M hiding ()	a, b, M.a, M.b
import M hiding (a)	b, M.b
import qualified M hiding ()	M.a, M.b
import qualified M hiding (a)	

Beispiel

```
module M(a,b) where
...

```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(<i>nothing</i>)
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	(<i>nothing</i>)
import qualified M(a)	M.a
import M hiding ()	a, b, M.a, M.b
import M hiding (a)	b, M.b
import qualified M hiding ()	M.a, M.b
import qualified M hiding (a)	M.b
import M as B	

Beispiel

```
module M(a,b) where
...

```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(<i>nothing</i>)
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	(<i>nothing</i>)
import qualified M(a)	M.a
import M hiding ()	a, b, M.a, M.b
import M hiding (a)	b, M.b
import qualified M hiding ()	M.a, M.b
import qualified M hiding (a)	M.b
import M as B	a, b, B.a, B.b
import M as B(a)	

Beispiel

```
module M(a,b) where
...

```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(<i>nothing</i>)
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	(<i>nothing</i>)
import qualified M(a)	M.a
import M hiding ()	a, b, M.a, M.b
import M hiding (a)	b, M.b
import qualified M hiding ()	M.a, M.b
import qualified M hiding (a)	M.b
import M as B	a, b, B.a, B.b
import M as B(a)	a, B.a
import qualified M as B	

Beispiel

```
module M(a,b) where  
...  
  
```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	(<i>nothing</i>)
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	(<i>nothing</i>)
import qualified M(a)	M.a
import M hiding ()	a, b, M.a, M.b
import M hiding (a)	b, M.b
import qualified M hiding ()	M.a, M.b
import qualified M hiding (a)	M.b
import M as B	a, b, B.a, B.b
import M as B(a)	a, B.a
import qualified M as B	B.a, B.b

Quelle: Haskell98-Report, Sect. 5.3.4

Ein typisches Beispiel

- ▶ Modul implementiert Funktion, die auch importiert wird
- ▶ Umbenennung nicht immer praktisch
- ▶ Qualifizierter Import führt zu **langen** Bezeichnern
- ▶ Einkaufswagen implementiert Funktionen `artikel` und `menge`, die auch aus Posten importiert werden:

```
import Posten hiding (artikel, menge)
import qualified Posten as P(artikel, menge)
```

```
artikel :: Posten → String
artikel p =
  formatL 20 (show (P.artikel p)) ++
  formatR 7 (menge (P.menge p)) ++
  formatR 10 (showEuro (cent p)) ++ "\n"
```

Was zum Nachdenken

Übung 8.1: Import

Warum schreibt man

```
import Prelude hiding (repeat)
```

und was bewirkt das? (Hinweis: `Prelude` ist das Modul der vordefinierten Funktionen.)

Was zum Nachdenken

Übung 8.1: Import

Warum schreibt man

```
import Prelude hiding (repeat)
```

und was bewirkt das? (Hinweis: `Prelude` ist das Modul der vordefinierten Funktionen.)

Lösung: Die Import-Anweisung `import` alle vordefinierten Funktionen **bis auf** `repeat`. Dadurch können wir `repeat` selber (anders) definieren.

II. Schnittstelle vs. Implementation

Schnittstelle vs. Implementation

- ▶ Gleiche **Schnittstelle** kann unterschiedliche **Implementationen** haben
- ▶ Beispiel: (endliche) Abbildungen

Endliche Abbildungen

- ▶ Viel gebraucht, oft in Abwandlungen (Hashtables, Sets, Arrays)
- ▶ Abstrakter Datentyp für **endliche Abbildungen**:

- ▶ Datentyp

```
data Map α β
```

- ▶ Leere Abbildung:

```
empty :: Map α β
```

- ▶ Abbildung auslesen:

```
lookup :: Ord α ⇒ α → Map α β → Maybe β
```

- ▶ Abbildung ändern:

```
insert :: Ord α ⇒ α ⇒ β ⇒ Map α β → Map α β
```

- ▶ Abbildung löschen:

```
delete :: Ord α ⇒ α ⇒ Map α β → Map α β
```

Eine naheliegende Implementation

- Modellierung als Haskell-Funktion:

```
data Map α β = Map (α → Maybe β)
```

- Damit einfaches `lookup`, `insert`, `delete`:

```
empty = Map (λx → Nothing)
```

```
lookup a (Map s) = s a
```

```
insert a b (Map s) = Map (λx → if x == a then Just b else s x)
```

```
delete a (Map s) = Map (λx → if x == a then Nothing else s x)
```

- Instanzen von `Eq`, `Show` **nicht möglich**

- **Speicherleck**: überschriebene Zellen werden nicht freigegeben

Endliche Abbildungen: Anwendungsbeispiel

- Lager als endliche Abbildung:

```
data Lager = Lager (M.Map Artikel Menge)
```

- Artikel suchen:

```
suche a (Lager l) = M.lookup a l
```

- Ins Lager hinzufügen:

```
einlagern :: Artikel → Menge → Lager → Lager
einlagern a m (Lager l) = case posten a m of
    Just _ → case M.lookup a l of
        Just q → Lager (M.insert a (addiere m q) l)
        Nothing → Lager (M.insert a m l)
    Nothing → Lager l
```

- Für Inventur fehlt Möglichkeit zur **Iteration**
- Daher: Map als **Assoziativliste**

Übung 8.2: Die Map als Assoziativliste

```
data Map α β = Map [(α, β)]
insert :: Ord α ⇒ α → β → Map α β → Map α β
insert a b m = (a, b) : m
```

Was ist der Nachteil dieser einfachen Implementation?

Mitmachfolie

Übung 8.2: Die Map als Assoziativliste

```
data Map α β = Map [(α, β)]
insert :: Ord α ⇒ α → β → Map α β → Map α β
insert a b m = (a, b) : m
```

Was ist der Nachteil dieser einfachen Implementation?

Lösung: Erzeugt ein Speicherleck — überschriebene Elemente bleiben in der Liste. Besser: beim Einfügen alte Elemente entfernen

```
insert :: Ord α ⇒ α → β → Map α β → Map α β
insert a b xs = (a, b) : filter ((a ≠ ).fst) xs
```

Nicht sehr effizient. Besser: Map als **sortierte** Liste.

Map als sortierte Assoziativliste

```
data Map α β = Map { toList :: [(α, β)] }
```

- ▶ Invariante: Liste ist in der ersten Komponente aufsteigend sortiert
- ▶ `lookup` ist vordefiniert; beim einfügen auch überschreiben;

```
insert :: Ord α ⇒ α ⇒ β ⇒ Map α β ⇒ Map α β
insert a v (Map s) = Map (insert' s) where
  insert' []          = [(a, v)]
  insert' s0@(b, w):s | a > b = (b, w): insert' s
                      | a == b = (a, v): s
                      | a < b = (a, v): s0
```

- ▶ ... ist aber **ineffizient** (Zugriff/Löschen in $\mathcal{O}(n)$)
- ▶ Deshalb: **balancierte Bäume**

AVL-Bäume und Balancierte Bäume

AVL-Bäume

Ein Baum ist **ausgeglichen**, wenn

- ▶ alle Unterbäume ausgeglichen sind, und
- ▶ der Höhenunterschied zwischen zwei Unterbäumen höchstens eins beträgt.

Balancierte Bäume

Ein Baum ist **balanciert**, wenn

- ▶ alle Unterbäume balanciert sind, und
- ▶ für den linken und rechten Unterbaum l, r gilt:

$$\text{size}(l) \leq w \cdot \text{size}(r) \quad (1)$$

$$\text{size}(r) \leq w \cdot \text{size}(l) \quad (2)$$

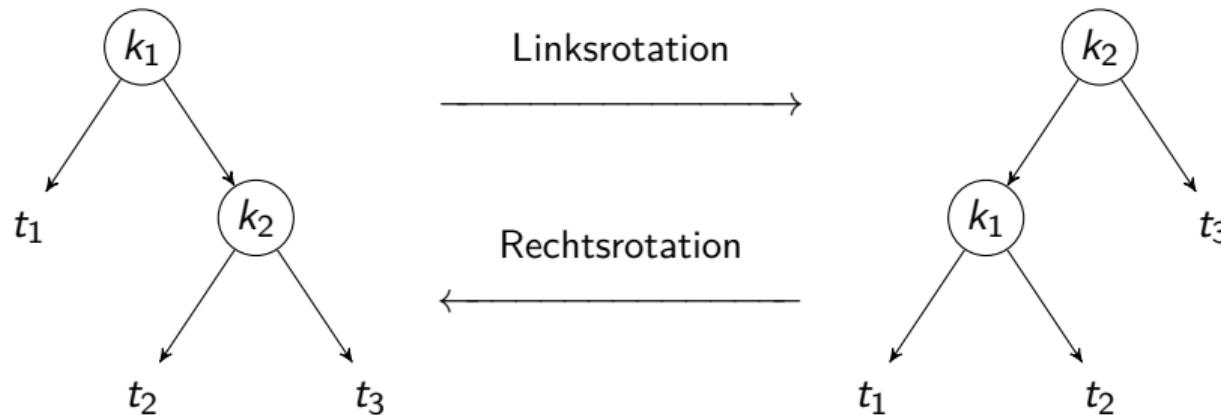
w — **Gewichtung** (Parameter des Algorithmus)

Implementation

- ▶ Balanciertheit ist **Invariante**
- ▶ Nach Einfügen oder Löschen: Balanciertheit wiederherstellen
- ▶ Dabei drei Fälle:
 - ① Linker Unterbaum größer $size(l) > w \cdot size(r)$
 - ② Rechter Unterbaum größer $size(r) > w \cdot size(l)$
 - ③ Keiner größer — Baum balanciert

Balanciertheit durch Einfache Rotation

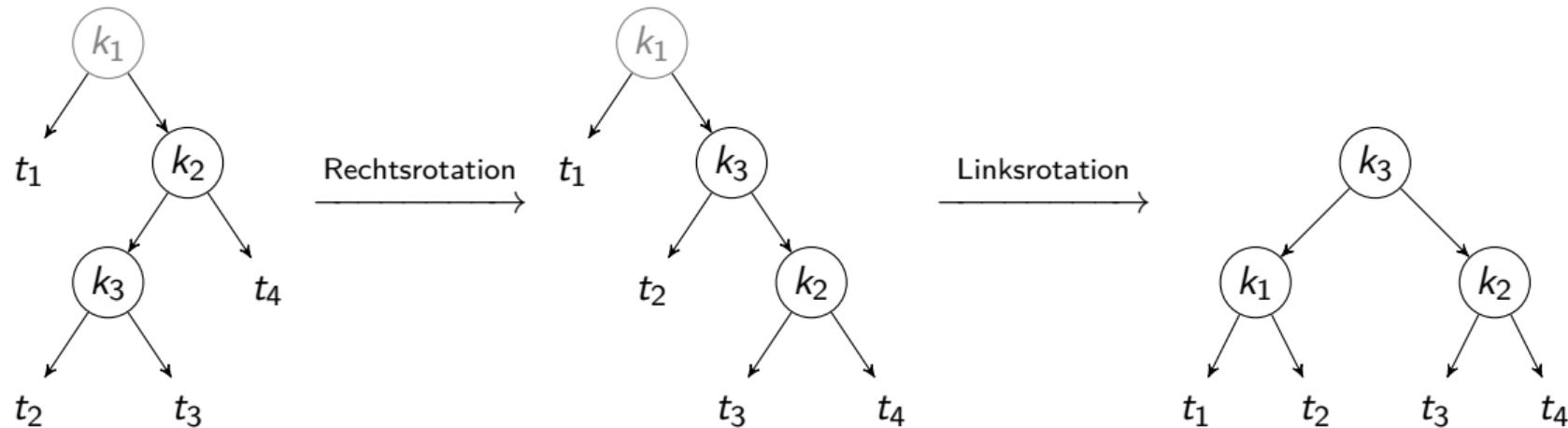
- Sei der rechte Unterbaum größer
- Zwei Unterfälle:
 - 1 Linkes Enkelkind t_2 größer
 - 2 Rechtes Enkelkind t_3 größer
- Einfache **Linksrotation** heilt (2)
- Ansonsten: **Doppelrotation** reduziert (1) zu (2)



Balanciertheit durch Doppelrotation

Falls linkes Enkelkind um Faktor α größer als rechtes:

- ▶ Nach einer einfachen Rechtsrotation des Unterbaumes ist rechtes Enkelkind größer
- ▶ Danach Linksrotation des gesamten Baumes



Implementation in Haskell

- ▶ Der Datentyp

```
data Map α β = Empty
              | Node α β Int (Map α β) (Map α β)
              deriving Eq
```

- ▶ Parameter:

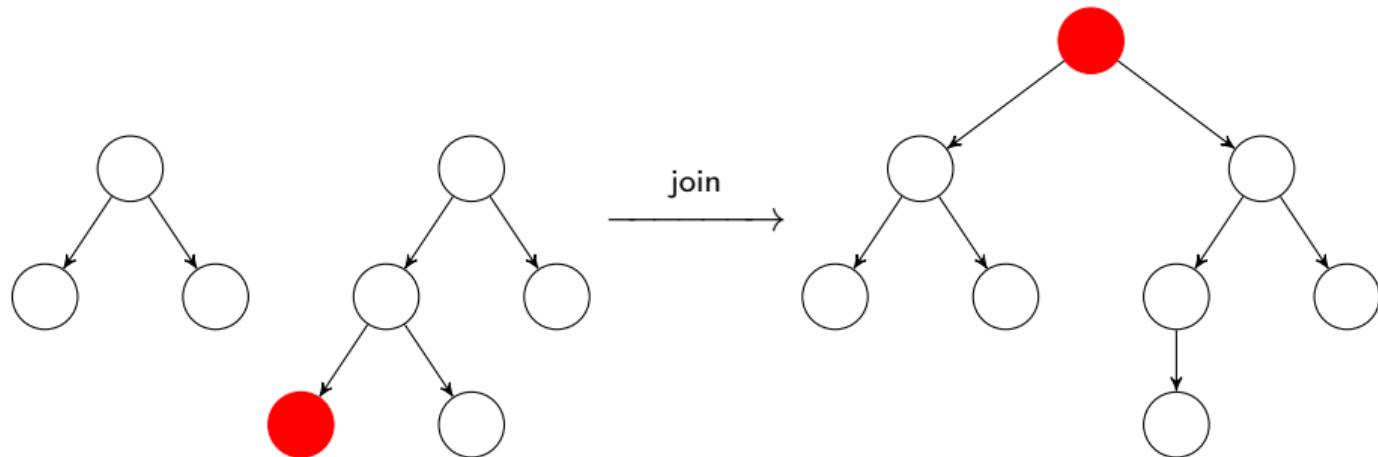
- ▶ `weight` Gewichtsfaktor w (für Einfachrotation)
- ▶ `ratio` Gewichtsfaktor α (für Doppelrotation)
- ▶ Hilfskonstruktor `node`, setzt Größe (`l`, `r` balanciert)
- ▶ Selektor `size` für Größe des Baumes (0 für `Empty`)

Hauptfunktion

- ▶ `balance k x l r` konstruiert balancierten Baum
- ▶ `l, r` sind balanciert und höchstens um einen Knoten unbalanciert
- ▶ Vier Fälle:
 - ① Beide Bäume zusammen höchstens einen Knoten → keine Rotation
 - ② $w \cdot \text{size}(l) < \text{size}(r)$: → Linksrotation
 - ③ $\text{size}(l) > w \cdot \text{size}(r)$: → Rechtsrotation
 - ④ Ansonsten: keine Rotation
- ▶ `balanceL k x l r` rotiert nach links. Sei r_l und r_r rechter und linker Unterbaum von r :
 - ① $\text{size}(r_l) < \alpha \cdot \text{size}(r_r)$, dann einfache Linksrotation
 - ② $\text{size}(r_l) \geq \alpha \cdot \text{size}(r_r)$ dann Doppelrotation (Rechtsrotation r , dann Linksrotation)

Hilfsfunktion join beim Löschen

- ▶ Zwei balancierte Bäume zusammenfügen (nachdem Wurzel gelöscht wurde)
- ▶ Linkester Knoten des rechten Unterbaumes wird neue Wurzel
- ▶ Mit `balance` wieder ausbalancieren



Was zum Selbermachen

Übung 8.3: Use the Source, Luke!

