

Deklarative (= fortgeschrittene) Programmierung

Vorlesung WS 09,10; SS 12–14, 16

Johannes Waldmann, HTWK Leipzig

13. Juli 2016

1 Einleitung

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

deklarativ: jedes (Teil-)Programm/Ausdruck hat einen *Wert*
(... und keine weitere (versteckte) *Wirkung*).

Werte können sein:

- “klassische” Daten (Zahlen, Listen, Bäume...)
- Funktionen (Sinus, ...)
- Aktionen (Datei schreiben, ...)

Softwaretechnische Vorteile

... der deklarativen Programmierung

- Beweisbarkeit: Rechnen mit Programmen wie in der Mathematik mit Termen
- Sicherheit: es gibt keine Nebenwirkungen und Wirkungen sieht man bereits am Typ
- Wiederverwendbarkeit: durch Entwurfsmuster (= Funktionen höherer Ordnung)
- Effizienz: durch Programmtransformationen im Compiler,
- Parallelisierbarkeit: durch Nebenwirkungsfreiheit

Beispiel Spezifikation/Test

```
import Test.SmallCheck

append :: forall t . [t] -> [t] -> [t]
append x y = case x of
  [] -> y
  h : t -> h : append t y

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

test1 = smallCheckI
  (associative (append :: [Int] -> [Int] -> [Int]))
```

Übung: Kommutativität (formulieren und testen)

Beispiel Verifikation

```
app :: forall t . [t] -> [t] -> [t]
app x y = case x of
  [] -> y
  h : t -> h : app t y
```

Beweise

$app\ x\ (app\ y\ z) == app\ (app\ x\ y)\ z$

Beweismethode: Induktion nach x .

- Induktionsanfang: $x == [] \dots$
- Induktionsschritt: $x == h : t \dots$

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 *Annotations*:

```
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;}
```

- $\sum_{x=0}^{2^6-1} f(x)$ `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)
- weitere Entwurfsmuster
- Code-Qualität, Code-Smells, Refactoring

Softwaretechnische Aspekte

- algebraische Datentypen, Pattern Matching, Termersetzungssysteme
Scale: case class, Java: Entwurfsmuster Kompositum, immutable objects, das Datenmodell von Git
- Funktionen (höherer Ordnung), Lambda-Kalkül, Rekursionsmuster
Lambda-Ausdrücke in C#, Entwurfsmuster Besucher
Codequalität, code smells, Refaktorisierung
- Typklassen zur Steuerung der Polymorphie
Interfaces in Java/C# , automatische Testfallgenerierung
- Bedarfsauswertung, unendl. Datenstrukturen
Iteratoren, Ströme, LINQ

Organisation der LV

- jede Woche eine Vorlesung, eine Übung
- Hausaufgaben (teilw. autotool)
`https://autotool.imn.htwk-leipzig.de/shib/cgi-bin/Super.cgi`
Identifizierung und Authentifizierung über Shibboleth-IDP des HTWK-Rechenzentrums, wie bei OPAL
- Prüfungszulassung: regelmäßiges (d.h. innerhalb der jeweiligen Deadline) und erfolgreiches (insgesamt $\geq 50\%$ der Pflichtaufgaben) Bearbeiten von Übungsaufgaben.
- Prüfung: Klausur (ohne Hilfsmittel)

Literatur

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

Übungen

- im Pool Z430, vgl. `http://www.imn.htwk-leipzig.de/~waldmann/etc/pool/`
- Beispiele f. deklarative Programmierung
 - funktional: Haskell mit ghci,
 - logisch: Prolog mit swipl,

- constraint: mit mathsat, z3
- Haskell-Entwicklungswerkzeuge
 - (eclipsefp, leksah, ..., <http://xkcd.org/378/>)
 - API-Suchmaschine <http://www.haskell.org/hoogle/>
- Commercial Uses of Functional Programming <http://www.syslog.cl.cam.ac.uk/2013/09/22/liveblogging-cufp-2013/>

2 Daten

Wiederholung: Terme

- (Prädikatenlogik) *Signatur* Σ ist Menge von Funktionssymbolen mit Stelligkeiten
ein Term t in Signatur Σ ist
 - Funktionssymbol $f \in \Sigma$ der Stelligkeit k mit Argumenten (t_1, \dots, t_k) , die selbst Terme sind.
- Term(Σ) = Menge der Terme über Signatur Σ
- (Graphentheorie) ein Term ist ein gerichteter, geordneter, markierter Baum
- (Datenstrukturen)
 - Funktionssymbol = Konstruktor, Term = Baum

Beispiele: Signatur, Terme

- Signatur: $\Sigma_1 = \{Z/0, S/1, f/2\}$
- Elemente von Term(Σ_1):
 $Z(), S(S(Z())), f(S(S(Z()))), Z()$
- Signatur: $\Sigma_2 = \{E/0, A/1, B/1\}$
- Elemente von Term(Σ_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 $t = f(t_1, \dots, t_k)$,
 dann $\text{Pos}(t) = \{\emptyset\} \cup \{[i-1] \uparrow p \mid 1 \leq i \leq k \wedge p \in \text{Pos}(t_i)\}$.

dabei bezeichnen:

- \emptyset die leere Folge,
- $[i]$ die Folge der Länge 1 mit Element i ,
- \uparrow 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: $S(f(S(S(Z())), Z()))[[0, 1] := S(Z)]x = \dots$
 Definition (durch Induktion über die Länge von p): \dots

Operationen mit Variablen in Termen

- $\text{Term}(\Sigma, V)$ = Menge der Terme über Signatur Σ mit Variablen aus V
 Beispiel: $\Sigma = \{Z/0, S/1, f/2\}, V = \{y\}, f(Z(), y) \in \text{Term}(\Sigma, V)$.
- Substitution σ : 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 Σ : Symbol $s \in \Sigma$ heißt

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

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

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

Übung: diese Eigenschaft formal spezifizieren

Beispiele: $R_1 = \{a(b(x)) \rightarrow b(a(x))\}$ 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. Terme (durch Konstruktoraufrufe)

Die *Größe* eines Terms t ist definiert durch

$$|f(t_1, \dots, t_k)| = 1 + \sum_{i=1}^k |t_i|.$$

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

Vervollständigen Sie die Definition der *Tiefe* von Termen:

$$\begin{aligned} \text{depth}(f()) &= 0 \\ k > 0 &\Rightarrow \text{depth}(f(t_1, \dots, t_k)) = \dots \end{aligned}$$

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

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

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

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

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

Abkürzung für Anwendung von 0-stelligen Symbolen: anstatt $Z()$ schreibe Z .

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

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

- für $\{A(B(x)) \rightarrow B(A(x))\}$
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.

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

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)

Bispiele (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 Constructoren sind Variablen \Rightarrow Case-Ausdruck ist disjunkt und vollständig.

