Praktische Informatik 3: Funktionale Programmierung
Vorlesung 1 vom 16.10.2018: Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Personal

- **Vorlesung**:
  Christoph Lüth <cxl@informatik.uni-bremen.de>
  www.informatik.uni-bremen.de/~cxl/ (MZH 4186, Tel. 59830)

- **Tutoren**:

Thomas Barkoswky	<barkowsky@informatik.uni-bremen.de>
Andreas Kästner	<andreask@informatik.uni-bremen.de>
Gerrit Marquard	<terrig@math.uni-bremen.de>
Tobias Haslop	<haslop@uni-bremen.de>
Matz Habermann	<matz@uni-bremen.de>
Berthold Hoffmann	<hof@informatik.uni-bremen.de>

- **Webseite**:
  www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws18

Termine

- **Vorlesung**: Di 16 – 18 NW1 H 1 – H0020
- **Tutorien**:

Mi	08–10	MZH 1470	Thomas Barkowsky
	10–12	MZH 1090	Tobias Haslop
	12–14	MZH 1470	Matz Habermann
	16–18	MZH 1090	Andreas Kästner
Do	12–14	MZH 1090	Gerrit Marquardt

- **“Fragestunde”**: Berthold Hoffmann
- **Anmeldung** zu den Übungsgruppen über stud.ip (ab 18:00)
- Evtl. Zusatztutorium Do 16– 18.

Übungsbetrieb

- Ausgabe der Übungsblätter über die Webseite **Dienstag abend**
- 6+1 Einzelübungsblätter:
  - Besprechung und Bearbeitung der Übungsblätter in den Tutorien
  - Bearbeitungszeit bis Freitag **Freitag 12:00**
- 3 Gruppenübungsblätter (doppelt gewichtet)
  - Bearbeitungszeit bis **Freitag folgender Woche 12:00**
  - Übungsgruppen: max. **drei Teilnehmer**
- **Abgabe** elektronisch (eventuell zusätzlich in Papier)
- **Bewertung**: Quellcode, Tests, Dokumentation

Scheinkriterien

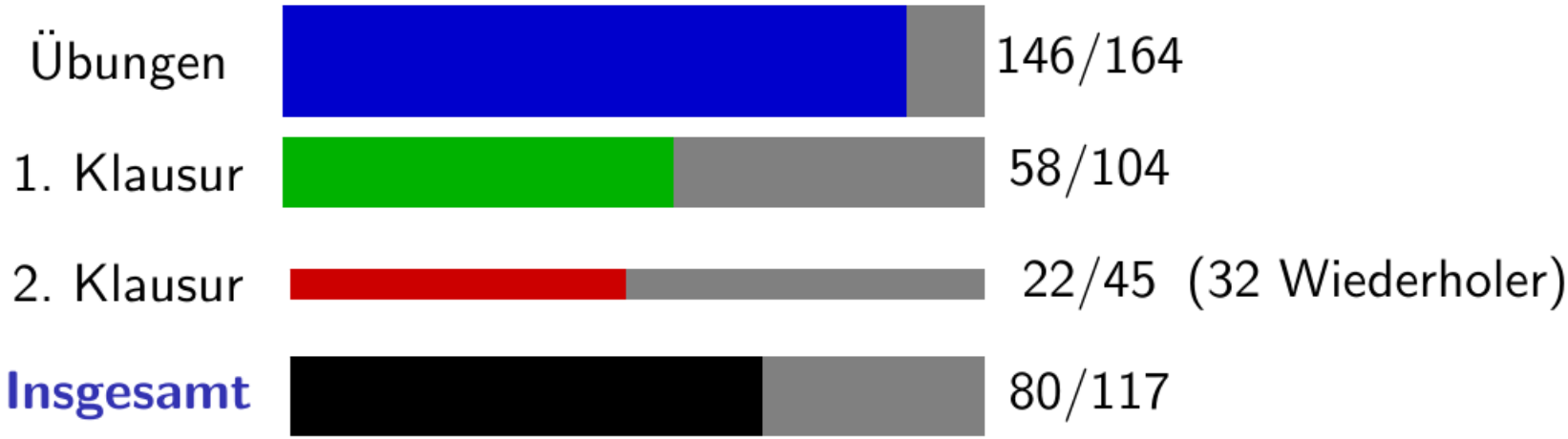
- Elektronische Klausur am Ende (Individualität der Leistung)
- Mind. 50% in allen Übungsblättern und mind. 50% in der E-Klausur
- Note = 50% Übungsblätter und 50% E-Klausur
- **Notenspiegel** (in Prozent aller Punkte):

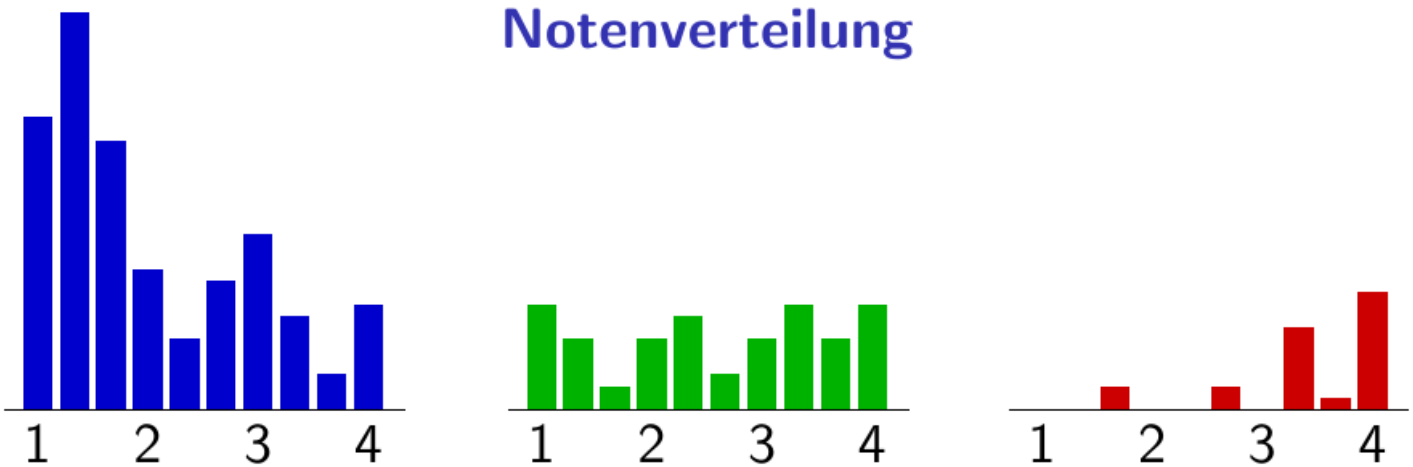
Pkt.%	Note	Pkt.%	Note	Pkt.%	Note	Pkt.%	Note
		89.5-85	1.7	74.5-70	2.7	59.5-55	3.7
≥ 95	1.0	84.5-80	2.0	69.5-65	3.0	54.5-50	4.0
94.5-90	1.3	79.5-75	2.3	64.5-60	3.3	49.5-0	**n/b**

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 mehrerer Gruppenmitglieder)
  - **Vorher** ankündigen, sonst **null** Punkte.

Statistik von PI3 im Wintersemester 17/18





Sprechstunde (“Frequently Asked Questions”)

- Ein **freiwilliges** Angebot

  Wer? Berthold Hoffmann
  <hof@informatik.uni-bremen.de>

  Wo? MZH 3250 (Büro)

  Wann? Nach Vereinbarung (per Email) oder Do 14–16

  Wozu? Überwindung von Anfangsschwierigkeiten

  - Funktionales Programmieren
  - Haskell

- **Besonders sinnvoll** in den ersten sechs Wochen

Fahrplan

- **Teil I: Funktionale Programmierung im Kleinen**
  - Einführung
  - Funktionen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Zyklische Datenstrukturen
  - Funktionen höherer Ordnung I
  - 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
- Enthält die **wesentlichen** Elemente moderner Programmierung

Zunkunft eingebaut

Funktionale Programmierung addressiert die
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

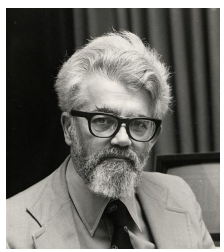
Geschichtliches: Die Anfänge

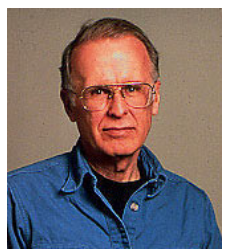
- **Grundlagen** 1920/30
  - Kombinatorlogik und λ-Kalkül (Schönfinkel, Curry, Church)
- Erste funktionale **Programmiersprachen** 1960
  - LISP (McCarthy), ISWIM (Landin)
- **Weitere** Programmiersprachen 1970– 80
  - FP (Backus); ML (Milner, Gordon); Hope (Burstall); Miranda (Turner)

Moses Schönfinkel

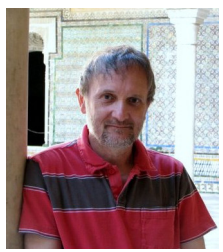
Haskell B. Curry

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`
- **Rein** funktional
  - Essenz der funktionalen Programmierung

Programme als Funktionen

- Programme als Funktionen:

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

- 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**:

$$\texttt{fac 2} \rightarrow \texttt{if 2 == 0 then 1 else 2* fac (2-1)}$$

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**:

$$\begin{aligned} \texttt{fac 2} &\rightarrow \texttt{if 2 == 0 then 1 else 2* fac (2-1)} \\ &\rightarrow \texttt{if False then 1 else 2* fac 1} \end{aligned}$$

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**:

$$
\begin{aligned}
\texttt{fac 2} &\rightarrow \texttt{if 2 == 0 then 1 else 2* fac (2-1)} \\
&\rightarrow \texttt{if False then 1 else 2* fac 1} \\
&\rightarrow \texttt{2* fac 1}
\end{aligned}
$$

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**:

$$
\begin{aligned}
\texttt{fac 2} &\rightarrow \texttt{if 2 == 0 then 1 else 2* fac (2-1)} \\
&\rightarrow \texttt{if False then 1 else 2* fac 1} \\
&\rightarrow \texttt{2* fac 1} \\
&\rightarrow \texttt{2* if 1 == 0 then 1 else 1* fac (1-1)}
\end{aligned}
$$

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**:

$$
\begin{aligned}
\texttt{fac 2} &\rightarrow \texttt{if 2 == 0 then 1 else 2* fac (2-1)} \\
&\rightarrow \texttt{if False then 1 else 2* fac 1} \\
&\rightarrow \texttt{2* fac 1} \\
&\rightarrow \texttt{2* if 1 == 0 then 1 else 1* fac (1-1)} \\
&\rightarrow \texttt{2* if False then 1 else 1* fac 0}
\end{aligned}
$$

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**:

$$
\begin{aligned}
\texttt{fac 2} &\rightarrow \texttt{if 2 == 0 then 1 else 2* fac (2-1)} \\
&\rightarrow \texttt{if False then 1 else 2* fac 1} \\
&\rightarrow \texttt{2* fac 1} \\
&\rightarrow \texttt{2* if 1 == 0 then 1 else 1* fac (1-1)} \\
&\rightarrow \texttt{2* if False then 1 else 1* fac 0} \\
&\rightarrow \texttt{2* 1* fac 0}
\end{aligned}
$$

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**:

$$
\begin{aligned}
\texttt{fac 2} &\rightarrow \texttt{if 2 == 0 then 1 else 2* fac (2-1)} \\
&\rightarrow \texttt{if False then 1 else 2* fac 1} \\
&\rightarrow \texttt{2* fac 1} \\
&\rightarrow \texttt{2* if 1 == 0 then 1 else 1* fac (1-1)} \\
&\rightarrow \texttt{2* if False then 1 else 1* fac 0} \\
&\rightarrow \texttt{2* 1* fac 0} \\
&\rightarrow \texttt{2* 1* if 0 == 0 then 1 else 0* fac (0-1)}
\end{aligned}
$$

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**:

$$
\begin{aligned}
\texttt{fac 2} &\rightarrow \texttt{if 2 == 0 then 1 else 2* fac (2-1)} \\
&\rightarrow \texttt{if False then 1 else 2* fac 1} \\
&\rightarrow \texttt{2* fac 1} \\
&\rightarrow \texttt{2* if 1 == 0 then 1 else 1* fac (1-1)} \\
&\rightarrow \texttt{2* if False then 1 else 1* fac 0} \\
&\rightarrow \texttt{2* 1* fac 0} \\
&\rightarrow \texttt{2* 1* if 0 == 0 then 1 else 0* fac (0-1)} \\
&\rightarrow \texttt{2* 1* if True then 1 else 0* fac (0-1)}
\end{aligned}
$$

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**:

$$
\begin{aligned}
\texttt{fac 2} &\rightarrow \texttt{if 2 == 0 then 1 else 2* fac (2-1)} \\
&\rightarrow \texttt{if False then 1 else 2* fac 1} \\
&\rightarrow \texttt{2* fac 1} \\
&\rightarrow \texttt{2* if 1 == 0 then 1 else 1* fac (1-1)} \\
&\rightarrow \texttt{2* if False then 1 else 1* fac 0} \\
&\rightarrow \texttt{2* 1* fac 0} \\
&\rightarrow \texttt{2* 1* if 0 == 0 then 1 else 0* fac (0-1)} \\
&\rightarrow \texttt{2* 1* if True then 1 else 0* fac (0-1)} \\
&\rightarrow \texttt{2* 1* 1} \rightarrow \texttt{2}
\end{aligned}
$$

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␣"
  → **if** 2 == 0 **then** "" **else** "hallo␣" ++ repeat (2−1) "hallo␣"

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

  repeat 2 "hallo␣"
  → **if** 2 == 0 **then** "" **else** "hallo␣" ++ repeat (2−1) "hallo␣"
  → "hallo␣"++ repeat 1 "hallo␣"

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

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

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

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

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

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

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

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

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

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

Auswertung als Ausführungsbegriff

- **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 $E\left[\begin{array}{c} t \\ x \end{array}\right]$ ersetzt
- Auswertung kann **divergieren**!

Ausdrücke und Werte

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

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'`	`'\n'`
Zeichenketten	`String`	`"yuck"`	`"hi\nho\"\n"`	
Wahrheitswerte	`Bool`	`True`	`False`	
Funktionen	`a -> b`			

- Später mehr. **Viel** mehr.

Das Rechnen mit Zahlen

Beschränkte Genauigkeit,
konstanter Aufwand

$\longleftrightarrow$

beliebige Genauigkeit,
wachsender Aufwand

Das Rechnen mit Zahlen

Beschränkte Genauigkeit,
konstanter Aufwand

$\longleftrightarrow$

beliebige Genauigkeit,
wachsender Aufwand

Haskell bietet die Auswahl:

- Int - ganze Zahlen als Maschinenworte ($\geq$ 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: (div x y)*y + mod x y == x

- Vergleich durch $==, \neq, \leq, <, \ldots$
- **Achtung:** Unäres Minus
  - Unterschied zum Infix-Operator -
  - Im Zweifelsfall klammern: abs (-34)

Fließkommazahlen: Double

- Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
  - Logarithmen, Wurzel, Exponentation, 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

toLower  :: Char→ Char
toUpper  :: Char→ Char
isDigit  :: Char→ Bool
isAlpha  :: Char→ Bool
```

- Zeichenketten: String

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 $\mathtt{f} :: \mathtt{a} \to \mathtt{b}$

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 2 vom 23.10.2016: Funktionen

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Fahrplan

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

Inhalt

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

Organisatorisches

- Verteilung der Tutorien (laut stud.ip):

					Abweichung
Mi	08–10	MZH 1470	Thomas Barkowsky	16	-18
	10–12	MZH 1090	Tobias Haslop	50	**16**
	12–14	MZH 1470	Matz Habermann	49	**15**
	16–18	MZH 1090	Andreas Kästner	18	-16
Do	12–14	MZH 1090	Gerrit Marquardt	50	**15**
	16–18	MZH 1110	Gerrit Marquardt	23	-11

  - Wenn möglich, frühe/späte Tutorien belegen.

- Bewertung der Übungsblätter:
  - Dokumentation: kurz und knapp für jede Funktion
  - Code: auf guten Stil achten
  - Tests und Testfälle

Definition von Funktionen

Definition von Funktionen

- Zwei wesentliche Konstrukte:
  - Fallunterscheidung
  - Rekursion
- Reicht das?

Definition von Funktionen

- Zwei wesentliche Konstrukte:
  - Fallunterscheidung
  - Rekursion
- Reicht das?

Satz

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

- Funktionen müssen **partiell** sein können.
  - Insbesondere nicht-terminierende Rekursion

Haskell-Syntax: Charakteristika

- **Leichtgewichtig**
  - Wichtigstes Zeichen:
- Funktionsapplikation: `f a`
  - Klammern sind optional
  - **Höchste** Priorität (engste Bindung)
- Abseitsregel: Gültigkeitsbereich durch Einrückung
  - Keine Klammern ($\{ \ldots \}$)
- 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 (Geltungsberereich)?

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 . . .
```

. . . **bedingte Gleichungen**:

```
f x y
  | B1 =. . .
  | B2 =. . .
```

- Auswertung der Bedingungen von oben nach unten
- Wenn keine Bedingung wahr ist: **Laufzeitfehler**! Deshalb:

```
  | otherwise =. . .
```

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!

Bedeutung von Programmen

Auswertung von Funktionen

- Auswertung durch **Anwendung** von Gleichungen
- **Auswertungsrelation** $s \longrightarrow t$:
  - Anwendung einer Funktionsdefinition
  - Anwendung von elementaren Operationen (arithmetisch, Zeichenketten)

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):

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

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))  → dbl (inc 3)+ 1
                   → 2*(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):

```
inc (dbl (inc 3))  → dbl (inc 3)+ 1
                   → 2*(inc 3)+ 1
                   → 2*(3+ 1)+ 1
                   → 2*4+1 → 9
```

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))` → dbl (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

- 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):

```
inc (dbl (inc 3))  → dbl (inc 3)+ 1
                   → 2*(inc 3)+ 1
                   → 2*(3+ 1)+ 1
                   → 2*4+1 → 9
```

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

```
inc (dbl (inc 3))  → 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):

  `inc (dbl (inc 3))` → dbl (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

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

  inc (dbl (inc 3)) → inc (dbl (3+1))
  → inc (2*(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):

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

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

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

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):

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

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

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

Auswertung von Ausdrücken

```
inc :: Int → Int
inc x = x+1
```

```
dbl :: Int → Int
dbl x = 2*x
```

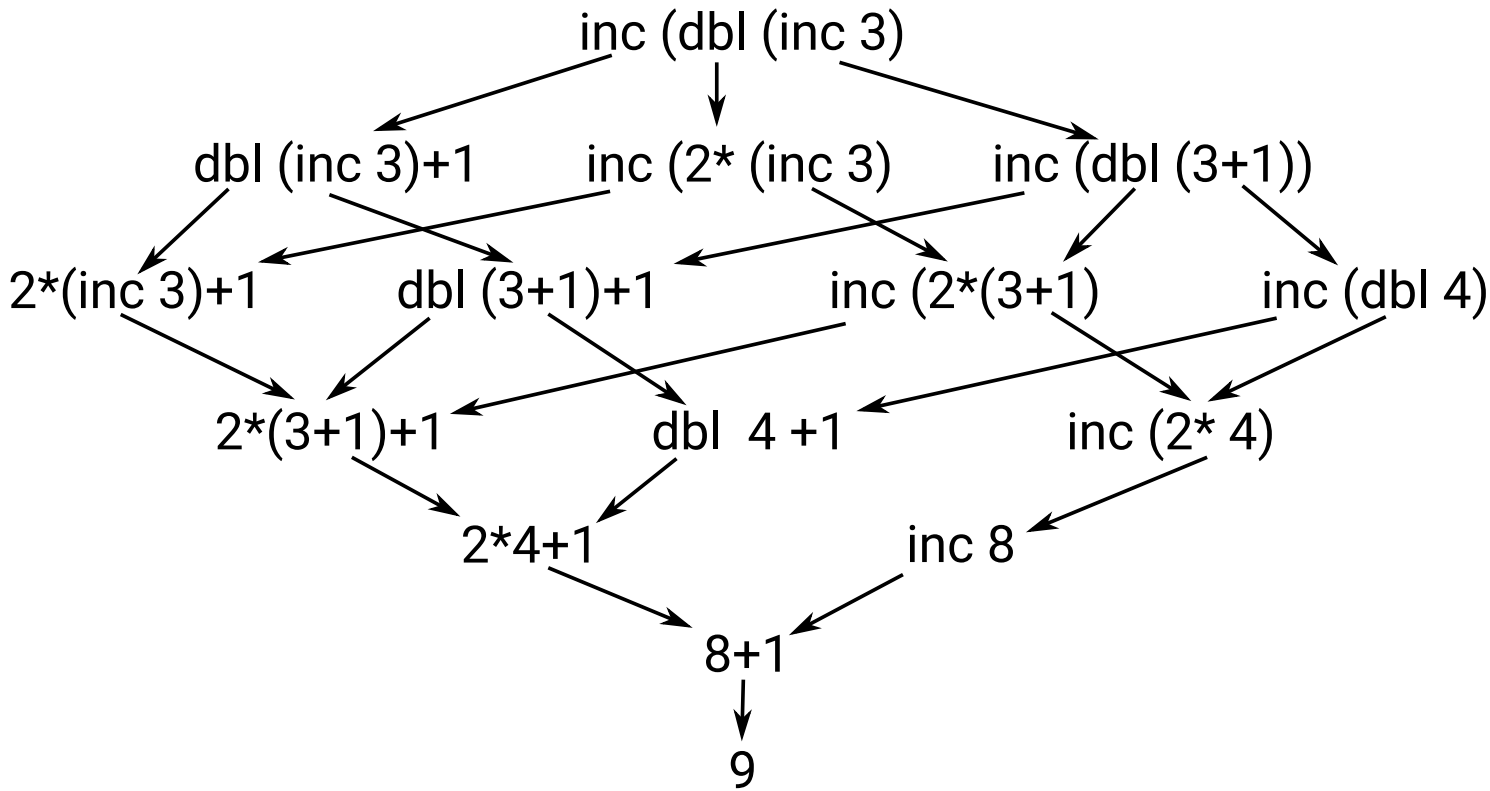
- Reduktion von `inc (dbl (inc 3))`

Auswertung von Ausdrücken

```
inc :: Int → Int
inc x = x+1
```

```
dbl :: Int → Int
dbl x = 2*x
```

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



Konfluenz

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

Definition (Konfluenz)

$\overset{*}{\rightarrow}$ ist **konfluent** gdw:
Für alle r, s, t mit $s \overset{*}{\leftarrow} r \overset{*}{\rightarrow} t$ gibt es u so dass $s \overset{*}{\rightarrow} u \overset{*}{\leftarrow} t$.

Konfluenz

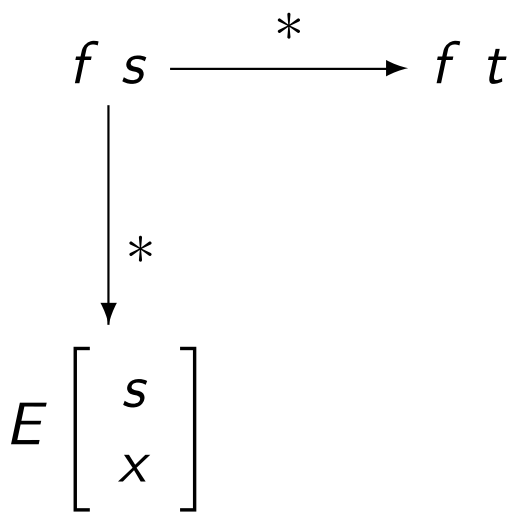
- Wenn wir von Laufzeitfehlern abstrahieren, gilt:

Theorem (Konfluenz)

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

- Beweisskizze:

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



Konfluenz

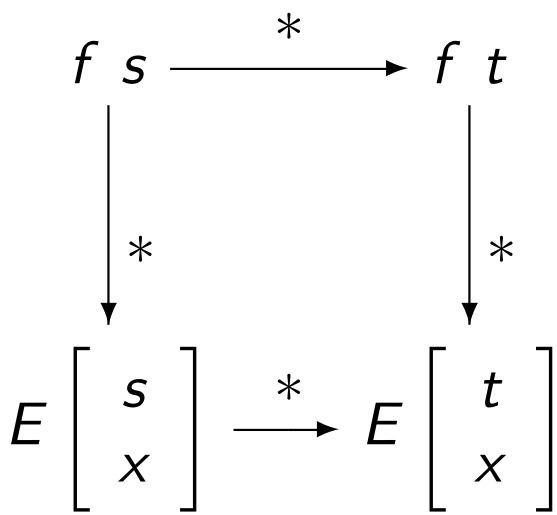
- Wenn wir von Laufzeitfehlern abstrahieren, gilt:

Theorem (Konfluenz)

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

- Beweisskizze:

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



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`:

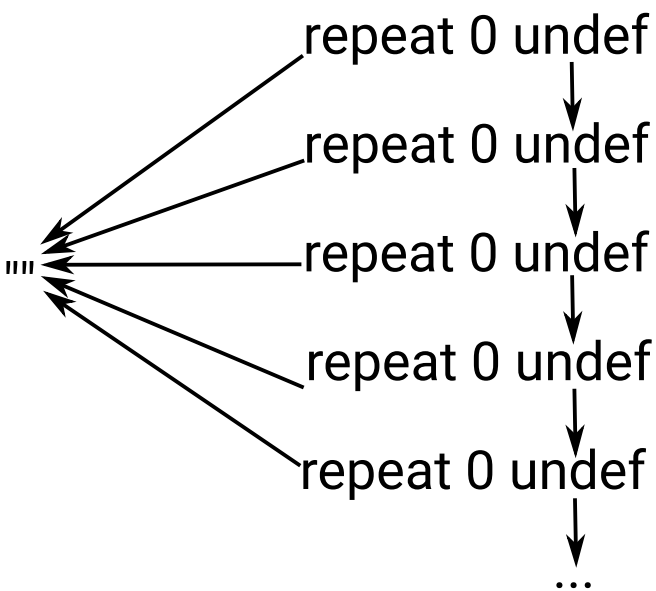
Auswirkung der Auswertungsstrategie

- 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`:



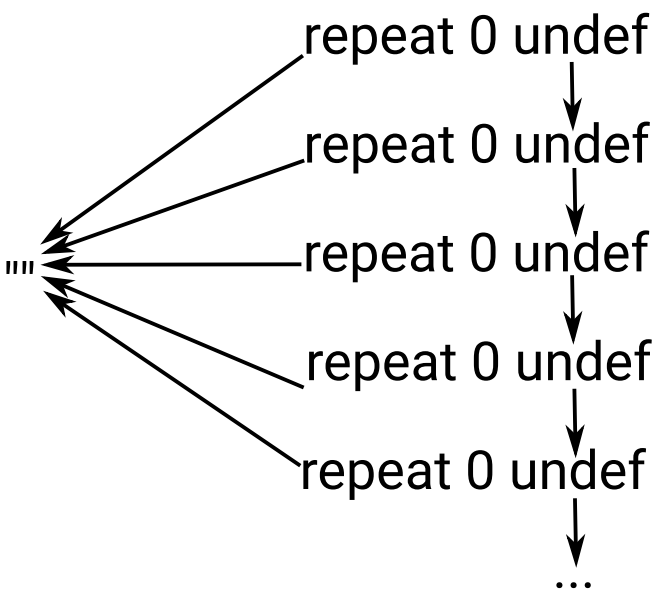
Auswirkung der Auswertungsstrategie

- 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**
- inntermost-first terminiert **nicht**

Termination und Normalform

Definition (Termination)

$\rightarrow$ ist **terminierend** gdw. es **keine unendlichen** Ketten gibt:

$$t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots t_n \rightarrow \ldots$$

Theorem (Normalform)

*Sei $\overset{*}{\rightarrow}$ 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 Auswertungsstragie 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)
- Auswertungsstrategie und Parameterübergabe:
  - Outermost-first entspricht **call-by-need**, verzögerte Auswertung.
  - Innermost-first entspricht **call-by-value**, strikte Auswertung
- Gibt es eine **semantische** Charakterisierung?

Bedeutung (Semantik) von Programmen

- **Operationale** Semantik:
  - Durch den **Ausführungsbegriff**
  - Ein Programm ist, was es tut.
  - In diesem Fall: $\rightarrow$
- **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, $\overset{*}{\rightarrow}$ die dadurch definierte Reduktion, und $[\![P]\!]$ das Denotat. Dann gilt für alle Ausdrücke t und Werte v

$$t \overset{*}{\rightarrow} v \iff [\![P]\!](t) = v$$

Striktheit

Definition (Striktheit)

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

- **Denotationelle** Eigenschaft (nicht operational)
- Haskell ist **nicht-strikt** (nach Sprachdefinition)
  - `repeat0 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.

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)
```

Suche in einer Zeichenkette

- Etwas lesbarer mit Guards:

```
count2 c s
  | null s = 0
  | head s == c = 1+count2 c (tail s)
  | otherwise   = count2 c (tail s)
```

- 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.
- Erster Versuch:

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

Strings analysieren: das Palindrom

- Zweiter Versuch:

```
palin2 :: String→ Bool
palin2 s =
  length s ≤ 1 || head s == last s && palin2 (init (tail s))
```

