Fortgeschrittene Programmierung Vorlesung WS 09,10; SS 12–14, 16,17

Johannes Waldmann, HTWK Leipzig

2. April 2018

1 Einleitung

Programmierung im Studium bisher

- 1. Sem: Modellierung (formale Spezifikationen)
- 1./2. Sem Grundlagen der (AO) Programmierung
  - imperatives Progr. (Programm ist Folge von Anweisungen, bewirkt Zustandsänderung)
  - strukturiertes P. (genau ein Eingang/Ausgang je Teilp.)
  - objektorientiertes P. (Interface = abstrakter Datentyp, Klasse = konkreter Datentyp)
- 2. Sem: Algorithmen und Datenstrukturen
  (Spezifikation, Implementierung, Korrektheit, Komplexität)
- 3. Sem: Softwaretechnik (industrielle Softwareproduktion)
- 3./4. Sem: Softwarepraktikum

Worin besteht jetzt der Fortschritt?

- *deklarative* Programmierung

  (Programm *ist* ausführbare Spezifikation)
- insbesondere: *funktionale* Programmierung

  Def: Programm berechnet *Funktion* f : Eingabe $\mapsto$ Ausgabe,

  (kein Zustand, keine Zustandsänderungen)

- – Daten (erster Ordnung) sind Bäume
  - – Programm ist Gleichungssystem
  - – Programme sind auch Daten (höherer Ordnung)
- ausdrucksstark, sicher, effizient, parallelisierbar

Formen der deklarativen Programmierung

- funktionale Programmierung: `foldr (+) 0 [1,2,3]`

```
foldr f z l = case l of
    [] -> z ; (x:xs) -> f x (foldr f z xs)
```

- logische Programmierung: `append(A,B,[1,2,3]).`

```
append([],YS,YS).
append([X|XS],YS,[X|ZS]):-append(XS,YS,ZS).
```

- Constraint-Programmierung

```
(set-logic QF_LIA) (set-option :produce-models true)
(declare-fun a () Int) (declare-fun b () Int)
(assert (and (>= a 5) (<= b 30) (= (+ a b) 20)))
(check-sat) (get-value (a b))
```

Definition: Funktionale Programmierung

- Rechnen = Auswerten von Ausdrücken (Termbäumen)
- Dabei wird ein *Wert* bestimmt
  und es gibt keine (versteckte) *Wirkung*.
  (engl.: side effect, dt.: Nebenwirkung)
- Werte können sein:
  - – "klassische" Daten (Zahlen, Listen, Bäume. . .)
    `True :: Bool, [3.5, 4.5] :: [Double]`
  - – Funktionen (Sinus, . . .)
    `[sin, cos] :: [Double -> Double]`
  - – Aktionen (Datei lesen, schreiben, . . .)
    `readFile "foo.text" :: IO String`

Softwaretechnische Vorteile

. . . der funktionalen Programmierung

- Beweisbarkeit: Rechnen mit Programmen wie in der Mathematik mit Termen
- Sicherheit: es gibt keine Nebenwirkungen und Wirkungen sieht man bereits am Typ
- Aussdrucksstärke, Wiederverwendbarkeit: durch Funktionen höherer Ordnung (sog. Entwurfsmuster)
- Effizienz: durch Programmtransformationen im Compiler,
- Parallelität: keine Nebenwirkungen $\Rightarrow$ keine *data races*, fktl. Programme sind *automatisch parallelisierbar*

Beispiel Spezifikation/Test

```
import Test.LeanCheck

append :: forall t . [t] -> [t] -> [t]
append [] y = y
append (h : t) y = h : (append t y)

associative f =
   \ x y z -> f x (f y z) == f (f x y) z
commutative f = \ x y -> ...

test = check
  (associative (append::[Bool]->[Bool]->[Bool]))
```

Übung: Kommutativität (formulieren und testen)

Beispiel Verifikation

```
app :: forall t . [t] -> [t] -> [t]
app [] y = y
app (h : t) y = h : (app t y)

Lemma: app x (app y z) .=. app (app x y) z
```

```
Proof by induction on List x
  Case []
    To show: app [] (app y z) .=. app (app [] y) z
  Case h:t
    To show: app (h:t) (app y z) .=. app (app (h:t) y) z
    IH: app t (app y z) .=. app (app t y) z
```

CYP `https://github.com/noschinl/cyp`,
ist vereinfachte Version von Isabelle `https://isabelle.in.tum.de/`

Beispiel Parallelisierung (Haskell)

Klassische Implementierung von Mergesort

```
sort :: Ord a => [a] -> [a]
sort [] = [] ; sort [x] = [x]
sort xs =  let ( left,right ) = split xs
               sleft  = sort left
               sright = sort right
           in  merge sleft sright
```

wird parallelisiert durch *Annotationen*:

```
  sleft  = sort left
             `using` rpar `dot` spineList
  sright = sort right `using` spineList
```

vgl. `http://thread.gmane.org/gmane.comp.lang.haskell.parallel/181/focus=202`

Beispiel Parallelisierung (C#, PLINQ)

- Die Anzahl der 1-Bits einer nichtnegativen Zahl:

```
Func<int,int>f =
   x=>{int s=0; while(x>0){s+=x%2;x/=2;}return s;}
```

- $\sum_{x=0}^{2^{26}-1} f(x)$ `Enumerable.Range(0,1<<26).Select(f).Sum()`
- automatische parallele Auswertung, Laufzeitvergleich:

```
Time(()=>Enumerable.Range(0,1<<26).Select(f).Sum())
Time(()=>Enumerable.Range(0,1<<26).AsParallel()
                                  .Select(f).Sum())
```

vgl. *Introduction to PLINQ* `https://msdn.microsoft.com/en-us/library/dd997425(v=vs.110).aspx`

Softwaretechnische Vorteile

. . . der statischen Typisierung

> The language in which you write profoundly affects the design of programs written in that language.
>
> For example, in the OO world, many people use UML to sketch a design. In Haskell or ML, one writes type signatures instead. Much of the initial design phase of a functional program consists of writing type definitions.
>
> **Unlike UML, though, all this design is incorporated in the final product, and is machine-checked throughout.**

Simon Peyton Jones, in: Masterminds of Programing, 2009; `http://shop.oreilly.com/product/9780596515171.do`

Deklarative Programmierung in der Lehre

- funktionale Programmierung: diese Vorlesung
- logische Programmierung: in *Angew. Künstl. Intell.*
- Constraint-Programmierung: als Master-Wahlfach

Beziehungen zu weiteren LV: Voraussetzungen

- Bäume, Terme (Alg.+DS, Grundlagen Theor. Inf.)
- Logik (Grundlagen TI, Softwaretechnik)

Anwendungen:

- Softwarepraktikum
- weitere Sprachkonzepte in *Prinzipien v. Programmiersprachen*
- *Programmverifikation* (vorw. f. imperative Programme)

Konzepte und Sprachen

Funktionale Programmierung ist ein *Konzept*. Realisierungen:

- in prozeduralen Sprachen:
  - Unterprogramme als Argumente (in Pascal)
  - Funktionszeiger (in C)
- in OO-Sprachen: Befehlsobjekte
- Multi-Paradigmen-Sprachen:
  - Lambda-Ausdrücke in C#, Scala, Clojure
- funktionale Programmiersprachen (LISP, ML, Haskell)

Die Erkenntnisse sind sprachunabhängig.

- A good programmer can write LISP in any language.
- Learn Haskell and become a better Java programmer.

Gliederung der Vorlesung

- Terme, Termersetzungssysteme algebraische Datentypen, Pattern Matching, Persistenz
- Funktionen (polymorph, höherer Ordnung), Lambda-Kalkül, Rekursionsmuster
- Typklassen zur Steuerung der Polymorphie
- Bedarfsauswertung, unendl. Datenstrukturen (Iterator-Muster)
- weitere Entwurfsmuster
- Code-Qualität, Code-Smells, Refactoring

Softwaretechnische Aspekte

- algebraische Datentypen, Pattern Matching, Termersetzungssysteme
  Scale: case class, Java: Entwurfsmuster Kompositum,
  immutable objects, das Datenmodell von Git
- Funktionen (höherer Ordnung), Lambda-Kalkül, Rekursionsmuster
  Lambda-Ausdrücke in C#, Entwurfsmuster Besucher
  Codequalität, code smells, Refaktorisierung

- Typklassen zur Steuerung der Polymorphie

  Interfaces in Java/C# , automatische Testfallgenerierung

- Bedarfsauswertung, unendl. Datenstrukturen

  Iteratoren, Ströme, LINQ

Organisation der LV

- jede Woche eine Vorlesung, eine Übung
- Hausaufgaben (teilw. autotool)

  `https://autotool.imn.htwk-leipzig.de/new/vorlesung/238/aufgaben/aktuell`

  Identifizierung und Authentifizierung über Shibboleth-IdP des HTWK-Rechenzentrums, wie bei OPAL

- Prüfungszulassung: regelmäßiges (d.h. innerhalb der jeweiligen Deadline) und erfolgreiches (ingesamt $\geq 50\%$ der Pflichtaufgaben) Bearbeiten von Übungsaufgaben.
- Prüfung: Klausur (ohne Hilfsmittel)

Literatur

- Skripte:
  - aktuelles Semester `http://www.imn.htwk-leipzig.de/~waldmann/lehre.html`
  - vorige Semester `http://www.imn.htwk-leipzig.de/~waldmann/lehre-alt.html`
- Entwurfsmuster: `http://www.imn.htwk-leipzig.de/~waldmann/draft/pub/hal4/emu/main.pdf`
- Maurice Naftalin und Phil Wadler: *Java Generics and Collections*, O'Reilly 2006
- `http://haskell.org/` (Sprache, Werkzeuge, Tutorials), `http://book.realworldhaskel org/`

Alternative Quellen

- – Q: Aber in Wikipedia/Stackoverflow steht, daß ...
  – A: Na und.
- Es mag eine in Einzelfällen nützliche Übung sein, sich mit dem Halbwissen von Nichtfachleuten auseinanderzusetzen.

  Beachte aber `https://xkcd.com/386/`
- In VL und Übung verwenden und diskutieren wir die durch Dozenten/Skript/Modulbeschreibung vorgegebenen Quellen (Lehrbücher, referierte Original-Artikel, Standards zu Sprachen und Bibliotheken)
- ... gilt entsprechend für Ihre Bachelor- und Master-Arbeit.

Übungen KW14

- Benutztung Rechnerpool, ghci aufrufen `http://www.imn.htwk-leipzig.de/~waldmann/etc/pool/`
- Auf wieviele Nullen endet die Fakultät von 100?
  - Benutze `foldr` zum Berechnen der Fakultät.
  - Beachte polymorphe numerische Literale.
    (Auflösung der Polymorphie durch Typ-Annotation.)
    Warum ist 100 Fakultät als `Int` gleich 0?
  - Welches ist der Typ der Funktion ? Beispiel:

    ```
    odd 3 ==> True ; odd 4 ==> False
    takeWhile odd [3,1,4,1,5,9] ==> [3,1]
    ```
  - ersetze in der Lösung `takeWhile` durch andere Funktionen des gleichen Typs (suche diese mit Hoogle), erkläre Semantik
  - typische Eigenschaften dieses Beispiels (nachmachen!)
    statische Typisierung, Schachtelung von Funktionsaufrufen, Funktion höherer Ordnung, Benutzung von Funktionen aus Standardbibliothek (anstatt selbstgeschriebener).
  - schlechte Eigenschaften (vermeiden!)
    Benutzung von Zahlen und Listen (anstatt anwendungsspezifischer Datentypen) vgl. `http://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/list-or-not-list/`

- Haskell-Entwicklungswerkzeuge
  - ghci (Fehlermeldungen, Holes)
  - API-Suchmaschine `http://www.haskell.org/hoogle/`
  - IDE? brauchen wir (in dieser VL) nicht.
    Ansonsten emacs/intero, `http://xkcd.org/378/`
- Softwaretechnik im autotool: `http://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/se/`
- Commercial Uses of Functional Programming `http://www.syslog.cl.cam.ac.uk/2013/09/22/liveblogging-cufp-2013/`

2 Daten

Wiederholung: Terme

- (Prädikatenlogik) *Signatur* Σ ist Menge von Funktionssymbolen mit Stelligkeiten
  ein Term t in Signatur Σ ist
  - Funktionssymbol $f \in \Sigma$ der Stelligkeit k mit Argumenten $(t_1, \ldots, t_k)$, die selbst Terme sind.

  $\mathrm{Term}(\Sigma) =$ Menge der Terme über Signatur Σ
- (Graphentheorie) ein Term ist ein gerichteter, geordneter, markierter Baum
- (Datenstrukturen)
  - Funktionssymbol = Konstruktor, Term = Baum

Beispiele: Signatur, Terme

- Signatur: $\Sigma_1 = \{Z/0, S/1, f/2\}$
- Elemente von $\mathrm{Term}(\Sigma_1)$:
  $Z(), \quad S(S(Z())), \quad f(S(S(Z())), Z())$
- Signatur: $\Sigma_2 = \{E/0, A/1, B/1\}$
- Elemente von $\mathrm{Term}(\Sigma_2)$: ...

