Praktische Informatik 3: Funktionale Programmierung
Vorlesung 1 vom 13.10.2014: Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Personal

- Vorlesung:
  Christoph Lüth cxl@informatik.uni-bremen.de
  MZH 3110, Tel. 59830
- Tutoren:

Jan Radtke	jradtke@informatik.uni-bremen.de
Sandor Herms	sanherms@informatik.uni-bremen.de
Daniel Müller	dmueller@informatik.uni-bremen.de
Felix Thielke	fthielke@informatik.uni-bremen.de
Sören Schulze	sschulze@informatik.uni-bremen.de
Henrik Reichmann	henrikr@informatik.uni-bremen.de

- Fragestunde:
  Berthold Hoffmann hof@informatik.uni-bremen.de
- Webseite: www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws14

Termine

- Vorlesung: Di 12 – 14, MZH 1380/1400
- Tutorien:

Mi	08 – 10	MZH 1110	Sören Schulze
Mi	10 – 12	MZH 1470	Sandor Herms
Mi	12 – 14	MZH 1110	Henrik Reichmann
Mi	14 – 16	SFG 1020	Felix Thielke
Do	08 – 10	MZH 1110	Jan Radtke
Do	10 – 12	MZH 1090	Daniel Müller

- Fragestunde : Di 10 – 12 Berthold Hoffmann (Cartesium 2.048)
- Anmeldung zu den Übungsgruppen über stud.ip

Übungsbetrieb

- Ausgabe der Übungsblätter über die Webseite Dienstag abend
- Besprechung der Übungsblätter in den Tutorien
- Bearbeitungszeit: eine Woche
- Abgabe: elektronisch bis Freitag nächste Woche 12:00
- Elf Übungsblätter (voraussichtlich) plus 0. Übungsblatt
- Übungsgruppen: max. drei Teilnehmer (nur in Ausnahmefällen vier)

Scheinkriterien

- Von n Übungsblättern werden $n-1$ bewertet (geplant $n=11$)
- Insgesamt mind. 50% aller Punkte
- 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

- Fachgespräch (Individualität der Leistung) am Ende

Spielregeln

- Quellen angeben bei
  - Gruppenübergreifender Zusammenarbeit;
  - Internetrecherche, Literatur, etc.
- Erster Täuschungsversuch: Null Punkte
- Zweiter Täuschungsversuch: Kein Schein.
- Deadline verpaßt?
  - Triftiger Grund (z.B. Krankheit mehrerer Gruppenmitglieder)
  - Vorher ankündigen, sonst null Punkte.

Fahrplan

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

Warum funktionale Programmierung lernen?

- Denken in Algorithmen, nicht in Programmiersprachen
- Abstraktion: Konzentration auf das Wesentliche
- Wesentliche Elemente moderner Programmierung:
  - Datenabstraktion und Funktionale Abstraktion
  - Modularisierung
  - Typisierung und Spezifikation

The Future is Bright — The Future is Functional

- Blick über den Tellerrand — Blick in die Zukunft
- Studium $\neq$ Programmierkurs — was kommt in 10 Jahren?
- Funktionale Programmierung ist bereit für die Herausforderungen der Zukunft:
  - Nebenläufige Systeme (Mehrkernarchitekturen)
  - Vielfach vernetzte Rechner („Internet der Dinge“)
  - Große Datenmengen („Big Data“)

Warum Haskell?

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

Geschichtliches

- 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)
- Konsolidierung 1990
  - CAML, Formale Semantik für Standard ML
  - Haskell als Standardsprache
- Kommerzialisierung 2010
  - Scala, Clojure, F#

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:

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

Beispiel: Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

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

- Auswertung durch Reduktion von Ausdrücken:

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

Beispiel: Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

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

- Auswertung durch Reduktion von Ausdrücken:

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

Beispiel: Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

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

- Auswertung durch Reduktion von Ausdrücken:

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

Beispiel: Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

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

- Auswertung durch Reduktion von Ausdrücken:

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

Beispiel: Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

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

- Auswertung durch Reduktion von Ausdrücken:

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

Beispiel: Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

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

- Auswertung durch Reduktion von Ausdrücken:

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

Beispiel: Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

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

- Auswertung durch Reduktion von Ausdrücken:

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

Beispiel: Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

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

- Auswertung durch Reduktion von Ausdrücken:

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

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

```
repeat 2 "hallo "
```

Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

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

- Auswertung:

```
repeat 2 "hallo "
→ 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

- Verschiedene `Typen`:
  - Basistypen (Zahlen, Zeichen)
  - strukturierte Typen (Listen, Tupel, etc)

Typisierung

- Typen unterscheiden Arten von Ausdrücken und Werten:

  `repeat n s = ...`

n	Zahl
s	Zeichenkette

- Verschiedene `Typen`:
  - Basistypen (Zahlen, Zeichen)
  - strukturierte Typen (Listen, Tupel, etc)
- Wozu Typen?
  - Typüberprüfung während Übersetzung erspart Laufzeitfehler
  - Programmsicherheit

Signaturen

- Jede Funktion hat eine Signatur

```
fac     :: Integer→ Integer

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 $==$, $\neq$, $\leq$, $<$, ...
- 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 $\rightarrow$ Double
  - fromInteger :: Integer $\rightarrow$ Double
  - round, truncate :: Double $\rightarrow$ 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
  - Strukturierte Typen: Listen, Tupel
  - Jede Funktion `f` hat eine Signatur $\mathsf{f} :: \mathsf{a} \to \mathsf{b}$

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 2 vom 21.10.2014: Funktionen und Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Rekursive Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
  - Typinferenz
- 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
- Definition von Datentypen
  - Aufzählungen
  - Produkte

Organisatorisches

- Verteilung der Tutorien (laut stud.ip):

Mi	08 – 10	MZH 1110	Sören Schulze	11
Mi	10 – 12	MZH 1470	Sandor Herms	46
Mi	12 – 14	MZH 1110	Henrik Reichmann	19
Mi	14 – 16	SFG 1020	Felix Thielke	14
Do	08 – 10	MZH 1110	Jan Radtke	37
Do	10 – 12	MZH 1090	Daniel Müller	40
Nicht zugeordnet				30

- Insgesamt: 197 Studenten, optimale Größe: ca. 33
- Bitte auf die kleineren Tutorien umverteilen, wenn möglich.

Definition von Funktionen

Definition von Funktionen

- Zwei wesentliche Konstrukte:
  - Fallunterscheidung
  - Rekursion

Satz

Fallunterscheidung und Rekursion auf natürlichen Zahlen sind Turing-mächtig.

- Funktion kann partiell sein.

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: Charakteristika

- Leichtgewichtig
  - Wichtigstes Zeichen:
- Funktionsapplikation: f a
  - Keine Klammern
  - Höchste Priorität (engste Bindung)
- Abseitsregel: Gültigkeitsbereich durch Einrückung
  - Keine Klammern
- Auch in anderen Sprachen (Python, Ruby)

Haskell-Syntax I: Die Abseitsregel

Funktionsdefinition:

$$\mathtt{f}\ \mathtt{x}_1\ \mathtt{x}_2 \ldots \mathtt{x}_n = \mathtt{E}$$

- Geltungsbereich 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

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 = Q where
    y = M
    f x = N x
```

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

- 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 Funktionen

Bedeutung (Semantik) von Programmen

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

$$t \to_P v \iff [\![P]\!](t) = v$$

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):
  inc (double (inc 3)) →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

inc (double (inc 3)) → double (inc 3)+ 1
→

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

inc (double (inc 3)) → double (inc 3)+ 1
→ 2*(inc 3)+ 1
→

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

inc (double (inc 3)) → double (inc 3)+ 1
→ 2*(inc 3)+ 1
→ 2*(3+ 1)+ 1
→ 2*4+1 → 9

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) → inc (double (3+1))
  →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) → inc (double (3+1))
  → inc (2*(3+ 1))
  →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) → inc (double (3+1))
  → inc (2*(3+ 1))
  → (2*(3+ 1))+ 1
  →

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+ 1
```

```
double :: Int → Int
double x = 2∗x
```

- Reduktion von inc (double (inc 3))
- Von außen nach innen (outermost-first):

  inc (double (inc 3)) → double (inc 3)+ 1
  → 2*(inc 3)+ 1
  → 2*(3+ 1)+ 1
  → 2*4+1 → 9

- Von innen nach außen (innermost-first):

  inc (double (inc 3)) → inc (double (3+1))
  → inc (2*(3+ 1))
  → (2*(3+ 1))+ 1
  → 2*4+1 → 9

Konfluenz und Termination

Sei $\xrightarrow{*}$ die Reduktion in null oder mehr Schritten.

Definition (Konfluenz)

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

Definition (Termination)

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

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

Auswertungsstrategien

- Wenn wir von Laufzeitfehlern abstrahieren, gilt:

Theorem (Konfluenz)

Funktionale Programme sind für jede Auswertungsstrategie konfluent.

Theorem (Normalform)

Terminierende funktionale Programme werten unter jeder Auswertungsstragie jeden Ausdruck zum gleichen Wert aus (der Normalform).

- Auswertungsstrategie für nicht-terminierende Programme relevant
- Nicht-Termination nötig (Turing-Mächtigkeit)

Auswirkung der Auswertungsstrategie

- Outermost-first entspricht call-by-need, verzögerte Auswertung.
- Innermost-first entspricht call-by-value, strikte Auswertung
- 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

Striktheit

Definition (Striktheit)

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

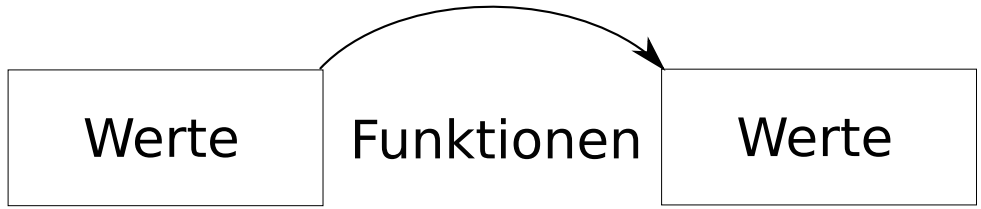
- Denotationelle Eigenschaft (nicht operational)
- Java, C etc. sind call-by-value (nach Sprachdefinition) und damit strikt
- Haskell ist nicht-strikt (nach Sprachdefinition)
  - repeat0 undef muss "" ergeben.
  - Meisten Implementationen nutzen verzögerte Auswertung
- Fallunterscheidung ist immer nicht-strikt.

Datentypen

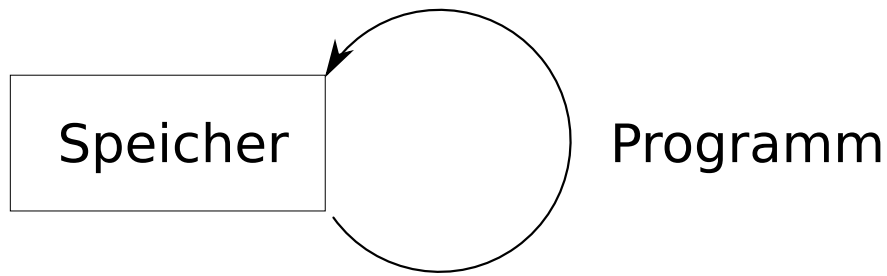
Datentypen als Modellierungskonstrukt

Programme manipulieren ein Modell (der Umwelt)

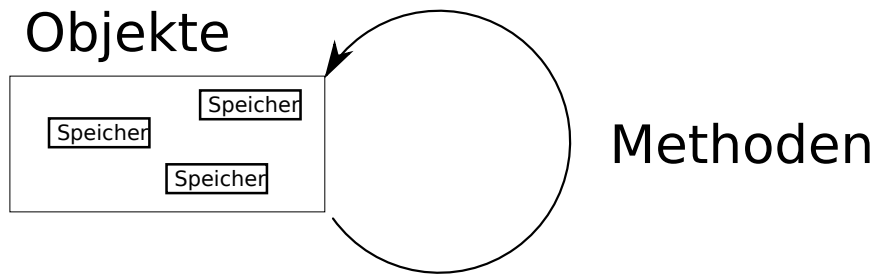
- Funktionale Sicht:



- Imperative Sicht:



- Objektorientierte Sicht:



Typkonstruktoren

- Aufzählungen
- Produkt
- Rekursion
- Funktionsraum

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

- 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

- 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):

$$
\begin{array}{l}
f\ c_1 == e_1 \\
\ldots \\
f\ c_n == e_n
\end{array}
\quad \longrightarrow \quad
f\ x == \textbf{case}\ x\ \textbf{of}\ \begin{array}{l}
c_1 \rightarrow e_1, \\
\ldots \\
c_n \rightarrow e_n
\end{array}
$$

- Damit:

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

Der einfachste Aufzählungstyp

- Einfachste Aufzählung: Wahrheitswerte

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

  - Genau zwei unterschiedliche Werte

- Definition von Funktionen:

  - Wertetabellen sind explizite Fallunterscheidungen

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

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

Wahrheitswerte: Bool

- Vordefiniert als

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

- Vordefinierte Funktionen:

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

- Konjunktion definiert als

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

- &&, || sind rechts nicht strikt
  - 1 ==0 && div 1 0 ==0 $\rightsquigarrow$ False
- **if** _ **then** _ **else** _ als syntaktischer Zucker:

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

Beispiel: Ausschließende Disjunktion

- Mathematische Definiton:

```
exOr :: Bool→ Bool→ Bool
exOr x y = (x || y) && (not (x && y))
```

- Alternative 1: explizite Wertetabelle:

```
exOr False False = False
exOr True  False = True
exOr False True  = True
exOr True  True  = False
```

- Alternative 2: Fallunterscheidung auf ersten Argument

```
exOr True  y = not y
exOr False y = y
```

- Was ist am besten?
  - Effizienz, Lesbarkeit, Striktheit

Produkte

- Konstruktoren können Argumente haben
- Beispiel: Ein Datum besteht aus Tag, Monat, Jahr
- Mathematisch: Produkt (Tupel)

$$\begin{aligned} \mathit{Date} &= \{\mathit{Date}\ (n,\ m,\ y) \mid n \in \mathbb{N}, m \in \mathit{Month}, y \in \mathbb{N}\} \\ \mathit{Month} &= \{\mathit{Jan}, \mathit{Feb}, \mathit{Mar}, \ldots\} \end{aligned}$$

- Funktionsdefinition:
  - Konstruktorargumente sind gebundene Variablen

$$\begin{aligned} \mathit{year}(D(n,m,y)) &= y \\ \mathit{day}(D(n,m,y)) &= n \end{aligned}$$

  - Bei der Auswertung wird gebundene Variable durch konkretes Argument ersetzt

Produkte in Haskell

- Konstruktoren mit Argumenten

```
data Date  = Date Int Month Int
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec
```

- Beispielwerte:

```
today     = Date 21 Oct 2014
bloomsday = Date 16 Jun 1904
```

- Über Fallunterscheidung Zugriff auf Argumente der Konstruktoren:

```
day  :: Date→ Int
year :: Date→ Int
day  d = case d of Date t m y→ t
year (Date d m y) = y
```

Beispiel: Tag im Jahr

- Tag im Jahr: Tag im laufenden Monat plus Summe der Anzahl der Tage der vorherigen Monate

```
yearDay :: Date→ Int
yearDay (Date d m y) = d + sumPrevMonths m where
  sumPrevMonths :: Month→ Int
  sumPrevMonths Jan = 0
  sumPrevMonths m   = daysInMonth (prev m) y +
                      sumPrevMonths (prev m)
```

- Tage im Monat benötigt Jahr als Argument (Schaltjahr!)

```
daysInMonth :: Month→ Int→ Int
prev :: Month→ Month
```

- Schaltjahr: Gregorianischer Kalender

```
leapyear :: Int→ Bool
leapyear y = if mod y 100 == 0 then mod y 400 == 0
             else mod y 4 == 0
```

Beispiel: Produkte in Bob's Shoppe

- Käsesorten und deren Preise:

```
data Kaese = Gouda | Appenzeller
```

```
kpreis :: Kaese → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270
```

Beispiel: Produkte in Bob's Shoppe

- Käsesorten und deren Preise:

```
data Kaese = Gouda | Appenzeller

kpreis :: Kaese → Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270
```

- Alle Artikel:

```
data Artikel =
    Apfel Apfel   | Eier
  | Kaese Kaese   | Schinken
  | Salami        | Milch Bool
```

Beispiel: Produkte in Bob's Shoppe

- Mengenangaben:

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

- 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)(Kilo kg)    = Cent (round(kg∗
                                    kpreis k))
preis Schinken (Gramm g)    = Cent (g/100∗ 199)
preis Salami (Gramm g)      = Cent (g/100∗ 159)
preis (Milch bio) (Liter l) =
   Cent (round (l∗ if not bio then 69 else 119))
preis _ _                   = Ungueltig
```

Auswertung der Fallunterscheidung

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

```
data Foo = Foo Int | Bar

f :: Foo→ Int
f foo = case foo of Foo i → i; Bar → 0

g :: Foo→ Int
g foo = case foo of Foo i → 9; Bar → 0
```

- Auswertungen:

```
                f Bar   →   0
f (Foo undefined)   →   *** Exception: undefined
                g Bar   →   0
g (Foo undefined)   →   9
```

Der Allgemeine Fall: Algebraische Datentypen

Definition eines algebraischen Datentypen T:

$$\begin{aligned} \text{data T} = \quad & C_1\ t_{1,1} \ldots t_{1,k_1} \\ & \ldots \\ \mid \quad & C_n\ t_{n,1} \ldots t_{n,k_n} \end{aligned}$$

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

Diese Eigenschaften machen Fallunterscheidung möglich.

Rekursion? $\rightsquigarrow$ Nächste Vorlesung!

Zusammenfassung

- Striktheit
  - Haskell ist spezifiziert als nicht-strikt
- Datentypen und Funktionsdefinition dual
  - Aufzählungen — Fallunterscheidung
  - Produkte — Projektion
- Algebraische Datentypen
  - Drei wesentliche Eigenschaften der Konstruktoren
- Nächste Vorlesung: Rekursive Datentypen

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 3 vom 28.10.2014: Rekursive Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

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

Inhalt

- Rekursive Datentypen
  - Rekursive Definition
  - ... und wozu sie nützlich sind
  - Rekursive Datentypen in anderen Sprachen
  - Fallbeispiel: Labyrinthe

Der Allgemeine Fall: Algebraische Datentypen

Definition eines algebraischen Datentypen T:

$$\begin{aligned} \text{data T} = \quad & C_1\ t_{1,1} \ldots t_{1,k_1} \\ & \ldots \\ \mid \quad & C_n\ t_{n,1} \ldots t_{n,k_n} \end{aligned}$$

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

Diese Eigenschaften machen Fallunterscheidung möglich.

Der Allgemeine Fall: Algebraische Datentypen

Definition eines algebraischen Datentypen T:

$$\begin{aligned} \text{data T} = \quad & C_1\ t_{1,1} \ldots t_{1,k_1} \\ & \ldots \\ \mid \quad & C_n\ t_{n,1} \ldots t_{n,k_n} \end{aligned}$$

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

Diese Eigenschaften machen Fallunterscheidung möglich.

Heute: Rekursion

Rekursive Datentypen

- Der definierte Typ T kann rechts benutzt werden.
- Rekursive Datentypen sind unendlich.
- Entspricht induktiver Definition
- Modelliert Aggregation (Sammlung von Objekten)
- Funktionen werden durch Rekursion definiert.

Algebraische Datentypen: Nomenklatur

Gegeben Definition $\quad \textbf{data}\ T = C_1\ t_{1,1} \ldots t_{1,k_1}$

$\qquad\qquad\qquad\qquad\qquad\quad \ldots$

$\qquad\qquad\qquad\qquad\qquad\quad \mid\ C_n\ t_{n,1} \ldots t_{n,k_n}$

- C_i sind Konstruktoren
  - Immer vordefiniert
- Selektoren sind Funktionen $sel_{i,j}$:

  $sel_{i,j}\ (C_i\ t_{i,1} \ldots\ t_{i,k_i}) = t_{i,j}$
  - Partiell, linksinvers zu Konstruktor
  - Können vordefiniert werden (erweiterte Syntax der **data** Deklaration)
- Diskriminatoren sind Funktionen dis_i:

  $dis_i :: T \to Bool$

  $dis_i\ (C_i \ldots) = True$

  $dis_i\ \_ = False$
  - Definitionsbereichsbereich des Selektors sel_i
  - Nie vordefiniert

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 weiteres.

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

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

- Mengen sollen aggregiert werden, d.h. 35l Milch und 20l Milch werden zu 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)
```

- 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

- Mengen sollen aggregiert werden, d.h. 35l Milch und 20l Milch werden zu 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)
```

- 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
  - z.B. einlagern Eier (Liter 3.0) l

Einlagern (verbessert)

- 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
```

- Gesamtsumme berechnen:

```
kasse :: Einkaufswagen→ Int
kasse LeererWagen = 0
kasse (Einkauf a m e) = cent a m+ kasse e
```

Beispiel: Kassenbon

```
kassenbon :: Einkaufswagen→ String
```

Ausgabe:

```
Bob's Aulde Grocery Shoppe

Artikel                  Menge      Preis
-------------------------------------------
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:                            4.82 EU
```

Unveränderlicher Kopf

Ausgabe von Artikel und Mange (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
```

Rekursive Typen 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;
  }
```

Rekursive Typen in C

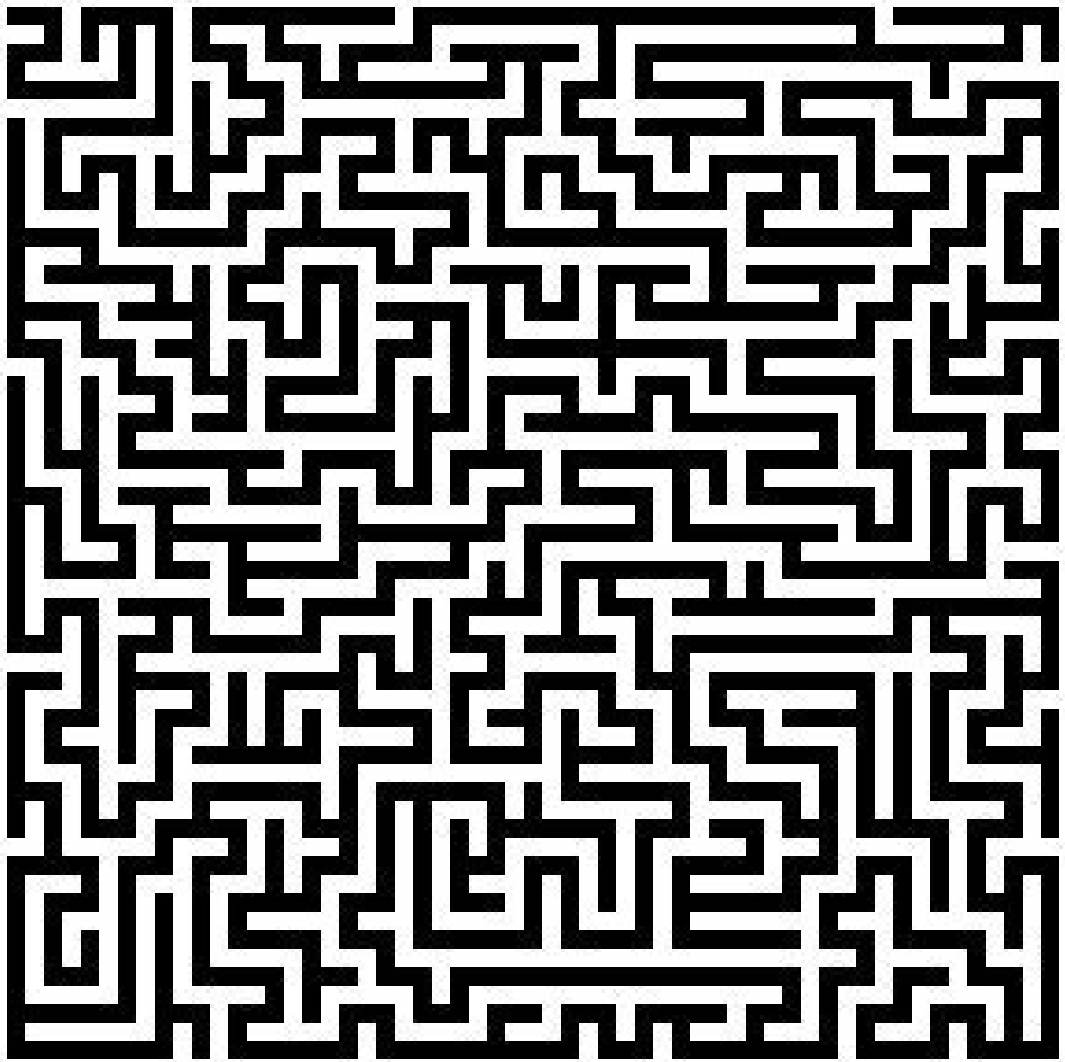
- C: Produkte, Aufzählungen, keine rekursiven Typen
- Rekursion durch Zeiger

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

- Konstruktoren nutzerimplementiert

```
list cons(void *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;
}
```