  - Terminiert wegen Nicht-Striktheit von ||

- Erweiterte Version:

```
palin3 s = palin2 (clean s)
```

  - 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)
```

Zusammenfassung

- **Bedeutung** von Haskell-Programmen:
  - Auswertungsrelation $\rightarrow$
  - Auswertungsstrategien: innermost-first, outermost-firsta
  - 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 30.10.2018: Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Organisatorisches

- Übungsbetrieb diese Woche
  - Übungsblatt mit 5 Punkten, Bearbeitungszeit bis Mo 12:00

Organisatorisches

- Übungsbetrieb diese Woche
  - Übungsblatt mit 5 Punkten, Bearbeitungszeit bis Mo 12:00
- Termine für E-Klausuren:
  - Probeklausur: vor Weihnachten
  - **Hauptklausur**: 08.03.2018 10:00 – 14:15
  - Wiederholungsklausur: 09.04. oder 11.04. (zweite Semesterwoche)

Fahrplan

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

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)

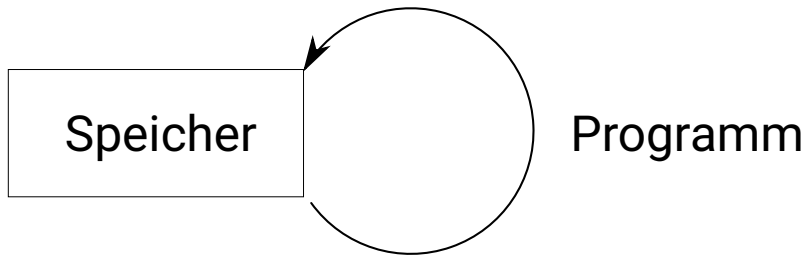
Warum Datentypen?

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

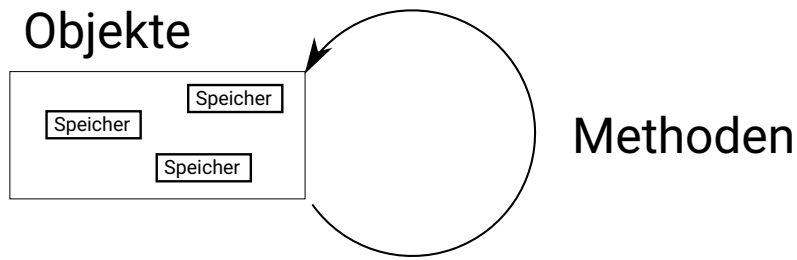
Datentypen als Modellierungskonstrukt

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

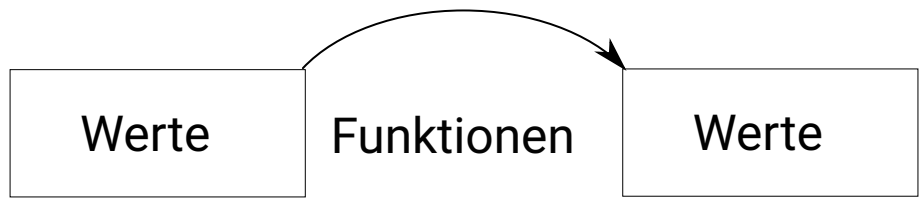
- Imperative Sicht:



- 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 Wertebereich *Apfel* muss drei Fälle unterscheiden
- Beispiel: $preis : Apfel \to \mathbb{N}$ mit

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

Aufzählung und Fallunterscheidung in Haskell

- **Definition**

```
data Apfel = Boskoop | CoxOrange | GrannySmith
```

  - Implizite Deklaration der **Konstruktoren** Boskoop :: Apfel als Konstanten
  - **Großschreibung** der Konstruktoren und Typsen

- **Fallunterscheidung**:

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

Aufzählung und Fallunterscheidung in Haskell

- **Definition**

```
data Apfel = Boskoop | CoxOrange | GrannySmith
```

  - Implizite Deklaration der **Konstruktoren** Boskoop :: Apfel als Konstanten
  - **Großschreibung** der Konstruktoren und Typsen

- **Fallunterscheidung**:

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

```
data Farbe = Rot | Grn
farbe :: Apfel→ Farbe
farbe d =
 case d of
  GrannySmith → Grn
  _ → Rot
```

Fallunterscheidung in der Funktionsdefinition

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

$$\begin{array}{l} f\ c_1 == e_1 \\ \ldots \\ f\ c_n == e_n \end{array} \longrightarrow f\ x == \text{case } x \text{ of } \begin{array}{lll} c_1 & \rightarrow & e_1 \\ \ldots & & \\ c_n & \rightarrow & e_n \end{array}$$

- Damit:

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

Der einfachste Aufzählungstyp

- **Einfachste** Aufzählung: Wahrheitswerte

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

  - Genau zwei unterschiedliche Werte

- **Definition** von Funktionen:

  - Wertetabellen sind explizite Fallunterscheidungen

$\wedge$	*true*	*false*
true	*true*	*false*
false	*false*	*false*

$$\begin{aligned} true \wedge true &= true \\ true \wedge false &= false \\ false \wedge true &= false \\ false \wedge false &= false \end{aligned}$$

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:

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

Striktheit Revisited

- **Konjunktion** definiert als

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

- Alternative Definition als Wahrheitstabelle:

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

Unterschied?

Striktheit Revisited

- **Konjunktion** definiert als

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

- Alternative Definition als Wahrheitstabelle:

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

Unterschied?

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

Produkte

- Konstruktoren können **Argumente** haben
- Beispiel: Ein **RGB-Wert** besteht aus drei Werten
- Mathematisch: Produkt (Tripel)

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

- 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
```

```
green :: Colour → Int
green (RGB _ g _) = g
```

  - Beispielauswertungen

```
red yellow ⇝ 255
green violet ⇝ 130
```

Beispiel: Produkte in Bob's Shoppe

- Käsesorten und deren Preise:

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

```
kpreis :: Kaesesorte → Int
kpreis Gouda = 1450
kpreis Appenzeller = 2270
```

Beispiel: Produkte in Bob's Shoppe

- Käsesorten und deren Preise:

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

```
kpreis :: Kaesesorte → Int
kpreis Gouda = 1450
kpreis Appenzeller = 2270
```

- Alle Artikel:

```
data Artikel =
    Apfel Apfelsorte   | Eier
  | Kaese Kaesesorte   | Schinken
  | Salami             | Milch Bio
```

```
data Bio = Bio | Konv
```

Beispiel: Produkte in Bob's Shoppe

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

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

- Preisberechung

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

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

Beispiel: Produkte in Bob's Shoppe

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

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

- Preisberechung

```
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: Produkte in Bob's Shoppe

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

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

- Preisberechung

```
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: Produkte in Bob's Shoppe

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

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

- Preisberechung

```
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: Produkte in Bob's 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; Konv → 69))
preis _ _ = 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 → 1; 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 → 1; Ungueltig → 0
```

<

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

- Auswertungen:

  f (Cent undefined) ⇝ *** Exception: Prelude.undefined
  g (Cent undefined) ⇝ 1
  f (Cent (g (Cent undefined))) ⇝ 1
  g (add (Cent 1) (Cent undefined)) ⇝ 1
  f (add (Cent undefined) Ungueltig) ⇝ 0

Der Allgemeine Fall: Algebraische Datentypen

$$
\begin{array}{lll}
\text{data T} = & C_1 \\
& \mid & C_2 \\
& & \vdots \\
& \mid & C_n
\end{array}
$$

- **Aufzählungen**

Der Allgemeine Fall: Algebraische Datentypen

$$\text{data T} = \quad \text{C}_1 \ \text{t}_{1,1} \ \ldots \ \text{t}_{1,k_1}$$

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

Der Allgemeine Fall: Algebraische Datentypen

$$
\begin{array}{lll}
\text{data T} = & \mathsf{C}_1\ \mathsf{t}_{1,1} & \ldots\ \mathsf{t}_{1,k_1} \\
\mid & \mathsf{C}_2\ \mathsf{t}_{2,1} & \ldots\ \mathsf{t}_{2,k_2} \\
& \vdots & \\
\mid & \mathsf{C}_n\ \mathsf{t}_{n,1} & \ldots\ \mathsf{t}_{n,k_n}
\end{array}
$$

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

Eigenschaften algebraischer Datentypen

$$
\begin{array}{rcl}
\text{data T} = & \mathsf{C}_1\ \mathsf{t}_{1,1} \ldots \mathsf{t}_{1,k_1} \\
\mid & \mathsf{C}_2\ \mathsf{t}_{2,1} \ldots \mathsf{t}_{2,k_2} \\
& \vdots \\
\mid & \mathsf{C}_n\ \mathsf{t}_{n,1} \ldots \mathsf{t}_{n,k_n}
\end{array}
$$

Drei Eigenschaften eines algebraischen Datentypen

1. Konstruktoren $\mathsf{C}_1, \ldots, \mathsf{C}_n$ sind **disjunkt**:

$$\mathsf{C}_i\ \mathsf{x}_1 \ldots \mathsf{x}_n = \mathsf{C}_j\ \mathsf{y}_1 \ldots \mathsf{y}_m \Longrightarrow i = j$$

Eigenschaften algebraischer Datentypen

$$
\begin{array}{lll}
\text{data } T = & C_1\ t_{1,1} \ldots t_{1,k_1} \\
\mid & C_2\ t_{2,1} \ldots t_{2,k_2} \\
& \vdots \\
\mid & C_n\ t_{n,1} \ldots t_{n,k_n}
\end{array}
$$

Drei Eigenschaften eines algebraischen Datentypen

1. Konstruktoren $C_1, \ldots, C_n$ sind **disjunkt**:
$$C_i\ x_1 \ldots x_n = C_j\ y_1 \ldots y_m \Longrightarrow i = j$$
2. Konstruktoren sind **injektiv**:
$$C\ x_1 \ldots x_n = C\ y_1 \ldots y_n \Longrightarrow x_i = y_i$$

Eigenschaften algebraischer Datentypen

$$
\begin{aligned}
\text{data T} = \quad & \mathsf{C}_1\ \mathsf{t}_{1,1} \ldots \mathsf{t}_{1,k_1} \\
\mid \quad & \mathsf{C}_2\ \mathsf{t}_{2,1} \ldots \mathsf{t}_{2,k_2} \\
& \vdots \\
\mid \quad & \mathsf{C}_n\ \mathsf{t}_{n,1} \ldots \mathsf{t}_{n,k_n}
\end{aligned}
$$

Drei Eigenschaften eines algebraischen Datentypen

1. Konstruktoren $\mathsf{C}_1, \ldots, \mathsf{C}_n$ sind **disjunkt**:
$$\mathsf{C}_i\ \mathsf{x}_1 \ldots \mathsf{x}_n = \mathsf{C}_j\ \mathsf{y}_1 \ldots \mathsf{y}_m \Longrightarrow i = j$$
2. Konstruktoren sind **injektiv**:
$$\mathsf{C}\ \mathsf{x}_1 \ldots \mathsf{x}_n = \mathsf{C}\ \mathsf{y}_1 \ldots \mathsf{y}_n \Longrightarrow \mathsf{x}_i = \mathsf{y}_i$$
3. Konstruktoren **erzeugen** den Datentyp:
$$\forall x \in T.\, x = \mathsf{C}_i\ \mathsf{y}_1 \ldots \mathsf{y}_m$$

Eigenschaften algebraischer Datentypen

$$
\begin{array}{rcl}
\text{data } T = & C_1\ t_{1,1} \ldots t_{1,k_1} \\
\mid & C_2\ t_{2,1} \ldots t_{2,k_2} \\
& \vdots \\
\mid & C_n\ t_{n,1} \ldots t_{n,k_n}
\end{array}
$$

Drei Eigenschaften eines algebraischen Datentypen

1. Konstruktoren $C_1, \ldots, C_n$ sind **disjunkt**:
$$C_i\ x_1 \ldots x_n = C_j\ y_1 \ldots y_m \Longrightarrow i = j$$
2. Konstruktoren sind **injektiv**:
$$C\ x_1 \ldots x_n = C\ y_1 \ldots y_n \Longrightarrow x_i = y_i$$
3. Konstruktoren **erzeugen** den Datentyp:
$$\forall x \in T.\, x = C_i\ y_1 \ldots y_m$$

Diese Eigenschaften machen **Fallunterscheidung** wohldefiniert.

Algebraische Datentypen: Nomenklatur

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

- C_i sind **Konstruktoren**
  - **Immer** implizit definiert und deklariert
- **Selektoren** sind Funktionen $sel_{i,j}$:

$$\begin{aligned} sel_{i,j} &:: T \to t_{i,k_i} \\ sel_{i,j}\ (C_i\ t_{i,1} \ldots\ t_{i,k_i}) &= t_{i,j} \end{aligned}$$

  - Partiell, linksinvers zu Konstruktor C_i
  - **Können** implizit definiert und deklariert werden
- **Diskriminatoren** sind Funktionen dis_i:

$$\begin{aligned} dis_i &:: T \to \text{Bool} \\ dis_i\ (C_i \ldots) &= \text{True} \\ dis_i\ \_ &= \text{False} \end{aligned}$$

  - Definitionsbereichsbereich des Selektors sel_i, **nie** implizit

Rekursive Algebraische Datentypen

$$\begin{array}{rcl} \text{data T} & = & \mathsf{C}_1\ \mathsf{t}_{1,1} \ldots \mathsf{t}_{1,k_1} \\ & \vdots & \\ & | & \mathsf{C}_n\ \mathsf{t}_{n,1} \ldots \mathsf{t}_{n,k_n} \end{array}$$

- 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 l)
  | art == lart = Gefunden m
  | otherwise   = suche 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 (35l Milch + 20l Milch = 55l 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 l)
  | a == al   = Lager a (addiere m ml) l
  | otherwise = Lager al ml (einlagern a m l)
```

- Problem:

Einlagern

- Damit einlagern:

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

- Problem: **Falsche Mengenangaben**
  - Bspw. `einlagern Eier (Liter 3.0) l`
  - 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 **Einkausfwagen:**

```
data Einkaufswagen = LeererWagen
                   | Einkauf Artikel Menge Einkaufswagen
```

- Artikel einkaufen:

```
einkauf :: Artikel→ Menge→ Einkaufswagen→ Einkaufswagen
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 :: Einkaufswagen→ String
```

Ausgabe:

```
* Bob's Aulde Grocery Shoppe *

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
```

Unveränderlicher Kopf

Ausgabe von Artikel und Menge (rekursiv)

Ausgabe von kasse

Kassenbon: Implementation

- Kernfunktion:

```
artikel :: Einkaufswagen→ String
artikel LeererWagen = ""
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
```

Beispiel: Zeichenketten selbstgemacht

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

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

- **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 :: String → Int
length Empty    = 0
length (c :+ s) = 1 + length s
```

Funktionen auf Zeichenketten

- Länge:

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

- Verkettung:

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

Funktionen auf Zeichenketten

- Länge:

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

- Verkettung:

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

- Umdrehen:

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

Zusammenfassung

- Algebraische Datentypen: Aufzählungen, Produkte, rekursive Datentypen
- 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 06.11.2018: Typvariablen und Polymorphie

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Fahrplan

- **Teil I: Funktionale Programmierung im Kleinen**
  - Einführung
  - Funktionen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Zyklische Datenstrukturen
  - Funktionen höherer Ordnung I
  - 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

Ähnliche Datentypen der letzten Vorlesung

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

```
data Einkaufswagen = LeererWagen
                   | Einkauf Artikel Menge Einkaufswagen
```

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

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

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufswagen→ Int
kasse LeererWagen = 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 :: String→ 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).

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: Typ

  Empty

Polymorphe Ausdrücke

- **Typkorrekte** Terme:

	Typ
Empty	List α

Polymorphe Ausdrücke

- **Typkorrekte** Terme:

	Typ
Empty	List α
Cons 57 Empty	

Polymorphe Ausdrücke

- **Typkorrekte** Terme:

	Typ
Empty	List α
Cons 57 Empty	List Int

Polymorphe Ausdrücke

- **Typkorrekte** Terme:

	Typ
`Empty`	`List` α
`Cons 57 Empty`	`List Int`
`Cons 7 (Cons 8 Empty)`	

Polymorphe Ausdrücke

- **Typkorrekte** Terme:

	Typ
Empty	List α
Cons 57 Empty	List Int
Cons 7 (Cons 8 Empty)	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` :: $\alpha \rightarrow$ `List` $\alpha \rightarrow$ `List` α

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 3 Empty ++ Cons 5 (Cons 57 Empty)
Cons 'p' (Cons 'i' Empty) ++ Cons '3' 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
- Lösung: zu einem Typ zusammenfassen

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

- Damit:

```
type Lager = [Posten]
```

```
type Einkaufswagen = [Posten]
```

- **Gleicher** Typ!
  - Bug or Feature?

Beispiel: Der Shop (refaktoriert)

- Einkaufswagen und Lager als Listen?
- Problem: zwei Typen als Argument
- Lösung: zu einem Typ zusammenfassen

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

- Damit:

```
type Lager = [Posten]
```

```
type Einkaufswagen = [Posten]
```

- **Gleicher** Typ!
  - Bug or Feature? Bug!
- Lösung: Datentyp **verkapseln**

```
data Lager = Lager [Posten]
```

```
data Einkaufswagen = Ekwg [Posten]
```

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  :: α→ β→ Pair α β
left  :: Pair α β → α
right :: Pair α β → β
```

- Beispielterm | Typ

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  :: α→ β→ Pair α β
left  :: Pair α β → α
right :: Pair α β → β
```

- Beispielterm — Typ

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  :: α→ β→ Pair α β
left  :: Pair α β → α
right :: Pair α β → β
```

- Beispielterm — Typ

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

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

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

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)

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: listToMaybe (maybeToList m) = m
  length l ≤ 1 ⟹ maybeToList (listToMaybe l) = l

Übersicht: vordefinierte Funktionen auf Listen I

```
(++)     :: [α] → [α] → [α]          — Verketten
(!!)     :: [α] → Int → α            — n-tes Element selektieren
concat   :: [[α]] → [α]              — "flachklopfen"
length   :: [α] → Int                — Länge
head, last :: [α] → α                — Erstes/letztes Element
tail, init :: [α] → [α]              — Hinterer/vorderer Rest
replicate  :: Int → α → [α]          — Erzeuge n Kopien
repeat   :: α → [α]                  — Erzeugt zyklische Liste
take     :: Int → [α] → [α]          — Erste n Elemente
drop     :: Int → [α] → [α]          — Rest nach n Elementen
splitAt  :: Int → [α] → ([α], [α])   — Spaltet an Index n
reverse  :: [α] → [α]                — Dreht Liste um
zip      :: [α] → [β] → [(α, β)]     — Erzeugt Liste von Paaren
unzip    :: [(α, β)] → ([α], [β])    — Spaltet Liste von Paaren
and, or  :: [Bool] → Bool            — Konjunktion/Disjunktion
sum      :: [Int] → Int              — Summe (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")
```

Typherleitung

Typen in Haskell (The Story So Far)

- Primitive Basisdatentypen: Bool, Double
- Funktionstypen Double→ Int→ Int, [Char] → Double
- Typkonstruktoren: [ ], (. . .), Foo
- Typvariablen

$$\begin{aligned} \mathsf{fst} &:: (\alpha, \beta) \to \alpha \\ \mathsf{length} &:: [\alpha] \to \mathsf{Int} \\ (+\!\!+) &:: [\alpha] \to [\alpha] \to [\alpha] \end{aligned}$$

- Typklassen :

$$\begin{aligned} \mathsf{elem} &:: \mathsf{Eq\ a} \Rightarrow \mathsf{a} \to [\mathsf{a}] \to \mathsf{Bool} \\ \mathsf{max} &:: \mathsf{Ord\ a} \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{a} \end{aligned}$$

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 + length 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

$$\begin{array}{llllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\ & & & & [\alpha] \rightarrow & \mathsf{Int} \end{array}$$

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

$$\begin{array}{lllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length\ \ xs} \\ & & & & [\alpha] \to \mathsf{Int} \\ & & & & \quad [\alpha] \end{array}$$

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

$$\begin{array}{lllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length}\ \ \mathsf{xs} \\ & & & & [\alpha] \to \mathsf{Int} \\ & & & & \qquad\qquad [\alpha] \\ & & & & \qquad\quad \mathsf{Int} \end{array}$$

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

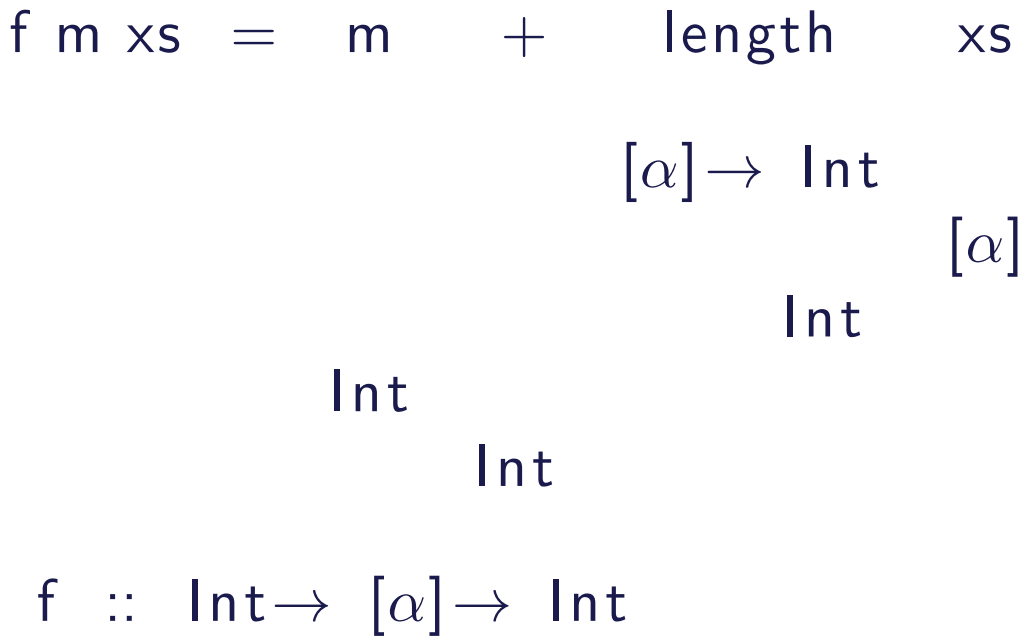
$$\begin{array}{lllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} \quad \mathsf{xs} \\ & & & & [\alpha] \to \mathsf{Int} \\ & & & & \qquad\qquad [\alpha] \\ & & & & \qquad\quad \mathsf{Int} \\ & & \mathsf{Int} & & \end{array}$$

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



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
```

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**:

$$\begin{array}{llllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\ & & \alpha & & [\beta] \to \mathsf{Int} & \gamma \end{array}$$

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**:

$$\begin{array}{llllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\ & & \alpha & & [\beta] \to \mathsf{Int} & \gamma \\ & & & & & [\beta] \quad \gamma \mapsto [\beta] \end{array}$$

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**:

$$\begin{array}{llllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\ & & \alpha & & [\beta] \to \mathsf{Int} & \gamma \\ & & & & & [\beta] \quad \gamma \mapsto [\beta] \\ & & & & \mathsf{Int} & \end{array}$$

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**:

$$\begin{array}{llllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length} & \mathsf{xs} \\ & & \alpha & & [\beta] \to \mathsf{Int} & \gamma \\ & & & & & [\beta] \quad \gamma \mapsto [\beta] \\ & & & & \mathsf{Int} & \\ & & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} & & & \end{array}$$

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**:

$$\begin{array}{lllll} \mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length\ xs} \\ & & \alpha & & [\beta] \to \mathsf{Int} \quad \gamma \\ & & & & [\beta] \quad \gamma \mapsto [\beta] \\ & & & & \mathsf{Int} \\ & & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} & & \\ & & \mathsf{Int} & & \alpha \mapsto \mathsf{Int} \end{array}$$

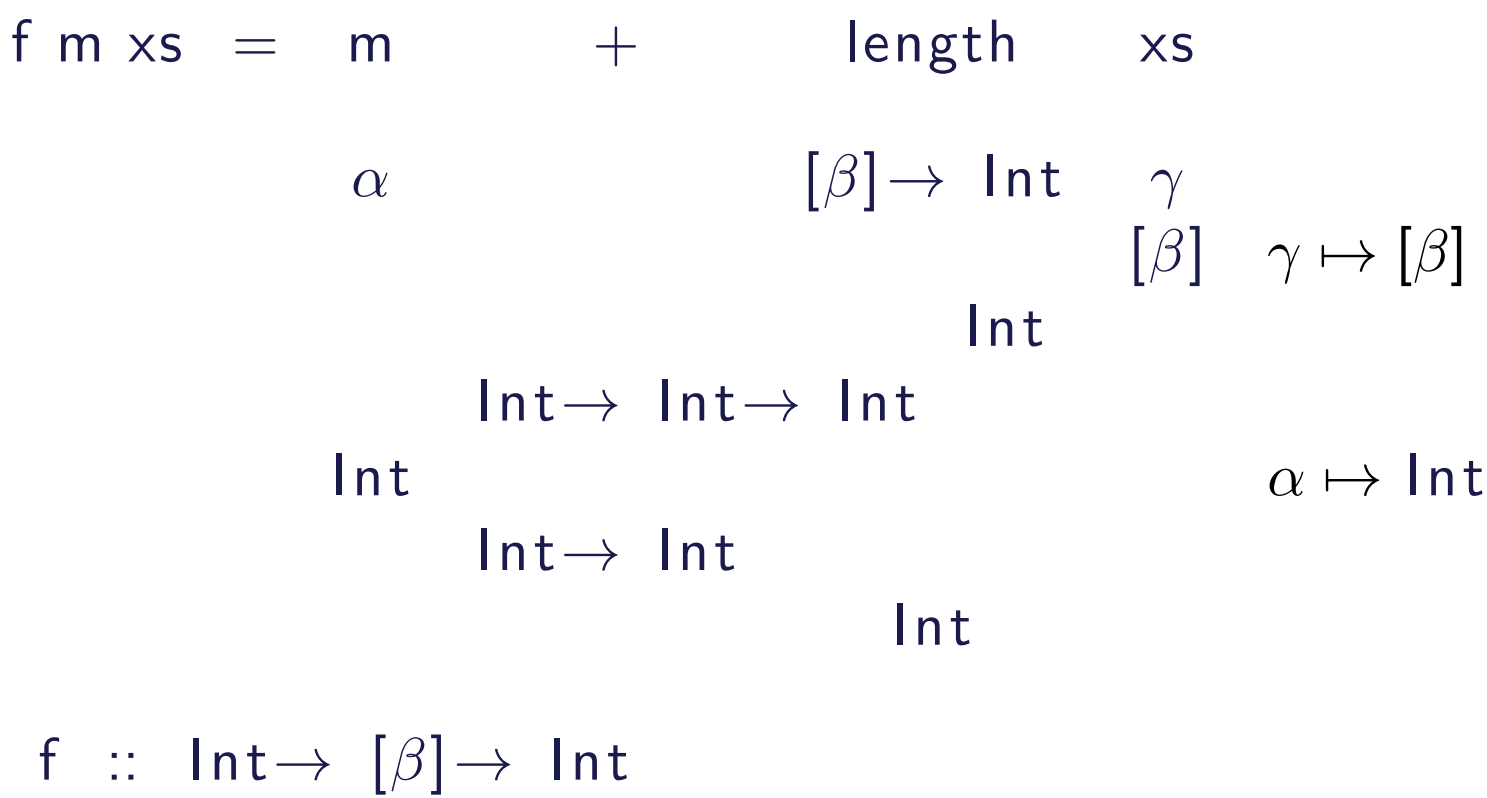
Typinferenz (nach Hindley-Milner)

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

$$
\begin{array}{lllll}
\mathsf{f\ m\ xs} & = & \mathsf{m} & + & \mathsf{length\ xs} \\
 & & \alpha & & [\beta] \to \mathsf{Int} \quad \gamma \\
 & & & & [\beta] \quad \gamma \mapsto [\beta] \\
 & & & & \mathsf{Int} \\
 & & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} & & \\
 & & \mathsf{Int} & & \alpha \mapsto \mathsf{Int} \\
 & & \mathsf{Int} \to \mathsf{Int} & &
\end{array}
$$

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

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 Substituitionen beinhalten:

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

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

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

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

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

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

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

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

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

Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

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

$$\mathsf{f} :: \mathsf{Char} \to \mathsf{Int} \to [(\mathsf{Char},\ \mathsf{Int})]$$

- Allgemeinster Typ **muss nicht** existieren (Typfehler!)
  - Bsp: $[\mathsf{True}] +\!\!+ [3]\ , \mathsf{x} : \mathsf{x}$

Ad-Hoc Polymorphie

Ad-Hoc Polymorphie und Overloading

Definition (Überladung)

Funktion f :: a→ b existiert für **mehr als einen**, aber **nicht** für **alle** Typen

- Beispiel:
  - Gleichheit: $(==)$ :: $\alpha \rightarrow \alpha \rightarrow$ Bool
  - Vergleich: $(\leq)$ :: $\alpha \rightarrow \alpha \rightarrow$ Bool
  - Anzeige: show :: $\alpha \rightarrow$ String
- Lösung: **Typklassen**
- Typklassen bestehen aus:
  - **Deklaration** der Typklasse
  - **Instantiierung** für bestimmte Typen

Typklassen: Syntax

- **Deklaration**:

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

- **Instantiierung**:

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

- Prominente vordefinierte Typklassen
  - Eq für $(==)$
  - Ord für $(\leq)$ (und andere Vergleiche)
  - Show für show
  - Num (uvm) für numerische Operationen (Literale überladen)