Algebraische Datentypen

```
data Foo = Foo { bar :: Int, baz :: String }
    deriving Show
```

Bezeichnungen (benannte Notation)

- `data Foo` ist Typname
- `Foo { .. }` ist Konstruktor
- `bar, baz` sind Komponenten

```
x :: Foo
x = Foo { bar = 3, baz = "hal" }
```

Bezeichnungen (positionelle Notation)

```
data Foo = Foo Int String
y = Foo 3 "bar"
```

Datentyp mit mehreren Konstruktoren

Beispiel (selbst definiert)

```
data T = A { foo :: Int }
       | B { bar :: String, baz :: Bool }
    deriving Show
```

Bespiele (in Prelude vordefiniert)

```
data Bool = False | True
data Ordering = LT | EQ | GT
```

Mehrsortige Signaturen

- (bisher) einsortige Signatur

  Abbildung von Funktionssymbol nach Stelligkeit

- (neu) mehrsortige Signatur
  - Menge von Sortensymbolen $S = \{S_1, \ldots\}$
  - Abb. von F.-Symbol nach Typ

- *Typ* ist Element aus $S^* \times S$
  Folge der Argument-Sorten, Resultat-Sorte

Bsp.: $S = \{Z, B\}, \Sigma = \{0 \mapsto ([], Z), p \mapsto ([Z, Z], Z),\ e \mapsto ([Z, Z], B), a \mapsto ([B, B], B)\}$.

- $\mathrm{Term}(\Sigma)$: konkrete Beispiele, allgemeine Definition?

Rekursive Datentypen

```
data Tree = Leaf {}
    | Branch { left :: Tree
             , right :: Tree }
```

Übung: Objekte dieses Typs erzeugen
(benannte und positionelle Notation der Konstruktoren)

Daten mit Baum-Struktur

- mathematisches Modell: Term über Signatur
- programmiersprachliche Bezeichnung: *algebraischer Datentyp* (die Konstruktoren bilden eine Algebra)
- praktische Anwendungen:
  - Formel-Bäume (in Aussagen- und Prädikatenlogik)
  - Suchbäume (in VL Algorithmen und Datenstrukturen, in `java.util.TreeSet<E>`)
  - DOM (Document Object Model) `https://www.w3.org/DOM/DOMTR`
  - JSON (Javascript Object Notation) z.B. für AJAX `http://www.ecma-international.org/publications/standards/Ecma-404.htm`

Bezeichnungen für Teilterme

- *Position*: Folge von natürlichen Zahlen
  (bezeichnet einen Pfad von der Wurzel zu einem Knoten)
  Beispiel: für $t = S(f(S(S(Z())), Z()))$
  ist $[0, 1]$ eine Position in t.

- $\mathrm{Pos}(t) =$ die Menge der Positionen eines Terms t
  Definition: wenn $t = f(t_1, \ldots, t_k)$,
  dann $\mathrm{Pos}(t) = \{[]\} \cup \{[i-1] +\!\!+ p \mid 1 \leq i \leq k \wedge p \in \mathrm{Pos}(t_i)\}$.

dabei bezeichnen:

- $[]$ die leere Folge,
- $[i]$ die Folge der Länge 1 mit Element i,
- $+\!\!+$ den Verkettungsoperator für Folgen

Operationen mit (Teil)Termen

- $t[p] =$ der Teilterm von t an Position p
  Beispiel: $S(f(S(S(Z())), Z()))[0, 1] = \ldots$
  Definition (durch Induktion über die Länge von p): ...
- $t[p := s]$: wie t, aber mit Term s an Position p
  Beispiel: $S(f(S(S(Z())), Z()))[[0, 1] := S(Z)] = \ldots$
  Definition (durch Induktion über die Länge von p): ...

Operationen mit Variablen in Termen

- $\mathrm{Term}(\Sigma, V) =$ Menge der Terme über Signatur Σ mit Variablen aus V
  Beispiel: $\Sigma = \{Z/0, S/1, f/2\}, V = \{y\}$, $f(Z(), y) \in \mathrm{Term}(\Sigma, V)$.
- Substitution σ: partielle Abbildung $V \to \mathrm{Term}(\Sigma)$
  Beispiel: $\sigma_1 = \{(y, S(Z()))\}$
- eine Substitution auf einen Term anwenden: $t\sigma$:
  Intuition: wie t, aber statt v immer $\sigma(v)$
  Beispiel: $f(Z(), y)\sigma_1 = f(Z(), S(Z()))$
  Definition durch Induktion über t

Termersetzungssysteme

- Daten = Terme (ohne Variablen)
- Programm R = Menge von Regeln
  Bsp: $R = \{(f(Z(), y), y),\ (f(S(x), y), S(f(x, y)))\}$
- Regel = Paar (l, r) von Termen mit Variablen
- Relation $\to_R$ ist Menge aller Paare (t, t') mit
  - es existiert $(l, r) \in R$
  - es existiert Position p in t
  - es existiert Substitution $\sigma : (\mathrm{Var}(l) \cup \mathrm{Var}(r)) \to \mathrm{Term}(\Sigma)$
  - so daß $t[p] = l\sigma$ und $t' = t[p := r\sigma]$.

Termersetzungssysteme als Programme

- $\to_R$ beschreibt *einen* Schritt der Rechnung von R,
- transitive und reflexive Hülle $\to_R^*$ beschreibt *Folge* von Schritten.
- *Resultat* einer Rechnung ist Term in R-Normalform (:= ohne $\to_R$-Nachfolger)

dieses Berechnungsmodell ist im allgemeinen

- *nichtdeterministisch* $R_1 = \{C(x, y) \to x, C(x, y) \to y\}$
  (ein Term kann mehrere $\to_R$-Nachfolger haben, ein Term kann mehrere Normalformen erreichen)
- *nicht terminierend* $R_2 = \{p(x, y) \to p(y, x)\}$
  (es gibt eine unendliche Folge von $\to_R$-Schritten, es kann Terme ohne Normalform geben)

Konstruktor-Systeme

Für TRS R über Signatur Σ: Symbol $s \in \Sigma$ heißt

- *definiert*, wenn $\exists (l, r) \in R : l[] = s(\dots)$ (das Symbol in der Wurzel ist s)
- sonst *Konstruktor*.

Das TRS R heißt *Konstruktor-TRS*, falls:

- definierte Symbole kommen links *nur* in den Wurzeln vor

Übung: diese Eigenschaft formal spezifizieren
Beispiele: $R_1 = \{a(b(x)) \to b(a(x))\}$ über $\Sigma_1 = \{a/1, b/1\}$,
$R_2 = \{f(f(x,y),z) \to f(x,f(y,z))$ über $\Sigma_2 = \{f/2\}$:
definierte Symbole? Konstruktoren? Konstruktor-System?

Funktionale Programme sind ähnlich zu Konstruktor-TRS.

Übung Terme, TRS

- Geben Sie die Signatur des Terms $\sqrt{a \cdot a + b \cdot b}$ an.
- Geben Sie ein Element $t \in \mathrm{Term}(\{f/1, g/3, c/0\})$ an mit $t[1] = c()$.

mit `ghci`:

- `data T = F T | G T T T | C deriving Show`
  erzeugen Sie o.g. Terme (durch Konstruktoraufrufe)

Die *Größe* eines Terms t ist definiert durch
$|f(t_1, \ldots, t_k)| = 1 + \sum_{i=1}^{k} |t_i|$.

- Bestimmen Sie $|\sqrt{a \cdot a + b \cdot b}|$.
- Beweisen Sie $\forall \Sigma : \forall t \in \mathrm{Term}(\Sigma) : |t| = |\,\mathrm{Pos}(t)|$.

Vervollständigen Sie die Definition der *Tiefe* von Termen:

$$\begin{array}{rcl} & & \mathrm{depth}(f()) = 0 \\ k > 0 & \Rightarrow & \mathrm{depth}(f(t_1, \ldots, t_k)) = \ldots \end{array}$$

- Bestimmen Sie $\mathrm{depth}(\sqrt{a \cdot a + b \cdot b})$
- Beweisen Sie $\forall \Sigma : \forall t \in \mathrm{Term}(\Sigma) : \mathrm{depth}(t) < |t|$.

Für die Signatur $\Sigma = \{Z/0, S/1, f/2\}$:

- für welche Substitution σ gilt $f(x, Z)\sigma = f(S(Z), Z)$?

- für dieses σ: bestimmen Sie $f(x, S(x))\sigma$.

Notation für Termersetzungsregeln: anstatt (l, r) schreibe $l \to r$.
Abkürzung für Anwendung von 0-stelligen Symbolen: anstatt $Z()$ schreibe Z.

- Für $R = \{f(S(x), y) \to f(x, S(y)), f(Z, y) \to y\}$
  bestimme alle R-Normalformen von $f(S(Z), S(Z))$.
- für $R_d = R \cup \{d(x) \to f(x, x)\}$
  bestimme alle R_d-Normalformen von $d(d(S(Z)))$.
- Bestimme die Signatur Σ_d von R_d.
  Bestimme die Menge der Terme aus $\mathrm{Term}(\Sigma_d)$, die R_d-Normalformen sind.
- für die Signatur $\{A/2, D/0\}$:
  definiere Terme $t_0 = D, t_{i+1} = A(t_i, D)$.
  Zeichne t_3. Bestimme $|t_i|$.
- für $S = \{A(A(D, x), y) \to A(x, A(x, y))\}$
  bestimme S-Normalform(en), soweit existieren, der Terme t_2, t_3, t_4. Zusatz: von t_i allgemein.

Abkürzung für mehrfache Anwendung eines einstelligen Symbols: $A(A(A(A(x)))) = A^4(x)$

- für $\{A(B(x)) \to B(A(x))\}$
  über Signatur $\{A/1, B/1, E/0\}$:
  bestimme Normalform von $A^k(B^k(E))$
  für $k = 1, 2, 3$, allgemein.
- für $\{A(B(x)) \to B(B(A(x)))\}$
  über Signatur $\{A/1, B/1, E/0\}$:
  bestimme Normalform von $A^k(B(E))$
  für $k = 1, 2, 3$, allgemein.

3 Programme

Funktionale Programme

... sind spezielle Term-Ersetzungssysteme. Beispiel:
Signatur: S einstellig, Z nullstellig, f zweistellig.
Ersetzungssystem $\{f(Z, y) \to y, f(S(x'), y) \to S(f(x', y))\}$.
Startterm $f(S(S(Z)), S(Z))$.
entsprechendes funktionales Programm:

```
data N = Z | S N
f :: N -> N -> N
f x y = case x of
    { Z   -> y  ;  S x' -> S (f x' y) }
```

Aufruf: `f (S (S Z)) (S Z)`

Auswertung = Folge von Ersetzungsschritten $\to_R^*$ Resultat = Normalform (hat keine $\to_R$-Nachfolger)

Pattern Matching

```
data Tree = Leaf | Branch Tree Tree
size :: Tree -> Int
size t = case t of { ... ; Branch l r -> ... }
```

- Syntax: `case <Diskriminante> of { <Muster> -> <Ausdruck> ; ... }`
- `<Muster>` enthält Konstruktoren und Variablen, entspricht linker Seite einer Term-Ersetzungs-Regel, `<Ausdruck>` entspricht rechter Seite
- statische Semantik:
  - jedes `<Muster>` hat gleichen Typ wie `<Diskrim.>`,
  - alle `<Ausdruck>` haben übereinstimmenden Typ.
- dynamische Semantik:
  - Def.: t *paßt* zum Muster l: es existiert σ mit $l\sigma = t$
  - für das erste passende Muster wird $r\sigma$ ausgewertet

Eigenschaften von Case-Ausdrücken

ein `case`-Ausdruck heißt

- *disjunkt*, wenn die Muster nicht überlappen

  (es gibt keinen Term, der zu mehr als 1 Muster paßt)

- *vollständig*, wenn die Muster den gesamten Datentyp abdecken

  (es gibt keinen Term, der zu keinem Muster paßt)

Bespiele (für `data N = F N N | S N | Z`)

```
-- nicht disjunkt:
case t of { F (S x) y -> .. ; F x (S y) -> .. }
-- nicht vollständig:
case t of { F x y -> .. ; Z -> .. }
```

`data` und `case`

typisches Vorgehen beim Verarbeiten algebraischer Daten vom Typ `T`:

- Für jeden Konstruktor des Datentyps

```
data T = C1 ...
       | C2 ...
```

- schreibe einen Zweig in der Fallunterscheidung

```
f x = case x of
   C1 ... -> ...
   C2 ... -> ...
```

- Argumente der Konstruktoren sind Variablen $\Rightarrow$ Case-Ausdruck ist disjunkt und vollständig.

Peano-Zahlen

```
data N = Z | S N

plus :: N -> N -> N
plus x y = case x of
    Z -> y
    S x' -> S (plus x' y)
```

Aufgaben:

- implementiere Multiplikation, Potenz
- beweise die üblichen Eigenschaften (Addition, Multiplikation sind assoziativ, kommutativ, besitzen neutrales Element)

Pattern Matching in versch. Sprachen

- Scala: case classes `http://docs.scala-lang.org/tutorials/tour/case-classes.html`
- C# (7): `https://github.com/dotnet/roslyn/blob/features/patterns/docs/features/patterns.md`
- Javascript?

Nicht verwechseln mit *regular expression matching* zur String-Verarbeitung. Es geht um algebraische (d.h. baum-artige) Daten!

