Praktische Informatik 3: Funktionale Programmierung
Vorlesung 1 vom 18.10.2016: Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Personal

- Vorlesung:
  Christoph Lüth <cxl@informatik.uni-bremen.de>
  www.informatik.uni-bremen.de/~cxl/ (MZH 4186, Tel. 59830)
- Tutoren:
  Tobias Brandt <to_br@uni-bremen.de>
  Tristan Bruns <tbruns@informatik.uni-bremen.de>
  Johannes Ganser <ganser@uni-bremen.de>
  Alexander Kurth <kurth1@uni-bremen.de>
  Berthold Hoffmann <hof@informatik.uni-bremen.de>
- "Fragestunde": Berthold Hoffmann n.V. (Cartesium 1.54, Tel. 64 222)
- Webseite: www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws16

Termine

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

Mi	08 – 10	GW1 A0160	Berthold Hoffmann
	10 – 12	GW1 A0160	Johannes Ganser
	12 – 14	MZH 1110	Johannes Ganser
	14 – 16	GW1 B2070	Alexander Kurth
Do	08 – 10	MZH 1110	Tobias Brandt
	10 – 12	GW1 B2130	Tristan Bruns

- Anmeldung zu den Übungsgruppen über stud.ip
  - Duale Studierende sollten im Tutorium Do 10– 12 registriert sein.

Übungsbetrieb

- Ausgabe der Übungsblätter über die Webseite Dienstag morgen
- Besprechung der Übungsblätter in den Tutorien
- Bearbeitungszeit: eine Woche
- Abgabe: elektronisch bis Freitag nächste Woche 12:00
- Zehn Übungsblätter (voraussichtlich) plus 0. Übungsblatt
- Übungsgruppen: max. drei Teilnehmer
- Bewertung: Quellcode 50%, Tests 25%, Dokumentation 25%
  - Nicht übersetzender Quellcode: 0 Punkte

Scheinkriterien

- Geplant: $n = 10$ Übungsblätter
- Mind. 50% in allen und in den ersten $n/2$ Übungsblättern
- 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
- Alternative: Modulprüfung (mündlich)

Spielregeln

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

Fahrplan

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

Warum funktionale Programmierung lernen?

- Funktionale Programmierung macht aus Programmierern Informatiker
- Blick über den Tellerrand — was kommt in 10 Jahren?
- Herausforderungen der Zukunft
- Enthält die wesentlichen Elemente moderner Programmierung

Zunkunft eingebaut

Funktionale Programmierung ist bereit für die Herausforderungen der Zukunft:

- Nebenläufige Systeme (Mehrkernarchitekturen)
- Massiv verteilte Systeme („Internet der Dinge“)
- Große Datenmengen („Big Data“)

The Future is Bright — The Future is Functional

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

Warum Haskell?

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

Geschichtliches: Die Anfänge

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


Moses Schönfinkel

Haskell B. Curry

Alonzo Church

John McCarthy

John Backus

Robin Milner

Mike Gordon

Geschichtliches: Die Gegenwart

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

Programme als Funktionen

- Programme als Funktionen:

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

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

Auswertung als Ausführungsbegriff

- Programme werden durch Gleichungen definiert:

$$f(x) = E$$

- Auswertung durch Anwenden der Gleichungen:
  - Suchen nach Vorkommen von f, e.g. $f(t)$
  - $f(t)$ wird durch $E\left[\begin{array}{c} t \\ x \end{array}\right]$ ersetzt
- Auswertung kann divergieren!

Ausdrücke und Werte

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

Typisierung

- Typen unterscheiden Arten von Ausdrücken und Werten:

```
repeat n s = ...
```

n Zahl
s Zeichenkette

- Wozu Typen?
  - Frühzeitiges Aufdecken "offensichtlicher" Fehler
  - Erhöhte Programmsicherheit
  - Hilfestellung bei Änderungen

Slogan

"Well-typed programs can't go wrong."

— *Robin Milner*

Signaturen

- Jede Funktion hat eine Signatur

```
fac    :: Int→ Int

repeat :: Int→ String→ String
```

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

Übersicht: Typen in Haskell

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

- Später mehr. Viel mehr.

Das Rechnen mit Zahlen

Beschränkte Genauigkeit, konstanter Aufwand ⟷ beliebige Genauigkeit, wachsender Aufwand

Haskell bietet die Auswahl:

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

Ganze Zahlen: `Int` und `Integer`

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

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

Es gilt: `(div x y)*y +mod x y == x`

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

Fließkommazahlen: Double

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

```
round (fromInt 10) :: Int
```

- Rundungsfehler!

Alphanumerische Basisdatentypen: Char

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

```
ord :: Char → Int
chr :: Int → Char

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

- Zeichenketten: String

Zusammenfassung

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

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 2 vom 25.10.2016: Funktionen und Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II und Effizenzaspekte
- 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	GW1 A0160	Berthold Hoffmann	16 (50)	4
	10 – 12	GW1 A0160	Johannes Ganser	43 (50)	9
	12 – 14	MZH 1110	Johannes Ganser	35 (35)	9
	14 – 16	GW1 B2070	Alexander Kurth	25 (25)	7
Do	08 – 10	MZH 1110	Tobias Brandt	33 (35)	10
	10 – 12	GW1 B2130	Tristan Bruns	25 (25)	11

- Insgesamt 50 Gruppen (ca. 8 pro Tutorium)
- Wenn möglich, frühes Mittwochstutorium belegen.

Definition von Funktionen

Definition von Funktionen

- Zwei wesentliche Konstrukte:
  - Fallunterscheidung
  - Rekursion

Satz

Fallunterscheidung und Rekursion auf natürlichen Zahlen sind Turing-mächtig.

- Funktionen müssen partiell sein können.

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: Funktionsdefinition

Generelle Form:

- Signatur:

```
max :: Int→ Int→ Int
```

- Definition:

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

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

Haskell-Syntax I: Die Abseitsregel

Funktionsdefinition:

$$f\ x_1\ x_2 \ldots x_n = E$$

- 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 (heben das Abseits nicht auf).

Haskell-Syntax II: Kommentare

- Pro Zeile: Ab −− bis Ende der Zeile

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

- Über mehrere Zeilen: Anfang {−, Ende −}

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

  - Kann geschachtelt werden.

Haskell-Syntax III: Bedingte Definitionen

- Statt verschachtelter Fallunterscheidungen ...

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

... bedingte Gleichungen:

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

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

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


Haskell-Syntax IV: Lokale Definitionen

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

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

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

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

Bedeutung von 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, $\rightarrow_P$ die dadurch definierte Reduktion, und $[\![P]\!]$ das Denotat. Dann gilt für alle Ausdrücke t und Werte v

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

Auswertungsstrategien

```
inc :: Int → Int
inc x = x+1
```

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

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


Auswertungsstrategien

```
addx :: String → String
addx s = 'x': s
```

```
double :: String→ String
double s = s++ s
```

- Reduktion von addx (double (addx "y"))
- Von außen nach innen (outermost-first):

```
addx (double (addx "y"))  → 'x':double (addx "y")
                          → 'x':(addx "y" ++ addx "y")
                          → 'x':(('x': "y") ++ addx "y")
                          → 'x':(('x': "y") ++ ('x': "y"))
                          → "xxyxy"
```

- Von innen nach außen (innermost-first):

```
addx (double (addx "y"))  → addx (double ('x': "y"))
                          → addx (double ("xy"))
                          → addx ("xy" ++ "xy")
                          → addx "xyxy"
                          → 'x': "xyxy" → "xxyxy"
```

Konfluenz

- 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 $s \xrightarrow{*} u \xleftarrow{*} t$.

- Wenn wir von Laufzeitfehlern abstrahieren, gilt:

Theorem (Konfluenz)

Funktionale Programme sind für jede Auswertungsstrategie konfluent.

Termination und Normalform

Definition (Termination)

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

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

Theorem (Normalform)

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.

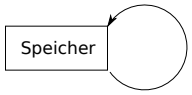
Datentypen

Datentypen als Modellierungskonstrukt

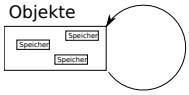
Programme manipulieren ein Modell (der Umwelt)

- 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
```

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


Fallunterscheidung in der Funktionsdefinition

- Abkürzende Schreibweisen (syntaktischer Zucker):

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

- Damit:

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


Der einfachste Aufzählungstyp

- Einfachste Aufzählung: Wahrheitswerte

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

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

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

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

Wahrheitswerte: Bool

- Vordefiniert als

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

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

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

Produkte

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

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

- Funktionsdefinition:
  - Konstruktorargumente sind gebundene Variablen

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

  - 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 25 Oct 2016
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 _ _ 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

- 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
```


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
```


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:

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

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

Diese Eigenschaften machen Fallunterscheidung möglich.

Rekursion? ⟶ 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 01.11.2016: Algebraische Datentypen

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Algebraische Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II und Effizenzaspekte
- 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: Labyrinth

Algebraische Datentypen

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

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

Heute: Rekursion

Der Allgemeine Fall: Algebraische Datentypen

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

Drei Eigenschaften eines algebraischen Datentypen

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

Diese Eigenschaften machen Fallunterscheidung wohldefiniert.