Peano-Zahlen

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

Aufgaben:

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

Pattern Matching in versch. Sprachen

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

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

Übung Pattern Matching, Programme

- Für die Deklarationen

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

entscheide/bestimme für jeden der folgenden Ausdrücke:

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

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

- Operationen auf Wahrheitswerten:


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

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

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

Syntax: Funktionsname

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

Operator als Funktion: `(&&) False True`, Funktion als Operator: `True `f` False`.

- Listen von Wahrheitswerten:

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

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

entsprechend `or :: List -> Bool`

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

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

benutze `Prelude.+` (das ist Operator), `Prelude.min`, `Prelude.max`

- für Peano-Zahlen `data N = Z | S N`
 implementieren Sie *plus*, *mal*, *min*, *max*

4 Polymorphie

Definition, Motivation

- Beispiel: binäre Bäume mit Schlüssel vom Typ e

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

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

Beispiele f. Typkonstruktoren (I)

- Kreuzprodukt:

```
data Pair a b = Pair a b
```

- disjunkte Vereinigung:

```
data Either a b = Left a | Right b
```

- `data Maybe a = Nothing | Just a`

- Haskell-Notation für Produkte:

```
(1, True) :: (Int, Bool)
```

für 0, 2, 3, ... 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: Datenkonstruktoren polymorpher Typen.

Bezeichnungen f. Polymorphie

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

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

Operationen auf Listen (I)

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

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

Operationen auf Listen (II)

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

Besser ist:

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

mit Spezifikation

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

Übung: daraus die Implementierung von `rev_app` ableiten

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

Operationen auf Bäumen

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

Knotenreihenfolgen

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

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

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

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

Übung Polymorphie

Geben Sie alle Elemente dieser Datentypen an:

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

Operationen auf Listen:

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

Operationen auf Bäumen:

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

Kochrezept: Objektkonstruktion

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

Lösung (Bsp):

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

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

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

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

Kochrezept: Typ-Bestimmung

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

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

Lösung:

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

Vorgehen zur Typbestimmung eines Namens:

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

Statische Typisierung und Polymorphie

Def: dynamische Typisierung:

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

Def: statische Typisierung:

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

Bsp. für Programm ohne statischen Typ (Javascript)

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

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

entsprechendes Haskell-Programm ist statisch fehlerhaft

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


Nutzen der statischen Typisierung:

- beim Programmieren: Entwurfsfehler werden zu Typfehlern, diese werden zur Entwurfszeit automatisch erkannt \Rightarrow früher erkannte Fehler lassen sich leichter beheben
- beim Ausführen: es gibt 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

Von der Spezifikation zur Implementierung (I)

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

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

aus der Typdeklaration wird abgeleitet:

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

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

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

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

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

Von der Spezifikation zur Implementierung (II)

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

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

Spezifikation:

```
maximum (Cons x1 Nil) = x1
```

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

- 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.

5 Unveränderliche Objekte

Überblick

- alle Attribute aller Objekte sind unveränderlich (`final`)
- anstatt Objekt zu ändern, konstruiert man ein neues

Eigenschaften des Programmierstils:

- vereinfacht Formulierung und Beweis von Objekteigenschaften
- parallelisierbar (keine updates, keine *data races*)
<http://fpcomplete.com/the-downfall-of-imperative-programming/>
- Persistenz (Verfügbarkeit früherer Versionen)
- Belastung des Garbage Collectors (... dafür ist er da)

Beispiel: Einfügen in Baum

- destruktiv:

```
interface Tree<K> { void insert (K key); }
Tree<String> t = ... ;
t.insert ("foo");
```

- persistent (Java):

```
interface Tree<K> { Tree<K> insert (K key); }
Tree<String> t = ... ;
Tree<String> u = t.insert ("foo");
```

- persistent (Haskell):

```
insert :: Tree k -> k -> Tree k
```

Beispiel: (unbalancierter) Suchbaum

```
data Tree k = Leaf
            | Branch (Tree k) k (Tree k)
insert :: Ord k => k -> Tree k -> Tree k
insert k t = case t of ...
```

Diskussion:

- Ord k entspricht K implements Comparable<K>, genaueres später (Haskell-Typklassen)
- wie teuer ist die Persistenz? (wieviel Müll entsteht bei einem insert?)

Beispiel: Sortieren mit Suchbäumen

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

insert :: Ord k => k -> Tree k -> Tree k

build :: Ord k => [k] -> Tree k
build = foldr ... ...

sort :: Ord k => [k] -> [k]
sort xs = ... ( ... xs )
```

Persistente Objekte in Git

<http://git-scm.com/>

- *Distributed* development.
- Strong support for *non-linear* development.
(Branching and merging are fast and easy.)
- Efficient handling of *large* projects.
(z. B. Linux-Kernel, <http://kernel.org/>)
- Toolkit design.
- Cryptographic authentication of history.

Objekt-Versionierung in Git

- Objekt-Typen:
 - Datei (blob),
 - Verzeichnis (tree), mit Verweisen auf blobs und trees
 - Commit, mit Verweisen auf tree u. commits (Vorgänger)

```
git cat-file -p <hash>
```

- Objekte sind *unveränderlich* und durch SHA1-Hash (160 bit = 40 Hex-Zeichen) identifiziert

- statt Überschreiben: neue Objekte anlegen
- jeder Zustand ist durch Commit-Hash (weltweit) eindeutig beschrieben und kann wiederhergestellt werden

Quelltexte zur Vorlesung: <https://gitlab.imn.htwk-leipzig.de/waldmann/fop-ss16>

6 Funktionen

Funktionen als Daten

bisher:

```
f :: Int -> Int
f x = 2 * x + 5
```

äquivalent: Lambda-Ausdruck

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

Lambda-Kalkül: Alonzo Church 1936, Henk Barendregt 198*, ...

Funktionsanwendung:

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

ist nur erlaubt, falls keine in A freie Variable durch ein Lambda in B gebunden wird.

Der Lambda-Kalkül

... als weiteres Berechnungsmodell,

(vgl. Termersetzungssysteme, Turingmaschine, Random-Access-Maschine)

Syntax: die Menge der Lambda-Terme Λ ist

- jede Variable ist ein Term: $v \in V \Rightarrow v \in \Lambda$
- Funktionsanwendung (Applikation):
 $F \in \Lambda, A \in \Lambda \Rightarrow (FA) \in \Lambda$
- Funktionsdefinition (Abstraktion):
 $v \in V, B \in \Lambda \Rightarrow (\lambda v. B) \in \Lambda$

Semantik: eine Relation \rightarrow_β auf Λ

(vgl. \rightarrow_R für Termersetzungssystem R)

Freie und gebundene Variablen(vorkommen)

- Das Vorkommen von $v \in V$ an Position p in Term t heißt *frei*, wenn „darüber kein $\lambda v \dots$ 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_β

Relation \rightarrow_β auf Λ (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_α

- Relation \rightarrow_α auf Λ , beschreibt *gebundene Umbenennung* einer lokalen Variablen.
- Beispiel $\lambda x.fxz \rightarrow_\alpha \lambda y.fyz$.
(f und z sind frei, können nicht umbenannt werden)
- Definition $t \rightarrow_\alpha t'$:
 - $\exists p \in \text{Pos}(t)$, 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_β zu erfüllen.

Bsp: betrachte den unterstrichenen Redex in

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

Umbenennung von lokalen Variablen

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

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

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

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

Lösung: lokal umbenennen

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

dann ist Ersetzung erlaubt

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

Lambda-Terme: verkürzte Notation

- Applikation ist links-assoziativ, Klammern weglassen:

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

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:

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

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 $\lambda fx.f(fx)$

- Ansatz mit Typvariablen $f :: t_1, x :: t_2$
- betrachte (fx) : der Typ von f muß ein Funktionstyp sein, also $t_1 = (t_{11} \rightarrow t_{12})$ 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 $(T_1 \rightarrow (T_2 \rightarrow B))$,
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
```

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

```
> f(4)
7
```

Beispiele Fkt. höherer Ord.

