

# **Fortgeschrittene Programmierung**

## **Vorlesung**

**WS 09,10; SS 12–14, 16–18**

Johannes Waldmann, HTWK Leipzig

2. Juli 2018

# Einleitung

## Programmierung im Studium bisher

- 1. Sem: Modellierung (formale Spezifikationen (von konkreten und abstrakten Datentypen))
- 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)

# Worin besteht jetzt der Fortschritt?

- *deklarative* Programmierung  
(Programm *ist* ausführbare Spezifikation)
- insbesondere: *funktionale* Programmierung  
Def: Programm berechnet *Funktion*  
 $f : \text{Eingabe} \mapsto \text{Ausgabe}$ ,  
(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
- Ausdrucksstärke, 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] -> [Boo
```

**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/noschin1/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](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 Programming, 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)
- funktional-reaktive Programmierung (deklarative interaktive Programme)
- 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
  - Vorrechnen von Ü-Aufgaben (an Tafel o. Bildschirm), gruppenweise
  - `https://autotool.imn.htwk-leipzig.de/new/`, individuell
- Prüfungszulassung: regelmäßiges (d.h. innerhalb der jeweiligen Deadline) und erfolgreiches (jeweils insgesamt  $\geq 50\%$  der Pflichtaufgaben) Bearbeiten (und Präsentieren) der 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`
- Maurice Naftalin und Phil Wadler: *Java Generics and Collections*, O'Reilly 2006
- `http://haskell.org/` (Sprache, Werkzeuge, Tutorials), `http://book.realworldhaskell.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

- Benutzung Rechnerpool, ghci aufrufen

`http://www.imn.htwk-leipzig.de/~waldmann/etc/pool/`  
(aus Hochschulnetz bzw. VPN)

- 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 `takeWhile`? 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 Hoogole), 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/](http://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/](http://liveblogging-cufp-2013/)

# Aufgaben (Auswertung in KW 15)

1. zu: E. W. Dijkstra: *Answers to Questions from Students of Software Engineering* (Austin, 2000) (EWD 1035)
  - „putting the cart before the horse“
    - übersetzen Sie wörtlich ins Deutsche,
    - geben Sie eine entsprechende idiomatische Redewendung in Ihrer Muttersprache an,
    - wofür stehen *cart* und *horse* hier konkret?
  - sind die empfohlenen exakten Techniken der Programmierung für große Systeme anwendbar?
2. Über ein Monoid  $(M, \circ, 1)$  mit Elementen  $a, b \in M$  (sowie eventuell weiteren) ist bekannt:  $a^2 = b^2 = (ab)^2 = 1$ .  
Dabei ist  $ab$  eine Abkürzung für  $a \circ b$  und  $a^2$  für  $a \circ a$ , usw.

- Geben Sie ein Modell mit  $1 \neq a \neq b \neq 1$  an (autotool).
- Überprüfen Sie  $ab = ba$  in Ihrem Modell.
- Leiten Sie  $ab = ba$  aus den Monoid-Axiomen und gegebenen Gleichungen ab.

Das ist eine Übung zur Wiederholung der Konzepte *abstrakter* und *konkreter* Datentyp sowie *Spezifikation*.

3. im Rechnerpool live vorführen:

- ein Terminal öffnen
- `ghci` starten (in der aktuellen Version), Fakultät von 100 ausrechnen
- Datei `F.hs` mit Texteditor anlegen und öffnen, Quelltext `f = ...` (Ausdruck mit Wert 100!) schreiben, diese Datei in `ghci` laden, `f` auswerten

Dabei wg. Projektion an die Wand:

Schrift 1. groß genug und 2. schwarz auf weiß.

Vorher Bildschirm(hintergrund) aufräumen, so daß bei Projektion keine personenbezogenen Daten sichtbar werden. Beispiel: `export PS1="$ "` ändert den Shell-Prompt (versteckt den Benutzernamen).

Wer (unsinnigerweise) eigenen Rechner im Pool benutzt:

- Aufgaben wie oben *und*
- `ssh`-Login auf einen Rechner des Pools  
(damit wird die Ausrede *GHC (usw.) geht auf meinem Rechner nicht* hinfällig)
- `ssh`-Login oder remote-Desktop-Zugriff *von* einem Rechner des Pools auf Ihren Rechner (damit das

projiziert werden kann, *ohne* den Beamer umzustöpseln)

(falls das alles zu umständlich ist, dann eben doch einen Pool-Rechner benutzen)

4. welcher Typ ist zu erwarten für die Funktion, die das Spiegelbild einer Zeichenkette berechnet?

Suchen Sie nach Funktionen dieses Typs mit

`https://www.haskell.org/hoogle/`, erklären Sie einige der Resultate, welches davon ist das passende?

# Daten

## Wiederholung: Terme

- (Prädikatenlogik) *Signatur*  $\Sigma$  ist Menge von Funktionssymbolen mit Stelligkeiten  
ein Term  $t$  in Signatur  $\Sigma$  ist
  - Funktionssymbol  $f \in \Sigma$  der Stelligkeit  $k$   
mit Argumenten  $(t_1, \dots, t_k)$ , die selbst Terme sind.  
 $\text{Term}(\Sigma) =$  Menge der Terme über Signatur  $\Sigma$
- (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  $\text{Term}(\Sigma_1)$ :  
 $Z(), S(S(Z())), f(S(S(Z()))), Z()$
- Signatur:  $\Sigma_2 = \{E/0, A/1, B/1\}$
- Elemente von  $\text{Term}(\Sigma_2)$ : ...

# 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
```

Beispiele (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  $S = \{S_1, \dots\}$

- Abb. von F.-Symbol nach Typ

- $Typ$  ist Element aus  $S^* \times S$

Folge der Argument-Sorten, Resultat-Sorte

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

- $\text{Term}(\Sigma)$ : 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  $t$ .
- $\text{Pos}(t)$  = die Menge der Positionen eines Terms  $t$   
Definition: wenn  $t = f(t_0, \dots, t_{k-1})$ , d.h.,  $k$  Kinder  
dann  $\text{Pos}(t) = \{[]\} \cup \{[i] \ ++ \ p \mid 0 \leq i < k \wedge p \in \text{Pos}(t_i)\}$ .

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:  $S(f(S(S(Z())), Z()))[0, 1] = \dots$

Definition (durch Induktion über die Länge von  $p$ ):  $\dots$

- $t[p := s]$  : wie  $t$ , aber mit Term  $s$  an Position  $p$

Beispiel:  $S(f(S(S(Z())), Z()))[[0, 1] := S(Z)] = \dots$

Definition (durch Induktion über die Länge von  $p$ ):  $\dots$

# Operationen mit Variablen in Termen

- $\text{Term}(\Sigma, V)$  = Menge der Terme über Signatur  $\Sigma$  mit Variablen aus  $V$

Beispiel:  $\Sigma = \{Z/0, S/1, f/2\}$ ,  $V = \{y\}$ ,  
 $f(Z(), y) \in \text{Term}(\Sigma, V)$ .

- Substitution  $\sigma$ : partielle Abbildung  $V \rightarrow \text{Term}(\Sigma)$

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  $\rightarrow_R$  ist Menge aller Paare  $(t, t')$  mit
  - es existiert  $(l, r) \in R$
  - es existiert Position  $p$  in  $t$
  - es existiert Substitution  $\sigma : (\text{Var}(l) \cup \text{Var}(r)) \rightarrow \text{Term}(\Sigma)$
  - so daß  $t[p] = l\sigma$  und  $t' = t[p := r\sigma]$ .