Übung Pattern Matching, Programme

- Für die Deklarationen

```
-- data Bool = False | True   (aus Prelude)
data T = F T | G T T T | C
```

  entscheide/bestimme für jeden der folgenden Ausdrücke:

  - syntaktisch korrekt?
  - statisch korrekt?
  - Resultat (dynamische Semantik)
  - disjunkt? vollständig?

```
1. case False of { True -> C }
2. case False of { C -> True }
3. case False of { False -> F F }
4. case G (F C) C (F C) of { G x y z -> F z }
5. case F C of { F (F x) -> False }
6. case F C of { F x -> False ; True -> False }
7. case True of { False -> C ; True -> F C }
8. case True of { False -> C ; False -> F C }
9. case C of { G x y z -> False; F x -> False; C -> True }
```

- Operationen auf Wahrheitswerten:

```
import qualified Prelude
data Bool = False | True deriving Prelude.Show
not :: Bool -> Bool -- Negation
not x = case x of
  ... -> ...
  ... -> ...
```

Syntax: wenn nach `of` kein { folgt: implizite `{ ; }` durch *Abseitsregel* (layout rule).

- `(&&) :: Bool -> Bool -> Bool`
  `x && y = case ... of ...`

  Syntax: Funktionsname

  - beginnt mit Buchstabe: steht vor Argumenten,
  - beginnt mit Zeichen: zwischen Argumenten (als Operator)

  Operator als Funktion: `(&&) False True`, Funktion als Operator: `True `f` False`.
- Listen von Wahrheitswerten:

  ```
  data List = Nil | Cons Bool List deriving Prelude.Show

  and :: List -> Bool
  and l = case l of ...
  ```

  entsprechend `or :: List -> Bool`
- (Wdhlg.) welche Signatur beschreibt binäre Bäume

  (jeder Knoten hat 2 oder 0 Kinder, die Bäume sind; es gibt keine Schlüssel)
- geben Sie die dazu äquivalente `data`-Deklaration an: `data T = ...`
- implementieren Sie dafür die Funktionen

  ```
  size  :: T -> Prelude.Int
  depth :: T -> Prelude.Int
  ```

  benutze `Prelude.+` (das ist Operator), `Prelude.min, Prelude.max`
- für Peano-Zahlen `data N = Z | S N`

  implementieren Sie *plus, mal, min, max*

4 Polymorphie

Definition, Motivation

- Beispiel: binäre Bäume mit Schlüssel vom Typ `e`

```
data Tree e = Leaf
            | Branch (Tree e) e (Tree e)
Branch Leaf True Leaf :: Tree Bool
Branch Leaf 42   Leaf :: Tree Int
```

- Definition:
  ein polymorpher Datentyp ist ein *Typkonstruktor* (= eine Funktion, die Typen auf einen Typ abbildet)
- unterscheide: `Tree` ist der Typkonstruktor, `Branch` ist ein Datenkonstruktor

Beispiele f. Typkonstruktoren (I)

- Kreuzprodukt:
  `data Pair a b = Pair a b`
- disjunkte Vereinigung:
  `data Either a b = Left a | Right b`
- `data Maybe a = Nothing | Just a`
- Haskell-Notation für Produkte:
  `(1,True)::(Int,Bool)`
  für $0, 2, 3, \ldots$ Komponenten

Beispiele f. Typkonstruktoren (II)

- binäre Bäume

```
data Bin a = Leaf
           | Branch (Bin a) a (Bin a)
```

- Listen

```
data List a = Nil
            | Cons a (List a)
```

- Bäume

```
data Tree a = Node a (List (Tree a))
```

Polymorphe Funktionen

Beispiele:

- Spiegeln einer Liste:

```
reverse :: forall e . List e -> List e
```

- Verketten von Listen mit gleichem Elementtyp:

```
append :: forall e .     List e -> List e
                       -> List e
```

Knotenreihenfolge eines Binärbaumes:

```
preorder :: forall e . Bin e -> List e
```

Def: der Typ einer polymorphen Funktion beginnt mit All-Quantoren für Typvariablen.
Bsp: Datenkonstruktoren polymorpher Typen.

Bezeichnungen f. Polymorphie

```
data List e = Nil | Cons e (List e)
```

- `List` ist ein *Typkonstruktor*
- `List e` ist ein *polymorpher* Typ
  (ein Typ-Ausdruck mit *Typ-Variablen*)
- `List Bool` ist ein *monomorpher* Typ
  (entsteht durch *Instantiierung*: Substitution der Typ-Variablen durch Typen)
- polymorphe Funktion: `reverse:: forall e . List e -> List e`
  monomorphe Funktion: `xor:: List Bool -> Bool`
  polymorphe Konstante: `Nil::forall e. List e`

Operationen auf Listen (I)

```
data List a = Nil | Cons a (List a)
```

- ```
  append xs ys = case xs of
        Nil        ->
        Cons x xs' ->
  ```
- Übung: formuliere und beweise: `append` ist assoziativ.
- ```
  reverse xs = case xs of
        Nil        ->
        Cons x xs' ->
  ```
- beweise:

  ```
  forall xs . reverse (reverse xs) == xs
  ```

Operationen auf Listen (II)

Die vorige Implementierung von `reverse` ist (für einfach verkettete Listen) nicht effizient.

Besser ist:

```
reverse xs = rev_app xs Nil
```

mit Spezifikation

```
rev_app xs ys = append (reverse xs) ys
```

Übung: daraus die Implementierung von `rev_app` ableiten

```
rev_app xs ys = case xs of ...
```

Operationen auf Bäumen

```
data List e = Nil | Cons e (List e)
data Bin e = Leaf | Branch (Bin e) e (Bin e)
```

Knotenreihenfolgen

- ```
  preorder :: forall e . Bin e -> List e
  preorder t = case t of ...
  ```
  ```

- entsprechend `inorder`, `postorder`
- und Rekonstruktionsaufgaben

Adressierug von Knoten (`False` = links, `True` = rechts)

- `get :: Tree e -> List Bool -> Maybe e`
- `positions :: Tree e -> List (List Bool)`

Übung Polymorphie

Geben Sie alle Elemente dieser Datentypen an:

- `Maybe ()`
- `Maybe (Bool, Maybe ())`
- `Either (Bool,Bool) (Maybe (Maybe Bool))`

Operationen auf Listen:

- append, reverse, rev_app

Operationen auf Bäumen:

- preorder, inorder, postorder, (Rekonstruktion)
- get, (positions)

Kochrezept: Objektkonstruktion

Aufgabe (Bsp): `x :: Either (Maybe ()) (Pair Bool ())`
Lösung (Bsp):

- der Typ `Either a b` hat Konstruktoren `Left a | Right b`. Wähle `Right b`.
  Die Substitution für die Typvariablen ist `a = Maybe (), b = Pair Bool ()`.
  `x = Right y` mit `y :: Pair Bool ()`
- der Typ `Pair a b` hat Konstruktor `Pair a b`.
  die Substitution für diese Typvariablen ist `a = Bool, b = ()`.
  `y = Pair p q` mit `p :: Bool, q :: ()`

- der Typ `Bool` hat Konstruktoren `False | True`, wähle `p = False`. der Typ `()` hat Konstruktor `()`, also `q=()`

Insgesamt `x = Right y = Right (Pair False ())`

Vorgehen (allgemein)

- bestimme den Typkonstruktor
- bestimme die Substitution für die Typvariablen
- wähle einen Datenkonstruktor
- bestimme Anzahl und Typ seiner Argumente
- wähle Werte für diese Argumente nach diesem Vorgehen.

Kochrezept: Typ-Bestimmung

Aufgabe (Bsp.) bestimme Typ von `x` (erstes Arg. von `get`):

```
at :: Position -> Tree a -> Maybe a
at p t = case t of
  Node f ts -> case p of
    Nil -> Just f
    Cons x p' -> case get x ts of
      Nothing -> Nothing
      Just t' -> at p' t'
```

Lösung:

- bestimme das Muster, durch welches `x` deklariert wird.
  Lösung: `Cons x p' ->`
- bestimme den Typ diese Musters
  Lösung: ist gleich dem Typ der zugehörigen *Diskriminante* `p`
- bestimme das Muster, durch das `p` deklariert wird
  Lösung: `at p t =`
- bestimme den Typ von `p`
  Lösung: durch Vergleich mit Typdeklaration von `at` (`p` ist das erste Argument) `p :: Position`, also `Cons x p' :: Position = List N`, also `x :: N`.

Vorgehen zur Typbestimmung eines Namens:

- finde die Deklaration (Muster einer Fallunterscheidung oder einer Funktionsdefinition)
- bestimme den Typ des Musters (Fallunterscheidung: Typ der Diskriminante, Funktion: deklarierter Typ)

Statische Typisierung und Polymorphie

Def: dynamische Typisierung:

- die Daten (zur Laufzeit des Programms, im Hauptspeicher) haben einen Typ

Def: statische Typisierung:

- Bezeichner, Ausdrücke (im Quelltext) haben einen Type (zur Übersetzungszeit bestimmt).
- für *jede* Ausführung des Programms gilt: der statische Typ eines Ausdrucks ist gleich dem dynamischen Typ seines Wertes

Bsp. für Programm ohne statischen Typ (Javascript)

```
function f (x) {
  if (x>0) { return function () { return 42; } }
  else { return "foobar"; } }
```

Dann: Auswertung von `f(1)()` ergibt 42, Auswertung von `f(0)()` ergibt Laufzeit-Typfehler.

entsprechendes Haskell-Programm ist statisch fehlerhaft

```
f x = case x > 0 of
  True  -> \ () -> 42
  False -> "foobar"
```

Nutzen der statischen Typisierung:

- beim Programmieren: Entwurfsfehler werden zu Typfehlern, diese werden zur Entwurfszeit automatisch erkannt ⇒ früher erkannte Fehler lassen sich leichter beheben
- beim Ausführen: es gibt keine Lauzeit-Typfehler ⇒ keine Typprüfung zur Laufzeit nötig, effiziente Ausführung

Nutzen der Polymorphie:

- Flexibilität, nachnutzbarer Code, z.B. Anwender einer Collection-Bibliothek legt Element-Typ fest (Entwickler der Bibliothek kennt den Element-Typ nicht)
- gleichzeitig bleibt statische Typsicherheit erhalten

Von der Spezifikation zur Implementierung (I)

Bsp: Addition von Peano-Zahlen `data N = Z | S N`

```
plus :: N -> N -> N
```

aus der Typdeklaration wird abgeleitet:

```
plus x y = case x of
  Z    ->
  S x' ->
```

erster Zweig: `plus Z y = 0 + y = y`

zweiter Zweig : `plus (S x') y = (1 + x') + y =`

mit Assoziativität von + gilt `... = 1 + (x' + y) = S (plus x' y)`

Bsp. (Ü): Multiplikation. Hinweis: benutze Distributivgesetz.

Von der Spezifikation zur Implementierung (II)

Bsp: homogene Listen `data List a = Nil | Cons a (List a)`

Aufgabe: implementiere `maximum :: List N -> N`

Spezifikation:

```
maximum (Cons x1 Nil) = x1
maximum (append xs ys) = max (maximum xs) (maximum ys)
```

- substitutiere `xs = Nil`, erhalte

```
  maximum (append Nil ys) = maximum ys
= max (maximum Nil) (maximum ys)
```

d.h. `maximum Nil` sollte das neutrale Element für `max` (auf natürlichen Zahlen) sein, also `0` (geschrieben `Z`).

- substitutiere `xs = Cons x1 Nil`, erhalte

```
  maximum (append (Cons x1 Nil) ys)
  = maximum (Cons x1 ys)
= max (maximum (Cons x1 Nil)) (maximum ys)
  = max x1 (maximum ys)
```

Damit kann der aus dem Typ abgeleitete Quelltext

```
maximum :: List N -> N
maximum xs = case xs of
  Nil        ->
  Cons x xs' ->
```

ergänzt werden.

Vorsicht: für `min, minimum` funktioniert das nicht so, denn `min` hat für `N` kein neutrales Element.

5 Funktionen

Funktionen als Daten

- bisher: Programm ist Regel(menge), `f x = 2 * x + 5`
- jetzt: Programm ist Lambda-Term