Fallbeispiel: Zyklische Datenstrukturen



Quelle: docs.gimp.org

Modellierung eines Labyrinths

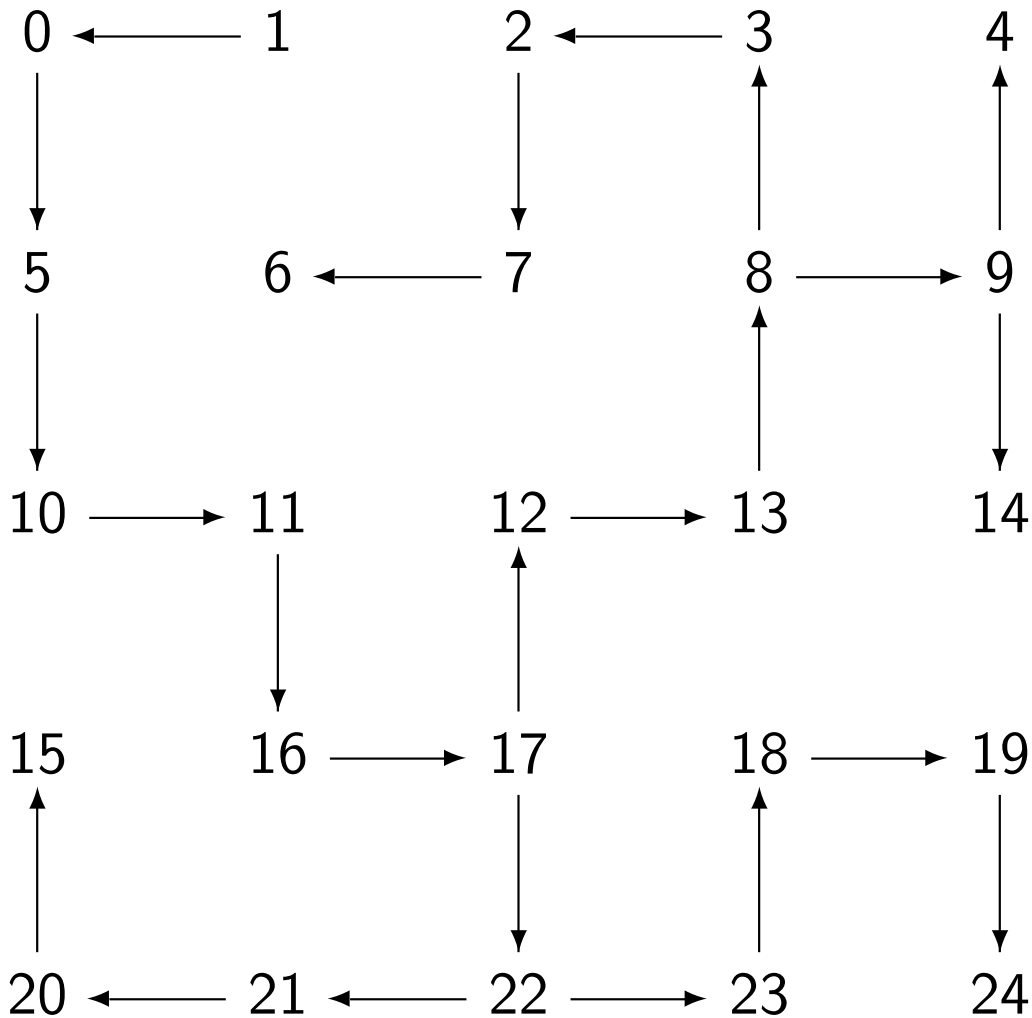
- Ein gerichtetes Labyrinth ist entweder
  - eine Sackgasse,
  - ein Weg, oder
  - eine Abzweigung in zwei Richtungen.

```
data Lab = Dead Id
         | Pass Id Lab
         | TJnc Id Lab Lab
```

- Ferner benötigt: eindeutige Bezeichner der Knoten

```
type Id  = Integer
```

Ein Labyrinth (zyklenfrei)



Traversion des Labyrinths

- Ziel: Pfad zu einem gegeben Ziel finden

- Benötigt Pfade und Traversion

```
data Path = Cons Id Path
          | Mt
```

```
data Trav = Succ Path
          | Fail
```

Traversionsstrategie

- Geht von zyklenfreien Labyrinth aus
- An jedem Knoten prüfen, ob Ziel erreicht, ansonsten
  - an Sackgasse Fail
  - an Passagen weiterlaufen
  - an Kreuzungen Auswahl treffen
- Erfordert Propagation von Fail:

```
cons :: Id → Trav → Trav
```

```
select :: Trav → Trav → Trav
```

Zyklenfreie Traversion

```
traverse1 :: Id→ Lab→ Trav
traverse1 t l
  | nid l == t = Succ (Cons (nid l) Mt)
  | otherwise = case l of
    Dead _ → Fail
    Pass i n → cons i (traverse1 t n)
    TJnc i n m → select (cons i (traverse1 t n))
                        (cons i (traverse1 t m))
```

Traversion mit Zyklen

- Veränderte Strategie: Pfad bis hierher übergeben
  - Pfad muss hinten erweitert werden.
- Wenn aktueller Knoten in bisherigen Pfad enthalten ist, Fail
- Ansonsten wie oben
- Neue Hilfsfunktionen:

```
contains :: Id → Path → Bool
```

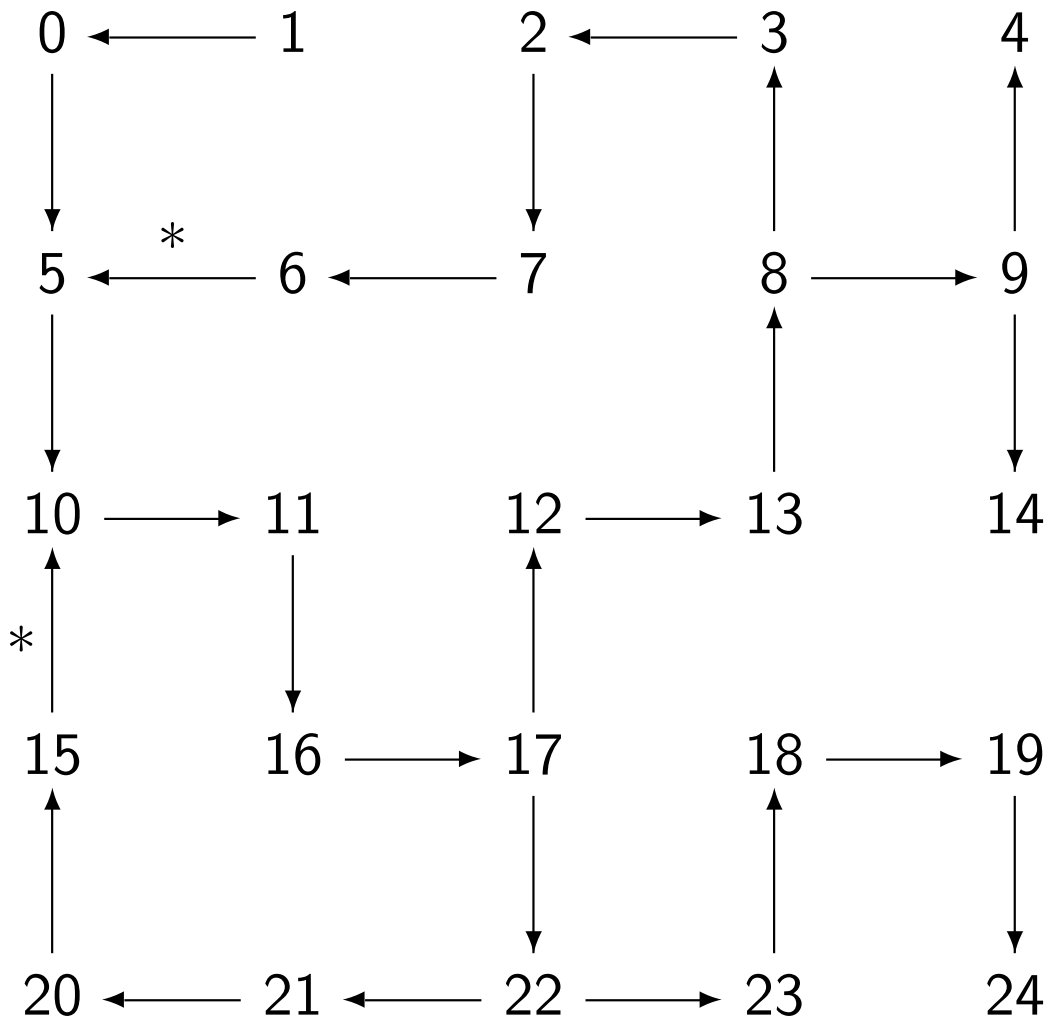
```
cat :: Path → Path → Path
```

```
snoc :: Path → Id → Path
```

Traversion mit Zyklen

```
traverse2 :: Id→ Lab→ Path→ Trav
traverse2 t l p
  | nid l == t = Succ (snoc p (nid l))
  | contains (nid l) p = Fail
  | otherwise = case l of
    Dead _ → Fail
    Pass i n → traverse2 t n (snoc p i)
    TJnc i n m → select (traverse2 t n (snoc p i))
                        (traverse2 t m (snoc p i))
```

Ein Labyrinth (mit Zyklen)



Ungerichtete Labyrinth

- In einem ungerichteten Labyrinth haben Passagen keine Richtung.
  - Sackgassen haben einen Nachbarn,
  - eine Passage hat zwei Nachbarn,
  - und eine Abzweigung drei Nachbarn.

```
data Lab = Dead Id Lab
         | Pass Id Lab Lab
         | TJnc Id Lab Lab Lab
```

- Andere Datentypen und Hilfsfunktionen bleiben (*mutatis mutandis*)
- Jedes nicht-leere ungerichtete Labyrinth hat Zyklen.
- Invariante (nicht durch Typ garantiert)

Traversion in ungerichteten Labyrinthen

- Traversionsfunktion wie vorher

```
traverse3 :: Id→ Lab→ Path→ Trav
traverse3 t l p
  | nid l == t = Succ (snoc p (nid l))
  | contains (nid l) p = Fail
  | otherwise = case l of
    Dead i n → traverse3 t n (snoc p i)
    Pass i n m → select (traverse3 t n (snoc p i))
                        (traverse3 t m (snoc p i))
    TJnc i n m k → select (traverse3 t n (snoc p i))
                          (select (traverse3 t m (snoc p i))
                                  (traverse3 t k (snoc p i)))
```

Zusammenfassung Labyrinth

- Labyrinth $\longrightarrow$ Graph oder Baum
- In Haskell: gleicher Datentyp
- Referenzen nicht explizit in Haskell
  - Keine undefinierten Referenzen (erhöhte Programmsicherheit)
  - Keine Gleichheit auf Referenzen
  - Gleichheit ist immer strukturell (oder selbstdefiniert)

Beispiel: Zeichenketten selbstgemacht

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

```
data MyString = Empty
              | Cons Char MyString
```

- Lineare Rekursion
  - Genau ein rekursiver Aufruf

Rekursive Definition

- Typisches Muster: Fallunterscheidung
  - Ein Fall pro Konstruktor
- Hier:
  - Leere Zeichenkette
  - Nichtleere Zeichenkette

Funktionen auf Zeichenketten

- Länge:

```
len :: MyString→ Int
len Empty          = 0
len (Cons c str) = 1+ len str
```

Funktionen auf Zeichenketten

- Länge:

```
len :: MyString→ Int
len Empty          = 0
len (Cons c str) = 1+ len str
```

- Verkettung:

```
cat :: MyString→ MyString→ MyString
cat Empty t        = t
cat (Cons c s) t = Cons c (cat s t)
```

Funktionen auf Zeichenketten

- Länge:

```
len :: MyString→ Int
len Empty          = 0
len (Cons c str) = 1+ len str
```

- Verkettung:

```
cat :: MyString→ MyString→ MyString
cat Empty t        = t
cat (Cons c s) t = Cons c (cat s t)
```

- Umkehrung:

```
rev :: MyString→ MyString
rev Empty        = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

Was haben wir gesehen?

- Strukturell ähnliche Typen:
  - Einkaufswagen, Path, MyString (Listen-ähnlich)
  - Resultat, Preis, Trav (Punktierte Typen)
- Ähnliche Funktionen darauf
- Besser: eine Typdefinition mit Funktionen, instantiierung zu verschiedenen Typen

$\rightsquigarrow$Nächste Vorlesung

Zusammenfassung

- Datentypen können rekursiv sein
- Rekursive Datentypen sind unendlich (induktiv)
- Funktionen werden rekursiv definiert
- Fallbeispiele: Einkaufen in Bob's Shoppe, Labyrinthtraversion
- Viele strukturell ähnliche Typen
- Nächste Woche: Abstraktion über Typen (Polymorphie)

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 4 vom 04.11.2014: Typvariablen und Polymorphie

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

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

Inhalt

- Letzte Vorlesung: rekursive 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 Path = Cons Id Path
          | Mt
```

```
data MyString = Empty
              | Cons Char MyString
```

- ein konstanter Konstruktor
- ein linear rekursiver Konstruktor

Ähnliche Funktionen der letzten Vorlesung

- Pfade:

```
cat :: Path→ Path→ Path
cat Mt q          = q
cat (Cons i p) q = Cons i (cat p q)
```

```
rev :: Path→ Path
rev Mt          = Mt
rev (Cons i p) = cat (rev p) (Cons i Mt)
```

- Zeichenketten:

```
cat :: MyString→ MyString→ MyString
cat Empty t       = t
cat (Cons c s) t = Cons c (cat s t)
```

```
rev :: MyString→ MyString
rev Empty       = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

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
```

```
len :: MyString→ Int
len Empty          = 0
len (Cons c str) = 1+ len str
```

- 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: Typvariablen

- Typvariablen abstrahieren über Typen

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

- α ist eine Typvariable
- α kann mit Id oder Char instantiiert werden
- List α ist ein polymorpher Datentyp
- Typvariable α wird bei Anwendung instantiiert
- Signatur der Konstruktoren

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

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 :: α→ List α→ List α
```

Polymorphe Funktionen

- Parametrische Polymorphie für Funktionen:

```
cat :: List α → List α → List α
cat Empty ys          = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
```

- Typvariable α wird bei Anwendung instantiiert:

```
cat (Cons 3 Empty) (Cons 5 (Cons 57 Empty))
cat (Cons 'p' (Cons 'i' Empty)) (Cons '3' Empty)
```

aber nicht

```
cat (Cons True Empty) (Cons 'a' (Cons 0 Empty))
```

- Typvariable: vergleichbar mit Funktionsparameter
- Restriktion: Typvariable auf Resultatposition?

Beispiel: Der Shop (refaktoriert)

- Einkaufswagen und Lager als Listen?
- Problem: zwei Typen als Argument
- Lösung 1: 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 1: 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 = Einkaufswagen [Posten]
```

Lösung 2: Tupel

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

```
data Pair α β = Pair α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

Lösung 2: Tupel

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

```
data Pair α β = Pair α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

- Beispielterm Typ

  Pair 4 'x'

Lösung 2: Tupel

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

```
data Pair α β = Pair α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

Beispielterm	Typ
Pair 4 'x'	Pair Int Char

Lösung 2: Tupel

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

```
data Pair α β = Pair α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

Beispielterm	Typ
Pair 4 'x'	Pair Int Char
Pair (Cons True Empty) 'a'	

Lösung 2: Tupel

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

```
data Pair α β = Pair α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

Beispielterm	Typ
Pair 4 'x'	Pair Int Char
Pair (Cons True Empty) 'a'	Pair (List Bool) Char

Lösung 2: Tupel

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

```
data Pair α β = Pair α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

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

Lösung 2: Tupel

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

```
data Pair α β = Pair α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

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

Lösung 2: Tupel

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

```
data Pair α β = Pair α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

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

Lösung 2: Tupel

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

```
data Pair α β = Pair α β
```

- Signatur des Konstruktors:

```
Pair :: α→ β→ Pair α β
```

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

Vordefinierte Datentypen: Tupel und Listen

- Eingebauter syntaktischer Zucker
- Tupel sind das kartesisches Produkt

```
data (α, β) = (α, β)
```

  - (a, b) = alle Kombinationen von Werten aus a und b
  - Auch n-Tupel: (a,b,c) etc.

- Listen

```
data [α] = [] | α : [α]
```

  - Weitere Abkürzungen: [x]=x:[], [x,y] =x:y:[] etc.

Vordefinierte Datentypen: Optionen

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

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

```
data Trav = Succ Path
          | Fail
```

Instanzen eines vordefinierten Typen:

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

Vordefinierten Funktionen (**import** Data.Maybe):

```
fromJust    :: Maybe α → α
fromMaybe   :: α→ Maybe α→ α
maybeToList :: Maybe α→ [α]
listToMaybe :: [α]→ Maybe α  — "sicheres" head
```

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 v. Paaren
unzip    :: [(α, β)] -> ([α], [β]) — Spaltet Liste v. 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 zur Eingabe:

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

- Beispiel:

```
cnt :: Char→ String→ Int
cnt c []     = 0
cnt c (x:xs) = if (c== x) then 1+ cnt c xs
               else cnt c xs
```

Zurück im Labyrinth

- Labyrinth als Instanz eines allgemeineren Datentyps?
- Erste Refaktorierung:

```
type Id   = Integer

type Path = [Id]

data Lab = Node Id [Lab]
```

Zurück im Labyrinth

- Labyrinth als Instanz eines allgemeineren Datentyps?
- Erste Refaktorierung:

```
type Id   = Integer
```

```
type Path = [Id]
```

```
data Lab = Node Id [Lab]
```

- Instanz eines variadischen Baumes

Labyrinth verallgemeinert: Variadische Bäume

- Variable Anzahl Kinderknoten: Liste von Kinderknoten

```
data VTree α = VNode α [VTree α]
```

- Anzahl Knoten zählen:

```
count :: VTree α→ Int
count (VNode _ ns) = 1+ count_nodes ns

count_nodes :: [VTree α] → Int
count_nodes []     = 0
count_nodes (t:ts) = count t+ count_nodes ts
```

Labyrinth verallgemeinert: Variadische Bäume

- Variable Anzahl Kinderknoten: Liste von Kinderknoten

```
data VTree α = VNode α [VTree α]
```

- Anzahl Knoten zählen:

```
count :: VTree α→ Int
count (VNode _ ns) = 1+ count_nodes ns
```

```
count_nodes :: [VTree α] → Int
count_nodes []     = 0
count_nodes (t:ts) = count t+ count_nodes ts
```

- Damit: das Labyrinth als variadischer Baum

```
type Lab = VTree Id
```

Ad-Hoc Polymorphie und Overloading

Definition (Überladung)

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

- Beispiel:
  - Gleichheit: $(==) :: \alpha \rightarrow \alpha \rightarrow \text{Bool}$
  - Vergleich: $(<) :: \alpha \rightarrow \alpha \rightarrow \text{Bool}$
  - 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 $(<)$ (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 :: (Eq 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 $\leq$

- Kann bei Instantiierung überschrieben werden

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
- Erstmal nicht relevant

Polymorphie in anderen Programmiersprachen: 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 anderen Programmiersprachen: 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));
```

Polymorphie in anderen Programmiersprachen: Java

- Ad-Hoc Polymorphie: Interface und abstrakte Klassen
- Flexibler in Java: beliebige Parameter etc.

Polymorphie in anderen Programmiersprachen: C

- "Polymorphie" in C: void *

```
struct list {
   void        *head;
   struct list *tail;
   }
```

- Gegeben:

```
int x = 7;
struct list s = { &x, NULL };
```

- s.head hat Typ void *:

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

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

Zusammenfassung

- Abstraktion über Typen
  - Uniforme Abstraktion: Typvariable, parametrische Polymorphie
  - Fallbasierte Abstraktion: Überladung, ad-hoc-Polymorphie
- In der Sprache:
  - Typklassen
  - polymorphe Funktionen und Datentypen
- Vordefinierte Typen: Listen `[a]` und Tupel `(a,b)`
- Nächste Woche: Abstraktion über Funktionen

$\rightsquigarrow$ Funktionen höherer Ordnung

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 5 vom 11.11.2014: Funktionen Höherer Ordnung I

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Rekursive Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
  - Typinferenz
- 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

Ähnliche Funktionen der letzten Vorlesung

- Pfade:

```
cat :: Path→ Path→ Path
cat Mt q          = q
cat (Cons i p) q = Cons i (cat p q)
```

```
rev :: Path→ Path
rev Mt          = Mt
rev (Cons i p) = cat (rev p) (Cons i Mt)
```

- Zeichenketten:

```
cat :: MyString→ MyString→ MyString
cat Empty t       = t
cat (Cons c s) t = Cons c (cat s t)
```

```
rev :: MyString→ MyString
rev Empty       = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

Ähnliche Funktionen der letzten Vorlesung

- Pfade:

```
cat :: Path→ Path→ Path
cat Mt q          = q
cat (Cons i p) q = Cons i (cat p q)
```

```
rev :: Path→ Path
rev Mt
rev (Co
```

Gelöst durch Polymorphie

- Zeichenl

```
cat :: MyString→ MyString→ MyString
cat Empty t       = t
cat (Cons c s) t = Cons c (cat s t)
```

```
rev :: MyString→ MyString
rev Empty       = Empty
rev (Cons c t) = cat (rev t) (Cons c Empty)
```

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
```

```
len :: MyString→ Int
len Empty          = 0
len (Cons c str) = 1+ len str
```

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
```

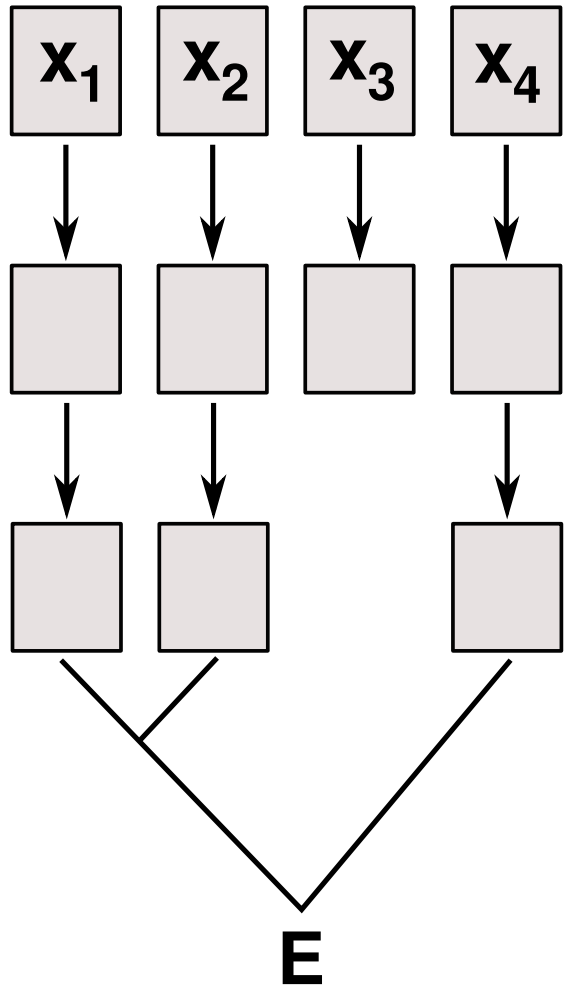
```
len :: MyString→ Int
len Empty        = 0
len (Cons c str) = 1+ len str
```

Gemeinsamkeiten:

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

Muster der primitiven Rekursion

- Anwenden einer Funktion auf jedes Element der Liste
- möglicherweise Filtern bestimmter Elemente
- Kombination der Ergebnisse zu einem Gesamtergebnis E



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 : toL cs
```

- Warum nicht ...

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 : toL cs
```

- Warum nicht ...

```
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 Höherer Ordnung

Slogan

"Functions are first-class citizens."

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

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- Auswertung:
  toL "AB"

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

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

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

```
map :: (α→ β)→ [α]→ [β]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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]
```

- Definition wie oben