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:

```
showsorted :: (Ord α, Show α)⇒ [α]→ String
showsorted 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

Abschließende Bemerkungen

Polymorphie: the missing link

	Parametrisch	Ad-Hoc
Funktionen	f :: $\alpha \rightarrow$ Int	**class** F α **where** f :: a $\rightarrow$ Int
Typen	**data** Maybe α = Just α | Nothing	

Polymorphie: the missing link

	Parametrisch	Ad-Hoc
Funktionen	f :: $\alpha \rightarrow$ Int	**class** F α **where** f :: a $\rightarrow$ Int
Typen	**data** Maybe α = Just α | 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 Maybe α
  - Tupel (α, β)

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 5 vom 13.11.2018: Rekursive und zyklische Datenstrukturen

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Organisatorisches

- Abgabefrist Übungsblätter ab jetzt bis **Dienstag 12:00**
- Termin Probeklausur: **17.12.18** ab **10:15**

Fahrplan

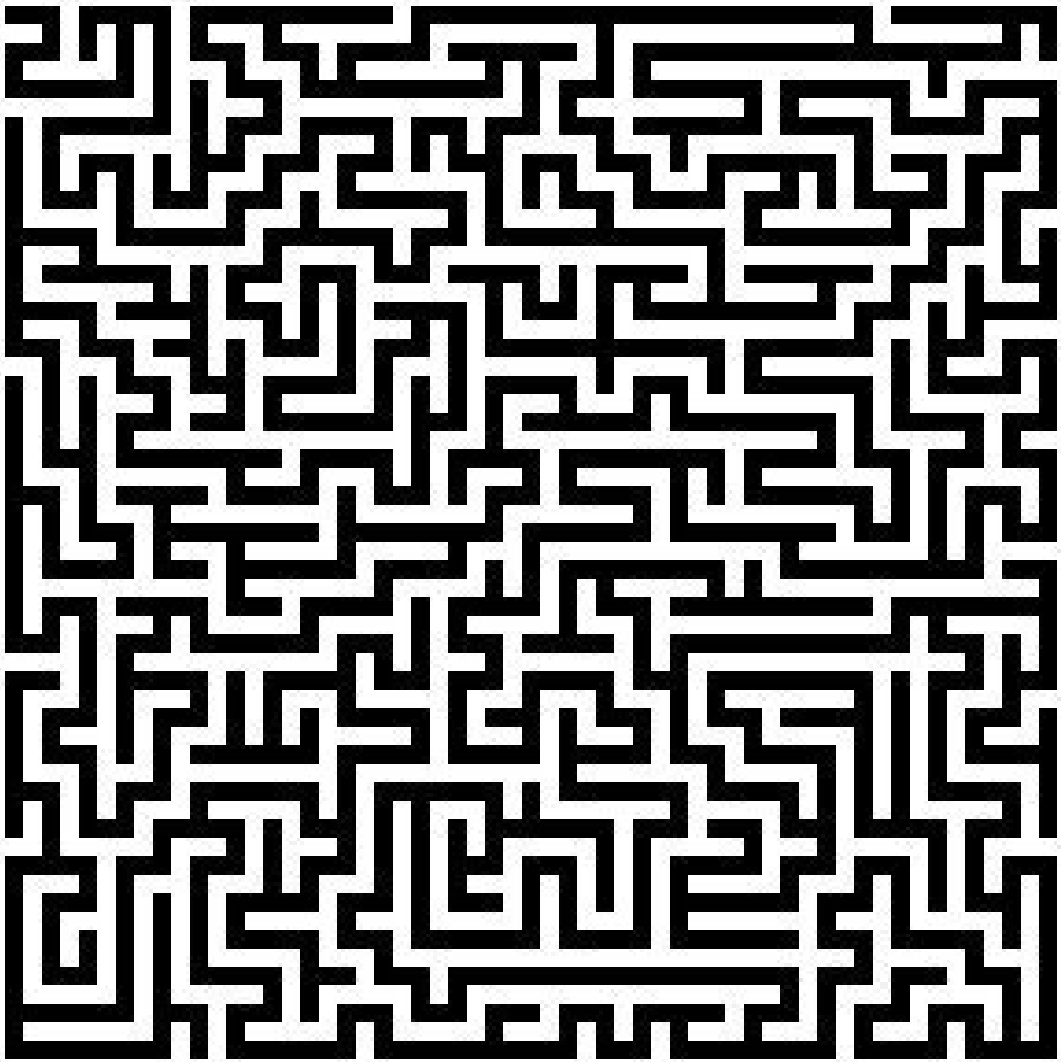
- **Teil I: Funktionale Programmierung im Kleinen**
  - Einführung
  - Funktionen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Zyklische Datenstrukturen
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale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

Inhalt

- **Rekursive** Datentypen und **zyklische** Daten
  - ... und wozu sie nützlich sind
  - Fallbeispiel: Labyrinth
- Datentypen und Polymorphie in anderen Sprachen
- Performance-Aspekte

Rekursive und Zyklische Datenstrukturen

Fallbeispiel: Zyklische Datenstrukturen



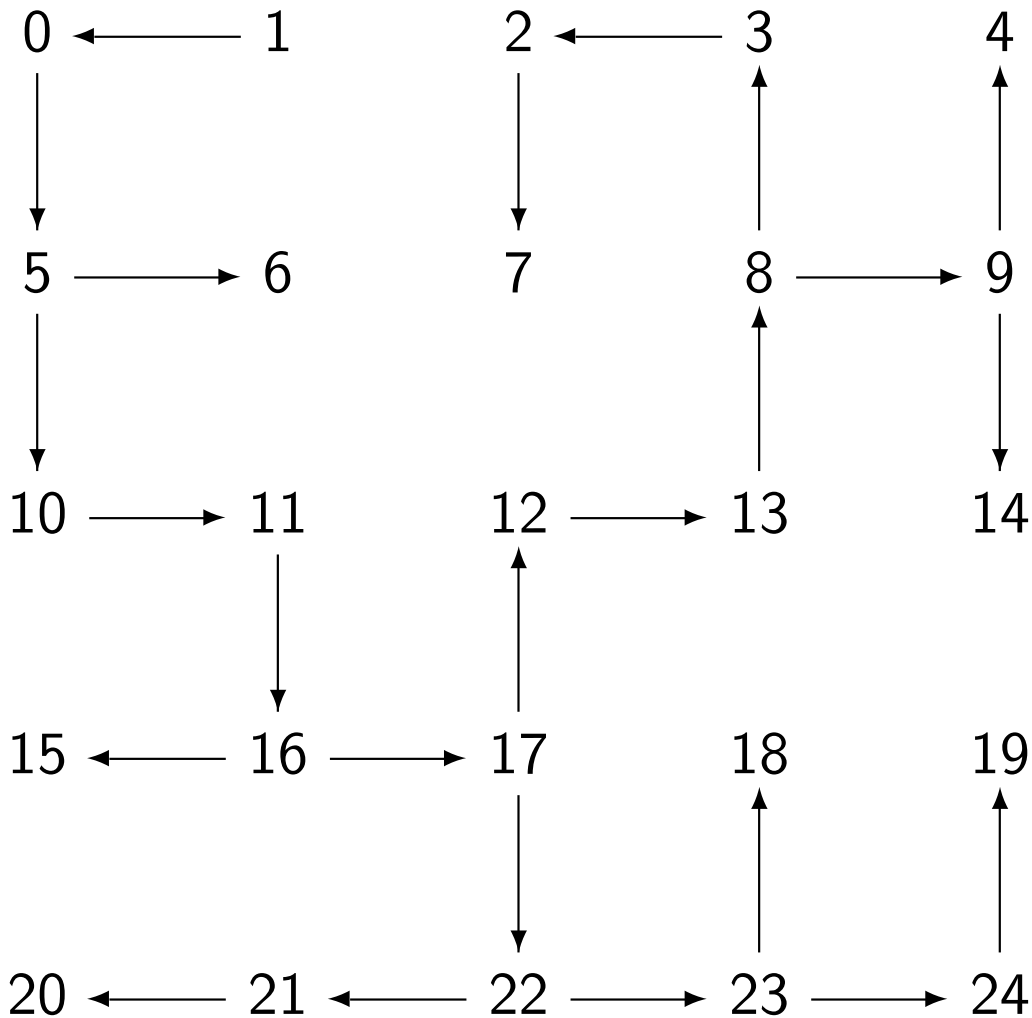
Quelle: docs.gimp.org

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.

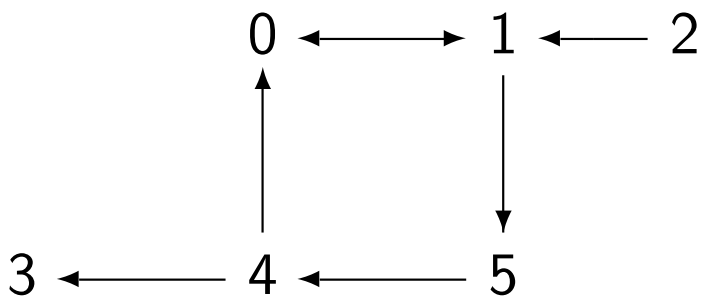
```
data Lab α = Dead α
           | Pass α (Lab α)
           | TJnc α (Lab α) (Lab α)
```

Ein Labyrinth (zyklenfrei)



Definition von Labyrinthen

Ein einfaches Labyrinth:



Definition in Haskell:

```
lab0 = Pass 0 lab1
lab1 = TJnc 1 lab0 lab5
lab2 = Pass 2 lab1
lab3 = Dead 3
lab4 = TJnc 4 lab0 lab3
lab5 = Pass 5 lab4
```

- **Rekursiv**!

- Freundlichere Notation:

```
0 ⟶ 1
1 ⟶ 0 5
2 ⟶ 1
3 ⟶
4 ⟶ 0 3
5 ⟶ 4
```

Traversion des Labyrinths

- Ziel: **Pfad** zu einem gegeben **Ziel** finden
- Benötigt Pfade und Traversion
- Pfade: Liste von Knoten

```
type Path α = [α]
```

- Traversion: 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 :: 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 Traversion

- Zusammengesetzt:

```
traverse_1 :: 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

Traversion 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

Traversion 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:

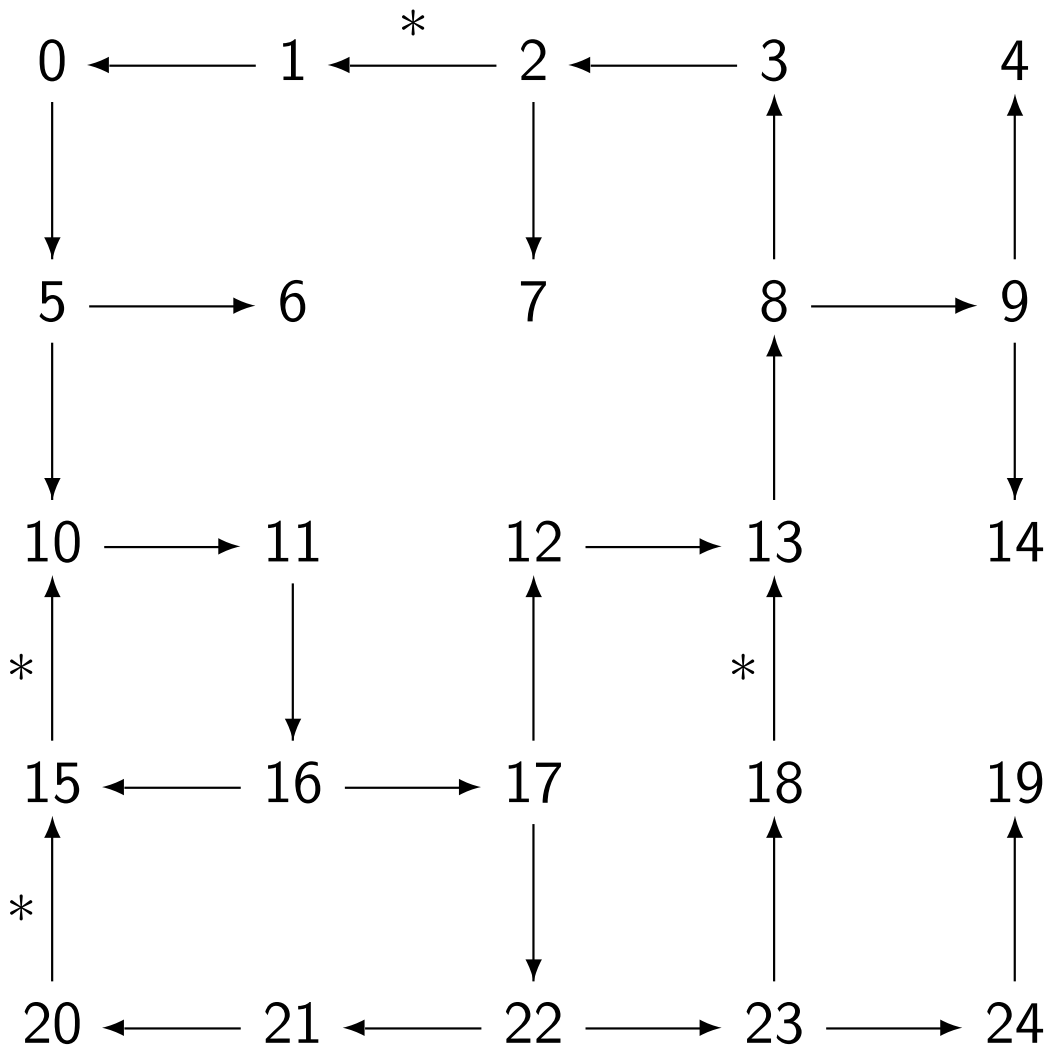
Traversion mit Zyklen

```
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 $O(n)$
  - Knoten können mehrfach besucht werden
  - Abhilfe: Menge der besuchten Knoten getrennt von aufgebautem Pfad
  - Erfordert effiziente Datenstrukturen für Mengen (Data.Set, Data.IntSet)

$\longrightarrow$ später

Ein Labyrinth (mit Zyklen)



Labyrinthe lesen

- Zerlegung in zwei Schritte:

```
readLab :: (Read α, Eq α, Show α)⇒ String→ α→ Lab α
readLab = makeLab ∘ parseString
```

- `parseString` zerlegt Zeichenkette in Zeilenbestandteile

```
parseString :: String→ [(String, [String])]
```

- `makeLab` erzeugt daraus das Labyrinth

```
makeLab :: (Read α, Eq α, Show α) ⇒
           [(String, [String])]→ α→ Lab α
```

Labyrinthe konstruieren

- Problem: Zyklische Referenzen
- Abbildung von α auf Lab α wird aus Argument konstruiert
- Das gesamte Labyrinth ist der Fixpunkt der Konstruktion

Labyrinthe konstruieren

- Problem: Zyklische Referenzen
- Abbildung von α auf Lab α wird aus Argument konstruiert
- Das gesamte Labyrinth ist der Fixpunkt der Konstruktion

```
makeLab :: (Read α, Eq α, Show α) ⇒
           [(String, [String])]→ α→ Lab α
makeLab vs id =
  let mk_lab map [] = []
      mk_lab map ((s, ts):rest) =
        let src = read s
            get v = fromJust (lookup (read v) map)
            l = case ts of
                  [] → Dead src
                  [v] → Pass src (get v)
                  [v1, v2] → TJnc src (get v1) (get v2)
                  _ → error ("Too many edges from "++ show src++ ": "
        in  (src, l): mk_lab map rest
      map = mk_lab map vs
  in fromMaybe (error ("Undefined label: "++ show id)) (lookup id map)
```

Der allgemeine Fall: variadische Bäume

- Labyrinth $\longrightarrow$ **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 (NT l vs) p
    | l == t = Just (reverse (l: p))
    | elem l p = Nothing
    | otherwise = select (travList (l: p) vs)
  travList p [] = []
  travList p (nt: nts) = trav nt p : travList p nts
```

Vorteile der Nicht-Strikten Auswertung

Unendliche Listen

- Auch Listen müssen nicht **endlich repräsentierbar** sein:
  - E.g. Unendliche Liste `[2,2,2,...]`

```
twos = 2 : twos
```

  - Liste der natürlichen Zahlen:

```
nat  = [1..]
```

  - Bildung von unendlichen Listen:

```
cycle :: [a]→ [a]
cycle xs = xs ++ cycle xs
```

- Repräsentation durch endliche, zyklische Datenstruktur
  - Kopf wird nur einmal ausgewertet.

```
cycle (trace "Foo!" [5])
```

- Nützlich für Listen mit a priori unbekannter Länge

Berechnung der ersten n Primzahlen

- Eratosthenes — aber bis wo sieben?

Berechnung der ersten n Primzahlen

- Eratosthenes — aber bis wo sieben?
- Lösung: Berechnung **aller** Primzahlen, davon die ersten n.

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

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

- Von allen Primzahlen die ersten:

```
primes :: Int→ [Integer]
primes n = take n allprimes
```

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 : zipPlus fibs (tail fibs) where
  zipPlus (a:as) (b:bs) = a+b: zipPlus as bs
```

- *n*-te Fibonaccizahl mit `fibs !! n`:

```
fib2 :: Integer→ Integer
fib2 n = genericIndex fibs n
```

- **Aufwand**: **linear**, da `fibs` nur einmal ausgewertet wird.

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)
```

- Beispiel: last (rev [1..10000])
- **Schneller** — warum?

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.
- Ist nicht merklich schneller?!

Verzögerte Auswertung und Speicherlecks

- **Garbage collection** gibt unbenutzten Speicher wieder frei.
  - **Unbenutzt**: Bezeichner nicht mehr 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 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 :: $\alpha \to \beta \to \beta$

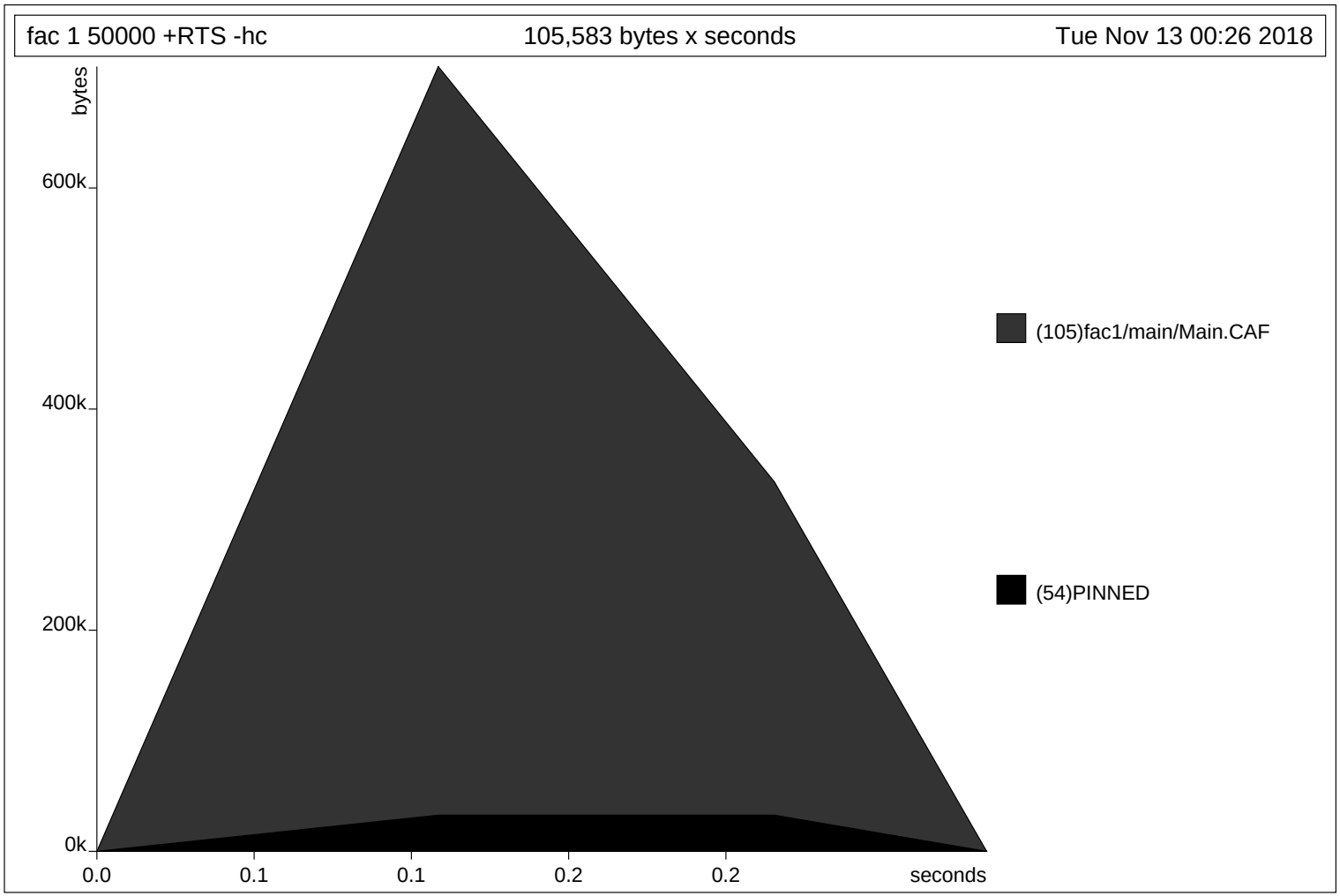
  $\bot$ `'seq'` b = $\bot$

  a `'seq'` b = b
  - seq vordefiniert (nicht in Haskell definierbar)
  - ($!) :: (a → b) → a → b strikte Funktionsanwendung

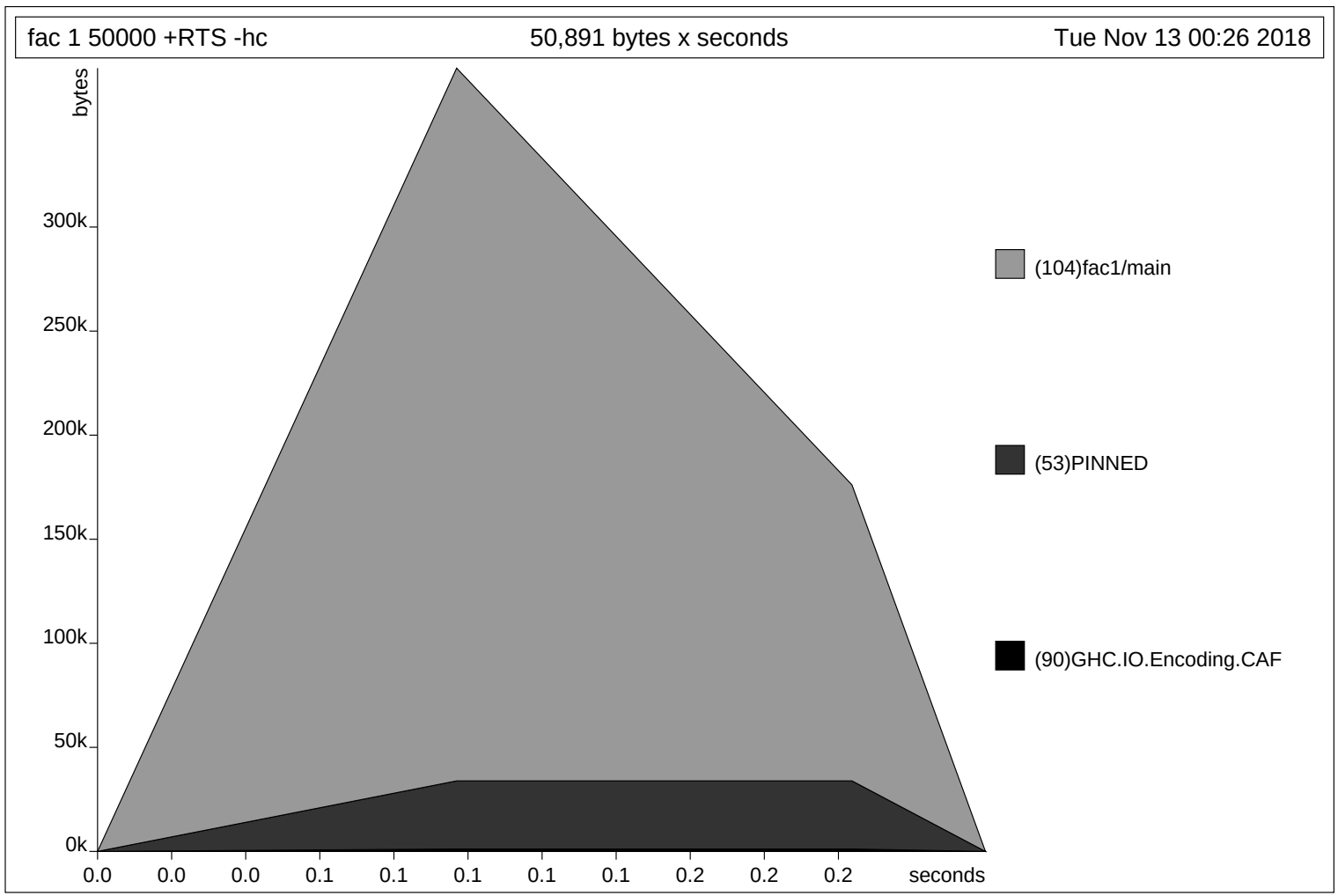
    ```
    f $! x = x `seq` f x
    ```
  - ghc macht Striktheitsanalyse
- Fakultät in konstantem Platzaufwand

```
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))
```

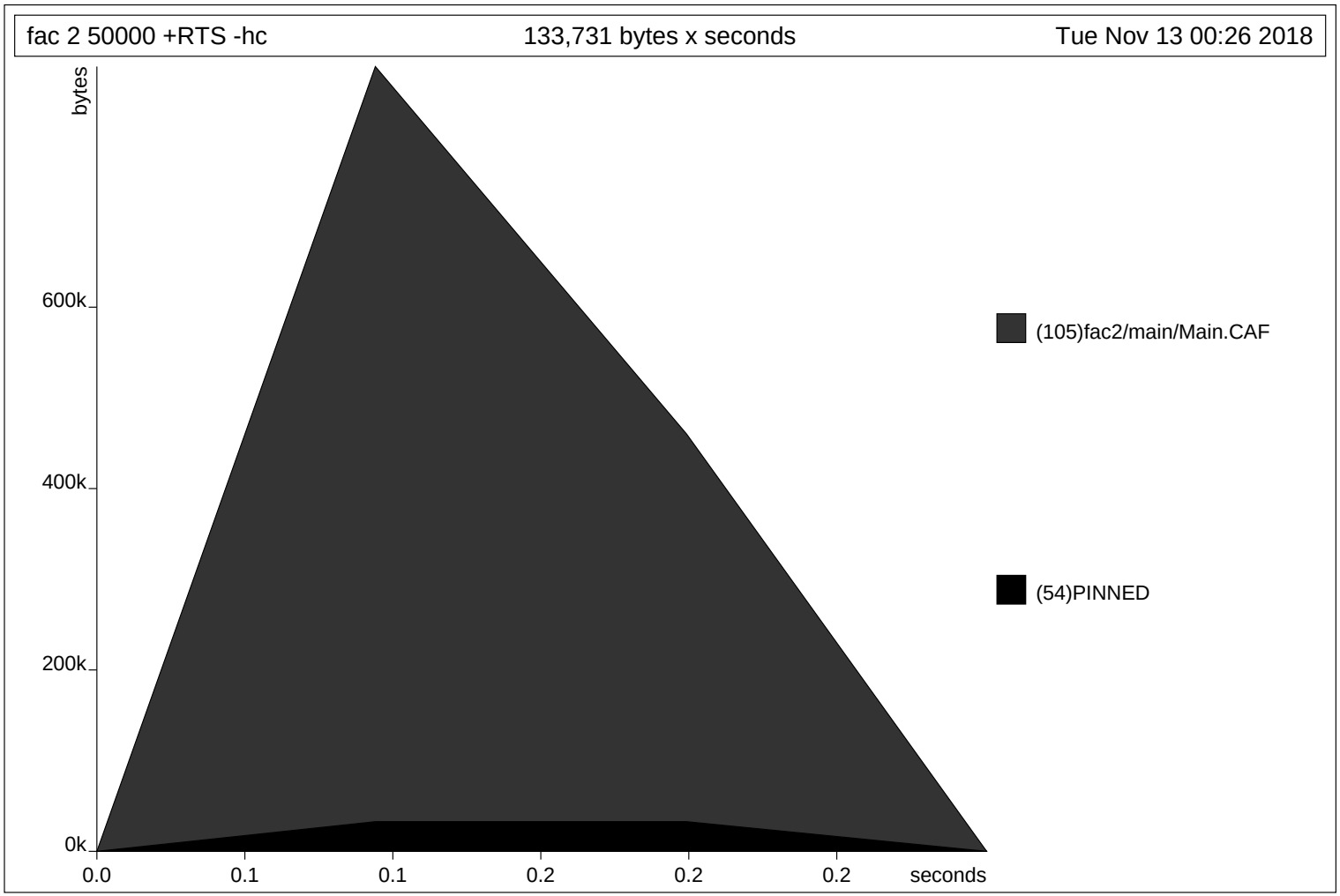
Speicherprofil: `fac1 50000`, nicht optimiert