Ladet euch von der Webseite der Veranstaltung die Quellen für die 8. Vorlesung herunter, und öffnet die Datei `MapTree.hs`.

Vergleicht die Haskell-Implementation mit den Beschreibung der Folien.

Welche der Funktionen `lookup`, `insert`, `delete` könnte man als `fold` realisieren?

Was zum Selbermachen

Übung 8.3: Use the Source, Luke!

Ladet euch von der Webseite der Veranstaltung die Quellen für die 8. Vorlesung herunter, und öffnet die Datei `MapTree.hs`.

Vergleicht die Haskell-Implementation mit den Beschreibung der Folien.

Welche der Funktionen `lookup`, `insert`, `delete` könnte man als `fold` realisieren?

Lösung: `lookup` lässt sich falten:

```
lookup' k = fold (\ak ax l r → if k == ak then Just ax  
                           else maybe r Just l) Nothing
```

Ist aber nicht so effizient (linear statt logarithmisch), weil es immer erst links, dann rechts sucht.

Zusammenfassung Balancierte Bäume

- ▶ Auslesen, einfügen und löschen: logarithmischer Aufwand ($\mathcal{O}(\log n)$)
- ▶ Fold: linearer Aufwand ($\mathcal{O}(n)$)
- ▶ Guten durchschnittlicher Aufwand
- ▶ Auch in der Haskell-Bücherei: `Data.Map` (schwer optimiert, mit vielen weiteren Funktionen)

Benchmarking: Setup

- ▶ Wie **schnell** sind die Implementationen **wirklich**?
- ▶ Benchmarking: nicht trivial
 - ▶ Verzögerte Auswertung und optimierender Compiler
 - ▶ Messen wir das **richtige**?
 - ▶ Benchmarking-Tool: Criterion
- ▶ Setup: Map Int String mit 50000 zufälligen Einträgen erzeugen
- ▶ Darin:
 - ▶ Einmal zufällig lesen (**lookup**), schreiben (**insert**), löschen (**delete**)
 - ▶ Sequenz aus fünfmal löschen und schreiben, zweihundertmal lesen (**mixed**)

Benchmarking: Resultate

	create	lookup	insert	delete	mixed
MapFun	333,3 ms 13,58 ms	1,634 ms 52,25 μ s	11,27 ns 130,8 ps	11,20 ns 120,3 ps	1,659 ms 79,22 μ s
MapList	5,629 s 168,7 ms	32,70 μ s 9,625 μ s	96,12 μ s 1,294 μ s	101,4 μ s 18,47 μ s	6,182 ms 2,059 μ s
MapTree	383,9 ms 19,62 ms	404,1 ns 135,3 ns	119,4 μ s 13,18 μ s	117,1 μ s 42,82 μ s	2,803 ms 521,5 μ s
Data.Map.Lazy	473,0 ms 44,97 ms	221,6 ns 59,58 ns	104,7 μ s 49,66 μ s	112,7 μ s 11,39 μ s	2,396 ms 278,8 μ s

Einträge: durchschnittl. Ausführungszeit, Standardabweichung

Defizite von Haskells Modulsystem

- ▶ Signatur ist nur **implizit**
 - ▶ Exportliste enthält nur Bezeichner
 - ▶ Wünschenswert: Signatur an der Exportliste annotierbar, oder Signaturen in separater Datei
 - ▶ In Java: **Interfaces**
- ▶ Klasseninstanzen werden **immer** exportiert.
- ▶ Kein **Paket-System**

Zusammenfassung

- ▶ **Abstrakte Datentypen** (ADTs):
 - ▶ Besteht aus **Typen** und **Operationen** darauf
 - ▶ Realisierung in Haskell durch **Module**
 - ▶ Beispieldatentypen: endliche Abbildungen
 - ▶ Nächste Vorlesung: ADTs durch **Eigenschaften** spezifizieren

A photograph of a snowy winter scene. In the foreground, there are several snow-covered evergreen trees. A person wearing a red jacket and dark pants is walking away from the camera, positioned in the center of the frame. The ground is covered in a thick layer of snow, and the overall atmosphere is cold and serene.

Frohe Weihnachten und einen Guten Rutsch!

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 9 vom 11.01.2021: Signaturen und Eigenschaften

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Organisatorisches

- ▶ Anmeldung zur **Klausur**:
 - ▶ Ab **Dienstag** bis **Ende der Woche** auf stud.ip (unverbindlich)
 - ▶ Ersetzt **nicht** die **Modulanmeldung**
- ▶ Klausurtermine:
 - ▶ Klausur: 03.02.2020, 10:00/11:30/15:00
 - ▶ Wiederholungstermin: 21.04.2020, 10:00/11:30/15:00
- ▶ Probeklausur (alte Klausuren vom letzten Jahr) werden veröffentlicht.
- ▶ Fragenkatalog für mündliche Prüfung
- ▶ Es gibt noch eine Extra-Sendung zur mündlichen Prüfung.

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ **Teil II: Funktionale Programmierung im Großen**
 - ▶ Abstrakte Datentypen
 - ▶ Signaturen und Eigenschaften
- ▶ Teil III: Funktionale Programmierung im richtigen Leben

Abstrakte Datentypen und Signaturen

- ▶ Letzte Vorlesung: **Abstrakte Datentypen**

- ▶ Typ plus Operationen

- ▶ Heute: **Signaturen** und **Eigenschaften**

Definition (Signatur)

Die **Signatur** eines abstrakten Datentyps besteht aus den Typen, und der Signatur der darüber definierten Funktionen.

- ▶ Keine direkte Repräsentation in Haskell
- ▶ Signatur: **Typ** eines Moduls

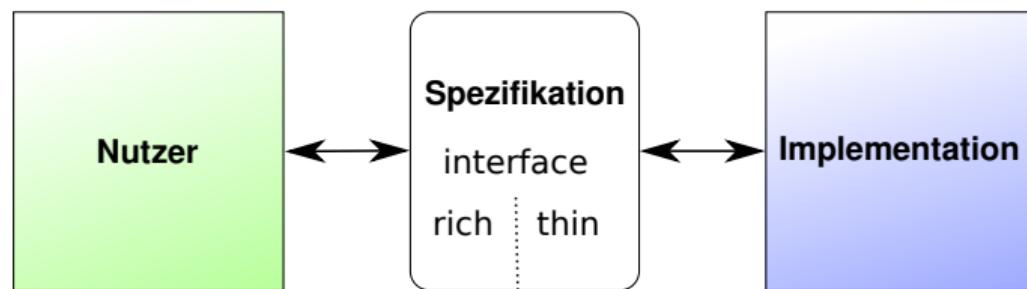
I. Eigenschaften

Signatur und Eigenschaften

- ▶ Signatur genug, um ADT **typkorrekt** zu benutzen
 - ▶ Insbesondere **Anwendbarkeit** und **Reihenfolge**
- ▶ Signatur beschreibt nicht die **Bedeutung** (Semantik):
 - ▶ Was wird **gelesen**?
 - ▶ Wie verhält sich die Abbildung?
- ▶ Signatur ist **Sprache** (Syntax) um **Eigenschaften** zu beschreiben

Axiome als Interface

- ▶ Axiome müssen **gelten**
 - ▶ für **alle** Werte der freien Variablen zu True auswerten
- ▶ Axiome **spezifizieren**:
 - ▶ nach außen das **Verhalten** (viele Operationen und Eigenschaften — *rich interface*)
 - ▶ nach innen die **Implementation** (wenig Operationen und Eigenschaften — *thin interface*)
- ▶ Signatur + Axiome = **Spezifikation**



Eigenschaften endlicher Abbildungen

Übung 9.1: Was denkt ihr?

Überlegt mindestens **drei** weitere Eigenschaften endlicher Abbildungen!

- 1 Aus der **leeren** Abbildung kann **nichts** gelesen werden.

Eigenschaften endlicher Abbildungen

Übung 9.1: Was denkt ihr?

Überlegt mindestens **drei** weitere Eigenschaften endlicher Abbildungen!

- 1 Aus der **leeren** Abbildung kann **nichts** gelesen werden.
- 2 Wenn etwas **gelesen** wird an der **gleichen** Stelle, an der etwas **geschrieben** worden ist, erhalte ich den geschriebenen Wert.
- 3 Wenn etwas **gelesen** wird an einer **anderen** Stelle, an der etwas **geschrieben** worden ist, kann das Schreiben vernachlässigt werden.
- 4 An der **gleichen** Stelle **zweimal geschrieben** überschreibt der zweite den ersten Wert.
- 5 An unterschiedlichen Stellen **geschrieben** kommutiert.

Formalisierung von Eigenschaften

- Ziel: Eigenschaften **formal** beschreiben, um sie testen oder beweisen zu können.

Definition (Axiome)

Axiome sind Prädikate über den Operationen der Signatur

- Elementare Prädikate P :
 - Gleichheit $s = t$, Ordnung $s < t$
 - Selbstdefinierte Prädikate
- Zusammengesetzte Prädikate
 - Negation **not** p , Konjunktion $p \&& q$, Disjunktion $p \mid\mid q$
 - **Implikation** $p \implies q$

Endliche Abbildung: Signatur für Map

- Adressen und Werte sind Parameter
- Typ $\text{Map } \alpha \beta$, Operationen:

```
data Map α β
```

```
empty :: Map α β
```

```
lookup :: Ord α ⇒ α → Map α β → Maybe β
```

```
insert :: Ord α ⇒ α → β → Map α β → Map α β
```

```
delete :: Ord α ⇒ α → Map α β → Map α β
```

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) = Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) = Nothing
```

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (insert a v (s :: Map Int String)) = Just v
```

```
lookup a (delete a (s :: Map Int String)) = Nothing
```

- ▶ Lesen an anderer Stelle liefert alten Wert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) = Nothing
```

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (insert a v (s :: Map Int String)) = Just v
```

```
lookup a (delete a (s :: Map Int String)) = Nothing
```

- ▶ Lesen an anderer Stelle liefert alten Wert:

```
a ≠ b ⇒ lookup a (delete b s) = lookup a (s :: Map Int String)
```

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) = Nothing
```

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (insert a v (s :: Map Int String)) = Just v
```

```
lookup a (delete a (s :: Map Int String)) = Nothing
```

- ▶ Lesen an anderer Stelle liefert alten Wert:

```
a ≠ b ⇒ lookup a (delete b s) = lookup a (s :: Map Int String)
```

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

```
insert a w (insert a v s) = insert a w (s :: Map Int String)
```

- ▶ Schreiben über verschiedene Stellen kommutiert:

Axiome für Map

- ▶ Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) = Nothing
```

- ▶ Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (insert a v (s :: Map Int String)) = Just v
```

```
lookup a (delete a (s :: Map Int String)) = Nothing
```

- ▶ Lesen an anderer Stelle liefert alten Wert:

```
a ≠ b ⇒ lookup a (delete b s) = lookup a (s :: Map Int String)
```

- ▶ Schreiben an dieselbe Stelle überschreibt alten Wert:

```
insert a w (insert a v s) = insert a w (s :: Map Int String)
```

- ▶ Schreiben über verschiedene Stellen kommutiert:

```
a ≠ b ⇒ insert a v (delete b s) = delete b (insert a vs)
```

- ▶ Sehr **viele** Axiome (insgesamt 13)!