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

Übung Lambda-Kalkül

- Wiederholung: konkrete Syntax, abstrakte Syntax, Semantik
- $S = \lambda xyz.xz(yz)$, $K = \lambda ab.a$, Normalform von $SKKc$
- (mit `data N=Z | S N`) bestimme Normalform von $ttSZ$ für $t = \lambda fx.f(fx)$,
- definiere Λ als algebraischen Datentyp `data L = ...` (3 Konstruktoren)
implementiere `size :: L -> Int`, `depth :: L -> Int`.
implementiere `bvar :: L -> S.Set String`, `fvar :: L -> S.Set String`,
siehe Folie mit Definitionen und dort angegebene Testfälle
benutze `import qualified Data.Set as S`, API-Dokumentation: <https://hackage.haskell.org/package/containers/docs/Data-Set.html>
- autotool-Aufgaben Lambda-Kalkül

Übung Fkt. höherer Ordnung

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

```
compose ::  
compose = \ f g -> \ x -> f (g x)
```
- Implementierung von `takeWhile`, `dropWhile`

7 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)

ein Rekursionsmuster anwenden = jeden Konstruktor durch eine passende Funktion ersetzen.

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

Rekursionsmuster instantiiieren = (Konstruktor-)Symbole interpretieren (durch Funktionen) = eine Algebra angeben.

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

Rekursionsmuster (Merksätze)

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

- Anzahl der Muster-Argumente = Anzahl der Konstruktoren (plus eins für das Datenargument)
- Stelligkeit eines Muster-Argumentes = Stelligkeit des entsprechenden Konstruktors
- Rekursion im Typ \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 ...
```

Übung Rekursionsmuster

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

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

Jeweils:

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

8 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
```

(Ü: implementiere durch unbalancierten Suchbaum)

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

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

hier ist `c` ein *Strategie-Objekt*

9 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 `O`) gibt es häufig auch „Verweis auf kein Objekt“— auch vom Typ `O`. Deswegen *null pointer exceptions*.

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

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

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

Alg. DT und Pattern Matching in Scala

<http://scala-lang.org>

algebraische Datentypen:

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

pattern matching:

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

beachte: Typparameter in eckigen Klammern

10 Objektorientierte Rekursionsschemata

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.*

Wiederholung Rekursionsschema

fold anwenden: jeden Konstruktor d. Funktion ersetzen

- Konstruktor \Rightarrow Schema-Argument
- ... mit gleicher Stelligkeit
- Rekursion im Typ \Rightarrow Anwendung auf Schema-Resultat

```

data Tree a = Leaf a | Branch (Tree a) (Tree a)
  Leaf    :: a -> Tree a
  Branch  :: Tree a -> Tree a -> Tree a
fold :: (a -> b) -> (b -> b -> b) -> Tree a -> b
fold leaf branch t = case t of
  Leaf k -> leaf k
  Branch l r -> branch (fold leaf branch l)
                    (fold leaf branch r)

depth :: Tree a -> Int
depth = fold ( \ k -> 0 ) ( \ x y -> 1 + max x y )

```

Wiederholung: Kompositum

Haskell: algebraischer Datentyp

```

data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
  Leaf    :: a -> Tree a
  Branch  :: Tree a -> Tree a -> Tree a

```

Java: Kompositum

```

interface Tree<A> { }
class Leaf<A> implements Tree<A> { A key; }
class Branch<A> implements Tree<A> {
  Tree<A> left; Tree<A> right;
}

```

(Scala: case class)

Übung Kompositum

```

public class Main {
  // vollst. Binärbaum der Tiefe d
  // mit Schlüssel 2^d * (c - 1) .. 2^d * c - 1
  static Tree<Integer> build (int d, int c);

  class Pair<A,B> { A first; B second; }
  // (Schlüssel links außen, Schl. rechts außen)
  static <A> Pair<A,A> bounds (Tree<A> t);

  public static void main(String[] args) {

```

```

    Tree<Integer> t = Main.build(4,1);
    System.out.println (Main.bounds(t));
} }

```

Kompositum und Visitor

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

```

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

```

Empfangen eines Besuchers: durch jeden Teilnehmer des Kompositums

```

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

```

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

Aufgabe: Besucher für Listen

Schreibe das Kompositum für

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

und den passenden Besucher. Benutze für

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

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

Quelltexte aus Vorlesung:

<https://gitlab.imn.htwk-leipzig.de/waldmann/fop-ss16>

Eine Funktion, die kein Fold ist

Das geht: `f xs = die Länge von xs ist gerade`

```
f = fold True ( \ x y -> not y )
```

Das geht nicht: `g xs = die Länge von xs ist >= 2` **Beweis:** falls doch `g = fold nil cons,`

dann betrachte

```
l0 = Nil ; g l0 = False -- nach Spez.
l1 = Cons 4 Nil ; g l1 = False -- nach Spez.
g (Cons 2 l0) = False -- nach Spezifikation
g (Cons 2 l0) = cons 2 (g l0) = cons 2 False
g (Cons 2 l1) = True -- nach Spezifikation
g (Cons 2 l1) = cons 2 (g l1) = cons 2 False
```

es folgt Widerspruch `False = cons 2 False = True`

d.h. die Annahme (`g = fold nil cons`) ist falsch.

11 Polymorphie

Arten der Polymorphie

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

moderne OO-Sprachen (u.a. Java, C#) bieten *beide* Formen der Polymorphie
mit statischer Sicherheit (d.h. statische Garantie, daß zur Laufzeit keine Methoden
fehlen)

Java-Notation f. generische Polymorphie

generischer *Typ* (Typkonstruktor):

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

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

statische generische *Methode*:

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

Typargumente können auch inferiert werden.