# Termersetzungssysteme als Programme

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

dieses Berechnungsmodell ist im allgemeinen

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

# Konstruktor-Systeme

Für TRS  $R$  über Signatur  $\Sigma$ : 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:  $R_1 = \{a(b(x)) \rightarrow b(a(x))\}$  über  $\Sigma_1 = \{a/1, b/1\}$ ,

$R_2 = \{f(f(x, y), z) \rightarrow f(x, f(y, z))\}$  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  $t \in \text{Term}(\{f/1, g/3, c/0\})$  an mit  $t[1] = c()$ .

mit `ghci`:

- `data T = F T | G T T T | C deriving Show`  
erzeugen Sie o.g. Term  $t$  (durch Konstruktoraufrufe)
- Geben Sie  $\text{Pos}(t)$  an

Die *Größe* eines Terms  $t$  ist definiert durch

$$|f(t_0, \dots, t_{k-1})| = 1 + \sum_{i=0}^{k-1} |t_i|.$$

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

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

- für welche Substitution  $\sigma$  gilt  $f(x, Z)\sigma = f(S(Z), Z)$ ?
- für dieses  $\sigma$ : bestimmen Sie  $f(x, S(x))\sigma$ .

Notation für Termersetzungeregeln: anstatt  $(l, r)$  schreibe  $l \rightarrow r$ .

Abkürzung für Anwendung von 0-stelligen Symbolen: anstatt  $Z()$  schreibe  $Z$ . (Vorsicht: dann kann man Variablen nicht mehr von 0-stelligen Symbolen unterscheiden. Man muß dann immer die Signatur explizit angeben oder auf andere Weise vereinbaren, wie man Variablen erkennt, z.B.

„Buchstaben am Ende des Alphabetes ( $\dots, x, y, \dots$ ) sind Variablen“, das ist aber riskant)

- Für  $R = \{f(S(x), y) \rightarrow f(x, S(y)), f(Z, y) \rightarrow y\}$   
bestimme alle  $R$ -Normalformen von  $f(S(Z), S(Z))$ .

- für  $R_d = R \cup \{d(x) \rightarrow f(x, x)\}$   
bestimme alle  $R_d$ -Normalformen von  $d(d(S(Z)))$ .

- Bestimme die Signatur  $\Sigma_d$  von  $R_d$ .

Bestimme die Menge der Terme aus  $\text{Term}(\Sigma_d)$ , die  $R_d$ -Normalformen sind.

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

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

# Hausaufgaben (Diskussion in KW16)

## 1. autotool-Aufgabe 15-2.

Allgemeine Hinweise zur Bearbeitung von Haskell-Lückentext-Aufgaben:

- Schreiben Sie den angezeigten Quelltext (vollständig! ohne zusätzliche Leerzeichen am Zeilenanfang!) in eine Datei mit Endung `.hs`, starten Sie `ghci` mit diesem Dateinamen als Argument
- ändern Sie den Quelltext: ersetzen Sie `undefined` durch einen geeigneten Ausdruck, hier z.B.  

```
solution = S.fromList [ False, G ]
```

im Editor speichern, in `ghci` neu laden (`:r`)
- reparieren Sie Typfehler, werten Sie geeignete Terme

aus, hier z.B. `s.size solution`

- werten Sie `test` aus, wenn `test` den Wert `True` ergibt, dann tragen Sie die Lösung in autotool ein.

2. Geben Sie einen Typ (eine `data`-Deklaration) mit genau 100 Elementen an. Sie können andere Data-Deklarationen benutzen (wie in autotool-Aufgabe). Minimieren Sie die Gesamtzahl aller deklarierten Konstruktoren.

3. Vervollständigen Sie die Definition der *Tiefe* von Termen:

$$\text{depth}(f()) = 0$$

$$k > 0 \Rightarrow \text{depth}(f(t_0, \dots, t_{k-1})) = \dots$$

- Bestimmen Sie  $\text{depth}(\sqrt{a \cdot a + b \cdot b})$
- Beweisen Sie  $\forall \Sigma : \forall t \in \text{Term}(\Sigma) : \text{depth}(t) \leq |t| - 1$ .
- Für welche Terme gilt hier die Gleichheit?

#### 4. autotool-Aufgabe 15-1

#### 5. (Zusatz-Aufgabe) für die Signatur $\{A/2, D/0\}$ :

- definiere Terme  $t_0 = D, t_{i+1} = A(t_i, D)$ .  
Zeichne  $t_3$ . Bestimme  $|t_i|, \text{depth}(t_i)$ .
- für  $S = \{A(A(D, x), y) \rightarrow A(x, A(x, y))\}$   
bestimme  $S$ -Normalform(en), soweit existieren, der  
Terme  $t_2, t_3, t_4$ .  
Geben Sie für  $t_2$  die ersten Ersetzungs-Schritte explizit  
an.
- Normalform von  $t_i$  allgemein.







# Programme

## Funktionale Programme

... sind spezielle Term-Ersetzungssysteme. Beispiel:

Signatur:  $S$  einstellig,  $Z$  nullstellig,  $f$  zweistellig.

Ersetzungssystem  $\{f(Z, y) \rightarrow y, f(S(x'), y) \rightarrow S(f(x', y))\}$ .

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  $\rightarrow_R^*$

Resultat = Normalform (hat keine  $\rightarrow_R$ -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  $\sigma$  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)

**Beispiele** (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.

# 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!

# 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)

# Spezifikation und Test

Bsp: Addition von Peano-Zahlen

- Spezifikation:

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

- Axiome (Bsp): `plus` ist assoziativ und kommutativ

- Test der Korrektheit durch

- Aufzählen einzelner Testfälle

```
plus (S (S Z)) (S Z) == plus (S Z) (S (S Z))
```

- Notieren von Eigenschaften (*properties*)

```
plus_comm :: N -> N -> Bool
```

```
plus_comm x y = plus x y == plus y x
```

und automatische typgesteuerte Testdatenerzeugung

```
Test.LeanCheck.checkFor 10000 plus_comm
```

# Spezifikation und Verifikation

Beweis für: Addition von Peano-Zahlen ist assoziativ

- zu zeigen ist

$$\text{plus } a \ (\text{plus } b \ c) == \text{plus } (\text{plus } a \ b) \ c$$

- Beweismethode: Induktion (nach  $x$ )  
und Umformen mit Gleichungen (äquiv. zu Implement.)

$$\text{plus } Z \ y = y$$

$$\text{plus } (S \ x') \ y = S \ (\text{plus } x' \ y)$$

- **Anfang:**  $\text{plus } Z \ (\text{plus } b \ c) == \dots$

- **Schritt:**  $\text{plus } (S \ a') \ (\text{plus } b \ c) ==$   
 $== S \ (\text{plus } a' \ (\text{plus } b \ c)) == \dots$

# 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 -> Fal`
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 -> Fal`

- **Operationen auf Wahrheitswerten:**

```
import qualified Prelude
data Bool = False | True deriving Prelude.S
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 P
```

```
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*

# Hausaufgaben (für KW 17)

## 1. Für die Deklarationen

```
-- data Bool = False | True      (aus Prelude)
data S = A Bool | B | C S S
```

entscheide/bestimme für jeden der folgenden Ausdrücke:

- syntaktisch korrekt?
- Resultat-Typ (statische Semantik)
- Resultat-Wert (dynamische Semantik)
- Menge der Muster ist: disjunkt? vollständig?