Thin vs. Rich Interfaces

- ▶ Benutzersicht: **reiches** Interface
 - ▶ Viele Operationen und Eigenschaften
- ▶ Implementationssicht: **schlankes** Interface
 - ▶ Wenig Operation und Eigenschaften
- ▶ Konversion dazwischen („Adapter“)

Thin vs. Rich Maps

- Rich interface:

```
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
```

```
delete :: Ord α⇒ α→ Map α β→ Map α β
```

- Thin interface:

```
put :: Ord α⇒ α→ Maybe β→ Map α β→ Map α β
```

- Konversion von thin auf rich:

```
insert a v = put a (Just v)
```

```
delete a = put a Nothing
```

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefined:

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v s) == v
```

- Lesen an anderer Stelle liefert alten Wert:

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v s) == v
```

- Lesen an anderer Stelle liefert alten Wert:

```
a ≠ b ⇒ lookup a (put b c s) == lookup a s
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v s) == v
```

- Lesen an anderer Stelle liefert alten Wert:

```
a ≠ b ⇒ lookup a (put b c s) == lookup a s
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

```
put a w (put a v s) == put a w s
```

- Schreiben über verschiedene Stellen kommutiert:

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v s) == v
```

- Lesen an anderer Stelle liefert alten Wert:

```
a ≠ b ⇒ lookup a (put b c s) == lookup a s
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

```
put a w (put a v s) == put a w s
```

- Schreiben über verschiedene Stellen kommutiert:

```
a ≠ b ⇒ put a v (put b w s) == put b w (put a v s)
```

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v s) == v
```

- Lesen an anderer Stelle liefert alten Wert:

```
a ≠ b ⇒ lookup a (put b c s) == lookup a s
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

```
put a w (put a v s) == put a w s
```

- Schreiben über verschiedene Stellen kommutiert:

```
a ≠ b ⇒ put a v (put b w s) == put b w (put a v s)
```

Thin: 5 Axiome
Rich: 13 Axiome

Quick Question

Übung 9.2: Gleichheiten

Betrachtet die letzten beiden Fälle:

$$\text{put } a \text{ } w \text{ (put } a \text{ } v \text{ } s) = \text{put } a \text{ } w \text{ } s$$

$$a \neq b \Rightarrow \text{put } a \text{ } v \text{ (put } b \text{ } w \text{ } s) = \text{put } b \text{ } w \text{ (put } a \text{ } v \text{ } s)$$

Wiese müssen wir die Fälle $a = b$ und $a \neq b$, aber nicht $w = v$ und $w \neq v$ unterscheiden?

Quick Question

Übung 9.2: Gleichheiten

Betrachtet die letzten beiden Fälle:

$$\text{put } a \text{ } w \text{ (put } a \text{ } v \text{ } s) = \text{put } a \text{ } w \text{ } s$$

$$a \neq b \Rightarrow \text{put } a \text{ } v \text{ (put } b \text{ } w \text{ } s) = \text{put } b \text{ } w \text{ (put } a \text{ } v \text{ } s)$$

Wiese müssen wir die Fälle $a = b$ und $a \neq b$, aber nicht $w = v$ und $w \neq v$ unterscheiden?

Lösung: Im Gegensatz zu a und b gelten beide Axiome sowohl für $w = v$ als auch für $w \neq v$:

$$\text{put } a \text{ } w \text{ (put } a \text{ } w \text{ } s) = \text{put } a \text{ } w \text{ } s$$

$$a \neq b \Rightarrow \text{put } a \text{ } w \text{ (put } b \text{ } w \text{ } s) = \text{put } b \text{ } w \text{ (put } a \text{ } w \text{ } s)$$

II. Testen von Eigenschaften

Axiome als Eigenschaften

- ▶ Axiome können **getestet** oder **bewiesen** werden
- ▶ Tests finden Fehler, Beweis zeigt **Korrektheit**

E. W. Dijkstra, 1972

Program testing can be used to show the presence of bugs, but never to show their absence.

- ▶ Arten von Tests:
 - ▶ Unit tests (JUnit, HUnit)
 - ▶ Black Box vs. **White Box**
 - ▶ Coverage-based (z.B. path coverage, MC/DC)
 - ▶ **Zufallsbasiertes** Testen
- ▶ Funktionale Programme eignen sich **sehr gut** zum Testen

Zufallsbasiertes Testen in Haskell

- ▶ Werkzeug: *QuickCheck*
- ▶ Zufällige Werte einsetzen, Auswertung auf `True` prüfen
- ▶ Polymorphe Variablen nicht `testbar`
 - ▶ Deshalb Typvariablen **instantiiieren**
 - ▶ Typ muss genug Element haben (hier `Map Int String`)
 - ▶ Durch Signatur `Typinstanz` erzwingen
- ▶ **Freie Variablen** der Eigenschaft werden `Parameter` der Testfunktion

Axiome mit *QuickCheck* testen

- ▶ Eigenschaften als **monomorphe Haskell-Prädikate**
- ▶ Für das Lesen:

```
prop1 :: TestTree
prop1 = QC.testProperty "read_empty" $ λa →
  lookup a (empty :: Map Int String) === Nothing
```

```
prop2 :: TestTree
prop2 = QC.testProperty "lookup_put_eq" $ λa v s →
  lookup a (put a v (s :: Map Int String)) === v
```

- ▶ *QuickCheck*-Axiome mit `QC.testProperty` in *Tasty* eingebettet
- ▶ Es werden N Zufallswerte generiert und getestet (Default $N = 100$)

Axiome mit *QuickCheck* testen

- ▶ **Bedingte** Eigenschaften:
 - ▶ $A \implies B$ mit A, B Eigenschaften
 - ▶ Typ ist **Property**
 - ▶ Es werden solange Zufallswerte generiert, bis N die Vorbedingung erfüllende gefunden und getestet wurden, andere werden ignoriert.

```
prop3 :: TestTree
prop3 = QC.testProperty "lookup_put\u222aother" $ \a b v s\-
  a \neq b \implies lookup a (put b v s) == lookup a (s :: Map Int String)
```

Axiome mit *QuickCheck* testen

- ▶ **Schreiben:**

```
prop4 :: TestTree
prop4 = QC.testProperty "put_put_eq" $ λa v w s→
  put a w (put a v s) === put a w (s :: Map Int String)
```

- ▶ **Schreiben** an anderer Stelle:

```
prop5 :: TestTree
prop5 = QC.testProperty "put_put_other" $ λa v b w s→
  a ≠ b ⇒ put a v (put b w s) === put b w (put a v s :: Map Int String)
```

- ▶ Test benötigt **Gleichheit** und **Zufallswerte** für Map a b

Beobachtbare und Abstrakte Typen

- ▶ **Beobachtbare** Typen: interne Struktur bekannt
 - ▶ Vordefinierte Typen ([Zahlen](#), [Zeichen](#)), algebraische Datentypen ([Listen](#))
 - ▶ Viele Eigenschaften und Prädikate bekannt
- ▶ **Abstrakte** Typen: interne Struktur unbekannt
 - ▶ Wenige Eigenschaften bekannt, Gleichheit nur wenn definiert
- ▶ Beispiel [Map](#):
 - ▶ [beobachtbar](#): Adressen und Werte
 - ▶ [abstrakt](#): Speicher

Beobachtbare Gleichheit

- ▶ Auf abstrakten Typen: nur **beobachtbare** Gleichheit
- ▶ Zwei Elemente sind **gleich**, wenn alle Operationen die gleichen Werte liefern
- ▶ Bei **Implementation**: Instanz für **Eq** (**Ord** etc.) entsprechend definieren
 - ▶ Die Gleichheit **==** muss die **beobachtbare** Gleichheit sein.
- ▶ Abgeleitete Gleichheit (**deriving Eq**) wird **immer** exportiert!

Zufallswerte selbst erzeugen

- ▶ Problem: **Zufällige** Werte von **selbstdefinierten** Datentypen
 - ▶ Gleichverteiltheit nicht immer erwünscht (z.B. $[\alpha]$)
 - ▶ Konstruktion nicht immer offensichtlich (z.B. `Map`)
- ▶ In *QuickCheck*:
 - ▶ **Typklasse** `class Arbitrary α` für Zufallswerte
 - ▶ Eigene **Instanziierung** kann Verteilung und Konstruktion berücksichtigen

```
instance (Ord a, QC.Arbitrary a, QC.Arbitrary b) ⇒
    QC.Arbitrary (Map a b) where
```
- ▶ Zufallswerte in Haskell?

Zufällige Maps erzeugen

- ▶ Erster Ansatz: zufällige Länge, dann aus sovielen zufälligen Werten `Map` konstruieren
- ▶ Berücksichtigt `delete` nicht
- ▶ Besser: über einen **smart constructor** zufällige Maps erzeugen
 - ▶ Muss entweder in `Map` implementiert werden
 - ▶ oder benötigt Zugriff auf interne Struktur

Was stimmt hier nicht?

Übung 9.3: Map als balancierte Bäume.

Warum ist diese Implementierung von Map als binärer Baum falsch?

```
data Map α β = Empty
  | Node α β Int (Map α β) (Map α β)
  deriving Eq
```

Was stimmt hier nicht?

Übung 9.3: Map als balancierte Bäume.

Warum ist diese Implementierung von Map als binärer Baum falsch?

```
data Map α β = Empty
  | Node α β Int (Map α β) (Map α β)
  deriving Eq
```

Lösung: Weil die abgeleitete Gleichheit nicht die beobachtbare Gleichheit ist. Die Gleichheit darf nur prüfen, ob die gleichen Schlüssel/Wert-Paare enthalten sind:

```
toList :: Map α β → [(α, β)]
toList = fold (λk x l r → l++[(k,x)]++r) []
```

```
instance (Eq α, Eq β) ⇒ Eq (Map α β) where
  t1 = t2 = toList t1 = toList t2
```

III. Syntax und Semantik

Signatur und Semantik

Stacks

Typ: $St \ \alpha$

Initialwert:

```
empty :: St α
```

Wert ein/auslesen:

```
push :: α → St α → St α
```

```
top :: St α → α
```

```
pop :: St α → St α
```

Last in first out (LIFO).

Queues

Typ: $Qu \ \alpha$

Initialwert:

```
empty :: Qu α
```

Wert ein/auslesen:

```
enq :: α → Qu α → Qu α
```

```
first :: Qu α → α
```

```
deq :: Qu α → Qu α
```

First in first out (FIFO)

Gleiche **Signatur**, unterschiedliche **Semantik**.

Eigenschaften von Stack

- ▶ Last in first out (LIFO):

$$\text{top}(\text{push } a_1 (\text{push } a_2 \dots (\text{push } a_n \text{ empty}))) = a_1$$

Eigenschaften von Stack

- ▶ Last in first out (LIFO):

$$\text{top}(\text{push } a_1 (\text{push } a_2 \dots (\text{push } a_n \text{ empty}))) = a_1$$

top (push a s) = a

pop (push a s) = s

push a s \neq empty

Eigenschaften von Queue

- ▶ First in first out (FIFO):

$$\text{first}(\text{enq } a_1 (\text{enq } a_2 \dots (\text{enq } a_n \text{ empty}))) = a_n$$

Eigenschaften von Queue

- ▶ First in first out (FIFO):

$$\text{first}(\text{enq } a_1 (\text{enq } a_2 \dots (\text{enq } a_n \text{ empty}))) = a_n$$

$$\text{first}(\text{enq } a \text{ empty}) = a$$

$$q \neq \text{empty} \implies \text{first}(\text{enq } a \text{ } q) = \text{first } q$$

$$\text{deq}(\text{enq } a \text{ empty}) = \text{empty}$$

$$q \neq \text{empty} \implies \text{deq}(\text{enq } a \text{ } q) = \text{enq } a \text{ } (\text{deq } q)$$

$$\text{enq } a \text{ } q \neq \text{empty}$$

Implementation von Stack: Liste

Sehr einfach: ein Stack ist eine Liste

```
data St a = St [a] deriving (Show, Eq)
```

```
empty = St []
```

```
push a (St s) = St (a:s)
```

```
top (St []) = error "St: top on empty stack"  
top (St s) = head s
```

```
pop (St []) = error "St: pop on empty stack"  
pop (St s) = St (tail s)
```

Implementation von Queue

- ▶ Mit einer Liste?
 - ▶ Problem: am Ende anfügen oder abnehmen (`last/init`) ist teuer ($O(n)$).
- ▶ Deshalb **zwei** Listen:
 - ▶ Erste Liste: zu **entnehmende** Elemente
 - ▶ Zweite Liste: **hinzugefügte** Elemente **rückwärts**
 - ▶ Invariante: erste Liste leer gdw. Queue leer

Repräsentation von Queue

Operation

Resultat

Interne Repräsentation

first

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])	9

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])	9
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])	9

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])	9
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])	9
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])	4

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])	9
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])	9
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])	4
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])	4

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])	9
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])	9
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])	4
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])	4
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])	4

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])	9
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])	9
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])	4
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])	4
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])	4
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	([7], [3, 5])	7

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])	9
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])	9
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])	4
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])	4
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])	4
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	([7], [3, 5])	7
deq	$\langle 3 \rightarrow 5 \rangle$	([5, 3], [])	5

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])	9
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])	9
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])	4
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])	4
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])	4
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	([7], [3, 5])	7
deq	$\langle 3 \rightarrow 5 \rangle$	([5, 3], [])	5
deq	$\langle 3 \rangle$	([3], [])	3

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])	9
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])	9
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])	4
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])	4
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])	4
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	([7], [3, 5])	7
deq	$\langle 3 \rightarrow 5 \rangle$	([5, 3], [])	5
deq	$\langle 3 \rangle$	([3], [])	3
deq	$\langle \rangle$	([], [])	error

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation	first
empty	$\langle \rangle$	([], [])	error
enq 9	$\langle 9 \rangle$	([9], [])	9
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])	9
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])	9
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])	4
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])	4
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])	4
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	([7], [3, 5])	7
deq	$\langle 3 \rightarrow 5 \rangle$	([5, 3], [])	5
deq	$\langle 3 \rangle$	([3], [])	3
deq	$\langle \rangle$	([], [])	error
deq	error		

Implementation: Datentyp

- ▶ Datentyp:

```
data Qu α = Qu [α] [α]
```

- ▶ Invariante:

- ① Anfang der Schlange ist der **Kopf** der ersten Liste
- ② Wenn erste Liste leer, dann ist auch die zweite Liste leer

- ▶ Invariante prüfen und ggf. herstellen (**smart constructor**):

```
queue :: [α] → [α] → Qu α
queue [] ys = Qu (reverse ys) []
queue xs ys = Qu xs ys
```

Implementation: Gleichheit

Übung 9.4:

Warum reicht für Gleichheit auf Schlangen nicht `derive Eq` und wie implementieren wir es dann?

Implementation: Gleichheit

Übung 9.4:

Warum reicht für Gleichheit auf Schlangen nicht `derive Eq` und wie implementieren wir es dann?

Lösung:

- Gegenbeispiel:

$$q_1 = \text{deq}(\text{enq} 7(\text{enq} 4(\text{enq} 9 \text{ empty}))), q_2 = \text{enq}(7(\text{enq} 4 \text{ empty}))$$

- Zwei Schlangen sind gleich, wenn der **Inhalt gleich** ist:

```
instance Eq α⇒ Eq (Qu α) where
  Qu xs1 ys1 == Qu xs2 ys2 =
    xs1 ++ reverse ys1 == xs2 ++ reverse ys2
```

Implementation: Operationen

- Leere Schlange: alles leer

```
empty :: Qu α
empty = Qu [] []
```

- Erstes Element steht vorne in erster Liste

```
first :: Qu α → α
first (Qu [] _) = error "Queue:_first_of_empty_Q"
first (Qu (x:xs) _) = x
```

- Bei `enq` und `deq` Invariante prüfen (Funktion `queue`)