Algebraische Datentypen: Nomenklatur

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

- C_i sind Konstruktoren
  - Immer vordefiniert
- Selektoren sind Funktionen $\mathsf{sel}_{i,j}$:
  $\mathsf{sel}_{i,j} :: \mathsf{T} \to \mathsf{t}_{i,k_i}$
  $\mathsf{sel}_{i,j}\ (\mathsf{C}_i\ \mathsf{t}_{i,1} \ldots\ \mathsf{t}_{i,k_i}) = \mathsf{t}_{i,j}$
  - Linksinvers zu Konstruktor C_i, partiell
  - Können vordefiniert werden (erweiterte Syntax der **data** Deklaration)
- Diskriminatoren sind Funktionen dis_i:
  $\mathsf{dis}_i :: \mathsf{T} \to \mathsf{Bool}$
  $\mathsf{dis}_i\ (\mathsf{C}_i \ldots) = \mathsf{True}$
  $\mathsf{dis}_i\ \_ = \mathsf{False}$
  - Definitionsbereichsbereich des Selektors sel_i, nie vordefiniert

Rekursive Datentypen

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

Uncle Bob's Auld Time Grocery Shoppe Revisited

- Das Lager für Bob's Shoppe:
  - ist entweder leer,
  - oder es enthält einen Artikel und Menge, und 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 (35l Milch + 20l Milch = 55l Milch)
- Dazu Hilfsfunktion:

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

- 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 imperativen Sprachen

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

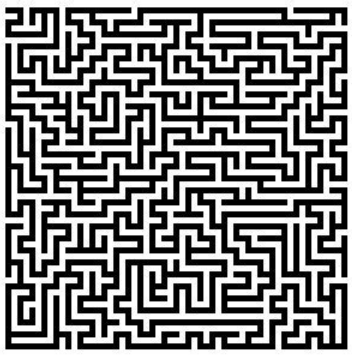
```
typedef struct list_t {
    void          *elem;
    struct list_t *next;
    } *list;
```

- 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

Fallbeispiel: Zyklische Datenstrukturen



Quelle: docs.gimp.org

Modellierung eines Labyrinths

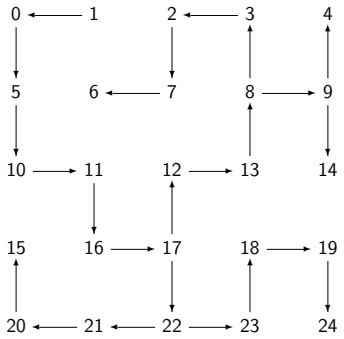
- Ein gerichtetes Labyrinth ist entweder
  - eine Sackgasse,
  - ein Weg, oder
  - eine Abzweigung in zwei Richtungen.

```
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))
```

- Wie mit Zyklen umgehen?
- An jedem Knoten prüfen ob schon im Pfad enthalten

Traversion mit Zyklen

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

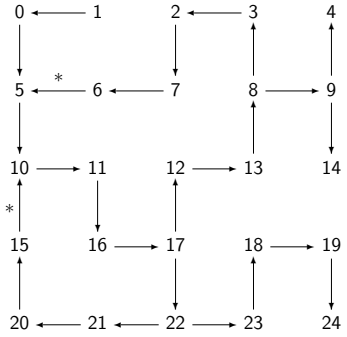
```
contains :: Id→ Path→ Bool
```

```
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 ⟶ 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
```

- 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

⟶ 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 08.11.2016: Typvariablen und Polymorphie

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

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

Organisatorisches

- Abgabe der Übungsblätter: Freitag 12 Uhr mittags
- Mittwoch, 09.11.16: Tag der Lehre
  - Tutorium Mi 14-16 (Alexander) verlegt auf Do 14-16 Cartesium 0.01
  - Alle anderen Tutorien finden statt.
- Hinweis: Quellcode der Vorlesung auf der Webseite verfügbar.

Inhalt

- Letzte Vorlesungen: algebraische Datentypen
- Diese Vorlesung:
  - Abstraktion über Typen: Typvariablen und Polymorphie
  - Arten der Polymorphie:
    - Parametrische Polymorphie
    - Ad-hoc Polymorphie

Ähnliche Datentypen der letzten Vorlesung

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

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

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

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

- ein konstanter Konstruktor
- ein linear rekursiver Konstruktor

Ähnliche Funktionen der letzten Vorlesung

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

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

```
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

Parametrische Polymorphie: Typvariablen

- Typvariablen abstrahieren über Typen

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

- α ist eine Typvariable
- α kann mit Int 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	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

Beispiel: Der Shop (refaktoriert)

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

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

- Damit:

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

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

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

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

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


Tupel

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

```
data Pair α β = Pair 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

Vordefinierte Datentypen: Tupel und Listen

- Eingebauter syntaktischer Zucker
- Tupel sind das kartesische Produkt

```
data (α, β) = (α, β)
```

  - (a, b) = alle Kombinationen von Werten aus a und b
  - Auch n-Tupel: (a,b,c) etc. (für $n \leq 9$)
- 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 = Nothing | Just α
```

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

```
fromJust    :: Maybe α → α      -- partiell
fromMaybe   :: α→ Maybe α→ α
listToMaybe :: [α]→ Maybe α     -- totale Variante von head
maybeToList :: Maybe α→ [α]     -- rechtsinvers zu listToMaybe
```

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]
```

- 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
```

- Damit: das Labyrinth als variadischer Baum

```
type Lab = VTree Id
```


Ad-Hoc Polymorphie

Ad-Hoc Polymorphie und Overloading

Definition (Überladung)

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

- Beispiel:
  - Gleichheit: (==) :: α → α → Bool
  - Vergleich: (<) :: α → α → Bool
  - Anzeige: show :: α → 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 :: (Ord α, Show α)⇒ [α] → String
showsorted x = show (qsort x)
```

Hierarchien von Typklassen

- Typklassen können andere voraussetzen:

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

- Default-Definition von $(\leq)$
- Kann bei Instantiierung überschrieben werden

Typherleitung

Typen in Haskell (The Story So Far)

- Primitive Basisdatentypen: Bool, Double
- Funktionstypen Double→ Int→ Int, [Char] → Double
- Typkonstruktoren: [], (...), Foo
- Typvariablen
  fst :: $(\alpha, \beta) \to \alpha$
  length :: $[\alpha] \to$ Int
  $(+)$ :: $[\alpha] \to [\alpha] \to [\alpha]$
- Typklassen : elem :: Eq a⇒ a → [a] → Bool
  max :: Ord a⇒ a → a → 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

```
f m xs = m + length xs
         [α]→ Int
              [α]
         Int
    Int
         Int

f :: Int→ [α]→ Int
```


Typinferenz

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

```
f m xs = m + length xs
         α   [β]→ Int  γ
                  [β]  γ ↦ β
             Int
    Int→ Int→ Int
Int                    α ↦ Int
    Int→ Int
             Int

f :: Int→ [α]→ Int
```


Typinferenz

- Unifikation kann mehrere Substituitionen beinhalten:

```
(x, 3)  :  ('f', y)  :  []
 α  Int    Char  β      [γ]
(α, Int)  (Char, β)
          (Char, β)   [(Char, β)]   γ ↦ (Char, β)
                [(Char, β)]         β ↦ Int,
                                    α ↦ Char
     [(Char, Int)]
```

- Allgemeinster Typ muss nicht existieren (Typfehler!)

Abschließende Bemerkungen

Polymorphie: the missing link

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

- Kann Entscheidbarkeit der Typherleitung gefährden

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

- Ad-Hoc Polymorphie in Java:
  - Interface und abstrakte Klassen
  - Flexibler in Java: beliebige Parameter etc.
- Dynamische Typisierung: Ruby, Python
  - "Duck typing": strukturell gleiche Typen sind gleich

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], Option Maybe α und Tupel (a,b)
- Nächste Woche: Abstraktion über Funktionen

$\longrightarrow$ Funktionen höherer Ordnung

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 5 vom 15.11.2016: Funktionen Höherer Ordnung I

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

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

Inhalt

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

Funktionen als Werte

Funktionen Höherer Ordnung

Slogan

"Functions are first-class citizens."

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

Ähnliche Datentypen der letzten Vorlesung

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

data Einkaufswagen = LeererWagen
                   | Einkauf Artikel Menge Einkaufswagen

data Path
```

Gelöst durch Polymorphie

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

- ein konstanter Konstruktor
- ein linear rekursiver Konstruktor

Ähnliche Funktionen der letzten Vorlesung

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

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

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

Gemeinsamkeiten:

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

Ein einheitlicher Rahmen

- Zwei ähnliche Funktionen:

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

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

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

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

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

- Funktion f als Argument
- Was hätte map für einen Typ?

Funktionen als Werte: Funktionstypen

- Was hätte map für einen Typ?

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

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

```
map :: (α → β) → [α] → [β]
```


Map und Filter

Funktionen als Argumente: map

- map wendet Funktion auf alle Elemente an
- Signatur:

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

- Funktionsausdrücke werden symbolisch reduziert
  - Keine Änderung

Funktionen als Argumente: filter

- Elemente filtern: `filter`
- Signatur:

```
filter :: (α → Bool) → [α] → [α]
```

- Definition

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

- Beispiel:

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


Beispiel filter: Primzahlen

- Sieb des Erathostenes
  - Für jede gefundene Primzahl p alle Vielfachen herausieben
  - Dazu: `filter (λq→ mod q 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..]      -- Wichtig: bei 2 anfangen!
```

- Die ersten n Primzahlen:

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


Funktionen Höherer Ordnung

Funktionen als Argumente: Funktionskomposition

- Funktionskomposition (mathematisch)

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

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

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

  - Nicht vordefiniert!

η-Kontraktion

- Vertauschen der Argumente (vordefiniert):

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

- Damit Funktionskomposition vorwärts:

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

- Da fehlt doch was?! Nein:

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

- Warum?

η-Äquivalenz und η-Kontraktion

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

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

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

- Spezialfall Funktionsdefinition (punktfreie Notation)

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

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

Partielle Applikation

- Funktionskonstruktor rechtsassoziativ:

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

  - Inbesondere: $(\alpha \to \beta) \to \gamma \neq \alpha \to (\beta \to \gamma)$
- Funktionsanwendung ist linksassoziativ:

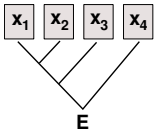
$$\mathsf{f\ a\ b} \equiv (\mathsf{f\ a})\ \mathsf{b}$$

  - Inbesondere: $\mathsf{f}\ (\mathsf{a\ b}) \neq (\mathsf{f\ a})\ \mathsf{b}$
- Partielle Anwendung von Funktionen:
  - Für f :: $\alpha \to \beta \to \gamma$, x :: α ist f x :: $\beta \to \gamma$
- Beispiele:
  - map toLower :: String → String
  - (3 ==) :: Int → Bool
  - concat ∘ map (replicate 2) :: String → String

Strukturelle Rekursion

Strukturelle Rekursion

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



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

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


Strukturelle Rekursion

- Allgemeines Muster:

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

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

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

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

Strukturelle Rekursion durch `foldr`

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

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

- Definition

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


Beispiele: `foldr`

- Summieren von Listenelementen.

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

- Flachklopfen von Listen.

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

- Länge einer Liste

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


Beispiele: `foldr`

- Konjunktion einer Liste

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

- Konjunktion von Prädikaten

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

Der Shoppe, revisited.

- 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 $O(n^2)$!