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

Beispiel f. dynamische Polymorphie

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

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

Klassen, Schnittstellen und Entwurfsmuster

- FP-Sichtweise: Entwurfsmuster = Fkt. höherer Ordnung
- OO-Sichtweise: E.M. = nützliche Beziehung zw. Klassen
... die durch Schnittstellen ausgedrückt wird.
⇒ Verwendung von konkreten Typen (Klassen) *ist ein Code Smell*, es sollen soweit möglich abstrakte Typen (Schnittstellen) sein. (Ü: diskutiere `IEnumerable`)
- insbesondere: in Java (ab 8):
funktionales Interface = hat genau eine Methode
eine implementierende anonyme Klasse kann als Lambda-Ausdruck geschrieben werden

Erzwingen von Abstraktionen

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

Wie kann `C x = new C()` verhindert werden,  
und `I x = new C()` erzwungen?
- Ansatz: 

```
class C { private C() { } }
```

aber dann ist auch `I x = new C()` verboten.
- Lösung: Fabrik-Methode

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

## Das Fabrik-Muster

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

die Gemeinsamkeit der Konstruktoren kann nicht in I ausgedrückt werden.

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

## Typklassen in Haskell: Überblick

- in einfachen Anwendungsfällen:  
Typklasse in Haskell ~ Schnittstelle in OO:  
beschreibt Gemeinsamkeit von konkreten Typen



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

## Beispiel

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

Kann mit Typklassen so formuliert werden:

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

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

## Unterschiede Typklasse/Interface (Bsp)

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

## Typklassen können mehr als Interfaces

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

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

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

## Grundwissen Typklassen

- Typklasse schränkt statische Polymorphie ein  
(Typvariable darf nicht beliebig 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`

## Übung Polymorphie

- Besucher für Listen-Kompositum hinzufügen (Quelltext aus VL), damit implementieren:

```
class Main {
 static Boolean and (List<Boolean> xs) {
 return xs.receive(new ...);
 }
}
```

```

public static void main (String [] argv) {
 List<Boolean> xs = new Cons<Boolean>(...);
 syso (Main.and(xs));
}
}

```

- nach Java übersetzen und implementieren

```

data Pair a b = Pair { first :: a, second :: b }
swap :: Pair a b -> Pair b a
p :: Pair Int Bool ; p = Pair 3 False ; q = swap p

```

- autotool: Aufgabe zu Typklassen (unbalancierte Suchbäume)

## 12 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
```

## Aufgaben zu Striktheit

Beispiel 1: untersuche Striktheit der Funktion

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

Antwort:

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

Beispiel 2: untersuche Striktheit der Funktion

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

Antwort (teilweise)

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

Aufgabe:

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

## Implementierung der verzögerten Auswertung

Begriffe:

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

bei jedem Konstruktor- und Funktionsaufruf:

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

## Bedarfsauswertung in Scala

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

<http://www.scala-lang.org/>

## Diskussion

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

## Anwendungen der verzögerten Auswertg. (I)

Abstraktionen über den Programm-Ablauf

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

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

- in Haskell geht das (direkt in ghci)

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

## Anwendungen der verzögerten Auswertg. (II)

unendliche Datenstrukturen

- Modell:

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

- man benutzt meist den eingebauten Typ `data [a] = [] | a : [a]`
- alle anderen Anwendungen von `[a]` sind *falsch*  
z.B. als Arrays, Strings, endliche Mengen  
dafür gibt es `Data.Seq`, `Data.Text`, `Data.Set` <http://hackage.haskell.org/package/containers> <http://hackage.haskell.org/package/text>

## 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  $xs$

## 13 OO-Simulation v. Bedarfsauswertung

### Motivation (Wdhlg.)

Unix:

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

Haskell:

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

C#:

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

- logische Trennung: Produzent  $\rightarrow$  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) { ... }`

### Beispiele Iterator

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

### Beispiel Iterator Java

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

Aufgabe: implementiere eine Methode

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

soll count Zahlen ab start liefern.

Testfälle dafür:

- @Test
 

```
public void t1() {
 assertTrue (3 == Main.range(3, 5).iterator().next());
}
@Test
public void t2() {
 assertTrue (5 == StreamSupport.stream(Main.range(3, 5).spliterator
}
```

## Enumerator (C#)

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

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

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

## Zusammenfassung Iterator

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

## Iteratoren mit yield

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

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

```

## Aufgaben Iterator C#

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

```

Implementiere „das merge aus mergesort“ (Spezifikation?)

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

```

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

```

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

```

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

## Streams in C#: funktional, Linq

### Funktional

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

Typ von Select? Implementierung?

Linq-Schreibweise:

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

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

# 14 Fkt. höherer Ord. für Streams

## Motivation

- Verarbeitung von Datenströmen,
- durch modulare Programme,  
zusammengesetzt aus elementaren Strom-Operationen
- angenehme Nebenwirkung (1):  
(einige) elementare Operationen sind parallelisierbar
- angenehme Nebenwirkung (2):  
externe Datenbank als Datenquelle, Verarbeitung mit Syntax und Semantik (Typsystem) der Gastsprache

## Strom-Operationen

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

## Strom-Transformationen (1)

elementweise (unter Beibehaltung der Struktur)

Vorbild:

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

Realisierung in C#:

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

Rechenregeln für map:

```
map f [] = ...
```

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

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

## Strom-Transformationen (2)

Änderung der Struktur, Beibehaltung der Elemente

Vorbild:

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

Realisierung: Take, Drop, Where

Übung: takeWhile, dropWhile, ...

- ausprobieren (Haskell, C#)
- implementieren
  - Haskell: 1. mit expliziter Rekursion, 2. mit fold
  - C# (Enumerator): 1. mit Current, MoveNext, 2. yield

## Strom-Transformationen (3)

neue Struktur, neue Elemente

Vorbild:

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

Realisierung:

SelectMany

Rechenregel (Beispiel):

```
map f xs = xs >>= ...
```

Übung:

Definition des Operators >=> durch

```
(s >=> t) = \ x -> (s x >>= t)
```

Typ von >=>? Assoziativität? neutrale Elemente?

## Strom-Verbraucher

„Vernichtung“ der Struktur

(d. h. kann danach zur Garbage Collection, wenn keine weiteren Verweise existieren)

Vorbild:

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

in der Version „von links“

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

Realisierung (Ü: ergänze die Typen)

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

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

## Zusammenfassung: Ströme

... und ihre Verarbeitung

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

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

## Bsp: die Spezifikation von TakeWhile

[https://msdn.microsoft.com/en-us/library/bb534804\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/bb534804(v=vs.110).aspx)

zeigt Nachteile natürlichsprachlicher Spezifikationen:

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

aber in welcher Reihenfolge? Da steht nur “contains”. Also ist als Wert von `(new int [] {1, 2, 3, 4}).TakeWhile(x => x < 8)` auch `{2, 1}` möglich. Oder `{1, 2, 1}`? Oder `{1, 5, 2, 7}`? Alle enthalten 1 und 2.