```
enq :: α → Qu α → Qu α
enq x (Qu xs ys) = queue xs (x:ys)
```

```
deq :: Qu α → Qu α
deq (Qu [] _) = error "Queue:_deq_of_empty_Q"
deq (Qu (_:xs) ys) = queue xs ys
```

Zusammenfassung

- ▶ **Signatur**: Typ und Operationen eines ADT
- ▶ **Axiome**: über Typen formulierte **Eigenschaften**
- ▶ **Spezifikation** = Signatur + Axiome
 - ▶ Interface zwischen Implementierung und Nutzung
 - ▶ Testen zur Erhöhung der Konfidenz und zum Fehlerfinden
 - ▶ Beweisen der Korrektheit
- ▶ **QuickCheck**:
 - ▶ Freie Variablen der Eigenschaften werden **Parameter** der Testfunktion
 - ▶ ⇒ für **bedingte** Eigenschaften

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 10 vom 18.01.2021: Aktionen und Zustände

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Fahrplan

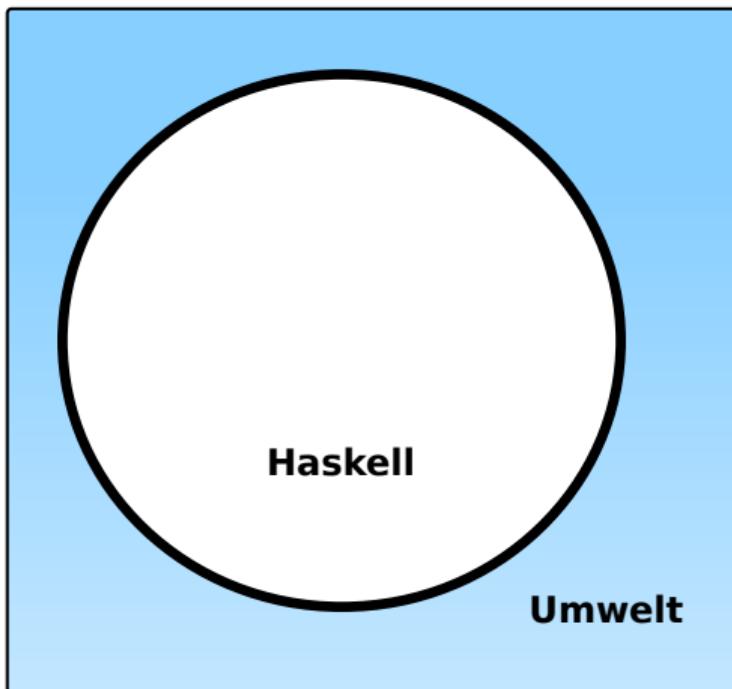
- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ **Teil III: Funktionale Programmierung im richtigen Leben**
 - ▶ Aktionen und Zustände
 - ▶ Monaden als Berechnungsmuster
 - ▶ Funktionale Webanwendungen
 - ▶ Scala — Eine praktische Einführung
 -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

Inhalt

- ▶ Ein/Ausgabe in funktionale Sprachen
- ▶ Wo ist das **Problem**?
- ▶ **Aktionen** und der Datentyp *IO*.
- ▶ Vordefinierte Aktionen
- ▶ Beispiel: Wortratespiel
- ▶ Aktionen als Werte

I. Funktionale Ein/Ausgabe

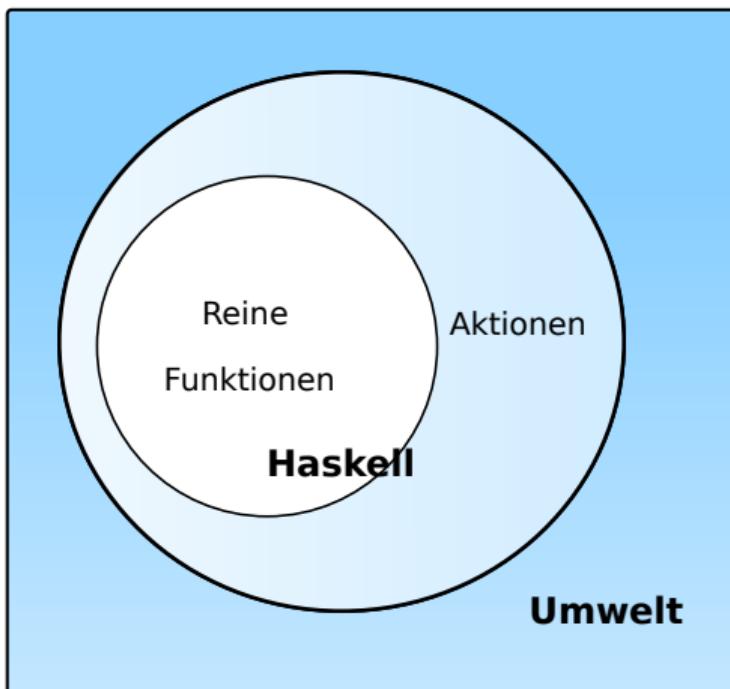
Ein- und Ausgabe in funktionalen Sprachen



Problem:

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

Ein- und Ausgabe in funktionalen Sprachen



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
```

- ▶ Dazu **elementare** Aktionen (lesen, schreiben etc)

Elementare Aktionen

- Zeile von Standardeingabe (`stdin`) **lesen**:

```
getLine :: IO String
```

- Zeichenkette auf Standardausgabe (`stdout`) **ausgeben**:

```
putStr :: String → IO ()
```

- Zeichenkette mit Zeilenvorschub **ausgeben**:

```
putStrLn :: String → IO ()
```

Einfache Beispiele

► Echo einfach

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

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
```



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

- ▶ Rechts sind $\gg=$, \gg implizit
- ▶ Mit \leftarrow gebundene Bezeichner **überlagern** vorherige
- ▶ Es gilt die **Abseitsregel**.
- ▶ Einrückung der ersten Anweisung nach **do** bestimmt Abseits.

Drittes Beispiel

- ▶ Zählendes, endliches Echo

```
echo3 :: Int → IO ()  
echo3 cnt = do  
    putStrLn (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

Zeit für eine Pause

Übung 10.1: Say My Name!

Wie sieht ein Haskell-Programm aus, das erst nach dem Namen des Gegenübers fragt, und dann mit **Hallo, Christoph!** (oder was eingegeben wurde) freundlich grüßt?

Zeit für eine Pause

Übung 10.1: Say My Name!

Wie sieht ein Haskell-Programm aus, das erst nach dem Namen des Gegenübers fragt, und dann mit `Hallo, Christoph!` (oder was eingegeben wurde) freundlich grüßt?

Lösung:

```
greeter :: IO ()  
greeter = do  
    putStrLn "What's your name, love?"  
    s ← getLine  
    putStrLn $ "Hello, " ++ s ++ ". Pleased to meet you."
```

- ▶ `putStr` statt `putStrLn` erlaubt „Prompting“
- ▶ Argumente von `putStrLn` klammern (oder \$)

II. Aktionen als Werte

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]
```

Jetzt ihr!

Übung 10.2: Eine „While-Schleife“ in Haskell

Schreibt einen Kombinator

`while :: IO Bool → IO α → IO ()`

der solange das zweite Argument (den Rumpf) auswertet wie das erste Argument zu `True` auswertet.

Jetzt ihr!

Übung 10.2: Eine „While-Schleife“ in Haskell

Schreibt einen Kombinator

`while :: IO Bool → IO α → IO ()`

der solange das zweite Argument (den Rumpf) auswertet wie das erste Argument zu `True` auswertet.

Lösung:

► Erste Lösung:

```
while c b = do a ← c; if a then b ≫ while c b else return ()
```

- Vorteil: ist **endrekursiv**.
- Wieso eigentlich `IO ()`?

III. Ein/Ausgabe

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
```

- ▶ “Lazy I/O”: Zugriff auf Dateien erfolgt **verzögert**

- ▶ Interaktion von nicht-strikter Auswertung mit zustandsbasiertem Dateisystem kann überraschend sein

Beispiel: Zeichen, Wörter, Zeilen zählen (wc)

```
wc :: String → IO ()
wc file =
  do cont ← readFile file
     putStrLn $ file ++ ":"
     show (length (lines cont)) ++ "lines," ++
     show (length (words cont)) ++ "words," ++
     show (length cont) ++ "bytes."
```

- ▶ Datei wird gelesen
- ▶ Anzahl Zeichen, Worte, Zeilen gezählt
- ▶ Erstaunlich (hinreichend) effizient

Ein/Ausgabe mit Dateien: Abstraktionsebenen

- ▶ **Einfach:** `readFile`, `writeFile`
- ▶ **Fortgeschritten:** Modul `System.IO` der Standardbücherei
 - ▶ `Buffered`/`Unbuffered`, `Seeking`, &c.
 - ▶ Operationen auf `Handle`
- ▶ **Systemnah:** Modul `System.Posix`
 - ▶ `Filedeskriptoren`, `Permissions`, `special devices`, etc.

IV. Ausnahmen und Fehlerbehandlung

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  $\gamma \Rightarrow \gamma \rightarrow \alpha$ 
catch :: Exception  $\gamma \Rightarrow \text{IO } \alpha \rightarrow (\gamma \rightarrow \text{IO } \alpha) \rightarrow \text{IO } \alpha$ 
try    :: Exception  $\gamma \Rightarrow \text{IO } \alpha \rightarrow \text{IO } (\text{Either } \gamma \alpha)$ 
```

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

Fehler fangen und behandeln

“Ask forgiveness not permission” (Grace Hopper)

Generelle Regel: **Fehlerbehandlung** durch **Ausnahmebehandlung** besser als vorherige Abfrage von Fehlerbedingungen.

- ▶ Warum?

Fehler fangen und behandeln

“Ask forgiveness not permission” (Grace Hopper)

Generelle Regel: **Fehlerbehandlung** durch **Ausnahmebehandlung** besser als vorherige Abfrage von Fehlerbedingungen.

- ▶ Warum? Umwelt nicht **sequentiell**.
- ▶ Fehlerbehandlung für `wc`:

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

- ▶ `IOException` kann analysiert werden (siehe `System.IO.Error`)
- ▶ `read` mit Ausnahme bei Fehler (statt Programmabbruch):

```
readIO :: Read α ⇒ String → IO α
```

Ausführbare Programme

- ▶ Eigenständiges Programm ist **Aktion**
- ▶ **Hauptaktion**: `main :: IO ()` in Modul `Main`
 - ▶ ... oder mit der Option `-main-is M.f` setzen
- ▶ `wc` als eigenständiges Programm:

```
module Main where

import System.Environment (getArgs)
import Control.Exception

main :: IO ()
main = do
  args ← getArgs
  putStrLn $ "Command-line arguments: " ++ show args
  mapM_ wc2 args
```

Beispiel: Traversal eines Verzeichnisbaums

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

```
travFS :: (FilePath → IO ()) → FilePath → IO ()  
travFS action p = catch (do  
    cs ← getDirectoryContents p  
    let cp = map (p </>) (cs \\ [".", ".."])  
    dirs ← filterM doesDirectoryExist cp  
    files ← filterM doesFileExist cp  
    mapM_ action files  
    mapM_ (travFS action) dirs  
(λe → putStrLn $ "ERROR:" ++ show (e :: IOException))
```

- ▶ Nutzt Funktionalität aus `System.Directory`, `System.FilePath`

Übung 10.3: Alles zählt

Kombiniert `Traverse` und `WC` zu einem Programm

```
ls :: FilePath → IO ()
```

welches in einem gegeben Verzeichnis den Inhalt aller darin enthaltenen Dateien zählt.

Übung 10.3: Alles zählt

Kombiniert `Traverse` und `WC` zu einem Programm

```
ls :: FilePath → IO ()
```

welches in einem gegeben Verzeichnis den Inhalt aller darin enthaltenen Dateien zählt.

Lösung: `wc2` (mit Fehlerbehandlung) wird einfach die Traversionsfunktion:

```
ls = travFS wc2
```

Das ist alles.

V. Anwendungsbeispiel

So ein Zufall!

- Zufallswerte:

```
randomIO :: (α, α) → IO α
```

- Warum ist `randomIO` **Aktion**?

So ein Zufall!

- Zufallswerte:

```
randomIO :: (α, α) → IO α
```

- Warum ist `randomIO` **Aktion**?

► Beispiele:

- Aktion zufällig oft ausführen:

```
atmost :: Int → IO α → IO [α]
atmost most a =
  do l ← randomIO (1, most)
    sequence (replicate l a)
```

- Zufälligen String erzeugen:

```
randomStr :: IO String
randomStr = atmost 40 (randomIO ('a', 'z'))
```

- Hinweis: Funktionen aus `System.Random` zu importieren, muss ggf. installiert werden.

Fallbeispiel: Wörter raten

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

Wörter raten: Programmstruktur

- ▶ Hauptschleife:

```
play :: String → String → String → IO ()
```

- ▶ Argumente: Geheimnis, geratene Buchstaben (enthalten, nicht enthalten)

- ▶ Benutzereingabe:

```
getGuess :: String → String → IO Char
```

- ▶ Argumente: geratene Zeichen (im Geheimnis enthalten, nicht enthalten)

- ▶ Hauptfunktion:

```
main :: IO ()
```

- ▶ Liest ein Lexikon, wählt Geheimnis aus, ruft Hauptschleife auf

Nunc est ludendum.

Übung 10.3: Linguistic Interlude

Ladet den Quellcode herunter, übersetzt das Spiel und ratet fünf Wörter.

Wer noch etwas tun möchte, kann das Spiel so erweitern, dass es nachdem das Wort erfolgreich geraten wurde, ein neues Wort rät, und insgesamt zählt, wieviele Worte schon (nicht) geraten wurden.

Zusammenfassung

- ▶ Ein/Ausgabe in Haskell durch **Aktionen**
- ▶ **Aktionen** (Typ $\text{IO } \alpha$) sind seiteneffektbehaftete Funktionen
- ▶ Komposition von Aktionen durch

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

- ▶ **do**-Notation
- ▶ Fehlerbehandlung durch Ausnahmen (`IOError`, `catch`, `try`).
- ▶ Verschiedene Funktionen der Standardbücherei:
 - ▶ Prelude: `getLine`, `putStr`, `putStrLn`, `readFile`, `writeFile`
 - ▶ Module: `System.IO`, `System.Random`
- ▶ Nächste Vorlesung: Wie sind Aktionen eigentlich **implementiert**? Schwarze Magie?

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 11 vom 25.01.2021: Monaden als Berechnungsmuster

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Organisatorisches

- ▶ Die Klausur am 03.02. ist gestern von der Uni **abgesagt** worden.
- ▶ Es bleibt der Klausurtermin am 21.04.2020.
- ▶ Wir bemühen uns um eine zusätzlichen Wiederholungstermin im Sommersemester.

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

Inhalt

- ▶ Wie geht das mit `IO`?
- ▶ Monaden als allgemeines Berechnungsmuster
- ▶ Fallbeispiel: Auswertung von Ausdrücken

Lernziele

Wir verstehen, wie wir Berechnungsmuster wie Seiteneffekte, Partialität oder Mehrdeutigkeit in Haskell funktional modellieren.

I. Zustandsabhängige Berechnungen

Funktionen mit Zustand

- ▶ Idee: Seiteneffekt **explizit** machen
- ▶ Funktion $f : \alpha \rightarrow \beta$ mit Seiteneffekt in **Zustand** σ :

$$\begin{aligned} f : \alpha \times \sigma &\rightarrow \beta \times \sigma \\ &\cong \\ f : \alpha &\rightarrow \sigma \rightarrow \beta \times \sigma \end{aligned}$$

- ▶ Datentyp für Zustand σ : $\sigma \rightarrow \beta \times \sigma$
- ▶ 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  :: ( $\sigma \rightarrow \alpha$ )  $\rightarrow$  State  $\sigma$   $\alpha$   
get f s = (f s, s)
```

- ▶ Zustand setzen:

```
set  :: ( $\sigma \rightarrow \sigma$ )  $\rightarrow$  State  $\sigma$  ()  
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
```

Food for Thought

Übung 11.1: Verkapselung

Warum **müssen** wir den Datentyp $\text{State } \sigma \alpha$ in einen Datentyp verkapseln, und wie sieht dessen Signatur aus?