Iteration mit foldl

- foldr faltet von rechts:
  foldr ⊗ [x1, . . ., xn] e = x1 ⊗ x2 (x2 ⊗ (. . . (xn ⊗ e)))
- Warum nicht andersherum?
  foldl ⊗ [x1, . . ., xn] e = (((e ⊗ x1) ⊗ x2). .) ⊗ xn
- Definition von foldl:

```
foldl :: (α → β → α) → α → [β] → α
foldl f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs
```

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

Beispiel: rev revisited

- Listenumkehr durch falten von links:

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

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

foldr vs. foldl

- f = foldr ⊗ e entspricht

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

  - Kann nicht-strikt in xs sein, z.B. and, or
  - Konsumiert nicht immer die ganze Liste
  - Auch für zyklische Listen anwendbar
- f = foldl ⊗ e entspricht

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

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

Wann ist foldl = foldr?

Definition (Monoid)

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

$$A \otimes x = x \quad \text{(Neutrales Element links)}$$
$$x \otimes A = x \quad \text{(Neutrales Element rechts)}$$
$$(x \otimes y) \otimes z = x \otimes (y \otimes z) \quad \text{(Assoziativität)}$$

Theorem

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

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

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

Übersicht: vordefinierte Funktionen auf Listen II

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

- Mehr: siehe `Data.List`

Funktionen Höherer Ordnung in anderen Sprachen

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 map1(void *f(void *x), list l);
```

- Keine direkte Syntax (e.g. namenlose Funktionen)
- Typsystem zu schwach (keine Polymorphie)
- Benutzung: `qsort` (C-Standard 7.20.5.2)

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```


Funktionen Höherer Ordnung: C

- Implementierung von map
- Rekursiv, erzeugt neue Liste:

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

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

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


Zusammenfassung

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

Praktische Informatik 3: Funktionale Programmierung

Vorlesung 6 vom 22.11.2016: Funktionen Höherer Ordnung II und Effizienzaspekte

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

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

Heute

- Mehr über map und fold
- map und fold sind nicht nur für Listen
- Effizient funktional programmieren

map als strukturerhaltende Abbildung

- Der Datentyp $()$ ist terminal: es gibt für jeden Datentyp α genau eine total Abbildung $\alpha \to ()$
- Struktur (Shape) eines Datentyps T α ist T $()$
  - Gegeben durch kanonische Funktion shape :: T $\alpha \to$ T $()$, die α durch $()$ ersetzt
  - Für Listen: $[()] \cong$ Nat

map ist die kanonische strukturerhaltende Abbildung

- Für map gelten folgende Aussagen:

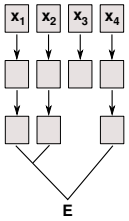
$$\text{map id} = \text{id}$$

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

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

map und filter als Berechnungsmuster

- map, filter, fold als Berechnungsmuster:
  1. Anwenden einer Funktion auf jedes Element der Liste
  2. möglicherweise Filtern bestimmter Elemente
  3. Kombination der Ergebnisse zu Endergebnis E
- Gut parallelisierbar, skaliert daher als Berechnungsmuster für große Datenmengen (*big data*)



- Besondere Notation: Listenkomprehension
  [f x | x← as, g x] ≡ map f (filter g as)
  - Beispiel:

```
digits str = [ord x −ord '0' | x← str, isDigit x]
```

  - Auch mit mehreren Generatoren:

```
idx = [ a: show i | a← ['a'.. 'z'], i← [0.. 9]]
```


foldr ist kanonisch

foldr ist die kanonische strukturell rekursive Funktion.

- Alle strukturell rekursiven Funktionen sind als Instanz von foldr darstellbar
- Insbesondere auch map und filter (siehe Übungsblatt)
- Es gilt: foldr (:) [] = id
- Jeder algebraischer Datentyp hat ein foldr

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
- Anmerkung: Typklasse Foldable schränkt Signatur von foldr ein

```
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: Fallunterscheidung:

```
data Bool = False | True

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

- Maybe α: Auswertung

```
data Maybe α = Nothing | Just α

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

  - Als maybe vordefiniert

fold für bekannte Datentypen

- Tupel: 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

- Instanz von fold:

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

  - Instanz von map, kein (offensichtliches) Filter

Funktionen mit foldT und mapT

- Höhe des Baumes berechnen:

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

- Inorder-Traversion der Knoten:

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


Kanonische Eigenschaften von foldT und mapT

- Auch hier gilt:

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

$$\text{mapT id} = \text{id}$$

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

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

- Mit shape :: Tree α→ 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:

```
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
```

- Problem:
  - foldT terminiert nicht für zyklische Strukturen
  - Auch nicht, wenn add prüft ob a schon enthalten ist
  - Pfade werden vom Ende konstruiert

Effizienzaspekte

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 even (x-2) else x == 0
```

- Übersetzt nach C:

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

- Äquivalente Formulierung:

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

- Iterative Variante mit Schleife:

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


Beispiel: Fakultät

- Fakultät nicht endrekursiv:

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

- Fakultät endrekursiv:

```
fac2 :: Integer→ Integer
fac2 n     = fac' n 1 where
  fac' :: Integer→ Integer→ Integer
  fac' n acc = if n == 0 then acc
               else fac' (n-1) (n*acc)
```

- fac1 verbraucht Stack, fac2 nicht.
- Ist nicht merklich schneller?!

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!
- Eine Funktion hat ein Speicherleck, wenn Speicher unnötig lange im Zugriff bleibt.
  - "Echte" Speicherlecks wie in C/C++ nicht möglich.
- Beispiel: fac2
  - Zwischenergebnisse werden nicht auswertet.
  - Insbesondere ärgerlich bei nicht-terminierenden Funktionen.

Striktheit

- Strikte Argumente erlauben Auswertung vor Aufruf
  - Dadurch konstanter Platz bei Endrekursion.
- Erzwungene Striktheit: seq :: $\alpha \to \beta \to \beta$

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

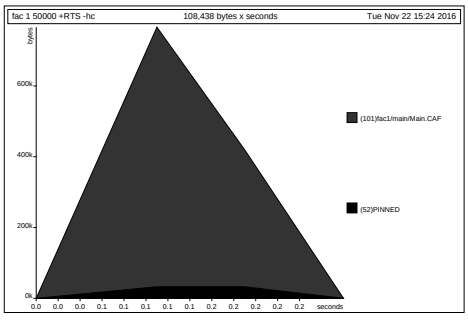
  a `seq` b = b
  - seq vordefiniert (nicht in Haskell definierbar)
  - ($!) :: (a→ b)→ a→ b strikte Funktionsanwendung

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

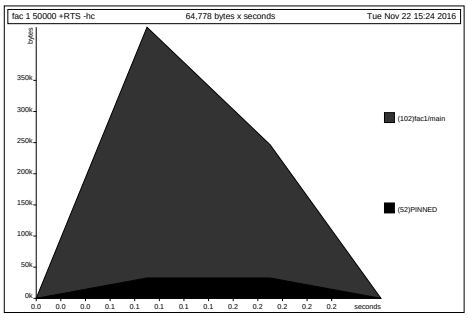
  - ghc macht Striktheitsanalyse
- Fakultät in konstantem Platzaufwand

```
fac3 :: Integer→ Integer
fac3 n = fac' n 1 where
  fac' n acc = seq acc $ if n == 0 then acc
                         else fac' (n-1) (n*acc)
```

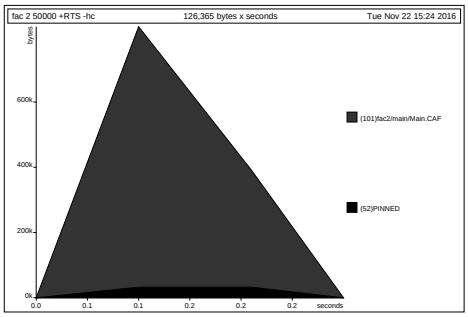

Speicherprofil: fac1 50000, nicht optimiert



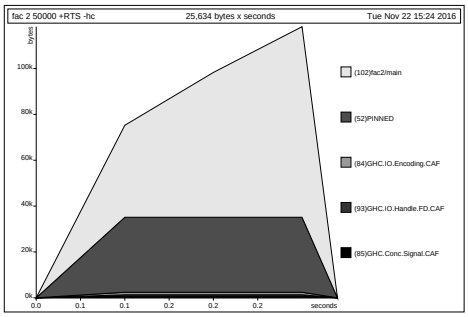
Speicherprofil: fac1 50000, optimiert



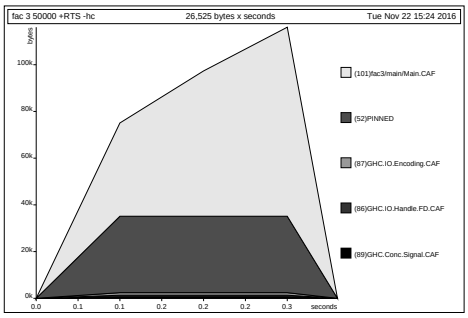
Speicherprofil: fac2 50000, nicht optimiert



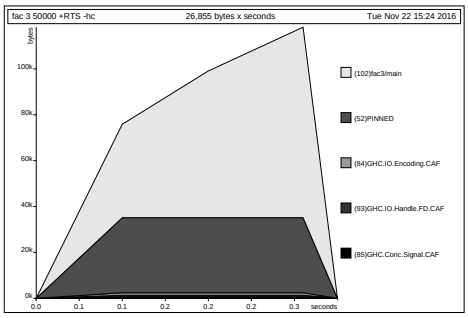
Speicherprofil: fac2 50000, optimiert



Speicherprofil: fac3 50000, nicht optimiert



Speicherprofil: fac3 50000, optimiert



Fazit Speicherprofile

- Endrekursion nur bei strikten Funktionen schneller
- Optimierung des *ghc*
  - Meist ausreichend für Striktheitsanalyse
  - Aber nicht für Endrekursion
- Deshalb:
  - Manuelle Überführung in Endrekursion sinnvoll
  - Compiler-Optimierung für Striktheit nutzen

Überführung in Endrekursion

- Voraussetzung 1: Funktion ist linear rekursiv
- Gegeben Funktion

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

  - Mit $K : S \to S$, $\otimes : T \to T \to T$, $E : S \to T$, $H : S \to T$.
- Voraussetzung 2: $\otimes$ assoziativ, $e : T$ neutrales Element
- Dann ist endrekursive Form:

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

Beispiel

- Länge einer Liste (nicht-endrekursiv)

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

- Zuordnung der Variablen:

$$
\begin{array}{llll}
K(x) & \mapsto & \text{tail } x & B(x) \mapsto \text{null } x \\
E(x) & \mapsto & 1 & H(x) \mapsto 0 \\
x \otimes y & \mapsto & x + y & e \mapsto 0
\end{array}
$$

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

Beispiel

- Damit endrekursive Variante:

```
length :: [α]→ 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 $(\otimes, e)$: $\otimes$ assoziativ, e neutrales Element.
  - Zusätzlicher Parameter akkumuliert Resultat.