  ```
  f = \ x -> 2 * x + 5
  ```

  λ-Terme: mit lokalen Namen
- Funktionsanwendung: Substitition

  (der freien Vorkommen von x)

  $(\lambda x.B)A \to B[x := A]$
- λ-Kalkül: Alonzo Church 1936, Henk Barendregt 198*

Der Lambda-Kalkül

... als weiteres Berechnungsmodell,
(vgl. Termersetzungssysteme, Turingmaschine, Random-Access-Maschine)
Syntax: die Menge der Lambda-Terme Λ ist

- jede Variable ist ein Term: $v \in V \Rightarrow v \in \Lambda$
- Funktionsanwendung (Applikation):

  $F \in \Lambda, A \in \Lambda \Rightarrow (FA) \in \Lambda$
- Funktionsdefinition (Abstraktion):

  $v \in V, B \in \Lambda \Rightarrow (\lambda v.B) \in \Lambda$

Semantik: eine Relation $\rightarrow_\beta$ auf Λ
(vgl. $\rightarrow_R$ für Termersetzungssystem R)

Freie und gebundene Variablen(vorkommen)

- Das Vorkommen von $v \in V$ an Position p in Term t heißt *frei*, wenn „darüber kein $\lambda v. \ldots$ steht“
- Def. $\mathrm{fvar}(t) =$ Menge der in t frei vorkommenden Variablen (definiere durch strukturelle Induktion)
- Eine Variable x heißt in A *gebunden*, falls A einen Teilausdruck $\lambda x.B$ enthält.
- Def. $\mathrm{bvar}(t) =$ Menge der in t gebundenen Variablen

Bsp: $\mathrm{fvar}(x(\lambda x.\lambda y.x)) = \{x\}$, $\mathrm{bvar}(x(\lambda x.\lambda y.x)) = \{x, y\}$,

Semantik des Lambda-Kalküls: Reduktion $\rightarrow_\beta$

Relation $\rightarrow_\beta$ auf Λ (ein Reduktionsschritt)
Es gilt $t \rightarrow_\beta t'$, falls

- $\exists p \in \mathrm{Pos}(t)$, so daß
- $t[p] = (\lambda x.B)A$ mit $\mathrm{bvar}(B) \cap \mathrm{fvar}(A) = \emptyset$
- $t' = t[p := B[x := A]]$

  dabei bezeichnet $B[x := A]$ ein Kopie von B, bei der jedes freie Vorkommen von x durch A ersetzt ist

Ein (Teil-)Ausdruck der Form $(\lambda x.B)A$ heißt *Redex*. (Dort kann weitergerechnet werden.)

Ein Term ohne Redex heißt *Normalform*. (Normalformen sind Resultate von Rechnungen.)

Semantik . . . : gebundene Umbenennung $\rightarrow_\alpha$

- Relation $\rightarrow_\alpha$ auf Λ, beschreibt *gebundene Umbenennung* einer lokalen Variablen.
- Beispiel $\lambda x.fxz \rightarrow_\alpha \lambda y.fyz$.

  (f und z sind frei, können nicht umbenannt werden)
- Definition $t \rightarrow_\alpha t'$:
  - $\exists p \in \mathrm{Pos}(t)$, so daß $t[p] = (\lambda x.B)$
  - $y \notin \mathrm{bvar}(B) \cup \mathrm{fvar}(B)$
  - $t' = t[p := \lambda y.B[x := y]]$
- wird angewendet, um $\mathrm{bvar}(B) \cap \mathrm{fvar}(A) = \emptyset$ in Regel für $\rightarrow_\beta$ zu erfüllen.

  Bsp: betrachte den unterstrichenen Redex in

  $(\lambda x.\underline{((\lambda f.(\lambda x.(x + f8)))(\lambda y.(x + y)))})3$

Umbenennung von lokalen Variablen

```
int x = 3;
int f(int y) { return x + y; }
int g(int x) { return (x + f(8)); }
// g(5) => 16
```

Darf `f(8)` ersetzt werden durch $f[y := 8]$? - Nein:

```
int x = 3;
int g(int x) { return (x + (x+8)); }
// g(5) => 18
```

Das freie x in $(x + y)$ wird fälschlich gebunden.

Lösung: lokal umbenennen

```
int g(int z) { return (z + f(8)); }
```

dann ist Ersetzung erlaubt

```
int x = 3;
int g(int z) { return (z + (x+8)); }
// g(5) => 16
```

Lambda-Terme: verkürzte Notation

- Applikation ist links-assoziativ, Klammern weglassen:

$$(\ldots((FA_1)A_2)\ldots A_n) \sim FA_1A_2\ldots A_n$$

  Beispiel: $((xz)(yz)) \sim xz(yz)$

  Wirkt auch hinter dem Punkt: $(\lambda x.xx)$ bedeutet $(\lambda x.(xx))$ — und nicht $((\lambda x.x)x)$

- geschachtelte Abstraktionen unter ein Lambda schreiben:

$$(\lambda x_1.(\lambda x_2.\ldots(\lambda x_n.B)\ldots)) \sim \lambda x_1x_2\ldots x_n.B$$

  Beispiel: $\lambda x.\lambda y.\lambda z.B \sim \lambda xyz.B$

Ein- und mehrstellige Funktionen

eine einstellige Funktion zweiter Ordnung:

```
f = \ x -> ( \ y -> ( x*x + y*y ) )
```

Anwendung dieser Funktion:

```
(f 3) 4 = ...
```

Kurzschreibweisen (Klammern weglassen):

```
f = \ x y -> x * x + y * y ; f 3 4
```

Übung:
gegeben `t = \ f x -> f (f x)`
bestimme `t succ 0, t t succ 0, t t t succ 0, t t t t succ 0, ...`

Typen

für nicht polymorphe Typen: tatsächlicher Argumenttyp muß mit deklariertem Argumenttyp übereinstimmen:

wenn $f :: A \to B$ und $x :: A$, dann $(fx) :: B$.

bei polymorphen Typen können der Typ von $f :: A \to B$ und der Typ von $x :: A'$ Typvariablen enthalten.

Beispiel: $\lambda x.x :: \forall t.t \to t$.

Dann müssen A und A' nicht übereinstimmen, sondern nur *unifizierbar* sein (eine gemeinsame Instanz besitzen).

Beispiel: $(\lambda x.x)$True

benutze Typ-Substitution $\sigma = \{(t, \text{Bool})\}$.

Bestimme allgemeinsten Typ von $t = \lambda fx.f(fx))$, von (tt).

Beispiel für Typ-Bestimmung

Aufgabe: bestimme den allgemeinsten Typ von $\lambda fx.f(fx)$

- Ansatz mit Typvariablen $f :: t_1, x :: t_2$
- betrachte (fx): der Typ von f muß ein Funktionstyp sein, also $t_1 = (t_{11} \to t_{12})$ mit neuen Variablen t_{11}, t_{12}. Dann gilt $t_{11} = t_2$ und $(fx) :: t_{12}$.
- betrachte $f(fx)$. Wir haben $f :: t_{11} \to t_{12}$ und $(fx) :: t_{12}$, also folgt $t_{11} = t_{12}$. Dann $f(fx) :: t_{12}$.
- betrachte $\lambda x.f(fx)$. Aus $x :: t_{12}$ und $f(fx) :: t_{12}$ folgt $\lambda x.f(fx) :: t_{12} \to t_{12}$.
- betrachte $\lambda f.(\lambda x.f(fx))$. Aus $f :: t_{12} \to t_{12}$ und $\lambda x.f(fx) :: t_{12} \to t_{12}$ folgt $\lambda fx.f(fx) :: (t_{12} \to t_{12}) \to (t_{12} \to t_{12})$

Verkürzte Notation für Typen

- Der Typ-Pfeil ist *rechts-assoziativ*:

  $T_1 \to T_2 \to \cdots \to T_n \to T$ bedeutet $(T_1 \to (T_2 \to \cdots \to (T_n \to T) \cdots))$

- das paßt zu den Abkürzungen für mehrstellige Funktionen:

  $\lambda(x :: T_1).\lambda(x :: T_2).(B :: T)$

  hat den Typ $(T_1 \to (T_2 \to B))$,

  mit o.g. Abkürzung $T_1 \to T_2 \to T$.

Lambda-Ausdrücke in C#

- Beispiel (Fkt. 1. Ordnung)

```
Func<int,int> f = (int x) => x*x;
f (7);
```

- Übung (Fkt. 2. Ordnung) — ergänze alle Typen:

```
???  t = (??? g) => (??? x) => g (g (x));
t (f)(3);
```

- Anwendungen bei Streams, später mehr

```
(new int[]{3,1,4,1,5,9}).Select(x => x * 2);
(new int[]{3,1,4,1,5,9}).Where(x => x > 3);
```

- Übung: Diskutiere statische/dynamische Semantik von

```
(new int[]{3,1,4,1,5,9}).Select(x => x > 3);
(new int[]{3,1,4,1,5,9}).Where(x => x * 2);
```

Lambda-Ausdrücke in Java(8)

funktionales Interface (FI): hat genau eine Methode

Lambda-Ausdruck („burger arrow“) erzeugt Objekt einer anonymen Klasse, die FI implementiert.

```
interface I { int foo (int x); }
I f = (x)-> x+1;
System.out.println (f.foo(8));
```

vordefinierte FIs:

```
import java.util.function.*;

Function<Integer,Integer> g = (x)-> x*2;
   System.out.println (g.apply(8));
Predicate<Integer> p = (x)-> x > 3;
   if (p.test(4)) { System.out.println ("foo"); }
```

Lambda-Ausdrücke in Javascript

```
$ node

> let f = function (x){return x+3;}
undefined

> f(4)
7

> ((x) => (y) => x+y) (3) (4)
7

> ((f) => (x) => f(f(x))) ((x) => x+1) (0)
2
```

Beispiele Fkt. höherer Ord.

- Haskell-Notation für Listen:
  ```
  data List a = Nil | Cons a (List a)
  data   [a]  =  []  |      a : [a]
  ```
- Verarbeitung von Listen:
  ```
  filter :: (a -> Bool) -> [a] -> [a]
  takeWhile :: (a -> Bool) -> [a] -> [a]
  partition :: (a -> Bool) -> [a] -> ([a],[a])
  ```
- Vergleichen, Ordnen:
  ```
  nubBy :: (a -> a -> Bool) -> [a] -> [a]
  data Ordering = LT | EQ | GT
  minimumBy
    :: (a -> a -> Ordering) -> [a] -> a
  ```

Übung Lambda-Kalkül

- Wiederholung: konkrete Syntax, abstrakte Syntax, Semantik
- $S = \lambda xyz.xz(yz), K = \lambda ab.a$, Normalform von $SKKc$
- (mit `data N=Z|S N`) bestimme Normalform von $ttSZ$ für $t = \lambda fx.f(fx)$,

- definiere Λ als algebraischen Datentyp `data L = ...` (3 Konstruktoren)

  implementiere `size :: L -> Int`, `depth :: L -> Int`.

  implementiere `bvar :: L -> S.Set String`, `fvar :: L -> S.Set String`,

  siehe Folie mit Definitionen und dort angegebene Testfälle

  benutze `import qualified Data.Set as S`, API-Dokumentation: `https://hackage.haskell.org/package/containers/docs/Data-Set.html`

- autotool-Aufgaben Lambda-Kalkül

Übung Fkt. höherer Ordnung

- Typisierung, Beispiele in Haskell, C#, Java, Javascript

```
compose ::
compose = \ f g -> \ x -> f (g x)
```

- Implementierung von takeWhile, dropWhile

6 Rekursionsmuster

Rekursion über Bäume (Beispiele)

```
data Tree a = Leaf
            | Branch (Tree a) a (Tree a)
summe :: Tree Int -> Int
summe t = case t of
  Leaf -> 0
  Branch l k r -> summe l + k + summe r
preorder :: Tree a -> List a
preorder t = case t of
  Leaf -> Nil
  Branch l k r ->
    Cons k (append (preorder l) (preorder r))
```

Rekursion über Bäume (Schema)

```
f :: Tree a -> b
f t = case t of
  Leaf -> ...
  Branch l k r -> ... (f l) k (f r)
```

dieses Schema *ist* eine Funktion höherer Ordnung:

```
fold :: ( ... ) -> ( ... ) -> ( Tree a -> b )
fold leaf branch = \ t -> case t of
  Leaf -> leaf
  Branch l k r ->
     branch (fold leaf branch l)
       k (fold leaf branch r)
summe = fold 0 ( \ l k r -> l + k + r )
```

Rekursion über Listen

```
and :: List Bool -> Bool
and xs = case xs of
  Nil -> True ; Cons x xs' -> x && and xs'
length :: List a -> N
length xs = case xs of
  Nil -> Z ; Cons x xs' ->  S (length xs')

fold :: b -> ( a -> b -> b ) -> List a -> b
fold nil cons xs = case xs of
    Nil -> nil
    Cons x xs' -> cons x ( fold nil cons xs' )
and = fold True (&&)
length = fold Z ( \ x y ->  S y)
```

Rekursionsmuster (Prinzip)

```
data List a = Nil | Cons a (List a)
fold ( nil :: b ) ( cons :: a -> b -> b )
   :: List a -> b
```

Rekursionsmuster anwenden
= jeden Konstruktor durch eine passende Funktion ersetzen
= (Konstruktor-)Symbole *interpretieren* (durch Funktionen)
= eine *Algebra* angeben.