Food for Thought

Übung 11.1: Verkapselung

Warum **müssen** wir den Datentyp $\text{State } \sigma \alpha$ in einen Datentyp verkapseln, und wie sieht dessen Signatur aus?

Lösung: Wenn wir den Zustand explizit durch die Gegend reichen, können wir ihn beliebig kopieren — das ist sicherlich nicht beabsichtigt, es sollte immer nur genau eine Kopie des Zustands geben.

Die Signatur besteht aus `comp`, `lift`, `map`, `get` und `set` — siehe nächsten Abschnitt.

II. 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**

Was ist ein Berechnungsmuster?

- ▶ Ein **Berechnungsmuster** hat eine **Einheit** und kann **verknüpft** werden.
- ▶ Beispiele:
 - ▶ **Seiteneffekte** (Zustand),
 - ▶ **Fehler** (Partialität),
 - ▶ **Mehrdeutigkeit**,
 - ▶ **Aktionen**.
- ▶ Eine Monade ist ein **Typkonstruktor**, der zu einem Typ **Berechnungsmuster hinzufügt**.
- ▶ **Mathematisch** ist eine Monade eine **verallgemeinerte algebraische Theorie** (durch Operationen und Gleichungen definiert).

Monaden in Haskell

- ▶ Monaden sind erstmal Funktoren:

```
class Functor f where
  fmap :: (α → β) → f α → f β
```

- ▶ Es sollte gelten (kann nicht geprüft werden):

$$\text{fmap id} = \text{id}$$

$$\text{fmap f} \circ \text{fmap g} = \text{fmap (f} \circ \text{g)}$$

- ▶ Standard: “*Instances of Functor should satisfy the following laws.*”

Monaden in Haskell

- ▶ Verkettung ($\gg=$) und Lifting (return):

```
class (Functor m, Applicative m)⇒ Monad m where
  (»=)  :: m α → (α → m β) → m β
  return :: α → m α
```

$\gg=$ ist assoziativ und return das neutrale Element:

```
return a »= k  =  k a
m »= return  =  m
m »= (x → k x »= h)  =  (m »= k) »= h
```

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

Beispiele für Monaden

- ▶ Zustandsmonaden: ST, State, Reader, Writer
- ▶ 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 und Either 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
```

```
instance Functor (Either ε) where
  fmap f (Right b) = Right (f b)
  fmap f (Left a) = Left a
```

```
instance Monad (Either ε) where
  Right b >>= g = g b
  Left a >>= _ = Left a
  return = Right
```

- Berechnungsmodell: **Ausnahmen** (Fehler)

- $f : \alpha \rightarrow \text{Maybe } \beta$ ist Berechnung mit möglichem (unspezifiertem) Fehler,
- $f : \alpha \rightarrow \text{Either } \epsilon \alpha$ ist Berechnung mit möglichem Fehler vom Typ ϵ
- 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
 - ▶ $f :: \alpha \rightarrow \beta$ ist Berechnung mit **mehreren** möglichen Ergebnissen
 - ▶ Verkettung: Anwendung der folgenden Funktion auf **jedes** Ergebnis

Beispiel

- Berechnung aller Permutationen einer Liste:

- ➊ 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
```

- ➋ Damit Permutationen (rekursiv):

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

Jetzt seit ihr dran.

Übung 11.2: Komposition in der Listenmonade

Betrachten wir noch mal die Komposition in der Listenmonade:

$$\begin{aligned} a : as \gg= g &= g \ a \ \# \ (as \gg= g) \\ [] \gg= g &= [] \end{aligned}$$

Welche uns (hoffentlich) wohlbekannte Funktion versteckt sich dahinter?

Jetzt seit ihr dran.

Übung 11.2: Komposition in der Listenmonade

Betrachten wir noch mal die Komposition in der Listenmonade:

a : as $\gg=$ g = g a + (as $\gg=$ g)
[] $\gg=$ g = []

Welche uns (hoffentlich) wohlbekannte Funktion versteckt sich dahinter?

Lösung: Das ist dasselbe wie `concatMap`, nur mit umgedrehten Argumenten:

`concatMap :: (α → [β]) → [α] → [β]`

```
concatMap f = concat ∘ map f
```

`(▷▷) ≡ flip concatMap`

Der Listenmonade in der Listenkomprehension

- ▶ Berechnung aller Permutationen einer Liste:

- ➊ Ein Element überall in eine Liste einfügen:

```
ins' :: α → [[α]]  
ins' x [] = [[x]]  
ins' x (y:ys) = [x:y:ys] ++ [ y:is | is ← ins' x ys ]
```

- ➋ Damit Permutationen (rekursiv):

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

- ▶ Listenkomprehension \cong Listenmonade

III. IO ist keine Magie

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 $A \rightarrow V$ (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

IV. Fallbeispiel: Auswertung von Ausdrücken

Monaden im Einsatz

- ▶ Auswertung von Ausdrücken:

Algebraische Ausdrücke:

```
data Expr = Var String
          | Num Double
          | Plus Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
          | Div Expr Expr
```

Monaden im Einsatz

- ▶ Auswertung von Ausdrücken:

Algebraische Ausdrücke:

```
data Expr = Var String
          | Num Double
          | Plus Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
          | Div Expr Expr
```

Auswertung ohne Effekte:

```
eval :: Expr → Double
eval (Var _) = 0
eval (Num n) = n
eval (Plus a b) = eval a + eval b
eval (Minus a b) = eval a - eval b
eval (Times a b) = eval a * eval b
eval (Div a b) = eval a / eval b
```

- ▶ Mögliche Arten von Effekten:

- ▶ Partialität (Division durch 0)
- ▶ Zustände (für die Variablen)
- ▶ Mehrdeutigkeit

Monaden im Einsatz

- ▶ Auswertung von Ausdrücken:

Algebraische Ausdrücke:

```
data Expr = Var String
          | Num Double
          | Plus Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
          | Div Expr Expr
```

Auswertung ohne Effekte:

```
eval :: Expr → Double
eval (Var _) = 0
eval (Num n) = n
eval (Plus a b) = eval a + eval b
eval (Minus a b) = eval a - eval b
eval (Times a b) = eval a * eval b
eval (Div a b) = eval a / eval b
```

- ▶ Mögliche Arten von Effekten:

- ▶ Partialität (Division durch 0)
- ▶ Zustände (für die Variablen)
- ▶ Mehrdeutigkeit

Auswertung mit Fehlern

- ▶ Partialität durch Fehlermonade (Either):

```
eval :: Expr → Either String Double
eval (Var x)  = Left $ "No variable" ++ x
eval (Num n)  = return n
eval (Plus a b) = do x ← eval a; y ← eval b; return $ x + y
eval (Minus a b) = do x ← eval a; y ← eval b; return $ x - y
eval (Times a b) = do x ← eval a; y ← eval b; return $ x * y
eval (Div a b)  = do
  x ← eval a; y ← eval b;
  if y == 0 then Left "Division by zero" else Right $ x / y
```

Auswertung mit Zustand

- Zustand durch Reader-Monade

```
import ReaderMonad
import qualified Data.Map as M

type State = M.Map String Double

eval :: Expr → Reader State Double
eval (Var i) = get (M.! i)
eval (Num n) = return n
eval (Plus a b) = do x ← eval a; y ← eval b; return $ x+ y
eval (Minus a b) = do x ← eval a; y ← eval b; return $ x- y
eval (Times a b) = do x ← eval a; y ← eval b; return $ x* y
eval (Div a b) = do x ← eval a; y ← eval b; return $ x/ y
```

Mehrdeutige Auswertung

- Dazu: Erweiterung von Expr:

```
data Expr = Var String
          | ...
          | Pick Expr Expr
```

```
eval :: Expr → [Double]
eval (Var i) = return 0
eval (Num n) = return n
eval (Plus a b) = do x ← eval a; y ← eval b; return $ x+ y
eval (Minus a b) = do x ← eval a; y ← eval b; return $ x- y
eval (Times a b) = do x ← eval a; y ← eval b; return $ x* y
eval (Div a b) = do x ← eval a; y ← eval b; return $ x/ y
eval (Pick a b) = do x ← eval a; y ← eval b; [x, y]
```

Kombination der Effekte

- ▶ Benötigt **Kombination** der Monaden.
- ▶ Monade **Res**:
 - ▶ Zustandsabhängig
 - ▶ Mehrdeutig
 - ▶ Fehlerbehaftet

```
type Exn α = Either String α
data Res σ α = Res { run :: σ → [Exn α] }
```

- ▶ Berechnungen sind von einem Zustand abhängig, der mehrere Ergebnisse geben kann, von denen einige Fehler sein können.
- ▶ Andere Kombinationen möglich.

Food For Thought.

Übung 11.3: Andere Kombinationen sind möglich:

- ❶ `data Res σ α = Res (σ → Exn [α])`
- ❷ `data Res σ α = Res (Exn [σ → α])`
- ❸ `data Res σ α = Res ([σ → Exn α])`

Was für eine Art Berechnung modellieren diese, und was ist hier der Unterschied?

Food For Thought.

Übung 11.3: Andere Kombinationen sind möglich:

- ❶ `data Res σ α = Res (σ → Exn [α])`
- ❷ `data Res σ α = Res (Exn [σ → α])`
- ❸ `data Res σ α = Res ([σ → Exn α])`

Was für eine Art Berechnung modellieren diese, und was ist hier der Unterschied?

Lösung:

- ❶ Berechnungen sind von einem Zustand abhängig, und geben entweder einen Fehler oder eine Liste von Ergebnissen;
- ❷ Berechnungen sind entweder fehlerhaft, oder eine Liste von Funktionen, die zu jedem Zustand ein Ergebnis liefern;
- ❸ Berechnungen sind eine Liste von Funktionen, die zu jedem Zustand entweder ein Fehler oder ein Ergebnis liefern können.

Unterschied zwischen (i) und (ii)/(iii): für (i) kann es für einen Zustand mehrere Ergebnisse geben, bei (ii)/(iii) für einen Zustand nur ein Ergebnis/Fehler.

Übung 11.4: Bonusfrage

Wir hatten also als Kombinationen

- ❶ `data Res σ α = Res (σ → [Exn α])`
- ❷ `data Res σ α = Res (σ → Exn [α])`
- ❸ `data Res σ α = Res (Exn [σ → α])`
- ❹ `data Res σ α = Res [σ → Exn α]`

Sind das alle, fehlen noch welche, und wenn ja wieviele?

Nachtisch

Übung 11.4: Bonusfrage

Wir hatten also als Kombinationen

- ❶ `data Res σ α = Res (σ → [Exn α])`
- ❷ `data Res σ α = Res (σ → Exn [α])`
- ❸ `data Res σ α = Res (Exn [σ → α])`
- ❹ `data Res σ α = Res [σ → Exn α]`

Sind das alle, fehlen noch welche, und wenn ja wieviele?

Lösung: Es fehlen noch

- ❺ `data Res σ α = Res [Exn (σ → α)]`
- ❻ `data Res σ α = Res (Exn (σ → [α]))`

Res: Monadeninstanz

- Res α ist Reader (List (Exn α))
- Functor durch Komposition der fmap:

```
instance Functor (Res σ) where
  fmap f (Res g) = Res $ fmap (fmap f). g
```

- Monad durch Kombination der jeweiligen Operationen return und $\gg=$:

```
instance Monad (Res σ) where
  return a = Res (const [Right a])
  Res f  $\gg=$  g = Res $ λs → do ma ← f s
                                case ma of
                                  Right a → run (g a) s
                                  Left e → return (Left e)
```

Res: Operationen

- ▶ Zugriff auf den Zustand:

```
get  :: ( $\sigma \rightarrow \text{Exn } \alpha$ )  $\rightarrow$  Res  $\sigma$   $\alpha$ 
get f = Res $  $\lambda s \rightarrow [f\ s]$ 
```

- ▶ Fehler:

```
fail  :: String  $\rightarrow$  Res  $\sigma$   $\alpha$ 
fail msg = Res $ const [Left msg]
```

- ▶ Mehrdeutige Ergebnisse:

```
join  ::  $\alpha \rightarrow \alpha \rightarrow$  Res  $\sigma$   $\alpha$ 
join a b = Res $  $\lambda s \rightarrow [\text{Right } a, \text{ Right } b]$ 
```

Auswertung mit Allem

- Im Monaden `Res` können alle Effekte benutzt werden:

```
type State = M.Map String Double

eval :: Expr → Res State Double
eval (Var i)  = get (λs→ case M.lookup i s of
                           Just x→ return x
                           Nothing→ Left $ "No_such_variable"++ i)
eval (Num n)  = return n
eval (Plus a b) = do x← eval a; y← eval b; return $ x+ y
eval (Minus a b) = do x← eval a; y← eval b; return $ x- y
eval (Times a b) = do x← eval a; y← eval b; return $ x* y
eval (Div a b)   = do x← eval a; y← eval b
                      if y == 0 then fail "Divison_by_zero." else return $ x / y
eval (Pick a b)  = do x← eval a; y← eval b; join x y
```

- Systematische Kombination durch **Monadentransformer**

Zusammenfassung

- ▶ Monaden sind **Muster** für **Berechnungen** mit **Seiteneffekten**
- ▶ Beispiele:
 - ▶ Zustandstransformer ([State](#))
 - ▶ Fehler und Ausnahmen ([Maybe](#), [Either](#))
 - ▶ Nichtdeterminismus ([List](#))
- ▶ Fallbeispiel Auswertung von Ausdrücken:
 - ▶ Kombination aus Zustand, Partialität, Mehrdeutigkeit
 - ▶ Grenze: Nebenläufigkeit

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 12 vom 01.02.2021: Anwendungsbeispiel: Funktionale Webanwendungen

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

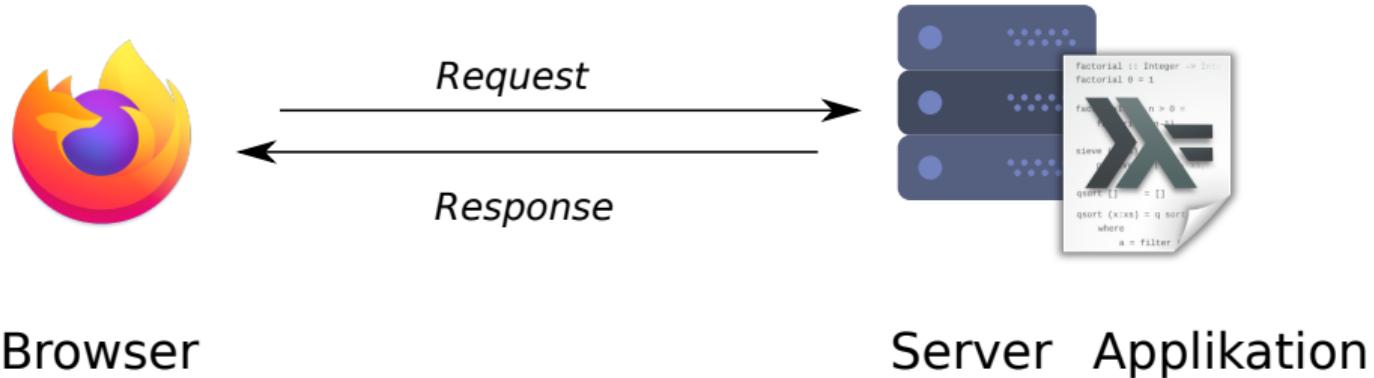
Wintersemester 2020/21

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.