```
1. case False of { True -> B }
2. case False of { B -> True }
3. case C B B of { A x -> x }
```

4. `case A True of { A x -> False }`

5. `case A True of { A x -> False ; True ->`

6. `case True of { False -> A ; True -> A Fa`

7. `case True of { False -> B ; False -> A F`

8. `case B of { C x y -> False; A x -> x; B`

2. für selbst definierte Wahrheitswerte (vgl.

Übungsaufgabe): deklarieren, implementieren und testen  
Sie eine zweistellige Funktion „exklusiv-oder“ (mit Namen  
`xor`)

3. für binäre Bäume ohne Schlüssel (vgl. Übungsaufgabe):  
deklarieren, implementieren und testen Sie ein  
einstelliges Prädikat über solchen Bäumen, das genau

dann wahr ist, wenn das Argument eine gerade Anzahl von Blättern enthält.

4. Peano-Zahlen: siehe autotool und:

Beweisen Sie, daß unsere Implementierung der Addition kommutativ ist. Hinweis: dazu ist ein Hilfssatz nötig, in dessen Behauptung  $\mathbb{Z}$  vorkommt.

bg

# 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, ... 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
```

- Verketteten 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: Datenkonstrukturen 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'           ->`
- **Ü: formuliere, teste und beweise: `append` ist assoziativ.**

- `reverse xs = case xs of`  
    `Nil                   ->`  
    `Cons x xs'           ->`

- **Ü: beweise:**

```
forall xs ys : reverse (append xs ys)  
  == append (reverse ys) (reverse xs)
```

# 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) (m
```

- **substituiere** `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`).

- **substituiere** `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.

# Operationen auf Listen (II)

- Die vorige Implementierung von `reverse` ist (für einfach verkettete Listen) nicht effizient (sondern quadratisch, vgl. <https://accidentallyquadratic.tumblr.com/> )

- Besser ist Verwendung einer Hilfsfunktion

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

mit Spezifikation

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

- noch besser ist es, *keine* Listen zu verwenden

<https://www.imn.htwk-leipzig.de/~waldmann/etc/untutorial/list-or-not-list/>

# 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**

## Adressierung von Knoten (`False = links`, `True = rechts`)

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

# 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 Typ, dieser wird zur Übersetzungszeit (d.h., ohne Programmausführung) 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 stat. Typisierung und Polymorphie

- Nutzen der statischen Typisierung:
  - beim Programmieren: Entwurfsfehler werden zu Typfehlern, diese werden zur Entwurfszeit automatisch erkannt  $\Rightarrow$  früher erkannte Fehler lassen sich leichter beheben
  - beim Ausführen: keine Laufzeit-Typfehler  $\Rightarrow$  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

# Konstruktion von Objekten eines Typs

Aufgabe (Bsp):

$x :: \text{Either } (\text{Maybe } ()) \ (\text{Pair Bool } ())$

Lösung (Bsp):

- der Typ  $\text{Either } a \ b$  hat Konstruktoren  $\text{Left } a \mid \text{Right } b$ . Wähle  $\text{Right } b$ .

Die Substitution für die Typvariablen ist

$a = \text{Maybe } (), \ b = \text{Pair Bool } ()$ .

$x = \text{Right } y$  mit  $y :: \text{Pair Bool } ()$

- der Typ  $\text{Pair } a \ b$  hat Konstruktor  $\text{Pair } a \ b$ .

die Substitution für diese Typvariablen ist

$a = \text{Bool}, \ b = ()$ .

$y = \text{Pair } p \ q \text{ mit } p :: \text{Bool}, q :: ()$

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

Insgesamt  $x = \text{Right } y = \text{Right } (\text{Pair } \text{False } ())$

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.

# Bestimmung des Typs eines Bezeichners

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 dieser Muster

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)

# Übung Polymorphie

Geben Sie alle Elemente dieser Datentypen an:

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

Operationen auf Listen:

- `append`, `reverse`, `rev_app`

Operationen auf Bäumen:

- `preorder`, `inorder`
- `get`, `(positions)`

**Quelltexte aus Vorlesung:** <https://gitlab.imn.htwk-leipzig.de/waldmann/fop-ss18>

# Hausaufgaben (für KW 18)

1. für die folgenden Datentypen: geben Sie einige Elemente an (ghci), geben Sie die Anzahl aller Elemente an (siehe auch autotool-Aufgabe)

(a) `Maybe (Maybe Bool)`

(b) `Either (Bool, ()) (Maybe ())`

(c) `Foo (Maybe (Foo Bool))` mit

```
data Foo a = C a | D
```

(d) **(Zusatz)** `T ()` mit

```
data T a = L a | B (T (a, a))
```

2. Implementieren Sie die Post-Order Durchquerung von Binärbäumen.

(Zusatz: Level-Order. Das ist schwieriger.)

### 3. Beweisen Sie

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

Sie dürfen

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

ohne Beweis verwenden.







# Funktionen

## Funktionen als Daten

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

$$f = \lambda x \rightarrow 2 * x + 5$$

$\lambda$ -Terme: mit lokalen Namen

- Funktionsanwendung: Substitution  
(der freien Vorkommen von  $x$ )

$$(\lambda x. B) A \rightarrow B[x := A]$$

- $\lambda$ -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  $\Lambda$  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  $\Lambda$

(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  $\lambda v. \dots$  steht“
- Def.  $\text{fvar}(t)$  = 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.  $\text{bvar}(t)$  = Menge der in  $t$  gebundenen Variablen

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

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

Relation  $\rightarrow_\beta$  auf  $\Lambda$  (ein Reduktionsschritt)

Es gilt  $t \rightarrow_\beta t'$ , falls

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

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

Bsp: betrachte den unterstrichenen Redex in

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

# 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:

$$(\dots ((F A_1) A_2) \dots A_n) \sim F A_1 A_2 \dots A_n$$

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:

$$(\lambda x_1.(\lambda x_2.\dots (\lambda x_n.B) \dots)) \sim \lambda x_1 x_2 \dots x_n.B$$

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

# Ein- und mehrstellige Funktionen

eine einstellige Funktion zweiter Ordnung:

$$f = \lambda x \rightarrow (\lambda y \rightarrow (x*x + y*y))$$

Anwendung dieser Funktion:

$$(f\ 3)\ 4 = \dots$$

Kurzschreibweisen (Klammern weglassen):

$$f = \lambda x\ y \rightarrow x * x + y * y ; f\ 3\ 4$$

Übung:

gegeben  $t = \lambda f\ x \rightarrow f\ (f\ x)$

bestimme  $t\ \text{succ}\ 0, t\ t\ \text{succ}\ 0,$

$t\ t\ t\ \text{succ}\ 0, t\ t\ t\ t\ \text{succ}\ 0, \dots$

# Typen

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

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

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

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

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

Beispiel:  $(\lambda x.x)\text{True}$

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  $\lambda f x. f(fx)$

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

# Verkürzte Notation für Typen

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

$T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T$  bedeutet  
 $(T_1 \rightarrow (T_2 \rightarrow \dots \rightarrow (T_n \rightarrow T) \dots))$

- das paßt zu den Abkürzungen für mehrstellige Funktionen:

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

hat den Typ  $(T_1 \rightarrow (T_2 \rightarrow T))$ ,

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

# 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

- abstrakten Syntaxbaum und Normalform von  $SKKc$ , wobei  $S = \lambda xyz.xz(yz)$ ,  $K = \lambda ab.a$ ,
- (mit `data N=Z | S N`) bestimme Normalform von  $ttSZ$  für  $t = \lambda fx.f(fx)$ ,
- definiere  $\Lambda$  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`

**Teillösung:**

```
data L = Var String | App L L | Abs String
```

```
bvar :: L -> S.Set String
```

```
bvar t = case t of
```

```
  Var v -> S.singleton v
```

```
  App l r -> S.union (bvar l) (bvar r)
```

```
  Abs v b -> S.insert v (bvar b)
```

- den allgemeinsten Typ eines Lambda-Ausdrucks bestimmen, Beispiel

`compose ::`

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

**Musterlösung:**

- wegen `g x` muß `g :: a -> b` gelten,  
dann `x :: a` und `g x :: b`
- wegen `f (g x)` muß `f :: b -> c` gelten,  
dann `f (g x) :: c`
- dann `\ x -> f (g x) :: a -> c`
- dann  
`\ f g -> .. :: (b->c) -> (a->b) -> (a->c)`

## • Implementierung von takeWhile

```
takeWhile :: (a -> Bool) -> List a -> List a
takeWhile p xs = case xs of
  Nil -> Nil
  Cons x xs' -> case p x of
    False -> Nil
    True -> Cons x (takeWhile p xs')
```

# Hausaufgaben für KW 20

1. (autotool) Reduktion im Lambda-Kalkül
2. Normalform eines Lambda-Ausdrucks berechnen (an der Tafel, der Ausdruck wird erst dann gegeben)
3. den allgemeinsten Typ eines Lambda-Ausdrucks bestimmen (an der Tafel, der Ausdruck wird erst dann gegeben)
4. (autotool) Implementierung von `takeWhile`, `dropWhile` o.ä.
5. Beweisen Sie für diese Implementierung

```
xs=append (takeWhile p xs) (dropWhile p xs)
```









# Nützliche Funktionen höherer Ordnung

- $\text{compose} :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

aus dem Typ folgt schon die Implementierung!

`compose f g x = ...`

diese Funktion in der Standard-Bibliothek:

der Operator `.` (Punkt)

- $\text{apply} :: (a \rightarrow b) \rightarrow a \rightarrow b$

`apply f x = ...`

das ist der Operator `$` (Dollar) ... ist *rechts-assoziativ*

- $\text{flip} :: \dots$

`flip f x y = f y x`

wie lautet der (allgemeinste) Typ?

# Stelligkeit von Funktionen

- ist `plus` in `flip` richtig benutzt? Ja!