Weiteres Beispiel: foldr vs. foldl

- foldr ist nicht endrekursiv:

```
foldr :: (α → β → β) → β → [α] → β
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

- foldl ist endrekursiv:

```
foldl :: (α → β → α) → α → [β] → α
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- foldl' ist strikt und endrekursiv:

```
foldl' :: (α→ β→ α)→ α→ [β]→ α
foldl' f a []     = a
foldl' f a (x:xs) = let a0= f a x in a0 `seq` foldl' f a0 xs
```

- Für Monoid $(\otimes, e)$ gilt: foldr ⊗ e l = foldl (flip ⊗) e l

Zusammenfassung

- map und fold sind kanonische Funktionen höherer Ordnung, und für alle Datentypen definierbar
- map, filter, fold sind ein nützliches, skalierbares und allgemeines Berechnungsmuster
- Effizient funktional programmieren:
  - Endrekursion: while für Haskell
  - Mit Striktheit und Endrekursion Speicherlecks vermeiden.
  - Für Striktheit Compileroptimierung nutzen
- Nächste Woche: Funktionale Programmierung im Großen — Abstrakte Datentypen

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 7 vom 29.11.2016: Abstrakte Datentypen

Christoph Lüth

Universität Bremen

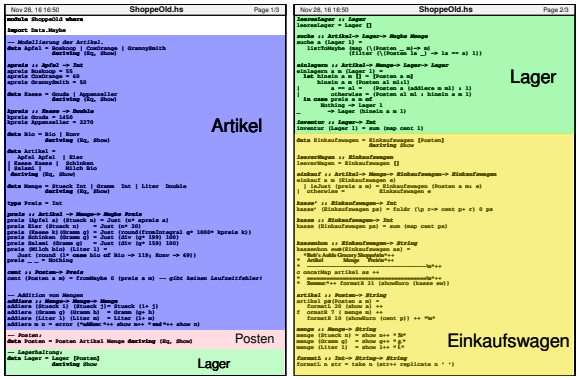
Wintersemester 2016/17

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

Inhalt

Refakturierung im Einkaufsparadies



Warum Modularisierung?

- Übersichtlichkeit der Module

  Lesbarkeit
- Getrennte Übersetzung

  technische Handhabbarkeit
- Verkapselung

  konzeptionelle Handhabbarkeit

Abstrakte Datentypen

Definition (Abstrakter Datentyp)

Ein abstrakter Datentyp (ADT) besteht aus einem (oder mehreren) Typen und Operationen darauf, mit folgenden Eigenschaften:

- Werte des Typen können nur über die bereitgestellten Operationen erzeugt werden:
- Eigenschaften von Werten des Typen werden nur über die bereitgestellten Operationen beobachtet;
- Einhaltung von Invarianten über dem Typ kann garantiert werden.

Implementation von ADTs in einer Programmiersprache:

- benötigt Möglichkeit der Kapselung (Einschränkung der Sichtbarkeit)
- bspw. durch Module oder Objekte

ADTs vs. algebraische Datentypen

- Algebraische Datentypen
  - Frei erzeugt
  - Keine Einschränkungen
  - Insbesondere keine Gleichheiten ($[] \neq x:xs$, $x:ls \neq y:ls$ etc.)
- ADTs:
  - Einschränkungen und Invarianten möglich
  - Gleichheiten möglich

ADTs in Haskell: Module

- Einschränkung der Sichtbarkeit durch Verkapselung
- Modul: Kleinste verkapselbare Einheit
- Ein Modul umfaßt:
  - Definitionen von Typen, Funktionen, Klassen
  - Deklaration der nach außen sichtbaren Definitionen
- Gleichzeitig: Modul $\triangleq$ Übersetzungseinheit (getrennte Übersetzung)

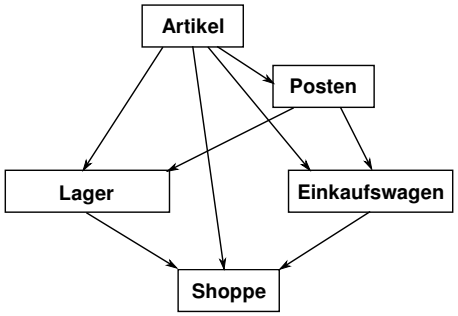
Module: Syntax

- 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

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

apreis :: Apfel → Int

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

kpreis :: Kaese → Double
```


Refakturierung im Einkaufsparadies II: Posten

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

- Implementiert ADT Posten:

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

  - Konstruktor wird nicht exportiert
- Garantierte Invariante:
  - Posten hat immer die korrekte Menge zu Artikel

```
posten a m =
  case preis a m of
    Just _ → Just (Posten a m)
    Nothing → Nothing
```


Refakturierung im Einkaufsparadies III: Lager

```
module Lager(
  Lager,
  leeresLager,
  einlagern,
  suche,
  inventur
  ) where

import Artikel
import Posten
```

- Implementiert ADT Lager
- Signatur der exportierten Funktionen:

```
leeresLager :: Lager

einlagern :: Artikel→ Menge→ Lager→ Lager

suche :: Artikel→ Lager→ Maybe Menge

inventur :: Lager→ Int
```

- Garantierte Invariante:
  - Lager enthält keine doppelten Artikel

Refakturierung im Einkaufsparadies IV: Einkaufswagen

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

- Implementiert ADT Einkaufswagen

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

- Garantierte Invariante:
  - Korrekte Menge zu Artikel im Einkaufswagen

```
einkauf :: Artikel→ Menge
                  → Einkaufswagen
                  → Einkaufswagen
einkauf a m (Einkaufswagen e) =
  case posten a m of
    Just p → Einkaufswagen (p: e)
    Nothing→ Einkaufswagen e
```

- Nutzt dazu ADT Posten

Benutzung von ADTs

- Operationen und Typen müssen importiert werden
- Möglichkeiten des Imports:
  - Alles importieren
  - Nur bestimmte Operationen und Typen importieren
  - Bestimmte Typen und Operationen nicht importieren

Importe in Haskell

- Syntax:

```
import [qualified] M [as N] [hiding][(Bezeichner)]
```

- *Bezeichner* geben an, was importiert werden soll:
  - Ohne Bezeichner wird alles importiert
  - Mit **hiding** werden Bezeichner nicht importiert
- Für jeden exportierten Bezeichner f aus M wird importiert
  - f und qualifizierter Bezeichner M.f
  - **qualified**: nur qualifizierter Bezeichner M.f
  - Umbenennung bei Import mit as (dann N.f)
  - Klasseninstanzen und Typsynonyme werden immer importiert
- Alle Importe stehen immer am Anfang des Moduls

Beispiel

```
module M(a,b) where...
```

Import(e)	Bekannte Bezeichner
import M	a, b, M.a, M.b
import M()	*(nothing)*
import M(a)	a, M.a
import qualified M	M.a, M.b
import qualified M()	*(nothing)*
import qualified M(a)	M.a
import M **hiding** ()	a, b, M.a, M.b
import M **hiding** (a)	b, M.b
import qualified M **hiding** ()	M.a, M.b
import qualified M **hiding** (a)	M.b
import M **as** B	a, b, B.a, B.b
import M **as** B(a)	a, B.a
import qualified M **as** B	B.a, B.b

Quelle: Haskell98-Report, Sect. 5.3.4

Ein typisches Beispiel

- Modul implementiert Funktion, die auch importiert wird
- Umbenennung nicht immer praktisch
- Qualifizierter Import führt zu langen Bezeichnern
- Einkaufswagen implementiert Funktionen `artikel` und `menge`, die auch aus Posten importiert werden:

```
import Posten hiding (artikel, menge)
import qualified Posten as P(artikel, menge)
```

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


Schnittstelle vs. Implementation

- Gleiche Schnittstelle kann unterschiedliche Implementationen haben
- Beispiel: (endliche) Abbildungen

Endliche Abbildungen

- Viel gebraucht, oft in Abwandlungen (Hashtables, Sets, Arrays)
- Abstrakter Datentyp für endliche Abbildungen:
  - Datentyp

    ```
    data Map α β
    ```

  - Leere Abbildung:

    ```
    empty :: Map α β
    ```

  - Abbildung auslesen:

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

  - Abbildung ändern:

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

  - Abbildung löschen:

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


Eine naheliegende Implementation

- Modellierung als Haskell-Funktion:

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

- Damit einfaches `lookup`, `insert`, `delete`:

```
empty = Map (λx→ Nothing)
```

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

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

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

- Instanzen von Eq, Show nicht möglich
- Speicherleck: überschriebene Zellen werden nicht freigegeben

Endliche Abbildungen: Anwendungsbeispiel

- Lager als endliche Abbildung:

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

- Artikel suchen:

```
suche :: Artikel→ Lager→ Maybe Menge
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
   Just p  → case M.lookup a l of
     Just q  → Lager (M.insert a (fromJust (hinzu q p)) l)
     Nothing → Lager (M.insert a p l)
   Nothing → Lager l
```

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

Map als Assoziativliste

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

- Zusatzfunktionalität:
  - Iteration (`foldr`)

    ```
    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
```

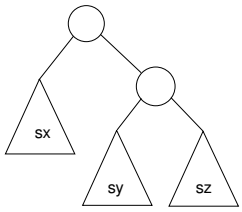

Balance sicherstellen

- Problem:

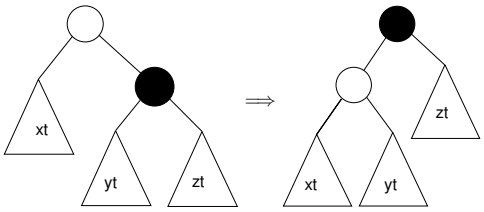
  Nach Löschen oder Einfügen zu großes Ungewicht