```
map f []     = []
map f (x:xs) = f x : map f xs
```

- 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"

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 : Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben

Beispiel filter : Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben
  - Dazu: filter (\n→ mod n p ≠0) ps
  - Namenlose (anonyme) Funktion

Beispiel filter : Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben
  - Dazu: filter (\n→ mod n p ≠0) ps
  - Namenlose (anonyme) Funktion

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

- Alle Primzahlen:

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

Beispiel filter : Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen heraussieben
  - Dazu: filter (\n→ mod n p ≠0) ps
  - Namenlose (anonyme) Funktion

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

- Alle Primzahlen:

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

- Die ersten n Primzahlen:

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

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

- Vertauschen der Argumente (vordefiniert):

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

η-Kontraktion

- 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

- 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:

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

- Warum?

η-Äquivalenz und *eta*-Kontraktion

η-Äquivalenz

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

- Warum? Extensionale Gleichheit von Funktionen
- In Haskell: η-Kontraktion
  - Bedingung: Ausdruck E :: $\alpha \to \beta$, Variable x :: α, E darf x nicht enthalten

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

- Syntaktischer Spezialfall Funktionsdefinition (punktfreie Notation)

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

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$a \to b \to c \equiv a \to (b \to c)$$

  - Inbesondere: $(a \to b) \to c \neq a \to (b \to c)$

- Funktionsanwendung ist linksassoziativ:

$$f\ a\ b \equiv (f\ a)\ b$$

  - Inbesondere: $f\ (a\ b) \neq (f\ a)\ b$

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

$$a \to b \to c \equiv a \to (b \to c)$$

  - Inbesondere: $(a \to b) \to c \neq a \to (b \to c)$

- Funktionsanwendung ist linksassoziativ:

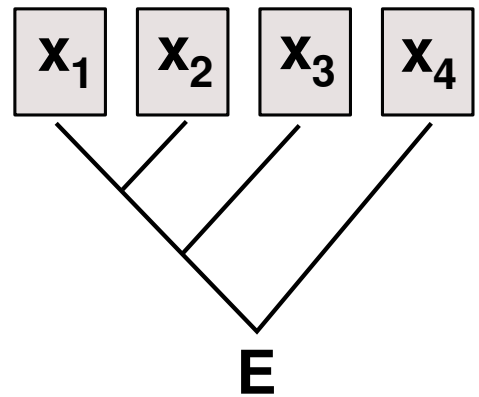
$$f\ a\ b \equiv (f\ a)\ b$$

  - Inbesondere: $f\ (a\ b) \neq (f\ a)\ b$

- Partielle Anwendung von Funktionen:
  - Für $f :: a \to b \to c$, $x :: a$ ist $f\ x :: b \to c$ (closure)
- Beispiele:
  - $\text{map toLower} :: \text{String} \to \text{String}$
  - $(3 ==) :: \text{Int} \to \text{Bool}$
  - $\text{concat} \circ \text{map}\ (\text{replicate}\ 2) :: \text{String} \to \text{String}$

Einfache Rekursion

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



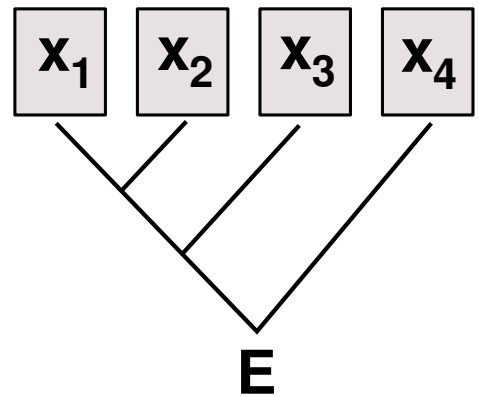
- Beispiel: kasse, inventur, sum, concat, length, $(+\!\!+)$, ...

- Auswertung:

sum [4,7,3] $\rightarrow$
concat [A, B, C] $\rightarrow$
length [4, 5, 6] $\rightarrow$

Einfache Rekursion

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



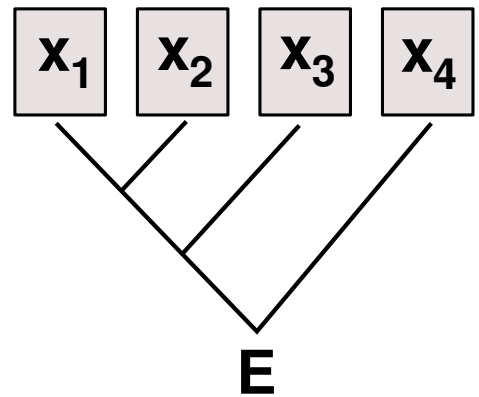
- Beispiel: kasse, inventur, sum, concat, length, $(+\!\!+)$, ...

- Auswertung:

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

Einfache Rekursion

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



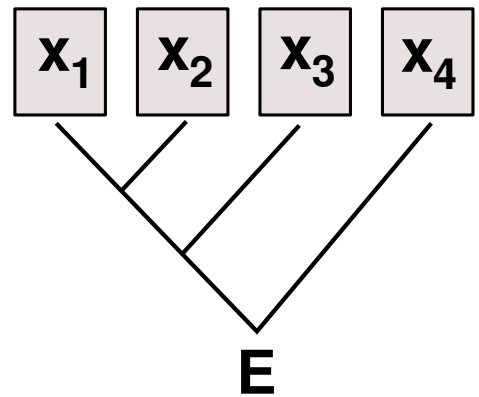
- Beispiel: kasse, inventur, sum, concat, length, $(+\!\!+)$, ...

- Auswertung:

sum [4,7,3]	$\rightarrow$	$4 + 7 + 3 + 0$
concat [A, B, C]	$\rightarrow$	$A +\!\!+ B +\!\!+ C +\!\!+ [\,]$
length [4, 5, 6]	$\rightarrow$	

Einfache Rekursion

- Einfache 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:

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

Einfache Rekursion

- Allgemeines Muster:

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

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

$$\mathsf{f}\ [\mathsf{x1},\ldots,\ \mathsf{xn}] = \mathsf{x1} \otimes \mathsf{x2} \otimes \ldots \otimes \mathsf{xn} \otimes \mathsf{A}$$

- Terminiert immer (wenn Liste endlich und $\otimes, A$ terminieren)
- Entspricht einfacher Iteration (while-Schleife)

Einfach Rekursion durch foldr

- Einfache 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.

- Suche nach einem Artikel alt:

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

- Suche nach einem Artikel neu:

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

Der Shoppe, revisited.

- Kasse alt:

```
kasse :: Einkaufswagen→ Int
kasse (Einkaufswagen []) = 0
kasse (Einkaufswagen (p: e)) = cent p+ kasse (Einkaufswagen e)
```

- Kasse neu:

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

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

Der Shoppe, revisited.

- Kassenbon formatieren neu:

```
kassenbon :: Einkaufswagen→ String
kassenbon ew@(Einkaufswagen as) =
  "Bob's␣Aulde␣Grocery␣Shoppe\n\n"++
  "Artikel␣␣␣␣␣␣␣␣␣␣␣␣␣␣Menge␣␣␣␣␣␣Preis\n"++
  "──────────────────────────────────────\n"++
  concatMap artikel as ++
  "══════════════════════════════════════\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"
```

Noch ein Beispiel: rev

- Listen umdrehen:

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

- Mit fold:

```
rev' = foldr snoc []
```

```
snoc :: α → [α] → [α]
snoc x xs = xs ++ [x]
```

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

Einfache Rekursion durch foldl

- foldr faltet von rechts:

$$\text{foldr} \otimes [x_1, ..., x_n]\ A = x_1 \otimes (x_2 \otimes (\ldots (x_n \otimes A)))$$

- Warum nicht andersherum?

$$\text{foldl} \otimes [x_1, ..., x_n]\ A = (((A \otimes x_1) \otimes x_2) \ldots) \otimes x_n$$

- 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
```

Beispiel: rev revisited

- Listenumkehr ist falten von links:

```
rev' xs = foldl (flip (:)) [] xs
```

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

foldr vs. foldl

- f = foldr $\otimes$ A entspricht

$$\begin{array}{lcl} \mathsf{f}\ [] & = & \mathsf{A} \\ \mathsf{f}\ (\mathsf{x}{:}\mathsf{xs}) & = & \mathsf{x} \otimes \mathsf{f}\ \mathsf{xs} \end{array}$$

  - Kann nicht-strikt in xs sein, z.B. and, or
  - Konsumiert nicht immer die ganze Liste
  - Auch für nichtendliche Listen anwendbar

- f = foldl $\otimes$ A entspricht

$$\begin{array}{lcl} \mathsf{f}\ \mathsf{xs} & = & \mathsf{g}\ \mathsf{A}\ \mathsf{xs} \\ \mathsf{g}\ \mathsf{a}\ [] & = & \mathsf{a} \\ \mathsf{g}\ \mathsf{a}\ (\mathsf{x}{:}\mathsf{xs}) & = & \mathsf{g}\ (\mathsf{a} \otimes \mathsf{x})\ \mathsf{xs} \end{array}$$

  - Endrekursiv (effizient) und strikt in xs
  - Konsumiert immer die ganze Liste
  - Divergiert immer für nichtendliche Listen

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 v. rechts
foldl     :: (β→ α→ β)→ β→ [α]→ β          — Falten v. 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])    — take und drop
any       :: (α → Bool) → [α] → Bool       — p gilt mind. einmal
all       :: (α → Bool) → [α] → Bool       — p gilt für alle
elem      :: (Eq α) ⇒ α → [α] → Bool       — Ist enthalten?
zipWith   :: (α → β → γ) → [α] → [β] → [γ]
                                           — verallgemeinertes zip
```

Funktionen Höherer Ordnung: Java

- Java: keine direkte Syntax für Funktionen höherer Ordnung
- Folgendes ist nicht möglich:

```
interface Collection {
  Object fold(Object f(Object a, Collection c), Object a); }
```

- Aber folgendes:

```
interface Foldable { Object f (Object a); }

interface Collection { Object fold(Foldable f, Object a); }
```

- Vergleiche Iterator aus Collections Framework (Java SE 6):

```
public interface Iterator<E> {
  boolean hasNext();
  E next(); }
```

- Seit Java SE 8 (März 2014): Anonyme Funktionen (Lambda-Ausdrücke)

Funktionen Höherer Ordnung: C

- Implizit vorhanden: Funktionen = Zeiger auf Funktionen

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

```
extern list map(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 *));
```

Funktionen Höherer Ordnung: C

Implementierung von map:

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

- Typsystem zu schwach:

```
  {
    *(int *)x= *(int *)x*2;
    return x;
  }
  void prt(void *x)
```

```
  printf("List 3: "); mapM_(prt, l); printf("\n");
```

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:
  - Einfache Rekursion entspricht foldr

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 6 vom 18.11.2014: Funktionen Höherer Ordnung II

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

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

Heute

- Die Geheimnisse von map und foldr gelüftet.
- map und foldr sind nicht nur für Listen.
- Funktionen höherer Ordnung als Entwurfsmuster

foldr ist kanonisch

- map und filter sind durch foldr darstellbar:

```
map :: (α→ β)→ [α]→ [β]
map f = foldr ((:). f) []
```

```
filter :: (α→ Bool)→ [α]→ [α]
filter p = foldr (λa as→ if p a then a:as
                                else as) []
```

foldr ist kanonisch

- map und filter sind durch foldr darstellbar:

```
map :: (α→ β)→ [α]→ [β]
map f = foldr ((:). f) []
```

```
filter :: (α→ Bool)→ [α]→ [α]
filter p = foldr (λa as→ if p a then a:as
                                else as) []
```

foldr ist die kanonische einfach rekursive Funktion.

- Alle einfach rekursiven Funktionen sind als Instanz von foldr darstellbar.

$$\text{foldr}\ \ (:)\ \ [\,] = \text{id}$$

map als strukturerhalten Abbildung

map ist die kanonische strukturerhaltende Abbildung.

- Struktur (Shape) eines Datentyps T α ist T ().
  - Für jeden Datentyp kann man kanonische Funktion shape :: T $\alpha \to$ T () angeben
  - Für Listen: [()] $\cong$ Nat.
- Für map gelten folgende Aussagen:

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

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

$$\text{shape. map f} = \text{shape}$$

Grenzen von foldr

- Andere rekursive Struktur über Listen
  - Quicksort: baumartige Rekursion

```
qsort :: Ord a⇒ [a]→ [a]
qsort [] = []
qsort xs = qsort (filter (< head xs) xs) ++
                 filter (head xs ==) xs ++
           qsort (filter (head xs <) xs)
```

- Rekursion nicht über Listenstruktur:
  - take: Rekursion über Int

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

  - Version mit foldr divergiert für nicht-endliche Listen

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

```
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 = True | False

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

- Maybe a:

fold für bekannte Datentypen

- Bool: Fallunterscheidung:

```
data Bool = True | False

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

- Maybe a: 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

- Instanzen von Map und Fold:

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

```
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)
```

  - Kein (offensichtliches) Filter

Funktionen mit fold und map

- 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:

$$\text{foldTree}\ \ \text{Node Mt} = \mathit{id}$$

$$\text{mapTree id} = \text{id}$$

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

$$\text{shape (mapTree f xs)} = \text{shape xs}$$

- Mit shape :: Tree $\alpha \rightarrow$ Tree ()

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

Alternativen: Breitensuche

- Alternative 1: Tiefensuche direkt rekursiv, mit Terminationsprädikat

```
dfts :: Eq α⇒ (Lab α→ Bool)→ Lab α→ [Path α]
```

- Alternative 2: Breitensuche für potentiell unendliche Liste aller Pfade

```
bfts :: Lab α→ [Path α]
bfts l = bfts0 [] [l] where
  bfts0 p [] = []
  bfts0 p (Node a cs:ns) =
    reverse (a:p) : (bfts0 p ns ++ bfts0 (a:p) cs)
```

- Gegensatz zur Tiefensuche: Liste kann konsumiert werden

Zusammenfassung map und fold

- map und fold sind kanonische Funktionen höherer Ordnung
- Für jeden Datentyp definierbar
- foldl nur für Listen (linearer Datentyp)
- fold kann bei zyklischen Argumenten nicht terminieren
  - Problem: Termination von fold nur lokal entscheidbar
  - Im Labyrinth braucht man den Kontext um zu entscheiden ob ein Knoten ein Blatt ist

Funktionen Höherer Ordnung als Entwurfsmethodik

- Kombination von Basisoperationen zu komplexen Operationen
- Kombinatoren als Muster zur Problemlösung:
  - Einfache Basisoperationen
  - Wenige Kombinationsoperationen
  - Alle anderen Operationen abgeleitet
- Kompositionalität:
  - Gesamtproblem läßt sich zerlegen
  - Gesamtlösung durch Zusammensetzen der Einzellösungen

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein Kombinator ist ein punktfrei definierte Funktion höherer Ordnung.

- Herkunft: Kombinatorlogik (Schönfinkel, 1924)

$$\begin{aligned} K\ x\ y &\rhd x \\ S\ x\ y\ z &\rhd x\ z\ (y\ z) \\ I\ x &\rhd x \end{aligned}$$

S, K, I sind Kombinatoren

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein Kombinator ist ein punktfrei definierte Funktion höherer Ordnung.

- Herkunft: Kombinatorlogik (Schönfinkel, 1924)

$$\begin{aligned} K\ x\ y &\rhd x \\ S\ x\ y\ z &\rhd x\ z\ (y\ z) \\ I\ x &\rhd x \end{aligned}$$

  S, K, I sind Kombinatoren

- Fun fact #1: kann alle berechenbaren Funktionen ausdrücken

Kombinatoren im engeren Sinne

Definition (Kombinator)

Ein Kombinator ist ein punktfrei definierte Funktion höherer Ordnung.

- Herkunft: Kombinatorlogik (Schönfinkel, 1924)

$$\begin{aligned} K\ x\ y &\rhd x \\ S\ x\ y\ z &\rhd x\ z\ (y\ z) \\ I\ x &\rhd x \end{aligned}$$

  S, K, I sind Kombinatoren

- Fun fact #1: kann alle berechenbaren Funktionen ausdrücken
- Fun fact #2: S und K sind genug: $I = S\ K\ K$

Beispiel: Parser

- Parser bilden Eingabe auf Parsierungen ab
  - Mehrere Parsierungen möglich
  - Backtracking möglich
- Kombinatoransatz:
  - Basisparser erkennen Terminalsymbole
  - Parserkombinatoren zur Konstruktion:
    - Sequenzierung (erst A, dann B)
    - Alternierung (entweder A oder B)
  - Abgeleitete Kombinatoren (z.B. Listen A^*, nicht-leere Listen A^+)

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse} = ?$$

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse}\ \alpha\ \beta = ?$$

- Parametrisiert über Eingabetyp (Token) α und Ergebnis β

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse}\ \alpha\ \beta = [\alpha] \rightarrow \beta$$

- Parametrisiert über Eingabetyp (Token) α und Ergebnis β
- Parser übersetzt Token in Ergebnis (abstrakte Syntax)

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse}\ \alpha\ \beta = [\alpha] \rightarrow (\beta, [\alpha])$$

- Parametrisiert über Eingabetyp (Token) α und Ergebnis β
- Parser übersetzt Token in Ergebnis (abstrakte Syntax)
- Muss Rest der Eingabe modellieren

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse}\ \alpha\ \beta = [\alpha] \rightarrow [(\beta, [\alpha])]$$

- Parametrisiert über Eingabetyp (Token) α und Ergebnis β
- Parser übersetzt Token in Ergebnis (abstrakte Syntax)
- Muss Rest der Eingabe modellieren
- Muss mehrdeutige Ergebnisse modellieren

Modellierung in Haskell

Welcher Typ für Parser?

$$\textbf{type}\ \text{Parse}\ \alpha\ \beta = [\alpha] \rightarrow [(\beta, [\alpha])]$$

- Parametrisiert über Eingabetyp (Token) α und Ergebnis β
- Parser übersetzt Token in Ergebnis (abstrakte Syntax)
- Muss Rest der Eingabe modellieren
- Muss mehrdeutige Ergebnisse modellieren
- Beispiel: "4*5+3" $\rightarrow$ [(4, "*4+3"),
  (4*5, "+3"),
  (4*5+3, "")]

Basisparser

- Erkennt nichts:

```
none :: Parse α β
none = const []
```

- Erkennt alles:

```
suceed :: β→ Parse α β
suceed b inp = [(b, inp)]
```

- Erkennt einzelne Token:

```
spot   :: (α→ Bool)→ Parse α α
spot p []       = []
spot p (x:xs) = if p x then [(x, xs)] else []

token :: Eq α⇒ α→ Parse α α
token t = spot (t ==)
```

  - Warum nicht none, suceed durch spot? Typ!

Basiskombinatoren: alt , $>\!*\!>$

- Alternierung:
  - Erste Alternative wird bevorzugt

```
infixl 3 `alt`
alt :: Parse α β→ Parse α β→ Parse α β
alt p1 p2 i = p1 i ++ p2 i
```

Basiskombinatoren: alt, >*>

- Alternierung:
  - Erste Alternative wird bevorzugt

```
infixl 3 `alt`
alt :: Parse α β→ Parse α β→ Parse α β
alt p1 p2 i = p1 i ++ p2 i
```

- Sequenzierung:
  - Rest des ersten Parsers als Eingabe für den zweiten

```
infixl 5 >*>
(>*>) :: Parse α β→ Parse α γ→ Parse α (β, γ)
(>*>) p1 p2 i =
  concatMap (λ(b, r)→
      map (λ(c, s)→ ((b, c), s)) (p2 r)) (p1 i)
```

Basiskombinatoren: use

- map für Parser (Rückgabe weiterverarbeiten):

```
infix 4 `use`, `use2`
use :: Parse α β→ (β→ γ)→ Parse α γ
use p f i = map (λ(o, r)→ (f o, r)) (p i)

use2 :: Parse α (β, γ)→ (β→ γ→ δ)→ Parse α δ
use2 p f = use p (uncurry f)
```

- Damit z.B. Sequenzierung rechts/links:

```
infixl 5 *>, >*
(*>) :: Parse α β→ Parse α γ→ Parse α γ
(>*) :: Parse α β→ Parse α γ→ Parse α β
p1 *> p2 = p1 >*> p2 `use` snd
p1 >* p2 = p1 >*> p2 `use` fst
```

Abgeleitete Kombinatoren

- Listen: $A^* ::= AA^* \mid \varepsilon$

```
list :: Parse α β→ Parse α [β]
list p = p >*> list p `use2` (:)
         `alt` suceed []
```

- Nicht-leere Listen: $A^+ ::= AA^*$

```
some :: Parse α β→ Parse α [β]
some p = p >*> list p `use2` (:)
```

- NB. Präzedenzen: $>*>$ (5) vor use (4) vor alt (3)

Verkapselung

- Hauptfunktion:
  - Eingabe muß vollständig parsiert werden
  - Auf Mehrdeutigkeit prüfen

```
parse :: Parse α β→ [α]→ Either String β
parse p i =
  case filter (null. snd) $ p i of
    []       → Left "Input␣does␣not␣parse"
    [(e, _)] → Right e
    _        → Left "Input␣is␣ambiguous"
```

- Schnittstelle:
  - Nach außen nur Typ Parse sichtbar, plus Operationen darauf

Grammatik für Arithmetische Ausdrücke

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Term} + \mathit{Term} \mid \mathit{Term} \\
\mathit{Term} & ::= & \mathit{Factor} * \mathit{Factor} \mid \mathit{Factor} \\
\mathit{Factor} & ::= & \mathit{Variable} \mid (\mathit{Expr}) \\
\mathit{Variable} & ::= & \mathit{Char}^{+} \\
\mathit{Char} & ::= & \texttt{a} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \cdots \mid \texttt{Z}
\end{array}
$$

Abstrakte Syntax für Arithmetische Ausdrücke

- Zur Grammatik abstrakte Syntax

```
data Expr  = Plus  Expr Expr
           | Times Expr Expr
           | Var   String
```

- Hier Unterscheidung Term, Factor, Number unnötig.

Parsierung Arithmetischer Ausdrücke

- Token: Char
- Parsierung von Factor

```
pFactor :: Parse Char Expr
pFactor = some (spot isAlpha) `use` Var
        `alt` token '(' *> pExpr >* token ')'
```

- Parsierung von Term

```
pTerm :: Parse Char Expr
pTerm =
  pFactor >* token '*' >*> pFactor `use2` Times
  `alt` pFactor
```

- Parsierung von Expr

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pTerm `use2` Plus
      `alt` pTerm
```

Die Hauptfunktion

- Lexing: Leerzeichen aus der Eingabe entfernen

```
parseExpr :: String→ Expr
parseExpr i =
  case parse pExpr (filter (not.isSpace) i) of
    Right e →  e
    Left err → error err
```

Ein kleiner Fehler

- Mangel: a+b+c führt zu Syntaxfehler — Fehler in der Grammatik
- Behebung: Änderung der Grammatik

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Term} + \mathit{Expr} \mid \mathit{Term} \\
\mathit{Term} & ::= & \mathit{Factor} * \mathit{Term} \mid \mathit{Factor} \\
\mathit{Factor} & ::= & \mathit{Variable} \mid (\mathit{Expr}) \\
\mathit{Variable} & ::= & \mathit{Char}^{+} \\
\mathit{Char} & ::= & \mathtt{a} \mid \cdots \mid \mathtt{z} \mid \mathtt{A} \mid \cdots \mid \mathtt{Z}
\end{array}
$$

- Abstrakte Syntax bleibt

Änderung des Parsers

- Entsprechende Änderung des Parsers in pTerm

```
pTerm :: Parse Char Expr
pTerm =
  pFactor >* token '*' >*> pTerm `use2` Times
  `alt` pFactor
```

- ... und in pExpr:

```
pExpr :: Parse Char Expr
pExpr = pTerm >* token '+' >*> pExpr `use2` Plus
      `alt` pTerm
```

- pFactor und Hauptfunktion bleiben.

Erweiterung zu einem Taschenrechner

- Zahlen:

$$\begin{aligned} \mathit{Factor} &::= \mathit{Variable} \mid \mathit{Number} \mid \ldots \\ \mathit{Number} &::= \mathit{Digit}^{+} \\ \mathit{Digit} &::= 0 \mid \cdots \mid 9 \end{aligned}$$

- Eine einfache Eingabesprache:

$$\mathit{Input} ::= \texttt{!}\ \mathit{Variable} = \mathit{Expr} \mid \texttt{\$}\ \mathit{Expr}$$

- Eine Auswertungsfunktion:

```
type State = [(String, Integer)]
```

```
eval :: State -> Expr -> Integer
```

```
run :: State -> String -> (State, String)
```

Zusammenfassung Parserkombinatoren

- Systematische Konstruktion des Parsers aus der Grammatik.
- Kompositional:
  - Lokale Änderung der Grammatik führt zu lokaler Änderung im Parser
  - Vgl. Parsergeneratoren (yacc/bison, antlr, happy)
- Struktur von Parse zur Benutzung irrelevant
  - Vorsicht bei Mehrdeutigkeiten in der Grammatik (Performance-Falle)
  - Einfache Implementierung (wie oben) skaliert nicht
  - Effiziente Implementation mit gleicher Schnittstelle auch für große Eingaben geeignet.

Zusammenfassung

- map und fold sind kanonische Funktionen höherer Ordnung
- ... und für alle Datentypen definierbar
- Kombinatoren: Funktionen höherer Ordnung als Entwurfsmethodik
  - Einfache Basisoperationen
  - Wenige aber mächtige Kombinationsoperationen
  - Reiche Bibliothek an abgeleiteten Operationen
- Nächste Woche: wie prüft man den Typ von

```
(>*>) p1 p2 i =
  concatMap (λ(b, r)→
      map (λ(c, s)→ ((b, c), s)) (p2 r)) (p1 i)
```

→ Typinferenz!

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 7 vom 25.11.2014: Typinferenz

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

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

Inhalt der Vorlesung

- Wozu Typen?
- Was ist ein Typsystem?
- Herleitung von Typen und Prüfung der Typkorrektheit (Typinferenz)

Wozu Typen?

- Frühzeitiges Aufdecken “offensichtlicher” Fehler
- “Once it type checks, it usually works”
- Hilfestellung bei Änderungen von Programmen
- Strukturierung großer Systeme auf Modul- bzw. Klassenebene
- Effizienz

Was ist ein Typsystem?