```
flip :: (a -> b -> c) -> b -> a -> c
```

```
data N = Z | S N
```

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

```
plus (S Z) (S (S Z)) ; flip plus (S Z) (S (S Z))
```

- 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

# Objektorientierte Entwurfsmuster

## Definition, Geschichte

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

- 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*, dessen Klasse die Vergleichsfunktion als (einzige) Methode enthält

# Java: Funktionale Schnittstellen

- Deklaration

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

- Benutzung (Notation bisher, anonyme innere Klasse)

```
Collections.sort(xs, new Comparator<Integer>() {  
    int compare(Integer x, Integer y) { ... } })
```

- ab Java 8

```
@FunctionalInterface Comparator<T> { .. }  
Collections.sort(xs, (x, y) -> ... )
```

ein Lambda-Ausdruck (“burger arrow”) erzeugt ein Objekt einer anonymen inneren Klasse, die ein funktionales Interface (mit genau einer Methode) implementiert

# 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 Constructoren (plus eins für das Datenargument)
- Stelligkeit eines Muster-Argumentes = Stelligkeit des entsprechenden Constructors
- Rekursion im Typ  $\Rightarrow$  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 ...
```

# 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*.

# Argumente für Rekursionsmuster finden

Vorgehen zur Lösung der Aufgabe:

„Schreiben Sie Funktion  $f : T \rightarrow R$  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,`

$f : N \rightarrow \text{Bool}$ ,  $f(x) =$  „ $x$  ist ungerade“

# Nicht durch Rekursionmuster darstellbare Fkt.

- Beispiel:  $\text{data } N = Z \mid S N,$

$$f : N \rightarrow \text{Bool}, f(x) = \text{„}x \text{ ist durch 3 teilbar“}$$

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

# Darstellung als fold mit Hilfswerten

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

# 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]`

# 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.

# Hausaufgaben für KW 21

1. (autotool) Rekursionsmuster auf Listen (inits, tails, ...)

2. Rekursionsmuster auf Bäumen

- Beweisen Sie, daß

`is_search_tree :: Tree Int -> Bool` kein fold ist.

- (autotool) diese Funktion kann jedoch als Projektion einer Hilfsfunktion

`h :: Tree Int -> (Bool, Maybe (Int, Int))`

erhalten werden, die für nichtleere Bäume auch noch das kleinste und größte Element bestimmt. Stellen Sie `h` als fold dar.

### 3. Rekursionsmuster auf Peano-Zahlen

- führen Sie Addition, Multiplikation und Potenzierung (jeweils realisiert als fold) vor
- Beweisen Sie, daß die modifizierte Vorgängerfunktion  $\text{pre} :: \mathbb{N} \rightarrow \mathbb{N}; \text{pre } z = z; \text{pre } (S \ x) = x$  kein fold ist.
- (autotool) Diese Funktion  $\text{pre}$  kann jedoch als Projektion einer geeigneten Hilfsfunktion  $h :: \mathbb{N} \rightarrow (\mathbb{N}, \mathbb{N})$  realisiert werden. Spezifizieren Sie  $h$  und geben Sie eine Darstellung von  $h$  als fold an.
- (autotool) Implementieren Sie die Subtraktion.

### 4. Wenden Sie die Vorschrift zur Konstruktion des Rekursionsmusters an auf den Typ

- Bool
- Maybe a

**Jeweils:**

- Typ und Implementierung (vorbereiteten Quelltext zeigen)
- Testfälle (in ghci vorführen)
- gibt es diese Funktion bereits? Suchen Sie nach dem Typ mit `https://www.haskell.org/hoogle/`



# 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  $T \Rightarrow$  gemeinsame Basisklasse (interface)
  - jeder Konstruktor  $C \Rightarrow$  implementierende Klasse
  - jedes Argument des Konstruktors  $\Rightarrow$  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  $\circ$ ) gibt es häufig auch „Verweis auf kein Objekt“ — auch vom Typ  $\circ$ .

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



# Objektorientierte Rekursionsmuster

## Plan

- algebraischer Datentyp = Kompositum  
(Typ  $\Rightarrow$  Interface, Konstruktor  $\Rightarrow$  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  $Tree\langle A \rangle$ )

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)

# Hausaufgaben für KW 22

1. (Zusatz, da die Folien in VL nicht besprochen wurde)  
zur Folie „Kompositum-Vermeidung“:

Das Problem bei `null` als Simulation für `Leaf` ist, daß man Blätter dann nicht richtig verarbeiten kann: Anstatt

```
Tree<K> t = ... ; int s = t.size();
```

muß man schreiben

```
Tree<K> t = ... ; int s = (null == t) ? 0 :
```

und das gilt für *jede* Methode.

Wie wird das in der Implementierung von `java.util.TreeMap<K, V>` gelöst?

Hinweis: Eclipse so konfigurieren (“attach source”), daß auch zu Quelltexten der Standardbibliothek navigiert werden kann.

## 2. Implementieren und testen Sie die Funktion

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = _
```

- (Zusatz) ... in Javascript (im Pool: ausführen mit `node`)

Bsp: Lambda-Ausdrücke in JS

```
/usr/local/waldmann/opt/node/latest/bin/node
```

```
> (x => y => x-y) (5) (3)
```

```
2
```

- ... in Java (im Pool: ausführen mit `jshell`).

Wie heißt der Typ für zweistellige Funktionen?

Welches ist dann der Typ für `flip`?

- Benutzen Sie `Collections.sort`, `flip` (vorige Teilaufgabe), `Arrays.asList`, `Integer.compare`, um eine Liste von Zahlen absteigend zu ordnen.

Beispiel (vgl. Folie Strategie-Muster und folgende)

```
/usr/local/waldmann/opt/java/latest/bin/jsh  
jshell> List<Integer> xs = Arrays.asList(3,  
xs ==> [3, 1, 2, 4]  
jshell> Collections.sort(xs, Integer::compa  
  
jshell> xs  
xs ==> [1, 2, 3, 4]
```

```
jshell> _ flip ( _ ) { _ }  
jshell> <T>Comparator<T> asComp ( _ f) { ret  
jshell> Collections.sort(xs, asComp(flip(Int
```

### 3. Java-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>`

Ergänze Quelltexte (Eclipse-Projekt)

**Repository:** <https://gitlab.imn.htwk-leipzig.de/waldmann/fop-ss18>, **Pfad im Repository:** `eclipse/fop-ss18`.

## 4. Binärzahlen:

- berechnen Sie den Wert einer Bitfolge als gespiegelte Binärzahl (LSB ist links), Bsp:  $[1, 1, 0, 1] \implies 11$ 
  - in Haskell (foldr)
  - in Java (Kompositum, Besucher wie vorige Teilaufgabe)
- Beweisen Sie, daß die entsprechende Aufgabenstellung *ohne Spiegelung* (Bsp.  $[1, 1, 0, 1] \implies 13$ ) *nicht lösbar* ist (diese Funktion besitzt keine Darstellung als foldr)



# Eingeschränkte Polymorphie

## 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 { String show (); }
```

- Instanzen/Implementierungen

```
data A = A1 ; 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 -> String
showList [A1, B1] 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 substituiert 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`

# Generische Instanzen (I)