```
length  = fold Z   ( \ _ l -> S l )
reverse = fold Nil ( \ x ys ->      )
```

Rekursionsmuster (Merksätze)
aus dem Prinzip *ein Rekursionsmuster anwenden = jeden Konstruktor durch eine passende Funktion ersetzen* folgt:

- Anzahl der Muster-Argumente = Anzahl der Konstruktoren (plus eins für das Datenargument)
- Stelligkeit eines Muster-Argumentes = Stelligkeit des entsprechenden Konstruktors
- Rekursion im Typ ⇒ Rekursion im Muster
  (Bsp: zweites Argument von `Cons`)
- zu jedem rekursiven Datentyp gibt es *genau ein* passendes Rekursionsmuster

Rekursion über Listen (Übung)
das vordefinierte Rekursionsschema über Listen ist:

```
foldr :: (a -> b -> b) -> b -> ([a] -> b)

length = foldr ( \ x y -> 1 + y ) 0
```

Beachte:

- Argument-Reihenfolge (erst cons, dann nil)
- foldr nicht mit foldl verwechseln (foldr ist das „richtige“)

Aufgaben:

- `append, reverse, concat, inits, tails` mit `foldr` (d. h., ohne Rekursion)

Weitere Beispiele für Folds

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

fold :: ...
```

- Anzahl der Blätter
- Anzahl der Verzweigungsknoten
- Summe der Schlüssel
- die Tiefe des Baumes
- der größte Schlüssel

Rekursionsmuster (Peano-Zahlen)

```
data N = Z | S N

fold :: ...
fold z s n = case n of
    Z    ->
    S n' ->

plus  = fold ...
times = fold ...
```

Übung Rekursionsmuster

- Rekursionsmuster `foldr` für Listen benutzen (filter, takeWhile, append, reverse, concat, inits, tails)
- Rekursionmuster für Peano-Zahlen hinschreiben und benutzen (plus, mal, hoch, Nachfolger, Vorgänger, minus)
- Rekursionmuster für binäre Bäume mit Schlüsseln *nur in den Blättern* hinschreiben und benutzen
- Rekursionmuster für binäre Bäume mit Schlüsseln *nur in den Verzweigungsknoten* benutzen für rekursionslose Programme für:

  - Anzahl der Branch-Knoten ist ungerade (nicht zählen!)
  - Baum (`Tree a`) erfüllt die AVL-Bedingung
    Hinweis: als Projektion auf die erste Komponente eines `fold`, das Paar von `Bool` (ist AVL-Baum) und `Int` (Höhe) berechnet.
  - Baum (`Tree Int`) ist Suchbaum (ohne `inorder`)
    Hinweis: als Projektion. Bestimme geeignete Hilfsdaten.

- Wende die Vorschrift zur Konstruktion des Rekursionsmusters an auf den Typ
  - `Bool`
  - `Maybe a`

  Jeweils:
  - Typ und Implementierung
  - Testfälle
  - gibt es diese Funktion bereits? Suche nach dem Typ mit `https://www.stackage.org/lts-5.17/hoogle`

Stelligkeit von Funktionen

- ist `fold` in `double` richtig benutzt? Ja!

```
fold :: r -> (r -> r) -> N -> r
double :: N -> N
double = fold Z (\ x -> S (S x))
double (S (S Z)) :: N
```

- beachte Unterschied zwischen:
  - Term-Ersetzung: Funktionssymbol $\rightarrow$ Stelligkeit
    abstrakter Syntaxbaum: Funktionss. über Argumenten
  - Lambda-Kalkül: nur einstellige Funktionen
    AST: Applikationsknoten, Funkt.-Symb. links unten.
    Simulation mehrstelliger Funktionen wegen
    Isomorphie zwischen $(A \times B) \rightarrow C$ und $A \rightarrow (B \rightarrow C)$
- `case`: Diskriminante u. Muster müssen `data`-Typ haben

Nützliche Funktionen höherer Ordnung

- `compose :: (b -> c) -> (a -> b) -> a -> c`
  aus dem Typ folgt schon die Implementierung!

  ```
  compose f g x = ...
  ```

  diese Funktion in der Standard-Bibliothek:
  der Operator `.` (Punkt)

- `apply :: (a -> b) -> a -> b`
  `apply f x = ...`
  das ist der Operator `$` (Dollar) . . . ist *rechts-assoziativ*

- `flip :: ...`
  `flip f x y = f y x`
  wie lautet der (allgemeinste) Typ?

Argumente für Rekursionsmuster finden

Vorgehen zur Lösung der Aufgabe:
„Schreiben Sie Funktion $f : T \to R$ als `fold`“

- eine Beispiel-Eingabe ($t \in T$) notieren (Baum zeichnen)
- für jeden Teilbaum s von t, der den Typ T hat:
  den Wert von $f(s)$ in (neben) Wurzel von s schreiben
- daraus Testfälle für die Funktionen ableiten,
  die Argumente des Rekursionsmusters sind.

Beispiel: `data N = Z | S N`,
f : `N` $\to$ `Bool`, $f(x) =$ „x ist ungerade“

Nicht durch Rekursionmuster darstellbare Fkt.

- Beispiel: `data N = Z | S N`,
  f : `N` $\to$ `Bool`, $f(x) =$ „x ist durch 3 teilbar“
- wende eben beschriebenes Vorgehen an,

- stelle fest, daß die durch Testfälle gegebene Spezifikation nicht erfüllbar ist
- Beispiel: binäre Bäume mit Schlüssel in Verzweigungsknoten,
  f : `Tree k` $\rightarrow$ `Bool`,
  $f(t) =$ „t ist höhen-balanciert (erfüllt die AVL-Bedingung)“

Darstellung als fold mit Hilfswerten

- f : `Tree k` $\rightarrow$ `Bool`,
  $f(t) =$ „t ist höhen-balanciert (erfüllt die AVL-Bedingung)“
  ist nicht als fold darstellbar
- g : `Tree k` $\rightarrow$ `Pair Bool Int`
  $g(t) = (f(t), \texttt{height}(t))$
  ist als fold darstellbar

Spezialfälle des Fold

- jeder Konstruktor durch sich selbst ersetzt, mit unveränderten Argumenten: *identische* Abbildung

```
data List a = Nil | Cons a (List a)
fold :: r -> (a -> r -> r) -> List a -> r
fold Nil Cons (Cons 3 (Cons 5 Nil))
```

- jeder Konstruktor durch sich,
  mit transformierten Argumenten:

```
fold Nil (\x y -> Cons (not x) y)
  (Cons True (Cons False Nil))
```

  struktur-erhaltende Abbildung. Diese heißt *map*.

Weitere Übungsaufgaben zu Fold

- ```
  data List a = Nil | Cons a (List a)
  fold :: r -> (a -> r -> r) -> List a -> r
  ```
- schreibe mittels `fold` (ggf. verwende `map`)
  - ```
    inits, tails :: List a -> List (List a)
    inits [1,2,3] = [[],[1],[1,2],[1,2,3]]
    tails [1,2,3] = [[1,2,3],[2,3],[3],[]]
    ```
  - ```
    filter :: (a -> Bool) -> List a -> List a
    filter odd [1,8,2,7,3] = [1,7,3]
    ```
  - ```
    partition :: (a -> Bool) -> List a
                 -> Pair (List a) (List a)
    partition odd [1,8,2,7,3]
      = Pair [1,7,3] [8,2]
    ```

7 Objektorientierte Entwurfmuster

Definition, Geschichte

- Ziel: flexibel wiederverwendbarer sicherer Quelltext
- Lösung: *Funktionen höherer Ordnung*
- Simulation davon im OO-Paragidma: *Entwurfsmuster*

  wir wollen: Funktion als Datum (z.B. Lambda-Ausdruck), wir konstruieren: Objekt, das zu einer (anonymen) Klasse gehört, die diese Funktion als Methode enthält.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster (design patterns)* — Elemente wiederverwendbarer objektorientierter Software, Addison-Wesley 1996.

Beispiel Strategie-Muster

- Aufgabe: Sortieren einer Liste bzgl. wählbarer Ordnung auf Elementen.
- Lösung (in `Data.List`)

```
data Ordering = LT | EQ | GT
sortBy :: (a -> a -> Ordering) -> List a -> List a
```

(Ü: implementiere durch unbalancierten Suchbaum)

- Simulation (in `java.util.*`)

```
interface Comparator<T> { int compare(T x, T y); }
static <T> void sort(List<T> list,Comparator<T> c);
```

hier ist `c` ein *Strategie-Objekt*

8 Algebraische Datentypen in OOP

Kompositum: Motivation

- Bsp: Gestaltung von zusammengesetzten Layouts.
  Modell als algebraischer Datentyp:

```
data Component = JButton { ... }
               | Container (List Component)
```

- Simulation durch Entwurfsmuster *Kompositum*:
  - `abstract class Component`
  - `class JButton extends Component`
  - `class Container extends Component`
  - `    { void add (Component c); }`

Kompositum: Beispiel

```
public class Composite  {
  public static void main(String[] args) {
    JFrame f = new JFrame ("Composite");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container c = new JPanel (new BorderLayout());
    c.add (new JButton ("foo"), BorderLayout.CENTER);
    f.getContentPane().add(c);
    f.pack(); f.setVisible(true);
  }
}
```

Übung: geschachtelte Layouts bauen, vgl. `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws06/informatik/manage/`

Kompositum: Definition

- Definition: *Kompositum* = algebraischer Datentyp (ADT)
- ADT `data T = .. | C .. T ..`
  als Kompositum:
  - Typ `T` ⇒ gemeinsame Basisklasse (interface)
  - jeder Konstruktor `C` ⇒ implementierende Klasse
  - jedes Argument des Konstruktors ⇒ Attribut der Klasse
  - diese Argumente können `T` benutzen (rekursiver Typ)

(Vorsicht: Begriff und Abkürzung nicht verwechseln mit *abstrakter* Datentyp = ein Typ, dessen Datenkonstruktoren wir *nicht* sehen)

Binäre Bäume als Komposita

- Knoten sind *innere* (Verzweigung) und *äußere* (Blatt).
- Die richtige Realisierung ist Kompositum

```
interface Tree<K>;
class Branch<K> implements Tree<K>;
class Leaf<K> implements Tree<K>;
```

- Schlüssel: in allen Knoten, nur innen, nur außen.

der entsprechende algebraische Datentyp ist:

```
data Tree k = Leaf { ... }
  | Branch { left :: Tree k , ...
           , right :: Tree k }
```

Übung: Anzahl aller Blätter, Summe aller Schlüssel (Typ?), der größte Schlüssel (Typ?)

Kompositum-Vermeidung

Wenn Blätter keine Schlüssel haben, geht es musterfrei?

```
class Tree<K> {
   Tree<K> left; K key; Tree<K> right;
}
```

Der entsprechende algebraische Datentyp ist

```
data Tree k =
     Tree { left :: Maybe (Tree k)
          , key :: k
          , right :: Maybe (Tree k)
          }
```

erzeugt in Java das Problem, daß …

Übung: betrachte Implementierung in `java.util.Map<K,V>`

Maybe = Nullable

Algebraischer Datentyp (Haskell):

```
data Maybe a = Nothing | Just a
```

`http://hackage.haskell.org/packages/archive/base/latest/doc/html/Prelude.html#t:Maybe`

In Sprachen mit Verweisen (auf Objekte vom Typ `O`) gibt es häufig auch „Verweis auf kein Objekt“— auch vom Typ `O`. Deswegen *null pointer exceptions*.

Ursache ist Verwechslung von `Maybe a` mit `a`.

Trennung in C#: `Nullable<T>` (für primitive Typen `T`)

`http://msdn.microsoft.com/en-us/library/2cf62fcy.aspx`

Alg. DT und Pattern Matching in Scala

`http://scala-lang.org`

algebraische Datentypen:

```
abstract class Tree[A]
case class Leaf[A](key: A) extends Tree[A]
case class Branch[A]
    (left: Tree[A], right: Tree[A])
        extends Tree[A]
```

pattern matching:

```
def size[A](t: Tree[A]): Int = t match {
    case Leaf(k) => 1
    case Branch(l, r) => size(l) + size(r)
  }
```

beachte: Typparameter in eckigen Klammern

9 Objektorientierte Rekursionsmuster

Plan

- algebraischer Datentyp = Kompositum

  (Typ ⇒ Interface, Konstruktor ⇒ Klasse)

- Rekursionsschema = Besucher (Visitor)

  (Realisierung der Fallunterscheidung)

(Zum Vergleich von Java- und Haskell-Programmierung)
sagte bereits Albert Einstein: *Das Holzhacken ist deswegen so beliebt, weil man den Erfolg sofort sieht.*

Kompositum und Visitor

Definition eines Besucher-Objektes (für Rekursionsmuster mit Resultattyp `R` über `Tree<A>`) entspricht einem Tupel von Funktionen

```
interface Visitor<A,R> {
  R leaf(A k);
  R branch(R x, R y);  }
```

Empfangen eines Besuchers: durch jeden Teilnehmer des Kompositums

```
interface Tree<A> { ..
  <R> R receive (Visitor<A,R> v);  }
```

- Implementierung
- Anwendung (Blätter zählen, Tiefe, Spiegelbild)

Aufgabe: Besucher für Listen

Schreibe das Kompositum für

```
data List a = Nil | Cons a (List a)
```

und den passenden Besucher. Benutze für

- Summe, Produkt für `List<Integer>`
- Und, Oder für `List<Boolean>`
- Wert als gespiegelte Binärzahl (LSB ist links)

  Bsp: `[1,1,0,1] ==> 11`

Quelltexte aus Vorlesung (Eclipse-Projekt)

Repository: `https://gitlab.imn.htwk-leipzig.de/waldmann/fop-ss17`, Pfad im Repository: `java`.

10 Polymorphie

Arten der Polymorphie

- generische Polymorphie: erkennbar an Typvariablen
  zur *Übersetzungszeit* werden Typvariablen durch konkrete Typen substituiert,
- dynamische Polymorphie ($\approx$ Objektorientierung)
  erkennbar an `implements` zw. Klasse und Schnittstelle
  zur *Laufzeit* wird Methodenimplementierung ausgewählt

moderne OO-Sprachen (u.a. Java, C#) bieten *beide* Formen der Polymorphie

mit statischer Sicherheit (d.h. statische Garantie, daß zur Laufzeit keine Methoden fehlen)

Java-Notation f. generische Polymorphie

generischer *Typ* (Typkonstruktor):

- Deklaration der Typparameter: `class C<S,T> {..}`
- bei Benutzung Angabe der Typargumente (Pflicht):