Ein Typsystem ist eine handhabbare syntaktische Methode, um die Abwesenheit bestimmter Programmverhalten zu beweisen, indem Ausdrücke nach der Art der Werte, die sie berechnen, klassifiziert werden.
(Benjamin C. Pierce, Types and Programming Languages, 2002)

Slogan:

Well-typed programs can't go wrong
(Robin Milner)

Typen in Haskell (The Story So Far)

- Primitive Basisdatentypen: Bool, Double
- Funktionstypen $\texttt{Int} \to \texttt{Int} \to \texttt{Int}$, $[\texttt{Double}] \to \texttt{Double}$
- Typkonstruktoren: [], (. . .), Foo
- Typvariablen
  $$\texttt{fst} :: (\alpha, \beta) \to \alpha$$
  $$\texttt{length} :: [\alpha] \to \texttt{Int}$$
  $$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
- Typklassen :
  $$\texttt{elem} :: \texttt{Eq a} \Rightarrow \texttt{a} \to [\texttt{a}] \to \texttt{Bool}$$
  $$\texttt{max} :: \texttt{Ord a} \Rightarrow \texttt{a} \to \texttt{a} \to \texttt{a}$$

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

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

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

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

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

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

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

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

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

```
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 Ausdruck der Form

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

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

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

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

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

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

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

Typinferenz: Das Problem

- Gegeben Ausdruck der Form

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

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

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

$$\mathsf{f} \;::\; \mathsf{Int}\to\; [\alpha]\to\; \mathsf{Int}$$

Typinferenz

- Mathematisch exaktes System zur Typbestimmung
  - Antwort auf die Frage: welchen Typ hat ein Ausdruck?
- Formalismus: Typableitungen der Form

$$\Gamma \vdash e :: \tau$$

  - Γ — Typumgebung (Zuordnung Symbole zu Typen)
  - e — Term
  - τ — Typ
- Beschränkung auf eine Kernsprache

Kernsprache: Ausdrücke

- Beschränkung auf eine kompakte Kernsprache:

$$
\begin{array}{rcl}
e & ::= & \mathit{var} \\
& \mid & \lambda\ \mathit{var}.\ e_1 \\
& \mid & e_1\ e_2 \\
& \mid & \textbf{let}\ \mathit{var} = e_1\ \textbf{in}\ e_2 \\
& \mid & \textbf{case}\ e_1\ \textbf{of} \\
& & \quad C_1\ \mathit{var}_1 \cdots \mathit{var}_n \to e_1 \\
& & \quad \ldots
\end{array}
$$

- Rest von Haskell hierin ausdrückbar:
  - **if** ... **then** ... **else**, Guards, Mehrfachapplikation, Funktionsdefinition, **where**

Kernsprache: Typen

- Typen sind gegeben durch:

$$\begin{array}{rcl} T & ::= & \mathit{tvar} \\ & | & C\ T_1\ \ldots\ T_n \end{array}$$

- *tvar* sind Typvariablen $\alpha, \beta, \ldots$
- C ist Typkonstruktur der Arität n. Beispiele:
  - Basistypen $n = 0$ (Int, Bool)
  - Listen $[t_1]$ mit $n = 1$
  - Funktionstypen $T_1 \to T_2$ mit $n = 2$

Typinferenzregeln

$$\frac{x :: t \in \Gamma}{\Gamma \vdash x :: t}\ \mathit{Var} \qquad \frac{\Gamma, x :: s \vdash e :: t}{\Gamma \vdash \lambda x.\ e :: s \to t}\ \mathit{Abs}$$

$$\frac{\Gamma \vdash e :: s \to t \quad \Gamma \vdash e' :: s}{\Gamma \vdash e\ e' :: t}\ \mathit{App}$$

$$\frac{\Gamma, x :: t_1 \vdash e_1 :: t_1 \quad \Gamma, x :: t_1 \vdash e_2 :: t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: t_2}\ \mathit{LetRec}$$

$$\frac{\Gamma \vdash f :: s \quad \Gamma, y_{i,j} :: t_{i,j} \vdash p_i :: s \quad \Gamma, y_{i,j} :: t_{i,j} \vdash e_i :: t}{\Gamma \vdash \textbf{case}\ f\ \textbf{of}\ p_i\ \to\ e_i :: t}\ \mathit{Cases}$$

Beispielableitung formal

- Haskell-Program: `f m xs = m + len xs`
- In unserer Sprache: $\lambda m\, xs.\ m + len\ xs$
- Initialer Kontext $C_0 \stackrel{def}{=} \{+ :: Int \to Int \to Int, len :: [\alpha] \to Int\}$
- Typableitungsproblem: $C_0 \vdash \lambda m\, xs.\ m + len\ xs :: ?$
- Ableitung als Baum:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{C_1 \vdash + :: Int \to Int \to Int}\,Var \quad \dfrac{}{C_1 \vdash m :: Int}\,Var}{C_1 \vdash m + :: Int \to Int}\,App \quad \dfrac{\dfrac{}{C_1 \vdash len :: [\alpha] \to Int}\,Var \quad \dfrac{}{C_1 \vdash xs :: [\alpha]}\,Var}{C_1 \vdash len\ xs :: Int}\,App}{C_1 \vdash m + len\ xs :: Int}\,App}{\dfrac{C_1 \stackrel{def}{=} C_0, m :: Int, xs :: [\alpha] \vdash m + len\ xs :: Int}{C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int}\,Abs}{C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int}\,Abs$$

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

1. $C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int$ $Abs[2]$

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

1. $C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int$ $Abs[2]$
2. $C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int$ $Abs[3]$

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

1. $C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int$ — $Abs[2]$
2. $C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int$ — $Abs[3]$
3. $C_1 \stackrel{def}{=} C_0, m :: Int, xs :: [\alpha] \vdash m + len\ xs :: Int$ — $App[4, 7]$

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

1.	$C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int$	$Abs[2]$
2.	$C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int$	$Abs[3]$
3.	$C_1 \stackrel{def}{=} C_0, m :: Int, xs :: [\alpha] \vdash m + len\ xs :: Int$	$App[4,7]$
4.	$C_1 \vdash m + :: Int \to Int$	$App[5,6]$

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

1.	$C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int$	$Abs[2]$
2.	$C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int$	$Abs[3]$
3.	$C_1 \stackrel{def}{=} C_0, m :: Int, xs :: [\alpha] \vdash m + len\ xs :: Int$	$App[4,7]$
4.	$C_1 \vdash m + :: Int \to Int$	$App[5,6]$
5.	$C_1 \vdash + :: Int \to Int \to Int$	Var

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

1.	$C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int$	$Abs[2]$
2.	$C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int$	$Abs[3]$
3.	$C_1 \stackrel{def}{=} C_0, m :: Int, xs :: [\alpha] \vdash m + len\ xs :: Int$	$App[4,7]$
4.	$C_1 \vdash m+ :: Int \to Int$	$App[5,6]$
5.	$C_1 \vdash + :: Int \to Int \to Int$	Var
6.	$C_1 \vdash m :: Int$	Var

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

1.	$C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int$	$Abs[2]$
2.	$C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int$	$Abs[3]$
3.	$C_1 \stackrel{def}{=} C_0, m :: Int, xs :: [\alpha] \vdash m + len\ xs :: Int$	$App[4,7]$
4.	$C_1 \vdash m + :: Int \to Int$	$App[5,6]$
5.	$C_1 \vdash + :: Int \to Int \to Int$	Var
6.	$C_1 \vdash m :: Int$	Var
7.	$C_1 \vdash len\ xs :: Int$	$App[8,9]$

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

1.	$C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int$	$Abs[2]$
2.	$C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int$	$Abs[3]$
3.	$C_1 \stackrel{def}{=} C_0, m :: Int, xs :: [\alpha] \vdash m + len\ xs :: Int$	$App[4,7]$
4.	$C_1 \vdash m + :: Int \to Int$	$App[5,6]$
5.	$C_1 \vdash + :: Int \to Int \to Int$	Var
6.	$C_1 \vdash m :: Int$	Var
7.	$C_1 \vdash len\ xs :: Int$	$App[8,9]$
8.	$C_1 \vdash len :: [\alpha] \to Int$	Var

Ableitung formal

Bessere Notation:

- linear
- Von der Konklusion ausgehend (der Wurzel des Baumes)
- Letzte Spalte enthält angewandte Regel und Voraussetzungen

1.	$C_0 \vdash \lambda m\, xs.\ m + len\ xs :: Int \to [\alpha] \to Int$	$Abs[2]$
2.	$C_0, m :: Int \vdash \lambda xs.\ m + len\ xs :: [\alpha] \to Int$	$Abs[3]$
3.	$C_1 \stackrel{def}{=} C_0, m :: Int, xs :: [\alpha] \vdash m + len\ xs :: Int$	$App[4,7]$
4.	$C_1 \vdash m + :: Int \to Int$	$App[5,6]$
5.	$C_1 \vdash + :: Int \to Int \to Int$	Var
6.	$C_1 \vdash m :: Int$	Var
7.	$C_1 \vdash len\ xs :: Int$	$App[8,9]$
8.	$C_1 \vdash len :: [\alpha] \to Int$	Var
9.	$C_1 \vdash xs :: [\alpha]$	Var

Problem: Typvariablen

- Sei $D \stackrel{def}{=} \{id : \alpha \to \alpha\}$
- Typableitung $D \vdash id\ id :: \alpha \to \alpha$ benötigt zwei unterschiedliche Instantiierung von *id*
- Andererseits: in $C \vdash \lambda x.\, x\ x :: ?$ darf x nicht unterschiedlich instantiiert werden.
- Deshalb: Typschemata

$$S \ ::= \ \forall tvar.\ S \mid T$$

- Zwei zusätzliche Regeln zur Instantiierung und Generalisierung

Typinferenzregeln (vollständig)

$$\frac{x :: t \in \Gamma}{\Gamma \vdash x :: t}\ \mathit{Var} \qquad \frac{\Gamma, x :: s \vdash e :: t}{\Gamma \vdash \lambda x.\ e :: s \to t}\ \mathit{Abs}$$

$$\frac{\Gamma \vdash e :: s \to t \quad \Gamma \vdash e' :: s}{\Gamma \vdash e\ e' :: t}\ \mathit{App}$$

$$\frac{\Gamma, x :: t_1 \vdash e_1 :: t_1 \quad \Gamma, x :: t_1 \vdash e_2 :: t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: t_2}\ \mathit{LetRec}$$

$$\frac{\Gamma \vdash f :: s \quad \Gamma, y_{i,j} :: t_{i,j} \vdash p_i :: s \quad \Gamma, y_{i,j} :: t_{i,j} \vdash e_i :: t}{\Gamma \vdash \textbf{case}\ f\ \textbf{of}\ p_i\ \to\ e_i :: t}\ \mathit{Cases}$$

Typinferenzregeln (vollständig)

$$\frac{x :: t \in \Gamma}{\Gamma \vdash x :: t}\ \mathit{Var} \qquad \frac{\Gamma, x :: s \vdash e :: t}{\Gamma \vdash \lambda x.\ e :: s \to t}\ \mathit{Abs}$$

$$\frac{\Gamma \vdash e :: s \to t \quad \Gamma \vdash e' :: s}{\Gamma \vdash e\ e' :: t}\ \mathit{App}$$

$$\frac{\Gamma, x :: t_1 \vdash e_1 :: t_1 \quad \Gamma, x :: t_1 \vdash e_2 :: t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: t_2}\ \mathit{LetRec}$$

$$\frac{\Gamma \vdash f :: s \quad \Gamma, y_{i,j} :: t_{i,j} \vdash p_i :: s \quad \Gamma, y_{i,j} :: t_{i,j} \vdash e_i :: t}{\Gamma \vdash \textbf{case}\ f\ \textbf{of}\ p_i\ \to\ e_i :: t}\ \mathit{Cases}$$

$$\frac{\Gamma \vdash e :: \forall \alpha.\, t}{\Gamma \vdash e :: t \left[{}^{s}_{\alpha}\right]}\ \mathit{Spec} \qquad \frac{\Gamma \vdash e :: t \quad \alpha \text{ nicht frei in } \Gamma}{\Gamma \vdash e :: \forall \alpha.\, t}\ \mathit{Gen}$$

Beispiel: *id* revisited

Damit ist jetzt $D \stackrel{def}{=} id : \forall\alpha.\, \alpha \to \alpha$

1.	$D \vdash id\ id :: \forall\beta.\beta \to \beta$	*Gen*
2.	$D \vdash id\ id :: \beta \to \beta$	*App*[3, 5]
3.	$D \vdash id :: (\beta \to \beta) \to (\beta \to \beta)$	*Spec*[4]
4.	$D \vdash id :: \forall\alpha.\alpha \to \alpha$	*Var*
5.	$D \vdash id :: \beta \to \beta$	*Spec*[6]
6.	$D \vdash id :: \forall\alpha.\alpha \to \alpha$	*Var*

Typinferenz: Typen ableiten

- Das Typinferenzproblem:
  - Gegeben Γ und e
  - Gesucht wird: τ und σ so dass $\sigma(\Gamma) \vdash e :: \tau$
- Berechung von τ und σ durch Algorithmus W (Damas-Milner)
- Informell:
  - Typbestimmung beginnt an den Blättern des Ableitungsbaumes (Regeln ohne Voraussetzungen)
  - Konstanten zuerst (Typ fest), dann Variablen (Typ offen)
  - Instantiierung für Typschemata und Typ für Variablen unbestimmt lassen, und konsistent anpassen
  - Typ mit Regeln wie Abs, App und Cases nach oben propagieren

Beispiel: Bestimmung des Typs von map

Die Funktion map:

```
map f  []  =     []
map f  (x:xs) = f x  :  map f  xs
```

In unserer Kernsprache:

$$\textbf{let } map = \lambda f\ ys.\, \textbf{case } ys \textit{ of } [] \to []; (x : xs) \to map\ f\ y \textbf{ in } map$$

Ableitung mit $C_0 \stackrel{def}{=} \{(:) : \forall\alpha.\alpha \to [\alpha] \to \alpha\}$

Beispiel: Bestimmung eines großen Typen

Die Sequenzierungsoperation für Parser:

```
(>*>) p1 p2 i =
  concatMap (λ(b, r)→
     map (λ(c, s)→ ((b, c), s)) (p2 r)) (p1 i)
```

In unserer Kernsprache:

$\lambda p1\, p2\, i.\, concatMap(\lambda br.\, map(\lambda cs.\, ((fst\ br, fst\ cs), snd\ cs))(p2\ (snd\ br)))(p1$

Eigenschaften von W

- Entscheidbarkeit und Korrektheit: Für Kontext Γ und Term e terminiert Algorithmus W immer, und liefert $W(\Gamma, e) = (\sigma, \tau) = (\sigma, \tau)$ so dass $\sigma(\Gamma) \vdash e :: \tau$
- Vollständigkeit: $W(\Gamma, e)$ berechnet den allgemeinsten Typen (principal type) von e (wenn es ihn gibt)
- Aufwand von W:
  - Theoretisch: exponentiell (*DEXPTIME*)
  - Praktisch: in relevanten Fällen annähernd linear

Typen in anderen Programmiersprachen

- Statische Typisierung (Typableitung während Übersetzung)
  - Haskell, ML
  - Java, C++, C (optional)
- Dynamische Typisierung (Typüberprüfung zur Laufzeit)
  - PHP, Python, Ruby (*duck typing*)
- Ungetypt
  - Lisp, LaTeX, Tcl, Shell

Zusammenfassung

- Haskell implementiert Typüberprüfung durch Typinferenz (nach Damas-Milner)
- Kernelemente der Typinferenz:
  - Bindung von Typvariablen in Typschema ($\forall\alpha.\tau$)
  - Berechnung des Typen von den Blättern des Ableitungsbaumes her
  - Typinferenz berechnet allgemeinsten Typ
- Typinferenz hat praktisch linearen, theoretisch exponentiellen Aufwand
- Nächste Woche: Module und abstrakte Datentypen in Haskell

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 8 vom 02.12.2014: Abstrakte Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Organisatorisches

- Raumänderung nächste Woche:

Vorlesung	Di, 09.12.	12-14	NW2 A0242
Tutorium	Mi, 10.12.	08-10	SpT C4180
Tutorium	Mi, 10.12.	10-12	entfällt!
Tutorium	Mi, 10.12.	12-14	GW1 A0160
Tutorium	Mi, 10.12.	14-16	SFG 1020

  Die Tutorien am Donnerstag finden wie gewohnt statt.

- Grund ist eine internationale Tagung...

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
  - Abstrakte Datentypen
  - Signaturen und Eigenschaften
  - Spezifikation und Beweis
  - Aktionen und Zustände
- Teil III: Funktionale Programmierung im richtigen Leben

Inhalt

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

Refakturierung im Einkaufsparadies

Nov 11, 14 14:28 **Shoppe3.hs** Page 1/3

```
module Shoppe3 where

import Data.Maybe

-- Modellierung der Artikel.

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

apreis :: Apfel -> Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50

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

kpreis :: Kaese -> Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

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

data Artikel =
   Apfel Apfel   | Eier
| Kaese Kaese |  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

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


-- 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 Show

-- Lagerhaltung:
data Lager = Lager [Posten]
```

Nov 11, 14 14:28 **Shoppe3.hs** Page 2/3

```
             deriving Show

leeresLager :: Lager
leeresLager = Lager []

suche :: Artikel-> Lager-> Maybe Menge
suche a (Lager l) =
   listToMaybe (map (\(Posten _ m)-> m)
                (filter (\(Posten la _) -> la == a) l))


einlagern :: Artikel-> Menge-> Lager-> Lager
einlagern a m (Lager l) =
  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 l
_        -> Lager (hinein a m l)

data Einkaufswagen = Einkaufswagen [Posten]
                     deriving Show

leererWagen :: Einkaufswagen
leererWagen = Einkaufswagen []

einkauf :: Artikel-> Menge-> Einkaufswagen-> Einkaufswagen
einkauf a m (Einkaufswagen e)
  | isJust (preis a m) = Einkaufswagen (Posten a m: e)
|  otherwise =          Einkaufswagen e

kasse' :: Einkaufswagen-> Int
kasse' (Einkaufswagen ps) = foldr (\p r-> cent p+ r) 0 ps

kasse :: Einkaufswagen-> Int
kasse (Einkaufswagen ps) = sum (map cent ps)


kassenbon :: Einkaufswagen-> String
kassenbon ew@(Einkaufswagen as) =
  "Bob's Aulde Grocery Shoppe\n\n"++
"   Artikel          Menge     Preis\n"++
"  ------------------------------------\n"++
c oncatMap artikel as ++
"  ====================================\n"++
"   Summe:"++ formatR 31 (showEuro (kasse ew))

artikel :: Posten-> String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
f  ormatR 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 ' ')
```

Refakturierung im Einkaufsparadies

Nov 11, 14 14:28 **Shoppe3.hs** Page 1/3

```
module Shoppe3 where

import Data.Maybe

-- Modellierung der Artikel.

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

apreis :: Apfel -> Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50

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

kpreis :: Kaese -> Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

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

data Artikel =
   Apfel Apfel  | Eier
| Kaese Kaese |  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

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


-- 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 Show

-- Lagerhaltung:
data Lager = Lager [Posten]
```

Artikel

Nov 11, 14 14:28 **Shoppe3.hs** Page 2/3

```
              deriving Show

leeresLager :: Lager
leeresLager = Lager []

suche :: Artikel-> Lager-> Maybe Menge
suche a (Lager l) =
   listToMaybe (map (\(Posten _ m)-> m)
                (filter (\(Posten la _) -> la == a) l))


einlagern :: Artikel-> Menge-> Lager-> Lager
einlagern a m (Lager l) =
  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 l
_        -> Lager (hinein a m l)

data Einkaufswagen = Einkaufswagen [Posten]
                     deriving Show

leererWagen :: Einkaufswagen
leererWagen = Einkaufswagen []

einkauf :: Artikel-> Menge-> Einkaufswagen-> Einkaufswagen
einkauf a m (Einkaufswagen e)
  | isJust (preis a m) = Einkaufswagen (Posten a m: e)
|  otherwise =          Einkaufswagen e

kasse' :: Einkaufswagen-> Int
kasse' (Einkaufswagen ps) = foldr (\p r-> cent p+ r) 0 ps

kasse :: Einkaufswagen-> Int
kasse (Einkaufswagen ps) = sum (map cent ps)


kassenbon :: Einkaufswagen-> String
kassenbon ew@(Einkaufswagen as) =
  "Bob's Aulde Grocery Shoppe\n\n"++
"   Artikel         Menge     Preis\n"++
"   ------------------------------------\n"++
c oncatMap artikel as ++
"   ====================================\n"++
"   Summe:"++ formatR 31 (showEuro (kasse ew))

artikel :: Posten-> String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
f  ormatR 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 ' ')
```

Refakturierung im Einkaufsparadies

Nov 11, 14 14:28 **Shoppe3.hs** Page 1/3

```
module Shoppe3 where

import Data.Maybe

-- Modellierung der Artikel.

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

apreis :: Apfel -> Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50

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

kpreis :: Kaese -> Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

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

data Artikel =
   Apfel Apfel   | Eier
| Kaese Kaese |   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

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


-- 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 Show

-- Lagerhaltung:
data Lager = Lager [Posten]
```

Artikel

Nov 11, 14 14:28 **Shoppe3.hs** Page 2/3

```
              deriving Show

leeresLager :: Lager
leeresLager = Lager []

suche :: Artikel-> Lager-> Maybe Menge
suche a (Lager l) =
   listToMaybe (map (\(Posten _ m)-> m)
                (filter (\(Posten la _) -> la == a) l))


einlagern :: Artikel-> Menge-> Lager-> Lager
einlagern a m (Lager l) =
  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 l
_        -> Lager (hinein a m l)

data Einkaufswagen = Einkaufswagen [Posten]
                     deriving Show

leererWagen :: Einkaufswagen
leererWagen = Einkaufswagen []

einkauf :: Artikel-> Menge-> Einkaufswagen-> Einkaufswagen
einkauf a m (Einkaufswagen e)
  | isJust (preis a m) = Einkaufswagen (Posten a m: e)
|  otherwise =          Einkaufswagen e

kasse' :: Einkaufswagen-> Int
kasse' (Einkaufswagen ps) = foldr (\p r-> cent p+ r) 0 ps

kasse :: Einkaufswagen-> Int
kasse (Einkaufswagen ps) = sum (map cent ps)


kassenbon :: Einkaufswagen-> String
kassenbon ew@(Einkaufswagen as) =
  "Bob's Aulde Grocery Shoppe\n\n"++
"   Artikel        Menge     Preis\n"++
"   ------------------------------------\n"++
c oncatMap artikel as ++
"   ====================================\n"++
"   Summe:"++ formatR 31 (showEuro (kasse ew))

artikel :: Posten-> String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
f  ormatR 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 ' ')
```

Lager

Refakturierung im Einkaufsparadies

Nov 11, 14 14:28 **Shoppe3.hs** Page 1/3

```haskell
module Shoppe3 where

import Data.Maybe

-- Modellierung der Artikel.

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

apreis :: Apfel -> Int
apreis Boskoop = 55
apreis CoxOrange = 60
apreis GrannySmith = 50

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

kpreis :: Kaese -> Double
kpreis Gouda = 1450
kpreis Appenzeller = 2270

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

data Artikel =
   Apfel Apfel   | Eier
| Kaese Kaese |  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

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


-- 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 Show

-- Lagerhaltung:
data Lager = Lager [Posten]
```

Artikel

Nov 11, 14 14:28 **Shoppe3.hs** Page 2/3

```haskell
             deriving Show

leeresLager :: Lager
leeresLager = Lager []

suche :: Artikel-> Lager-> Maybe Menge
suche a (Lager l) =
   listToMaybe (map (\(Posten _ m)-> m)
                (filter (\(Posten la _) -> la == a) l))


einlagern :: Artikel-> Menge-> Lager-> Lager
einlagern a m (Lager l) =
  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 l
_        -> Lager (hinein a m l)
```

Lager

```haskell
data Einkaufswagen = Einkaufswagen [Posten]
                     deriving Show

leererWagen :: Einkaufswagen
leererWagen = Einkaufswagen []

einkauf :: Artikel-> Menge-> Einkaufswagen-> Einkaufswagen
einkauf a m (Einkaufswagen e)
  | isJust (preis a m) = Einkaufswagen (Posten a m: e)
|  otherwise =          Einkaufswagen e

kasse' :: Einkaufswagen-> Int
kasse' (Einkaufswagen ps) = foldr (\p r-> cent p+ r) 0 ps

kasse :: Einkaufswagen-> Int
kasse (Einkaufswagen ps) = sum (map cent ps)


kassenbon :: Einkaufswagen-> String
kassenbon ew@(Einkaufswagen as) =
  "Bob's Aulde Grocery Shoppe\n\n"++
"   Artikel        Menge     Preis\n"++
"   ------------------------------------\n"++
c oncatMap artikel as ++
"   ====================================\n"++
"   Summe:"++ formatR 31 (showEuro (kasse ew))

artikel :: Posten-> String
artikel p@(Posten a m) =
   formatL 20 (show a) ++
f  ormatR 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 ' ')
```

Einkaufswagen

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 auf diesem.

- 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
- 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)

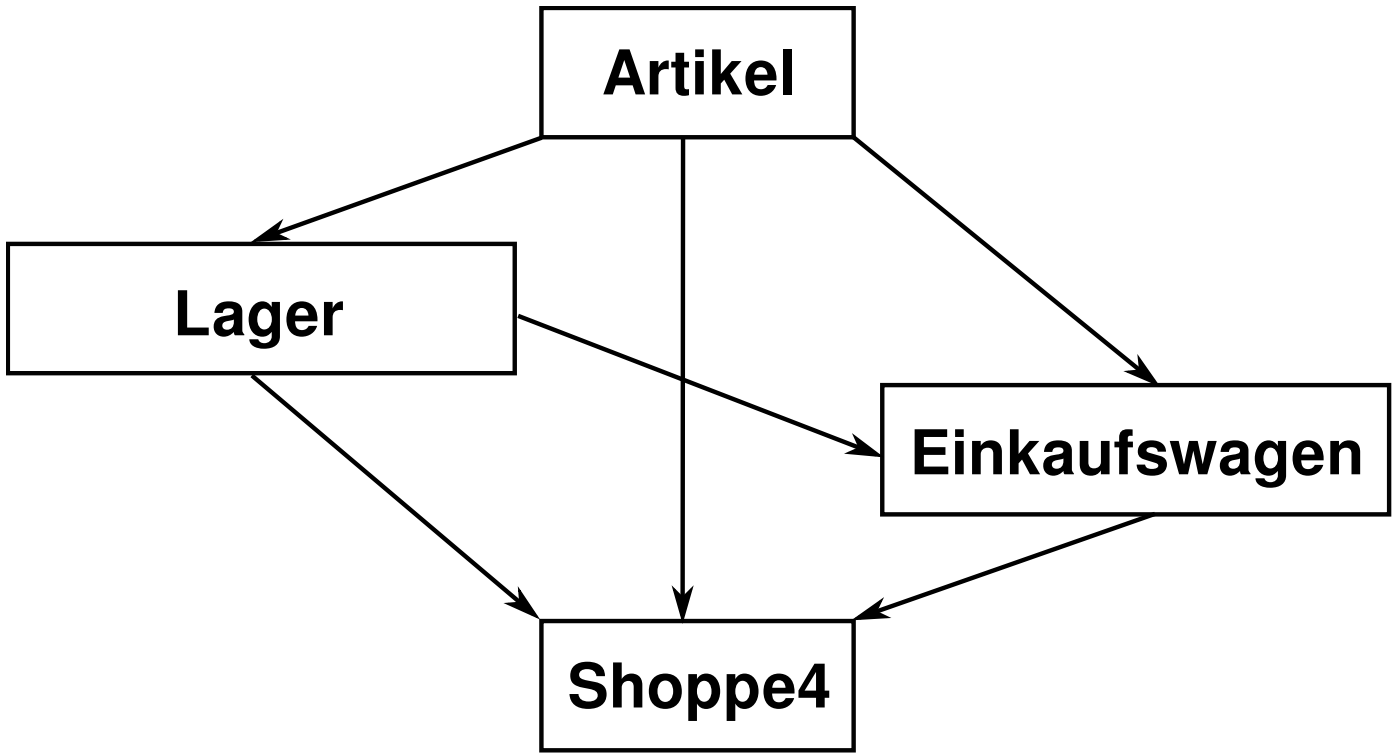
Module: Syntax

- 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: Modularchitektur



Refakturierung im Einkaufsparadies I: Artikel

- Es wird alles exportiert
- Reine Datenmodellierung

```
module Artikel where

import Data.Maybe

-- Modellierung der Artikel.