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

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

Es wäre schon gegangen, man hätte nur wollen müssen:

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

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

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

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

## Arbeiten mit Collections in Haskell

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

Beispiel-Funktionen mit typischen Eigenschaften:

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

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

Anwendungen:

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



## Linq-Syntax (type-safe SQL)

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

wird vom Compiler übersetzt in

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

Beachte:

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

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

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

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

wird vom Compiler übersetzt in

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

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

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

[https://wiki.haskell.org/All\\_About\\_Monads](https://wiki.haskell.org/All_About_Monads)

## Linq und Parallelität

... das ist ganz einfach: anstatt

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

schreibe

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

Dadurch werden

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

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

## Übung Stream-Operationen

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

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

das `map` verschwindet dabei  $\Rightarrow$  *stream fusion* (Coutts, Leshchinsky, Stewart, 2007)

<http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.7401>

- die Regel ergänzen (autotool)

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

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

### **Bsp: nicht durch fold darstellbare Funktion**

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

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

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

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

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

## **15 Serialisierung, Persistenz**

### **Motivation**

Die meisten Daten leben länger als ein Programmlauf, vgl.

- Akten (Papier), Archiv, ...
- Bearbeitung/Ergänzung einer Akte

Akten (Daten) in maschinenlesbarer Form:

- Lochkarten (US-Volkszählung 1890)
- Magnetbänder, Festplatten

Programmtexte sprechen nur über Daten während des Programmlaufes.

## Ansätze

- Programm bestimmt Form der Daten  
externe Repräsentation (DB-Schema) wird aus interner Repräsentation (Typ, Klassen) abgeleitet (automatisch, unsichtbar)
- Programm verarbeitet vorhandene Daten  
interne Repräsentation (Typen) wird aus externer Repr. (DB-Schema) abgeleitet
- Programm läuft (scheinbar) immer  
Application Server verwaltet Softwarekomponenten und Datenkomponenten

## Enterprise Java Beans

Klasse als Entity Bean (vereinfacht):

```
import javax.persistence.*;
@Entity public class C {
 @Id int id;
 String foo;
 double bar;
}
```

Application Server (z. B. JBoss) verwaltet diese Beans, Datenbankschema kann autom. generiert werden.

JSR 220: Enterprise JavaBeans™ 3.0 <http://www.jcp.org/en/jsr/detail?id=220>

## DB-Anfragen in Java EE

```
public List findWithName(String name) {
 return em.createQuery(
 "SELECT c FROM Customer c WHERE c.name LIKE :custName")
 .setParameter("custName", name)
 .setMaxResults(10).getResultList(); } }
```

<http://docs.oracle.com/javaee/5/tutorial/doc/bnbqw.html#bnbrg>  
beachte: Query ist hier String, aber gemeint ist: Funktion ( $\lambda$  custName  $\rightarrow$  ...)  
Nachteile (vgl. auch <http://xkcd.com/327/>)

- drei Namensbereiche
- keine statische Typisierung
- keine Syntaxprüfung

## LINQ und SQLmetal (1)

<http://msdn.microsoft.com/en-us/library/bb386987.aspx>  
generiert C#-Typdeklaration aus DB-Schema

```
sqlmetal /namespace:nwind /provider:Sqlite
'/conn:Data Source=Northwind.db3' /code:nwind.cs
```

Objekte können dann statisch typisiert verarbeitet werden.

## LINQ und SQLmetal (2)

Datenbankverbindung herstellen:

```
using System; using System.Data.Linq;
using System.Linq; using Mono.Data.Sqlite;
var conn = new SqliteConnection
("DbLinqProvider=Sqlite; Data Source=Northwind.db3");
var db = new nwind.Main (conn);
```

Datenquelle benutzen:

```
var rs = from c in db.Customers
 select new { c.City, c.Address} ;
foreach (var r in rs) { Console.WriteLine (r); }
```

beachte LINQ-Notation (from, select)  
und Verwendung von anonymen Typen (new) (für Tupel)

## Übung LINQ und SQLmetal

Quellen: <https://github.com/DbLinq/dblinq2007>

- Beispiel-Datenbank ansehen: (sqlite3 Northwind.db3)
- Schnittstellenbeschreibung herstellen und ansehen

```
sqlmetal /namespace:nwind /provider:Sqlite "/conn:Data Source=Northwind.db3
```

- Hauptprogramm kompilieren und ausführen

```
mcs db.cs nwind.cs -r:Mono.Data.Sqlite, System.Data, System.Data.Linq
mono db.exe
```

## 16 Modellierung von Zustandsänderungen

### (Un)veränderliche Objekte

- Zustand eines Objektes = Belegung seiner Attribute
- veränderlicher Zustand erschwert Programm-Benutzung und -Verifikation  
(denn er muß bei jedem Methodenaufruf berücksichtigt werden, ist aber *implizit*)
- Abhilfe: Zustand wird *unveränderlich (immutable)*  
(d.h. beim Konstruktor-Aufruf festgelegt)
- Methode, die Zustand *ändert*, erzeugt stattdessen neues Objekt

### (Un)veränderliche Objekte, Bsp. 1

- Objekt mit veränderlichem Zustand

```
class C0 {
 private int z = 0;
 public void step () { this.z++; }
}
```

- Objekt mit unveränderlichem Zustand

```
class C1 {
 private final int z;
 public C1 step () {
 return new C1 (this.z + 1); }
}
```

### (Un)veränderliche Objekte, Bsp. 2

veränderlich:

```
class Stack<E> {
 void push (E item);
 E pop ();
 private List<E> contents;
}
```

unveränderlich:

```
class Stack<E> {
 List<E> push (List<E> contents, E item);
 Pair<List<E>,E> pop (List<E> contents);
}
```

Testfrage: wie heißt die Funktion `push` sonst? (bei vertauschten Argumenten)

### Zustand, Spezifikation, Tests

- Für Programm-Spezifikation (und -Verifikation) muß der Zustand sowieso benannt werden,
- und verschiedene Zustände brauchen verschiedene Namen (wenigstens: vorher/nachher)
- also kann man sie gleich durch verschiedene Objekte repräsentieren.

explizite Zustandsobjekte sind auch beim Testen nützlich:

- Test soll in bestimmtem Zustand stattfinden,
- bestimmten Zustand erzeugen.

### Zustand in Services

- wiederverwendbare Komponenten („Software als Service“) dürfen *keinen* Zustand enthalten.  
das garantiert Thread-Sicherheit, gestatte einfaches Load-Balancing
- vgl.: Unterprogramme dürfen keine globalen Variablen benutzen,  
dann sind Aufrufe (auch nebenläufige) unabhängig voneinander
- in der (reinen) funktionalen Programmierung passiert das von selbst: dort *gibt es keine Zuweisungen* (nur const-Deklarationen mit einmaliger Initialisierung).  
⇒ Thread-Sicherheit ohne Zusatzaufwand

## Verhaltensmuster: Beobachter

zur Programmierung von Reaktionen auf Zustandsänderung von Objekten

- 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 (...)

Objektbeziehungen sind damit konfigurierbar.