```
{ C<Boolean,Integer> x = ... }
```

statische generische *Methode*:

- Deklaration: `class C { static <T> int f(T x) }`
- Benutzung: `C.<Integer>f (3)`

  Typargumente können auch inferiert werden.

(Übung: Angabe der Typargumente für polymorphe nicht statische Methode)

Beispiel f. dynamische Polymorphie

```
interface I { int m (); }
class A implements I
    { int m () { return 0; }}
class B implements I
    { int m () { return 1; }}
I x =    // statischer Typ von x ist I
    new A(); // dynamischer Typ ist hier A
System.out.println (x.m());
x = new B(); // dynamischer Typ ist jetzt B
System.out.println (x.m());
```

- statischer Typ: eines Bezeichners im Programmtext
- dynamischer Typ: einer Stelle im Speicher

Klassen, Schnittstellen und Entwurfsmuster

- FP-Sichtweise: Entwurfsmuster = Fkt. höherer Ordnung
- OO-Sichtweise: E.M. = nützliche Beziehung zw. Klassen

  ... die durch Schnittstellen ausgedrückt wird.

  $\Rightarrow$ Verwendung von konkreten Typen (Klassen) *ist ein Code Smell*, es sollen soweit möglich abstrakte Typen (Schnittstellen) sein. (Ü: diskutiere `IEnumerable`)
- insbesondere: in Java (ab 8):

  funkionales Interface = hat genau eine Methode

  eine implementierende anonyme Klasse kann als Lambda-Ausdruck geschrieben werden

Erzwingen von Abstraktionen

- `interface I { .. }`
  `class C implements I { .. } ;`

  Wie kann `C x = new C()` verhindert werden,

  und `I x = new C()` erzwungen?

- Ansatz: `class C { private C() { } }`

  aber dann ist auch `I x = new C()` verboten.

- Lösung: Fabrik-Methode

```
class C { ..
  static I make () { return new C (); } }
```

Das Fabrik-Muster

```
interface I { }
class A implements I { A (int x) { .. } }
class B implements I { B (int x) { .. } }
```

die Gemeinsamkeit der Konstruktoren kann nicht in `I` ausgedrückt werden.

```
interface F // abstrakte Fabrik
   { I construct (int x); }
class FA implements F // konkrete Fabrik
{ I construct (int x) { return new A(x); } }
class FB implements F { .. }
main () {
   F f = Eingabe ? new FA() : new FB();
   I o1=f.construct(3); I o2=f.construct(4);
```

Typklassen in Haskell: Überblick

- in einfachen Anwendungsfällen:

  Typklasse in Haskell $\sim$ Schnittstelle in OO:

  beschreibt Gemeinsamkeit von konkreten Typen

- – Bsp. der Typ hat eine totale Ordnung
  Haskell: `class Ord a`, Java: `interface Comparable<E>`
  – Bsp. der Typ besitzt Abbildung nach `String`
  Haskell `class Show a`, Java?
- unterschiedliche Benutzung und Implementierung
  Haskell - statisch, OO - dynamisch

Beispiel

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
sortBy ( \ x y -> ... ) [False, True, False]
```

Kann mit Typklassen so formuliert werden:

```
class Ord a where
    compare :: a -> a -> Ordering
sort :: Ord a => [a] -> [a]
instance Ord Bool where compare x y = ...
sort [False, True, False]
```

- `sort` hat *eingeschränkt polymorphen Typ*
- die Einschränkung (das Constraint `Ord a`) wird in ein zusätzliches Argument (eine Funktion) übersetzt. Entspricht OO-Methodentabelle, liegt aber *statisch* fest.

Unterschiede Typklasse/Interface (Bsp)

- Typklasse/Schnittstelle `class Show a where show :: a -> String` `interface Show`
- Instanzen/Implementierungen `data A = .. ; instance Show A where ..`
  `class A implements Show { .. }` entspr. für `B`
- in Java ist `Show` ein Typ: `static String showList(List<Show> xs) { .. }`
  `showList (Arrays.asList (new A(),new B()))`
  in Haskell ist `Show` ein Typconstraint und kein Typ: `showList :: Show a => List a -> St`
  `showList [A,B]` ist Typfehler

Typklassen können mehr als Interfaces

in Java, C#, ... kann Schnittstelle (interface) in Deklarationen wie Typ (class) benutzt werden, das ist

1. praktisch, aber nur 2. soweit es eben geht

- (?) Fkt. mit > 1 Argument, Bsp. compareTo, `static <T extends Comparable<? super T> void sort(List<T> list)`
- (–) Beziehungen zwischen mehreren Typen, `class Autotool problem solution`
- (–) Typkonstruktorklassen, `class Foldable c where toList :: c a -> [a]; data Tree a = ..; instance Foldable Tree`

  (wichtig für fortgeschrittene Haskell-Programmierung)

Grundwissen Typklassen

- Typklasse schränkt statische Polymorphie ein

  (Typvariable darf nicht beliebig substitutiert werden)
- Einschränkung realisiert durch *Wörterbuch*-Argument

  (W.B. = Methodentabelle, Record von Funktionen)
- durch Instanz-Deklaration wird Wörterbuch erzeugt
- bei Benutzung einer eingeschränkt polymorphen Funktion: passendes Wörterbuch wird statisch bestimmt
- nützliche, häufige Typklassen: `Show`, `Read`, `Eq`, `Ord`.

  (`Test.SmallCheck.Serial`, `Foldable`, `Monad`,...)
- Instanzen automatisch erzeugen mit `deriving`

Übung Polymorphie

- Deklarationen von `compare` (Haskell), `compareTo` (Java), `(==)` (Haskell), `equals` (Java)
- Unterschiede feststellen und begründen

```
new Integer(3).compareTo(new Boolean(true))
compare (3 :: Integer) (True :: Bool)
new Integer(3).equals(new Boolean(true))
(3 :: Integer) == (True :: Bool)
```

- ```
  data C = D | E deriving (Eq, Show)
  instance Ord C where compare x y = ...
  ```
- nach Java übersetzen:

  ```
  data Pair a b = Pair { first :: a, second :: b }
  ```

  Testfall für Konstruktor-Aufruf und Zugriff auf Komponenten
- implementiere `equals` als die mathematische Gleichheit von Paaaren
- implementiere

  ```
  swap :: Pair a b -> Pair b a
  p :: Pair Int Bool ; p = Pair 3 False ; q = swap p
  ```

  Testfälle: z.B.

  ```
  p.swap().swap().equals(p)
  ```
- implementiere `compareTo` als die lexikografische Ordnung von Paaraen

11 Verzögerte Auswertung (lazy evaluation)

Motivation: Datenströme
Folge von Daten:

- erzeugen (producer)
- transformieren
- verarbeiten (consumer)

aus softwaretechnischen Gründen diese drei Aspekte im Programmtext trennen,
aus Effizienzgründen in der Ausführung verschränken (bedarfsgesteuerte Transformation/Erzeugung)
  ```

Bedarfs-Auswertung, Beispiele

- Unix: Prozesskopplung durch Pipes

  ```
  cat foo.text | tr ' ' '\n' | wc -l
  ```

  Betriebssystem (Scheduler) simuliert Nebenläufigkeit
- OO: Iterator-Muster

  ```
  Enumerable.Range(0,10).Select(n=>n*n).Sum()
  ```

  ersetze Daten durch Unterprogr., die Daten produzieren

- FP: lazy evaluation (verzögerte Auswertung)

  ```
  let nats = nf 0 where nf n = n : nf (n + 1)
  sum $ map ( \ n -> n * n ) $ take 10 nats
  ```

  Realisierung: Termersetzung $\Rightarrow$ Graphersetzung,

Beispiel Bedarfsauswertung

```
data Stream a = Cons a (Stream a)
nats :: Stream Int ; nf :: Int -> Stream Int
nats = nf 0 ; nf n = Cons n (nf (n+1))
head (Cons x xs) = x ; tail (Cons x xs) = xs
```

Obwohl `nats` unendlich ist, kann Wert von `head (tail (tail nats))` bestimmt werden:

```
= head (tail (tail (nf 0)))
= head (tail (tail (Cons 0 (nf 1))))
= head (tail (nf 1))
= head (tail (Cons 1 (nf 2)))
= head (nf 2) = head (Cons 2 (nf 3)) = 2
```

es wird immer ein *äußerer* Redex reduziert
(Bsp: `nf 3` ist ein *innerer* Redex)

Strictness

zu jedem Typ T betrachte $T_\perp = \{\perp\} \cup T$
dabei ist $\perp$ ein „Nicht-Resultat vom Typ T“

- Exception `undefined :: T`
- oder Nicht-Termination `let { x = x } in x`

Def.: Funktion f heißt *strikt*, wenn $f(\perp) = \perp$.
Fkt. f mit n Arg. heißt *strikt in* i,
falls $\forall x_1 \ldots x_n : (x_i = \perp) \Rightarrow f(x_1, \ldots, x_n) = \perp$
verzögerte Auswertung eines Arguments $\Rightarrow$ Funktion ist dort nicht strikt
einfachste Beispiele in Haskell:

- Konstruktoren (Cons,...) sind nicht strikt,
- Destruktoren (head, tail,...) sind strikt.

Beispiele Striktheit

- `length :: [a] -> Int` ist strikt:

```
length undefined   ==>  exception
```

- `(:) :: a->[a]->[a]` ist nicht strikt im 1. Argument:

```
length (undefined : [2,3]) ==> 3
```

d.h. (`undefined : [2,3]`) ist nicht $\perp$

- `(&&)` ist strikt im 1. Arg, nicht strikt im 2. Arg.

```
undefined && True   ==> (exception)
False && undefined  ==> False
```

Implementierung der verzögerten Auswertung

Begriffe:

- *nicht strikt*: nicht zu früh auswerten
- verzögert (*lazy*): höchstens einmal auswerten (ist Spezialfall von *nicht strikt*)

bei jedem Konstruktor- und Funktionsaufruf:

- kehrt *sofort* zurück
- Resultat ist *thunk* (Paar von Funktion und Argument)
- thunk wird erst bei Bedarf ausgewertet
- Bedarf entsteht durch Pattern Matching
- nach Auswertung: thunk durch Resultat überschreiben
  (das ist der Graph-Ersetzungs-Schritt)
- bei weiterem Bedarf: wird Resultat nachgenutzt

Bedarfsauswertung in Scala

```
def F (x : Int) : Int = {
      println ("F", x) ; x*x
}
lazy val a = F(3);
println (a);
println (a);
```

`http://www.scala-lang.org/`

Diskussion

- John Hughes: *Why Functional Programming Matters*, 1984 `http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html`
- Bob Harper 2011 `http://existentialtype.wordpress.com/2011/04/24/the-real-point-of-laziness/`
- Lennart Augustsson 2011 `http://augustss.blogspot.de/2011/05/more-points-for` `html`

Anwendungen der verzögerten Auswertg. (I)

Abstraktionen über den Programm-Ablauf

- Nicht-Beispiel (warum funktioniert das nicht in Java?)
  (mit `jshell` ausprobieren)

```
<R> R wenn (boolean b, R x, R y)
  { if (b) return x; else return y; }
int f (int x)
  { return wenn(x<=0,1,x*f(x-1)); }
f (3);
```

- in Haskell geht das (direkt in `ghci`)

```
let wenn b x y = if b then x else y
let f x = wenn (x<= 0) 1 (x * f (x-1))
f 3
```

Anwendungen der verzögerten Auswertg. (II)

unendliche Datenstrukturen

- Modell:

```
data Stream e = Cons e (Stream e)
```

- man benutzt meist den eingebauten Typ `data [a] = [] | a : [a]`
- alle anderen Anwendungen des Typs `[a]` sind *falsch*
  (z.B. als Arrays, Strings, endliche Mengen)
  mehr dazu: `https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/list-or-not-list/`

Primzahlen

```
primes :: [ Int ]
primes = sieve ( enumFrom 2 )

enumFrom :: Int -> [ Int ]
enumFrom n = n : enumFrom ( n+1 )

sieve :: [ Int ] -> [ Int ]
sieve (x : xs) = x : ys
```

wobei `ys` = die nicht durch `x` teilbaren Elemente von `xs`
(Das ist (sinngemäß) das Code-Beispiel auf `https://www.haskell.org/`)

Aufgaben zu Striktheit

- Beispiel 1: untersuche Striktheit der Funktion

```
f :: Bool -> Bool -> Bool
f x y = case y of { False -> x ; True  -> y }
```

  Antwort:

  - f ist nicht strikt im 1. Argument,
    denn `f undefined True = True`
  - f ist strikt im 2. Argument,
    denn dieses Argument (`y`) ist die Diskriminante der obersten Fallunterscheidung.

- Beispiel 2: untersuche Striktheit der Funktion

```
g :: Bool -> Bool -> Bool -> Bool
g x y z =
  case (case y of False -> x ; True -> z) of
    False -> x
    True  -> False
```

  Antwort (teilweise)

  - ist strikt im 2. Argument, denn die Diskriminante (`case y of ..`) der obersten Fallunterscheidung verlangt eine Auswertung der inneren Diskriminante `y`.

- Aufgabe: strikt in welchen Argumenten?