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

Refakturierung im Einkaufsparadies II: Lager

```
module Lager(
  Posten,
  artikel,
  menge,
  posten,
  cent,
  Lager,
  leeresLager,
  einlagern,
  suche,
  inventur
  ) where
```

- Implementiert ADTs Posten und Lager
- Garantierte Invarianten:
  - Posten hat immer die korrekte Artikel und Menge:

    ```
    posten :: Artikel→ Menge→ Maybe Posten
    ```

  - Lager enthält keine doppelten Artikel:

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

Refakturierung im Einkaufsparadies III: Einkaufswagen

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

- Implementiert ADT Einkaufswagen
- Garantierte Invariante:
  - Korrekte Artikel und Menge im Einkaufswagen

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

- Nutzt dazu Posten aus Modul Lager

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 A(x,y) where...
```

Import(e)	Bekannte Bezeichner
import A	x, y, A.x, A.y
import A()	*(nothing)*
import A(x)	x, A.x
import qualified A	A.x, A.y
import qualified A()	*(nothing)*
import qualified A(x)	A.x
import A hiding ()	x, y, A.x, A.y
import A hiding (x)	y, A.y
import qualified A hiding ()	A.x, A.y
import qualified A hiding (x)	A.y
import A as B	x, y, B.x, B.y
import A as B(x)	x, B.x
import qualified A as B	B.x, B.y

Quelle: Haskell98-Report, Sect. 5.3.4

Schnittstelle vs. Implementation

- Gleiche Schnittstelle kann unterschiedliche Implementationen haben

- Beispiel: (endliche) Abbildungen

Endliche Abbildungen

- Eine Sichtweise: Ersatz für Hashtables in imperativen Sprachen. Sehr nützlich!
- Abstrakter Datentyp für endliche Abbildungen:
  - Datentyp

    ```
    data Map α β
    ```

  - Leere Abbildung:

    ```
    empty :: Map α β
    ```

  - Abbildung auslesen:

    ```
    lookup :: Ord α⇒ α→ Map α β→ Maybe β
    ```

  - Abbildung ändern:

    ```
    insert :: Ord α⇒ α→ β→ Map α β→ Map α β
    ```

  - Abbildung löschen:

    ```
    delete :: Ord α⇒ α→ Map α β→ Map α β
    ```

Eine naheliegende Implementation

- Modellierung als Haskell-Funktion:

```
data Map α β = Map (α→ Maybe β)
```

- Damit einfaches lookup, insert , delete:

```
empty = Map (λx→ Nothing)
```

```
lookup a (Map s) = s a
```

```
insert a b (Map s) =
  Map (λx→ if x == a then Just b else s x)
```

```
delete a (Map s) =
  Map (λx→ if x == a then Nothing else s x)
```

- Instanzen von Eq, Show nicht möglich
- Speicherleck

Endliche Abbildungen: Anwendungsbeispiel

- Artikel im Lager:

```
data Lager = Lager (M.Map Artikel Posten)
```

- Artikel suchen:

```
suche a (Lager l) = fmap menge (M.lookup a l)
```

- Ins Lager hinzufügen:

```
einlagern :: Artikel→ Menge→ Lager→ Lager
einlagern a m (Lager l) =
  case posten a m of
      Nothing → Lager l
      Just p → Lager (M.insert a p l)
```

- Für Inventur fehlt Möglichkeit zur Iteration
- Daher: Map als Assoziativliste

Map als Assoziativliste

```
newtype Map α β = Map [(α, β)]
```

- Zusatzfunktionalität:
  - Iteration (fold)

```
fold :: Ord α⇒ ((α, β)→ γ→ γ)→ γ→ Map α β→ γ
fold f e (Map ms) = foldr f e ms
```

  - Instanzen von Eq und Show

```
instance (Eq α, Eq β)⇒ Eq (Map α β) where
  Map s1 == Map s2 =
    null (s1 \\ s2) && null (s1 \\ s2)
```

  - ... 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 Weight (Tree α) α (Tree α)
```

- Gewichtung (Parameter des Algorithmus):

```
type Weight = Int
```

```
weight :: Weight
```

- Hilfskonstruktor, setzt Größe (l, r balanciert)

```
node :: Tree α→ α→ Tree α→ Tree α
node l n r = Node h l n r where
                 h = 1+ size l+ size r
```

- Hilfskonstruktor, balanciert ggf. neu aus:

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

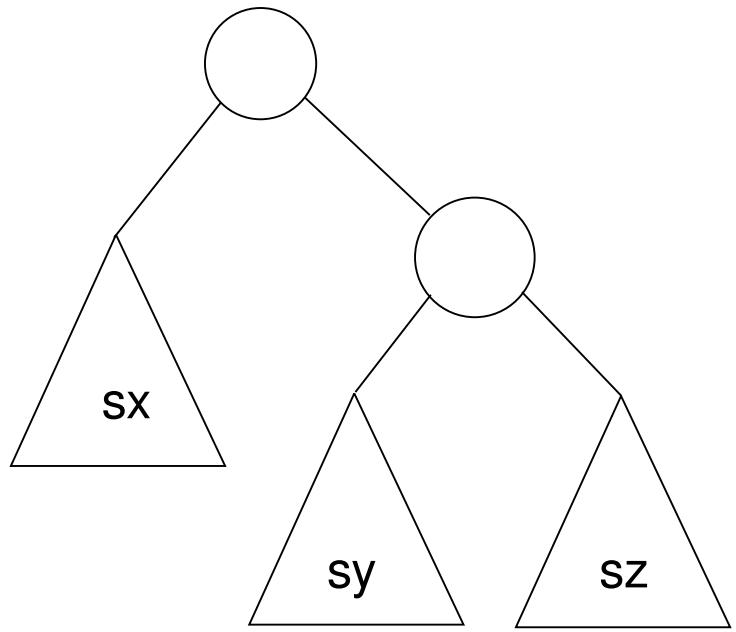
Balance sicherstellen

- Problem:

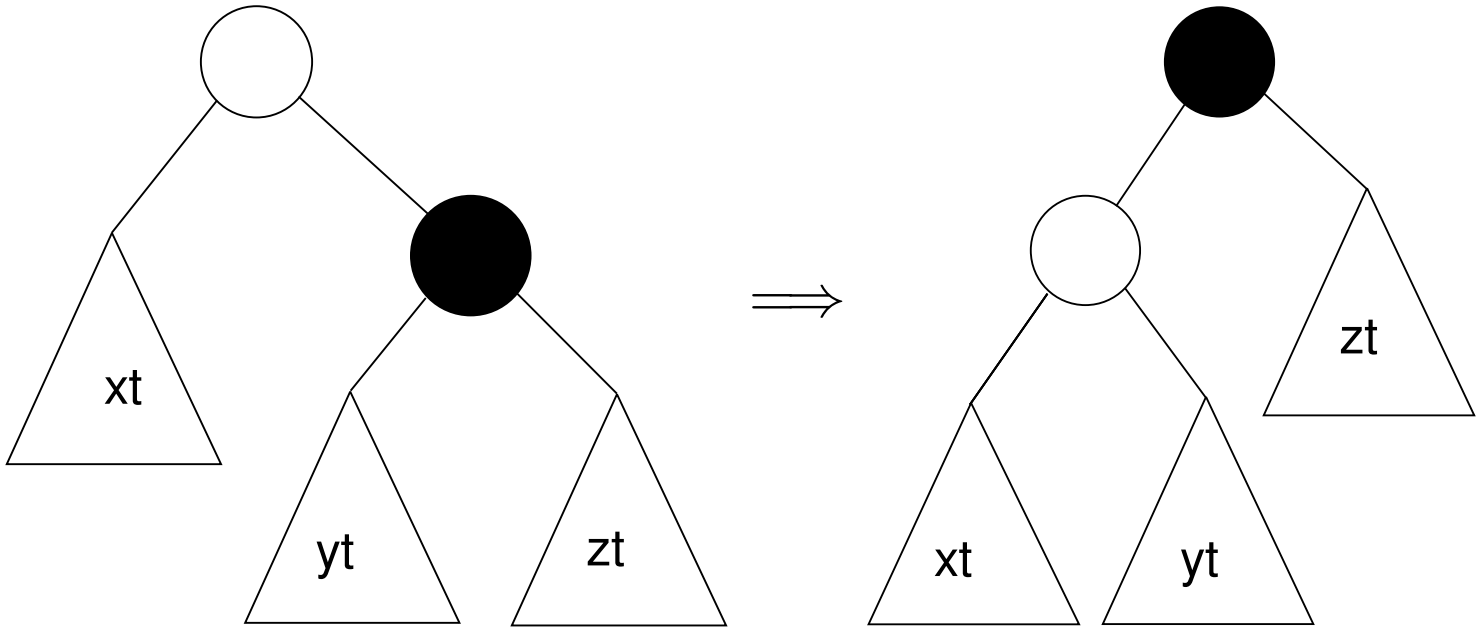
  Nach Löschen oder Einfügen zu großes Ungewicht

- Lösung:

  Rotieren der Unterbäume

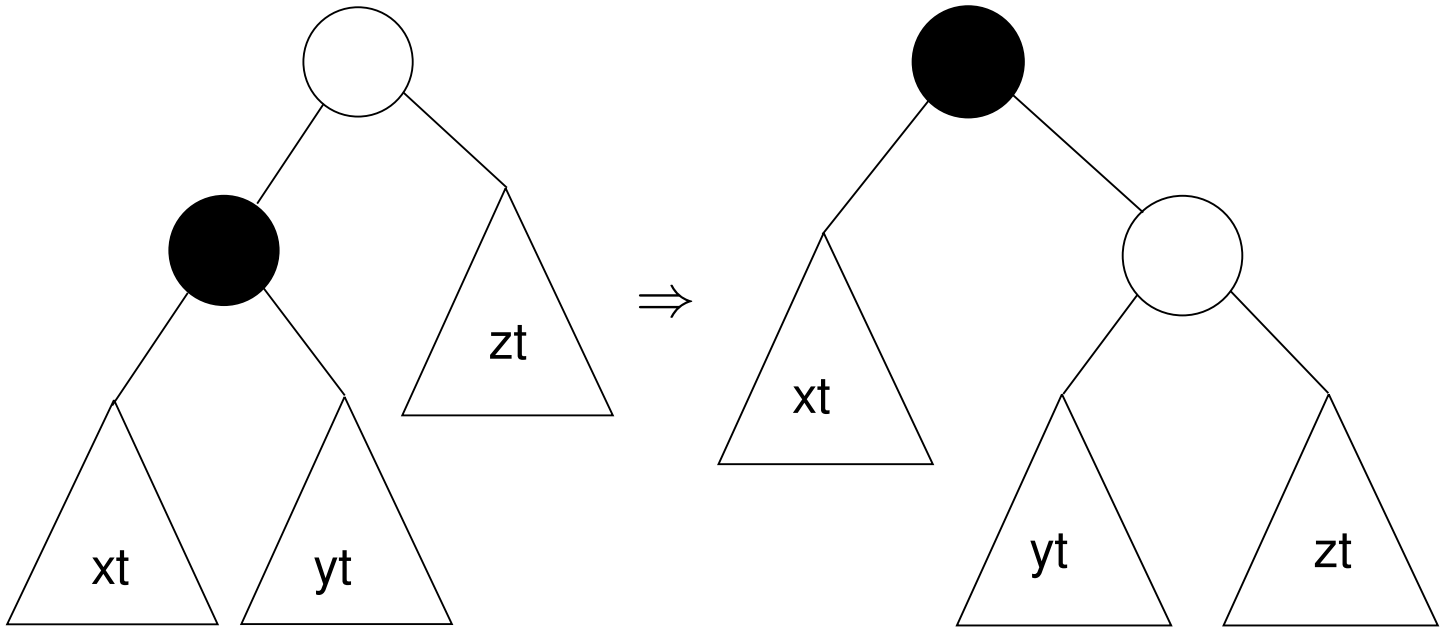


Linksrotation




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

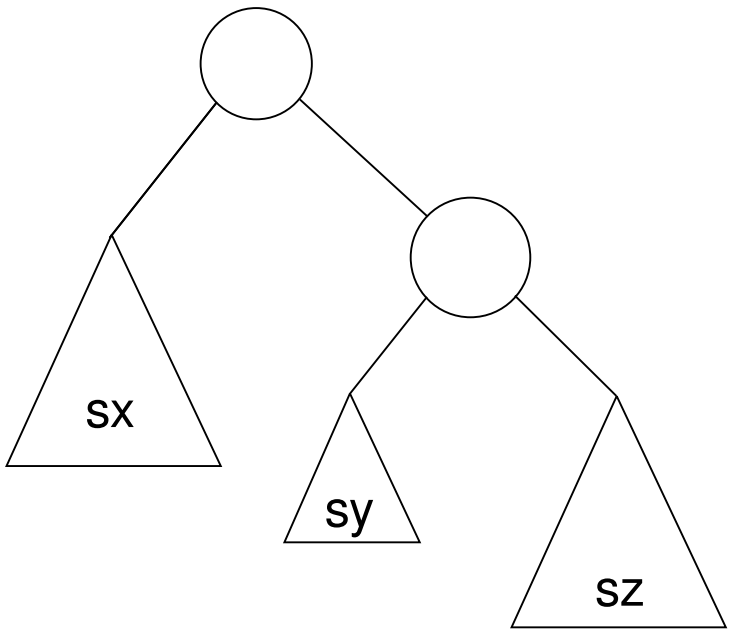
Rechtsrotation




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

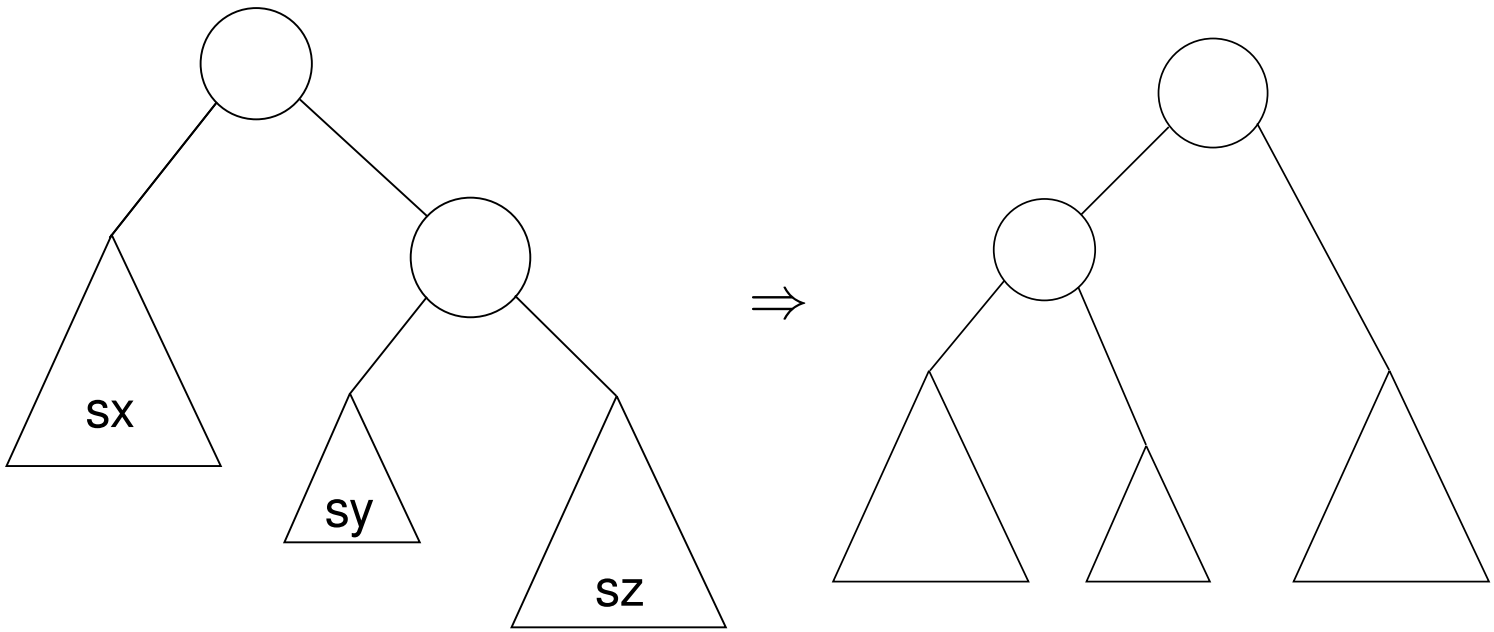
Balanciertheit sicherstellen

- Fall 1: Äußerer Unterbaum zu groß



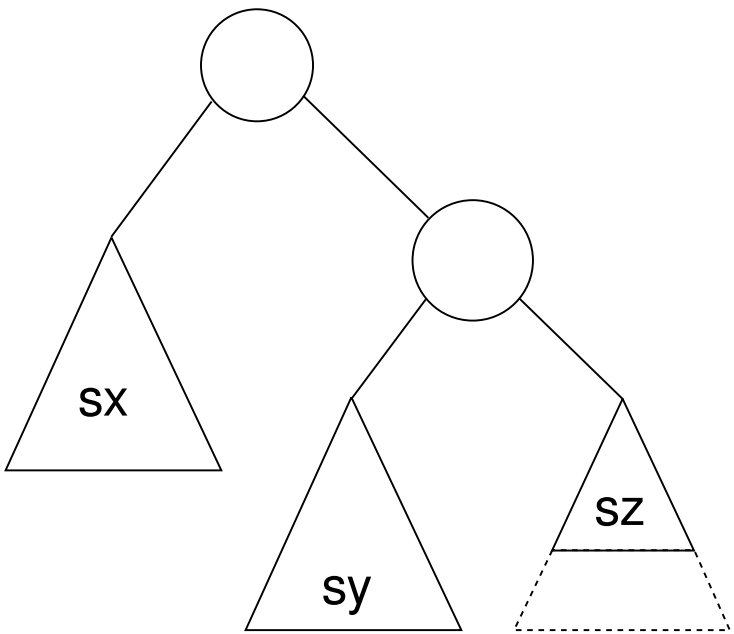
Balanciertheit sicherstellen

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



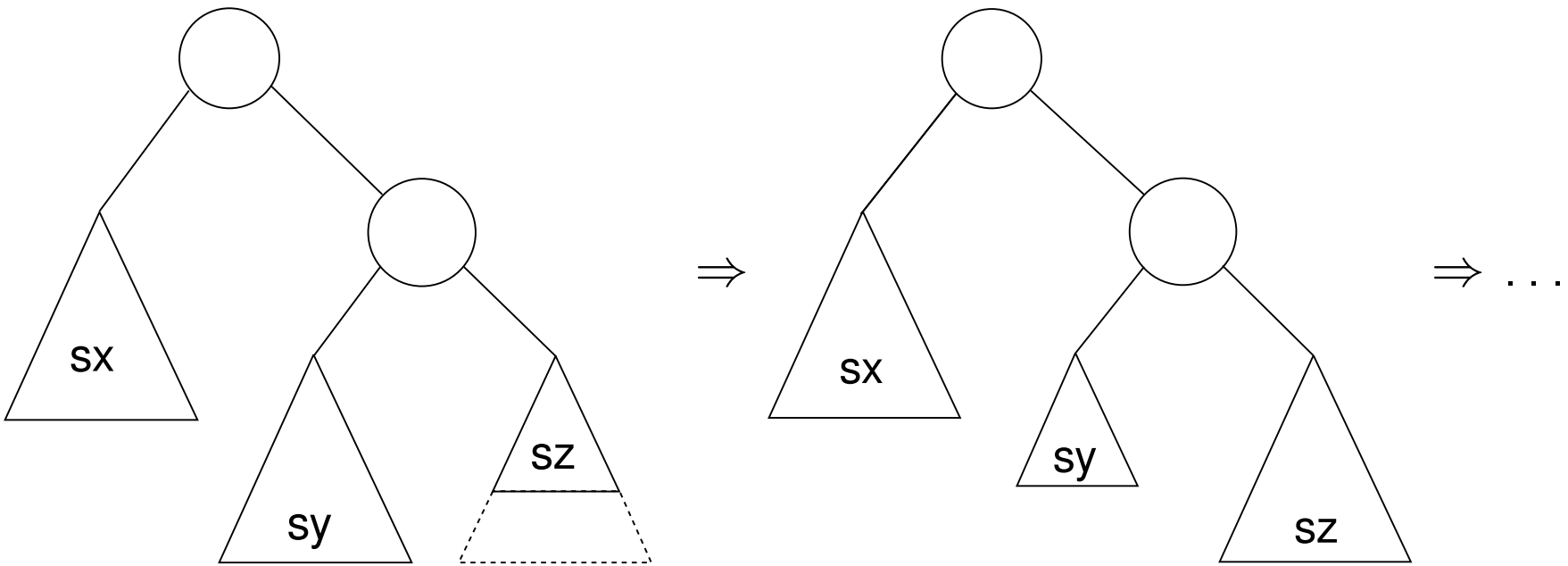
Balanciertheit sicherstellen

- 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 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

```
type Map α β = Tree (α, β)
```

- insert fügt neues Element ein:

```
insert :: Ord α⇒ α→ β→ Map α β→ Map 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)
```

- lookup liest Element aus
- remove löscht ein Element
  - Benötigt Hilfsfunktion join :: Tree α→ Tree α→ Tree α

Zusammenfassung Balancierte Bäume

- 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)

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
- Keine parametrisierten Module
  - Vgl. Lager
  - In ML-Notation:

```
module Lager(Map : MapSig) : LagerSig =...

module Lager1 = Lager(MapList)
module Lager2 = Lager(MapFun)
```

  - In Java: abstrakte Klassen

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 09.12.2012: Signaturen und Eigenschaften

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
  - Abstrakte Datentypen
  - Signaturen und Eigenschaften
  - Spezifikation und Beweis
- 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

```
data Map α β
```

- Leere Abbildung:

```
empty :: Map α β
```

- Abbildung auslesen:

```
lookup :: Ord α⇒ α→ Map α β→ Maybe β
```

- Abbildung ändern:

```
insert :: Ord α⇒ α→ β→ Map α β→ Map α β
```

- Abbildung löschen:

```
delete :: Ord α⇒ α→ Map α β→ Map α β
```

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

Beschreibung von Eigenschaften

Definition (Axiome)

Axiome sind Prädikate über den Operationen der Signatur

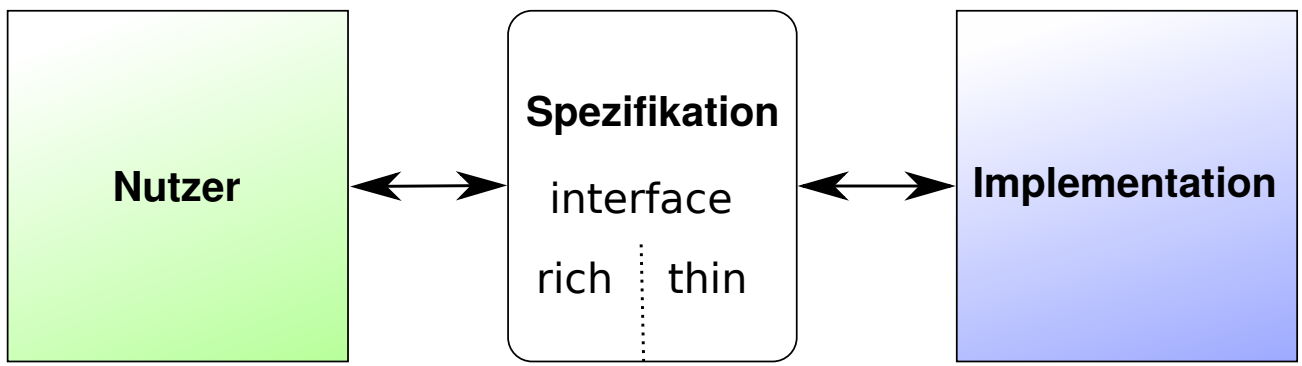
- Elementare Prädikate P :
  - Gleichheit s == t
  - Ordnung s < t
  - Selbstdefinierte Prädikate
- Zusammengesetzte Prädikate
  - Negation not p
  - Konjunktion p && q
  - Disjunktion p || q
  - Implikation p $\Longrightarrow$ q

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

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



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

- Lesen aus leerer Abbildung undefiniert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

```
lookup a empty == Nothing
```

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

```
lookup a (put a v s) == v
```

- Lesen an anderer Stelle liefert alten Wert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

$$\mathsf{lookup\ a\ empty} = \mathsf{Nothing}$$

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\mathsf{lookup\ a\ (put\ a\ v\ s)} = \mathsf{v}$$

- Lesen an anderer Stelle liefert alten Wert:

$$\mathsf{a} \neq \mathsf{b} \Longrightarrow \mathsf{lookup\ a\ (put\ b\ v\ s)} = \mathsf{lookup\ a\ s}$$

- Schreiben an dieselbe Stelle überschreibt alten Wert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

$$\mathsf{lookup\ a\ empty} = \mathsf{Nothing}$$

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\mathsf{lookup\ a\ (put\ a\ v\ s)} = \mathsf{v}$$

- Lesen an anderer Stelle liefert alten Wert:

$$\mathsf{a} \neq \mathsf{b} \Longrightarrow \mathsf{lookup\ a\ (put\ b\ v\ s)} = \mathsf{lookup\ a\ s}$$

- Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\mathsf{put\ a\ w\ (put\ a\ v\ s)} = \mathsf{put\ a\ w\ s}$$

- Schreiben über verschiedene Stellen kommutiert:

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ empty} == \text{Nothing}$$

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (put } a \text{ } v \text{ } s) == v$$

- Lesen an anderer Stelle liefert alten Wert:

$$a \neq b \implies \text{lookup } a \text{ (put } b \text{ } v \text{ } s) == \text{lookup } a \text{ } s$$

- Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\text{put } a \text{ } w \text{ (put } a \text{ } v \text{ } s) == \text{put } a \text{ } w \text{ } s$$

- Schreiben über verschiedene Stellen kommutiert:

$$a \neq b \implies \text{put } a \text{ } v \text{ (put } b \text{ } w \text{ } s) == \text{put } b \text{ } w \text{ (put } a \text{ } v \text{ } s)$$

Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

$$\text{lookup } a \text{ empty} == \text{Nothing}$$

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

$$\text{lookup } a \text{ (put } a \text{ } v \text{ } s) == v$$

- Lesen an anderer Stelle liefert alten Wert:

$$a \neq b \implies \text{lookup } a \text{ (put } b \text{ } v \text{ } s) == \text{lookup } a \text{ } s$$

- Schreiben an dieselbe Stelle überschreibt alten Wert:

$$\text{put } a \text{ } w \text{ (put } a \text{ } v \text{ } s) == \text{put } a \text{ } w \text{ } s$$

- Schreiben über verschiedene Stellen kommutiert:

$$a \neq b \implies \text{put } a \text{ } v \text{ (put } b \text{ } w \text{ } s) == \text{put } b \text{ } w \text{ (put } a \text{ } v \text{ } s)$$

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:

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

```
prop_readPut :: Int→ Maybe String→
                Map Int String→ Bool
prop_readPut a v s =
  lookup a (put a v s) == v
```

- Eigenschaften als Haskell-Prädikate
- Es werden N Zufallswerte generiert und getestet ($N = 100$)

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.