- Lösung:

  Rotieren der Unterbäume

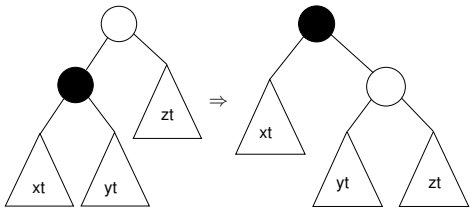


Linksrotation




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

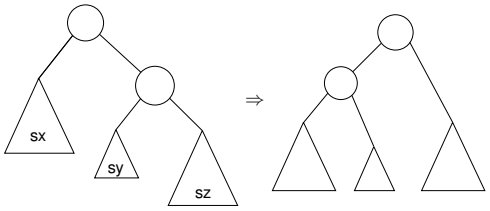

Rechtsrotation




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

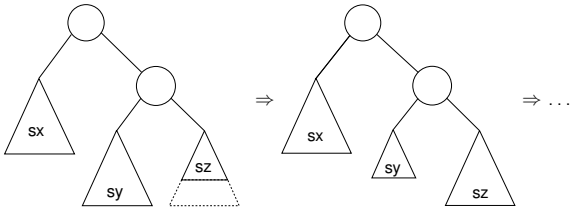

Balanciertheit sicherstellen

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



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
- Klasseninstanzen werden immer exportiert.
- Kein Paket-System

ADTs vs. Objekte

- ADTs (Haskell): Typ plus Operationen
- Objekte (z.B. Java): Interface, Methoden.
- Gemeinsamkeiten:
  - Verkapselung (information hiding) der Implementation
- Unterschiede:
  - Objekte haben internen Zustand, ADTs sind referentiell transparent;
  - Objekte haben Konstruktoren, ADTs nicht (Konstruktoren nicht unterscheidbar)
  - Vererbungsstruktur auf Objekten (Verfeinerung für ADTs)
  - Java: `interface` eigenes Sprachkonstrukt
  - Java: `packages` für Sichtbarkeit

Zusammenfassung

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

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 8 vom 06.12.2016: Signaturen und Eigenschaften

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

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
- Typ Map α β, Operationen:

```
data Map α β

empty  :: Map α β

lookup :: Ord α⇒ α→ Map α β→ Maybe β

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

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


Signatur und Eigenschaften

- Signatur genug, um ADT typkorrekt zu benutzen
  - Insbesondere Anwendbarkeit und Reihenfolge
- Signatur beschreibt nicht die Bedeutung (Semantik):
  - Was wird gelesen?
  - Wie verhält sich die Abbildung?
- Signatur ist Sprache (Syntax) um Eigenschaften zu beschreiben

Eigenschaften Endlicher Abbildungen

1. Aus der leeren Abbildung kann nichts gelesen werden.
2. Wenn etwas geschrieben wird, und an der gleichen Stelle wieder gelesen, erhalte ich den geschriebenen Wert.
3. Wenn etwas geschrieben wird, und an anderer Stelle etwas gelesen wird, kann das Schreiben vernachlässigt werden.
4. An der gleichen Stelle zweimal geschrieben überschreibt der zweite den ersten Wert.
5. An unterschiedlichen Stellen geschrieben kommutiert.

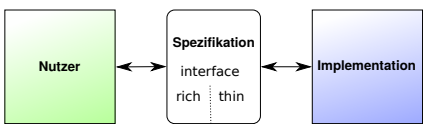
Formalisierung von Eigenschaften

Definition (Axiome)
Axiome sind Prädikate über den Operationen der Signatur

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

Axiome als Interface

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



Axiome für Map

- Lesen aus leerer Abbildung undefiniert:

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

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

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

```
lookup a (delete a (s :: Map Int String)) == Nothing
```

- Lesen an anderer Stelle liefert alten Wert:

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

- Schreiben an dieselbe Stelle überschreibt alten Wert:

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

- Schreiben über verschiedene Stellen kommutiert:

```
a ≠ b ⟹ insert a v (delete b s) ==
          delete b (insert a v s :: Map Int String)
```

- Sehr viele Axiome (insgesamt 12)!

Thin vs. Rich Interfaces

- Benutzersicht: reiches Interface
  - Viele Operationen und Eigenschaften
- Implementationssicht: schlankes Interface
  - Wenig Operation und Eigenschaften
- Beispiel Map:
  - Rich interface:

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

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

  - Thin interface:

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

  - Thin-to-rich:

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

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


Axiome für Map (thin interface)

- Lesen aus leerer Abbildung undefiniert:

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

- Lesen an vorher geschriebener Stelle liefert geschriebenen Wert:

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

- Lesen an anderer Stelle liefert alten Wert:

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

- Schreiben an dieselbe Stelle überschreibt alten Wert:

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

- Schreiben über verschiedene Stellen kommutiert:

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

Thin: 5 Axiome
Rich: 12 Axiome

Axiome als Eigenschaften

- Axiome können getestet oder bewiesen werden
- Tests finden Fehler, Beweis zeigt Korrektheit

> E.W. Dijkstra, 1972
> Program testing can be used to show the presence of bugs, but never to show their absence.

- Arten von Tests:
  - Unit tests (JUnit, HUnit)
  - Black Box vs. White Box
  - Coverage-based (z.B. path coverage, MC/DC)
  - Zufallsbasiertes Testen
- Funktionale Programme eignen sich sehr gut zum Testen

Zufallsbasiertes Testen in Haskell

- Werkzeug: *QuickCheck*
- Zufällige Werte einsetzen, Auswertung auf True prüfen
- Polymorphe Variablen nicht testbar
  - Deshalb Typvariablen instantiieren
  - Typ muss genug Element haben (hier Map Int String)
  - Durch Signatur Typinstanz erzwingen
- Freie Variablen der Eigenschaft werden Parameter der Testfunktion

Axiome mit *QuickCheck* testen

- Für das Lesen:

```
prop1 :: TestTree
prop1 = QC.testProperty "read_empty" $ λa→
  lookup a (empty :: Map Int String) == Nothing
```

```
prop2 :: TestTree
prop2 = QC.testProperty "lookup_put␣eq" $ λa v s→
  lookup a (put a v (s :: Map Int String)) == v
```

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

Axiome mit *QuickCheck* testen

- Bedingte Eigenschaften:
  - A ⟹ B mit A, B Eigenschaften
  - Typ ist Property
  - Es werden solange Zufallswerte generiert, bis N die Vorbedingung erfüllende gefunden und getestet wurden, andere werden ignoriert.

```
prop3 :: TestTree
prop3 = QC.testProperty "lookup_put␣other" $ λa b v s→
  a ≠ b ⟹ lookup a (put b v s) ==
            lookup a (s :: Map Int String)
```


Axiome mit *QuickCheck* testen

- Schreiben:

```
prop4 :: TestTree
prop4 = QC.testProperty "put_put␣eq" $ λa v w s→
  put a w (put a v s) == put a w (s :: Map Int String)
```

- Schreiben an anderer Stelle:

```
prop5 :: TestTree
prop5 = QC.testProperty "put_put␣other" $ λa v b w s→
  a ≠ b ⟹ put a v (put b w s) ==
            put b w (put a v s :: Map Int String)
```

- Test benötigt Gleichheit und Zufallswerte für Map a b

Zufallswerte selbst erzeugen

- Problem: Zufällige Werte von selbstdefinierten Datentypen
  - Gleichverteiltheit nicht immer erwünscht (z.B. $[\alpha]$)
  - Konstruktion nicht immer offensichtlich (z.B. Map)
- In *QuickCheck*:
  - Typklasse **class** Arbitrary α für Zufallswerte
  - Eigene Instanziierung kann Verteilung und Konstruktion berücksichtigen

```
instance (Ord a, QC.Arbitrary a, QC.Arbitrary b)⇒
         QC.Arbitrary (Map a b) where
```

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

Beobachtbare und Abstrakte Typen

- Beobachtbare Typen: interne Struktur bekannt
  - Vordefinierte Typen (Zahlen, Zeichen), algebraische Datentypen (Listen)
  - Viele Eigenschaften und Prädikate bekannt
- Abstrakte Typen: interne Struktur unbekannt
  - Wenige Eigenschaften bekannt, Gleichheit nur wenn definiert
- Beispiel Map:
  - beobachtbar: Adressen und Werte
  - abstrakt: Speicher

Beobachtbare Gleichheit

- Auf abstrakten Typen: nur beobachtbare Gleichheit
  - Zwei Elemente sind gleich, wenn alle Operationen die gleichen Werte liefern
- Bei Implementation: Instanz für Eq (Ord etc.) entsprechend definieren
  - Die Gleichheit $==$ muss die beobachtbare Gleichheit sein.
- Abgeleitete Gleichheit (**deriving** Eq) wird immer exportiert!

Signatur und Semantik

Stacks
Typ: St α
Initialwert:

```
empty :: St α
```

Wert ein/auslesen:

```
push :: α→ St α→ St α
top  :: St α→ α
pop  :: St α→ St α
```

Last in first out (LIFO).

Queues
Typ: Qu α
Initialwert:

```
empty :: Qu α
```

Wert ein/auslesen:

```
enq  :: α→ Qu α→ Qu α
first :: Qu α→ α
deq  :: Qu α→ Qu α
```

First in first out (FIFO)

Gleiche Signatur, unterscheidliche Semantik.

Eigenschaften von Stack

- Last in first out (LIFO):

```
top (push a s) == a
```

```
pop (push a s) == s
```

```
push a s ≠ empty
```


Eigenschaften von Queue

- First in first out (FIFO):

```
first (enq a empty) == a
```

```
q ≠ empty ⟹ first (enq a q) == first q
```

```
deq (enq a empty) == empty
```

```
q ≠ empty ⟹ deq (enq a q) = enq a (deq q)
```

```
enq a q ≠ empty
```


Implementation von Stack: Liste

Sehr einfach: ein Stack ist eine Liste