### Beobachter: Beispiel (I)

```
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 (); }
```

### Beobachter: Beispiel Sudoku, Semantik

- Spielfeld ist Abbildung von Position nach Zelle,
- Menge der Positionen ist  $\{0, 1, 2\}^4$
- Zelle ist leer (Empty) oder besetzt (Full)
- leerer Zustand enthält Menge der noch möglichen Zahlen



- Invariante?
- Zelle  $C_1$  beobachtet Zelle  $C_2$ , wenn  $C_1$  und  $C_2$  in gemeinsamer Zeile, Spalte, Block

Test: eine Sudoku-Aufgabe laden und danach Belegung der Zellen auf Konsole ausgeben.

### **Beobachter: Beispiel Sudoku, GUI**

Plan:

- Spielfeld als JPanel (mit GridLayout) von Zellen
- Zelle ist JPanel, Inhalt:
  - leer: JButton für jede mögliche Eingabe
  - voll: JLabel mit gewählter Zahl

Hinweise:

- JPanel löschen: `removeAll()`, neue Komponenten einfügen: `add()`, danach Layout neu berechnen: `validate()`
- JPanel für die Zelle einrahmen: `setBorder()`

### **Model/View/Controller**

(Modell/Anzeige/Steuerung)

(engl. *to control* = steuern, *nicht*: kontrollieren)

Bestandteile (Beispiel):

- Model: Counter (`getCount`, `step`)
- View: JLabel ( $\leftarrow$  `getCount`)
- Controller: JButton ( $\rightarrow$  `step`)

Zusammenhänge:

- Controller steuert Model
- View beobachtet Model

## 17 Code Smells und Refactoring

### Für wen schreibt man Code?

Donald Knuth 1993, vgl. <http://tex.loria.fr/historique/interviews/knuth-clb1993.html>:

- Programming is: telling a *human* what a computer should do.

Donald Knuth 1974,

- Premature optimization is the root of all evil.

<http://www-cs-faculty.stanford.edu/~uno/>

### Wie soll guter Quelltext aussehen?

- clarity and simplicity are of paramount importance
- the user of a module should never be surprised by its behaviour
- modules should be as small as possible but not smaller
- code should be reused rather than copied
- dependencies between modules should be minimal
- errors should be detected as soon as possible, ideally at compile time

(Joshua Bloch: *Effective Java*, Addison-Wesley 2008

<https://www.pearsonhighered.com/program/Bloch-Effective-Java-2nd-Edition/PGM310651.html>

### Definition Refactoring

Martin Fowler: *Refactoring: Improving the Design of Existing Code*, A.-W. 1999,

<http://www.refactoring.com/>

Def: Software so ändern, daß sich

- externes Verhalten nicht ändert,
- interne Struktur verbessert.

siehe auch William C. Wake: *Refactoring Workbook*, A.-W. 2004 <http://www.xp123.com/rwb/>

## Refactoring anwenden

- mancher Code „riecht“ (schlecht)  
(Liste von *smells*)
- er (oder anderer) muß geändert werden  
(Liste von *refactorings*, Werkzeugunterstützung)
- Änderungen (vorher!) durch Tests absichern  
(JUnit)

## Refaktorisierungen

- Abstraktionen einführen:  
neue Schnittstelle, Klasse (Entwurfsmuster!)  
Methode, (temp.) Variable
- Abstraktionen ändern:  
Attribut/Methode bewegen (in andere Klasse)
- (unnötige) Abstraktionen entfernen

## Code Smell # 1: Duplicated Code

jede Idee sollte an *genau einer* Stelle im Code formuliert werden:  
Code wird dadurch

- leichter verständlich
- leichter änderbar

Verdoppelter Quelltext (copy–paste) führt immer zu Wartungsproblemen.

## Duplicated Code → Schablonen

duplizierter Code wird verhindert/entfernt durch

- *Schablonen* (beschreiben das Gemeinsame)
- mit *Parametern* (beschreiben die Unterschiede).

Beispiel dafür:

- Unterprogramm (Parameter: Daten, Resultat: Programm)
- polymorphe Klasse (Parameter: Typen, Resultat: Typ)
- Unterprogramm höherer Ordnung (Parameter: Programm, Resultat: Programm)

## Code Smell: Kommentar (Deodorant)

- Kommentar vorhanden ⇒ Code ist ohne Kommentar nicht verständlich
- “don’t comment bad code — rewrite it”

```
String f = "foo.bar"; // Ausgabedateiname
FilePath outfile = "foo.bar";
```

```
int state = 0; // initialize component
Component c ; c.initialize();
```

- drücke Inhalt des Kommentars durch Mittel der Programmiersprache aus!  
dann kann er *statisch* überprüft werden.

## Klassen-Entwurf

- benutze Klassen! (sonst: primitive obsession)
- ordne Attribute und Methoden richtig zu  
(Refactoring: move method, usw.)
- dokumentiere Invarianten für Objekte, Kontrakte für Methoden
- stelle Beziehungen zwischen Klassen durch Interfaces dar  
(... Entwurfsmuster)

## Primitive Daten (*primitive obsession*)

Symptome: Benutzung von `int`, `float`, `String`...

Ursachen:

- fehlende Klasse:  
z. B. `String` → `FilePath`, `Email`, `URI` ...
- schlecht implementiertes Fliegengewicht  
z. B. `int i` bedeutet `x[i]`
- simulierter Attributname:  
z. B. `Map<String, String> m; m.get("foo");`

Behebung: Klassen benutzen, Array durch Objekt ersetzen  
(z. B. `class M { String foo; ...}`)

## Verwendung von Daten: Datenklumpen

Fehler: Klumpen von Daten wird immer gemeinsam benutzt

```
String infile_base; String infile_ext;
String outfile_base; String outfile_ext;
```

```
static boolean is_writable
 (String base, String ext);
```

Indikator: ähnliche, schematische Attributnamen

Lösung: Klasse definieren

```
class File
 { String base; String extension; }
static boolean is_writable (File f);
```

Ü: vgl. mit `java.nio.file.Path`

## Datenklumpen—Beispiel

Beispiel für Datenklumpen und -Vermeidung:

```
java.awt
```

```
Rectangle(int x, int y, int width, int height)
Rectangle(Point p, Dimension d)
```

Vergleichen Sie die Lesbarkeit/Sicherheit von:

```
new Rectangle (20, 40, 50, 10);
new Rectangle (new Point (20, 40)
 , new Dimension (50, 10));
```

Vergleichen Sie:

```
java.awt.Graphics: drawRectangle(int,int,int,int)
java.awt.Graphics2D: draw (Shape);
class Rectangle implements Shape;
```

### Verwendung von Daten: Data Class

Fehler:

Klasse mit Attributen, aber ohne Methoden.

```
class File { String base; String ext; }
```

Lösung:

finde typische Verwendung der Attribute in Client-Klassen, (Bsp: `f.base + "." + f.ext`)  
schreibe entsprechende Methode, versteckte Attribute (und deren Setter/Getter)

```
class File { ...
 String toString () { ... }
}
```

### Mehrfachverzweigungen

Symptom: switch wird verwendet

```
class C {
 int tag; int FOO = 0;
 void foo () {
 switch (this.tag) {
 case FOO: { .. }
 case 3: { .. }
 }
 }
}
```

Ursache: Objekte der Klasse sind nicht ähnlich genug

Abhilfe: Kompositum-Muster

```
interface C { void foo (); }
class Foo implements C { void foo () { .. } }
class Bar implements C { void foo () { .. } }
```

## **null-Objekte**

Symptom: `null` (in Java) bzw. `0` (in C++) bezeichnet ein besonderes Objekt einer Klasse, z. B. den leeren Baum oder die leere Zeichenkette

Ursache: man wollte Platz sparen oder „Kompositum“ vermeiden.