Architekturerwägungen

- ▶ 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`.

- ▶ 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 konvertierten.

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 veränderliche Variablen.
- ▶ Kann **leer** oder **gefüllt** sein.

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

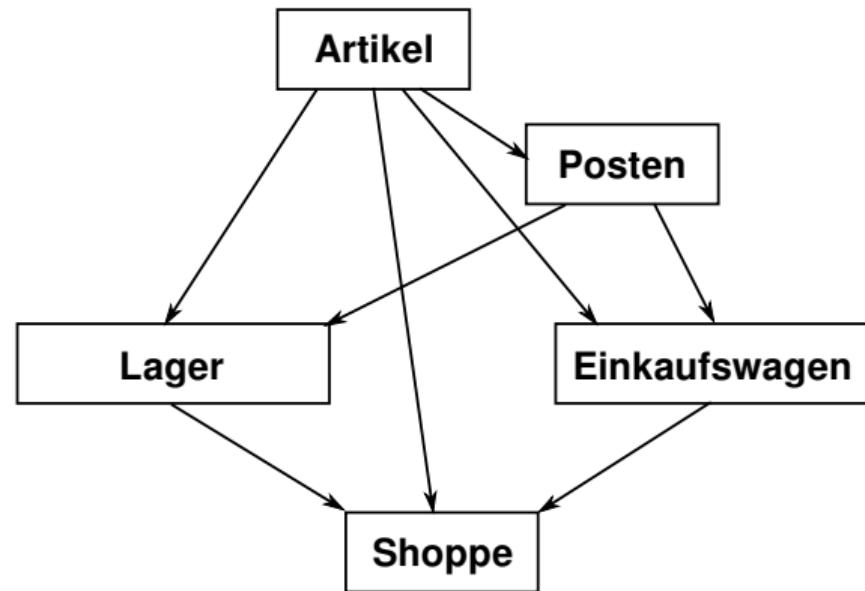
- ▶ Das **Model** ist der eigentliche (und persistente) Teil der Anwendung, bestehend aus den Datentypen samt der Funktionen darauf.
- ▶ Die **Views** sind Funktionen, die Webseiten aufbauen.
- ▶ Der **Controller** übersetzt Anfragen von außen in die Aufrufe der Model-Funktionen, erzeugt aus den Ergebnissen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

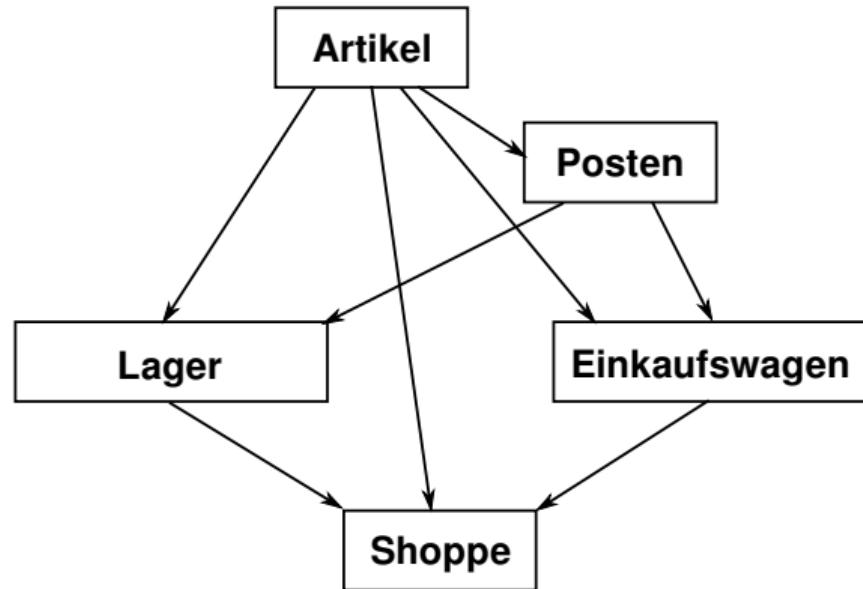
Model: der Shop

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



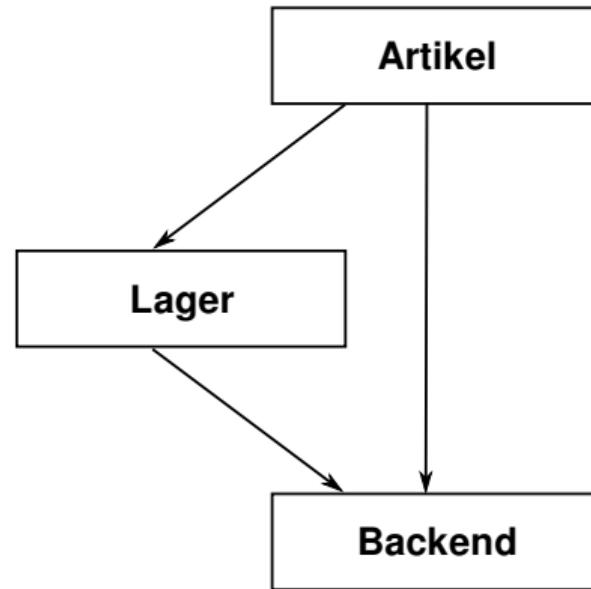
Model: der Shop

- ▶ 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**.

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 13 vom 08.02.21: Scala — Eine praktische Einführung

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität Bremen

Wintersemester 2020/21

Organisatorisches

- ▶ Prüfungssituation: dynamisch
- ▶ Nächste Vorlesung: **synchron** (nicht aufgezeichnet) am

15.02.2021 um 12:00

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

Heute: Scala

- ▶ A **scalable** **language**
- ▶ Rein objektorientiert
- ▶ Funktional
- ▶ Eine “JVM-Sprache”
- ▶ Seit 2004 von Martin Odersky, EPFL Lausanne (<http://www.scala-lang.org/>).
- ▶ Seit 2011 kommerziell durch Lightbend Inc. (formerly Typesafe)

I. Scala am Beispiel

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  return b
}

def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd (y, x % y)
```

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

► Variablen, veränderlich (**var**)

```
def gcdLoop(x: Long, y: Long): Long = {  
    var a = x  
    var b = y  
    while (a != 0) {  
        val temp = a  
        a = b % a  
        b = temp  
    }  
    return b  
}  
  
def gcd(x: Long, y: Long): Long =  
    if (y == 0) x else gcd (y, x % y)
```

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {  
    var a = x  
    var b = y  
    while (a != 0) {  
        val temp = a  
        a = b % a  
        b = temp  
    }  
    return b  
}  
  
def gcd(x: Long, y: Long): Long =  
    if (y == 0) x else gcd (y, x % y)
```

- ▶ Variablen, veränderlich (`var`)
- ▶ *Mit Vorsicht benutzen!*

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {  
    var a = x  
    var b = y  
    while (a != 0) {  
        val temp = a  
        a = b % a  
        b = temp  
    }  
    return b  
}  
  
def gcd(x: Long, y: Long): Long =  
    if (y == 0) x else gcd (y, x % y)
```

- ▶ Variablen, veränderlich (`var`)
 - ▶ *Mit Vorsicht benutzen!*
- ▶ Werte, unveränderlich (`val`)

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {  
    var a = x  
    var b = y  
    while (a != 0) {  
        val temp = a  
        a = b % a  
        b = temp  
    }  
    return b  
}  
  
def gcd(x: Long, y: Long): Long =  
    if (y == 0) x else gcd (y, x % y)
```

- ▶ Variablen, veränderlich (`var`)
 - ▶ *Mit Vorsicht benutzen!*
- ▶ Werte, unveränderlich (`val`)
- ▶ `while`-Schleifen

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {  
    var a = x  
    var b = y  
    while (a != 0) {  
        val temp = a  
        a = b % a  
        b = temp  
    }  
    return b  
}  
  
def gcd(x: Long, y: Long): Long =  
    if (y == 0) x else gcd (y, x % y)
```

- ▶ Variablen, veränderlich (`var`)
 - ▶ *Mit Vorsicht benutzen!*
- ▶ Werte, unveränderlich (`val`)
- ▶ `while`-Schleifen
 - ▶ *Unnötig!*

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {  
    var a = x  
    var b = y  
    while (a != 0) {  
        val temp = a  
        a = b % a  
        b = temp  
    }  
    return b  
}  
  
def gcd(x: Long, y: Long): Long =  
    if (y == 0) x else gcd (y, x % y)
```

- ▶ Variablen, veränderlich (`var`)
 - ▶ *Mit Vorsicht benutzen!*
- ▶ Werte, unveränderlich (`val`)
- ▶ `while`-Schleifen
 - ▶ *Unnötig!*
- ▶ Rekursion
 - ▶ Endrekursion wird optimiert

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {  
    var a = x  
    var b = y  
    while (a != 0) {  
        val temp = a  
        a = b % a  
        b = temp  
    }  
    return b  
}  
  
def gcd(x: Long, y: Long): Long =  
    if (y == 0) x else gcd (y, x % y)
```

- ▶ Variablen, veränderlich (`var`)
 - ▶ *Mit Vorsicht benutzen!*
- ▶ Werte, unveränderlich (`val`)
- ▶ `while`-Schleifen
 - ▶ *Unnötig!*
- ▶ Rekursion
 - ▶ Endrekursion wird optimiert
- ▶ Typinferenz
 - ▶ Mehr als Java, weniger als Haskell

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {  
    var a = x  
    var b = y  
    while (a != 0) {  
        val temp = a  
        a = b % a  
        b = temp  
    }  
    return b  
}  
  
def gcd(x: Long, y: Long): Long =  
    if (y == 0) x else gcd (y, x % y)
```

- ▶ Variablen, veränderlich (`var`)
 - ▶ *Mit Vorsicht benutzen!*
- ▶ Werte, unveränderlich (`val`)
- ▶ `while`-Schleifen
 - ▶ *Unnötig!*
- ▶ Rekursion
 - ▶ Endrekursion wird optimiert
- ▶ Typinferenz
 - ▶ Mehr als Java, weniger als Haskell
- ▶ Interaktive Auswertung

Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {  
  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    override def toString = numer + "/" + denom  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {  
  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    override def toString = numer + "/" + denom  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

► Klassenparameter

Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {  
  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    override def toString = numer + "/" + denom  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (**this**)

Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {  
  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    override def toString = numer + "/" + denom  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (**this**)
- ▶ Klassenvorbedingungen (**require**)

Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {  
  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    override def toString = numer + "/" + denom  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (`this`)
- ▶ Klassenvorbedingungen (`require`)
- ▶ private Werte und Methoden

Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {  
  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    override def toString = numer + "/" + denom  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (`this`)
- ▶ Klassenvorbedingungen (`require`)
- ▶ private Werte und Methoden
- ▶ Methoden, Syntax für Methodenanwendung

Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {  
  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    override def toString = numer + "/" + denom  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (**this**)
- ▶ Klassenvorbedingungen (**require**)
- ▶ private Werte und Methoden
- ▶ Methoden, Syntax für Methodenanwendung
- ▶ **override** (nicht optional)

Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {  
  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    override def toString = numer + "/" + denom  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (**this**)
- ▶ Klassenvorbedingungen (**require**)
- ▶ private Werte und Methoden
- ▶ Methoden, Syntax für Methodenanwendung

- ▶ **override** (nicht optional)
- ▶ Overloading

Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {  
  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    override def toString = numer + "/" + denom  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (**this**)
- ▶ Klassenvorbedingungen (**require**)
- ▶ private Werte und Methoden
- ▶ Methoden, Syntax für Methodenanwendung

- ▶ **override** (nicht optional)
- ▶ Overloading
- ▶ Operatoren

Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {  
  
    require(d != 0)  
  
    private val g = gcd(n.abs, d.abs)  
    val numer = n / g  
    val denom = d / g  
  
    def this(n: Int) = this(n, 1)  
  
    def add(that: Rational): Rational =  
        new Rational(  
            numer * that.denom + that.numer * denom,  
            denom * that.denom  
        )  
  
    override def toString = numer + "/" + denom  
  
    private def gcd(a: Int, b: Int): Int =  
        if (b == 0) a else gcd(b, a % b)  
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (**this**)
- ▶ Klassenvorbedingungen (**require**)
- ▶ private Werte und Methoden
- ▶ Methoden, Syntax für Methodenanwendung

- ▶ **override** (nicht optional)
- ▶ Overloading
- ▶ Operatoren
- ▶ Singleton objects (**object**)

Your Turn

Übung 13.1: Scala die Erste

Ladet Euch die Quellen für die Vorlesung von unserer Webseite, und den Scala-Interpreter und Compiler von

<https://www.scala-lang.org/>

herunter. Startet den Interpreter (scala), ladet das Beispiel oben mit

```
scala> :load 02-Rational-2.scala
```

Was passiert, wenn ihr ein Rational-Objekt konstruiert, bei dem die Vorbedingung verletzt ist?