```
data St α = St [α]   deriving (Show, Eq)
```

```
empty =  St []
```

```
push a (St s) = St (a:s)
```

```
top  (St [])  = error "St: top on empty stack"
top  (St s)   = head s
```

```
pop  (St [])  = error "St: pop on empty stack"
pop  (St s)   = St (tail s)
```


Implementation von Queue

- Mit einer Liste?
  - Problem: am Ende anfügen oder abnehmen ist teuer.
- Deshalb zwei Listen:
  - Erste Liste: zu entnehmende Elemente
  - Zweite Liste: hinzugefügte Elemente rückwärts
  - Invariante: erste Liste leer gdw. Queue leer

Repräsentation von `Queue`

Operation	Resultat	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:

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


Implementation

- Bei enq und deq Invariante prüfen

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

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

  - Prüfung der Invariante nach dem Einfügen und Entnehmen

- check garantiert Invariante

```
check :: [α]→ [α]→ Qu α
check [] ys = Qu (reverse ys) []
check xs ys = Qu xs ys
```


Zusammenfassung

- Signatur: Typ und Operationen eines ADT
- Axiome: über Typen formulierte Eigenschaften
- Spezifikation = Signatur + Axiome
  - Interface zwischen Implementierung und Nutzung
  - Testen zur Erhöhung der Konfidenz und zum Fehlerfinden
  - Beweisen der Korrektheit
- QuickCheck:
  - Freie Variablen der Eigenschaften werden Parameter der Testfunktion
  - $\Longrightarrow$ für bedingte Eigenschaften

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 9 vom 13.12.2016: Spezifikation und Beweis

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

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 :: $\alpha \to$ 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
□

Sonderfall Gleichungen:

Lemma (2) a= b

```
  a
= x1   — Begründung
= x2   — Begründung
= ...
= b
```

□

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:

Für alle x in T 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

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 10 vom 20.12.2016: Aktionen und Zustände

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

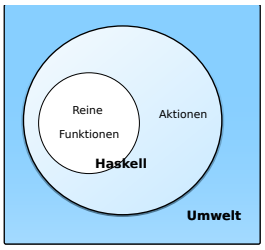
Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Inhalt

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

Ein- und Ausgabe in funktionalen Sprachen



Problem:

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

Lösung:

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

Aktionen als abstrakter Datentyp

- ADT mit Operationen Komposition und Lifting
- Signatur:

```
type IO α

(>>=)  :: IO α → (α→ IO β) → IO β

return :: α→ IO α
```

- Dazu elementare Aktionen (lesen, schreiben etc)

Elementare Aktionen

- Zeile von Standardeingabe (stdin) lesen:

```
getLine :: IO String
```

- Zeichenkette auf Standardausgabe (stdout) ausgeben:

```
putStr  :: String→ IO ()
```

- Zeichenkette mit Zeilenvorschub ausgeben:

```
putStrLn :: String→ IO ()
```


Einfache Beispiele

- Echo einfach

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

- Echo mehrfach

```
echo :: IO ()
echo = getLine >>= putStrLn >>= λ_ → echo
```

- Was passiert hier?
  - Verknüpfen von Aktionen mit >>=
  - 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, Control.Exception)
- 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 System.IO der Standardbücherei
  - Buffered/Unbuffered, Seeking, &c.
  - Operationen auf Handle
- Noch mehr Operationen in System.Posix
  - Filedeskriptoren, Permissions, special devices, etc.

Beispiel: Zeichen, Wörter, Zeilen zählen (wc)

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

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

Aktionen als Werte

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

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

- Iteration (feste Anzahl):

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


Kontrollstrukturen

- Vordefinierte Kontrollstrukturen (Control.Monad):

```
when :: Bool→ IO ()→ IO ()
```

- Sequenzierung:

```
sequence :: [IO α]→ IO [α]
```

  - Sonderfall: [()] als ()

```
sequence_ :: [IO ()]→ IO ()
```

- Map und Filter für Aktionen:

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


Fehlerbehandlung

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

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

- 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 :: IOError))
```

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

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


Ausführbare Programme

- Eigenständiges Programm ist Aktion
- Hauptaktion: main :: IO () in Modul Main
  - ... oder mit der Option `-main-is M.f` setzen
- wc als eigenständiges Programm:

```
module Main where

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

...

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


Beispiel: Traversion eines Verzeichnisbaums

- Verzeichnisbaum traversieren, und für jede Datei eine Aktion ausführen:

```
travFS :: (FilePath→ IO ())→ FilePath→ IO ()
```

- Nutzt Funktionalität aus System.Directory, System.FilePath

```
travFS action p = do
  res ← try (getDirectoryContents p)
  case res of
    Left  e → putStrLn $ "ERROR: "++ show (e :: IOError)
    Right cs → do let cp = map (p </>) (cs \\ [".", ".."])
                  dirs  ← filterM doesDirectoryExist cp
                  files ← filterM doesFileExist cp
                  mapM_ action files
                  mapM_ (travFS action) dirs
```


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

- Trennung zwischen Spiel-Logik und Nutzerschnittstelle
- Spiel-Logik (GuessGame):
  - Programmzustand:

```
data State = St { word   :: String — Zu ratendes Wort
                , hits   :: String — Schon geratene Buchstaben
                , miss   :: String — Falsch geratene Buchstaben
                }
```

  - Initialen Zustand (Wort auswählen):

```
initialState :: [String]→ IO State
```

  - Nächsten Zustand berechnen (Char ist Eingabe des Benutzers):

```
data Result = Miss | Hit | Repetition | GuessedIt | TooManyTries

processGuess :: Char→ State→ (Result, State)
```


Wörter raten: Nutzerschnittstelle

- Hauptschleife (play)
  - Zustand anzeigen
  - Benutzereingabe abwarten
  - Neuen Zustand berechnen
  - Rekursiver Aufruf mit neuem Zustand
- Programmanfang (main)
  - Lexikon lesen
  - Initialen Zustand berechnen
  - Hauptschleife aufrufen

```
play :: State→ IO ()
play st = do
  putStrLn (render st)
  c ← getGuess st
  case (processGuess c st) of
    (Hit, st)  → play st
    (Miss, st) → do putStrLn "Sorry, no."; play st
    (Repetition, st)→ do putStrLn "You already tried that."; play st
    (GuessedIt, st)→ putStrLn "Congratulations, you guessed it."
    (TooManyTries, st) →
      putStrLn $ "The word was "++ word st ++ " — you lose."
```


Kontrollumkehr

- Trennung von Logik (State, processGuess) und Nutzerinteraktion nützlich und sinnvoll
- Wird durch Haskell Tysystem unterstützt (keine UI ohne IO)
- Nützlich für andere UI mit Kontrollumkehr
- Beispiel: ein GUI für das Wörterratespiel (mit Gtk2hs)
  - GUI ruft Handler-Funktionen des Nutzerprogramms auf
  - Spielzustand in Referenz (IORef) speichern
- Vgl. MVC-Pattern (Model-View-Controller)

Zusammenfassung

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

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

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

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 11 vom 10.01.2017: Monaden als Berechnungsmuster

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Frohes Neues Jahr!

Organisatorisches

- Fachgespräche: 1.–3. Februar (letzte Semesterwoche)
- Mündliche Prüfung: entweder in der Zeit, oder individuell zu vereinbaren.
- Prüfungen müssen bis zum 31.03.2017 (Semesterende) stattgefunden haben.

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Inhalt

- Wie geht das mit IO?
- Das M-Wort
- Monaden als allgemeine Berechnungsmuster
- Fallbeispiel: Interpreter für IMP

Zustandsabhängige Berechnungen

Funktionen mit Zustand

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

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

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

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


In Haskell: Zustände explizit

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

```
type State σ α = σ → (α, σ)
```

- Komposition zweier solcher Berechnungen:

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

- Trivialer Zustand:

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

- Lifting von Funktionen:

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

Zugriff auf den Zustand

- Zustand lesen:

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

- Zustand setzen:

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


Einfaches Beispiel

- Zähler als Zustand:

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

- Beispiel: Funktion, die in Kleinbuchstaben konvertiert und zählt

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

- Hauptfunktion (verkapselt State):

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


Monaden

Monaden als Berechnungsmuster

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

State:

```
type State σ α

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

lift :: α→ State σ α

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

Aktionen:

```
type IO α

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

return :: α→ IO α

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

Berechnungsmuster: **Monade**

Monaden als Berechngsmuster

Eine Monade ist:

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

Monaden in Haskell

- Aktion auf Funktionen:

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

fmap bewahrt Identität und Komposition:

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

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

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

>>=ist assoziativ und return das neutrale Element:

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

- Den syntaktischen Zucker (**do**-Notation) gibt's umsonst dazu.

Beispiele für Monaden

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

Fehler und Ausnahmen

- Maybe als Monade:

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

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

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

Mehrdeutigkeit

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

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

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

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

IO ist keine Magie

Implizite vs. explizite Zustände

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

Aktionen als Zustandstransformationen

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

Fallbeispiel: Die Sprache IMP

Monaden im Einsatz

- Gegeben: imperative Programmiersprache IMP
- Ein Interpreter für IMP benötigt:
  - Parser
  - Interpreter zur Auswertung

IMP — Grammatik

identifier	::=	*Char* (*Char* | *Digit*)*
number	::=	*Digit*+ (. *Digit*+)?
expr	::=	*aterm* <= *expr* | *aterm* = *expr* | *aterm*
aterm	::=	*term* + *aterm* | *term*
term	::=	*factor* * *term* | *factor* / *term* | *factor*
factor	::=	*identifier* | *number* | (*expr*) | - *expr*
expr	::=	*expr* <= *expr* | *expr* = *expr*
	|	*expr* + *expr*
	|	*expr* * *expr* | *expr* / *expr*
	|	*identifier* | *number* | (*expr*) | - *expr*
cmd	::=	*identifier* := *expr*
	|	while *expr* { *cmds* }
	|	if *expr* { *cmds* } (else{*cmds* })?
	|	print *expr*
cmds	::=	*cmd* ; *cmds* | *cmd*
decl	::=	var *identifier* ;
Prog	::=	*decl** *cmds*

Beispielprogramm: Fakultät

```
var fak;
var n;

n := 10;

fak := 1;
while 1 ≤ n {
  print fak;
  fak := fak * n;
  n := n+(−1)
  }
```

Parser

- Monadischer Kombinatorparser
  - nach Graham Hutton, Erik Meijer: *Monadic parsing in Haskell*, J. Funct. Program. **8**:4, 1998, p 437-444.
- Eingabe ist Sequenz von Eingabetoken (Char), Rückgabe ist abstrakter Syntaxbaum (AST)
- Zustand des Parsers: noch zu lesende Eingabesequenz
- Typ (generisch über Eingabetoken α und AST β):

```
data Parser α β = Parser { parse :: [α] → [(β, [α])] }
```

  - Kombination aus State-Monade (Zustand) und Listen-Monade (Nichtdeterminismus)

Parser

- Basisparser: `satisfy`, erkennt einzelne Token

```
satisfy :: (α→ Bool)→ Parser α α
```

- Kombinator: Sequenzierung des Monaden:

```
(>>=) :: Parser α β→ (β→ Parser α γ)→ Parser α γ
```

- Kombinator: $(+\!+)$ ist Auswahl

```
(++) :: Parser a b → Parser a b → Parser a b
```

- Darauf aufgebaut: optional, Kleene-Stern, ...