Nachteil: `null` bzw. `*0` haben keine Methoden.

Abhilfe: ein extra Null-Objekt deklarieren, das wirklich zu der Klasse gehört.

## **Code-Größe und Komplexität**

Motto: was der Mensch nicht *auf einmal* überblicken/verstehen kann, versteht er *gar nicht*.

Folgerung: jede Sinn-Einheit (z. B. Implementierung einer Methode, Schnittstelle einer Klasse) muß auf eine Bildschirmseite passen

Code smells:

- Methode hat zu lange Argumentliste
- Klasse enthält zuviele Attribute
- Klasse enthält zuviele Methoden
- Methode enthält zuviele Anweisungen (Zeilen)
- Anweisung ist zu lang (enthält zu große Ausdrücke)

## **Benannte Abstraktionen**

überlangen Code in überschaubare Bestandteile zerlegen:

- Abstraktionen (Konstante, Methode, Klasse, Schnittstelle) einführen ... und dafür *passende Namen* vergeben.

Code smell: Name drückt Absicht nicht aus.

Symptome:

- besteht aus nur 1 ... 2 Zeichen, enthält keine Vokale
- numerierte Namen (`panel1`, `panel2`, `\dots`)
- unübliche Abkürzungen, irreführende Namen

Behebung: umbenennen, so daß Absicht deutlicher wird. (Dazu muß diese dem Programmierer selbst klar sein!)

Werkzeugunterstützung!

## Name enthält Typ

Symptome:

- Methodenname enthält Typ des Arguments oder Resultats

```
class Library { addBook(Book b); }
```

- Attribut- oder Variablenname bezeichnet Typ (sog. Ungarische Notation) z. B. `char ** ppcFoo`  
<http://ootips.org/hungarian-notation.html>
- (grundsätzlich) Name bezeichnet Implementierung statt Bedeutung

Namenskonventionen: schlecht, statische Typprüfung: gut.

## Vererbung bricht Kapselung

(Implementierungs-Vererbung: schlecht, Schnittstellen-Vererbung: gut.)

Problem: `class C extends B` ⇒

*C* hängt ab von Implementations-Details von *B*.

⇒ wenn Implementierung von *B* unbekannt, dann korrekte Implementierung von *C* nicht möglich.

⇒ Wenn man Implementierung von *B* ändert, kann *C* kaputtgehen.

Beispiel: `class CHS<E> extends HashSet<E>`, Methoden `add` und `addAll`,  
nach: Bloch: Effective Java, Abschnitt 14 (Favor composition over inheritance)

## Vererbung bricht Kapselung

Bloch, Effective Java, Abschnitt 15:

- design and document for inheritance...

API-Beschreibung muß Teile der Implementierung dokumentieren (welche Methoden rufen sich gegenseitig auf), damit man diese sicher überschreiben kann.

- ... or else prohibit it.

– am einfachsten: `final class C { ... }`

– mglw.: `class C { private C () { ... } ... }`

statt Vererbung: benutze Komposition (Wrapper) und dann Delegation.

Übung: `Counting(Hash) Set<E>` mittels Wrapper



## Wann darf man *extends* benutzen?

Konzepte:

- konkreter Datentyp (Klasse)
- abstrakter Datentyp (Schnittstelle)

Merksätze:

- Gut: Beziehung zw. konkretem und abstraktem Typ (`class C implements interface I`)
- Schlecht: Implementierungsvererbung (`class E extends class C`)
- Gut: Beziehung zw. Schnittstellen (`interface I1 extends interface I2, class Eq a => Ord a where ...`)

## Immutability

(Joshua Bloch: Effective Java, Abschnitt 13: Favor Immutability) — immutable = unveränderlich

Beispiele: String, Integer, BigInteger

- keine Set-Methoden
- keine überschreibbaren Methoden
- alle Attribute final

leichter zu entwerfen, zu implementieren, zu benutzen.

## Immutability

- immutable Objekte können mehrfach benutzt werden (sharing).  
(statt Konstruktor: statische Fabrikmethode oder Fabrikobjekt. Suche Beispiele in Java-Bibliothek)
- auch die Attribute der immutable Objekte können nachgenutzt werden (keine Kopie nötig)  
(Beispiel: negate für BigInteger)
- immutable Objekte sind sehr gute Attribute anderer Objekte:  
weil sie sich nicht ändern, kann man die Invariante des Objektes leicht garantieren

## Zustandsänderungen

Programmzustand ist immer implizit (d. h. unsichtbar).

⇒ jede Zustandsänderung (eines Attributes eines Objektes, einer Variablen in einem Block) erschwert

- Spezifikation, Tests, Korrektheitsbeweis,
- Lesbarkeit, Nachnutzung.

Code smells:

- Variable wird deklariert, aber nicht initialisiert (Refactoring: Variable später deklarieren)
- Konstruktor, der Attribute nicht initialisiert (d. h., der die Klasseninvariante nicht garantiert)

## Code smell: Temporäre Attribute

Symptom: viele `if (null == foo)`

Ursache: Attribut hat nur während bestimmter Programmteile einen sinnvollen Wert

Abhilfe: das ist kein Attribut, sondern eine temporäre Variable.

## Refaktorisierung von Ausdrücken

- code smells: ein langer Ausdruck, mehrfach der gleiche Ausdruck (z. B. ein Zahl- oder String-Literal)

refactoring: Konstante einführen

- *One man's constant is another man's variable.*

(Alan Perlis, 1982, <http://www.cs.yale.edu/quotes.html>)

- code smell: mehrere ähnliche Ausdrücke

refactoring: Unterprogramm (Funktion) einführen

(Funktion = Unterprogramm, das einen Wert liefert)

## Refaktorisierung durch Funktionen

Gegeben: (Code smell: duplizierter/ähnlicher Code)

```
{ int a = ... ; int b = ... ;
 int x = a * 13 + b; int y = a * 15 + b; }
```

Mögliche Refaktorisierungen:

- lokale Funktion (C#) (mit einem Parameter)
- globale Funktion (Java) (mit einem Parameter)?  
(welches Problem entsteht?)
- globale Funktion (Java), die dieses Problem vermeidet

Beobachtung: in Sprachen ohne lokale Unterprogramme werden solche Abstraktionen zu schwerfällig.

vgl. <http://openjdk.java.net/projects/lambda/>

## Refaktorisierung durch Prozeduren

(Prozedur = Unterprogramm, das den Programmzustand ändert)