```
f x y z = case y && z of
  False -> case x || y of
    False -> z
    True  -> False
  True -> y
```

Übung: Rekursive Stream-Definitionen

Bestimmen Sie jeweils die ersten Elemente dieser Folgen (1. auf Papier durch Umformen, 2. mit ghci).

Für die Hilfsfunktionen (`map`, `zipWith`, `concat`): 1. Typ feststellen, 2. Testfälle für endliche Listen

1. `f = 0 : 1 : f`
2. `n = 0 : map (\ x -> 1 + x) n`
3. `xs = 1 : map (\ x -> 2 * x) xs`
4. ```
   ys = False
      : tail (concat (map (\y -> [y,not y]) ys))
   ```
5. `zs = 0 : 1 : zipWith (+) zs (tail zs)`

siehe auch `https://www.imn.htwk-leipzig.de/~waldmann/etc/stream/`

12 OO-Simulation v. Bedarfsauswertung

Motivation (Wdhlg.)

Unix:

```
cat stream.tex | tr -c -d aeuio | wc -m
```

Haskell:

```
sum $ take 10 $ map ( \ x -> x^3 ) $ naturals
```

C#:

```
Enumerable.Range(0,10).Select(x=>x*x*x).Sum();
```

- logische Trennung: Produzent → Transformator(en) → Konsument
- wegen Speichereffizienz: verschränkte Auswertung.
- gibt es bei *lazy* Datenstrukturen geschenkt, wird ansonsten durch Iterator (Enumerator) simuliert.
```

Iterator (Java)

```
interface Iterator<E> {
  boolean hasNext(); // liefert Status
  E next(); // schaltet weiter
}
interface Iterable<E> {
  Iterator<E> iterator();
}
```

typische Verwendung:

```
Iterator<E> it = c.iterator();
while (it.hasNext()) {
  E x = it.next (); ...
}
```

Abkürzung: `for (E x : c) { ... }`

Beispiele Iterator

- ein Iterator (bzw. Iterable), der/das die Folge der Quadrate natürlicher Zahlen liefert
- Transformation eines Iterators (map)
- Zusammenfügen zweier Iteratoren (merge)
- Anwendungen: Hamming-Folge, Mergesort

Beispiel Iterator Java

```
Iterable<Integer> nats = new Iterable<Integer>() {
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      private int state = 0;
      public Integer next() {
        int result = this.state;
        this.state++; return res;
      }
      public boolean hasNext() { return true; }
    }; } };
for (int x : nats) { System.out.println(x); }
```

Aufgabe: implementiere eine Methode

```
static Iterable<Integer> range(int start, int count)
```

soll `count` Zahlen ab `start` liefern.

Testfälle dafür:

- ```
  @Test
  public void t1() {
    assertEquals (new Integer(3), Main.range(3, 5).iterator().next());
  }
  @Test
  public void t2() {
    assertEquals (5, StreamSupport.stream(Main.range(3, 5).spliterator
  }
  ```

Enumerator (C#)

```
interface IEnumerator<E> {
  E Current; // Status
  bool MoveNext (); // Nebenwirkung
}
interface IEnumerable<E> {
  IEnumerator<E> GetEnumerator();
}
```

Ü: typische Benutzung (schreibe die Schleife, vgl. mit Java-Programm)

Abkürzung: `foreach (E x in c) { ... }`

Zusammenfassung Iterator

- Absicht: bedarfsweise Erzeugung von Elementen eines Datenstroms
- Realisierung: Iterator hat Zustand
  und Schnittstelle mit Operationen:
  - (1) Test (ob Erzeugung schon abgeschlossen)
  - (2) Ausliefern eines Elementes
  - (3) Zustandsänderung
- Java: 1 : `hasNext()`, 2 *und* 3: `next()`
  C#: 3 *und* 1: `MoveNext()`, 2: `Current`
```

Iteratoren mit yield

- der Zustand des Iterators ist die Position im Programm
- `MoveNext()`:
  - bis zum nächsten `yield` weiterrechnen,
  - falls das `yield return` ist: Resultat `true`
  - falls `yield break`: Resultat `false`
- benutzt das (uralte) Konzept *Co-Routine*

```
using System.Collections.Generic;
IEnumerable<int> Range (int lo, int hi) {
    for (int x = lo; x < hi ; x++) {
       yield return x;
    }
    yield break; }
```

Aufgaben Iterator C#

```
IEnumerable<int> Nats () {
    for (int s = 0; true; s++) {
        yield return s;
    }
}
```

Implementiere „das merge aus mergesort“(Spezifikation?)

```
static IEnumerable<E> Merge<E>
    (IEnumerable<E> xs, IEnumerable<E> ys)
  where E : IComparable<E>
```

zunächst für unendliche Ströme, Test: `Merge(Nats().Select(x=>x*x),Nats().Select(x=>`
(benötigt `using System.Linq` und Assembly `System.Core`)
Dann auch für endliche Ströme, Test: `Merge(new int [] {1,3,4}, new int [] {2,7,8})`
Dann Mergesort

```
static IEnumerable<E> Sort<E> (IEnumerable<E> xs)
     where E : IComparable<E> {
     if (xs.Count() <= 1) {
         return xs;
     } else { // zwei Zeilen folgen
         ...
     }
 }
```

Test: `Sort(new int [] { 3,1,4,1,5,9})`

Streams in C#: funktional, Linq

Funktional

```
IEnumerable.Range(0,10).Select(x => x^3).Sum();
```

Typ von Select? Implementierung?

Linq-Schreibweise:

```
(from x in new Range(0,10) select x*x*x).Sum();
```

Beachte: SQL-select „vom Kopf auf die Füße gestellt“.

13 Fkt. höherer Ord. für Streams

Motivation

- Verarbeitung von Datenströmen,
- durch modulare Programme,

  zusammengesetzt aus elementaren Strom-Operationen
- angenehme Nebenwirkung (1):

  (einige) elementare Operationen sind parallelisierbar
- angenehme Nebenwirkung (2):

  externe Datenbank als Datenquelle, Verarbeitung mit Syntax und Semantik (Typsystem) der Gastsprache

Strom-Operationen

- erzeugen (produzieren):
  - `Enumerable.Range(int start, int count)`
  - eigene Instanzen von `IEnumerable`
- transformieren:
  - elementweise: `Select`
  - gesamt: `Take, Skip, Where`
- verbrauchen (konsumieren):
  - `Aggregate`
  - Spezialfälle: `All, Any, Sum, Count`

Strom-Transformationen (1)

elementweise (unter Beibehaltung der Struktur)
Vorbild:

```
map :: (a -> b) -> [a] -> [b]
```

Realisierung in C#:

```
IEnumerable<B> Select<A,B>
   (this IEnumerable <A> source,
    Func<A,B> selector);
```

Rechenregeln für `map`:

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

map f (map g xs) = ...
```

Strom-Transformationen (2)

Änderung der Struktur, Beibehaltung der Elemente
Vorbild:

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
```

Realisierung: `Take, Drop, Where`
Übung: `takeWhile, dropWhile, ...`

- ausprobieren (Haskell, C#)
- implementieren

  Haskell: 1. mit expliziter Rekursion, 2. mit `fold`

  C# (Enumerator): 1. mit `Current,MoveNext`, 2. `yield`

Strom-Transformationen (3)

neue Struktur, neue Elemente
Vorbild:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

Realisierung:

`SelectMany`

Rechenregel (Beispiel):
`map f xs = xs >>= ...`

Übung:
Definition des Operators >=> durch
`(s >=> t) = \ x -> (s x >>= t)`
Typ von >=>? Assoziativität? neutrale Elemente?

Strom-Verbraucher

„Vernichtung“ der Struktur
(d. h. kann danach zur Garbage Collection, wenn keine weiteren Verweise existieren)
Vorbild:

```
fold  :: r -> (e -> r -> r) -> [e] -> r
```

in der Version „von links “

```
foldl :: (r -> e -> r) -> r -> [e] -> r
```

Realisierung (Ü: ergänze die Typen)

```
R Aggregate<E,R>
  (this IEnumerable<E> source,
     ... seed, ... func)
```

(Beachte `this`. Das ist eine *extension method*)

Zusammenfassung: Ströme

. . . und ihre Verarbeitung

C# (Linq)	Haskell
`IEnumerable<E>`	`[e]`
Select	map
SelectMany	>>= (bind)
Where	filter
Aggregate	foldl

- mehr zu Linq: `https://msdn.microsoft.com/en-us/library/system.linq(v=vs.110).aspx`
- Ü: ergänze die Tabelle um die Spalte für Streams in Java-8 `http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`

Bsp: die Spezifikation von TakeWhile

`https://msdn.microsoft.com/en-us/library/bb534804(v=vs.110).aspx`
zeigt Nachteile natürlichsprachlicher Spezifikationen:

- *ungenau*: "Return Value: ... An `IEnumerable<T>` that contains the elements from the input sequence that occur before ..."

  aber in welcher Reihenfolge? Da steht nur "contains". Also ist als Wert von `(new int [] {1,2,3,` auch `{2,1}` möglich. Oder `{1,2,1}`? Oder `{1,5,2,7}`? Alle enthalten 1 und 2.

- *unvollständig*: "... occur before the element at which the test no longer passes"

  `(new int [] {1,2,3,4}).TakeWhile(x => x<8)` Hier gibt es kein solches Element. Was nun — die Spezifikation verbietet diesen Aufruf, d.h. wenn man es doch tut, erhält man eine Exception? Oder sie gestattet ihn und erlaubt ein beliebiges Resultat?

Es wäre schon gegangen, man hätte nur wollen müssen:

"`w.TakeWhile(p)` ist der *maximale Präfix* von `w`, dessen Elemente sämtlich die Bedingung p erfüllen."

(Notation $u \sqsubseteq w$ für u *ist Präfix von* w, Def.: $\exists v : u \cdot v = w$)

korrekte Spezifikation: `w.TakeWhile(p) = u` iff

- $u \sqsubseteq w$ und $\forall y \in u : p(y)$
- und $\forall u' : (u' \sqsubseteq w \wedge (\forall y \in u' : p(y))) \Rightarrow u' \sqsubseteq u$

Arbeiten mit Collections in Haskell

Bsp: `Data.Set` und `Data.Map` aus `https://hackage.haskell.org/package/containers`

Beispiel-Funktionen mit typischen Eigenschaften:

```
unionWith
  :: Ord k => (v->v->v)->Map k v->Map k v->Map k v
fromListWith
  :: Ord k => (v->v->v) -> [(k, v)] -> Map k v
```

- polymorpher Typ, eingeschränkt durch `Ord k`
- Funktion höherer Ordnung (siehe 1. Argument)
- Konversion von/nach Listen, Tupeln

Anwendungen:

- bestimme Vielfachheit der Elemente einer Liste
- invertiere eine `Map k v` (Resultat-Typ?)

Linq-Syntax (type-safe SQL)

```
var stream = from c in cars
   where c.colour == Colour.Red
   select c.wheels;
```

wird vom Compiler übersetzt in

```
var stream = cars
   .Where (c => c.colour == Colour.Red)
   .Select (c.wheels);
```

Beachte:

- das Schlüsselwort ist `from`
- Typinferenz (mit `var`)

Übung: Ausdrücke mit mehreren `from`, mit `group .. by ..`

Linq-Syntax (type-safe SQL) (II)

```
var stream =
  from x in Enumerable.Range(0,10)
  from y in Enumerable.Range(0,x) select y
```

wird vom Compiler übersetzt in

```
var stream = Enumerable.Range(0,10)
      .SelectMany(x=>Enumerable.Range(0,x))
```

- aus diesem Grund ist `SelectMany` wichtig
- ... und die entsprechende Funktion >>= (bind) in Haskell
- deren allgemeinster Typ ist

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
```

`https://wiki.haskell.org/All_About_Monads`

Linq und Parallelität

... das ist ganz einfach: anstatt

```
var s = Enumerable.Range(1, 20000)
     .Select( f ).Sum() ;
```

schreibe

```
var s = Enumerable.Range(1, 20000)
     .AsParallel()
     .Select( f ).Sum() ;
```

Dadurch werden

- Elemente parallel verarbeitet (`.Select(f)`)
- Resultate parallel zusammengefaßt (`.Sum()`)

vgl. `http://msdn.microsoft.com/en-us/library/dd460688.aspx`

Übung Stream-Operationen

- die Funktion `reverse :: [a] -> [a]` als `foldl`
- die Funktion `fromBits :: [Bool] -> Integer`, Beispiel `fromBits [True,False,Fal`
  ... als `foldr` oder als `foldl` ?
- die Regel vervollständigen und ausprobieren:

  ```
  foldl f a (map g xs) = foldl ? ?
  ```

  das `map` verschwindet dabei $\Rightarrow$ *stream fusion* (Coutts, Leshchinsky, Stewart, 2007)
  `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.7401`
- die Regel ergänzen (autotool)

  ```
  foldr f a xs = foldl ? ? (reverse xs)
  ```

- `map` durch >>= implementieren (entspr. `Select` durch `SelectMany`)
- `filter` durch `foldr` implementieren (autotool)

Bsp: nicht durch fold darstellbare Funktion

`filter, takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]`

die ersten beiden lassen sich durch `fold` darstellen, aber `dropWhile` nicht. Beweis (indirekt):

Falls doch `dropWhile p xs = fold n c xs`, dann entsteht folgender Widerspruch:

```
   [False,True]
== dropWhile id [False,True]
== fold n c [False,True]
== c False (fold n c [True])
== c False (dropWhile id [True])
== c False []
== c False (dropWhile id [])
== c False (fold n c [])
== fold n c [False]
== dropWhile id [False]
== [ False ]
```

Ü: läßt sich `dropWhile` als `foldl` schreiben?

Fold-Left: Eigenschaften

- Beispiel: foldl $f\ s\ [x_1, x_2, x_3] = f(f(f\ s\ x_1)\ x_2)\ x_3)$
  vgl. foldr $f\ s\ [x_1, x_2, x_3] = f\ x_1\ (f\ x_2\ (f\ x_3\ s))$
- Eigenschaft:
  foldl $f\ s\ [x_1, \ldots, x_n] = f$ (foldl $f\ s\ [x_1, \ldots, x_{n-1}])\ x_n$
  vgl. foldr $f\ s\ [x_1, \ldots, x_n] = f\ x_1$ (foldr $f\ s\ [x_2, \ldots, x_n]$)
- Anwend.: bestimme `f,s` mit `reverse = foldl f s`

```
[3,2,1]=reverse [1,2,3] = foldl f s [1,2,3]
  = f (foldl f s [1,2]) 3
  = f (reverse [1,2]) 3 = f [2,1] 3
```

also `f [2,1] 3 = [3,2,1]`, d.h., `f x y = y : x`

Fold-Left: Implementierung

- Eigenschaft (vorige Folie) sollte nicht als Implementierung benutzt werden, denn $[x_1, \ldots, x_{n-1}]$ ist teuer (erfordert Kopie)
- ```
  foldl :: (b -> a -> b) -> b -> [a] -> b
  foldl f s xs = case xs of
    []     -> s
    x : xs' -> foldl f (f s x) xs'
  ```
  zum Vergleich

  ```
  foldr :: (a -> b -> b) -> b -> [a] -> b
  foldr f s xs = case xs of
    []     -> s
    x : xs' -> f x (foldl f s xs')
  ```

Fold-Left: allgemeiner Typ

- der Typ von `Prelude.foldl` ist tatsächlich

  ```
  Foldable t => (b->a->b) -> b -> t a -> b
  ```

- hierbei ist `Foldable` eine (Typ)Konstruktor-Klasse
  mit der einzigen (konzeptuell) wesentlichen Methode

  ```
  class Foldable t where toList :: t a -> [a]
  ```

  und Instanzen für viele generische Container-Typen
- weitere Methoden aus Effizienzgründen
- `https://www.reddit.com/r/haskell/comments/3okick/foldable_for_nonhaskellers_haskells_controversial/`
  ```

14 Organisatorisches

KW 26 und KW 27

- KW 26 (diese Woche)
  - VL: Bsp. fortgeschrittene Anwendung der fktl. Prog.
  - Ü: weitere Aufgaben zu Streams (foldr, foldl)
  - dazu auch autotool-Aufgabe (ab Mo. abend)
- KW 27 (letzte VL-Woche)
  - VL:
    - Zusammenfassung
    - Preisverleihung autotool-Highscore
  - Ü: Hörsaal-Übung (Hr. Wenzel).
    Fragen vorher per Mail an ihn.

15 Funktional-Reactive Programmierung

Motivation

- nebenläufige, interaktive Systeme
- imperatives Modell:
  - Komponenten haben Zustand, dieser ist veränderlich
  - Komponenten reagieren auf Zustandsänderungen in anderen Komponenten (d.h. ändern eigenen Zustand)
  - in OO ausgdrückt durch das *Beobachter*-Muster
- funktionales Modell (FRP):
  - Zustand als Funktion der Zeit
  - Ereignis $\approx$ stückweise konstanter Zustand

Verhaltensmuster: Beobachter

- Subjekt: class Observable
  - anmelden: void addObserver (Observer o)
  - abmelden: void deleteObserver (Observer o)
  - Zustandsänderung: void setChanged ()
  - Benachrichtigung: void notifyObservers(...)
- Beobachter: interface Observer
  - aktualisiere: void update (...)
- Vorteil: Objekbeziehungen sind konfigurierbar.
- Nachteil: Spaghetti-Programmierung (impliziter globaler Zustand, unübersichtliche Änderungen)
- (FRP: Komposition ohne Nebenwirkungen, Modularität)

Beobachter: Beispiel

```
public class Counter extends Observable {
    private int count = 0;
    public void step () { this.count ++;
        this.setChanged();
        this.notifyObservers();   }   }
public class Watcher implements Observer {
    private final int threshold;
    public void update(Observable o, Object arg) {
       if (((Counter)o).getCount() >= this.threshold) {
           System.out.println ("alarm");  }   }  }
public static void main(String[] args) {
    Counter c = new Counter ();
    Watcher w = new Watcher (3);
    c.addObserver(w); c.step(); c.step (); c.step (); }
```

Funktional-Reaktive Programmierung

- Verhalten (Behaviour) und Ereignisse (Events)

  als Funktionen der Zeit, repräsentiert durch Streams

- Conal Elliot, Paul Hudak: *Functional Reactive Animation*, ICFP 1997, `http://conal.net/papers/icfp97/` 2007 als „most influential paper“ ausgezeichnet
- Stephen Blackheath: FRP `https://www.manning.com/books/functional-reactive-progra` 2016

  Kapitel 1 des Buches heißt *Stop Listening!*
- React? `http://reactivex.io/`? Jein.

  Das grundsätzliche Modell von FRP benutzt reelle Zeit.

FRP-Beispiel

```
dollar <- UI.input ; euro   <- UI.input
getBody window #+ [
  column [ grid [[string "Dollar:", element dollar]
                ,[string "Euro:"  , element euro  ]]
       ]]
euroIn   <- stepper "0" $ UI.valueChange euro
dollarIn <- stepper "0" $ UI.valueChange dollar
let rate = 0.7 :: Double
    withString f = maybe "-" (printf "%.2f") . fmap f . readMay
    dollarOut = withString (/ rate) <$> euroIn
    euroOut   = withString (* rate) <$> dollarIn
element euro   # sink value euroOut
element dollar # sink value dollarOut
```

`https://github.com/HeinrichApfelmus/threepenny-gui/tree/master/samples`

16 Anwendungen von Typklassen

Testdaten für *property based testing*

- ```
  smallCheck 3 $ \ (xs::[Int]) ->
    sum xs == sum (reverse (xs))|
  ```
- Koen Classen, John Hughes: QuickCheck, ICFP 2000 (most influential paper award 2010)
- Colin Runciman et al. (2008), Rudy Matela (2017) `https://www.cs.york.ac.uk/fp/smallcheck/` `https://hackage.haskell.org/package/leancheck`
  ```

Typklassen für small/lean-check

- ```
  check :: Testable a => a -> IO ()
  class Testable a where
    results :: a -> [([String],Bool)]
  instance Testable Bool where
    results p = [([],p)]
  instance (Listable a, Testable b)
    => Testable (a -> b) where ...
  ```
- ```
  class Listable a where tiers :: [[a]]
  instance Listable Int where
   tiers = [0] : map (\x->[negate x,x]) [1..]
  ```
- Compiler beweist Aussage mittels o.g. Regeln `instance Testable (Int -> (Int -> Bool`

(Data) dependent types

- Motivation: statt `m :: List (List Int)`

  besser wäre `m :: List 2 (List 2 Int))`

  z.B. statische Sicherheit bei Matrix-Add., -Mult.
- geht aber nicht, denn 2 ist Datum (`Int` ist Typ)

  der Typ von `m` ist daten-abhängig.
- Programmierspachen mit *dependent types*:

  Cayenne (Augustsson 1998); Agda (Norell, 2008–) `http://wiki.portal.chalmers.se/agda/`; Idris (Brady, 2005–) `https://www.idris-lang.org/`

  Bei der Typ-Prüfung wird ggf. auch mit Daten gerechnet. Ist aufwendiger, aber gibt mehr statische Sicherheit

Simulation v. dependent types d. Typklassen

- benutzt *generalized* algebraic data types (GADT) (eingeschränkte Polymorphie f. Konstruktoren)
- Rechnung auf Daten zur Laufzeit $\Rightarrow$ Rechnung auf Typen mittels Typklassen zur Übersetzungszeit

- `List 2 a` ⇒ `List (Succ (Succ Zero)) a`

```
class Nat a ; data Zero;  instance Nat Zero
data Succ n ; instance Nat n => Nat (Succ n)
data List n a where
  Nil  :: List Zero a
  Cons :: a -> List n a -> List (Succ n) a
tail :: Nat n => List (Succ n) a -> List n a
tail (Cons x xs) = xs
```

- Bsp: `tail Nil` ist dann ein Typfehler

17 Zusammenfassung, Ausblick

Themen

- Terme, algebraische Datentypen (OO: Kompositum)
- Muster, Regeln, Term-Ersetzung (Progr. 1. Ordnung)
- Polymorphie, Typvariablen, Typkonstruktoren
- Funktionen, Lambda-Kalkül (Progr. höherer Ord.)
- Rekursionsmuster (fold) (OO: Visitor)
- Eingeschränkte Polymorphie (Typklassen, Interfaces)
- Streams, Bedarfsauswertung (OO: Iterator)
- Stream-Verarbeitung mit foldl, map, filter, LINQ
- Funktional-Reaktive Progr. (OO: Beobachter)

Aussagen

- statische Typisierung ⇒
  - findet Fehler zur Entwicklungszeit (statt Laufzeit)
  - effizienter Code (keine Laufzeittypprüfungen)
- generische Polymorphie: flexibler *und* sicherer Code

- Funktionen als Daten, F. höherer Ordnung $\Rightarrow$
  - ausdrucksstarker, modularer, flexibler Code

Programmierer(in) sollte

- die abstrakten Konzepte kennen
- sowie ihre Realisierung (oder Simulation) in konkreten Sprachen (er)kennen und anwenden.

Eigenschaften und Grenzen von Typsystemen

- Ziel: vollständige statische Sicherheit, d.h.
  - vollständige Spezifikation = Typ
  - Implementierung erfüllt Spezifikation
    $\Longleftrightarrow$ Implementierung ist korrekt typisiert
- Schwierigkeit: es ist nicht entscheidbar, ob die Implementierung die Spezifikation erfüllt

  (denn das ist äquivalent zu Halteproblem)
- Lösung: Programmierer schreibt Programm

  und Korrektheitsbeweis
- ... mit Werkzeugunterstützung

  zur Automatisierung trivialer Beweisschritte

Software-Verifikation (Beispiele)

- Sprachen mit *dependent types*, z.B. `http://wiki.portal.chalmers.se/agda/`
- (interaktive) Beweis-Systeme, z.B. `http://isabelle.in.tum.de/`, `https://coq.inria.fr/`
- verifizierter C-Compiler `http://compcert.inria.fr/`
- Research in Software Engineering (Spezifikations- Sprache FORMULA, Constraint-Solver Z3)

  `http://research.microsoft.com/rise`

CYP — check your proofs

- `https://github.com/noschinl/cyp#readme` „...verifies proofs about Haskell-like programs“
- ```
  Lemma: length (xs ++ ys) .=. length xs + length ys
  Proof by induction on List xs
  Case []
   To show: length ([]++ys) .=. length [] + length ys
   Proof                        length ([] ++ ys)
      (by def ++)           .=. length ys
                                length [] + length ys
      (by def length)       .=. 0 + length ys
      (by arith)            .=. length ys
    QED
  Case x:xs
    To show: length ((x : xs) ++ ys) .=. length (x : xs) + length ys
    IH: length (xs ++ ys) .=. length xs + length ys
  ```

Anwendungen der funktionalen Progr.

Beispiel: Framework Yesod `http://www.yesodweb.com/`

- “Turn runtime bugs into compile-time errors”
- “Asynchronous made easy”
- domainspezifische, statisch typisierte Sprachen für
  - Routes (mit Parametern)
  - Datenbank-Anbindung
  - Html-Generierung

Anwendung: `https://gitlab.imn.htwk-leipzig.de/autotool/all0/tree/master/yesod`

Industrielle Anwendg. d funkt. Progr.

siehe Workshops *Commercial users of functional programming* `http://cufp.org/2015/`

- siehe Adressen/Arbeitgeber der Redner der Konferenz
  ```

Diskussion:

- Amanda Laucher: An Enterprise Software Consultant's view of FP `http://cufp.org/2015/amanda-laucher-keynote.html`
- Paul Graham: Beating the Averages `http://www.paulgraham.com/avg.html`
- Joel Spolsky: `http://www.joelonsoftware.com/articles/ThePerilsofJavaSchool` `html`

Anwendungen v. Konzepten der fktl. Prog.

- `https://www.rust-lang.org/` Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.
- `https://developer.apple.com/swift/`

  ... Functional programming patterns, e.g., map and filter, ... designed for safety.
- `https://github.com/dotnet/roslyn/blob/features/patterns/docs/features/patterns.md` enable many of the benefits of algebraic data types and pattern matching from functional languages

Ein weiterer Vorzug der Fktl. Prog.

- `https://jobsquery.it/stats/language/group`

  (1. Juli 2017)