```
opt :: Parser a b → Parser a (Maybe b)

many :: Parser a b → Parser a [b]

sepby :: Eq a⇒ Parser a b → a → Parser a [b]
```


Auswertung

- Auswertung: Systemzustand und eventueller Fehler:

```
data State = State { vars :: M.Map Id Val
                   , output :: [String]
                   }
data St a = St { run :: State→ Error (a, State) }
```

- Ausführung von Kommandos:

```
exec :: Cmd → St ()
```

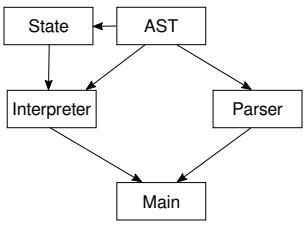
- Auswertung von Ausdrücken (keine Änderung des Systemzustands):

```
eval :: Expr→ State→ Error Val
```


Ausführung von Kommandos

```
exec :: Cmd → St ()
exec w@(While e cs) = do
  v← evaluate e
  vs ← get vars
  if isTrue v then do {execs cs; exec w} else return ()
exec (If e cs1 cs2) = do
  v← evaluate e
  if isTrue v then execs cs1 else execs cs2
exec (Assign i e) = do
  v← evaluate e
  set $ λs→ s{vars= M.insert i v (vars s)}
exec (Print e) = do
  v← evaluate e
  set $ λs→ s{output= show v : output s}
```


IMP-Interpreter: Modulstruktur



Zusammenfassung

- Monaden sind Muster für Berechnungen mit Seiteneffekten
- Beispiele:
  - Zustandstransformer (State)
  - Fehler und Ausnahmen (Maybe, Either)
  - Nichtdeterminismus (List)
- Fallbeispiel IMP:
  - Parser ist Kombination aus State und List
  - Auswertung ist Kombination aus State und Either
- Grenze: Nebenläufigkeit

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 12 vom 17.01.17: Domänenspezifische Sprachen (DSLs)

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Domain-Specific Languages (DSLs)

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

Programmiersprachen sind überall

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

Vom Allgemeinen zum Speziellen

- Modellierung von Problemen und Lösungen

Allgemein ⟷ Spezifisch

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

Spezifische Lösung: DSL
- Maßgeschneiderte Sprache
- Wohldefinierte Unterklasse (Domäne) von Problemen
- Geringer Abstand zum Problem
- Domain-Specific Language (DSL)
- Als Teil einer Programmiersprache (eingebettet) oder alleinstehend (stand-alone)

DSL: Definition 1

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

(van Deursen et al., 2000)

Eigenschaften von DSLs

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

DSL-Beispiel: Relax NG

Adressbuchformat

```
grammar {
  start = entries
  entries = element entries { entry* }
  entry = element entry {
    attribute name { text },
    attribute birth { xsd:dateTime },
    text }
}
```

- Beschreibung von XML-Bäumen
  - Erlaubte Element-Verschachtelungen & -Reihenfolgen
  - Datentypen von Attributen & Elementwerten
- Automatische Generierung von Validatoren
- Nicht Turing-mächtig (?)

Domain-Specific Embedded Languages

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

Beispiel: Reguläre Ausdrücke

Ein regulärer Ausdruck ist:

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

Haskell-Implementierung — Signatur:

```
type RegEx
```

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

Implementierung: siehe `RegExS.hs`

Regular Ausdrücke: Suche

- Wie modellieren wir mehrfache Suche?
- Signatur:

```
type RegEx =
  String→ [String]
```

- Wie modellieren wir ersetzen?

Besser: Repräsentation durch Datentypen

```
data RE = Eps
        | Chr Char
        | Str String
        | Arb
        | Seq RE RE
        | Alt RE RE
        | Star RE
        | Plus RE
        | Range [Char]
       deriving (Eq, Show)

intp :: RE→ RegEx
```

```
searchAll :: RE→ String→
                    [String]
```


Flache Einbettung vs. Tiefe Einbettung

- Flache Einbettung:
  - Domänenfunktionen direkt als Haskell-Funktionen
  - Keine explizite Repräsentation der Domänenobjekte in Haskell
- Tiefe Einbettung:
  - Repräsentation der Domänenobjekte durch Haskell-Datentyp (oder ADT)
  - Domänenfunktionen auf diesem Datentyp

Flach oder Tief?

- Vorteile flache Einbettung:
  - Schnell geschrieben, weniger 'boilerplate'
  - Flexibel erweiterbar
- Vorteile tiefe Einbettung:
  - Mächtiger: Manipulation der Domänenobjekte
  - Transformation, Übersetzung, ...
  - Bsp: Übersetzung RE in Zustandsautomaten

Beispiel: Grafik

- Erzeugung von SVG-Grafiken
- Eingebettete DSL:
  - Erste Näherung: TinySVG (modelliert nur die Daten)
  - Erweiterung: Monade Draw (Zustandsmonade)
  - Funktionen zum Zeichnen:

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

  - "Ausführen":

```
draw :: Double→ Double→ String→ Draw ()→ IO ()
```


Beispielprogramm: Sierpińksy-Dreieck

Dreieck mit Eckpunkten zeichnen:

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

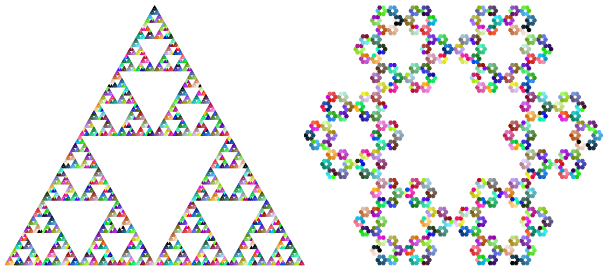
Mitte zwischen zwei Punkten:

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

Sierpińksy-Dreieck rekursiv

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


Resultat: Sierpińsky-Dreieck und Schneeflocke



Erweiterung: Transformation

- Allgemein: Transformation von Grafiken

```
xform :: (Graphics→ Graphics)→ Draw()→ Draw()
```

- Speziell:
  - Rotation um einen Punkt:

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

  - Skalierung um einen Faktor:

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

  - Verschiebung um einen Vektor (Punkt):

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


Beispiele: Verschiebung und Skalierung

Weitere Abgrenzung

Programmierschnittstellen (APIs)

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

Skriptsprachen

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

Beispiel: Hardware Description Languages

- Ziel: Funktionalität von Schaltkreisen beschreiben
- Einfachster Fall:

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

- Moderne Schaltkreise sind etwas komplizierter ...

CλaSH

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

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

Beispiel: SQL

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

Vorteile der Verwendung von DSLs

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

Nachteile der Verwendung von DSLs

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

Zusammenfassung

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

Literatur

- Koen Claessen and David Sands.
  Observable sharing for functional circuit description.
  In P. S. Thiagarajan and R. Yap, editors, *Advances in Computing Science – ASIAN'99*, volume 1742 of *LNCS*, pages 62–73, 1999.
- Paul Hudak.
  Building domain-specific embedded languages.
  ACM Comput. Surv., 28, 1996.
- Marjan Mernik, Jan Heering, and Anthony M. Sloane.
  When and how to develop domain-specific languages.
  ACM Comput. Surv., 37(4):316–344, 2005.
- Arie van Deursen, Paul Klint, and Joost Visser.
  Domain-specific languages: an annotated bibliography.
  SIGPLAN Not., 35(6):26–36, 2000.

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 13 vom 24.01.17: Scala — Eine praktische Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Organisatorisches

- Anmeldung zu den Fachgesprächen ab sofort möglich
  - Unter `stud.ip`, Reiter „Terminvergabe“
- Nächste Woche noch mehr zu den Fachgesprächen
- Es gibt eine Liste mit Übungsfragen (auf der Homepage, unter Übungsblätter)

Heute: Scala

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

Scala am Beispiel: `01-GCD.scala`

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {
    var a = x
    var b = y
    while (a != 0) {
      val temp = a
      a = b % a
      b = temp
    }
    return b
  }

def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd (y, x % y)
```

- Variablen, veränderlich (**var**)
  - *Mit Vorsicht benutzen!*
- Werte, unveränderlich (**val**)
- **while**-Schleifen
  - *Unnötig!*
- Rekursion
  - Endrekursion wird optimiert
- Typinferenz
  - Mehr als Java, weniger als Haskell
- Interaktive Auswertung

Scala am Beispiel: `02-Rational-1.scala`

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {

  require(d != 0)

  private val g = gcd(n.abs, d.abs)
  val numer = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer *
          denom,
      denom * that.denom
    )

  override def toString = numer +"/"+ denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

- Klassenparameter
- Konstruktoren (**this**)
- Klassenvorbedingungen (require)
- private Werte und Methoden
- Methoden, Syntax für Methodenanwendung
- **override** (nicht optional)
- Overloading
- Operatoren
- Companion objects (**object**)

Algebraische Datentypen: `03-Expr.scala`

Was sehen wir hier?

```
abstract class Expr
  case class Var(name: String) extends Expr
  case class Number(num: Double) extends Expr
  case class UnOp(operator: String, arg: Expr)
      extends Expr
  case class BinOp(operator: String,
      left: Expr, right: Expr) extends Expr
```

```
def eval(expr: Expr): Double = expr match {
    case v: Var ⇒ 0     // Variables evaluate to 0
    case Number(x) ⇒ x
    case BinOp("+", e1, e2) ⇒ eval(e1) + eval(e2)
    case BinOp("*", e1, e2) ⇒ eval(e1) * eval(e2)
    case UnOp("-", e) ⇒ - eval(e)
    }

val e = BinOp("*", Number(12),
          UnOp("-", BinOp("+", Number(2.3),
                       Number(3.7))))
```