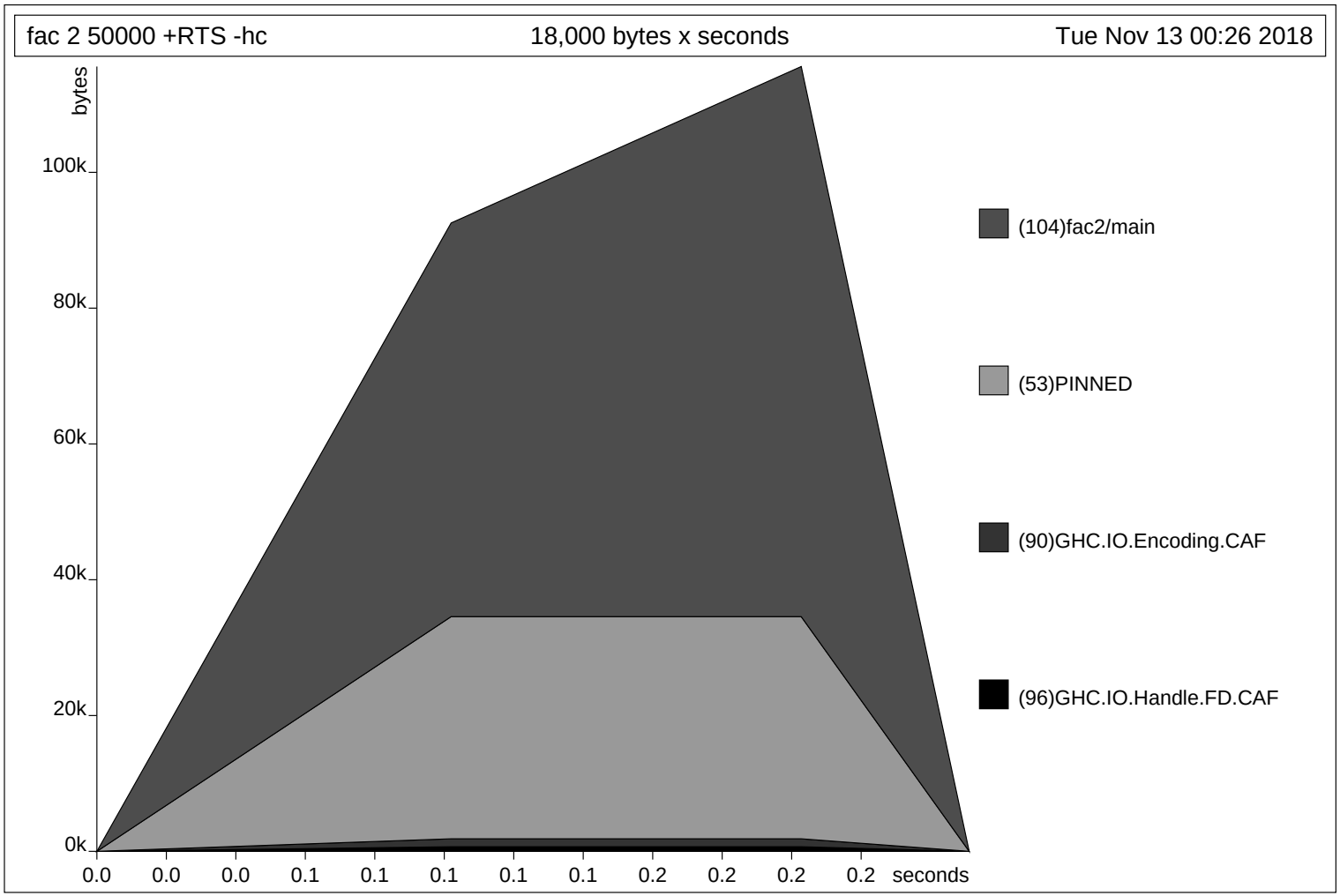
Speicherprofil: `fac1 50000`, optimiert



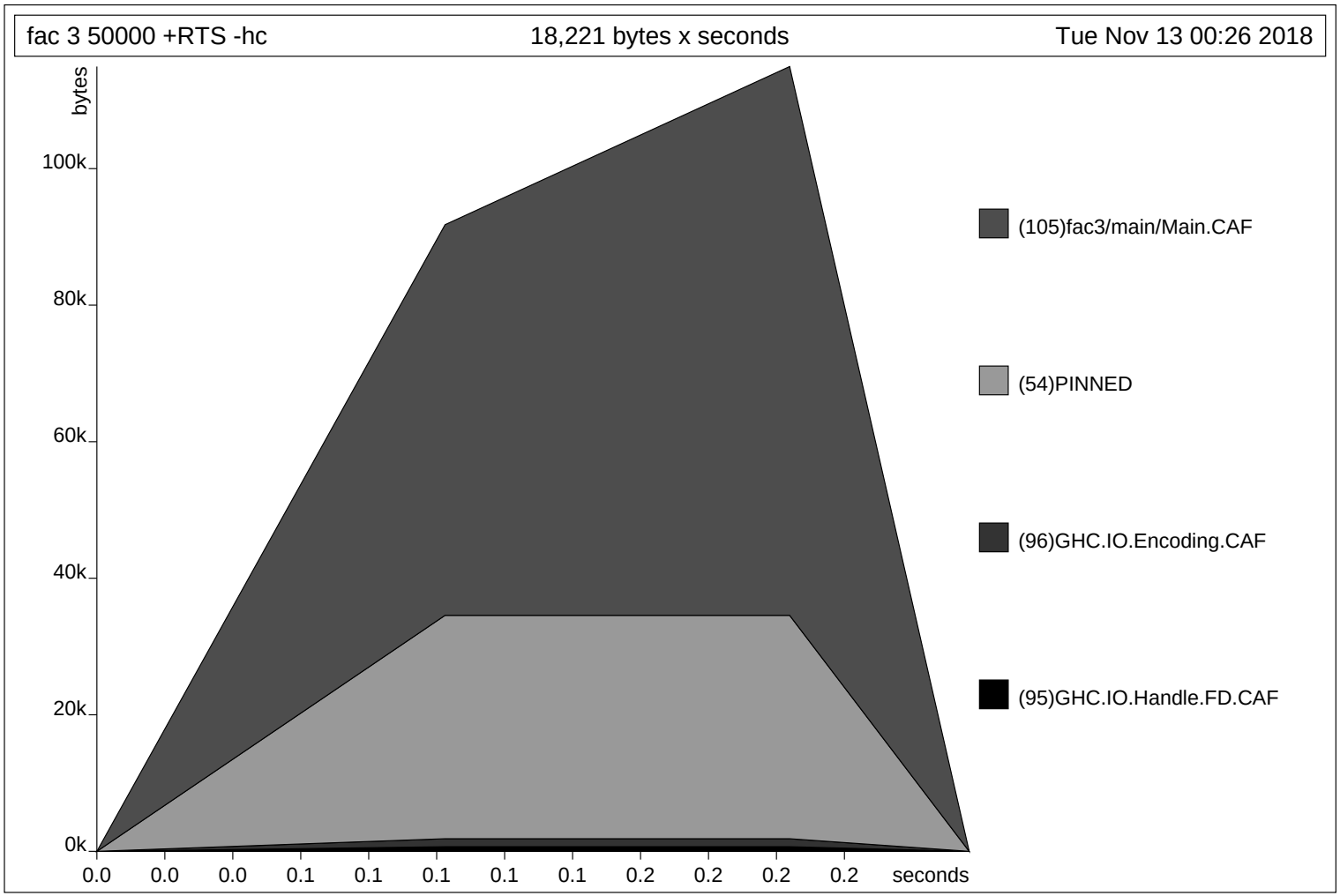
Speicherprofil: fac2 50000, nicht optimiert



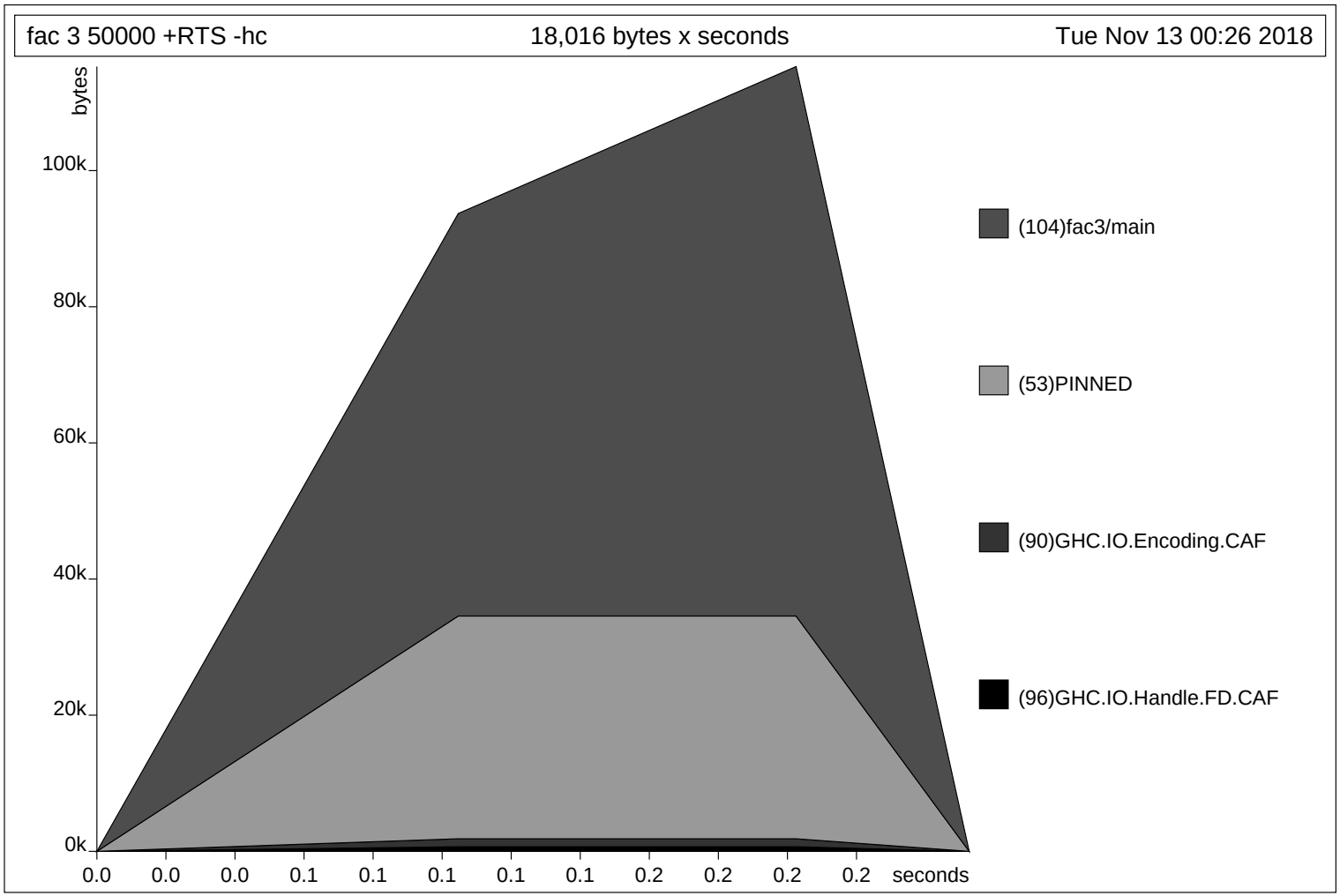
Speicherprofil: fac2 50000, optimiert



Speicherprofil: fac3 50000, nicht optimiert



Speicherprofil: fac3 50000, optimiert



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

Datentypen in anderen Programmiersprachen

Datentypen in C

- **C**: Produkte, Aufzählungen, keine rekursiven Typen
- Rekursion **nur** durch **Zeiger**

```
typedef struct list_t {
      int elem;
      struct list_t *next;
      } *list;
```

- Konstruktoren **nutzerimplementiert**

```
list cons(int hd, list tl)
{ list l;
  if ((l= (list)malloc(sizeof(struct list_t)))== NULL) {
    printf("Out of memory\n"); exit(-1);
  }
  l-> elem= hd; l-> next= tl;
  return l;
}
```

Polymorphie in anderen Programmiersprachen: C

- "Polymorphie" in C: **void** $*$

```
typedef struct list_t {
    void *head;
    struct list_t *next;
    } *list;
```

- Gegeben:

```
int x = 7;
struct list_t l1  = { &x, NULL};
```

- l2.head hat Typ **void** $*$:

```
int y;
y = *(int *)l1.head;
```

- Nicht möglich: head direkt als Skalar (e.g. int)
- C++: Templates

Datentypen in Java

- Nachbildung durch Klassen, z.B. für Listen:

```
class List {
  public List(Object el, List tl) {
    this.elem= el;
    this.next= tl;
  }
  public Object elem;
  public List   next;
```

- Länge (iterativ):

```
  int length() {
    int i= 0;
    for (List cur= this; cur != null; cur= cur.next)
      i++;
    return i;
  }
```

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

```
class AbsList<T> {
  public AbsList(T el, AbsList<T> tl) {
    this.elem= el;
    this.next= tl;
  }
  public T elem;
  public AbsList<T> next;
```

Polymorphie in Java

- Typkorrekte Konkatenenation:

```
void concat(AbsList<T> o)
{
  AbsList<T> cur= this;
  while (cur.next != null) cur= cur.next;
  cur.next= o;
}
```

- **Nachteil:** Benutzung umständlich, weil keine Typherleitung

```
AbsList<Integer> l=
  new AbsList<Integer>(new Integer(1),
    new AbsList<Integer>(new Integer(2), null));
```

- **Vorteil:** Typkorrektheit sichergestellt:

```
AbsList<Character> l3 =
    new AbsList<Character>(new Character('a'), null);
l.concat(l3); // Does not work
```

Ad-Hoc Polymorphie in Java

- **interface** und **abstract class**
- Flexibler in Java: beliebig viele Parameter etc.
- Eingeschränkt durch Vererbungshierarchie
- Ähnliche Standardklassen
  - toString
  - equals und ==, keine abgeleitete strukturelle Gleichheit

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

Polymorphie in Pyton

- 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

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 6 vom 20.11.2018: Funktionen Höherer Ordnung I

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Fahrplan

- **Teil I: Funktionale Programmierung im Kleinen**
  - Einführung
  - Funktionen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Zyklische Datenstrukturen
  - Funktionen höherer Ordnung I
  - 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

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 Berechungsmuster**
- Kontrollabstraktion

Ähnliche Datentypen der letzten Vorlesung

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

```
data Einkaufswagen = LeererWagen
                   | Einkauf Artikel Menge Einkaufswagen
```

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

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

Ähnliche Datentypen der letzten Vorlesung

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

```
data Eink
```

Gelöst durch Polymorphie

```
swagen
```

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

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

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufswagen→ Int
kasse LeererWagen = 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 :: String→ Int
length Empty    = 0
length (c :+ s) = 1+ length s
```

Gemeinsamkeiten:

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

Ähnliche Funktionen der letzten Vorlesung

```
kasse :: Einkaufswagen→ Int
kasse LeererWagen = 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 :: String→ Int
length Empty    = 0
length (c :+ s) = 1+ length s
```

Gemeinsamkeiten:

- ein Fall pro Konstruktor
- **linearer** rekursiver Aufruf
- **Nicht** durch Polymorphie gelöst (keine Instanz **einer** Definition)

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 angewant 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 angewant 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 angewant wird: $[\alpha]$
- Was ist der Ergebnistyp?
  - Eine Liste von Elementen aus dem Wertebereich von f: $[\beta]$
- Alles **zusammengesetzt**:

```
map :: (α → β) → [α] → [β]
```

Map und Filter

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
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 :: (α→ β)→ [α]→ [β]
map f []     = []
map f (c:cs) = f c : map f cs
```

- Auswertung:
  toL "AB" → map toLower ('A':'B':[])

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
map f []     = []
map f (c:cs) = f c : map f cs
```

- Auswertung:

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

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
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':[])
```

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
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 []
```

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
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 :: (α→ β)→ [α]→ [β]
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':[]
```

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
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 :: (α→ Bool)→ [α]→ [α]
```

- Definition

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

- Beispiel:

```
letters :: String→ String
letters = filter isAlpha
```

Beispiel `filter`: Sieb des Erathostenes

- Für jede gefundene Primzahl p alle Vielfachen heraussieben:

```
sieve (p:ps) = p: sieve (filter ps) where
   filter (q: qs)
     | q `mod` p ≠ 0 = q: filter qs
     | otherwise     = filter qs
```

Beispiel `filter`: Sieb des Erathostenes

- Für jede gefundene Primzahl p alle Vielfachen heraussieben:

```
sieve (p:ps) = p: sieve (filter ps) where
   filter (q: qs)
     | q `mod` p ≠ 0 = q: filter qs
     | otherwise     = filter qs
```

- Einfacher mit `filter`
- Es wird gefiltert mit `mod q p ≠ 0` (Funktionsparameter `q`)

Beispiel `filter`: Sieb des Erathostenes

- Für jede gefundene Primzahl p alle Vielfachen heraussieben:

```
sieve (p:ps) = p: sieve (filter ps) where
   filter (q: qs)
     | q `mod` p ≠ 0 = q: filter qs
     | otherwise     = filter qs
```

- Einfacher mit `filter`
- Es wird gefiltert mit `mod q p ≠ 0` (Funktionsparameter `q`)
- **Namenlose** (anonyme) Funktion `λq→ mod q p ≠ 0`

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

Funktionen Höherer Ordnung

Funktionen als Argumente: Funktionskomposition

- **Funktionskomposition** (mathematisch)

```
(∘) :: (β→ γ) → (α→ β)→ α→ γ
(f ∘ g) x = f (g x)
```

  - Vordefiniert
  - Lies: f nach g

- Funktionskomposition **vorwärts**:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(f >.> g) x = g (f x)
```

  - **Nicht** vordefiniert

η-Kontraktion

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

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

η-Kontraktion

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

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(>.>) = flip (∘)
```

- **Da fehlt doch was?!**

η-Kontraktion

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

```
flip :: (α→ β→ γ)→ β→ α→ γ
flip f b a = f a b
```

- Damit Funktionskomposition vorwärts:

```
(>.>) :: (α→ β)→ (β→ γ)→ α→ γ
(>.>) = flip (∘)
```

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

```
(>.>) = flip (∘)   ≡   (>.>) f g a = flip (∘) f g a
```

- Warum?

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

Sei f eine Funktion $f : A \to B$, dann gilt $f = \lambda x.\, f\ x$

- In Haskell: **η-Kontraktion**
  - Bedingung: Ausdruck E :: $\alpha \to \beta$, Variable x :: α, E darf x nicht enthalten

$$\lambda x \to \ \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{E}$$

- Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$\mathsf{f}\ \mathsf{x} = \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{f} = \mathsf{E}$$

- Hier:

```
(>.>) f g a = flip (∘) f g a  ≡  (>.>) f g a = flip (∘) f g a
```

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

Sei f eine Funktion $f : A \to B$, dann gilt $f = \lambda x.\, f\ x$

- In Haskell: **η-Kontraktion**
  - Bedingung: Ausdruck E :: $\alpha \to \beta$, Variable x :: α, E darf x nicht enthalten

$$\lambda x \to \ \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{E}$$

- Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$\mathsf{f}\ \mathsf{x} = \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{f} = \mathsf{E}$$

- Hier:

$$(>.>)\ \mathsf{f}\ \mathsf{g}\ \mathsf{a} = \mathsf{flip}\ (\circ)\ \mathsf{f}\ \mathsf{g}\ \mathsf{a} \equiv (>.>)\ \mathsf{f}\ \mathsf{g} \quad = \mathsf{flip}\ (\circ)\ \mathsf{f}\ \mathsf{g}$$

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

Sei f eine Funktion $f : A \to B$, dann gilt $f = \lambda x.\, f\ x$

- In Haskell: **η-Kontraktion**
  - Bedingung: Ausdruck E :: $\alpha \to \beta$, Variable x :: α, E darf x nicht enthalten

$$\lambda x \to \ \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{E}$$

- Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$\mathsf{f}\ \mathsf{x} = \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{f} = \mathsf{E}$$

- Hier:

$$(>.>)\ \mathsf{f}\ \mathsf{g}\ \mathsf{a} = \mathsf{flip}\ (\circ)\ \mathsf{f}\ \mathsf{g}\ \mathsf{a} \equiv (>.>)\ \mathsf{f} \quad = \mathsf{flip}\ (\circ)\ \mathsf{f}$$

η-Äquivalenz und η-Kontraktion

η-Äquivalenz

Sei f eine Funktion $f : A \to B$, dann gilt $f = \lambda x.\, f\ x$

- In Haskell: **η-Kontraktion**
  - Bedingung: Ausdruck E :: $\alpha \to \beta$, Variable x :: α, E darf x nicht enthalten

$$\lambda x \to \ \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{E}$$

- Spezialfall Funktionsdefinition (**punktfreie** Notation)

$$\mathsf{f}\ \mathsf{x} = \mathsf{E}\ \mathsf{x} \quad \equiv \quad \mathsf{f} = \mathsf{E}$$

- Hier:

$$(>.>)\ \mathsf{f}\ \mathsf{g}\ \mathsf{a} = \mathsf{flip}\ (\circ)\ \mathsf{f}\ \mathsf{g}\ \mathsf{a} \equiv (>.>) \qquad = \mathsf{flip}\ (\circ)$$

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:

$$\mathsf{f\ a\ b} \equiv (\mathsf{f\ a})\ \mathsf{b}$$

  - **Inbesondere**: $\mathsf{f\ (a\ b)} \neq (\mathsf{f\ a})\ \mathsf{b}$

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$\alpha \to \beta \to \gamma \equiv \alpha \to (\beta \to \gamma)$$

  - **Inbesondere**: $(\alpha \to \beta) \to \gamma \neq \alpha \to (\beta \to \gamma)$

- Funktionsanwendung ist linksassoziativ:

$$\mathsf{f\ a\ b} \equiv (\mathsf{f\ a})\ \mathsf{b}$$

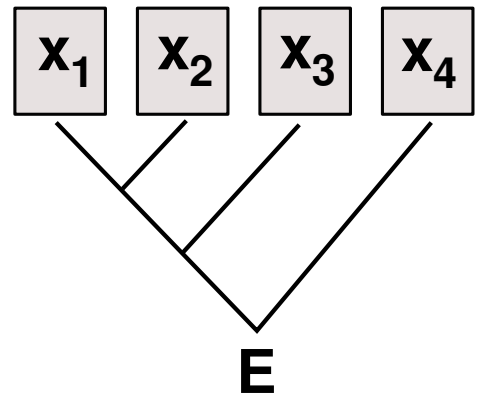
  - **Inbesondere**: $\mathsf{f\ (a\ b)} \neq \mathsf{(f\ a)\ b}$

- **Partielle** Anwendung von Funktionen:
  - Für $\mathsf{f} :: \alpha \to \beta \to \gamma$, $\mathsf{x} :: \alpha$ ist $\mathsf{f\ x} :: \beta \to \gamma$
- Beispiele:
  - $\mathsf{map\ toLower} :: \mathsf{String} \to \mathsf{String}$
  - $(3 ==) :: \mathsf{Int} \to \mathsf{Bool}$
  - $\mathsf{concat} \circ \mathsf{map\ (replicate\ 2)} :: \mathsf{String} \to \mathsf{String}$

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)



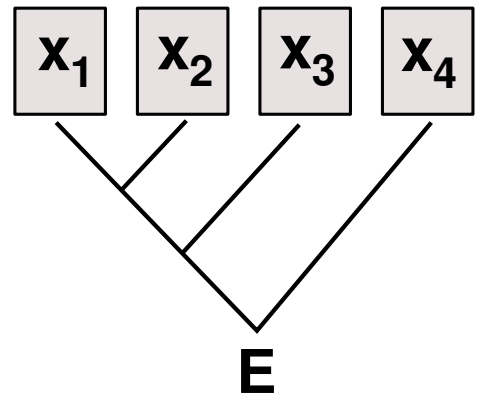
- Beispiel: kasse, inventur, sum, concat, length, (++), ...

- 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)



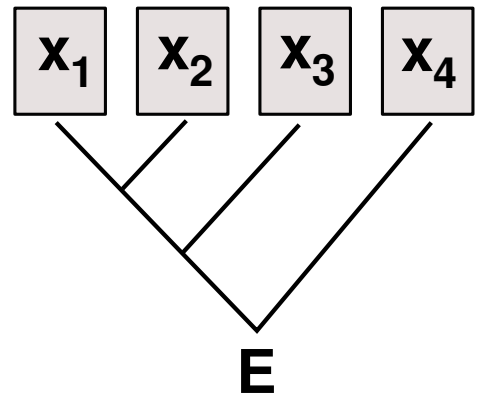
- Beispiel: kasse, inventur, sum, concat, length, (++), ...

- Auswertung:

$$
\begin{array}{lll}
\text{sum } [4,7,3] & \rightarrow & 4 + 7 + 3 + 0 \\
\text{concat } [A,\ B,\ C] & \rightarrow & \\
\text{length } [4,\ 5,\ 6] & \rightarrow &
\end{array}
$$

Strukturelle Rekursion

- **Strukturelle Rekursion**: gegeben durch
  - eine Gleichung für die leere Liste
  - eine Gleichung für die nicht-leere Liste
    (mit **einem** rekursiven Aufruf)



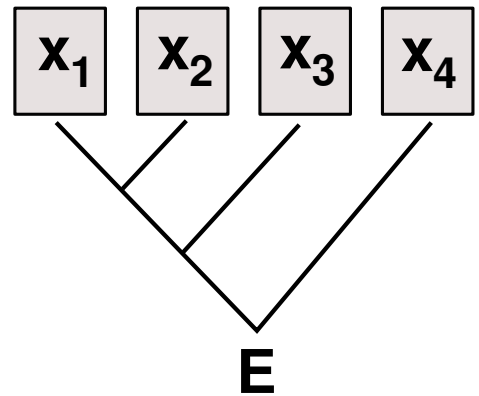
- Beispiel: kasse, inventur, sum, concat, length, $(+\!\!+)$, ...

- Auswertung:

$$
\begin{array}{lcl}
\text{sum } [4,7,3] & \rightarrow & 4 + 7 + 3 + 0 \\
\text{concat } [A,\ B,\ C] & \rightarrow & A +\!\!+ B +\!\!+ C +\!\!+ [\,] \\
\text{length } [4,\ 5,\ 6] & \rightarrow &
\end{array}
$$

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:

$$
\begin{array}{lcl}
\text{sum } [4,7,3] & \rightarrow & 4+7+3+0 \\
\text{concat } [A,\ B,\ C] & \rightarrow & A +\!\!+ B +\!\!+ C +\!\!+ [\,] \\
\text{length } [4,\ 5,\ 6] & \rightarrow & 1+1+1+0
\end{array}
$$

Strukturelle Rekursion

- **Allgemeines Muster**:

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

- Parameter der Definition:
  - Startwert (für die leere Liste) e :: β
  - Rekursionsfunktion $\otimes$ :: $\alpha \to \beta \to \beta$
- Auswertung:

$$f\ [x1, \ldots, xn] = x1 \otimes x2 \otimes \ldots \otimes xn \otimes e$$

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

Strukturelle Rekursion durch `foldr`

- **Stukturelle** Rekursion
  - Basisfall: leere Liste
  - Rekursionsfall: Kombination aus Listenkopf und Rekursionswert
- Signatur

```
foldr :: (α → β → β) → β → [α] → β
```

- 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 :: (α→ Bool)→ [α] → Bool
all p = and ∘ map p
```

Der Shoppe, revisited.

- Kasse alt:

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

- Kasse neu:

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

Besser:

```
kasse :: Einkaufswagen→ Int
kasse (Ekwg 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 :: Einkaufswagen→ String
kassenbon ew@(Ekwg ps) =
  "Bob's␣Aulde␣Grocery␣Shoppe\n\n"++
  "Artikel␣␣␣␣␣␣␣␣␣␣␣␣␣␣Menge␣␣␣␣␣␣Preis\n"++
  "————————————————————————————————————\n"++
  concatMap artikel ps ++
  "════════════════════════════════════\n"++
  "Summe:"++ formatR 31 (showEuro (kasse ew))
```

```
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:

  $$\texttt{foldr}\ \otimes\ [x1, \ldots, xn]\ e = x1 \otimes x2\ (x2 \otimes (\ldots (xn \otimes e)))$$

- Warum nicht **andersherum**?

Iteration mit foldl

- foldr faltet von rechts:
  $$\texttt{foldr}\ \otimes\ [x1, \ldots, xn]\ e = x1 \otimes x2\ (x2 \otimes (\ldots (xn \otimes e)))$$
- Warum nicht **andersherum**?
  $$\texttt{foldl}\ \otimes\ [x1, \ldots, xn]\ e = (((e \otimes x1) \otimes x2) \ldots) \otimes xn$$
- Definition von foldl:

```
foldl :: (α → β → α) → α → [β] → α
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)

foldr vs. foldl

- f = foldr ⊗ e entspricht

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

  - **Nicht-strikt** in xs, z.B. and, or
  - Konsumiert nicht immer die ganze Liste
  - Auch für unendliche Listen anwendbar

- f = foldl ⊗ e entspricht

```
f xs = g e xs where
  g a []     = a
  g a (x:xs) = g (a ⊗ x) xs
```

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

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 $O(n)$!

Wann ist foldl = foldr?

Definition (Monoid)

$(\otimes, A)$ ist ein **Monoid** wenn

$$A \otimes x = x \qquad \text{(Neutrales Element links)}$$

$$x \otimes A = x \qquad \text{(Neutrales Element rechts)}$$

$$(x \otimes y) \otimes z = x \otimes (y \otimes z) \qquad \text{(Assoziativät)}$$

Theorem

*Wenn $(\otimes, A)$ **Monoid**, dann für alle A, xs*

$$foldl \otimes A\ xs = foldr \otimes A\ xs$$

- Beispiele: `length`, `concat`, `sum`
- Gegenbeispiele: `rev`, `all`

Übersicht: vordefinierte Funktionen auf Listen II

```
map       :: (α→ β)→ [α]→ [β]              — Auf alle anwenden
filter    :: (α→ Bool)→ [α]→ [α]           — Elemente filtern
foldr     :: (α→ β→ β)→ β→ [α]→ β          — Falten von rechts
foldl     :: (β→ α→ β)→ β→ [α]→ β          — Falten von links
mapConcat :: (α→ [β])→ [α]→ [β]            — map und concat
takeWhile :: (α→ Bool)→ [α]→ [α]           — längster Prefix mit p
dropWhile :: (α→ Bool)→ [α]→ [α]           — Rest von takeWhile
span      :: (α→ Bool)→ [α]→ ([α], [α])    — takeWhile und dropWhile
all       :: (α → Bool) → [α] → Bool       — p gilt für alle
any       :: (α → Bool) → [α] → Bool       — p gilt mind. einmal
elem      :: (Eq α) ⇒ α → [α] → Bool       — Ist Element enthalten?
zipWith   :: (α → β → γ) → [α] → [β] → [γ]
                                           — verallgemeinertes zip
```

- Mehr: siehe Data.List

Zusammenfassung

- Funktionen **höherer Ordnung**
  - Funktionen als gleichberechtigte Objekte und Argumente
  - Partielle Applikation, η-Kontraktion, namenlose Funktionen
  - Spezielle Funktionen höherer Ordnung: map, `filter`, `fold` und Freunde
- Formen der **Rekursion**:
  - Strukturelle Rekursion entspricht `foldr`
  - Iteration entspricht `foldl`
- Nächste Woche: `fold` für andere Datentypen

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 7 vom 27.11.2018: Funktionen Höherer Ordnung II: Jenseits der Liste

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Organisatorisches

- Diese Woche **zwei** Übungsblätter.

Organisatorisches

- Diese Woche **zwei** Übungsblätter.
- Ein Bonusübungsblatt für diese Woche.
- Das erste Gruppenübungsblatt — ab **nächste** Woche zwei Wochen.
- Nächste Woche **Tag der Lehre** — Mittwochstutorien fallen aus.