- gleiche Betrachtung (lokal, global, Hilfsvariablen) wie für Funktionen
- erschwert durch Nebenwirkungen auf lokale Variablen

Eclipse:

- Extract method (mit Bezug auf 1, 2 lokale Variablen)
- Change local variable to field

Übung: Zusammenhang zwischen Code Smell *Kommentar* und Unterprogrammen

## Richtig refaktorisieren

- immer erst die Spezifikation (die Tests) schreiben
- Code kritisch lesen (eigenen, fremden), eine Nase für Anrührigkeiten entwickeln (und für perfekten Code).
- jede Faktorisierung hat ein Inverses.  
(neue Methode deklarieren ↔ Methode inline expandieren)  
entscheiden, welche Richtung stimmt!
- Werkzeug-Unterstützung erlernen

## Aufgaben zu Refactoring (I)

- Code Smell Cheat Sheet (Joshua Kerievsky): <http://industriallogic.com/papers/smellstorefactorings.pdf>
- Smell-Beispiele <http://www.imn.htwk-leipzig.de/~waldmann/edu/ss05/case/rwb/> (aus Refactoring Workbook von William C. Wake <http://www.xp123.com/rwb/>)  
ch6-properties, ch6-template, ch14-ttt

## Aufgaben zu Refactoring (II)

Refactoring-Unterstützung in Eclipse:

```
package simple;

public class Cube {
 static void main (String [] argv) {
 System.out.println (3.0 + " " + 6 * 3.0 * 3.0);
 System.out.println (5.5 + " " + 6 * 5.5 * 5.5);
 }
}
```

extract local variable, extract method, add parameter, ...

## Aufgaben zu Refactoring (II)

- Eclipse → Refactor → Extract Interface
- “Create Factory”
- Finde Beispiel für “Use Supertype”

# 18 Verfrühte Optimierung ...

... ist die Quelle allen Übels

So ist es richtig:

1. passende Datenstrukturen und Algorithmen festlegen ...

2. ... und korrekt implementieren,
3. Ressourcenverbrauch messen,
4. *nur bei nachgewiesenem Bedarf* Implementierung ändern, um Ressourcenverbrauch zu verringern.

und jede andere Reihenfolge ist falsch, sinnlos oder riskant.

### **Sprüche zur Optimierung**

(so zitiert in J. Bloch: Effective Java)

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity. – W. A. Wulf

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. – Donald E. Knuth

We follow two rules in the matter of optimization:

- Rule 1. Don't do it.
- Rule 2 (for experts only). Don't do it yet – that is, not until you have a perfectly clear and unoptimized solution.

– M.A. Jackson

### **Rekursion ist teuer? Falsch!**

Welches Programm ist schneller?

```
int gcd (int x, int y) { // Rekursion:
 if (0==y) return x else return gcd(y, x%y);
}
```

```
int gcd (int x, int y) { // Schleife:
 while (0!=y) {int h = x%y ; x = y; y = h;}
 return x;
}
```

Antwort: keines, gcc erzeugt identischen Assemblercode.

Das funktioniert immer für *Endrekursion* (= die letzte Aktion eines Unterprogramms ist der rekursive Aufruf), diese kann durch Sprung ersetzt werden.

## Java ist langsam? Falsch!

```
static int gcd (int x, int y) {
 if (0==y) return x; else return gcd(y,x%y);
}
```

Testtreiber:  $10^8$  Aufrufe, Laufzeit:

- C/gcc: 6.6 s
- Java: 7.1 s
- C#/Mono: 7.9 s

## Array-Index-Prüfungen sind teuer? Falsch!

James Gosling:

One of the magics of modern compilers is that they're able to "theorem-prove away" potential all [array] subscript checks. . . .

You might do a little bit of checking on the outside of the loop, but inside the loop, it just screams.

[The VM] had a crew of really bright people working on it for a decade, a lot of PhD compiler jockeys.

Quelle: Biancuzzi und Warden: Masterminds of Programming, O'Reilly, 2009

Codebeispiel: <http://www.imn.htwk-leipzig.de/~waldmann/etc/safe-speed/>

## Lokale Variablen sind teuer? Falsch!

Welches Programm braucht weniger Zeit oder Platz?

1) Variable h ist „global“:

```
int s = 0; int h;
for (int i = 0; i<n; i++) {
 h = i*i; s += h;
}
```

2) Variable h ist lokal:

```
int s = 0;
for (int i = 0; i<n; i++) {
 int h = i*i; s += h;
}
```

Antwort: keines, javac erzeugt identischen Bytecode.

# 19 Zusammenfassung, Ausblick

## Themen

- Terme, algebraische Datentypen
- Muster, Regeln, Term-Ersetzung
- Polymorphie, Typvariablen
- Funktionen, Lambda-Kalkül
- Rekursionsmuster
- abstrakte Datentypen (Schnittstellen, Typklassen)
- Strategie, Kompositum, Besucher
- Streams (Bedarfsauswertung, Iterator)
- Stream-Verarbeitung mit foldl, map, filter, LINQ
- Zustand, (im)mutability
- Code smells und Refactoring

## 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
  - ⇔ 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/>
- verifizierter C-Compiler <http://compcert.inria.fr/>
- Lösung der Übungsaufgabe foldr/foldl
  - mit Isabelle <http://isabelle.in.tum.de/>
  - Lösungsplan: <https://mail.haskell.org/pipermail/haskell-cafe/2009-March/058004.html>, Durchführung:

```
lemma lr : "foldl f a xs
 = foldr (\<lambda> x y . f y x) a (reverse xs) "
proof (induction xs arbitrary : a)
 case Nil show ?case by simp
 case Cons ... qed
```

Vollständiger Quelltext: <https://gitlab.imn.htwk-leipzig.de/waldmann/fop-ss16/blob/master/kw27/isabelle/Fold.thy>

Isabelle-Installation im Pool: `/usr/local/waldmann/share/isabelle/latest/Isabelle`

### 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/all/tree/master/yesod>

### **Industrielle Anwendg. d funkt. Progr.**

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

- siehe Adressen/Arbeitgeber der Redner der Konferenz

Diskussion:

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

### **Anwendungen v. Konzepten der fktl. Prog.**

- <https://www.rust-lang.org/> Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety.
- <https://developer.apple.com/swift/>  
... Functional programming patterns, e.g., map and filter, ... designed for safety.
- <https://github.com/dotnet/roslyn/blob/features/patterns/docs/features/patterns.md> enable many of the benefits of algebraic data types and pattern matching from functional languages

## Übung, Diskussion

- Auswertung der Umfrage: <http://www.imn.htwk-leipzig.de/~waldmann/edu/ss16/fop/umf/>
- Platz für lokale Namen in einer Methode

```
class C {
 static void m () { for (...) { int h = .. } }
}
javac C.java ; javap -c -v C
```

```
static void m(); descriptor: ()V flags: ACC_STATIC
Code: stack=2, locals=2, args_size=0
```