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

Axiome mit QuickCheck testen

- Schreiben:

```
prop_putPut :: Int→ Maybe String→ Maybe String→
               Map Int String→ Bool
prop_putPut a v w s =
  put a w (put a v s) == put a w s
```

- Schreiben an anderer Stelle:

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

- Test benötigt Gleichheit und Zufallswerte für Map a b

Zufallswerte selbst erzeugen

- Problem: Zufällige Werte von selbstdefinierten Datentypen
  - Gleichverteiltheit nicht immer erwünscht (e.g. $[\alpha]$)
  - Konstruktion nicht immer offensichtlich (e.g. Map)
- In QuickCheck:
  - Typklasse **class** Arbitrary α für Zufallswerte
  - Eigene Instanziierung kann Verteilung und Konstruktion berücksichtigen
  - E.g. Konstruktion einer Map:
    - Zufällige Länge, dann aus sovielen zufälligen Werten Map konstruieren
    - Zufallswerte in Haskell?

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):

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

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

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

Eigenschaften von Queue

- First in first out (FIFO):

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

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

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

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

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

Implementation von Stack: Liste

Sehr einfach: ein Stack ist eine Liste

```
newtype 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	Queue	Repräsentation

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
empty			([], [])

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
`empty`			([], [])
`enq 9`		9	([9], [])

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 $\rightarrow$9	([9], [4])

Repräsentation von `Queue`

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

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 $\rightarrow$9	([9], [4])
enq 7		7 $\rightarrow$4 $\rightarrow$9	([9], [7, 4])
deq	9	7 $\rightarrow$4	([4, 7], [])

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 →9	([9], [4])
enq 7		7 →4 →9	([9], [7, 4])
deq	9	7 →4	([4, 7], [])
enq 5		5 →7 →4	([4, 7], [5])

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 $\rightarrow$9	([9], [4])
enq 7		7 $\rightarrow$4 $\rightarrow$9	([9], [7, 4])
deq	9	7 $\rightarrow$4	([4, 7], [])
enq 5		5 $\rightarrow$7 $\rightarrow$4	([4, 7], [5])
enq 3		3 $\rightarrow$5 $\rightarrow$7 $\rightarrow$4	([4, 7], [3, 5])

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 →9	([9], [4])
enq 7		7 →4 →9	([9], [7, 4])
deq	9	7 →4	([4, 7], [])
enq 5		5 →7 →4	([4, 7], [5])
enq 3		3 →5 →7 →4	([4, 7], [3, 5])
deq	4	3 →5 →7	([7], [3, 5])

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 →9	([9], [4])
enq 7		7 →4 →9	([9], [7, 4])
deq	9	7 →4	([4, 7], [])
enq 5		5 →7 →4	([4, 7], [5])
enq 3		3 →5 →7 →4	([4, 7], [3, 5])
deq	4	3 →5 →7	([7], [3, 5])
deq	7	3 →5	([5, 3], [])

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 →9	([9], [4])
enq 7		7 →4 →9	([9], [7, 4])
deq	9	7 →4	([4, 7], [])
enq 5		5 →7 →4	([4, 7], [5])
enq 3		3 →5 →7 →4	([4, 7], [3, 5])
deq	4	3 →5 →7	([7], [3, 5])
deq	7	3 →5	([5, 3], [])
deq	5	3	([3], [])

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 →9	([9], [4])
enq 7		7 →4 →9	([9], [7, 4])
deq	9	7 →4	([4, 7], [])
enq 5		5 →7 →4	([4, 7], [5])
enq 3		3 →5 →7 →4	([4, 7], [3, 5])
deq	4	3 →5 →7	([7], [3, 5])
deq	7	3 →5	([5, 3], [])
deq	5	3	([3], [])
deq	3		([], [])

Repräsentation von `Queue`

Operation	Resultat	Queue	Repräsentation
empty			([], [])
enq 9		9	([9], [])
enq 4		4 →9	([9], [4])
enq 7		7 →4 →9	([9], [7, 4])
deq	9	7 →4	([4, 7], [])
enq 5		5 →7 →4	([4, 7], [5])
enq 3		3 →5 →7 →4	([4, 7], [3, 5])
deq	4	3 →5 →7	([7], [3, 5])
deq	7	3 →5	([5, 3], [])
deq	5	3	([3], [])
deq	3		([], [])
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:

```
instance Eq α⇒ Eq (Qu α) where
  Qu xs1 ys1 == Qu xs2 ys2 =
    xs1 ++ reverse ys1 == xs2 ++ reverse ys2
```

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 18.12.2012: Spezifikation und Beweis

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
  - Abstrakte Datentypen
  - Signaturen und Eigenschaften
  - Spezifikation und Beweis
- Teil III: Funktionale Programmierung im richtigen Leben

Formaler Beweis

- Warum?
  - Formaler Beweis zeigt Korrektheit
- Wie?
  - Formale Notation
  - Beweisformen (Schließregeln)
- Wozu?
  - Formaler Beweis zur Analyse des Algorithmus
  - Haskell als Modellierungssprache

Eigenschaften

- Prädikate:
  - Haskell-Ausdrücke vom Typ Bool
  - Quantifizierte Aussagen:
    Wenn P :: $\alpha \rightarrow$ Bool, dann ist ALL x. P x :: Bool ein Prädikat und
    EX x. P x :: Bool ein Prädikat
- Sonderfall Gleichungen `s == t` und transitive Relationen
- Prädikate müssen nicht ausführbar sein

Wie beweisen?

- Beweis $\longleftrightarrow$ Programmdefinition

Gleichungsumformung	Funktionsdefinition
Fallunterscheidung	Fallunterscheidung (Guards)
Induktion	Rekursive Funktionsdefinition

- Wichtig: formale Notation

Notation

Allgemeine Form:

Lemma (1)	P		
$\Leftrightarrow$	P_1	— Begründung	
$\Leftrightarrow$	P_2	— Begründung	
$\Leftrightarrow$	...		
$\Leftrightarrow$	True		
		$\square$	

Sonderfall Gleichungen:

Lemma (2)	a= b		
	a		
$=$	x_1	— Begründung	
$=$	x_2	— Begründung	
$=$	...		
$=$	b		

$\square$

Beweis durch vollständige Induktion

Zu zeigen:

Für alle natürlichen Zahlen x gilt $P(x)$.

Beweis:

- Induktionsbasis: $P(0)$
- Induktionsschritt:
  - Induktionsvoraussetzung $P(x)$
  - zu zeigen $P(x+1)$

Beweis durch strukturelle Induktion (Listen)

Zu zeigen:

Für alle endlichen Listen xs gilt P xs

Beweis:

- Induktionsbasis: P []
- Induktionsschritt:
  - Induktionsvoraussetzung P xs
  - zu zeigen P (x:xs)

Beweis durch strukturelle Induktion (Allgemein)

Zu zeigen:

$$\text{Für alle } x \text{ in } T \text{ gilt } P(x)$$

Beweis:

- Für jeden Konstruktor C_i:
  - Voraussetzung: für alle $t_{i,j}$ gilt $P(t_{i,j})$
  - zu zeigen $P(C_i\ t_{i,1}\ \ldots\ t_{i,k_i})$

Beweisstrategien

- Fold-Unfold:
  - Im Induktionsschritt Funktionsdefinition auffalten
  - Ausdruck umformen, bis Induktionsvoraussetzung anwendbar
  - Funktionsdefinitionen zusammenfalten
- Induktionsvoraussetzung stärken:
  - Stärkere Behautung $\Longrightarrow$ stärkere Induktionsvoraussetzung, daher:
  - um Behauptung P zu zeigen, stärkere Behauptung P' zeigen, dann P als Korollar

Zusammenfassung

- Beweise beruhen auf:
  - Gleichungs- und Äquivalenzumformung
  - Fallunterscheidung
  - Induktion
- Beweisstrategien:
  - Sinnvolle Lemmata
  - Fold/Unfold
  - Induktionsvoraussetzung stärken
- Warum Beweisen?
  - Korrektheit von Haskell-Programmen
  - Haskell als Modellierungssprache

Zusammenfassung

- Beweise beruhen auf:
  - Gleichungs- und Äquivalenzumformung
  - Fallunterscheidung
  - Induktion
- Beweisstrategien:
  - Sinnvolle Lemmata
  - Fold/Unfold
  - Induktionsvoraussetzung stärken
- Warum Beweisen?
  - Korrektheit von Haskell-Programmen
  - Haskell als Modellierungssprache

Frohes Fest!

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 11 vom 06.01.2015: Aktionen und Zustände

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Frohes Neues Jahr!

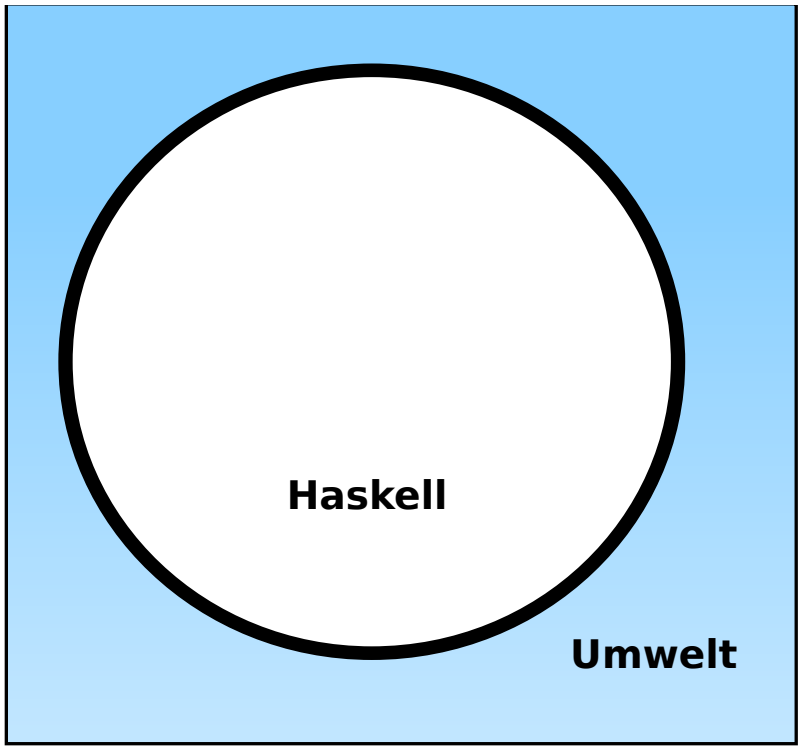
Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Effizienzaspekte
  - Eine Einführung in Scala
  - Rückblich & Ausblick

Inhalt

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

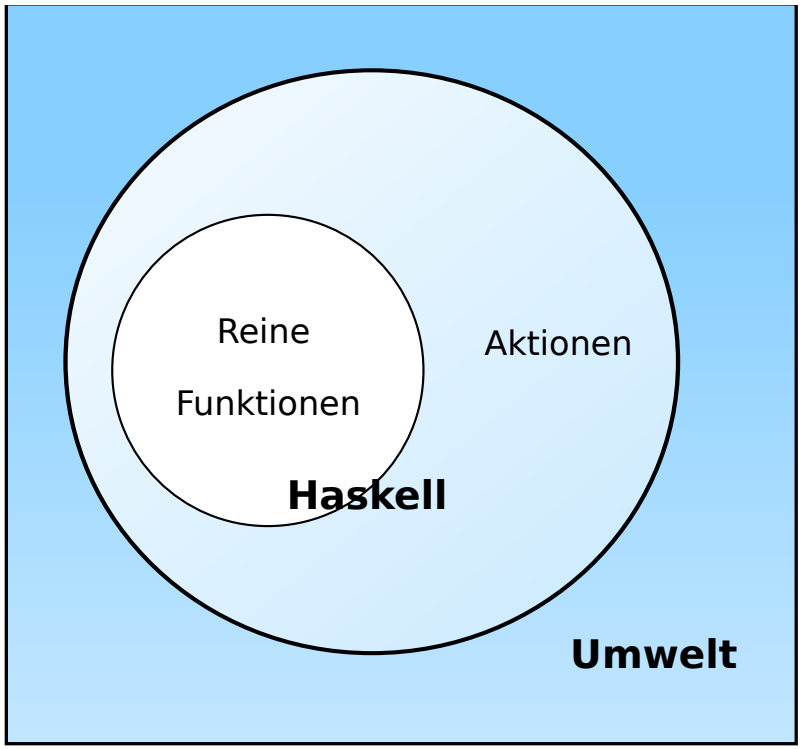
Ein- und Ausgabe in funktionalen Sprachen



Problem:

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

Ein- und Ausgabe in funktionalen Sprachen



Problem:

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

Lösung:

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

Aktionen als abstrakter Datentyp

- ADT mit Operationen Komposition und Lifting
- Signatur:

```
type IO α

(>>=)  :: IO α → (α→ IO β) → IO β

return :: α→ IO α
```

- Plus elementare Operationen (lesen, schreiben etc)

Elementare Aktionen

- Zeile von stdin lesen:

```
getLine  :: IO String
```

- Zeichenkette auf stdout ausgeben:

```
putStr   :: String→ IO ()
```

- Zeichenkette mit Zeilenvorschub ausgeben:

```
putStrLn :: String→ IO ()
```

Einfache Beispiele

- Echo einfach

```
echo1 :: IO ()
echo1 = getLine >>= putStrLn
```

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

Module in der Standardbücherei

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

Ein/Ausgabe mit Dateien

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

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

- Datei lesen (verzögert):

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

- Mehr Operationen im Modul IO der Standardbücherei
  - Buffered/Unbuffered, Seeking, &c.
  - Operationen auf Handle

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
```

- Vordefinierte Kontrollstrukturen (Control.Monad):
  - when, mapM, forM, sequence, ...

Fehlerbehandlung

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

```
catch :: Exception e⇒ IO α → (e→ IO α) → IO α
try   :: Exception e⇒ IO α → IO (Either e a)
```

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

Fehler fangen und behandeln

- Fehlerbehandlung für wc:

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

- IOError kann analysiert werden (siehe System.IO.Error)

- read mit Ausnahme bei Fehler (statt Programmabbruch):

```
readIO :: Read a⇒ String→ IO a
```

Ausführbare Programme

- Eigenständiges Programm ist Aktion
- Hauptaktion: main :: IO () in Modul Main
- wc als eigenständiges Programm:

```
module Main where

import System.Environment (getArgs)
import Control.Exception
```

...

```
main :: IO ()
main = do
  args ← getArgs
  mapM_ wc2 args
```

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)
```

  - Zufälliges Element aus einer nicht-leeren Liste auswählen:

```
pickRandom :: [α] → IO α
pickRandom [] = error "pickRandom: empty list"
pickRandom xs = do
  i ← randomRIO (0, length xs − 1)
  return $ xs !! i
```

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 String
```

  - Argumente: geratene Zeichen (im Geheimnis enthalten, nicht enthalten)

- Hauptfunktion:

```
main :: IO ()
```

  - Liest ein Lexikon, wählt Geheimnis aus, ruft Hauptschleife auf

Funktionen mit Zustand

Theorem (Currying)

Folgende Typen sind isomorph:

$$A \times B \to C \cong A \to B \to C$$

- In Haskell: folgende Funktionen sind invers:

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

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

In Haskell: Zustände explizit

- Datentyp: 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
```

- Lifting:

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

Beispiel: Ein Zähler

- Datentyp:

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

- Zähler erhöhen:

```
tick :: WithCounter ()
tick i = ((), i+1)
```

- Zähler auslesen:

```
read :: WithCounter Int
read i = (i, i)
```

- Zähler zurücksetzen:

```
reset :: WithCounter ()
reset i = ((), 0)
```

Implizite vs. explizite Zustände

- Nachteil: Zustand ist explizit
  - Kann dupliziert werden
- Daher: Zustand implizit machen
  - Datentyp verkapseln
  - Signatur State, comp, lift , elementare Operationen

Aktionen als Zustandstransformationen

- Idee: Aktionen sind Transformationen auf Systemzustand S
- S beinhaltet
  - Speicher als Abbildung $A \rightharpoonup V$ (Adressen A, Werte V)
  - Zustand des Dateisystems
  - Zustand des Zufallsgenerators
- In Haskell: Typ RealWorld
  - "Virtueller" Typ, Zugriff nur über elementare Operationen
  - Entscheidend nur Reihenfolge der Aktionen

Zusammenfassung

- 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).
- Verschiedene Funktionen der Standardbücherei:
  - Prelude: getLine, putStr, putStrLn, readFile, writeFile
  - Module: IO, Random
- Aktionen sind implementiert als Zustandstransformationen

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 12 vom 13.01.2015: Effizienzaspekte

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Effizienzaspekte
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Organisatorisches

- Fachgespräche: 2/3. Februar oder 9/10. Februar?

- Zusatztermin: 27. Februar

Inhalt

- Zeitbedarf: Endrekursion — while in Haskell
- Platzbedarf: Speicherlecks
- "Unendliche" Datenstrukturen
- Verschiedene andere Performancefallen:
  - Überladene Funktionen, Listen

Inhalt

- Zeitbedarf: Endrekursion — while in Haskell
- Platzbedarf: Speicherlecks
- "Unendliche" Datenstrukturen
- Verschiedene andere Performancefallen:
  - Überladene Funktionen, Listen
- "Usual Disclaimers Apply":
  - Erste Lösung: bessere Algorithmen
  - Zweite Lösung: Büchereien nutzen

Effizienzaspekte

- Zur Verbesserung der Effizienz:
  - Analyse der Auswertungsstrategie
  - ... und des Speichermanagement
- Der ewige Konflikt: Geschwindigkeit vs. Platz

Effizienzaspekte

- Zur Verbesserung der Effizienz:
  - Analyse der Auswertungsstrategie
  - ... und des Speichermanagement
- Der ewige Konflikt: Geschwindigkeit vs. Platz
- Effizenzverbesserungen durch
  - Endrekursion: Iteration in funktionalen Sprachen
  - Striktheit: Speicherlecks vermeiden (bei verzögerter Auswertung)
- Vorteil: Effizienz muss nicht im Vordergrund stehen

Endrekursion

Definition (Endrekursion)

Eine Funktion ist endrekursiv, wenn

(i) es genau einen rekursiven Aufruf gibt,

(ii) der nicht innerhalb eines geschachtelten Ausdrucks steht.

- D.h. darüber nur Fallunterscheidungen: **case** oder **if**
- Entspricht goto oder while in imperativen Sprachen.
- Wird in Sprung oder Schleife übersetzt.
- Braucht keinen Platz auf dem Stack.

Einfaches Beispiel

- In Haskell:

```
even x = if x< 1 then x == 0 else even (x-2)
```

- Übersetzt nach C:

```
int even (int x)
{ if (x<1) return x == 0;
  else return (even (x-2));  }
```

- Äquivalente Formulierung:

```
int even (int x)
{ if (!(x< 1)) { x -= 2; return even(x); }
  else return x == 0; }
```

Einfaches Beispiel

- In Haskell:

```
even x = if x< 1 then x == 0 else even (x-2)
```

- Übersetzt nach C:

```
int even (int x)
{ if (x<1) return x == 0;
  else return (even (x-2));  }
```

- Äquivalente Formulierung:

```
int even (int x)
{ if (!(x< 1)) { x -= 2; return even(x); }
  else return x == 0; }
```

Einfaches Beispiel

- In Haskell:

```
even x = if x< 1 then x == 0 else even (x-2)
```

- Übersetzt nach C:

```
int even (int x)
{ if (x<1) return x == 0;
  else return (even (x-2));  }
```

- Äquivalente Formulierung:

```
int even (int x)
{ if (!(x< 1)) { x -= 2; return even(x); }
  else return x == 0; }
```

- Iterative Variante mit Schleife:

```
int even (int x)
{ while (!(x<1)) x -= 2;
  return x == 0;  }
```

Beispiel: Fakultät

- fac1 nicht endrekursiv:

```
fac1 :: Integer→ Integer
fac1 n = if n == 0 then 1 else n * fac1 (n−1)
```

Beispiel: Fakultät

- fac1 nicht endrekursiv:

```
fac1 :: Integer→ Integer
fac1 n = if n == 0 then 1 else n * fac1 (n−1)
```

- fac2 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?!

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?

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: Speicherlecks!

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: Speicherlecks!
- 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 :: $\alpha \to \beta \to \beta$

```
⊥ `seq` b  =  ⊥
a `seq` b  =  b
```

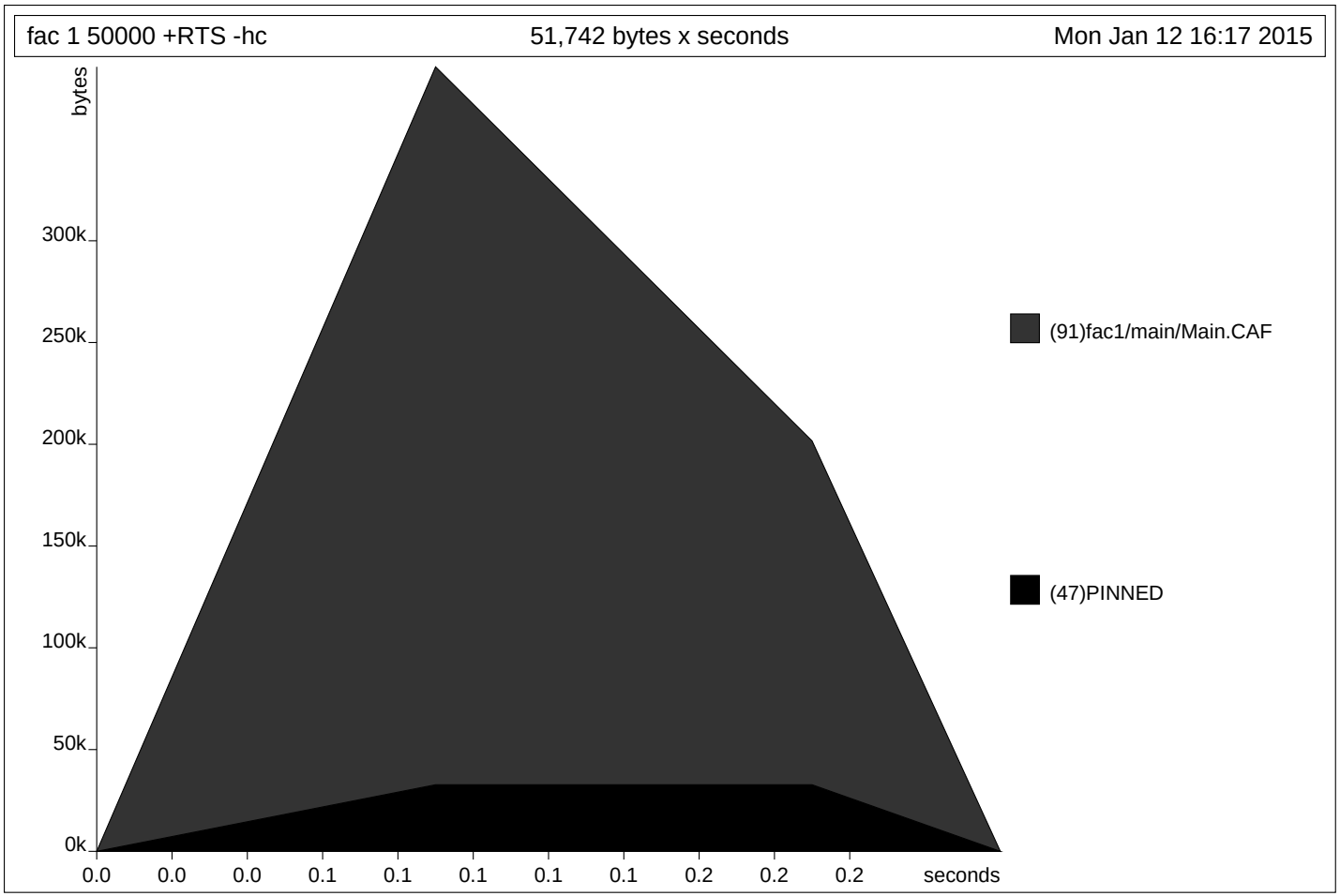
  - seq vordefiniert (nicht in Haskell definierbar)
  - ($!) :: (a→ b)→ a→ b strikte Funktionsanwendung