Fahrplan

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

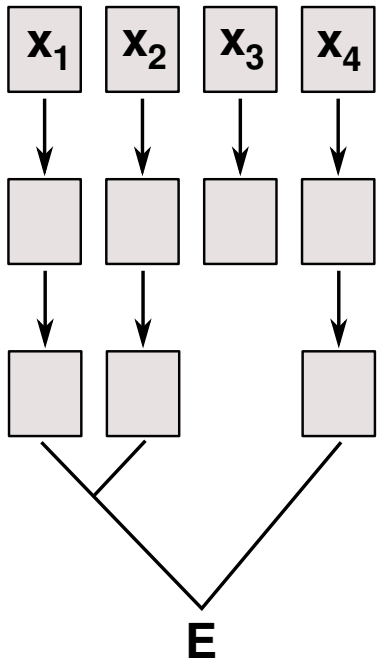
Heute

- Mehr über `map` und `fold`
- `map` und `fold` sind nicht nur für Listen
- Funktionen höherer Ordnung in anderen Programmiersprachen

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 | x<- as, g x ] ≡ map f (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 ]
```

  - Anderes Beispiel:

```
digits str = [ord x -ord '0' | x<- str, isDigit x]
```

Listenkomprehension mit mehreren Generatoren

- 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

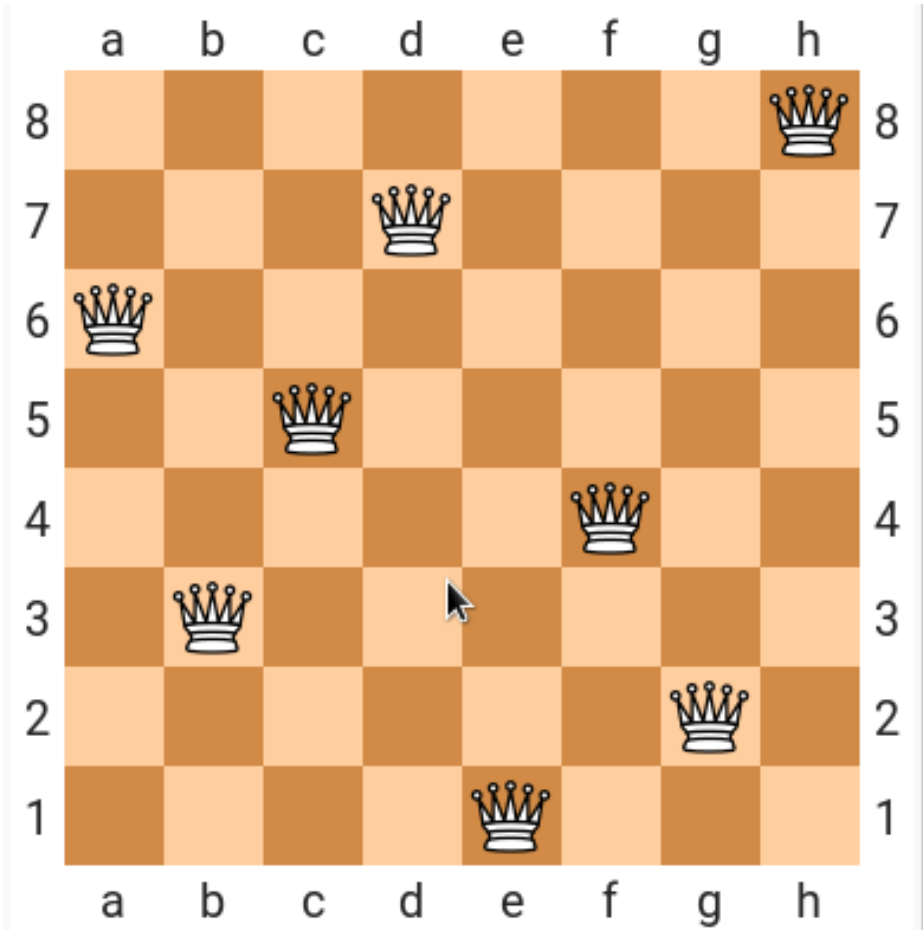
- Quicksort per Listenkomprehension:

```
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: plaziere 8 Damen sicher auf einem Schachbrett



Source: wikipedia

Beispiel II: n-Damen-Problem

- Spezifikation: Position der Königinnen, Hauptfunktion:

```
type Pos = (Int, Int)
type Board = [Pos]
```

- Rekursive Lösung:
  - Lösung für $n-1$ Königinnen, n-te sicher dazu positionieren
  - Invariante: n-te Königin in n-ter Spalte

```
queens :: Int→ [Board]
queens n = qu n where
  qu :: Int→ [Board]
  qu i | i == 0  = [[]]
       | otherwise =
     [ p++ [(i, j)] | p ← qu (i-1), j ← [1.. n],
                      safe p (i, j)]
```

```
safe :: Board→ Pos→ Bool
```

Map und Fold: Jenseits der Listen

map als strukturerhaltende Abbildung

map ist die kanonische **strukturerhaltende Abbildung**

- Für map gelten folgende Aussagen:

$$\text{map}\ \text{id} = \text{id}$$

$$\text{map}\ f \circ \text{map}\ g = \text{map}\ (f \circ g)$$

$$\text{length} \circ \text{map}\ f = \text{length}$$

- Was davon ist spezifisch für Listen?
- Wie können wir das verallgemeineren?

map als strukturerhaltende Abbildung

map ist die kanonische **strukturerhaltende Abbildung**

- Für map gelten folgende Aussagen:

$$\texttt{map id} = \texttt{id}$$

$$\texttt{map f} \circ \texttt{map g} = \texttt{map (f} \circ \texttt{g)}$$

$$\texttt{length} \circ \texttt{map f} = \texttt{length}$$

- Was davon ist spezifisch für Listen?
- Wie können wir das verallgemeineren?

$$\longrightarrow \text{Typklassen?}$$

map als strukturerhaltende Abbildung

map ist die kanonische **strukturerhaltende Abbildung**

- Für map gelten folgende Aussagen:

$$\mathsf{map}\ \mathsf{id} = \mathsf{id}$$

$$\mathsf{map}\ \mathsf{f} \circ \mathsf{map}\ \mathsf{g} = \mathsf{map}\ (\mathsf{f} \circ \mathsf{g})$$

$$\mathsf{length} \circ \mathsf{map}\ \mathsf{f} = \mathsf{length}$$

- Was davon ist spezifisch für Listen?
- Wie können wir das verallgemeineren?

$\longrightarrow$ Konstruktorklassen!

Funktoren

- **Konstruktorklassen** sind Typklassen für Typkonstruktoren.
- Die Konstruktorklasse Functor für alle Typen mit einer stukturerhaltenden 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

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`
- Es gilt: `foldr (:) [] = id`
- Jeder algebraischer Datentyp hat ein `foldr`
- 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 Funktionsparameter f_C auf Argumente an
- Beispiel:

```
data IL = Cons Int IL | Err String | Mt

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
```

fold für bekannte Datentypen

- Bool:

fold für bekannte Datentypen

- Bool: Fallunterscheidung:

```
data Bool = False | True

foldBool :: β→ β→ Bool→ β
foldBool a1 a2 False = a1
foldBool a1 a2 True  = a2
```

- Maybe α:

fold für bekannte Datentypen

- Bool: Fallunterscheidung:

```
data Bool = False | True

foldBool :: β→ β→ Bool→ β
foldBool a1 a2 False = a1
foldBool a1 a2 True  = a2
```

- Maybe α: Auswertung

```
data Maybe α = Nothing | Just α

foldMaybe :: β→ (α→ β)→ Maybe α→ β
foldMaybe b f Nothing  = b
foldMaybe b f (Just a) = f a
```

  - Als maybe vordefiniert

fold für bekannte Datentypen

- Tupel:

fold für bekannte Datentypen

- Tupel: die uncurry-Funktion

```
foldPair :: (α→ β→ γ)→ (α, β)→ γ
foldPair f (a, b)= f a b
```

- Natürliche Zahlen:

fold für bekannte Datentypen

- Tupel: die uncurry-Funktion

```
foldPair :: (α→ β→ γ)→ (α, β)→ γ
foldPair f (a, b)= f a b
```

- Natürliche Zahlen: Iterator

```
data Nat = Zero | Succ Nat

foldNat :: β→ (β→ β)→ Nat→ β
foldNat e f Zero  = e
foldNat e f (Succ n) = f (foldNat e f n)
```

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 :: (α→ β→ β→ β)→ β→ Tree α→ β
foldT f e Mt = e
foldT f e (Node a l r) = f a (foldT f e l) (foldT f e r)
```

- Instanz von `Functor`, kein (offensichtliches) Filter

```
instance Functor Tree where
  fmap :: (α→ β)→ Tree α→ Tree β
  fmap f Mt = Mt
  fmap f (Node a l r)= Node (f a) (fmap f l) (fmap f r)
```

Funktionen mit foldT und mapT

- Höhe des Baumes berechnen:

```
height :: Tree α→ Int
height = foldT (λ_ l r→ 1+max l r) 0
```

- Inorder-Traversion der Knoten:

```
inorder :: Tree α→ [α]
inorder = foldT (λa l r→ l++ [a]++ r) []
```

Kanonische Eigenschaften von foldT und mapT

- Auch hier gilt:

$$\mathsf{foldT}\ \mathsf{Node}\ \mathsf{Mt} = \mathsf{id}$$

$$\mathsf{mapT}\ \mathsf{id} = \mathsf{id}$$

$$\mathsf{mapT}\ f \circ \mathsf{mapT}\ g = \mathsf{mapT}\ (f \circ g)$$

Das Labyrinth

- Das Labyrinth als variadischer Baum:

```
data VTree α = Node α [VTree α]
```

```
type Lab α = VTree α
```

- Auch hierfür foldT und mapT:

Das Labyrinth

- Das Labyrinth als variadischer Baum:

```
data VTree α = Node α [VTree α]
```

```
type Lab α = VTree α
```

- Auch hierfür foldT und mapT:

```
foldT :: (α→ [β]→ β)→ VTree α→ β
foldT f (Node a ns) = f a (map (foldT f) ns)
```

```
mapT :: (α→ β)→ VTree α→ VTree β
mapT f (Node a ns) = Node (f a) (map (mapT f) ns)
```

Suche im Labyrinth

- Tiefensuche via foldT

```
dfts' :: Lab α→ [Path α]
dfts' = foldT add where
  add a [] = [[a]]
  add a ps = concatMap (map (a :)) ps
```

Suche im Labyrinth

- Tiefensuche via foldT

```
dfts' :: Lab α→ [Path α]
dfts' = foldT add where
  add a [] = [[a]]
  add a ps = concatMap (map (a :)) ps
```

- 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

- Andere rekursive Struktur über Listen
  - Quicksort: baumartige Rekursion
- Rekursion nicht über Listenstruktur:
  - take: Rekursion über take

```
take :: Int→ [α]→ [α]
take n _      | n ≤ 0 =  []
take _ []             =  []
take n (x:xs)         =  x : take (n−1) xs
```

  - Version mit foldr divergiert für nicht-endliche Listen

Funktionen Höherer Ordnung in anderen Sprachen

C

- Implizit vorhanden: Funktionen = Zeiger auf Funktionen

```
extern list filter(int f(void *x), list l);

extern list map1(void *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 *));
```

C

- Implementierung von map
- Rekursiv, erzeugt neue Liste:

```
list map1(void *f(void *x), list l)
{
  return l== NULL ?
         NULL : cons(f(l-> elem), map1(f, l-> next));
}
```

- Iterativ, Liste wird in-place geändert (**Speicherleck**):

```
list map2(void *f(void *x), list l)
{
  list c;
  for (c= l; c!= NULL; c= c-> next) {
    c-> elem= f(c-> elem); }
  return l;
}
```

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)

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

- map, `filter`, `fold` sind ein nützliches, skalierbares und allgemeines **Berechnungsmuster**.
- Listenkomprehensionen sind nützlicher syntaktischer Zucker.
- map und `fold` sind **kanonische Funktionen höherer Ordnung**, und für alle Datentypen definierbar.
- Nächste Woche: Funktionale Programmierung im Großen — Abstrakte Datentypen

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 8 vom 04.12.2018: Abstrakte Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Organisatorisches

- Morgen ist **Tag der Lehre**
  - Mittwochs-Tutorien **fallen aus**
  - Donnerstags-Tutorien finden statt.
- Abgabe des 8. Übungsblattes in Gruppen zu **drei** Studenten.
  - Bitte **jetzt** eine Gruppe suchen!
- Klausurtermine:
  - Übungsklausur: 17.12.2018 10– 12
  - Hauptklausur: 08.03.2019 10– 14

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

Inhalt

- **Abstrakte Datentypen**
  - Allgemeine Einführung
  - Realisierung in Haskell
  - Beispiele

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:

- Werte des Typen können nur über die bereitgestellten Operationen erzeugt werden
- Eigenschaften von Werten des Typen werden nur über die bereitgestellten Operationen beobachtet
- 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**
  - Keine Einschränkungen
  - Insbesondere keine Gleichheiten ($[\,] \neq x : xs$, $x : ls \neq y : ls$ etc.)
- ADTs:
  - Einschränkungen und Invarianten möglich
  - Gleichheiten möglich

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, ..., cn), T(..)
  - **Klassen**: C, C(f1, ..., fn), 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 Shoppe4 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 Kaesesorte = Gouda | Appenzeller
                deriving (Eq, Show)

kpreis :: Kaesesorte → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

data Bio = Bio | Konv
           deriving (Eq, Show)

data Artikel =
    Apfel Apfelsorte  | Eier
  | Kaese Kaesesorte  | Schinken
  | Salami          | Milch Bio
  deriving (Eq, Show)

data Menge = Stueck Int | Gramm  Int | Liter  Double
           deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel → Menge→ Preis
preis (Apfel a) (Stueck n) = Just (n∗ apreis a)
preis Eier (Stueck n)     = Just (n∗ 20)
preis (Kaese k)(Gramm g) = Just (round(fromIntegral g∗ 1000∗ kpreis k))
preis Schinken (Gramm g) = Just (div (g∗ 199) 100)
preis Salami (Gramm g)    = Just (div (g∗ 159) 100)
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 (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)

-- 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 [ m | Posten la m ← ps,  la == a ]

einlagern :: Artikel→ Menge→ Lager→ Lager
einlagern a m (Lager ps) =
  let hinein a m [] = [Posten a m]
      hinein a m (Posten al ml: l)
        | a == al   = (Posten a (addiere m ml): l)
        | otherwise = (Posten al ml: hinein a m l)
  in case preis a m of
       Nothing → Lager ps
       _ → Lager (hinein a m ps)

data Einkaufswagen = Ekwg [Posten]
                     deriving (Eq, Show)

leererWagen :: Einkaufswagen
leererWagen = Ekwg []

einkauf :: Artikel→ Menge→ Einkaufswagen→ Einkaufswagen
einkauf a m (Ekwg ps)
  | isJust (preis a m) = Ekwg (Posten a m: ps)
  | otherwise          = Ekwg ps

kasse :: Einkaufswagen→ Int
kasse (Ekwg ps) = sum (map cent ps)

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

artikel :: Posten→ String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
   formatR 7  (menge m) ++
   formatR 10 (showEuro (cent p)) ++ "\n"

menge :: Menge→ String
menge (Stueck n) = show n++ " St"
menge (Gramm g)  = show g++ " g."
menge (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 Shoppe4 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 Kaesesorte = Gouda | Appenzeller
                deriving (Eq, Show)

kpreis :: Kaesesorte → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

data Bio = Bio | Konv
         deriving (Eq, Show)

data Artikel =
    Apfel Apfelsorte  | Eier
  | Kaese Kaesesorte  | Schinken
  | Salami          | Milch Bio
  deriving (Eq, Show)

data Menge = Stueck Int | Gramm  Int | Liter  Double
           deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel → Menge→ Preis
preis (Apfel a) (Stueck n) = Just (n* apreis a)
preis Eier (Stueck n)     = Just (n* 20)
preis (Kaese k)(Gramm g) = Just (round(fromIntegral g* 1000* kpreis k))
preis Schinken (Gramm g) = Just (div (g* 199) 100)
preis Salami (Gramm g)    = Just (div (g* 159) 100)
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 (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)
```

Artikel

```
-- 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 [ m | Posten la m ← ps,  la == a ]

einlagern :: Artikel→ Menge→ Lager→ Lager
einlagern a m (Lager ps) =
  let hinein a m [] = [Posten a m]
      hinein a m (Posten al ml: l)
        | a == al   = (Posten a (addiere m ml): l)
        | otherwise = (Posten al ml: hinein a m l)
  in case preis a m of
       Nothing → Lager ps
       _ → Lager (hinein a m ps)

data Einkaufswagen = Ekwg [Posten]
                   deriving (Eq, Show)

leererWagen :: Einkaufswagen
leererWagen = Ekwg []

einkauf :: Artikel→ Menge→ Einkaufswagen→ Einkaufswagen
einkauf a m (Ekwg ps)
  | isJust (preis a m) = Ekwg (Posten a m: ps)
  | otherwise          = Ekwg ps

kasse :: Einkaufswagen→ Int
kasse (Ekwg ps) = sum (map cent ps)

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

artikel :: Posten→ String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
   formatR 7  (menge m) ++
   formatR 10 (showEuro (cent p)) ++ "\n"

menge :: Menge→ String
menge (Stueck n) = show n++ " St"
menge (Gramm g)  = show g++ " g."
menge (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 Shoppe4 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 Kaesesorte = Gouda | Appenzeller
                deriving (Eq, Show)

kpreis :: Kaesesorte → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

data Bio = Bio | Konv
         deriving (Eq, Show)

data Artikel =
    Apfel Apfelsorte  | Eier
  | Kaese Kaesesorte  | Schinken
  | Salami          | Milch Bio
  deriving (Eq, Show)

data Menge = Stueck Int | Gramm  Int | Liter  Double
           deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel → Menge→ Preis
preis (Apfel a) (Stueck n) = Just (n* apreis a)
preis Eier (Stueck n)     = Just (n* 20)
preis (Kaese k)(Gramm g) = Just (round(fromIntegral g* 1000* kpreis k))
preis Schinken (Gramm g) = Just (div (g* 199) 100)
preis Salami (Gramm g)   = Just (div (g* 159) 100)
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 (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)
```

Artikel

```
-- Posten:
data Posten = Posten Artikel Menge
            deriving (Eq, Show)

cent :: Posten→ Int
cent (Posten a m) = fromMaybe 0 (preis a m) -- gibt keinen Laufzeitfehler!
```

Posten

```
-- Lagerhaltung:
data Lager = Lager [Posten]
           deriving (Eq, Show)

leeresLager :: Lager
leeresLager = Lager []

suche :: Artikel→ Lager→ Maybe Menge
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 [] = [Posten a m]
      hinein a m (Posten al ml: l)
        | a == al   = (Posten a (addiere m ml): l)
        | otherwise = (Posten al ml: hinein a m l)
  in case preis a m of
       Nothing → Lager ps
       _ → Lager (hinein a m ps)

data Einkaufswagen = Ekwg [Posten]
                   deriving (Eq, Show)

leererWagen :: Einkaufswagen
leererWagen = Ekwg []

einkauf :: Artikel→ Menge→ Einkaufswagen→ Einkaufswagen
einkauf a m (Ekwg ps)
  | isJust (preis a m) = Ekwg (Posten a m: ps)
  | otherwise          = Ekwg ps

kasse :: Einkaufswagen→ Int
kasse (Ekwg ps) = sum (map cent ps)

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

artikel :: Posten→ String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
   formatR 7  (menge m) ++
   formatR 10 (showEuro (cent p)) ++ "\n"

menge :: Menge→ String
menge (Stueck n) = show n++ " St"
menge (Gramm g)  = show g++ " g."
menge (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 Shoppe4 where

import Data.Maybe
```

Artikel

```
-- Modellierung der Artikel.

data Apfelsorte = Boskoop | CoxOrange | GrannySmith
                deriving (Eq, Show)

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

data Kaesesorte = Gouda | Appenzeller
                deriving (Eq, Show)

kpreis :: Kaesesorte → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

data Bio = Bio | Konv
            deriving (Eq, Show)

data Artikel =
   Apfel Apfelsorte  | Eier
 | Kaese Kaesesorte  | Schinken
 | Salami          | Milch Bio
 deriving (Eq, Show)

data Menge = Stueck Int | Gramm  Int | Liter  Double
            deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel → Menge→ Preis
preis (Apfel a) (Stueck n) = Just (n∗ apreis a)
preis Eier (Stueck n)     = Just (n∗ 20)
preis (Kaese k)(Gramm g) = Just (round(fromIntegral g∗ 1000∗ kpreis k))
preis Schinken (Gramm g) = Just (div (g∗ 199) 100)
preis Salami (Gramm g)   = Just (div (g∗ 159) 100)
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 (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)
```

Posten

```
-- Posten:
data Posten = Posten Artikel Menge
              deriving (Eq, Show)

cent :: Posten→ Int
cent (Posten a m) = fromMaybe 0 (preis a m) -- gibt keinen Laufzeitfehler!
```

Lager

```
-- Lagerhaltung:
data Lager = Lager [Posten]
             deriving (Eq, Show)

leeresLager :: Lager
leeresLager = Lager []
```

Lager

```
suche :: Artikel→ Lager→ Maybe Menge
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 [] = [Posten a m]
      hinein a m (Posten al ml: l)
         | a == al   = (Posten a (addiere m ml): l)
         | otherwise = (Posten al ml: hinein a m l)
  in case preis a m of
       Nothing → Lager ps
       _ → Lager (hinein a m ps)
```

```
data Einkaufswagen = Ekwg [Posten]
                     deriving (Eq, Show)

leererWagen :: Einkaufswagen
leererWagen = Ekwg []

einkauf :: Artikel→ Menge→ Einkaufswagen→ Einkaufswagen
einkauf a m (Ekwg ps)
  | isJust (preis a m) = Ekwg (Posten a m: ps)
  | otherwise          = Ekwg ps

kasse :: Einkaufswagen→ Int
kasse (Ekwg ps) = sum (map cent ps)

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

artikel :: Posten→ String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
   formatR 7  (menge m) ++
   formatR 10 (showEuro (cent p)) ++ "\n"

menge :: Menge→ String
menge (Stueck n) = show n++ " St"
menge (Gramm g)  = show g++ " g."
menge (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 Shoppe4 where

import Data.Maybe
```

Artikel

```
-- Modellierung der Artikel.

data Apfelsorte = Boskoop | CoxOrange | GrannySmith
                deriving (Eq, Show)

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

data Kaesesorte = Gouda | Appenzeller
                deriving (Eq, Show)

kpreis :: Kaesesorte → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

data Bio = Bio | Konv
         deriving (Eq, Show)

data Artikel =
    Apfel Apfelsorte  | Eier
  | Kaese Kaesesorte  | Schinken
  | Salami          | Milch Bio
  deriving (Eq, Show)

data Menge = Stueck Int | Gramm  Int | Liter  Double
           deriving (Eq, Show)

type Preis = Maybe Int

preis :: Artikel → Menge→ Preis
preis (Apfel a) (Stueck n) = Just (n* apreis a)
preis Eier (Stueck n)     = Just (n* 20)
preis (Kaese k)(Gramm g) = Just (round(fromIntegral g* 1000* kpreis k))
preis Schinken (Gramm g) = Just (div (g* 199) 100)
preis Salami (Gramm g)   = Just (div (g* 159) 100)
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 (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)
```

Posten

```
-- Posten:
data Posten = Posten Artikel Menge
            deriving (Eq, Show)

cent :: Posten→ Int
cent (Posten a m) = fromMaybe 0 (preis a m) -- gibt keinen Laufzeitfehler!
```

Lager

```
-- Lagerhaltung:
data Lager = Lager [Posten]
           deriving (Eq, Show)

leeresLager :: Lager
leeresLager = Lager []
```

Lager

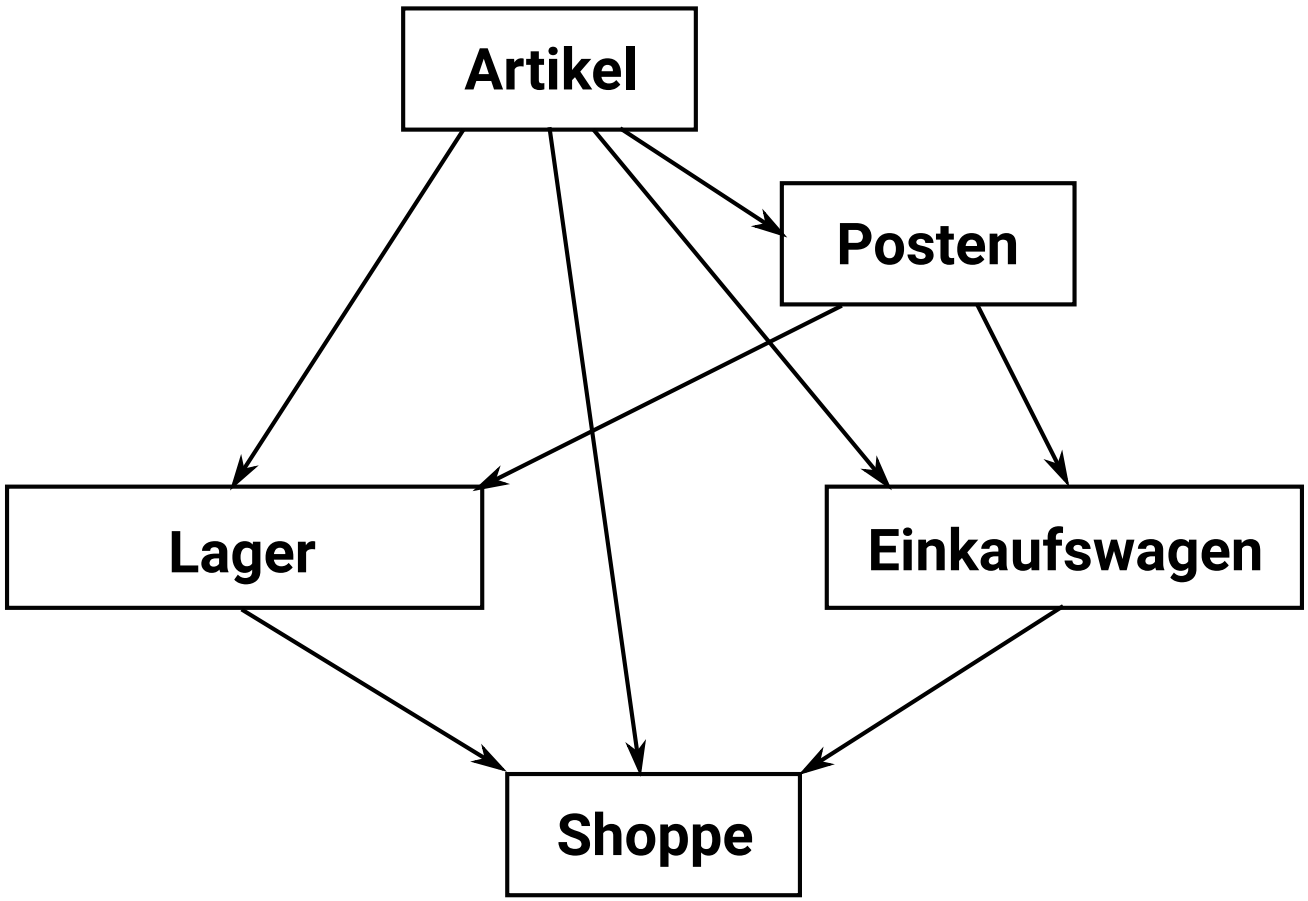
```
suche :: Artikel→ Lager→ Maybe Menge
suche a (Lager ps) =
   listToMaybe [ m | Posten la m ← ps,  la == a ]

einlagern :: Artikel→ Menge→ Lager→ Lager
einlagern a m (Lager ps) =
  let hinein a m [] = [Posten a m]
      hinein a m (Posten al ml: l)
        | a == al   = (Posten a (addiere m ml): l)
        | otherwise = (Posten al ml: hinein a m l)
  in case preis a m of
       Nothing → Lager ps
       _ → Lager (hinein a m ps)
```

Einkaufswagen

```
data Einkaufswagen = Ekwg [Posten]
                   deriving (Eq, Show)

leererWagen :: Einkaufswagen
leererWagen = Ekwg []

einkauf :: Artikel→ Menge→ Einkaufswagen→ Einkaufswagen
einkauf a m (Ekwg ps)
  | isJust (preis a m) = Ekwg (Posten a m: ps)
  | otherwise          = Ekwg ps

kasse :: Einkaufswagen→ Int
kasse (Ekwg ps) = sum (map cent ps)

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

artikel :: Posten→ String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
   formatR 7  (menge m) ++
   formatR 10 (showEuro (cent p)) ++ "\n"

menge :: Menge→ String
menge (Stueck n) = show n++ " St"
menge (Gramm g)  = show g++ " g."
menge (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)
```

Einkaufswagen

Refakturierung im Einkaufsparadies: Modularchitektur



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

```
module Posten(
  Posten,
  artikel,
  menge,
  posten,
  cent,
  hinzu) where
```

- Implementiert ADT Posten:

```
data Posten = Posten { artikel :: Artikel
                     , menge   :: Menge
                     }
```

  - Konstruktor wird **nicht** exportiert
- Garantierte 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 :: Artikel→ Lager→ Maybe Menge

liste :: Lager→ [(Artikel, Menge)]

inventur :: Lager→ Int
```

- Garantierte **Invariante**:
  - Lager enthält keine doppelten Artikel

Refakturierung IV: Einkaufswagen

```
module Einkaufswagen(
  Einkaufswagen,
  leererWagen,
  einkauf,
  kasse,
  kassenbon
  ) where
```

- Implementiert ADT Einkaufswagen

```
data Einkaufswagen = Ekwg [Posten]
```

- Garantierte 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

```
l0= leeresLager
l1= einlagern (Apfel Boskoop) (Stueck 1) l0
l2= einlagern Schinken (Gramm 50) l1
l3= einlagern (Milch Bio) (Liter 6) l2
l4= einlagern (Apfel Boskoop) (Stueck 4) l3
l5= einlagern (Milch Bio) (Liter 4) l4
l6= einlagern Schinken (Gramm 50) l5
```

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	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** ()	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 p =
   formatL 20 (show (P.artikel p)) ++
   formatR 7  (menge (P.menge p)) ++
   formatR 10 (showEuro (cent p)) ++ "\n"
```

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 $\alpha \Rightarrow$ $\alpha \rightarrow$ Map α $\beta \rightarrow$ Maybe β

  - Abbildung ändern:

    insert :: Ord $\alpha \Rightarrow$ $\alpha \rightarrow$ $\beta \rightarrow$ Map α $\beta \rightarrow$ Map α β