Your Turn

Übung 13.1: Scala die Erste

Ladet Euch die Quellen für die Vorlesung von unserer Webseite, und den Scala-Interpreter und Compiler von

<https://www.scala-lang.org/>

herunter. Startet den Interpreter (scala), ladet das Beispiel oben mit

```
scala> :load 02-Rational-2.scala
```

Was passiert, wenn ihr ein Rational-Objekt konstruiert, bei dem die Vorbedingung verletzt ist?

Lösung: Es gibt (wenig überraschend) eine Exception:

```
scala> new Rational(6,0)
java.lang.IllegalArgumentException: requirement failed
  at scala.Predef$.require(Predef.scala:327)
  ... 29 elided
```

II. Das Typsystem

Algebraische Datentypen: 03-Expr.scala

Was sehen wir hier?

```
abstract class Expr
case class Var(name: String) extends Expr
case class Num (num: Double) extends Expr
case class Plus (left: Expr, right: Expr) extends Expr
case class Minus (left: Expr, right: Expr) extends Expr
case class Times (left: Expr, right: Expr) extends Expr
case class Div (left: Expr, right: Expr) extends Expr
```

```
// Evaluating an expression
def eval(expr: Expr): Double = expr match {
  case v: Var ⇒ 0      // Variables evaluate to 0
  case Num(x) ⇒ x
  case Plus(e1, e2) ⇒ eval(e1) + eval(e2)
  case Minus(e1, e2) ⇒ eval(e1) - eval(e2)
  case Times(e1, e2) ⇒ eval(e1) * eval(e2)
  case Div(e1, e2) ⇒ eval(e1) / eval(e2)
}

val e = Times(Num(12), Plus(Num(2.3),Num(3.7)))
```

Algebraische Datentypen: 03-Expr.scala

Was sehen wir hier?

```
abstract class Expr
case class Var(name: String) extends Expr
case class Num (num: Double) extends Expr
case class Plus (left: Expr, right: Expr) extends Expr
case class Minus (left: Expr, right: Expr) extends Expr
case class Times (left: Expr, right: Expr) extends Expr
case class Div (left: Expr, right: Expr) extends Expr
```

```
// Evaluating an expression
def eval(expr: Expr): Double = expr match {
  case v: Var => 0      // Variables evaluate to 0
  case Num(x) => x
  case Plus(e1, e2) => eval(e1) + eval(e2)
  case Minus(e1, e2) => eval(e1) - eval(e2)
  case Times(e1, e2) => eval(e1) * eval(e2)
  case Div(e1, e2) => eval(e1) / eval(e2)
}

val e = Times(Num(12), Plus(Num(2.3),Num(3.7)))
```

- ▶ **case class** erzeugt
 - ▶ Factory-Methode für Konstruktoren
 - ▶ Parameter als implizite **val**
 - ▶ abgeleitete Implementierung für **toString**, **equals**
 - ▶ ... und pattern matching (**match**)
- ▶ Pattern sind
 - ▶ **case 4** ⇒ Literale
 - ▶ **case C(4)** ⇒ Konstruktoren
 - ▶ **case C(x)** ⇒ Variablen
 - ▶ **case C(_)** ⇒ Wildcards
 - ▶ **case x: C** ⇒ getypte pattern
 - ▶ **case C(D(x: T, y), 4)** ⇒ geschachtelt

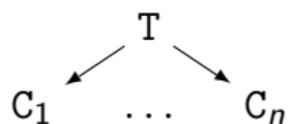
Implementierung algebraischer Datentypen

Haskell:

```
data T = C1 | ... | Cn
```

- ▶ Ein Typ T
- ▶ Konstruktoren erzeugen Datentyp

Scala:



- ▶ Varianten als **Subtypen**
- ▶ Problem und Vorteil:

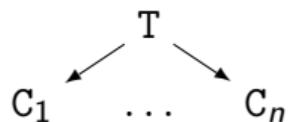
Implementierung algebraischer Datentypen

Haskell:

```
data T = C1 | ... | Cn
```

- ▶ Ein Typ T
- ▶ Konstruktoren erzeugen Datentyp

Scala:



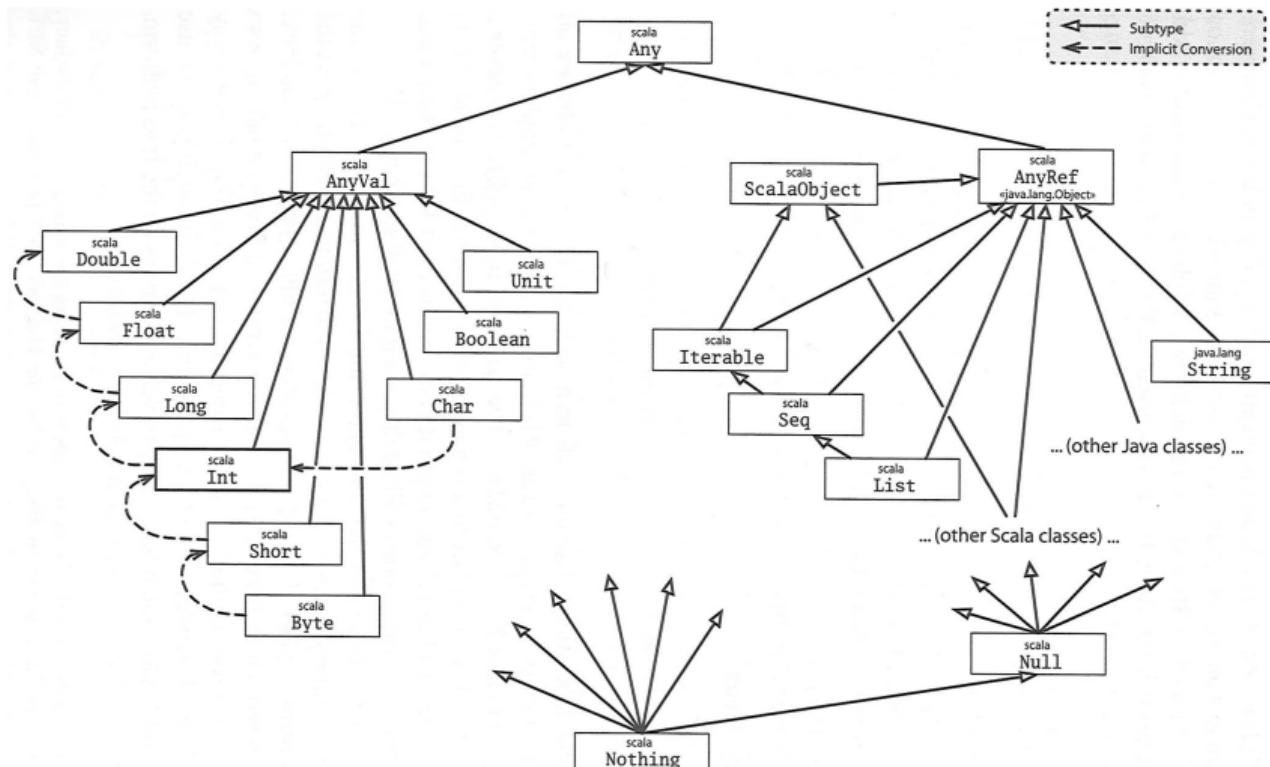
- ▶ Varianten als **Subtypen**
- ▶ Problem und Vorteil: **Erweiterbarkeit**
- ▶ **sealed** verhindert Erweiterung

Das Typsystem

Das Typsystem behebt mehrere Probleme von Java:

- ▶ Werte vs. Objekte
- ▶ Scala vs. Java
- ▶ NULL references

Vererbungshierarchie



Quelle: Odersky, Spoon, Venners: *Programming in Scala*

Your Turn

Übung 13.2: Scala die Zweite

Öffnen Sie die Datei 02-Expr-fold.scala, und vervollständigen Sie die Definition der fold-Funktion für den Datentyp Expr.

Your Turn

Übung 13.2: Scala die Zweite

Öffnen Sie die Datei 02-Expr-fold.scala, und vervollständigen Sie die Definition der fold-Funktion für den Datentyp Expr.

Lösung:

```
def fold[A]( v: String ⇒ A
            , n: Double ⇒ A
            , p: (A, A) ⇒ A
            , m: (A, A) ⇒ A
            , t: (A, A) ⇒ A
            , d: (A, A) ⇒ A): A = this match
{
  case Var(variable) ⇒ v(variable)
  case Num(num) ⇒ n(num)
  case Plus(e1, e2) ⇒ p( e1.fold(v, n, p, m, t, d)
                         , e1.fold(v, n, p, m, t, d))
  case Minus(e1, e2) ⇒ m( e1.fold(v, n, p, m, t, d)
                         , e1.fold(v, n, p, m, t, d))
  case Times(e1, e2) ⇒ t( e1.fold(v, n, p, m, t, d)
                         , e1.fold(v, n, p, m, t, d))
  case Div(e1, e2) ⇒ d( e1.fold(v, n, p, m, t, d)
                         , e1.fold(v, n, p, m, t, d))
}
```

III. Polymorphie und Vererbung

Parametrische Polymorphie

- ▶ Typparameter (wie in Haskell, Generics in Java), Bsp. `List[T]`
- ▶ Problem: Vererbung und Polymorphie
- ▶ Ziel: wenn $S < T$, dann `List[S] < List[T]`
- ▶ **Does not work** — 04-Ref.hs

Parametrische Polymorphie

- ▶ Typparameter (wie in Haskell, Generics in Java), Bsp. `List[T]`
- ▶ Problem: Vererbung und Polymorphie
- ▶ Ziel: wenn $S < T$, dann `List[S] < List[T]`
- ▶ **Does not work** — 04-Ref.hs
- ▶ Warum?
 - ▶ Funktionsraum nicht monoton im ersten Argument
 - ▶ Sei $X \subseteq Y$, dann $Z \rightarrow X \subseteq Z \rightarrow Y$, aber $X \rightarrow Z \not\subseteq Y \rightarrow Z$
 - ▶ Sondern $Y \rightarrow Z \subseteq X \rightarrow Z$

Typvarianz

class C[+T]

- ▶ **Kovariant**
- ▶ Wenn $S < T$, dann $C[S] < C[T]$
- ▶ Parameter T nur im Wertebereich von Methoden

class C[T]

- ▶ **Rigide**
- ▶ Kein Subtyping
- ▶ Parameter T kann beliebig verwendet werden

class C[-T]

- ▶ **Kontravariant**
- ▶ Wenn $S < T$, dann $C[T] < C[S]$
- ▶ Parameter T nur im Definitionsbereich von Methoden

Your Turn

Übung 13.3: Scala die Dritte

Betrachten Sie folgendes Beispiel:

```
class Function[S, T] {  
  def apply(x:S) : T  
}
```

Wie müssen hier die Varianz-Annotation für die Typvariablen S und T lauten?

Your Turn

Übung 13.3: Scala die Dritte

Betrachten Sie folgendes Beispiel:

```
class Function[S, T] {  
  def apply(x:S) : T  
}
```

Wie müssen hier die Varianz-Annotation für die Typvariablen S und T lauten?

Lösung:

```
class Function[-S, +T] {  
  def apply(x:S) : T  
}
```

IV. Strukturierung mit Traits

Traits: 05-Funny.scala

Was sehen wir hier?

- ▶ Trait (Mix-ins): abstrakte Klassen, Interfaces; Haskell: Typklassen
- ▶ „Abstrakte Klassen ohne Oberklasse“
- ▶ Unterschied zu Klassen:
 - ▶ Mehrfachvererbung möglich
 - ▶ Keine feste Oberklasse (`super` dynamisch gebunden)
 - ▶ Nützlich zur Strukturierung (Aspektorientierung)
- ▶ Nützlich zur Strukturierung:

thin interface + trait = rich interface

Beispiel: 05-Ordered.scala, 05-Rational.scala

Was wir ausgelassen haben...

- ▶ **Komprehension** (nicht nur für Listen)
- ▶ **Gleichheit**: == (final), equals (nicht final), eq (Referenzen)
- ▶ *string interpolation*
- ▶ **Implizite** Parameter und Typkonversionen
- ▶ **Nebenläufigkeit** (Aktoren, Futures)
- ▶ Typsichere **Metaprogrammierung**
- ▶ Das *simple build tool* sbt
- ▶ Der JavaScript-Compiler `scala.js`

Schlamschlacht der Programmiersprachen

	Haskell	Scala	Java
Klassen und Objekte	-	+	+
Funktionen höherer Ordnung	+	+	-
Typinferenz	+	(+)	-
Parametrische Polymorphie	+	+	+
Ad-hoc-Polymorphie	+	+	-
Typsichere Metaprogrammierung	+	+	-

Alle: Nebenläufigkeit, Garbage Collection, FFI

- ▶ Objekt-orientiert:
 - ▶ Veränderlicher, gekapselter **Zustand**
 - ▶ **Subtypen** und Vererbung
 - ▶ **Klassen** und **Objekte**
- ▶ Funktional:
 - ▶ Unveränderliche **Werte**
 - ▶ Parametrische und Ad-hoc **Polymorphie**
 - ▶ Funktionen höherer Ordnung
 - ▶ Hindley-Milner **Typinferenz**

Beurteilung

► **Vorteile:**

- Funktional programmieren, in der Java-Welt leben
- Gelungene Integration funktionaler und OO-Konzepte
- Sauberer Sprachentwurf, effiziente Implementierung, reiche Büchereien

► **Nachteile:**

- Manchmal etwas **zu** viel
 - Entwickelt sich ständig weiter
 - One-Compiler-Language, vergleichsweise langsam
- ## ► Mehr Scala?
- Besuchen Sie auch die Veranstaltung **Reaktive Programmierung** (soweit verfügbar)

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 14 vom 15.02.21: Rückblick & Ausblick

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. Prüfungen

Inhalt

- ▶ Warum **Fachgespräche**?
- ▶ Was bedeutet das für die **Scheinbedingungen**?
- ▶ **Ablauf** der Fachgespräche
- ▶ **Termin** für die Fachgespräche
- ▶ **Beispiele**

Warum Fachgespräche?