```
class Eq a where
```

```
  (==) :: a -> a -> Bool
```

Vergleichen von Listen (elementweise)

*wenn*  $a$  in Eq, *dann*  $[a]$  in Eq:

```
instance Eq a => Eq [a] where
```

```
  l == r = case l of
```

```
    [] -> case r of
```

```
      [] -> True ; y : ys -> False
```

```
    x : xs -> case r of
```

```
      [] -> False
```

```
      y : ys -> (x == y) && ( xs == ys )
```

Übung: wie sieht `instance Ord a => Ord [a]` aus?  
(lexikografischer Vergleich)

# Generische Instanzen (II)

```
class Show a where
  show :: a -> String
instance Show Bool where
  show False = "False" ; show True = "True"
instance Show a => Show [a] where
  show xs = brackets
           $ concat
           $ intersperse ", "
           $ map show xs
instance (Show a, Show b) => Show (a,b) where
  show False = "False"
show ([True,False], True)
  = " ([True,False], True) "
```

# Benutzung von Typklassen bei Smallcheck

Colin Runciman, Matthew Naylor, Fredrik Lindblad:

*SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values*

- Testen von universellen Eigenschaften  
( $\forall a \in A : \forall b \in B : p\ a\ b$ )
- automatische Generierung der Testdaten ...
- ... aus dem Typ von  $p$
- ... mittels generischer Instanzen

http:

//hackage.haskell.org/package/smallcheck

# Test.LeanCheck—Beispiel

- `assoc op = \ a b c ->`  
    `op a (op b c) == op (op a b) c`  
`main = check`  
    `(assoc ((++) :: [Bool] -> [Bool] -> [Bool]`

- dabei werden benutzt (in vereinfachter Darstellung)

```
class Testable p where check :: p -> Bericht
```

Instanzen sind alle Typen, die testbare Eigenschaften repräsentieren

```
class Serial a where series :: Int -> [a]
```

Instanzen sind alle Typen, für die eine Aufzählung (für Elemente gegebener Größe) definiert wurde

# Testen für beliebige Stelligkeit

- warum geht eigentlich beides (einstellig, zweistellig)

```
check $ \ x -> x || not x
```

```
check $ \ x y ->
```

```
  not (x&&y) == not x || not y
```

- **weil gilt** `instance Testable (Bool -> Bool)` **und**  
`instance Testable (Bool -> (Bool -> Bool))`

- **es gibt diese Instanzen:**

```
instance Testable Bool
```

```
instance (Serial a, Testable b)
```

```
  => Testable (a -> b)
```

# Generieren von Testdaten

- `class Serial a where series :: Int -> [a]`  
`instance Serial Int where`  
`series s = [s, negate s]`
- `instance (Serial a, Serial b)`  
`=> Serial (a,b) where`  
`series d = do d1 <- [0 .. d]`  
`x1 <- series d1; x2 <- series (d-d1)`  
`return (x1, x2)`  
`cons2 c = map (\(x,y) -> c x y) (series d)`
- `instance Serial a => Serial [a] where`  
`series 0 = [[]]; series d = cons2 (:)`

# Hausaufgaben für KW 23

bei beiden Aufgaben: vorbereitete Lösung in Editor/Eclipse zeigen, mit `ghci/unit tests` vorführen.

## 1. für den Typ

```
data T p = A Bool | B p deriving (Show, Eq)
```

und mit `import Test.LeanCheck:`

### (a) implementieren Sie

```
instance Listable p => Listable (T p)
```

unter Benutzung von `cons*` und `\/`

### (b) die Eigenschaft „die (abgeleitete) Gleichheit (`==`) auf `T Int` ist symmetrisch“ formulieren und testen

### (c) zusätzlich `Ord` ableiten und testen, daß dann (`<=`) transitiv ist

(d) eine eigene `instance Ord p => Ord (T p)` schreiben, die verschieden ist von der abgeleiteten, aber trotzdem `(<=)` transitiv und antisymmetrisch (bzgl. des abgeleiteten `(==)`). Eigenschaften durch Tests nachweisen.

## 2. Übersetzen Sie nach Java

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

(in *eine* polymorphe Klasse, deren Komponenten `final` sind)

(a) Geben Sie Testfälle für Konstruktor-Aufruf und Zugriff auf Komponenten an, Bsp

```
assertEquals(false, new Pair<Integer, Boolean>...
```

(b) implementiere `equals` als die mathematische Gleichheit von Paaren, geben Sie Testfälle an

(c) implementieren Sie eine Methode äquivalent zu  
`swap :: Pair a b -> Pair b a`

**Testfälle: z.B.**

```
Pair<Integer, Boolean> p
  = new Pair<>(42, false);
assertEquals (p, p.swap().swap());
```

(d) implementieren Sie für diese Klasse

`Comparable<Pair<A, B>>` als die lexikografische Ordnung auf Paaren, mit Testfällen





# 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  $\forall x_1 \dots x_n : (x_i = \perp) \Rightarrow f(x_1, \dots, x_n) = \perp$

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-lazy-evaluation-in.html>

# Anwendg. Lazy Eval.: Ablaufsteuerung

- Nicht-Beispiel (JS hat strikte Auswertung)

```
function wenn (b, x, y) { return b ? x : y; }  
function f(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
```

- in JS simulieren (wie sieht dann f aus?)

```
function wenn (b, x, y) { return b ? x() : y() }
```

# Anwendg. Lazy Eval.: Modularität

- `foldr :: (e -> r -> r) -> r -> [e] -> r`  
`or = foldr (||) False`  
`or [ False, True, undefined ]`  
`and = not . or . map not`
- (vgl. Augustson 2011) strikte Rekursionsmuster könnte man kaum sinnvoll benutzen (zusammensetzen)
- übliche Tricks zur nicht-strikten Auswertung zur Ablaufsteuerung
  - eingebaute Syntax (if-then-else)
  - benutzerdefinierte Syntax (macros)gehen hier nicht wg. Rekursion

# Anwendg. Lazy Eval.: Streams

## 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/>)

# Semantik von `let`-Bindungen

- der Teilausdruck `undefined` wird nicht ausgewertet:

```
let { x = undefined ; y = () } in y
```

- alle Namen sind nach jedem `=` sichtbar:

```
let { x = y ; y = () } in x
```

- links von `=` kann beliebiges Muster stehen

```
let { (x, y) = (3, 4) } in x
```

```
let { (x, y) = (y, 5) } in x
```

- es muß aber passen, sonst

```
let { Just x = Nothing } in x
```

# Beispiel für Lazy Semantik in Let

- Modell: binäre Bäume wie üblich, mit `fold` dazu

```
data T k = L | B (T k) k (T k)
```

- Aufgabe 1: jeder Schlüssel soll durch Summe aller Schlüssel ersetzt werden.

```
f (B (B L 2 L) 3 L) = B (B L 5 L) 5 L
```

```
f t = let s = fold 0 (\ x y z -> x+y+z) t
      in fold L (\ x y z -> Branch x s z) t
```

- Aufgabe 2: dasselbe mit *nur einem fold*. Hinweis:

```
f t = let { (s, r) = fold _ _ t } in r
```

# Übungsaufgaben 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`.

# Hausaufgaben für KW 24

1. Aufgabe: strikt in welchen Argumenten?

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

2. 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

(a)  $f = 0 : 1 : f$

(b) `n = 0 : map (\ x -> 1 + x) n`

(c) `xs = 1 : map (\ x -> 2 * x) xs`

(d) `ys = False`

`: tail (concat (map (\y -> [y, not y]) ys))`

(e) `zs = 0 : 1 : zipWith (+) zs (tail zs)`

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

3. (autotool) Jeden Schlüssel durch Summe aller Schlüssel ersetzen mit nur einem fold.

4. (Zusatz) Algorithmus aus Appendix A aus Chris Okasaki: *Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design* (ICFP 2000) implementieren (von

where `auf let` `umschreiben`), `testen` und `erklären`











# OO-Simulation v. Bedarfsauswertung

## Motivation (Wdhlg.)

Unix:

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

Haskell:

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

C#:

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

- logische Trennung:

Produzent  $\rightarrow$  Transformator(en)  $\rightarrow$  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) { ... }`

# 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 result;  
            }  
            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(  
}
```