  - Abbildung löschen:

    delete :: Ord $\alpha \Rightarrow$ $\alpha \rightarrow$ Map α $\beta \rightarrow$ 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 :: Artikel→ Lager→ Maybe Menge
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**

Map als sortierte Assoziativliste

```
data Map α β = Map { toList :: [(α, β)] }
```

- Einfache Implementierung:

```
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
```

- Zusatzfunktionalität:
  - Iteration (Selektor `toList`)
  - Instanzen von Eq und Show (abgeleitet)
- ... 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:

$$size(l) \leq w \cdot size(r) \quad (1)$$
$$size(r) \leq w \cdot size(l) \quad (2)$$

w — **Gewichtung** (Parameter des Algorithmus)

Implementation von balancierten Bäumen

- Der Datentyp

```
data Tree α = Null
            | Node Int (Tree α) α (Tree α)
                                  deriving Eq
```

- Gewichtung (Parameter des Algorithmus):

```
weight :: Int
```

- Hilfskonstruktor, setzt Größe (`l`, `r` balanciert)

```
node :: Tree α→ α→ Tree α→ Tree α
```

- Selektor: Größe des Baumes (0 für `Null`)

```
size :: Tree α→ Int
```

Implementation von balancierten Bäumen

- Hilfskonstruktor, balanciert ggf. neu aus:

```
mkNode :: Tree α→ α→ Tree α→ Tree α
```

  - Voraussetzungen:
    - l, r balanciert
    - Gesamtbaum “fast” balanciert:

$$size(l) - 1 \leq w \cdot size(r) \quad (3)$$

$$size(r) - 1 \leq w \cdot size(l) \quad (4)$$

  - Wird beim Löschen und Einfügen benutzt

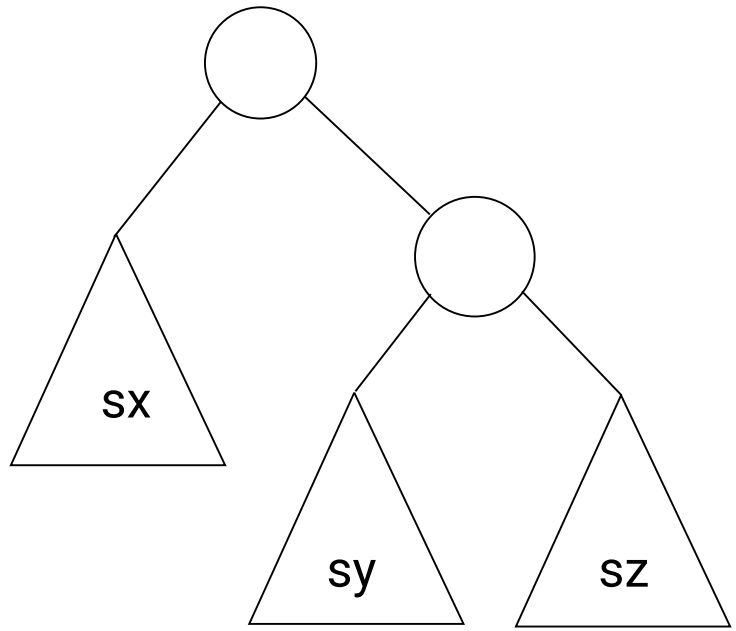
Balance sicherstellen

- Problem:

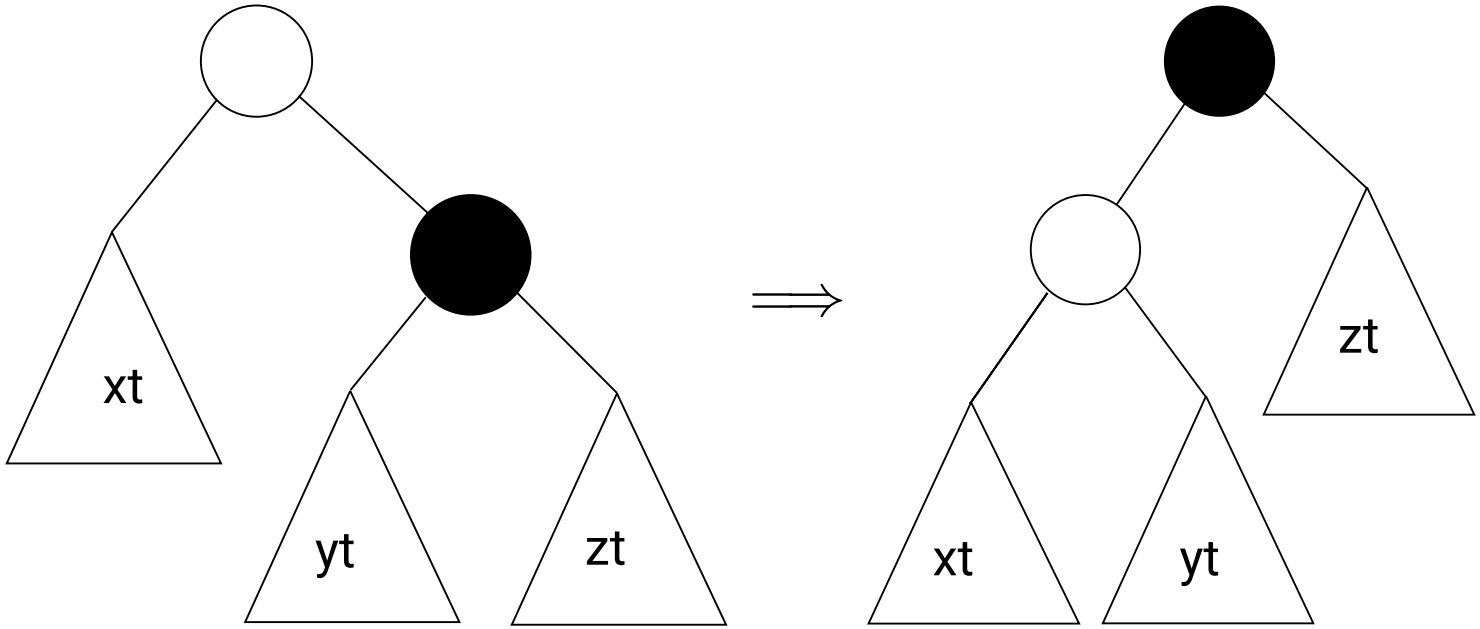
  Nach Löschen oder Einfügen zu großes Ungewicht

- Lösung:

  Rotieren der Unterbäume

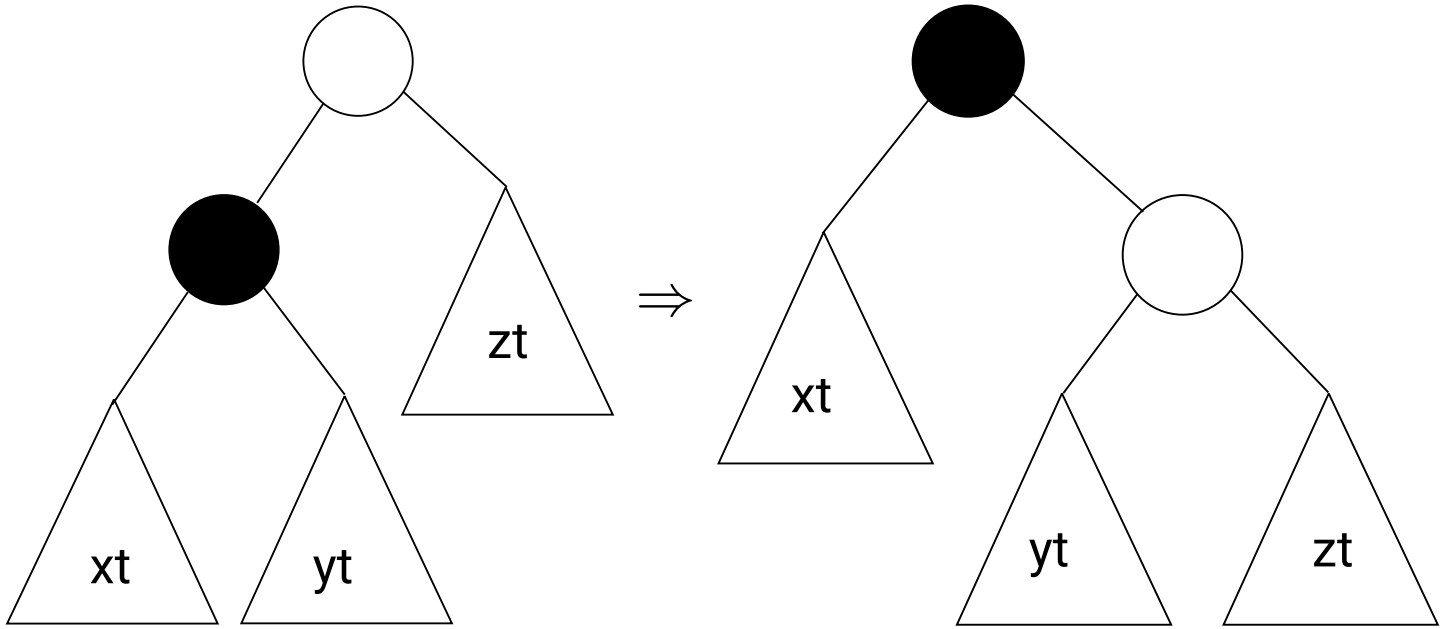


Linksrotation




```
rotl :: Tree α→ Tree α
rotl (Node _ xt y (Node _ yt x zt)) =
  node (node xt y yt) x zt
```

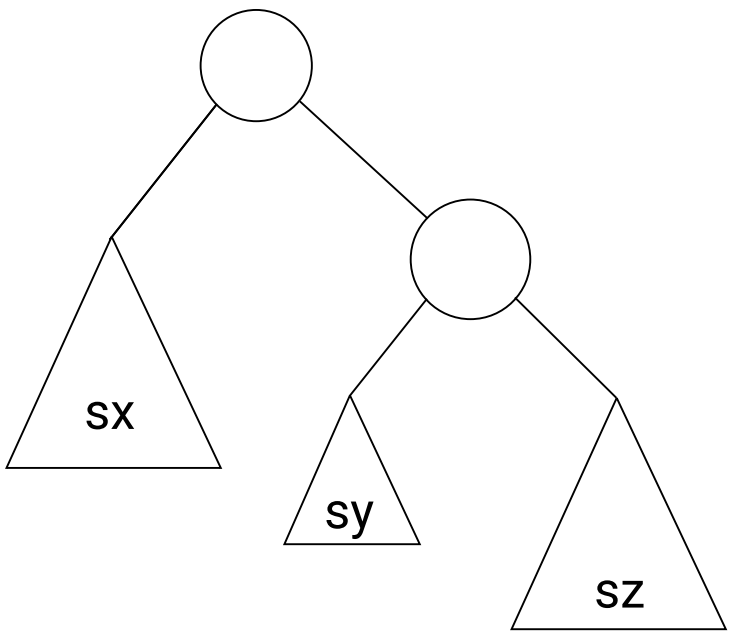
Rechtsrotation




```
rotr :: Tree α→ Tree α
rotr (Node _ (Node _ ut y vt) x rt) =
  node ut y (node vt x rt)
```

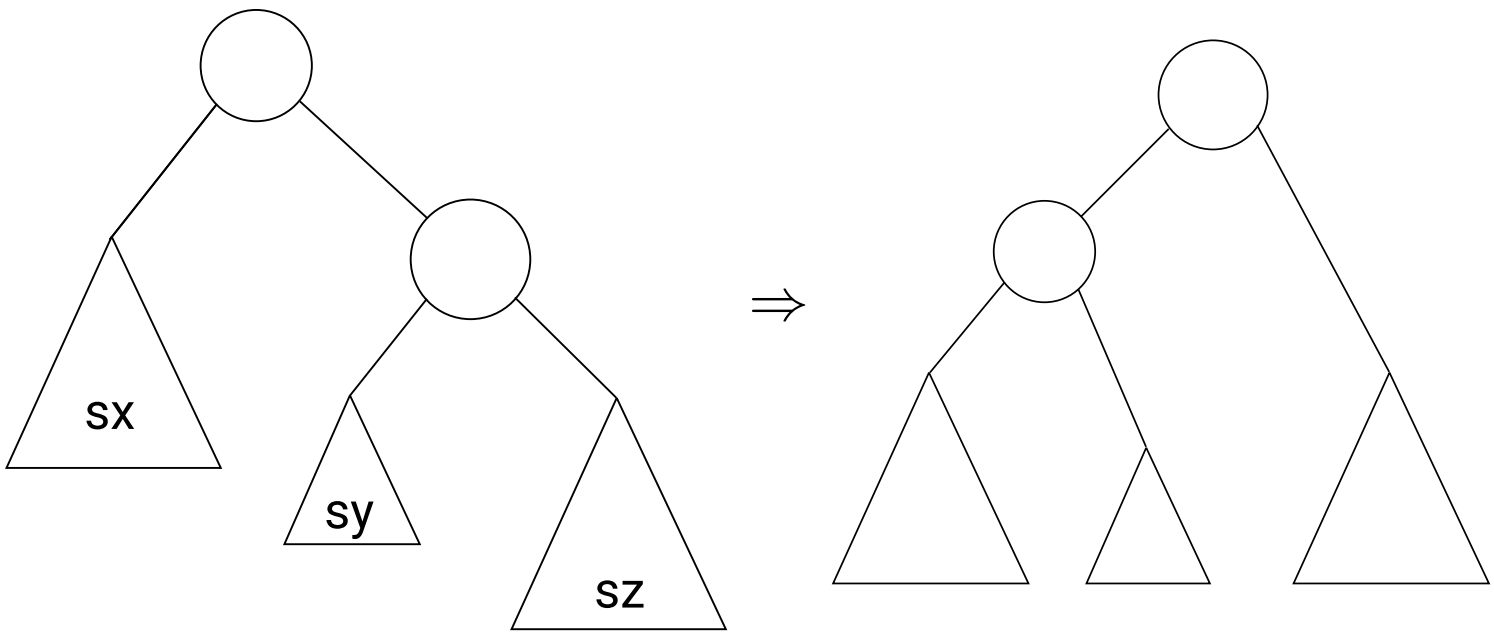
Balanciertheit sicherstellen

- Fall 1: Äußerer Unterbaum zu groß



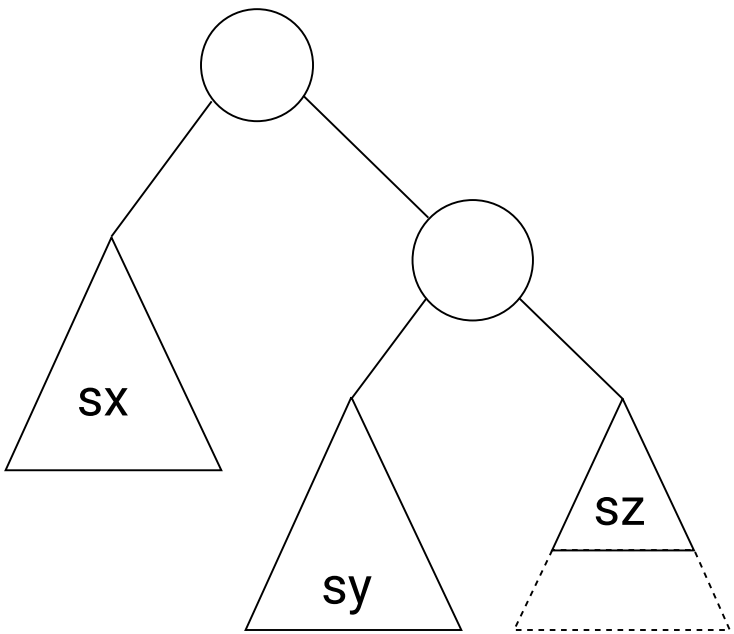
Balanciertheit sicherstellen

- Fall 1: Äußerer Unterbaum zu groß
- Lösung: Linksrotation



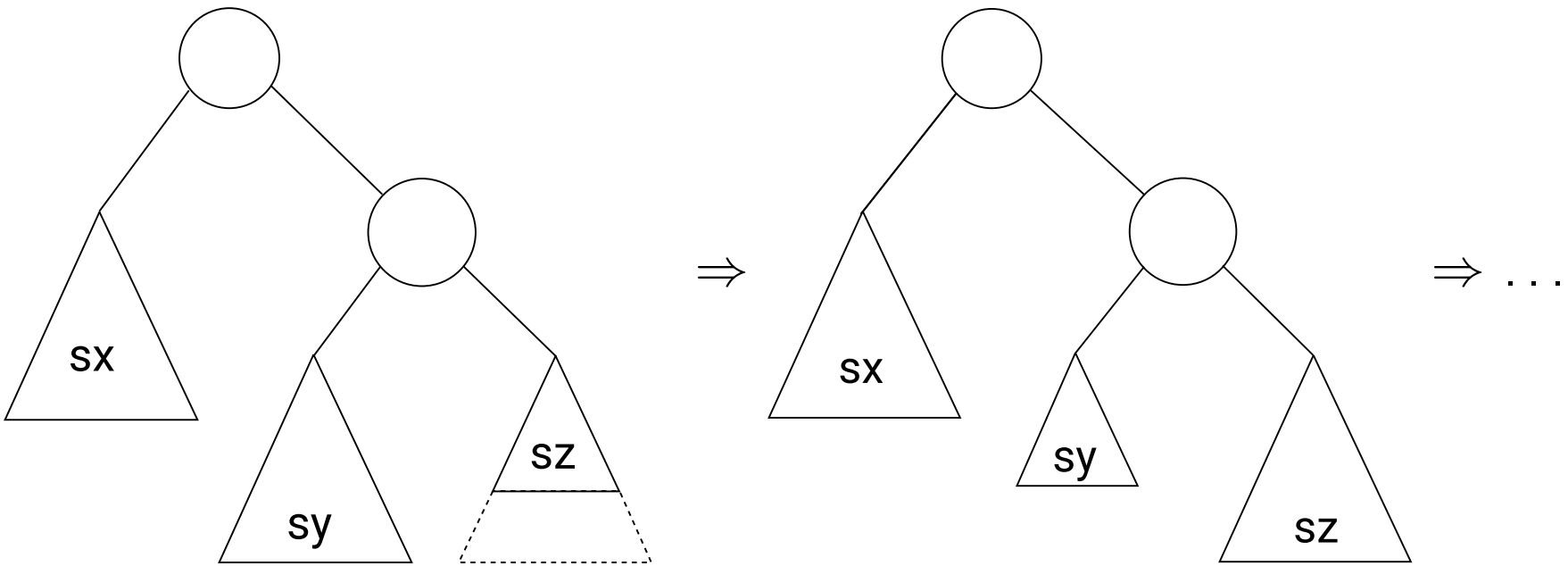
Balanciertheit sicherstellen

- Fall 2: Innerer Unterbaum zu groß oder gleich groß



Balanciertheit sicherstellen

- Fall 2: Innerer Unterbaum zu groß oder gleich groß
- Reduktion auf vorherigen Fall durch Rechtsrotation des Unterbaumes



Balance sicherstellen

- Hilfsfunktion: **Balance** eines Baumes

```
bias :: Tree α→ Ordering
bias Null = EQ
bias (Node _ lt _ rt) = compare (size lt) (size rt)
```

- Zu implementieren: mkNode lt y rt
  - Voraussetzung: lt, rt balanciert
  - Konstruiert neuen balancierten Baum mit Knoten y
- Fallunterscheidung:
  - rt zu groß, zwei Unterfälle:
    - Linker Unterbaum von rt kleiner (Fall 1): bias rt == LT
    - Linker Unterbaum von rt größer/gleich groß (Fall 2): bias rt == EQ, bias rt == GT
  - lt zu groß, zwei Unterfälle (symmetrisch).

Konstruktion eines ausgeglichenen Baumes

- Voraussetzung: `lt`, `rt` balanciert

```
mkNode :: Tree α→ α→ Tree α→ Tree α
mkNode lt x rt
  | ls+rs < 2  = node lt x rt
  | weight* ls < rs  =
      if bias rt == LT then rotl (node lt x rt)
      else rotl (node lt x (rotr rt))
  | ls > weight* rs =
      if bias lt == GT then rotr (node lt x rt)
      else rotr (node (rotl lt) x rt)
  | otherwise   = node lt x rt where
    ls = size lt; rs= size rt
```

Balancierte Bäume als Maps

- Endliche Abbildung: Bäume mit (key, value) Paaren
- lookup' liest Element aus:

```
lookup' :: Ord α⇒ α→ Tree (α, β)→ Maybe β
```

- insert' fügt neues Element ein:

```
insert' :: Ord α⇒ α→ β→ Tree (α, β)→ Tree (α, β)
insert' k v Null = node Null (k, v) Null
insert' k v (Node n l a@(kn, _) r)
  | k < kn  = mkNode (insert' k v l) a r
  | k == kn = Node n l (k, v) r
  | k > kn  = mkNode l a (insert' k v r)
```

- remove' löscht ein Element
  - Benötigt Hilfsfunktion join :: Tree $\alpha \to$ Tree $\alpha \to$ Tree α

Zusammenfassung Balancierte Bäume

- Verkapselung des Datentypen:

```
data Map α β = Map { tree :: Tree (α, β) }
```

```
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
insert k v (Map t) = Map (insert' k v t)
```

- Auslesen, einfügen und löschen: logarithmischer Aufwand ($\mathcal{O}(\log n)$)
- Fold: linearer Aufwand ($\mathcal{O}(n)$)
- Guten durchschnittlichen Aufwand
- Auch in der Haskell-Bücherei: Data.Map (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
(1)	358.4 ms 7.483 ms	2.66 ms 134.70 μs	10.75 ns 159.70 ps	10.90 ns 170.90 ps	1.83 ms 111.80 μs
(2)	6.20 s 37.59 ms	11.57 μs 351.3 ns	133.8 μs 2.36 μs	148.40 μs 1.99 μs	5.67 ms 128.10 μs
(3)	470.00 ms 2.69 ms	265.10 ns 4.54 ns	138.90 μs 2.35 μs	137.60 μs 3.00 μs	2.18 ms 81.68 μs
(4)	392.7 ms 5.02 ms	189.2 ns 13.41 ns	135.7 μs 2.00 μs	134.50 μs 3.10 μs	2.08 ms 80.22 μs

(1) MapFun, (2) MapList, (3) MapWeighted, (4) Data.Map.Lazy
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**

ADTs vs. Objekte

- ADTs (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 (Konstruktoren nicht unterscheidbar)
  - Vererbungsstruktur auf Objekten (Verfeinerung für ADTs)
  - Java: `interface` eigenes Sprachkonstrukt
  - Java: `packages` für Sichtbarkeit

Zusammenfassung

- **Abstrakte Datentypen** (ADTs):
  - Besteht aus **Typen** und **Operationen** darauf
- Realisierung in Haskell durch **Module**
- Beispieldatentypen: endliche Abbildungen
- Nächste Vorlesung: ADTs durch **Eigenschaften** spezifizieren

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 9 vom 11.12.2018: Signaturen und Eigenschaften

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Organisatorisches

- Anmeldung zur **Probeklausur**:
  - Ab sofort auf der stud.ip-Seite.
  - Bis **Do 12:00**
- Termin: Montag, 17.12.2018 10:00 (**pünktlich**) und 10:30
- Ort: Testzentrum des ZMML, neben der Uni-Bücherei auf dem Boulevard
- Dauer: 30 Minuten
- Inhalt: zwei kleine Funktionen implementieren, vier M/C-Fragen

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

Endliche Abbildung: Signatur

- Adressen und Werte sind Parameter
- Typ Map α β, 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
```

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

Eigenschaften Endlicher Abbildungen

1. Aus der **leeren** Abbildung kann **nichts** gelesen werden.
2. Wenn etwas **geschrieben** wird, und an der **gleichen** Stelle wieder **gelesen**, erhalte ich den geschriebenen Wert.
3. Wenn etwas **geschrieben** wird, und an **anderer** Stelle etwas **gelesen** wird, 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

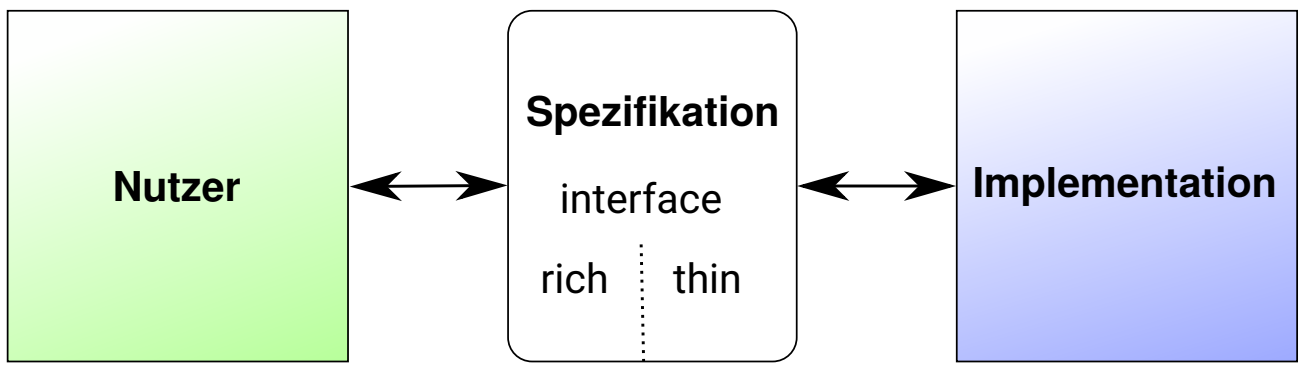
Definition (Axiome)

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 || q
  - **Implikation** p $\Longrightarrow$ q

Axiome als Interface

- Axiome müssen **gelten**
  - für alle Werte der freien Variablen zu `True` auswerten
- Axiome **spezifizieren**:
  - nach außen das **Verhalten**
  - nach innen die **Implementation**
- Signatur + Axiome = **Spezifikation**



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 v s :: Map Int String)
```

- Sehr **viele** Axiome (insgesamt 13)!

Thin vs. Rich Interfaces

- Benutzersicht: **reiches** Interface
  - Viele Operationen und Eigenschaften
- Implementationssicht: **schlankes** Interface
  - Wenig Operation und Eigenschaften
- Beispiel Map:
  - 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
```

  - Thin-to-rich:

```
insert a v = put a (Just v)
```

```
delete a = put a Nothing
```

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v (s :: Map Int String)) == v
```

- Lesen an anderer Stelle liefert alten Wert:

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v (s :: Map Int String)) == v
```

- Lesen an anderer Stelle liefert alten Wert:

```
a ≠ b ⟹ lookup a (put b v s) ==
             lookup a (s :: Map Int String)
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v (s :: Map Int String)) == v
```

- Lesen an anderer Stelle liefert alten Wert:

```
a /= b ==> lookup a (put b v s) ==
           lookup a (s :: Map Int String)
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

```
put a w (put a v s) == put a w (s :: Map Int String)
```

- Schreiben über verschiedene Stellen kommutiert:

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v (s :: Map Int String)) == v
```

- Lesen an anderer Stelle liefert alten Wert:

```
a ≠ b ⟹ lookup a (put b v s) ==
            lookup a (s :: Map Int String)
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

```
put a w (put a v s) == put a w (s :: Map Int String)
```

- Schreiben über verschiedene Stellen kommutiert:

```
a ≠ b ⟹ put a v (put b w s) ==
            put b w (put a v s :: Map Int String)
```

Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

```
lookup a (empty :: Map Int String) == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v (s :: Map Int String)) == v
```

- Lesen an anderer Stelle liefert alten Wert:

```
a ≠ b ⟹ lookup a (put b v s) ==
           lookup a (s :: Map Int String)
```

- Schreiben an dieselbe Stelle überschreibt alten Wert:

```
put a w (put a v s) == put a w (s :: Map Int String)
```

- Schreiben über verschiedene Stellen kommutiert:

```
a ≠ b ⟹ put a v (put b w s) ==
           put b w (put a v s :: Map Int St
```

Thin: 5 Axiome
Rich: 13 Axiome

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 **instantiieren**
  - 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

- 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
```

- Hier: Eigenschaften als **Haskell-Prädikate**
- *QuickCheck*-Axiome mit `QC.testProperty` in *Tasty* eingebettet
- Es werden N Zufallswerte generiert und getestet (Default $N = 100$)

Axiome mit *QuickCheck* testen

- **Bedingte** Eigenschaften:
  - $A \Longrightarrow B$ 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␣other" $ λa b v s→
  a ≠ b ⟹ 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

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
```

  - Bspw. Konstruktion einer Map:
    - Zufällige Länge, dann aus sovielen zufälligen Werten Map konstruieren
    - Zufallswerte in Haskell?

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!

Signatur und Semantik

Stacks

Typ: St α

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 α

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, unterscheidliche Semantik.