- ▶ Fachgespräche dienen dazu, „Individualität der Leistung“ sicherzustellen.
 - ▶ Besonders in Corona-Zeiten.
- ▶ Format: Eine Gruppe (3 Studierende), 15 Minuten
 - ▶ Individuelle Prüfungen (30 Minuten) zeitlich nicht darstellbar (127 (Pabo) – 190 (stud.ip))
 - ▶ Open-Book Klausur o.ä. keine Alternative, da keine wesentliche Verbesserung zu Übungsbetrieb
 - ▶ Parallelisierung erlaubt höheren Durchsatz und längere Antwortzeit
- ▶ Gruppe muss keine Übungsgruppe sein

Scheinkriterien

- ▶ Fachgespräch am Ende (Individualität der Leistung)
- ▶ Mind. 50% in den Einzelübungsblättern und mind. 50% in allen Übungsblättern
- ▶ **Vornote** aus den Übungsblättern
- ▶ **Notenspiegel** (in Prozent aller Punkte):

Pkt.%	Note	Pkt.%	Note	Pkt.%	Note	Pkt.%	Note
≥ 95	1.0	89.5-85	1.7	74.5-70	2.7	59.5-55	3.7
94.5-90	1.3	84.5-80	2.0	69.5-65	3.0	54.5-50	4.0

- ▶ Fachgespräch **bestätigt** Vornote (Abänderung bis max. 1 Notenstufe)

Ablauf des Fachgeprächs

- ▶ Fachgespräche finden über **Zoom** statt
- ▶ Fachgespräche bestehen aus der schriftlichen Bearbeitung einer kurzen **Programmieraufgabe** sowie kurzen **Verständnisfragen** dazu
- ▶ Aufgaben werden online auf HedgeDoc bearbeitet.
 - ▶ Kein Syntaxcheck, kein Compiler
 - ▶ Beispielfragen auf der Webseite

Ablauf und Ziel des Fachgespräches

- ▶ Kamera ist **Pflicht** (**eingeschaltet** lassen)
- ▶ **Ruhiger** Ort ohne Hintergrundgeräusche
- ▶ Niemand **sonst** im Raum
- ▶ Bearbeitung **live** im HedgeDoc, nicht aus Editor kopieren
- ▶ Der Weg ist das Ziel...
- ▶ **Nicht**: Gelöste Aufgabe.
- ▶ **Sondern**: wie löst ihr die Aufgabe?

Termine

- ▶ Fachgespräche finden an **drei** Tagen statt.
- ▶ Mögliche **Termine**: 11/12.03, 18/19.03., 25/26.03.
- ▶ Dazu **Umfrage**.
- ▶ Anmeldung wird **nach der Vorlesung** freigeschaltet.

Beispielfrage (leicht)

Definieren Sie eine Funktion `format`, die eine Zahl in einer Zeichenkette gegebener Länge rechtsbündig ausgibt.

Bsp. `format 4 35` \rightsquigarrow " 35"

Lösung:

```
format :: Int → Int → String
```

Beispielfrage (leicht)

Definieren Sie eine Funktion `format`, die eine Zahl in einer Zeichenkette gegebener Länge rechtsbündig ausgibt.

Bsp. `format 4 35` \rightsquigarrow "35"

Lösung:

```
format :: Int → Int → String
format n x =
  replicate ' ' (n - length s) ++ s
  where s = show x
```

Beispielfrage (leicht)

Definieren Sie eine Funktion `format`, die eine Zahl in einer Zeichenkette gegebener Länge rechtsbündig ausgibt.

Bsp. `format 4 35` \rightsquigarrow "35"

Lösung:

```
format :: Int → Int → String
format n x =
  replicate ' ' (n - length s) ++ s
  where s = show x
```

Fragen:

- ▶ Was ist daran falsch?

Beispielfrage (leicht)

Definieren Sie eine Funktion `format`, die eine Zahl in einer Zeichenkette gegebener Länge rechtsbündig ausgibt.

Bsp. `format 4 35` \rightsquigarrow "35"

Lösung:

```
format :: Int → Int → String  
  
format n x =  
  replicate ' ' (n - length s) ++ s  
  where s = show x
```

Fragen:

- ▶ Was ist daran falsch?
- ▶ Funktioniert das auch für negative Zahlen?

Beispielfrage (leicht)

Definieren Sie eine Funktion `format`, die eine Zahl in einer Zeichenkette gegebener Länge rechtsbündig ausgibt.

Bsp. `format 4 35` \rightsquigarrow " 35"

Lösung:

```
format :: Int → Int → String
format n x =
  replicate ' ' (n - length s) ++ s
  where s = show x
```

Fragen:

- ▶ Was ist daran falsch?
- ▶ Funktioniert das auch für negative Zahlen?
- ▶ Wie kann ich Länge auf `n` begrenzen?

Beispielfrage (mittel)

Definieren Sie eine Funktion `mean`, die den arithmetischen Durchschnitt einer Liste von ganzen Zahlen berechnet.

Bsp. `mean [2,1,5,4,3]` $\rightsquigarrow 3.0$

Lösung:

```
mean :: [Int] → Double
```

Beispielfrage (mittel)

Definieren Sie eine Funktion `mean`, die den arithmetischen Durchschnitt einer Liste von ganzen Zahlen berechnet.

Bsp. `mean [2,1,5,4,3]` $\rightsquigarrow 3.0$

Lösung:

```
mean :: [Int] → Double
```

```
mean xs = sum xs / length xs
```

Beispielfrage (mittel)

Definieren Sie eine Funktion `mean`, die den arithmetischen Durchschnitt einer Liste von ganzen Zahlen berechnet.

Bsp. `mean [2,1,5,4,3] ~> 3.0`

Lösung:

```
mean :: [Int] → Double
```

```
mean xs = sum xs / length xs
```

Fragen:

- ▶ Was ist daran falsch?

Beispielfrage (mittel)

Definieren Sie eine Funktion `mean`, die den arithmetischen Durchschnitt einer Liste von ganzen Zahlen berechnet.

Bsp. `mean [2,1,5,4,3] ~> 3.0`

Lösung:

```
mean :: [Int] → Double
```

```
mean xs = sum xs / length xs
```

Fragen:

- ▶ Was ist daran falsch?
- ▶ Wie konvertiert man `Int` nach `Double`?

Beispielfrage (mittel)

Definieren Sie eine Funktion `mean`, die den arithmetischen Durchschnitt einer Liste von ganzen Zahlen berechnet.

Bsp. `mean [2,1,5,4,3] ~> 3.0`

Lösung:

```
mean :: [Int] → Double
```

```
mean xs = sum xs / length xs
```

Fragen:

- ▶ Was ist daran falsch?
 - ▶ Wie konvertiert man `Int` nach `Double`?
-
- ```
mean xs = fromIntegral (sum xs) / ...
```
- ▶ Schwer: wie vereinfacht man  
`fromIntegral (length xs)`

# Beispielfrage (schwer)

Zwei Int-Listen sollen **ähnlich** heißen, wenn Sie die gleichen Zahlen unabhängig von ihrer Reihenfolge und Häufigkeit enthalten. Schreiben Sie eine Testfunktion `sim` dafür.

*Bsp.*

```
sim [3,2,2,1,3] [1,2,3] ~> True
```

```
sim :: [Int] → [Int] → Bool
```

# Beispielfrage (schwer)

Zwei Int-Listen sollen **ähnlich** heißen, wenn Sie die gleichen Zahlen unabhängig von ihrer Reihenfolge und Häufigkeit enthalten. Schreiben Sie eine Testfunktion `sim` dafür.

*Bsp.*

```
sim [3,2,2,1,3] [1,2,3] ~> True
```

```
sim :: [Int] → [Int] → Bool
```

```
sim xs ys = all (λx → elem x ys) xs
```

# Beispielfrage (schwer)

Zwei Int-Listen sollen **ähnlich** heißen, wenn Sie die gleichen Zahlen unabhängig von ihrer Reihenfolge und Häufigkeit enthalten. Schreiben Sie eine Testfunktion `sim` dafür.

*Bsp.*

```
sim [3,2,2,1,3] [1,2,3] ~> True
```

```
sim :: [Int] → [Int] → Bool
```

```
sim xs ys = all (λx → elem x ys) xs
```

Fragen:

- ▶ Was fehlt da?

# Beispielfrage (schwer)

Zwei Int-Listen sollen **ähnlich** heißen, wenn Sie die gleichen Zahlen unabhängig von ihrer Reihenfolge und Häufigkeit enthalten. Schreiben Sie eine Testfunktion `sim` dafür.

*Bsp.*

```
sim [3,2,2,1,3] [1,2,3] ~> True
```

```
sim :: [Int] → [Int] → Bool
```

```
sim xs ys = all (λx → elem x ys) xs
```

Fragen:

- ▶ Was fehlt da?

```
sim xs ys =
```

```
.. && all (λy → elem y xs) ys
```

- ▶ Kann man die Signatur verallgemeinern?

# Beispielfrage (schwer)

Zwei Int-Listen sollen **ähnlich** heißen, wenn Sie die gleichen Zahlen unabhängig von ihrer Reihenfolge und Häufigkeit enthalten. Schreiben Sie eine Testfunktion `sim` dafür.

*Bsp.*

```
sim [3,2,2,1,3] [1,2,3] ~> True
```

```
sim :: [Int] → [Int] → Bool
```

```
sim xs ys = all (λx → elem x ys) xs
```

Fragen:

- ▶ Was fehlt da?

```
sim xs ys =
```

```
.. && all (λy → elem y xs) ys
```

- ▶ Kann man die Signatur verallgemeinern?

```
sim :: Eq α ⇒ [α] → [α] → Bool
```

# Mündliche Prüfung

- ▶ **Dauer:** in der Regel 20-30 Minuten
- ▶ **Einzelprüfung**, ggf. mit Beisitzer
- ▶ **Inhalt:** Programmieren mit Haskell und Vorlesungsstoff
- ▶ **Ablauf:** “Fachgespräch plus”
  - ▶ Einstieg mit leichter Programmieraufgabe wie im Fachgespräch
  - ▶ Daran anschließend Fragen über den Stoff
- ▶ Liste von **Verständnisfragen** auf der Webseite.

## II. Rückblick und Ausblick

# Warum funktionale Programmierung lernen?

- ▶ Funktionale Programmierung macht aus Programmierern Informatiker
- ▶ Blick über den Tellerrand — was kommt in 10 Jahren?
- ▶ **Herausforderungen** der Zukunft
- ▶ Enthält die **wesentlichen** Elemente moderner Programmierung

# Zusammenfassung Haskell

## Stärken:

- ▶ Abstraktion durch
  - ▶ Polymorphie und Typsystem
  - ▶ algebraische Datentypen
  - ▶ Funktionen höherer Ordnung
- ▶ Flexible Syntax
- ▶ Haskell als Meta-Sprache
- ▶ Ausgereifter Compiler
- ▶ Viele Büchereien

## Schwächen:

- ▶ Komplexität
- ▶ Büchereien
  - ▶ Nicht immer gut gepflegt
- ▶ Viel im Fluß
  - ▶ Kein stabiler und brauchbarer Standard
- ▶ Divergierende Ziele:
  - ▶ Forschungsplattform **und** nutzbares Werkzeug

# Andere Funktionale Sprachen

## ► **Standard ML** (SML):

- ▶ Streng typisiert, strikte Auswertung
- ▶ Standardisiert, formal definierte Semantik
- ▶ Drei aktiv unterstützte Compiler
- ▶ Verwendet in Theorembeweisern (Isabelle, HOL)
- ▶ <http://www.standardml.org/>

## ► **Caml, O'Caml**:

- ▶ Streng typisiert, strikte Auswertung
- ▶ Hocheffizienter Compiler, byte code & nativ
- ▶ Nur ein Compiler (O'Caml)
- ▶ <http://caml.inria.fr/>

# Andere Funktionale Sprachen

## ► **LISP** und **Scheme**

- Ungetypt/schwach getypt
- Seiteneffekte
- Viele effiziente Compiler, aber viele Dialekte
- Auch industriell verwendet

## ► **Hybridsprachen:**

- Scala (Functional-OO, JVM)
- F# (Functional-OO, .Net)
- Clojure (Lisp, JVM)

# Was spricht gegen funktionale Programmierung?

- ▶ Mangelnde Unterstützung:
  - ▶ Libraries, Dokumentation, Entwicklungsumgebungen
  - ▶ Wird besser (Scala)...
- ▶ Programmierung nur kleiner Teil der SW-Entwicklung
- ▶ Nicht verbreitet — funktionale Programmierer zu teuer
- ▶ Konservatives Management
  - ▶ “Nobody ever got fired for buying IBM”

# Was spricht gegen funktionale Programmierung?

- ▶ Mangelnde Unterstützung:
  - ▶ Libraries, Dokumentation, Entwicklungsumgebungen
  - ▶ Wird besser (Scala)...
- ▶ Programmierung nur kleiner Teil der SW-Entwicklung
- ▶ Nicht verbreitet — funktionale Programmierer zu teuer
- ▶ Konservatives Management
  - ▶ “Nobody ever got fired for buying SAP”

# Haskell in der Industrie

- ▶ Simon Marlow bei Facebook
- ▶ Simon Peyton-Jones bei Microsoft.
- ▶ secuCloud in Hamburg (<https://www.secucloud.com/>)
- ▶ Bluespec: Schaltkreisentwicklung, DSL auf Haskell-Basis
- ▶ Galois, Inc: Cryptography (Cryptol DSL)
- ▶ Finanzindustrie: Barclays Capital, Credit Suisse, Deutsche Bank
- ▶ Siehe auch: Haskell in Industry ([https://wiki.haskell.org/Haskell\\_in\\_industry](https://wiki.haskell.org/Haskell_in_industry))
- ▶ Andere Sprachen: Scala, Erlang, FL, ...

# Perspektiven funktionaler Programmierung

## ► **Forschung:**

- Ausdrucksstärkere Typsysteme
- für effiziente Implementierungen
- und eingebaute Korrektheit (Typ als Spezifikation)
- Parallelität?

## ► **Anwendungen:**

- Eingebettete domänenspezifische Sprachen
- Zustandsfreie Berechnungen (MapReduce, Hadoop, Spark)
- **Big Data** and **Cloud Computing**

## If you liked this course, you might also like ...

- ▶ Die Veranstaltung **Reaktive Programmierung** (findet irregulär stattt)
- ▶ Scala, nebenläufige Programmierung, fortgeschrittene Techniken der funktionalen Programmierung
- ▶ Wir suchen **studentische Hilfskräfte** am DFKI, FB CPS
  - ▶ Scala als Entwicklungssprache
- ▶ Wir suchen **Tutoren für PI3**
  - ▶ Im WS 2021/22 — **meldet Euch** bei Thomas Barkowsky (oder bei mir)!

Tschüß!