- @Test

```
public void t2() {  
    assertEquals (5, StreamSupport.stream(Mai  
}
```

# Weitere 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

# 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: Position im Programm und Belegung der lokalen Variablen
- `MoveNext()`:
  - bis zum nächsten `yield` weiterrechnen,
  - falls das `yield` return ist: **Resultat** `true`
  - falls `yield break`: **Resultat** `false`
- benutzt das Konzept *Co-Routine* (Wang und Dahl 1971)

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

# 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

# Motivation: Parallel-Verarbeitung

geeignete Fkt. höherer Ordnung  $\Rightarrow$  triviale Parallelisierung:

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

```
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>

# 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:

$$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

Realisierung in C#:

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

Rechenregeln für map:

$$\text{map } f \ [] = \dots$$
$$\text{map } f \ (x : xs) = \dots$$
$$\text{map } f \ (\text{map } g \ xs) = \dots$$

# 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:

$$(>>=) :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]$$

Realisierung:

`SelectMany`

Rechenregel (Beispiel):

$$\text{map } f \text{ } xs = xs >>= \dots$$

Übung:

Definition des Operators  $>=>$  durch

$$(s >=> t) = \lambda x \rightarrow (s \ x >>= t)$$

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*)

# Fold-Left: Eigenschaften

• Beispiel:  $\text{foldl } f \ s \ [x_1, x_2, x_3] = f(f(f \ s \ x_1) \ x_2) \ x_3$

vgl.  $\text{foldr } f \ s \ [x_1, x_2, x_3] = f \ x_1 (f \ x_2 (f \ x_3 \ s))$

• Eigenschaft:

$\text{foldl } f \ s \ [x_1, \dots, x_n] = f (\text{foldl } f \ s \ [x_1, \dots, x_{n-1}]) \ x_n$

vgl.  $\text{foldr } f \ s \ [x_1, \dots, x_n] = f \ x_1 (\text{foldr } f \ s \ [x_2, \dots, x_n])$

• Anwend.: bestimme  $f, s$  mit  $\text{reverse} = \text{foldl } f \ s$

$[3, 2, 1] = \text{reverse } [1, 2, 3] = \text{foldl } f \ s \ [1, 2, 3]$

$= f (\text{foldl } f \ s \ [1, 2]) \ 3$

$= f (\text{reverse } [1, 2]) \ 3 = f [2, 1] \ 3$

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  $[x_1, \dots, x_{n-1}]$  ist teuer (erfordert Kopie)

- $\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

```
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 (foldr 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/](https://www.reddit.com/r/haskell/comments/3okick/foldable_for_nonhaskellers_haskells_controversial/)

# Zusammenfassung: Ströme

... und ihre Verarbeitung

C# (Linq)	Haskell
<code>IEnumerable&lt;E&gt;</code>	<code>[e]</code>
<b>Select</b>	<b>map</b>
<b>SelectMany</b>	<b>&gt;&gt;= (bind)</b>
<b>Where</b>	<b>filter</b>
<b>Aggregate</b>	<b>foldl</b>

- **mehr zu Linq:** [https://msdn.microsoft.com/en-us/library/system.linq\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.linq(v=vs.110).aspx)
- **Ü:** ergänze die Tabelle um die Spalte für Streams in Java  
<http://docs.oracle.com/javase/10/docs/api/java/util/stream/package-summary.html>

# Hausaufgaben für KW 25

1. Übersetzen Sie dieses Programm (vollständiger Quelltext im git-repo)

```
static IEnumerable<Tuple<int,int>> rectangle  
    for (int x = 0; x<width; x++) {  
        for (int y = 0; y<height; y++) {  
            yield return new Tuple<int,int>(x,y);  
        }  
    }  
yield break;  
}
```

in einen expliziten Iterator (C# oder Java).

Verallgemeinern Sie auf

```
static IEnumerable<Tuple<A,B>> rectangle<A,
```

## 2. implementieren Sie

```
static IEnumerable<int> Merge (IEnumerable<i
```

(Spezifikation: beide Eingaben sind (schwach) monoton steigend, Ausgabe auch und ist Multimengen-äquivalent zur Vereinigung der Eingaben).

Verallgemeinern Sie von Element-Typ `int` auf einen Typparameter.

Zusatz: `Mergesort` unter Verwendung dieses `Merge`

## 3. Implementieren Sie `map`, `filter` und `rectangle` (siehe oben) nur durch `>>=` (bzw. `SelectMany` in C#)

#### 4. Definieren Sie den Operator

$$f \gg g = \lambda xs \rightarrow f xs \gg g.$$

Untersuchen Sie (in ghci vorführen, ggf. an der Tafel beweisen), ob  $\gg$

- assoziativ, kommutativ ist,
- rechts-, links-neutrale Element(e) besitzt.

#### 5. In ghci vorführen und an der Tafel beweisen:

für alle  $f, g, z, xs$  vom passenden Typ gilt:

$$\text{foldr } f \ z \ (\text{map } g \ xs) == \\ \text{foldr } (\lambda x \ y \rightarrow f \ (g \ x) \ y) \ z \ xs$$

Geben Sie eine ähnliche Beziehung für `foldl` und `map` an.

# Hinweise:

- **C#-Compiler und Interpreter**

(<http://www.mono-project.com/>) ist im Pool installiert.

- **Beispiel Interpreter:**

```
$ csharp
```

```
csharp> Enumerable.Range(0, 10).Where(x => 0 < x)
```

- **Beispiel Compiler**

```
$ cat C.cs
```

```
using System;
```

```
class C {
```

```
public static void Main(string [] argv) {  
    Console.WriteLine("hello");  
}  
  
}  
$ mcs C.cs  
$ mono C.exe
```

# Mehr zu Streams und Collections

## 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);
```

- LINQ-Syntax nach Schlüsselwort `from`  
(das steht vorn — „SQL vom Kopf auf die Füße gestellt“)
- Typinferenz (mit `var`)

# 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 allgemeinsten Typ ist

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

aber wir benutzen das hier nur für `m a = [a]`

# Bind (>>=): Definition, Beispiel

- $(\gg=) :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]$   
 $xs \gg= f = \text{concat } (\text{map } f \text{ } xs)$
- $[1 \dots 10] \gg= \backslash x \rightarrow$   
   $[x \dots 10] \gg= \backslash y \rightarrow$   
     $[y \dots 10] \gg= \backslash z \rightarrow$   
       $\text{guard } (x^2 + y^2 == z^2) \gg= \backslash \_ \rightarrow$   
         $\text{return } (x, y, z)$
- $\text{guard} :: \text{Bool} \rightarrow [()]$   
 $\text{guard } b = \text{case } b \text{ of False} \rightarrow []; \text{True} \rightarrow [()]$   
 $\text{return} :: a \rightarrow [a]; \text{return } x = [x]$
- **bind, return, guard, fold(l/r): Stream-API**

# do-Notation

- spezielle Syntax für geschachtelte bind

- ```
[1 .. 10] >>= \ x ->  
  [x .. 10] >>= \ y ->  
    [y .. 10] >>= \ z ->  
      guard (x2 + y2 == z2) >>= \ _ ->  
        return (x, y, z)
```

- ```
do x <- [1 .. 10]  
  y <- [x .. 10]  
  z <- [y .. 10]  
  guard $ x2 + y2 == z2  
  return (x, y, z)
```

# Arbeiten mit Collections in Haskell

- *bisher* haben wir Datenströme elementweise verarbeitet
  - dafür ist Bedarfsauswertung (lazy evaluation oder Iterator) passend
- *jetzt* betrachten wir Verarbeitungen von Collections als Ganzes (bulk operations)
  - (mglw. Nachteil) alle Elemente gleichzeitig im Speicher
  - bulk-Operation kann effizienter implementiert werden
- Beispiel: Durchschnitt von Mengen, repräsentiert als balancierte Suchbäume:  
  
Disjunktheit von Teilbäumen früh feststellen

# Beispiel Implementierung Mengenop.

- `data Set e = Leaf | Branch (Set e) e (Set e)`  
`intersection :: Ord e => Set e -> Set e -> Set e`

- **naive (elementweise) Realisierung**,  $\Theta(|a| \cdot \log |b|)$