Eigenschaften von Stack

- Last in first out (LIFO):

$$\text{top}\ (\text{push}\ a_1\ (\text{push}\ a_2\ \ldots\ (\text{push}\ a_n\ \text{empty}))) = a_1$$

Eigenschaften von Stack

- Last in first out (LIFO):

$$\text{top}\ (\text{push}\ a_1\ (\text{push}\ a_2\ \ldots\ (\text{push}\ a_n\ \text{empty}))) = a_1$$

$$\text{top}\ (\text{push}\ a\ s) = a$$

$$\text{pop}\ (\text{push}\ a\ s) = s$$

$$\text{push}\ a\ s \neq \text{empty}$$

Eigenschaften von Queue

- First in first out (FIFO):

$$\texttt{first}\ (\text{enq}\ a_1\ (\text{enq}\ a_2\ \ldots\ (\text{enq}\ a_n\ \text{empty}))) = a_n$$

Eigenschaften von Queue

- First in first out (FIFO):

$$\texttt{first}\ (\texttt{enq}\ a_1\ (\texttt{enq}\ a_2\ \ldots\ (\texttt{enq}\ a_n\ \texttt{empty}))) = a_n$$

$$\texttt{first}\ (\texttt{enq}\ a\ \texttt{empty}) == a$$

$$q \neq \texttt{empty} \implies \texttt{first}\ (\texttt{enq}\ a\ q) == \texttt{first}\ q$$

$$\texttt{deq}\ (\texttt{enq}\ a\ \texttt{empty}) == \texttt{empty}$$

$$q \neq \texttt{empty} \implies \texttt{deq}\ (\texttt{enq}\ a\ q) = \texttt{enq}\ a\ (\texttt{deq}\ q)$$

$$\texttt{enq}\ a\ q \neq \texttt{empty}$$

Implementation von Stack: Liste

Sehr einfach: ein Stack ist eine Liste

```
data St α = St [α]  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 ist teuer.
- 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

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\,\rangle$	([], [])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
`empty`	$\langle\,\rangle$	([], [])
`enq 9`	$\langle 9 \rangle$	([9], [])
`enq 4`	$\langle 4 \rightarrow 9 \rangle$	([9], [4])
`enq 7`	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])
`first`	9	

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
`empty`	$\langle\ \rangle$	([], [])
`enq 9`	$\langle 9 \rangle$	([9], [])
`enq 4`	$\langle 4 \rightarrow 9 \rangle$	([9], [4])
`enq 7`	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])
`first`	9	
`deq`	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])
`enq 5`	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])
`enq 3`	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])
`first`	4	

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	([7], [3, 5])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \to 9 \rangle$	([9], [4])
enq 7	$\langle 7 \to 4 \to 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \to 4 \rangle$	([4, 7], [])
enq 5	$\langle 5 \to 7 \to 4 \rangle$	([4, 7], [5])
enq 3	$\langle 3 \to 5 \to 7 \to 4 \rangle$	([4, 7], [3, 5])
first	4	
deq	$\langle 3 \to 5 \to 7 \rangle$	([7], [3, 5])
first	7	

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	([7], [3, 5])
first	7	
deq	$\langle 3 \rightarrow 5 \rangle$	([5, 3], [])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \to 9 \rangle$	([9], [4])
enq 7	$\langle 7 \to 4 \to 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \to 4 \rangle$	([4, 7], [])
enq 5	$\langle 5 \to 7 \to 4 \rangle$	([4, 7], [5])
enq 3	$\langle 3 \to 5 \to 7 \to 4 \rangle$	([4, 7], [3, 5])
first	4	
deq	$\langle 3 \to 5 \to 7 \rangle$	([7], [3, 5])
first	7	
deq	$\langle 3 \to 5 \rangle$	([5, 3], [])
first	5	

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\ \rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \to 9 \rangle$	([9], [4])
enq 7	$\langle 7 \to 4 \to 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \to 4 \rangle$	([4, 7], [])
enq 5	$\langle 5 \to 7 \to 4 \rangle$	([4, 7], [5])
enq 3	$\langle 3 \to 5 \to 7 \to 4 \rangle$	([4, 7], [3, 5])
first	4	
deq	$\langle 3 \to 5 \to 7 \rangle$	([7], [3, 5])
first	7	
deq	$\langle 3 \to 5 \rangle$	([5, 3], [])
first	5	
deq	$\langle 3 \rangle$	([3], [])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\,\rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \to 9 \rangle$	([9], [4])
enq 7	$\langle 7 \to 4 \to 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \to 4 \rangle$	([4, 7], [])
enq 5	$\langle 5 \to 7 \to 4 \rangle$	([4, 7], [5])
enq 3	$\langle 3 \to 5 \to 7 \to 4 \rangle$	([4, 7], [3, 5])
first	4	
deq	$\langle 3 \to 5 \to 7 \rangle$	([7], [3, 5])
first	7	
deq	$\langle 3 \to 5 \rangle$	([5, 3], [])
first	5	
deq	$\langle 3 \rangle$	([3], [])
first	3	

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\,\rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \rightarrow 9 \rangle$	([9], [4])
enq 7	$\langle 7 \rightarrow 4 \rightarrow 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \rightarrow 4 \rangle$	([4, 7], [])
enq 5	$\langle 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [5])
enq 3	$\langle 3 \rightarrow 5 \rightarrow 7 \rightarrow 4 \rangle$	([4, 7], [3, 5])
first	4	
deq	$\langle 3 \rightarrow 5 \rightarrow 7 \rangle$	([7], [3, 5])
first	7	
deq	$\langle 3 \rightarrow 5 \rangle$	([5, 3], [])
first	5	
deq	$\langle 3 \rangle$	([3], [])
first	3	
deq	$\langle\,\rangle$	([], [])

Repräsentation von Queue

Operation	Resultat	Interne Repräsentation
empty	$\langle\,\rangle$	([], [])
enq 9	$\langle 9 \rangle$	([9], [])
enq 4	$\langle 4 \to 9 \rangle$	([9], [4])
enq 7	$\langle 7 \to 4 \to 9 \rangle$	([9], [7, 4])
first	9	
deq	$\langle 7 \to 4 \rangle$	([4, 7], [])
enq 5	$\langle 5 \to 7 \to 4 \rangle$	([4, 7], [5])
enq 3	$\langle 3 \to 5 \to 7 \to 4 \rangle$	([4, 7], [3, 5])
first	4	
deq	$\langle 3 \to 5 \to 7 \rangle$	([7], [3, 5])
first	7	
deq	$\langle 3 \to 5 \rangle$	([5, 3], [])
first	5	
deq	$\langle 3 \rangle$	([3], [])
first	3	
deq	$\langle\,\rangle$	([], [])
first	error	
deq	error	

Implementation

- Datentyp:

```
data Qu α = Qu [α] [α]
```

- Leere Schlange: alles leer

```
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
```

- Gleichheit:

```
valid :: Qu α→ Bool
valid (Qu [] ys) = null ys
valid (Qu (_:_) _) = True
```

Implementation

- Bei enq und deq Invariante prüfen

```
enq x (Qu xs ys) = check xs (x:ys)
```

```
deq (Qu [] _ )      = error "Queue: deq of empty Q"
deq (Qu (_:xs) ys) = check xs ys
```

  - Prüfung der Invariante nach dem Einfügen und Entnehmen

- check **garantiert** Invariante

```
check :: [α]→ [α]→ Qu α
check [] ys = Qu (reverse ys) []
check xs ys = Qu 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
  - $\Longrightarrow$ für bedingte Eigenschaften

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 10 vom 18.12.2018: Aktionen und Zustände

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Organisatorisches

- Probeklausur
  - Echte Klausur: 2 Stunden, wahrscheinlich 4 Programmieraufgaben
  - Probeklausur und Fragen werden **veröffentlicht**.
- Tutorium Do 16– 18: **Raumänderung** (diese Woche in MZH 1450)

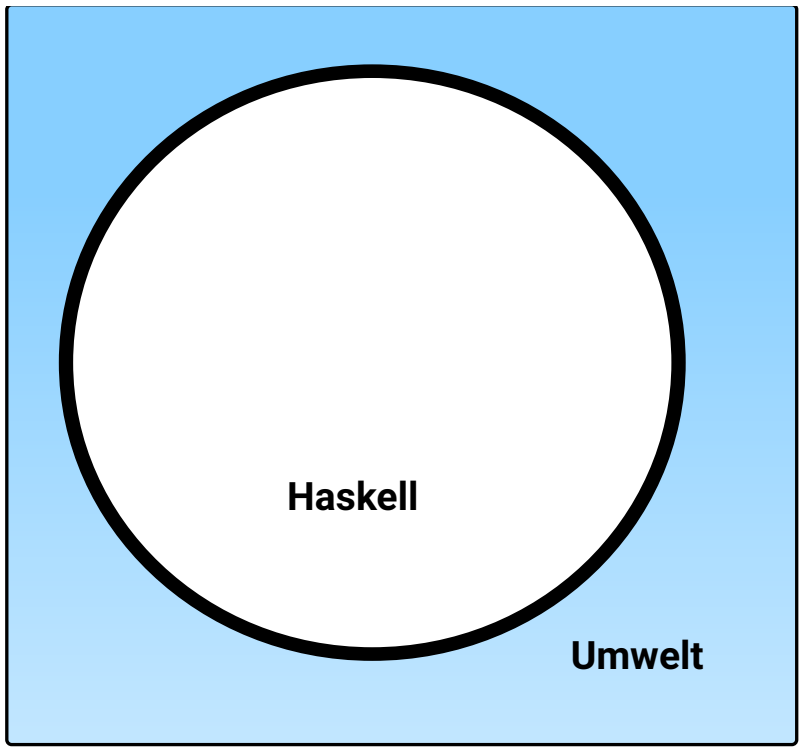
Fahrplan

- 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
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Inhalt

- Ein/Ausgabe in funktionale Sprachen
- Wo ist das **Problem**?
- **Aktionen** und der Datentyp *IO*.
- Vordefinierte Aktionen
- Beispiel: Wortratespiel
- Aktionen als Werte

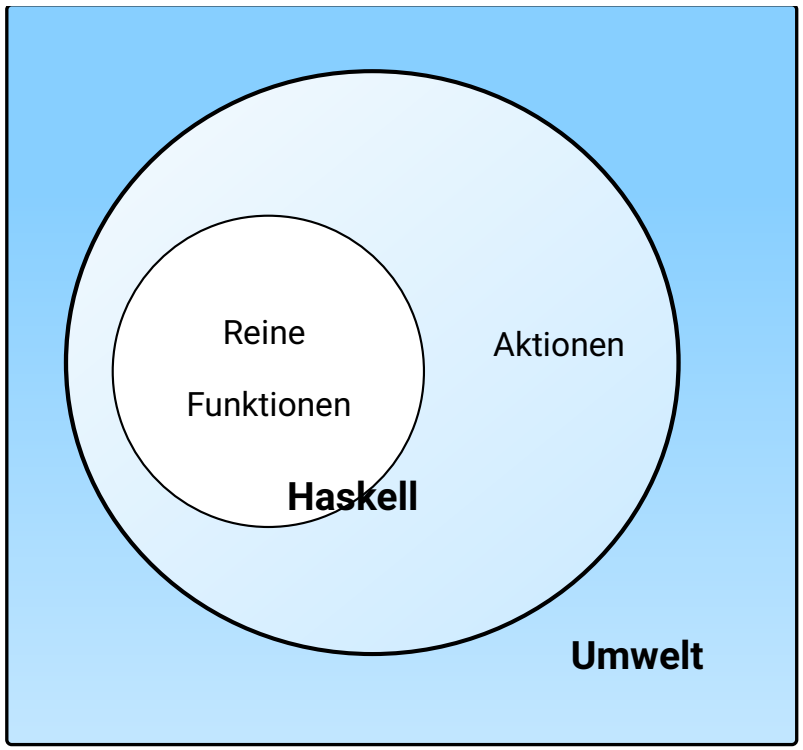
Ein- und Ausgabe in funktionalen Sprachen



Problem:

- Funktionen mit Seiteneffekten nicht referentiell transparent.
- readString :: ... $\rightarrow$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:

$$\textbf{type}\ \mathsf{IO}\ \alpha$$

$$(\gg\!=) \quad :: \ \mathsf{IO}\ \alpha \to (\alpha \to \mathsf{IO}\ \beta) \to \mathsf{IO}\ \beta$$

$$\mathtt{return} \ :: \ \alpha \to \mathsf{IO}\ \alpha$$

- 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
```

$\Longleftrightarrow$

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

  - Rechts sind $\gg=$, $\gg$ implizit.

- Es gilt die **Abseitsregel**.

  - Einrückung der ersten Anweisung nach **do** bestimmt Abseits.

Drittes Beispiel

- Zählendes, endliches Echo

```
echo3 :: Int→ IO ()
echo3 cnt = do
  putStr (show cnt ++ ":␣")
  s← getLine
  if s ≠ "" then do
      putStrLn $ show cnt ++ ":␣"++ s
      echo3 (cnt+1)
    else return ()
```

- Was passiert hier?
  - Kombination aus Kontrollstrukturen und Aktionen
  - **Aktionen** als **Werte**
  - Geschachtelte **do**-Notation

Ein/Ausgabe mit Dateien

- Im Prelude **vordefiniert**:
  - Dateien schreiben (überschreiben, anhängen):

```
type FilePath = String
writeFile     :: FilePath → String → IO ()
appendFile    :: FilePath → String → IO ()
```

- Datei lesen (verzögert):

```
readFile      :: FilePath → IO String
```

- "Lazy I/O": Zugriff auf Dateien erfolgt **verzögert**
  - Interaktion von nicht-strikter Auswertung mit zustandsbasiertem Dateisystem kann überraschend sein

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.

Beispiel: Zeichen, Wörter, Zeilen zählen (wc)

```
wc :: String→ IO ()
wc file =
  do cont ← readFile file
     putStrLn $ file++ ": "++
                show (length (lines cont),
                      length (words cont),
                      length cont)
```

- Datei wird gelesen
- Anzahl Zeichen, Worte, Zeilen gezählt
- Erstaunlich (hinreichend) effizient

Aktionen als Werte

- **Aktionen** sind **Werte** wie alle anderen.
- Dadurch **Definition** von **Kontrollstrukturen** möglich.
- Endlosschleife:

```
forever :: IO α→ IO α
forever a = a ≫ forever a
```

- Iteration (feste Anzahl):

```
forN :: Int→ IO α→ IO ()
forN n a | n == 0     = return ()
         | otherwise = a ≫ forN (n−1) a
```

Kontrollstrukturen

- Vordefinierte Kontrollstrukturen (Control.Monad):

```
when :: Bool → IO () → IO ()
```

- Sequenzierung:

```
sequence :: [IO α] → IO [α]
```

  - Sonderfall: [()] als ()

```
sequence_ :: [IO ()] → IO ()
```

- Map und Filter für Aktionen:

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

Fehlerbehandlung

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

$$
\begin{aligned}
&\mathsf{throw} &&:: \mathsf{Exception}\ \gamma \Rightarrow \gamma \rightarrow \alpha \\
&\mathsf{catch} &&:: \mathsf{Exception}\ \gamma \Rightarrow \mathsf{IO}\ \alpha \rightarrow (\gamma \rightarrow \mathsf{IO}\ \alpha) \rightarrow \mathsf{IO}\ \alpha \\
&\mathsf{try} &&:: \mathsf{Exception}\ \gamma \Rightarrow \mathsf{IO}\ \alpha \rightarrow \mathsf{IO}\ (\mathsf{Either}\ \gamma\ \alpha)
\end{aligned}
$$

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

Fehler fangen und behandeln

"Ask forgiveness not permission"

Generelle Regel: **Fehlerbehandlung** durch **Ausnahmebehandlung** besser als Fehlerbedingungen abzufragen

- Fehlerbehandlung für wc:

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

- IOError kann analysiert werden (siehe System.IO.Error)
- read mit Ausnahme bei Fehler (statt Programmabbruch):

```
readIO :: Read 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
  mapM_ wc2 args
```

Beispiel: Traversion 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 :: IOError))
```

- Nutzt Funktionalität aus System.Directory, System.FilePath

So ein Zufall!

- Zufallswerte:

```
randomRIO :: (α, α) → IO α
```

  - Warum ist randomIO **Aktion**?

So ein Zufall!

- Zufallswerte:

```
randomRIO :: (α, α) → IO α
```

  - Warum ist randomIO **Aktion**?

- **Beispiele:**
  - Aktion zufällig oft ausführen:

```
atmost :: Int → IO α → IO [α]
atmost most a =
  do l ← randomRIO (1, most)
     sequence (replicate l a)
```

  - Zufälligen String erzeugen:

```
randomStr :: IO String
randomStr = atmost 40 (randomRIO ('a','z'))
```

Fallbeispiel: Wörter raten

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

Wörter raten: Programmstruktur

- 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

Zusammenfassung

- Ein/Ausgabe in Haskell durch **Aktionen**
- **Aktionen** (Typ IO α) sind seiteneffektbehaftete Funktionen
- Komposition von Aktionen durch

```
(>>=)  :: IO α -> (α -> IO β) -> IO β
return :: α -> IO α
```

- **do**-Notation
- Fehlerbehandlung durch Ausnahmen (IOError, catch, try).
- Verschiedene Funktionen der Standardbücherei:
  - Prelude: getLine, putStr, putStrLn, readFile, writeFile
  - Module: System.IO, System.Random
- Aktionen sind implementiert als **Zustandstransformationen**

Frohe Weihnachten und einen Guten Rutsch!

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 11 vom 08.01.2019: Monaden als Berechnungsmuster

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

Frohes Neues Jahr!

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
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Inhalt

- Wie geht das mit IO?
- Das M-Wort
- Monaden als allgemeine Berechnungsmuster
- Fallbeispiel: Auswertung von Ausdrücken

Zustandsabhängige Berechnungen

Funktionen mit Zustand

- Idee: Seiteneffekt **explizit** machen
- Funktion $f : A \to B$ mit Seiteneffekt in **Zustand** S:

$$f : A \times S \to B \times S$$
$$\cong$$
$$f : A \to S \to B \times S$$

- Datentyp: $S \to B \times S$
- Komposition: Funktionskomposition und uncurry

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

In Haskell: Zustände explizit

- **Zustandstransformer:** Berechnung mit Seiteneffekt in Typ σ (polymorph über α)

```
type State σ α = σ → (α, σ)
```

- Komposition zweier solcher Berechnungen:

```
comp :: State σ α → (α → State σ β) → State σ β
comp f g = uncurry g ∘ f
```

- Trivialer Zustand:

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

- Lifting von Funktionen:

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

Zugriff auf den Zustand

- Zustand lesen:

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

- Zustand setzen:

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

Einfaches Beispiel

- Zähler als Zustand:

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

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

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

- Hauptfunktion (verkapselt State):

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

Monaden

Monaden als Berechnungsmuster

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

State:

```
type State σ α
```

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

```
lift  ::  α→ State σ α
```

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

Aktionen:

```
type IO α
```

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

```
return  ::  α→ IO α
```

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

Berechnungsmuster: **Monade**

Monaden als Berechngsmuster

Eine Monade ist:

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

Monaden in Haskell

- Aktion auf Funktionen:

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

fmap bewahrt Identität und Komposition:

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

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

Monaden in Haskell

- Verkettung ($\gg=$) und Lifting (`return`):

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

$\gg=$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

- Zustandstransformer: 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 als Monade:

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

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

  - Ähnlich mit Either

- Berechnungsmodell: **Ausnahmen** (Fehler)
  - f :: $\alpha \rightarrow$ Maybe β ist Berechnung mit möglichem Fehler
  - Fehlerfreie Berechnungen werden verkettet
  - Fehler (Nothing oder Left x) werden propagiert

Mehrdeutigkeit

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

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

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

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

Beispiel

- Berechnung aller Permutationen einer Liste:

1. Ein Element überall in eine Liste einfügen:

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

2. Damit Permutationen (rekursiv):

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

Der Listenmonade in der Listenkomprehension

- Berechnung aller Permutationen einer Liste:

  1. Ein Element überall in eine Liste einfügen:

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

  2. Damit Permutationen (rekursiv):

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

- Listenkomprehension $\cong$ Listenmonade

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 $A \rightharpoonup V$ (Adressen A, Werte V)
  - Zustand des Dateisystems
  - Zustand des Zufallsgenerators
- In Haskell: Typ RealWorld
  - "Virtueller" Typ, Zugriff nur über elementare Operationen
  - Entscheidend nur Reihenfolge der Aktionen

Fallbeispiel: Auswertung von Ausdrücken

Monaden im Einsatz

- Auswertung von Ausdrücken:

```
data Expr = Var String
          | Num Double
          | Plus Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
          | Div Expr Expr
```

- Mögliche Arten von Effekten:
  - Partialität (Division durch 0)
  - Zustände (für die Variablen)
  - Mehrdeutigkeit

Monaden im Einsatz

- Auswertung von Ausdrücken:

```
data Expr = Var String
          | Num Double
          | Plus Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
          | Div Expr Expr
```

- Mögliche Arten von Effekten:
  - Partialität (Division durch 0)
  - Zustände (für die Variablen)
  - Mehrdeutigkeit

- 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
```

Auswertung mit Fehlern

- Partialität durch Maybe-Monade

```
eval :: Expr → Maybe Double
eval (Var _)   = 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; if y == 0 then Nothing else Just $ 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

```
data Res σ α = Res { run :: σ→ [Maybe α] }
```

- Andere Kombinationen möglich:

```
data Res σ α = Res (σ→ Maybe [α])
```

Kombination der Effekte

- Benötigt **Kombination** der Monaden.
- Monade Res:
  - Zustandsabhängig
  - Mehrdeutig
  - Fehlerbehaftet

```
data Res σ α = Res { run :: σ→ [Maybe α] }
```

- Andere Kombinationen möglich:

```
data Res σ α = Res (σ→ Maybe [α])
```

```
data Res σ α = Res (σ→ [α])
```

Kombination der Effekte

- Benötigt **Kombination** der Monaden.
- Monade Res:
  - Zustandsabhängig
  - Mehrdeutig
  - Fehlerbehaftet

```
data Res σ α = Res { run :: σ→ [Maybe α] }
```

- Andere Kombinationen möglich:

```
data Res σ α = Res (σ→ Maybe [α])
```

```
data Res σ α = Res (σ→ [α])
```

```
data Res σ α = Res ([σ→ α])
```

Res: Monadeninstanz

- Functor durch Komposition der fmap:

```
instance Functor (Res σ) where
  fmap f (Res g) = Res $ fmap (fmap f). g
```

- Monad ist Kombination

```
instance Monad (Res σ) where
  return a = Res (const [Just a])
  Res f >>= g = Res $ λs→ do ma← f s
                             case ma of
                               Just a→ run (g a) s
                               Nothing→ return Nothing
```

Res: Operationen

- Zugriff auf den Zustand:

```
get :: (σ→ α)→ Res σ α
get f = Res $ λs→ [Just $ f s]
```

- Fehler:

```
fail :: Res σ α
fail = Res $ const [Nothing]
```

- Mehrdeutige Ergebnisse:

```
join :: α→ α→ Res σ α
join a b = Res $ λs→ [Just a, Just 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 (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
                      if y == 0 then fail else return $ x / y
eval (Pick a b)  = do x← eval a; y← eval b; join x y
```

- Systematische Kombination durch **Monadentransformer**
  - Monade mit Platzhalter für weitere Monaden

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 15.01.2019: Domänenspezifische Sprachen (DSLs)

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

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
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Domain-Specific Languages (DSLs)

- Was ist das?
- Wie macht man das?
- Wozu braucht man so etwas?

Programmiersprachen sind überall

- Beispiel 1: **SQL** — Anfragesprache für relationale Datenbanken

Programmiersprachen sind überall

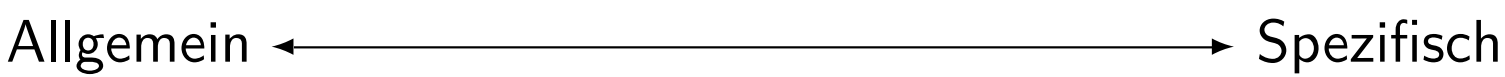
- Beispiel 1: **SQL** — Anfragesprache für relationale Datenbanken
- Beispiel 2: **Excel** — Modellierung von Berechnungen

Programmiersprachen sind überall

- Beispiel 1: **SQL** — Anfragesprache für relationale Datenbanken
- Beispiel 2: **Excel** — Modellierung von Berechnungen
- Beispiel 3: **HTML** oder **LaTeX** oder **Word** — Typesetting

Vom Allgemeinen zum Speziellen

- Modellierung von **Problemen** und **Lösungen**



Allgemeine Lösung: **GPL**

- Mächtige Sprache (Turing-mächtig)
- Große Klasse von Problemen
- Großer Abstand zum Problem
- Java, Haskell, C ...
- General purpose language (GPL)

Spezifische Lösung: **DSL**

- Maßgeschneiderte Sprache
- Wohldefinierte Unterklasse (Domäne) von Problemen
- Geringer Abstand zum Problem
- Als Teil einer Programmiersprache (eingebettet) oder alleinstehend (stand-alone)

DSL: Definition 1

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

(van Deursen et al., 2000)

Eigenschaften von DSLs

- **Fokussierte** Ausdrucksmächtigkeit
  - Turing-Mächtigkeit nicht Ziel der Sprache (aber kein Ausschlusskriterium)
  - Oftmals deutlich weniger mächtig: Reguläre Ausdrücke, Makefiles, HTML
- Üblicherweise **klein** (“little languages”, “micro-languages”)
- Anzahl der Sprachkonstrukte eingeschränkt und auf die Anwendung zugeschnitten
- Meist **deklarativ**: XSLT, Relax NG Schemas, Excel Formeln...
- Spiegeln in Sprachkonstrukten und Vokabular die Domäne wider

Domain-Specific Embedded Languages

- DSL direkt in eine GPL **einbetten**
  - Vorhandenes Ausführungsmodell und Werkzeuge
- Funktionale Sprachen eignen sich hierfür besonders gut
  - Algebraische Datentypen zur Termrepräsentation
  - Funktional $\subseteq$ Deklarativ
  - Funktionen höherer Ordnung ideal für **Kombinatoren**
  - Interpreter (`ghci`, `ocaml`, ...) erlauben "rapid prototyping"
  - Erweiterung zu **stand-alone** leicht möglich
- Andere Sprachen:
  - Java: Eclipse Modelling Framework, Xtext

Beispiel: Reguläre Ausdrücke

Ein regulärer Ausdruck ist:

- Leeres Wort ϵ
- Einzelnes Zeichen c
- Beliebiges Zeichen ?
- Sequenzierung $e_1\ e_2$
- Alternierung $e_1 \mid e_2$
- Kleene-Stern $e^* = \epsilon \mid ee^*$
- Abgeleitet:
  - Kleene-Plus $e^+ = e\ e^*$

Beispiel: Reguläre Ausdrücke

Ein regulärer Ausdruck ist:

- Leeres Wort ϵ
- Einzelnes Zeichen c
- Beliebiges Zeichen ?
- Sequenzierung $e_1\ e_2$
- Alternierung $e_1 \mid e_2$
- Kleene-Stern $e^* = \epsilon \mid ee^*$
- Abgeleitet:
  - Kleene-Plus $e^+ = e\ e^*$

Haskell-Implementierung — Signatur:

```
type RegEx

eps  :: RegEx
char :: Char→ RegEx
arb  :: RegEx
seq  :: RegEx→ RegEx→ RegEx
alt  :: RegEx→ RegEx→ RegEx
star :: RegEx→ RegEx
```

Beispiel: Reguläre Ausdrücke

Ein regulärer Ausdruck ist:

- Leeres Wort ϵ
- Einzelnes Zeichen c
- Beliebiges Zeichen ?
- Sequenzierung $e_1\ e_2$
- Alternierung $e_1 \mid e_2$
- Kleene-Stern $e^* = \epsilon \mid ee^*$
- Abgeleitet:
  - Kleene-Plus $e^+ = e\ e^*$

Haskell-Implementierung — Signatur:

```
type RegEx

eps  :: RegEx
char :: Char→ RegEx
arb  :: RegEx
seq  :: RegEx→ RegEx→ RegEx
alt  :: RegEx→ RegEx→ RegEx
star :: RegEx→ RegEx
```

Implementierung: siehe `SimpleRegEx.hs`

Flache Einbettung vs. Tiefe Einbettung

- **Flache Einbettung:**
  - Domänenfunktionen direkt als Haskell-Funktionen
  - Keine explizite Repräsentation der Domänenobjekte in Haskell
  - Schnell geschrieben und flexibel erweiterbar
- **Tiefe Einbettung:**
  - Repräsentation der Domänenobjekte durch Haskell-Datentyp (oder ADT)
  - Domänenfunktionen auf diesem Datentyp
  - Mächtiger: Manipulation der Domänenobjekte möglich (Reflektion)

Beispiel: Grafik

- Erzeugung von SVG-Grafiken
- Eingebettete DSL:
  - Monade Draw (Zustandsmonade)
  - Funktionen zum Zeichnen:

```
line :: Point→ Point→ Draw ()
polygon :: [Point]→ Draw ()
```

  - "Ausführen": Darstellung in S.Svg, rendering:

```
render :: Int→ Int→ Draw()→ S.Svg
draw :: String→ S.Svg → IO ()
```

Beispielprogramm: Sierpińksy-Dreieck

Dreieck mit Eckpunkten zeichnen:

```
drawTriangle :: Point→ Point→ Point→ Draw ()
```

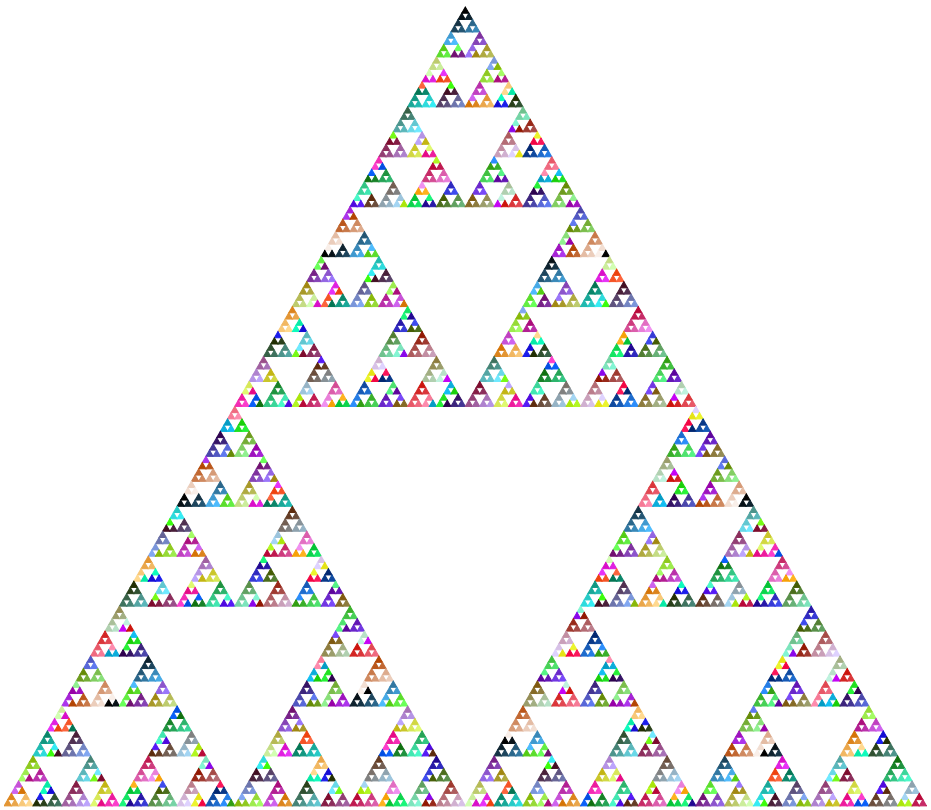
Mitte zwischen zwei Punkten:

```
midway :: Point→ Point→ Point
midway p q = 0.5 `smult` (p+q)
```