```
f $! x = x `seq` f x
```

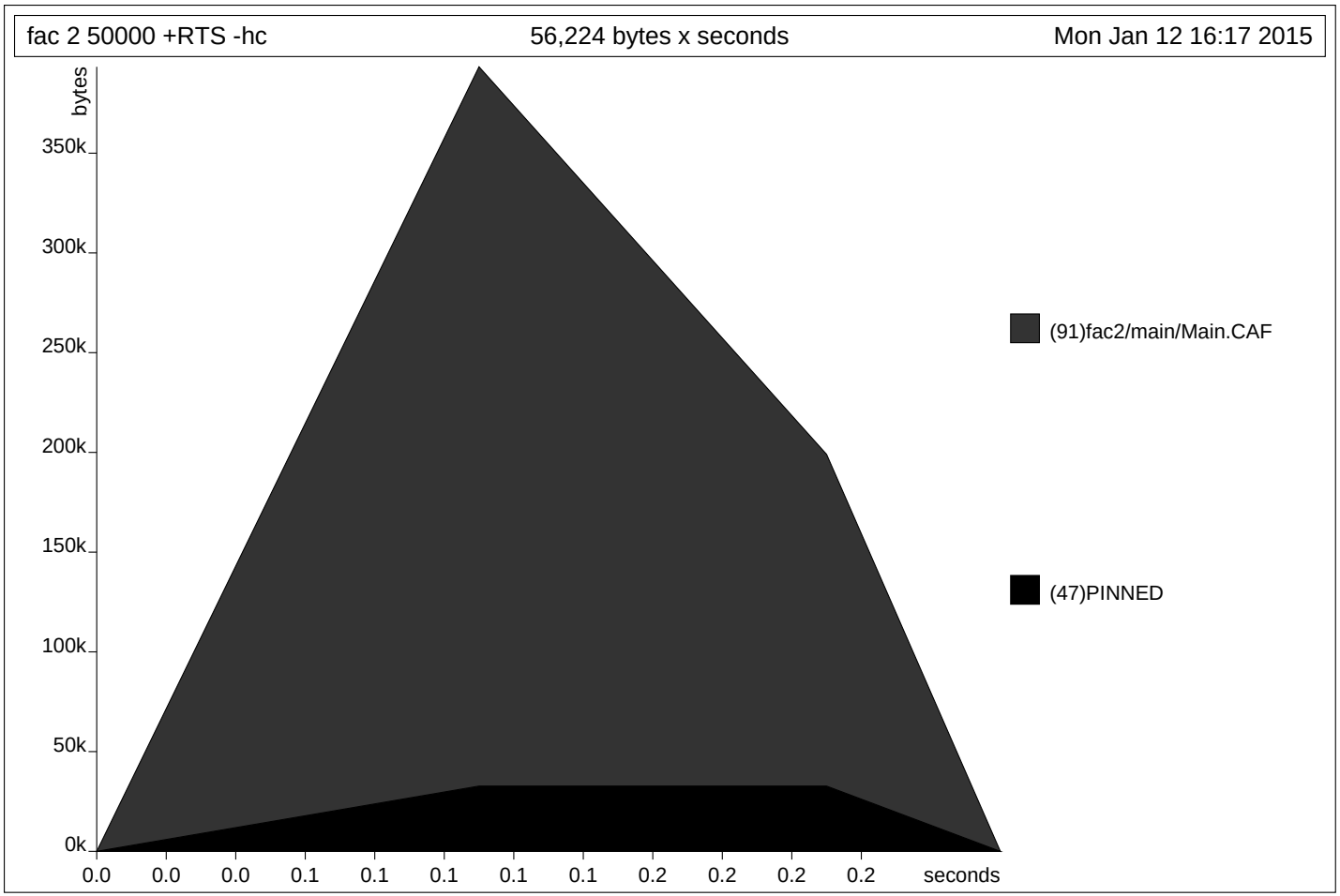
  - ghc macht Striktheitsanalyse
- Fakultät in konstantem Platzaufwand

```
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)
```

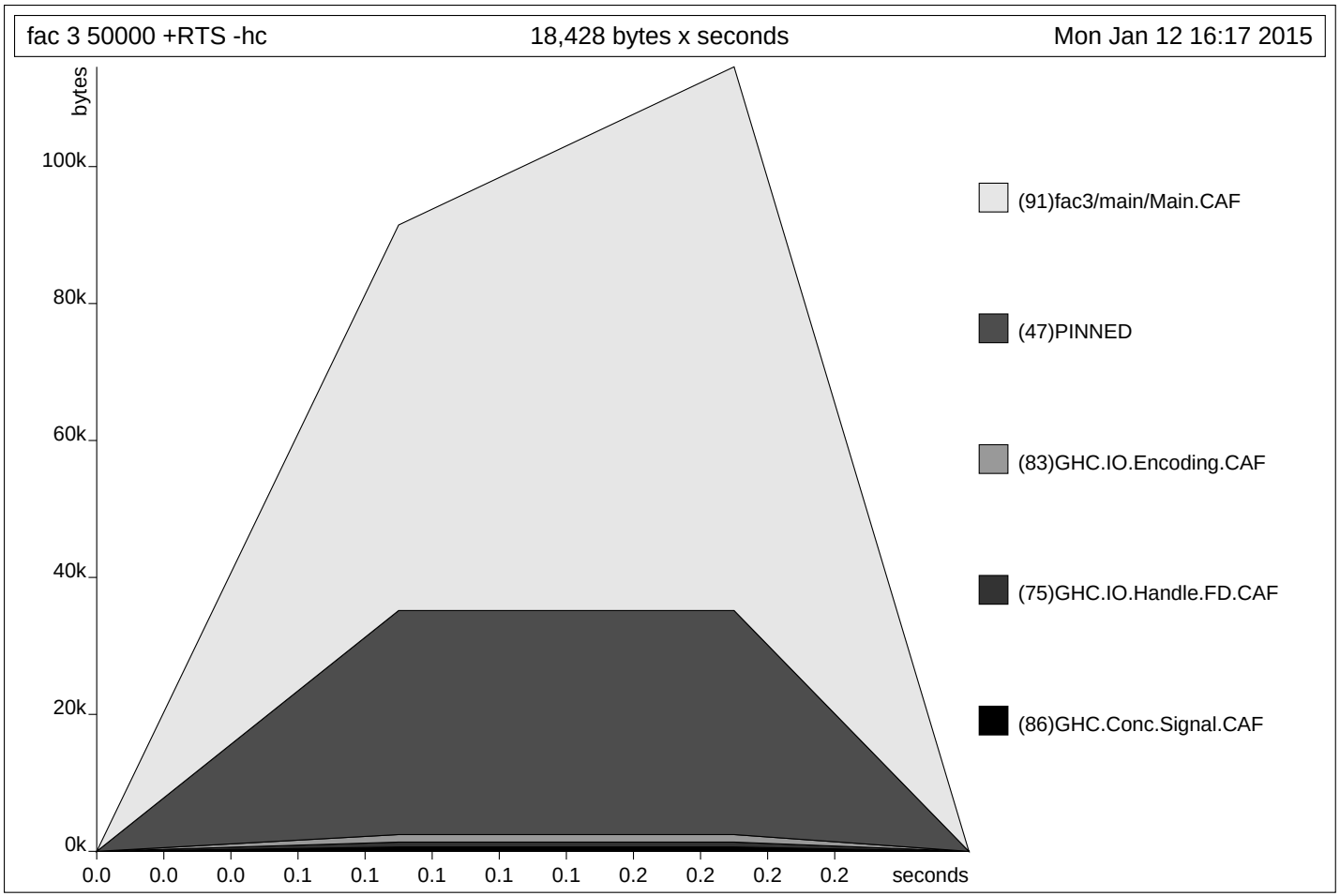
Speicherprofil: fac1 50000, nicht optimiert



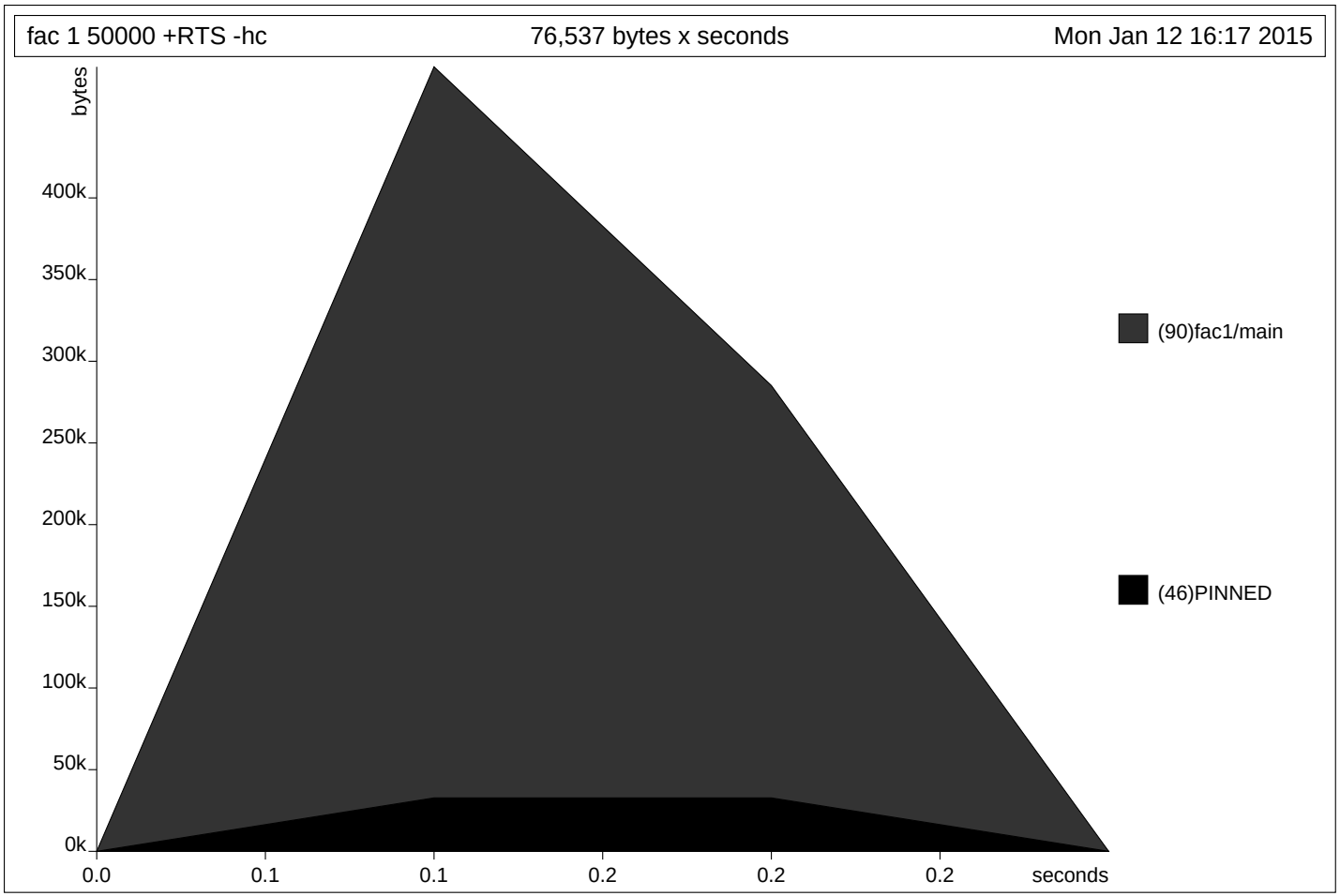
Speicherprofil: fac2 50000, nicht optimiert



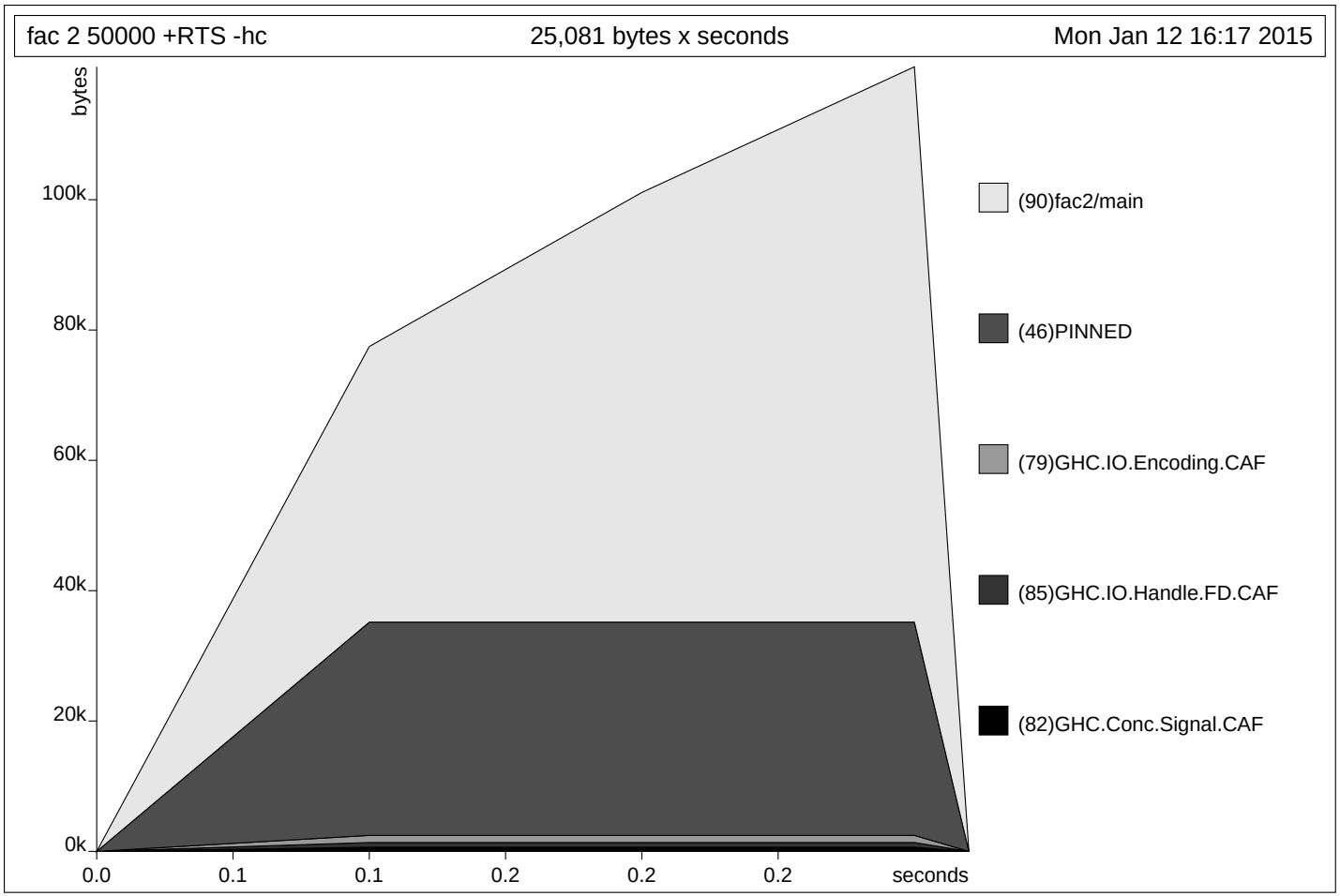
Speicherprofil: fac3 50000, nicht optimiert



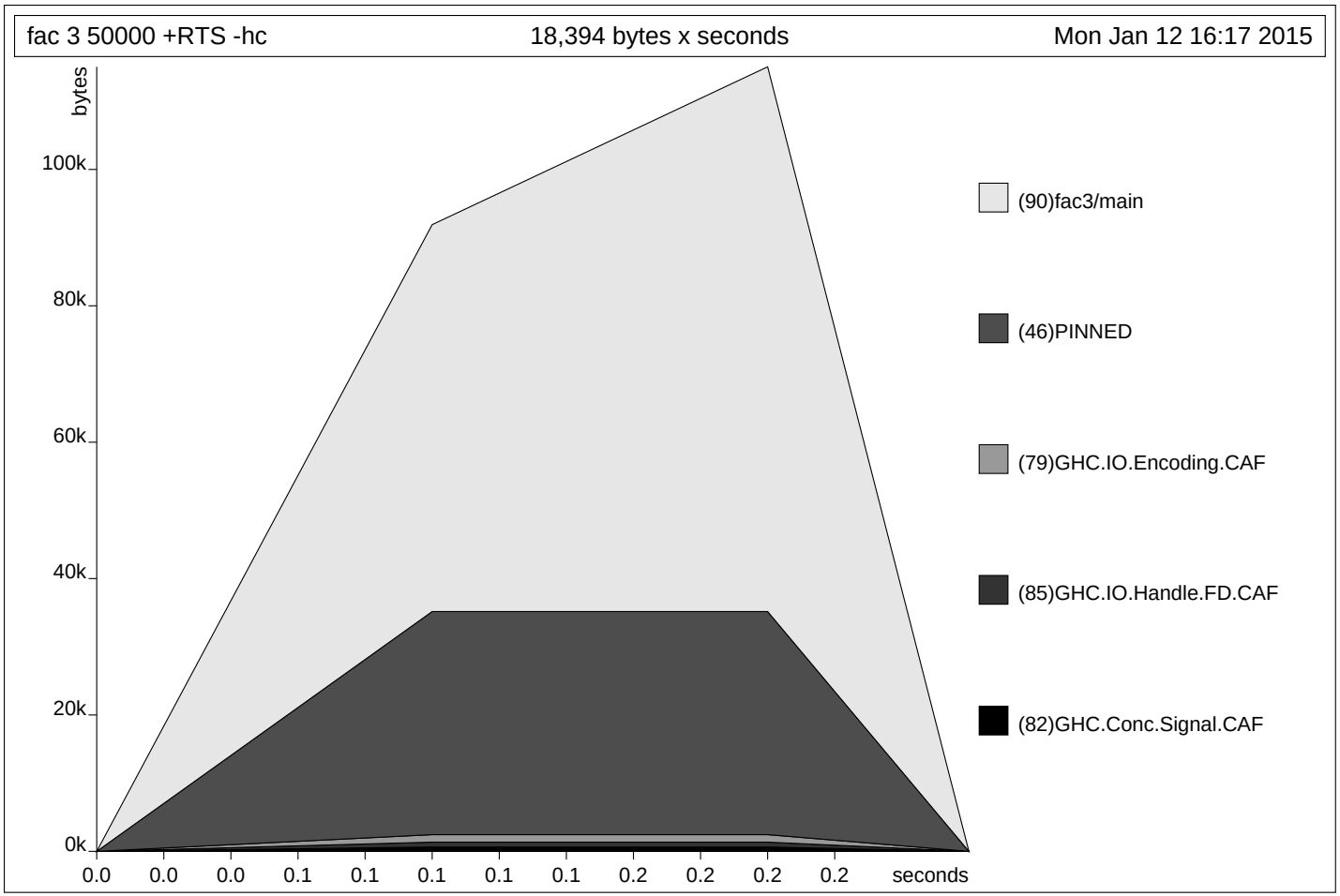
Speicherprofil: fac1 50000, optimiert



Speicherprofil: fac2 50000, 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

Überführung in Endrekursion

- Gegeben Funktion

$$
\begin{aligned}
&f' : S \to T \\
&f'\ x\ =\ \textbf{if}\ B\ x\ \ \textbf{then}\ H\ x \\
&\qquad\qquad\qquad\ \ \textbf{else}\ \phi\ (f'\ (K\ x))\ (E\ x)
\end{aligned}
$$

  - Mit $K : S \to S$, $\phi : T \to T \to T$, $E : S \to T$, $H : S \to T$.

- Voraussetzung: ϕ assoziativ, $e : T$ neutrales Element

- Dann ist endrekursive Form:

$$
\begin{aligned}
&f : S \to T \\
&f\ x\ =\ g\ x\ e\ \textbf{where} \\
&\qquad g\ x\ y\ =\ \textbf{if}\ B\ x\ \ \textbf{then}\ \phi\ (H\ x)\ y \\
&\qquad\qquad\qquad\ \ \textbf{else}\ g\ (K\ x)\ (\phi\ (E\ x)\ y)
\end{aligned}
$$

Beispiel

- Länge einer Liste (nicht-endrekursiv)

```
length' :: [a]→ Int
length' xs = if null xs then 0
             else 1+ length' (tail xs)
```

- Zuordnung der Variablen:

$$\begin{array}{rcl@{\qquad}rcl} K(x) & \mapsto & \text{tail } x & B(x) & \mapsto & \text{null } x \\ E(x) & \mapsto & 1 & H(x) & \mapsto & 0 \\ \phi(x,y) & \mapsto & x+y & e & \mapsto & 0 \end{array}$$

- Es gilt: $\phi(x, e) = x + 0 = x$ (0 neutrales Element)

Beispiel

- Damit endrekursive Variante:

```
length :: [a]→ Int
length xs = len xs 0 where
  len xs y = if null xs then y -- was: y+ 0
             else len (tail xs) (1+ y)
```

- Allgemeines Muster:
  - Monoid (ϕ, e): ϕ assoziativ, e neutrales Element.
  - Zusätzlicher Parameter akkumuliert Resultat.

Weiteres Beispiel: foldr vs. foldl

- foldr ist nicht endrekursiv:

```
foldr :: (a → b → b) → b → [a] → b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- foldl ist endrekursiv:

```
foldl :: (a → b → a) → a → [b] → a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Weiteres Beispiel: foldr vs. foldl

- foldr ist nicht endrekursiv:

```
foldr :: (a → b → b) → b → [a] → b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

- foldl ist endrekursiv:

```
foldl :: (a → b → a) → a → [b] → a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- foldl' ist strikt und endrekursiv:

```
foldl' :: (a→ b→ a)→ a→ [b]→ a
foldl' f a []     = a
foldl' f a (x:xs) =
  let a' = f a x in a' `seq` foldl' f a' xs
```

- Für Monoid (ϕ, e) gilt: foldr ϕ e l = foldl (flip ϕ) e l

Wann welches fold?

- foldl endrekursiv, aber traversiert immer die ganze Liste.
  - foldl' ferner strikt und konstanter Platzaufwand
- Wann welches fold?
  - Strikte Funktionen mit foldl' falten:

```
rev2 :: [a]→ [a]
rev2 = foldl' (flip (:)) []
```

  - Wenn nicht die ganze Liste benötigt wird, mit foldr falten:

```
all :: (a→ Bool)→ [a]→ Bool
all p = foldr ((&&) ∘ p) True
```

  - Potenziell unendliche Listen immer mit foldr falten.

Endrekursive Aktionen

- Nicht endrekursiv:

```
getLines' :: IO String
getLines' = do str← getLine
               if null str then return ""
                  else  do rest← getLines'
                           return (str++ rest)
```

- Endrekursiv:

```
getLines :: IO String
getLines = getit "" where
   getit res = do str← getLine
                  if null str then return res
                     else getit (res++ str)
```

Fortgeschrittene Endrekursion

- Akkumulation von Ergebniswerten durch partiell applizierte Funktionen
  - Sonderfall von Continuations: es wird nicht das Ergebnis zurückgegeben, sondern eine Funktion, welche das Ergebnis erhält
- Beispiel: die Klasse `Show`
  - Nur Methode `show` wäre zu langsam ($O(n^2)$):

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

Fortgeschrittene Endrekursion

- Akkumulation von Ergebniswerten durch partiell applizierte Funktionen
  - Sonderfall von Continuations: es wird nicht das Ergebnis zurückgegeben, sondern eine Funktion, welche das Ergebnis erhält
- Beispiel: die Klasse `Show`
  - Nur Methode `show` wäre zu langsam ($O(n^2)$):

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

  - Deshalb zusätzlich

```
  showsPrec :: Int→ a→ String→ String
  show x = showsPrec 0 x ""
```

  - String wird erst aufgebaut, wenn er ausgewertet wird ($O(n)$).

Beispiel: Mengen als Listen

```
data Set a = Set [a]
```

Zu langsam wäre

```
instance Show a⇒ Show (Set a) where
  show (Set elems) =
     "{" ++ intercalate "," (map show elems) ++ "}"
```

Deshalb besser

```
instance Show a⇒ Show (Set a) where
  showsPrec i (Set elems) = showElems elems where
    showElems []     = ("{}" ++)
    showElems (x:xs) = ('{' :) ∘ shows x ∘ showl xs
       where showl []     = ('}' :)
             showl (x:xs) = (',' :) ∘ shows x ∘ showl xs
```

Effizienz durch “unendliche” Datenstrukturen

- Listen müssen nicht endlich repräsentierbar sein:
  - Nützlich für Listen mit unbekannter Länge
  - Allerdings Induktion nur für endliche Listen gültig.
- Beispiel: Fibonacci-Zahlen
  - Aus der Kaninchenzucht.
  - Sollte jeder Informatiker kennen.