```
intersection a b = S.fromList
  $ filter (\x -> S.member x b) $ S.toList a
```

- **unter Ausnutzung der Struktur**,  $\Theta(|a| \cdot (1 + \log(|b|/|a|)))$

```
i (Branch al ak ar) b =
  let (bl,present,br) = splitMember ak b
  in  if present then branch (i al bl) ak (i ar br)
     else merge (i al bl) (i ar br)
```

**Beweis der Laufzeit siehe: Guy Blelloch et al. 2016,**

<https://arxiv.org/abs/1602.02120v3>

# Mengenoperationen

- `Data.Set` (Mengen), `Data.Map` (Abbildungen mit endlichem Definitions-Bereich) aus `https://hackage.haskell.org/package/containers`
- Beispiel-Funktionen mit typischen Eigenschaften:

```
unionWith
```

```
:: Ord k => (v->v->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

# Beispiel Anwendung Mengen-Op. (I)

- die Anzahl der Vorkommen der Elemente einer Folge

```
frequencies
```

```
:: Ord e => [e] -> M.Map Int (S.Set e)
```

```
frequencies ws
```

```
= M.fromListWith ( \ x y -> _ )
```

```
$ map ( \ (w,c) -> (c, _) )
```

```
$ M.toList $ M.fromListWith ( \ x y -> _ )
```

```
$ zip xs $ repeat 1
```

- keine explizite Iteration/Rekursion, stattdessen Fkt.
  - teilw. höherer Ordnung
  - aus API von `Data.Map` (`fromListWith`, `toList`)
  - aus Standard-API (`map`, `zip`, `repeat`)
- vollständiger Quelltext siehe `fop-ss18/kw25/`

# Beispiel Anwendung Mengen-Op. (II)

- Verreinigung, Produkt (,..) von Relationen

```
type Rel s t = M.Map s (S.Set t)
```

- Vereinigung (Durchschnitt ähnlich)

```
union :: (Ord s, Ord t)  
      => Rel s t -> Rel s t -> Rel s t  
union = M.unionWith _
```

- Produkt

```
times :: _
```

keine effiziente Implementierung möglich, besseres Datenmodell nötig, siehe Übungsaufgabe

# Hausaufgaben für KW 26

1. `Data.Set` benutzt balancierte Bäume.

(a) Erzeugen Sie einen Baum, der kein AVL-Baum ist.

Hinweis:

```
import qualified Data.Set as S
putStrLn $ S.showTree $ S.fromList [ 1 ..
```

(b) Erzeugen Sie zwei unterschiedliche Bäume, die die gleiche Menge repräsentieren.

(c) Zeigen Sie die Stelle im offiziellen Quelltext, die die Balance testet bzw. herstellt.

2. Führen Sie das Beispiel „Anzahl der Vorkommen der Elemente einer Folge“ möglichst ähnlich in C#/LINQ vor.

Hinweise: `GroupBy`,

API erforschen und benutzen. Keine explizite Iteration/Rekursion. `foldl` ist gestattet, das heißt `Aggregate`.

Hinweis: benutzen Sie Original-Dokumentation (z.B. [https://msdn.microsoft.com/en-us/library/bb549218\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/bb549218(v=vs.110).aspx) und keine zweifelhafte Sekundärliteratur (Hausaufgabenwebseiten).

3. Wir betrachten diese Realisierung von Relationen auf einer Menge  $s$ :

```
type Rel s = M.Map s (S.Set s, S.Set s)
```

Dabei soll für den Wert  $(V, N)$  zum Schlüssel  $x$  gelten:

$V$  ist die Menge aller direkten Vorgänger von  $x$ ,

$N$  ist die Menge aller direkten Nachfolger von  $x$ .

Für die folgenden Operationen: jeweils

- (allgemeinsten) Typ angeben,
- Implementierung angeben,
- Testfälle angeben und vorführen:

- (a) die leere Relation, die Einer-Relation  $\{(x, y)\}$
- (b) das Einfügen eines Paares  $(x, y)$  in eine Relation
- (c) die Funktion `fromList`, die Funktion `toList`
- (d) der Durchschnitt von zwei Relationen
- (e) das Produkt von zwei Relationen

die letzte Teilaufgabe (Produkt) ist entscheidend, alles andere ist nur Vorbereitung.

4. Eine frühere Variante der vorigen Aufgabenstellung war:

... Relationen  $R \subseteq s \times t$

```
type Rel s t = M.Map s (S.Set t, S.Set t)
```

das geht aber nicht, denn damit kann man schon die  
Funktion

```
singleton :: (Ord s, Ord t) => s -> t -> Rel
```

nicht realisieren. Man muß dann stattdessen diesen Typ  
verwenden

```
type Rel s t = (M.Map s (S.Set t), M.Map t
```







# Monaden (bind, SelectMany, flatMap)

## Wiederholung: Rechnen mit Listen

- Kreuzprodukt von  $xs :: [a]$  mit  $ys :: [b]$

```
rectangle :: [a] -> [b] -> [(a,b)]
```

```
rectangle xs ys =
```

```
  xs >>= \ x -> ys >>= \ y -> return (x,y)
```

- äquiv. `do x <- xs ; y <- ys ; return (x,y)`

- `return :: a -> [a] ; return x = [x]`

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

```
xs >>= f = concat (map f xs)
```

# Die Konstruktorklasse Monad

- `class Monad c where -- Typisierung`  
    `return :: a -> c a`  
    `( >>= ) :: c a -> (a -> c b) -> c b`
- `instance Monad [] where -- Implementierung`  
    `return = \ x -> [x]`  
    `m >>= f = concat ( map f m )`
- **Axiome:** für  $f \Rightarrow g = \lambda x \rightarrow f\ x \gg= g$  gilt
  - **return ist links und rechts neutral:**  
     $(\text{return} \Rightarrow f) = f = (f \Rightarrow \text{return})$
  - **( $\Rightarrow$ ) ist assoziativ:**  
     $(f \Rightarrow (g \Rightarrow h)) = ((f \Rightarrow g) \Rightarrow h)$

# Anwendung: Rechnen mit Maybe

- `data Maybe a = Nothing | Just a`
- Anwendung: Rechnungen, die fehlschlagen können

```
case ( evaluate l ) of
  Nothing -> Nothing
  Just a   -> case ( evaluate r ) of
    Nothing -> Nothing
    Just b   -> if b == 0 then Nothing
                else Just ( div a b )
```

- äquivalent (mit passendem `>>=`), `return`, `empty`)

```
evaluate l >>= \ a ->
  evaluate r >>= \ b ->
    if b == 0 then empty else return (div a b)
```

# Maybe als Monade

- `instance Monad Maybe where`  
    `return x = Just x`  
    `m >>= f = case m of`  
        `Nothing -> Nothing ; Just x -> f x`  
`instance Alternative Maybe where empty = Nothing`
- **Ü: die Monaden-Axiome: testen (Smallcheck), beweisen.**

- **alternative Schreibweise**

```
do a <- evaluate l
    b <- evaluate r
    if b == 0 then empty else return (div a b)
```

- `Maybe a  $\approx$  [a]` mit **Länge**  $\in \{0, 1\}$

**Ü: definiert das folgende eine Monade für [ ]?**

```
xs >>= f = take 2 (concat (map f xs))
```

# Maybe in C#

- `data Maybe a = Nothing | Just a`
- **C#:** Typkonstruktor `Nullable<T>`, abgekürzt `T?`

```
Nullable<int> x = null; x.HasValue  
int? y = 7 ; y.HasValue ; y.Value  
int? a = x + y; int b = x + y ?? 8
```