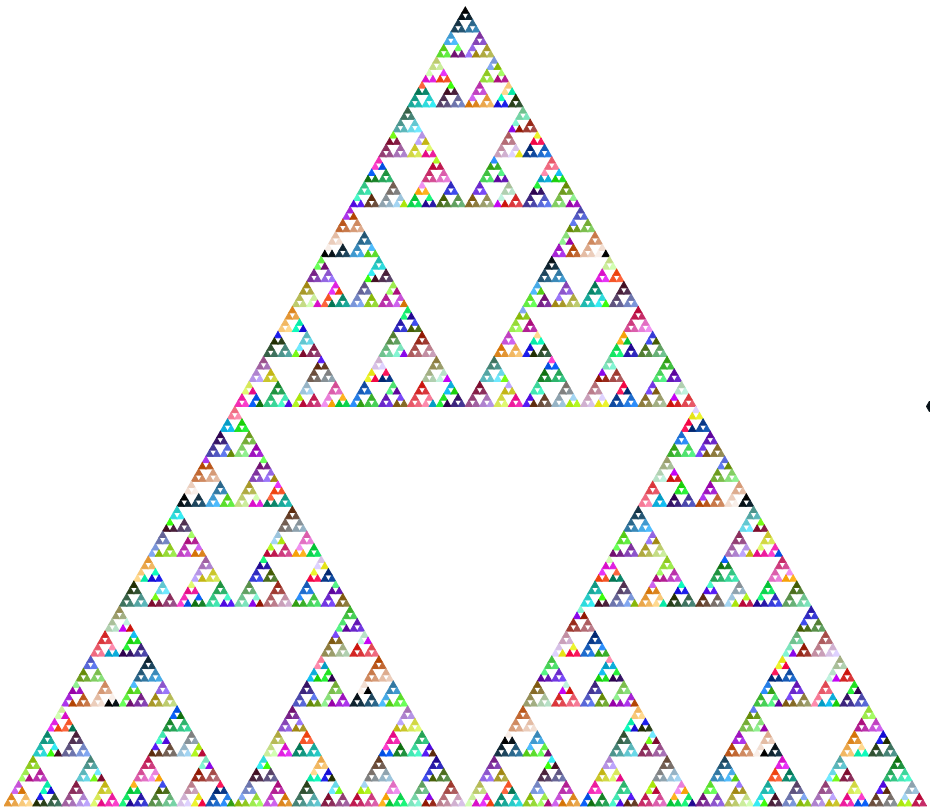
Sierpińksy-Dreieck rekursiv

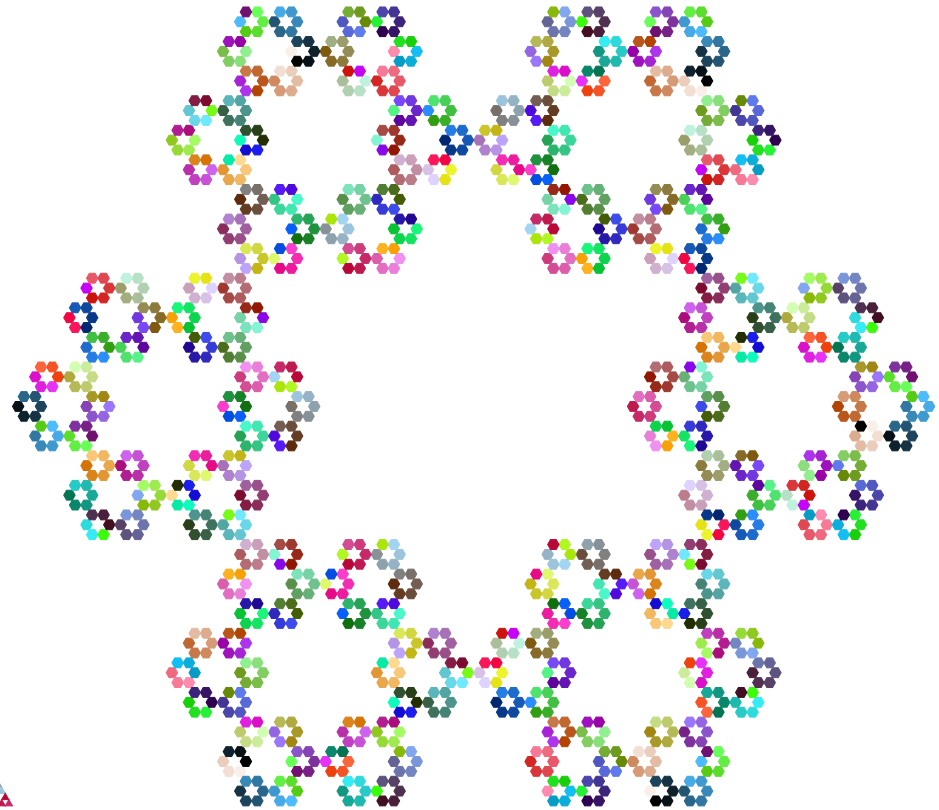
```
spTri :: Double→ Int→ Draw ()
spTri sz limit = sp3 a b c 0 where
  h = sz* sqrt 3/4
  a = Pt 0 (−h); b = Pt (−sz/2) h; c= Pt (sz/2) h
  sp3 :: Point→ Point→ Point→ Int→ Draw ()
  sp3 a b c n
    | n ≥ limit = drawTriangle a b c
    | otherwise = do
        let ab = midway a b; bc = midway b c; ca = midway c a
        sp3 a ab ca (n+1); sp3 ab b bc (n+1); sp3 ca bc c (n+1)
```

Resultat: Sierpińsky-Dreieck und Schneeflocke



Resultat: Sierpińsky-Dreieck und Schneeflocke





Erweiterung: Transformation

- Allgemein: **Transformation** von Grafiken

```
transform :: S.AttributeValue → Draw() → Draw()
```

- Speziell:
  - Rotation um einen Punkt:

```
rotate :: Point → Double → Draw () → Draw ()
```

  - Skalierung um einen Faktor:

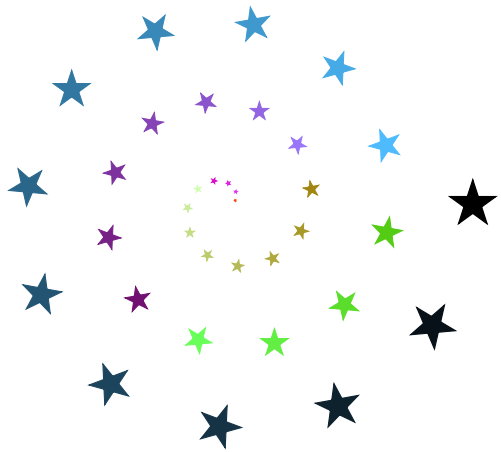
```
scale :: Double → Draw() → Draw ()
```

  - Verschiebung um einen Vektor (Punkt):

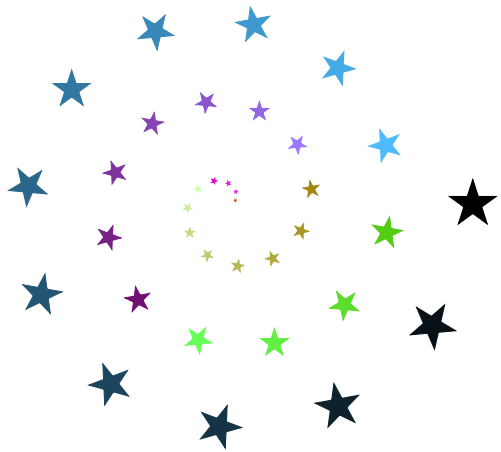
```
translate :: Point → Draw () → Draw ()
```

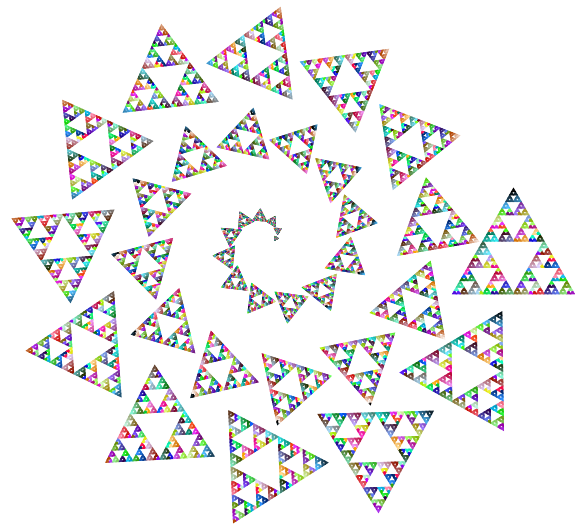
Beispiele: Verschiebung und Skalierung

Beispiele: Verschiebung und Skalierung



Beispiele: Verschiebung und Skalierung





Implementierung

- Abbildung auf weitere DSL: `Text.Blaze.SVG`
- Industrie-taugliche Modellierung von Markup-Sprachen (SVG, HTML)
  - Kann mehr als `String` (`ByteString`, `Text`)
  - Skaliert auch für große Texte
- Umgeht Probleme:
  - `String` nicht effizient
  - Insbesondere Zeichenketten von vorne aufzubauen ist **ineffizient**
- Aber noch nahe an SVG/HTML: keine Typisierung, keine Ausdrucksmächtigkeit

Weitere Abgrenzung

Programmierschnittstellen (APIs)

- Etwa jUnit: `assertTrue()`, `assertEquals()` Methoden & `@Before`, `@Test`, `@After` Annotationen
- Funktionsnamen spiegeln ebenfalls Domänenvokabular wider
- Gängige Sprachen (Java, C/C++) erschweren weitere Abstraktion: Syntaxerweiterungen, Konzepte höherer Ordnung
- Imperative Programmiersprache vs. deklarative DSL

Weitere Abgrenzung

Programmierschnittstellen (APIs)

- Etwa jUnit: `assertTrue()`, `assertEquals()` Methoden & `@Before`, `@Test`, `@After` Annotationen
- Funktionsnamen spiegeln ebenfalls Domänenvokabular wider
- Gängige Sprachen (Java, C/C++) erschweren weitere Abstraktion: Syntaxerweiterungen, Konzepte höherer Ordnung
- Imperative Programmiersprache vs. deklarative DSL

Skriptsprachen

- JavaScript, PHP, Lua, Tcl, Ruby werden für DS-artige Aufgaben verwendet
  - HTML/XML DOM-Manipulation
  - Game Scripting, GUIs, ...
  - Webprogrammierung (Ruby on Rails)
- Grundausrichtung: programmatische Erweiterung von Systemen

Beispiel: Hardware Description Languages

- Ziel: Funktionalität von Schaltkreisen beschreiben
- Einfachster Fall:

```
and  ::  Bool→ Bool→ Bool
or   ::  Bool→ Bool→ Bool
```

- Moderne Schaltkreise sind etwas komplizierter ...

CλaSH

- Modellierung und Simulation von Schaltkreisen in Haskell
- Typ Signal α für synchrone sequentielle Schaltkreise
- Rekursion für Feedback
- Simulation des Verhalten des Schaltkreises möglich
- Generiert VHDL, Verilog, SystemVerilog, und Testdaten

- Verwandt: Chisel (in Scala), Bluespec (kommerziell), Lava (veraltet)

Beispiel: SQL

- SQL-Anfragen werden in Haskell modelliert, dann übersetzt und an DB geschickt
- Vorteil: typsicher, ausdrucksstark
- Wie modelliert man das **Ergebnis**? $\rightarrow$ Abbildung Haskell-Typen auf DB
- Haskell: Squeal
- Scala: Slick

Vorteile der Verwendung von DSLs

- Ausdruck von Problemen/Lösungen in der Sprache und auf dem Abstraktionslevel der Anwendungsdomäne
- Notation matters: Programmiersprachen bieten oftmals nicht die Möglichkeit, Konstrukte der Domäne angemessen wiederzugeben
- DSL-Lösungen sind oftmals selbstdokumentierend und knapp
- Bessere (automatische) Analyse, Optimierung und Testfallgenerierung von Programmen
  - Klar umrissene Domänensemantik
  - eingeschränkte Sprachmächtigkeit $\Rightarrow$ weniger Berechenbarkeitsfallen
- Leichter von Nicht-Programmierern zu erlernen als GPLs

Nachteile der Verwendung von DSLs

- Hohe initiale Entwicklungskosten
- Schulungsbedarf
- Sprachdesign ist eine äußerst schwierige und komplexe Angelegenheit, deren Aufwand nahezu immer unterschätzt wird
- Fehlender Tool-Support
  - Debugger
  - Generierung von (Online-)Dokumentation
  - Statische Analysen, ...
- Effizienz: Interpretation ggf. langsamer als direkte Implementierung in GPL

Zusammenfassung

- DSL: Maßgeschneiderte Sprache für wohldefinierten Problemkreis
- Vorteile: näher am Problem, näher an der Lösung
- Nachteile: Initialer Aufwand
- Klassifikation von DSLs:
  - Flache vs. tiefe Einbettung
  - Stand-alone vs. embedded
- Nächste Woche: Scala — eine Einführung.

Literatur

Koen Claessen and David Sands.
Observable sharing for functional circuit description.
In P. S. Thiagarajan and R. Yap, editors, *Advances in Computing Science – ASIAN'99*, volume 1742 of *LNCS*, pages 62–73, 1999.

Paul Hudak.
Building domain-specific embedded languages.
ACM Comput. Surv., 28, 1996.

Marjan Mernik, Jan Heering, and Anthony M. Sloane.
When and how to develop domain-specific languages.
ACM Comput. Surv., 37(4):316–344, 2005.

Arie van Deursen, Paul Klint, and Joost Visser.
Domain-specific languages: an annotated bibliography.
SIGPLAN Not., 35(6):26–36, 2000.

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 13 vom 22.01.19: Scala — Eine praktische Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

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
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Organisatorisches

- Anmeldung zur E-Klausur: ab Mitte der Woche
- Evaluation der Veranstaltung auf stud.ip: **bitte teilnehmen!**

Heute: Scala

- A **sca**lable **la**nguage
- 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)

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?

```
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**)

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
- Companion objects (**object**)

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

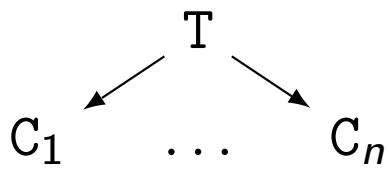
Implementierung algebraischer Datentypen

Haskell:

```
data T = C1 | ... | Cn
```

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



- Varianten als **Subtypen**
- Problem und Vorteil:

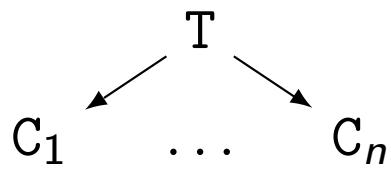
Implementierung algebraischer Datentypen

Haskell:

```
data T = C1 | ... | Cn
```

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



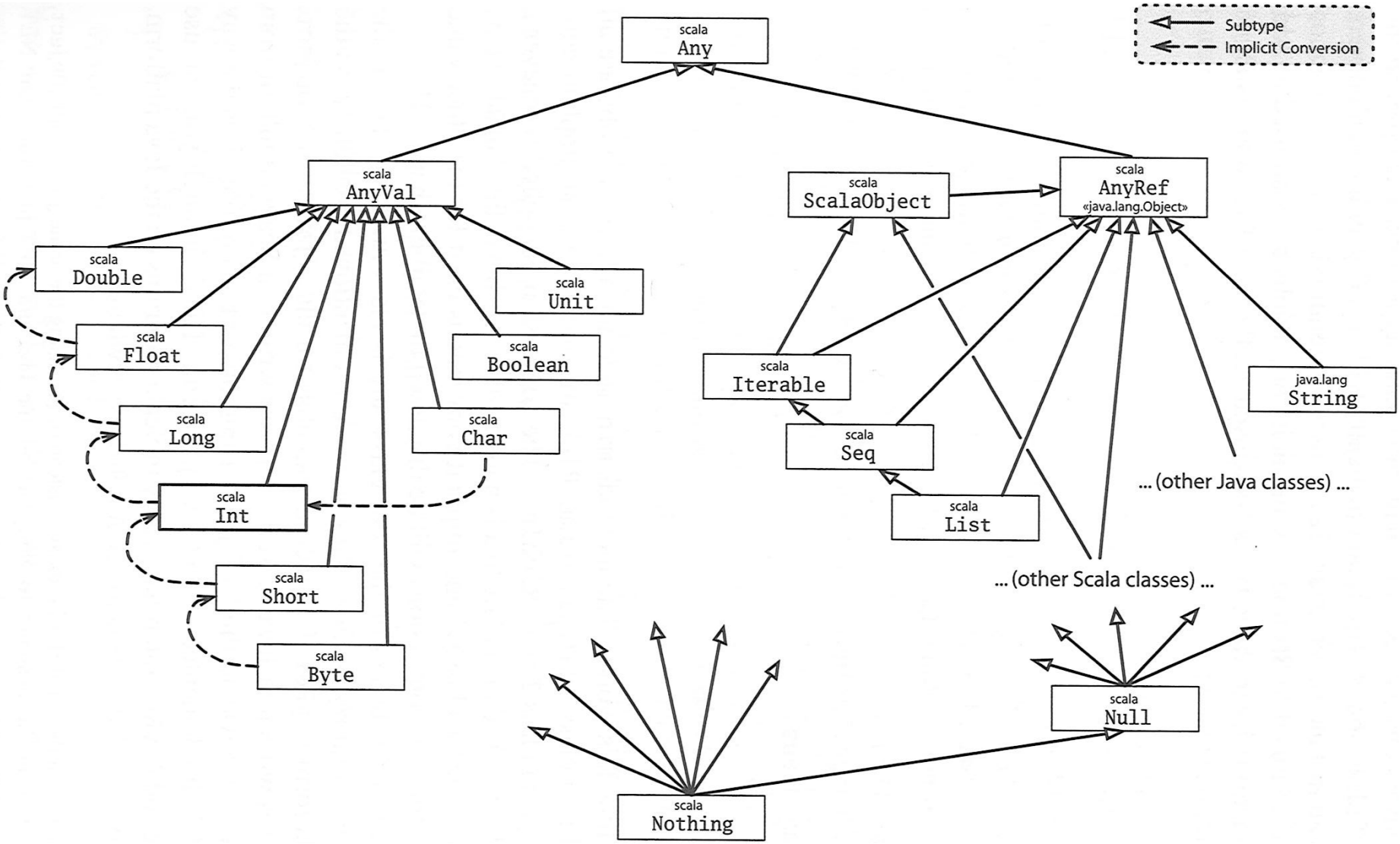
- Varianten als **Subtypen**
- Problem und Vorteil: **Erweiterbarkeit**
- **sealed** verhindert Erweiterung

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*

Parametrische Polymorphie

- Typparameter (wie in Haskell, Generics in Java), Bsp. $\mathsf{List}\,[\mathsf{T}]$
- Problem: Vererbung und Polymorphie
- Ziel: wenn $\mathsf{S} < \mathsf{T}$, dann $\mathsf{List}\,[\mathsf{S}] < \mathsf{List}\,[\mathsf{T}]$
- **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 $Z \longrightarrow X \subseteq Z \longrightarrow Y$, aber $X \longrightarrow Z \not\subseteq Y \longrightarrow Z$
  - Sondern $Y \longrightarrow Z \subseteq X \longrightarrow Z$

Typvarianz

class C[+T]

- **Kovariant**
- Wenn S $<$ T, dann C[S] $<$ C[T]
- Parametertyp T nur im Wertebereich von Methoden

class C[T]

- **Rigide**
- Kein Subtyping
- Parametertyp T kann beliebig verwendet werden

class C[−T]

- **Kontravariant**
- Wenn S $<$ T, dann C[T] $<$ C[S]
- Parametertyp T nur im Definitionsbereich von Methoden

Beispiel:

```
class Function[-S, +T] {
  def apply(x:S) : T
  }
```

Traits: `05-Funny.scala`

Was sehen wir hier?

- Trait (Mix-ins): abstrakte Klassen, Interfaces; Haskell: Typklassen
- „Abstrakte Klassen ohne Konstruktur“
- Unterschied zu Klassen:
  - Mehrfachvererbung möglich
  - Keine feste Oberklasse (**super** dynamisch gebunden)
  - Nützlich zur Strukturierung (Aspektorientierung)
- Nützlich zur Strukturierung:

$$\textit{thin interface} + \textit{trait} = \textit{rich interface}$$

Beispiel: `05-Ordered.scala`, `05-Rational.scala`

More Traits

- Ad-Hoc Polymorphie mit Traits
- Typklasse:

```
trait Show[T] {
  def show(value: T): String
}
```

- Instanz:

```
implicit object ShowInt extends Show[Int] {
  def show(value: Int) = value.toString
}
```

- Benutzung:

```
def print[T](value: T)(implicit show: Show[T])= {
  println(show.show(value));
}
```

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

Schlammschlacht der Programmiersprachen

	Haskell	Scala	Java
Klassen und Objekte	-	+	+
Funktionen höherer Ordnung	+	+	-
Typinferenz	+	(+)	-
Parametrische Polymorphie	+	+	+
Ad-hoc-Polymorphie	+	+	-
Typsichere Metaprogrammierung	+	+	-

Alle: Nebenläufigkeit, Garbage Collection, FFI

Scala — Die Sprache

- 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 **Reaktive Programmierung** (SoSe 2017)

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 14 vom 29.01.19: Rückblick & Ausblick

Christoph Lüth

Universität Bremen

Wintersemester 2018/19

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
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Organisatorisches

- Bitte fü die Programmierübung (“E-Klausur”) anmelden (stud.ip)
- Bitte an der **Online-Evaluation** teilnehmen (stud.ip)

Inhalt

- E-Klausur
- Rückblick und Ausblick

Elektronische Programmierübung

Erinnerung: Scheinkriterien

- Elektronische Klausur am Ende (Individualität der Leistung)
- Mind. 50% in allen Übungsblättern und mind. 50% in der E-Klausur
- Note = 50% Übungsblätter und 50% E-Klausur
- **Notenspiegel** (in Prozent aller Punkte):

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

Aufbau

- Kleine **Programmierübungen**
  - Rahmen vorgegeben, mit kurzen Unit-Tests
  - Tests sind nicht vollständing — Erfüllung **notwendig** aber nicht **hinreichend**.
  - Ziel: Prüfung **elementarer Haskellkenntnisse** (Individualität der Prüfungsleistung)
- **Verständnisfragen**
  - Multiple-Choice-Tests
  - Zusatzaufgaben — Übung auch ohne Verständnisfragen zu bestehen
  - Ziel: Prüfung des **vertieften Verständnisses** des Stoffs
- Wertung: Programmierübung 80%, Verständnisfragen 20%

Beispiel Programmierübungen

Definieren Sie eine eine Funktion

```
ostern :: String→ Int
```

die zählt, wie oft in einer Zeichenkette die Zeichenkette "ei" enthalten ist.

Beispiel:

```
ostern "ei, ei, oh, eiaiei" ⇝ 4
```

Beispiel Programmierübungen

Definieren Sie eine Funkion concatSnd, welche eine Liste aus Paaren von Elementen und Listen auf eine Liste von Paaren von Elementen abbildet (also die Eingabelisten der zweiten Komponente konkateniert).

Beispiel:

```
concatSnd [(True, "xy"), (False, "foo")] ⇝
   [(True,'x'),(True,'y'),(False,'f'), (False,'o'),(False,'o')]
concatSnd [(1, [2, 3]), (7, [9, 5])] ⇝
   [(1, 2), (1, 3), (7, 9), (7, 5)]
```

Beispiel Programmierübung

Eine Matrix ist als Liste ihrer Spaltenvektoren dargestellt:

```
data Matrix a = M [[a]]
```

Schreiben Sie eine Funktion

```
row :: Matrix a → Int → [a]
```

die die *i*-te Zeile (gezählt ab 1) einer Matrix zurückgibt.

Beispiel:

```
row (M [[3,7,5],[9,2,0],[5,8,1]]) 2 ⇝ [7,2,8]
```

Beispiel Programmierübung

Definieren Sie eine Funktion

```
subseqs :: [a]→ [[a]]
```

welche die nichtleeren Teillisten einer Liste berechnet.

Beispiel:

```
subseqs "pi3" ⇝ ["p", "pi", "pi3", "i", "i3", "3"]
```

Beispiel: Verständnisfrage

Betrachten Sie folgende Werte:

```
Otto
Karl Otto "Heinz"
Karl (Karl Otto [1,7]) "17"
```

Für welche der folgenden Typdeklarationen sind diese Werte wohlgetypt:

- ☐ **data** T a = Otto | Karl (T a) [a]
- ☐ **data** T a b = Otto | Karl a b
- ☐ **data** T a = Otto | Karl (T a) String
- ☐ **data** T a b = Otto a | Karl b [a]

Beispiel: Verständnisfrage

Betrachten Sie folgende Werte:

```
Otto
Karl Otto "Heinz"
Karl (Karl Otto [1,7]) "17"
```

Für welche der folgenden Typdeklarationen sind diese Werte wohlgetypt:

- ☐ **data** T a = Otto | Karl (T a) [a]
- ☑ **data** T a b = Otto | Karl a b
- ☐ **data** T a = Otto | Karl (T a) String
- ☑ **data** T a b = Otto a | Karl b [a]

Beispiel: Verständnisfrage

Betrachten Sie folgende Funktionsdefinition:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

count :: Ord a⇒ a→ Tree a→ Int
count _ Leaf = 0
count a (Node l b r) | a < b  = count a l
                     | a == b = 1+ count a r
                     | a > b  = count a r
```

Welche der folgenden Invarianten eines Baumes `Node l a r` stellt sicher, dass `count` korrekt ist:

- ☐ `a` ist kleiner-gleich allem in `l` und größer-gleich allem in `r`
- ☐ `a` ist größer-gleich allem in `l` und kleiner-gleich allem in `r`
- ☐ `a` ist größer als alles in `l` und kleiner-gleich allem in `r`
- ☐ `a` ist kleiner-gleich allem in `l` und größer als alles in `r`

Beispiel: Verständnisfrage

Betrachten Sie folgende Funktionsdefinition:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

count :: Ord a⇒ a→ Tree a→ Int
count _ Leaf = 0
count a (Node l b r) | a < b  = count a l
                     | a == b = 1+ count a r
                     | a > b  = count a r
```

Welche der folgenden Invarianten eines Baumes `Node l a r` stellt sicher, dass `count` korrekt ist:

- ☐ `a` ist kleiner-gleich allem in `l` und größer-gleich allem in `r`
- ☐ `a` ist größer-gleich allem in `l` und kleiner-gleich allem in `r`
- ☑ `a` ist größer als alles in `l` und kleiner-gleich allem in `r`
- ☐ `a` ist kleiner-gleich allem in `l` und größer als alles in `r`

Beispiel: Verständnisfrage

Betrachten Sie folgende Funktionsdefinition:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

count :: Ord a⇒ a→ Tree a→ Int
count _ Leaf = 0
count a (Node l b r) | a < b  = count a l
                     | a == b = 1+ count a r
                     | a > b  = count a r
```

Welche der folgenden Eigenschaften erfüllt count?

- ☐ count ist **injektiv**
- ☐ count ist **total**
- ☐ count ist **partiell**
- ☐ count ist **strikt** im **ersten** Argument
- ☐ count ist **strikt** im **zweiten** Argument

Beispiel: Verständnisfrage

Betrachten Sie folgende Funktionsdefinition:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

count :: Ord a⇒ a→ Tree a→ Int
count _ Leaf = 0
count a (Node l b r) | a < b  = count a l
                     | a == b = 1+ count a r
                     | a > b  = count a r
```

Welche der folgenden Eigenschaften erfüllt count?

- ☐ count ist **injektiv**
- ☑ count ist **total**
- ☐ count ist **partiell**
- ☐ count ist **strikt** im **ersten** Argument
- ☑ count ist **strikt** im **zweiten** Argument

Beispiel: Verständnisfrage

Gegeben folgende Funktionsdefinition:

```
f :: a→ [a]→ [a]
f a (b:bs) = b: f a bs
f a []     = [a]
```

Welche Definitionen sind **äquivalent**?

- ☐ `f1 x xs = foldr (:) xs [x]`
- ☐ `f2 = (flip (++) ∘ (:[]))`
- ☐ `f3 x xs = foldl (flip (:)) xs x`
- ☐ `f4 a = (++ [a])`

Beispiel: Verständnisfrage

Gegeben folgende Funktionsdefinition:

```
f :: a→ [a]→ [a]
f a (b:bs) = b: f a bs
f a []     = [a]
```

Welche Definitionen sind **äquivalent**?

- ☐ `f1 x xs = foldr (:) xs [x]`
- ☑ `f2 = (flip (++) ∘ (:[]))`
- ☐ `f3 x xs = foldl (flip (:)) xs x`
- ☑ `f4 a = (++ [a])`

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 unterstütze 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: Sigma — Fighting spam with Haskell
- Finanzindustrie: Barclays Capital, Credit Suisse, Deutsche Bank
- Bluespec: Schaltkreisentwicklung, DSL auf Haskell-Basis
- Galois, Inc: Cryptography (Cryptol DSL)
- Siehe auch: Haskell in Industry

Funktionale Programmierung in der Industrie

- **Scala**:
  - Twitter, Foursquare, Guardian, ...
- **Erlang**
  - schwach typisiert, nebenläufig, strikt
  - Fa. Ericsson (Telekom-Anwendungen), WhatsApp
- `FL`
  - ML-artige Sprache
  - Chip-Verifikation der Fa. Intel
- **Python** (und andere Skriptsprachen):
  - Listen, Funktionen höherer Ordnung (map, fold), anonyme Funktionen, Listenkomprehension

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** (Sommersemester 2019)
  - 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 2019/20 — **meldet Euch** bei Thomas Barkowsky (oder bei mir)!

Tschüß!