```
fib' :: Int→ Integer
fib' 0 = 1
fib' 1 = 1
fib' n = fib' (n−1)+ fib' (n−2)
```

  - Problem: baumartige Rekursion, exponentieller Aufwand.

Fibonacci-Zahlen als Strom

- Lösung: zuvor berechnete Teilergebnisse wiederverwenden.
- Sei `fibs :: [Integer]` Strom aller Fibonaccizahlen:

```
           fibs  1  1  2  3  5  8 13 21 34 55
      tail fibs  1  2  3  5  8 13 21 34 55
tail (tail fibs)  2  3  5  8 13 21 34 55
```

- Damit ergibt sich:

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- n-te Fibonaccizahl mit `fibs !! n`
- Aufwand: linear, da `fibs` nur einmal ausgewertet wird.

Implementation und Repräsentation von Datenstrukturen

- Datenstrukturen werden intern durch Objekte in einem Heap repräsentiert
- Bezeichner werden an Referenzen in diesen Heap gebunden
- Unendliche Datenstrukturen haben zyklische Verweise
  - Kopf wird nur einmal ausgewertet.

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

- Anmerkung: unendlich Datenstrukturen nur sinnvoll für nicht-strikte Funktionen

Überladene Funktionen sind langsam.

- Typklassen sind elegant aber langsam.
  - Implementierung von Typklassen: Verzeichnis (dictionary) von Klassenfunktionen.
  - Überladung wird zur Laufzeit aufgelöst

Überladene Funktionen sind langsam.

- Typklassen sind elegant aber langsam.
  - Implementierung von Typklassen: Verzeichnis (dictionary) von Klassenfunktionen.
  - Überladung wird zur Laufzeit aufgelöst
- Bei kritischen Funktionen: Spezialisierung erzwingen durch Angabe der Signatur
- NB: Zahlen (numerische Literale) sind in Haskell überladen!
  - Bsp: `facs` hat den Typ `Num a=> a-> a`

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

Listen als Performance-Falle

- Listen sind keine Felder oder endliche Abbildungen
- Listen:
  - Beliebig lang
  - Zugriff auf n-tes Element in linearer Zeit ($O(n)$)
  - Abstrakt: frei erzeugter Datentyp aus Kopf und Rest
- Felder `Array ix a` (Modul Data.Array aus der Standardbücherei)
  - Feste Größe (Untermenge von ix)
  - Zugriff auf n-tes Element in konstanter Zeit ($O(1)$)
  - Abstrakt: Abbildung Index auf Daten
- Endliche Abbildung `Map k v` (Modul Data.Map)
  - Beliebige Größe
  - Zugriff auf n-tes Element in sublinearer Zeit ($O(\log n)$)
  - Abstrakt: Abbildung Schlüsselbereich `k` auf Wertebereich `v`
  - Sonderfall: `Set k` $\equiv$ `Map k Bool`

Zusammenfassung

- Endrekursion: while für Haskell.
  - Überführung in Endrekursion meist möglich.
  - Noch besser sind strikte Funktionen.
- Speicherlecks vermeiden: Striktheit und Endrekursion
- Compileroptimierung nutzen
- Datenstrukturen müssen nicht endlich repräsentierbar sein
- Überladene Funktionen sind langsam.
- Listen sind keine Felder oder endliche Abbildungen.

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 13 vom 20.01.15: Scala — Eine praktische Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Effizienzaspekte
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Heute: Scala

- A scalable language
- Rein objektorientiert
- Funktional
- Eine "JVM-Sprache"
- Seit 2004 von Martin Odersky, EPFL Lausanne (`http://www.scala-lang.org/`).
- Seit 2011 kommerziell durch Typesafe Inc.

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

- Interaktive Auswertung
- Variablen, veränderlich
- Werte, unveränderlich
- `while`-Schleifen
- Rekursion — einfache Endrekursion wird optimiert
- Typinferenz — mehr als Java, weniger als Haskell

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

- Interaktive Auswertung
- Variablen, veränderlich — *Mit Vorsicht benutzen!*
- Werte, unveränderlich
- `while`-Schleifen
- Rekursion — einfache Endrekursion wird optimiert
- Typinferenz — mehr als Java, weniger als Haskell

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

- Interaktive Auswertung
- Variablen, veränderlich — *Mit Vorsicht benutzen!*
- Werte, unveränderlich
- `while`-Schleifen — *Unnötig!*
- Rekursion — einfache Endrekursion wird optimiert
- Typinferenz — mehr als Java, weniger als Haskell

Scala am Beispiel: `02-Rational.scala`

Was sehen wir hier?

Scala am Beispiel: `02-Rational.scala`

Was sehen wir hier?

- Klassenparameter
- `this`
- Methoden, Syntax für Methodenanwendung
- `override` (nicht optional)
- private Werte und Methoden
- Klassenvorbedingunge (`require`)
- Overloading
- Operatoren

Algebraische Datentypen: `03-Expr.scala`

Was sehen wir hier?

Algebraische Datentypen: `03-Expr.scala`

Was sehen wir hier?

- `case class` erzeugt
  - Factory-Methode für Konstruktoren
  - Parameter als implizite `val`
  - abgeleitete Implementierung für `toString`, `equals`
  - ...und pattern matching
- 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

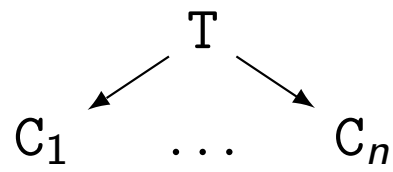
Implementierung algebraischer Datentypen

Haskell:

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

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



- Varianten als Subtypen
- Problem und Vorteil:

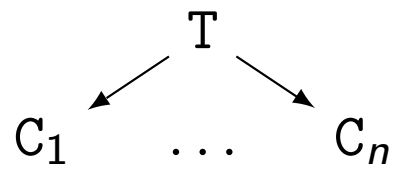
Implementierung algebraischer Datentypen

Haskell:

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

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



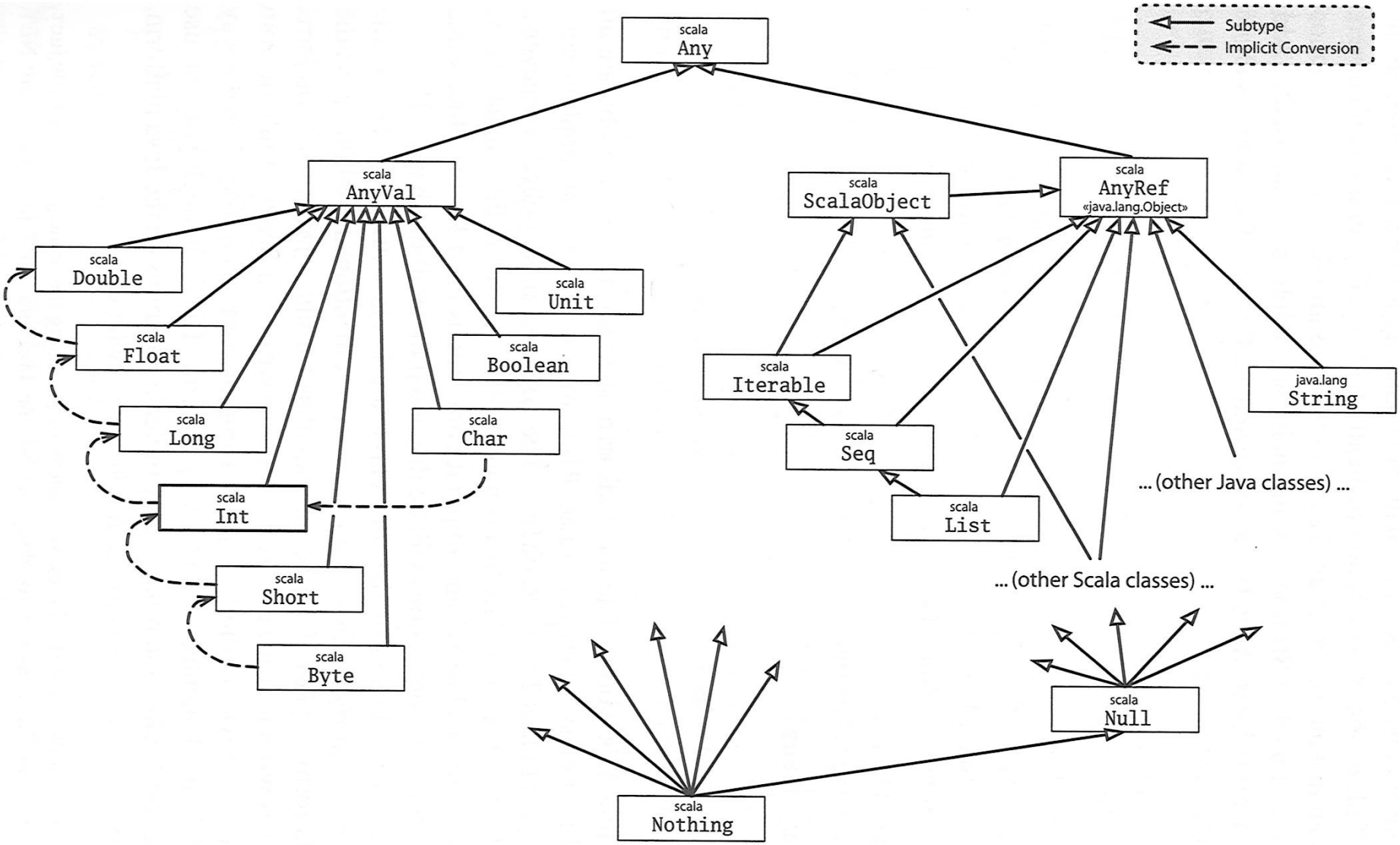
- Varianten als Subtypen
- Problem und Vorteil: Erweiterbarkeit
- `sealed` verhindert Erweiterung

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. `List[T]`
- Problem: Vererbung und Polymorphie
- Ziel: wenn `S` $<$ `T`, dann `List[S]` $<$ `List[T]`
- Does not work — `04-Ref.hs`

Parametrische Polymorphie

- Typparameter (wie in Haskell, Generics in Java), Bsp. `List[T]`
- Problem: Vererbung und Polymorphie
- Ziel: wenn `S` $<$ `T`, dann `List[S]` $<$ `List[T]`
- Does not work — `04-Ref.hs`
- Warum?
  - Funktionsraum nicht monoton im ersten Argument
  - Sei $X \subseteq Y$, dann $Z \longrightarrow X \subseteq Z \longrightarrow Y$, aber $X \longrightarrow Z \not\subseteq Y \longrightarrow Z$
  - Sondern $Y \longrightarrow Z \subseteq X \longrightarrow Z$

Typvarianz

`class C[+T]`

- Kovariant
- `S` $<$ `T`, dann `C[S]` $<$ `C[T]`
- Parameter `T` nur im Wertebereich von Methoden

`class C[T]`

- Rigide
- Kein Subtyping
- Parameter `T` kann beliebig verwendet werden

`class C[-T]`

- Kontravariant
- `S` $<$ `T`, dann `C[T]` $<$ `C[S]`
- Parameter `T` nur im Definitionsbereich von Methoden

Beispiel:

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

Traits: `05-Funny.scala`

Was sehen wir hier?

- `Traits` (Mix-ins): abstrakte Klassen, Interfaces; Haskell: Typklassen
- Unterschied zu Klassen:
  - Keine Parameter
  - Keine feste Oberklasse (`super` dynamisch gebunden)
- Nützlich zur Strukturierung:

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

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

Was wir ausgelassen haben...

- Komprehension (nicht nur für Listen)
- Gleichheit (==, `equals`)
- Implizite Parameter und Typkonversionen
- Nebenläufigkeit (Aktoren)

Scala — Die Sprache

- Objekt-orientiert:
  - Veränderlicher, gekapselter Zustand
  - Subtypen und Vererbung
  - Klassen und Objekte
- Funktional:
  - Unveränderliche Werte
  - Polymorphie
  - Funktionen höherer Ordnung

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 unsere Veranstaltung Reaktive Programmierung

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 14 vom 27.01.15: Schlußbemerkungen

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Effizienzaspekte
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Organisatorisches

- Scheinvordruck benötigt?

- Bitte an der Online-Evaluation teilnehmen (stud.ip)

Inhalt

- Wiederholung
- Rückblick
- Ausblick

Fachgespräch: Beispielaufgabe

Fachgespräch: Beispielaufgabe

Definieren Sie eine Funktion leastSpaces, welche aus einer nicht-leeren Liste von Zeichenketten diejenige zurückgibt, welche die wenigsten Leerzeichen enthält.

Beispiel:

leastSpaces ["a␣bc", "pqr", "x␣y␣z"] $\rightsquigarrow$ "pqr"

Verständnisfragen

Auf allen Übungsblättern finden sich Verständnisfragen zur Vorlesung. Diese sind nicht Bestandteil der Abgabe, können aber im Fachgesprach thematisiert werden. Wenn Sie das Gefühl haben, diese Fragen nicht sicher beantworten zu können, wenden Sie sich gerne an Ihren Tutor, an Berthold Hoffmann in seiner Fragestunde, oder an den Dozenten.

Verständnisfragen

1. Was bedeutet Striktheit, und welche in Haskell definierbaren Funktionen haben diese Eigenschaft?

Achtung: waren nicht auf dem Übungsblatt 1.

Verständnisfragen

1. Was bedeutet Striktheit, und welche in Haskell definierbaren Funktionen haben diese Eigenschaft?

2. Was ist ein algebraischer Datentyp, und was ist ein Konstruktor?

Achtung: waren nicht auf dem Übungsblatt 1.

Verständnisfragen

1. Was bedeutet Striktheit, und welche in Haskell definierbaren Funktionen haben diese Eigenschaft?

2. Was ist ein algebraischer Datentyp, und was ist ein Konstruktor?

3. Was sind die drei Eigenschaften, welche die Konstruktoren eines algebraischen Datentyps auszeichnen, was ermöglichen sie und warum?

Achtung: waren nicht auf dem Übungsblatt 1.

Verständnisfragen: Übungsblatt 2

1. Welche zusätzliche Mächtigkeit wird durch Rekursion bei algebraischen Datentypen in der Modellierung erreicht? Was läßt sich mit rekursiven Datentypen modellieren, was sich nicht durch nicht-rekursive Datentypen erreichen läßt?

Verständnisfragen: Übungsblatt 2

1. Welche zusätzliche Mächtigkeit wird durch Rekursion bei algebraischen Datentypen in der Modellierung erreicht? Was läßt sich mit rekursiven Datentypen modellieren, was sich nicht durch nicht-rekursive Datentypen erreichen läßt?

2. Was ist der Unterschied zwischen Bäumen und Graphen, in Haskell modelliert?

Verständnisfragen: Übungsblatt 2

1. Welche zusätzliche Mächtigkeit wird durch Rekursion bei algebraischen Datentypen in der Modellierung erreicht? Was läßt sich mit rekursiven Datentypen modellieren, was sich nicht durch nicht-rekursive Datentypen erreichen läßt?

2. Was ist der Unterschied zwischen Bäumen und Graphen, in Haskell modelliert?

3. Was sind die wesentlichen Gemeinsamkeiten, und was sind die wesentlichen Unterschiede zwischen algebraischen Datentypen in Haskell, und Objekten in Java?

Verständnisfragen: Übungsblatt 3

1. Was ist Polymorphie?

Verständnisfragen: Übungsblatt 3

1. Was ist Polymorphie?

2. Welche zwei Arten der Polymorphie haben wir kennengelernt, und wie unterschieden sie sich?

Verständnisfragen: Übungsblatt 3

1. Was ist Polymorphie?

2. Welche zwei Arten der Polymorphie haben wir kennengelernt, und wie unterschieden sie sich?

3. Was ist der Unterschied zwischen Polymorphie in Haskell, und Polymorphie in Java?

Verständnisfragen: Übungsblatt 4

1. Was kennzeichnet einfach rekursive Funktionen, wie wir sie in der Vorlesung kennengelernt haben, und wie sind sie durch die Funktion foldr darstellbar?

Verständnisfragen: Übungsblatt 4

1. Was kennzeichnet einfach rekursive Funktionen, wie wir sie in der Vorlesung kennengelernt haben, und wie sind sie durch die Funktion foldr darstellbar?

2. Welche anderen geläufigen Funktionen höherer Ordnung kennen wir?

Verständnisfragen: Übungsblatt 4

1. Was kennzeichnet einfach rekursive Funktionen, wie wir sie in der Vorlesung kennengelernt haben, und wie sind sie durch die Funktion foldr darstellbar?

2. Welche anderen geläufigen Funktionen höherer Ordnung kennen wir?

3. Was ist η-Kontraktion, und warum ist es zulässig?

Verständnisfragen: Übungsblatt 4

1. Was kennzeichnet einfach rekursive Funktionen, wie wir sie in der Vorlesung kennengelernt haben, und wie sind sie durch die Funktion foldr darstellbar?

2. Welche anderen geläufigen Funktionen höherer Ordnung kennen wir?

3. Was ist η-Kontraktion, und warum ist es zulässig?

4. Wann verwendet man foldr, wann foldl, und unter welchen Bedingungen ist das Ergebnis das gleiche?

Verständnisfragen: Übungsblatt 5

1. foldr ist die „kanonische einfach rekursive Funktion“ (Vorlesung). Was bedeutet das, und warum ist das so? Für welche Datentypen gilt das?

Verständnisfragen: Übungsblatt 5

1. `foldr` ist die „kanonische einfach rekursive Funktion“ (Vorlesung). Was bedeutet das, und warum ist das so? Für welche Datentypen gilt das?

2. Wann kann `foldr f a xs` auch für ein zyklisches Argument `xs` (bspw. eine zyklische Liste) terminieren?

Verständnisfragen: Übungsblatt 5

1. `foldr` ist die „kanonische einfach rekursive Funktion“ (Vorlesung). Was bedeutet das, und warum ist das so? Für welche Datentypen gilt das?

2. Wann kann `foldr f a xs` auch für ein zyklisches Argument `xs` (bspw. eine zyklische Liste) terminieren?

3. Was ist die Grundidee hinter Parserkombinatoren, und funktioniert diese Idee nur für Parser oder auch für andere Problemstellungen?

Verständnisfragen: Übungsblatt 6

1. Warum ist es hilfreich, Typen abzuleiten und nicht nur die gegebene Typisierung zu überprüfen?

Verständnisfragen: Übungsblatt 6

1. Warum ist es hilfreich, Typen abzuleiten und nicht nur die gegebene Typisierung zu überprüfen?

2. Welches sind drei charakteristische Eigenschaften von Haskells Typsystem (Hindley-Milner)?

Verständnisfragen: Übungsblatt 6

1. Warum ist es hilfreich, Typen abzuleiten und nicht nur die gegebene Typisierung zu überprüfen?

2. Welches sind drei charakteristische Eigenschaften von Haskells Typsystem (Hindley-Milner)?

3. Was ist ein Typschema, und wozu wird es im Hindley-Milner-Typsystem benötigt?

Verständnisfragen: Übungsblatt 7

1. Was ist ein abstrakter Datentyp (ADT)?

Verständnisfragen: Übungsblatt 7

1. Was ist ein abstrakter Datentyp (ADT)?

2. Was sind Unterschiede und Gemeinsamkeiten zwischen ADTs und Objekten, wie wir sie aus Sprachen wie Java kennen?

Verständnisfragen: Übungsblatt 7

1. Was ist ein abstrakter Datentyp (ADT)?

2. Was sind Unterschiede und Gemeinsamkeiten zwischen ADTs und Objekten, wie wir sie aus Sprachen wie Java kennen?

3. Wozu dienen Module in Haskell?

Verständnisfragen: Übungsblatt 8

1. Wie können wir die Typen und Operationen der Signatur eines abstrakten Datentypen grob klassifizieren, und welche Auswirkungen hat diese Klassifikation auf die zu formulierenden Eigenschaften?

Verständnisfragen: Übungsblatt 8

1. Wie können wir die Typen und Operationen der Signatur eines abstrakten Datentypen grob klassifizieren, und welche Auswirkungen hat diese Klassifikation auf die zu formulierenden Eigenschaften?

2. Warum „finden Tests Fehler“, aber „zeigen Beweise Korrektheit“, wie in der Vorlesung behauptet? Stimmt das immer?

Verständnisfragen: Übungsblatt 8

1. Wie können wir die Typen und Operationen der Signatur eines abstrakten Datentypen grob klassifizieren, und welche Auswirkungen hat diese Klassifikation auf die zu formulierenden Eigenschaften?

2. Warum „finden Tests Fehler“, aber „zeigen Beweise Korrektheit“, wie in der Vorlesung behauptet? Stimmt das immer?

3. Müssen Axiome immer ausführbar sein? Welche Axiome wären nicht ausführbar?

Verständnisfragen: Übungsblatt 9

1. Der Datentyp Stream α ist definiert als
   data Stream α = Cons α (Stream α).
   Gibt es für diesen Datentyp ein Induktionsprinzip? Ist es sinnvoll?

Verständnisfragen: Übungsblatt 9

1. Der Datentyp Stream α ist definiert als
   data Stream α = Cons α (Stream α).
   Gibt es für diesen Datentyp ein Induktionsprinzip? Ist es sinnvoll?

2. Welche nichtausführbaren Prädikate haben wir in der Vorlesung kennengelernt?

Verständnisfragen: Übungsblatt 9

1. Der Datentyp Stream α ist definiert als
   data Stream α = Cons α (Stream α).
   Gibt es für diesen Datentyp ein Induktionsprinzip? Ist es sinnvoll?

2. Welche nichtausführbaren Prädikate haben wir in der Vorlesung kennengelernt?

3. Wie kann man in einem Induktionsbeweis die Induktionsvoraussetzung stärken?

Verständnisfragen: Übungsblatt 9

1. Der Datentyp Stream α ist definiert als
   data Stream α = Cons α (Stream α).
   Gibt es für diesen Datentyp ein Induktionsprinzip? Ist es sinnvoll?

2. Welche nichtausführbaren Prädikate haben wir in der Vorlesung kennengelernt?

3. Wie kann man in einem Induktionsbeweis die Induktionsvoraussetzung stärken?

4. Gibt es einen Weihnachtsmann?

Verständnisfragen: Übungsblatt 10

1. Warum ist die Erzeugung von Zufallszahlen eine Aktion?

Verständnisfragen: Übungsblatt 10

1. Warum ist die Erzeugung von Zufallszahlen eine Aktion?

2. Warum sind Aktionen nicht explizit als Zustandsübergang modelliert, sondern implizit als abstrakter Datentyp IO?

Verständnisfragen: Übungsblatt 10

1. Warum ist die Erzeugung von Zufallszahlen eine Aktion?

2. Warum sind Aktionen nicht explizit als Zustandsübergang modelliert, sondern implizit als abstrakter Datentyp IO?

3. Was ist (außer dem Mangel an referentieller Transparenz) die entscheidende Eigenschaft, die Aktionen von reinen Funktionen unterscheidet?

Verständnisfragen: Übungsblatt 11

1. Warum sind endrekursive Funktionen im allgemeinen schneller als nicht-endrekursive Funktionen? Unter welchen Voraussetzungen kann ich eine Funktion in endrekursive Form überführen?

Verständnisfragen: Übungsblatt 11

1. Warum sind endrekursive Funktionen im allgemeinen schneller als nicht-endrekursive Funktionen? Unter welchen Voraussetzungen kann ich eine Funktion in endrekursive Form überführen?

2. Ist eine in allen Argumenten als strikt erkannte und übersetzte Funktion immer schneller in der Ausführung als dieselbe als nicht-strikt übersetzte Funktion?

Verständnisfragen: Übungsblatt 11

1. Warum sind endrekursive Funktionen im allgemeinen schneller als nicht-endrekursive Funktionen? Unter welchen Voraussetzungen kann ich eine Funktion in endrekursive Form überführen?

2. Ist eine in allen Argumenten als strikt erkannte und übersetzte Funktion immer schneller in der Ausführung als dieselbe als nicht-strikt übersetzte Funktion?

3. Warum kann ich die Funktion seq nicht in Haskell definieren?

Warum funktionale Programmierung lernen?

- Denken in Algorithmen, nicht in Programmiersprachen
- Abstraktion: Konzentration auf das Wesentliche
- Wesentliche Elemente moderner Programmierung:
  - Datenabstraktion und Funktionale Abstraktion
  - Modularisierung
  - Typisierung und Spezifikation
- Blick über den Tellerrand — Blick in die Zukunft
- Studium $\neq$ Programmierkurs — was kommt in 10 Jahren?

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, F#)...
- Programmierung nur kleiner Teil der SW-Entwicklung
- Nicht verbreitet — funktionale Programmierer zu teuer
- Konservatives Management
  - "Nobody ever got fired for buying IBM"

Funktionale Programmierung in der Industrie

- Erlang
  - schwach typisiert, nebenläufig, strikt
  - Fa. Ericsson — Telekom-Anwendungen
- `FL`
  - ML-artige Sprache
  - Chip-Verifikation der Fa. Intel
- Python (und andere Skriptsprachen):
  - Listen, Funktionen höherer Ordnung (map, fold), anonyme Funktionen, Listenkomprehension
- Scala:
  - Zum Beispiel Twitter, Foursquare, ...

Perspektiven funktionaler Programmierung

- Forschung:
  - Ausdrucksstärkere Typsysteme
  - für effiziente Implementierungen
  - und eingebaute Korrektheit (Typ als Spezifikation)
  - Parallelität?
- Anwendungen:
  - Eingebettete domänenspezifische Sprachen
  - Zustandsfreie Berechnungen (MapReduce, Hadoop)
  - Big Data and Cloud Computing

If you liked this course, you might also like . . .

- Die Veranstaltung Reaktive Programmierung (Sommersemester 2015)
  - 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 15/16 — meldet Euch bei Berthold Hoffmann (oder bei mir)!

Tschüß!