- **case class** erzeugt
  - Factory-Methode für Konstruktoren
  - Parameter als implizite **val**
  - abgeleitete Implementierung für toString, equals
  - ... und pattern matching (**match**)
- Pattern sind
  - **case** 4 ⇒ Literale
  - **case** C(4) ⇒ Konstruktoren
  - **case** C(x) ⇒ Variablen
  - **case** C(_)⇒ Wildcards
  - **case** x: C ⇒ getypte pattern
  - **case** C(D(x: T, y), 4) ⇒ geschachtelt

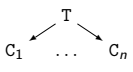
Implementierung algebraischer Datentypen

Haskell:

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

- Ein Typ T
- Konstruktoren erzeugen Datentyp

Scala:



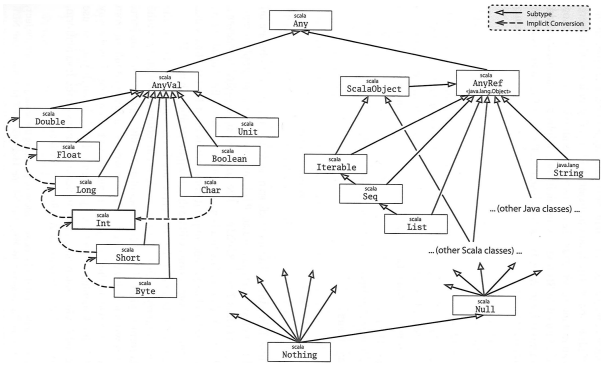
- Varianten als Subtypen
- Problem und Vorteil: Erweiterbarkeit
- **sealed** verhindert Erweiterung

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`
- Warum?
  - Funktionsraum nicht monoton im ersten Argument
  - Sei $X \subseteq Y$, dann $Z \longrightarrow X \subseteq Z \longrightarrow Y$, aber $X \longrightarrow Z \not\subseteq Y \longrightarrow Z$
  - Sondern $Y \longrightarrow Z \subseteq X \longrightarrow Z$

Typvarianz

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

class C[T]
- Rigide
- Kein Subtyping
- Parametertyp T kann beliebig verwendet werden

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

Beispiel:

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


Traits: `05-Funny.scala`

Was sehen wir hier?

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

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

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

More Traits

- Ad-Hoc Polymorphie mit Traits
- Typklasse:

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

- Instanz:

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

- In Aktion:

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


Was wir ausgelassen haben...

- Komprehension (nicht nur für Listen)
- Gleichheit: == (final), equals (nicht final), eq (Referenzen)
- Implizite Parameter und Typkonversionen
- Nebenläufigkeit (Aktoren, Futures)
- Typsichere Metaprogrammierung
- Das *simple build tool* sbt
- Der JavaScript-Compiler scala . js

Schlammschlacht der Programmiersprachen

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

Alle: Nebenläufigkeit, Garbage Collection, FFI

Scala — Die Sprache

- Objekt-orientiert:
  - Veränderlicher, gekapselter Zustand
  - Subtypen und Vererbung
  - Klassen und Objekte
- Funktional:
  - Unveränderliche Werte
  - Parametrische und Ad-hoc Polymorphie
  - Funktionen höherer Ordnung
  - Hindley-Milner Typinferenz

Beurteilung

- Vorteile:
  - Funktional programmieren, in der Java-Welt leben
  - Gelungene Integration funktionaler und OO-Konzepte
  - Sauberer Sprachentwurf, effiziente Implementierung, reiche Büchereien
- Nachteile:
  - Manchmal etwas zu viel
  - Entwickelt sich ständig weiter
  - One-Compiler-Language, vergleichsweise langsam
- Mehr Scala?
  - Besuchen Sie auch Reaktive Programmierung (SoSe 2017)

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 14 vom 31.01.15: Rückblick & Ausblick

Christoph Lüth

Universität Bremen

Wintersemester 2016/17

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Monaden als Berechnungsmuster
  - Domänenspezifische Sprachen (DSLs)
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

Organisatorisches

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

Inhalt

- Fachgespräche und Prüfungen
- Rückblick und Ausblick

Fachgespräche und Prüfungen

Fachgespräche

- Fachgespräche bestehen aus der schriftlichen, nichtelektronischen Bearbeitung einer kurzen Programmieraufgabe
  - Abstufung nach Vornote: A (1– 1.7), B (2.0 – 3.0), C (3.3 – 4.0)
- Beispielfragen auf der Webseite (unter "Übungsaufgaben")

Beispielfrage (C)

Definieren Sie eine Funktion format, die eine Zahl in einer Zeichenkette gegebener Länge rechtsbündig ausgibt.

Bsp. format 4 xy $\rightsquigarrow$ "␣␣xy"

Beispielfrage (B)

Definieren Sie eine Funktion mean, die den arithmetischen Durchschnitt einer Liste von ganzen Zahlen berechnet.

Bsp. mean [2,1,5,4,3] $\rightsquigarrow$ 3.0

Beispielfrage (A)

Zwei Int-Listen sollen ähnlich heißen, wenn Sie die gleichen Zahlen unabhängig von ihrer Reihenfolge und Häufigkeit enthalten. Schreiben Sie eine Testfunktion similar dafür.

Bsp. similar [3,2,2,1,3] [1,2,3] ⇝ True

Mündliche Prüfung

- Dauer: in der Regel 30 Minuten
- Einzelprüfung, ggf. mit Beisitzer
- Inhalt: Programmieren mit Haskell und Vorlesungsstoff
- Ablauf: "Fachgespräch plus"
  - Einstieg mit leichter Programmieraufgabe wie im Fachgespräch
  - Daran anschließend Fragen über den Stoff

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?
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?

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?
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 strukturell 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?
2. Wann kann `foldr f a xs` auch für ein zyklisches Argument `xs` (bspw. eine zyklische Liste) terminieren?
3. Warum sind endrekursive Funktionen im allgemeinen schneller als nicht-endrekursive Funktionen? Unter welchen Voraussetzungen kann ich eine Funktion in endrekursive Form überführen?

Verständnisfragen: Übungsblatt 6

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 7

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 8

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, und wann ist das nötig?

Verständnisfragen: Übungsblatt 9

1. Warum ist die Erzeugung von Zufallszahlen eine Aktion?
2. Warum ist auch das Schreiben in eine Datei eine Aktion?
3. Was ist (bedingt durch den Mangel an referentieller Transparenz) die entscheidende Eigenschaft, die Aktionen von reinen Funktionen unterscheidet?

Rückblick und Ausblick

Warum funktionale Programmierung lernen?

- Funktionale Programmierung macht aus Programmierern Informatiker
- Blick über den Tellerrand — was kommt in 10 Jahren?
- Herausforderungen der Zukunft
- Enthält die wesentlichen Elemente moderner Programmierung

Zusammenfassung Haskell

Stärken:

- Abstraktion durch
  - Polymorphie und Typsystem
  - algebraische Datentypen
  - Funktionen höherer Ordnung
- Flexible Syntax
- Haskell als Meta-Sprache
- Ausgereifter Compiler
- Viele Büchereien

Schwächen:

- Komplexität
- Büchereien
  - Nicht immer gut gepflegt
- Viel im Fluß
  - Kein stabiler und brauchbarer Standard
- Divergierende Ziele:
  - Forschungsplattform und nutzbares Werkzeug

Andere Funktionale Sprachen

- Standard ML (SML):
  - Streng typisiert, strikte Auswertung
  - Standardisiert, formal definierte Semantik
  - Drei aktiv unterstütze Compiler
  - Verwendet in Theorembeweisern (Isabelle, HOL)
  - `http://www.standardml.org/`
- Caml, O'Caml:
  - Streng typisiert, strikte Auswertung
  - Hocheffizienter Compiler, byte code & nativ
  - Nur ein Compiler (O'Caml)
  - `http://caml.inria.fr/`

Andere Funktionale Sprachen

- LISP und Scheme
  - Ungetypt/schwach getypt
  - Seiteneffekte
  - Viele effiziente Compiler, aber viele Dialekte
  - Auch industriell verwendet
- Hybridsprachen:
  - Scala (Functional-OO, JVM)
  - F# (Functional-OO, .Net)
  - Clojure (Lisp, JVM)

Was spricht gegen funktionale Programmierung?

- Mangelnde Unterstützung:
  - Libraries, Dokumentation, Entwicklungsumgebungen
  - Wird besser (Scala)...
- Programmierung nur kleiner Teil der SW-Entwicklung
- Nicht verbreitet — funktionale Programmierer zu teuer
- Konservatives Management
  - "Nobody ever got fired for buying IBMSAP"

Haskell in der Industrie

- Simon Marlow bei Facebook: Sigma — Fighting spam with Haskell
- Finanzindustrie: Barclays Capital, Credit Suisse, Deutsche Bank
- Bluespec: Schaltkreisentwicklung, DSL auf Haskell-Basis
- Galois, Inc: Cryptography (Cryptol DSL)
- Siehe auch: Haskell in Industry

Funktionale Programmierung in der Industrie

- Scala:
  - Twitter, Foursquare, Guardian, ...
- Erlang
  - schwach typisiert, nebenläufig, strikt
  - Fa. Ericsson (Telekom-Anwendungen), WhatsApp
- FL
  - ML-artige Sprache
  - Chip-Verifikation der Fa. Intel
- Python (und andere Skriptsprachen):
  - Listen, Funktionen höherer Ordnung (map, fold), anonyme Funktionen, Listenkomprehension

Perspektiven funktionaler Programmierung

- Forschung:
  - Ausdrucksstärkere Typsysteme
  - für effiziente Implementierungen
  - und eingebaute Korrektheit (Typ als Spezifikation)
  - Parallelität?
- Anwendungen:
  - Eingebettete domänenspezifische Sprachen
  - Zustandsfreie Berechnungen (MapReduce, Hadoop, Spark)
  - Big Data and Cloud Computing

If you liked this course, you might also like ...

- Die Veranstaltung Reaktive Programmierung (Sommersemester 2017)
  - 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 2017/18 — meldet Euch bei Berthold Hoffmann (oder bei mir)!

An Important Public Service Announcement

Tschüß!