- **der Operator `??` ist (`<|>`) aus Alternative**  
es gilt `a ?? (b ?? c) = (a ?? b) ?? c`
- **Ü:** Erklären Sie `Nullable<string> s = null`
- `string s = null; int? c = s?.Count()`
- **Ü:** vergleiche `Optional<T>` (Java) mit `Maybe t`

# Zustandsänderungen als Monade

- Ziel: Modellierung von Interaktionen mit der Außenwelt (Zugriffe auf Hardware, mittels des Betriebssystems)
- $a :: IO\ r$  bedeutet:  $a$  ist *Aktion* mit *Resultat*  $:: r$
- ```
readFile :: FilePath -> IO String
putStrLn :: String -> IO ()
main :: IO ()
main = readFile "foo.bar" >>= \ cs -> putStrLn cs
```
- Verkettung von Aktionen durch  $>>=$
- ein Haskell-Hauptprogramm ist ein Ausdruck 

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

, dessen Aktion wird ausgeführt
- das Typsystem trennt streng zw. *Aktion* und *Resultat*  
Bsp:  $f :: Int \rightarrow Int$  benutzt OS *garantiert* nicht

# Hausaufgaben zu Monaden (für KW 27)

1. Die Monaden-Axiome für `instance Monad Maybe` testen und beweisen.

2. Erfüllt die folgende Instanz die Monaden-Axiome?

```
return x = [x]
xs >>= f = take 2 (concat (map f xs))
```

**Hinweis:** `import Prelude hiding ((>>=))`, dann `(>>=)` wie hier. `return` wird unverändert importiert.

Die `do`-Notation wird trotzdem mittels der Original-Definition von `(>>=)` übersetzt. Das könnte man ändern durch `:set -XRebindableSyntax`.

3. `Nullable<T>` in C#: Ist ?? tatsächlich assoziativ?

Vergleichen Sie die Bedeutung von  $(a \ \ ?? \ b) \ \ ?? \ c$  und  $a \ \ ?? \ (b \ \ ?? \ c)$  unter diesen Aspekten:

- statische Semantik: Typisierung
- dynamische Semantik:
  - Resultat (für nebenwirkungsfreie  $a, b, c$ )
  - Striktheit (falls  $a, b, c$  Exceptions auslösen)
  - Nebenwirkungen (z.B., falls  $a, b, c$  Unterprogramm-Aufrufe enthalten, deren Wirkung man beobachten kann, z.B. Konsolen-Ausgabe)

Definition:

<https://github.com/dotnet/csharp-lang/blob/master/spec/expressions.md#the-null-coalescing-operator>

4. `Optional<T>` in Java: welche Methoden realisieren `bind` und `return`? Testen Sie die Monaden-Axiome in `jshell`.

# 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 ausgedrückt 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: Objektbeziehungen 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-programming](https://www.manning.com/books/functional-reactive-programming) **2016**  
Kapitel 1 dieses Buches heißt *Stop Listening!*

# FRP-Beispiel (native Anwendung mit SDL2)

- **Julian Götz:** `https://gitlab.imn.htwk-leipzig.de/jgoetz/masterprojekt`
- benutzt Bibliothek Yampa (Nilsson, Courtney, Peterson, Haskell Workshop 2002)  
`https://hackage.haskell.org/package/Yampa`
- auch zitiert in Nilsson, Chupin: Funky Grooves, PADL 2017, `http://eprints.nottingham.ac.uk/38657/`
- das zugrundeliegende Modell:
  - `Signal a = Time -> a`
  - **Signal-Funkt.** `SF a b = Signal a -> Signal b`

# Signale und Signal-Funktionen

- <https://hackage.haskell.org/package/Yampa-0.11/docs/FRP-Yampa.html>
- **Signal-Funkt.** `SF a b = Signal a -> Signal b`
- `reactimate :: m a -- Initialisierung  
-> (Bool -> m (DTime, Maybe a)) -- Eingabe  
-> (Bool -> b -> m Bool) -- Ausgabe  
-> SF a b -- Signal-Funktion  
-> m ()`  
`embed :: SF a b -> (a, [(DTime, Maybe a)])`
- `embed (sscan (+) 0) (8, [(1, Just 5), (4, Nothing)])`

# FRP-Beispiel (mit JS im Browser)

```
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
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
```

# Hausaufgaben FRP (für KW 27)

1. Signalfunktionen in Yampa (siehe Paper von Nilsson und Chupin sowie API-Doc). In ghci: `import FRP.Yampa`, testen mit `embed`, Bsp. siehe Folie.

(a) Rufen Sie die Kombinatoren

`arr`, `(>>>)`, `(&&&)`, `loop` auf.

(b) Benutzen Sie `switch`.

(c) Implementieren Sie `sscan` durch `loopPre`.

Hinweise: Der Typ dieser Kombinatoren ist teilw. allgemeiner als im Paper angegeben; verwenden Sie die Typisierung aus dem Paper. Benutzen Sie keine Arrow-Syntax.

2. Currency-Converter in threepenny-gui:

Fügen Sie ein Ausgabefeld hinzu, in dem immer die Summe der beiden jetzt gezeigten Zahlen steht.

### 3. Schützenfisch-Spiel (Yampa):

- (a) wo ist der Zeit-Balken (oben im Bild) definiert?
- (b) Erhöhen Sie die maximale Länge für eine Spielrunde.
- (c) `https://gitlab.imn.htwk-leipzig.de/jgoetz/masterprojekt/issues/4`

# Zusammenfassung, Ausblick

## Plan

- Diskussion einiger noch offener Hausaufgaben
  - (für KW 25) foldr, foldl, map
  - (für KW 26) Produkt von Relationen
- Zusammenfassung, Ausblick
- Ü diese Woche: Sie fragen. Nur sinnvoll mit Vorbereitung!
  - Nicht: „können Sie bitte noch ein Beispiel für...“,
  - sondern: „ich habe in Beispiel  $X$  bis Schritt  $Y$  gerechnet und verstehe dann Schritt  $Z$  nicht“
  - Nicht: „aber in (Hausaufgabenwebseite) steht, daß ...“

# 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)  
Beispiele: Eq, Ord sowie Testdatenerzeugung
- Striktheit, Bedarfsauswertung, Streams (OO: Iterator)
- Stream-Verarbeitung mit foldl, map, filter, bind, LINQ
- Rechnen mit Mengen und Abbildungen im Ganzen
- (Funktional-Reaktive Progr. – OO: Beobachter)

# Aussagen

- statische Typisierung  $\Rightarrow$ 
  - 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
    - $\iff$  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/noschin1/cyp#readme>  
„... verifies proofs about Haskell-like programs“
- Lemma:  $\text{length } (xs ++ ys) =. \text{length } xs + \text{length } ys$   
Proof by induction on List xs  
Case []  
To show:  $\text{length } ([] ++ ys) =. \text{length } [] + \text{length } ys$   
Proof  $\text{length } ([] ++ ys)$   
    (by def ++)  $=. \text{length } ys$   
                   $\text{length } [] + \text{length } ys$   
    (by def length)  $=. 0 + \text{length } ys$   
    (by arith)  $=. \text{length } ys$   
QED  
Case x:xs  
To show:  $\text{length } ((x : xs) ++ ys) =. \text{length } (x : xs) + \text{length } ys$   
IH:  $\text{length } (xs ++ ys) =. \text{length } xs + \text{length } ys$

# 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`

# Funktionale Programmierung in der Industrie

- Workshops *Commercial users of functional programming*  
<http://cufp.org/2017/>, Adressen/Arbeitgeber der Redner der Konferenzen
- Paul Graham (2003) *Beating the Averages*  
<http://www.paulgraham.com/avg.html>  
(2003. LISP. Beachte: “programs that write programs, macros”  $\Rightarrow$  Fkt. höherer Ordnung)
- Joel Spolsky (2005) <http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html>  
Java is not, generally, a hard enough programming language that it can be used to discriminate between great programmers and mediocre programmers.

# 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/csharp-lang/blob/master/proposals/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, 2. Juli 2018)