

Deklarative
(= fortgeschrittene)
Programmierung
Vorlesung

WS 09, WS 10, SS 12, SS 13, SS
14

Johannes Waldmann, HTWK Leipzig

10. Juli 2014

Formen der deklarative 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])
```

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

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>

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

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

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.

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

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 Z423



```
export PATH=/usr/local/waldmann/bin:$PATH
```

- ▶ 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/
```

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

Algebraische Datentypen

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

Bezeichnungen (benannte Notation)

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

```
x :: Foo
```

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

Bezeichnungen (positionelle Notation)

```
data Foo = Foo Int String
```

```
v = Foo 3 "bar"
```

Datentyp mit mehreren Konstruktoren

Beispiel (selbst definiert)

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

Bespiele (in Prelude vordefiniert)

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

Rekursive Datentypen

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

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

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 $\text{Pos}(t) = \{\square\} \cup \{[i-1] \# p \mid 1 \leq i \leq k \wedge p \in \text{Pos}(t_i)\}$.

dabei bezeichnen:

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

- ▶ to_R beschreibt *einen* Schritt der Rechnung von R ,
- ▶ transitive 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

Konstruktor-Systeme

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

- ▶ *definiert*, wenn $\exists (l, r) \in R : l[] = s$
- ▶ sonst *Konstruktor*.

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

- ▶ die definierten Symbol kommen links *nur* in den Wurzeln vor (rechts egal)

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?

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 $|\sqrt{a \cdot a + b \cdot b}|$.

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 -Nachfolger)

data und case

typisches Vorgehen beim Programmieren einer Funktion

```
f :: T -> ...
```

- ▶ Für jeden Konstruktor des Datentyps

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

- ▶ schreibe einen Zweig in der Fallunterscheidung

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

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)

Übung Programme

- ▶ (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 -> Int`
`depth :: T -> Int`
- ▶ für Peano-Zahlen `data N = Z | S N`
implementieren Sie *plus*, *mal*, *min*, *max*

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-programmin>
- ▶ Persistenz (Verfügbarkeit früherer Versionen)
- ▶ Belastung des Garbage Collectors (... dafür ist

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 und commits (Vorgänger)

```
git cat-file [-t|-p] <hash>
```

```
git ls-tree [-t|-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

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 enthält all-quantifizierte Typvariablen

Datenkonstruktoren polymorpher Typen sind polymorph.

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`)

Polymorphie in OO-Sprachen

Definitionen:

- ▶ generische Polymorphie: zur *Übersetzungszeit* werden Werte von Typparametern festgelegt.
- ▶ dynamische Polymorphie: es wird die Methodenimplementierung des *Laufzeittyps* benutzt
(dieser kann vom statischen Typ abgeleitet sein)

Anwendungen der generischen Polymorphie:

- ▶ polymorphe Typen (Klassen, Interfaces)
- ▶ polymorphe Funktionen (Methoden)

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  
System.out.println (x.m());  
x = new B(); // dynamischer Typ ist jetzt  
System.out.println (x.m());
```

Kompositum, Wiederholung

Gestaltung von zusammengesetzten Layouts:

- ▶ `Container extends Component`
- ▶ `JButton extends Component`
- ▶ `Container { add (Component c); }`

Modellierung als algebraischer Datentyp ist:

```
data Component
  = JButton { ... }
  | Container [ Component ]
```

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/](http://www.waldmann.edu/ws06/informatik/manage/)

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.

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 }
```

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

Maybe = Nullable

Algebraischer Datentyp (Haskell):

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

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

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

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

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

<http://msdn.microsoft.com/en-us/>

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)
```

Funktionen als Daten

bisher:

$$f :: \text{Int} \rightarrow \text{Int}$$

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

äquivalent: Lambda-Ausdruck

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

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. $fvar(t)$ = 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. $bvar(t)$ = Menge der in t gebundenen Variablen

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

$$bvar(x(\lambda x.\lambda y.x)) = \{x, y\},$$

Semantik des Lambda-Kalküls

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.)

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;
```

Lambda-Terme: verkürzte Notation

- ▶ Applikation als links-assoziativ auffassen, Klammern weglassen:

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

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

- ▶ geschachtelte Abstraktionen unter ein Lambda schreiben:

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

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

Ein- und mehrstellige Funktionen

eine einstellige Funktion zweiter Ordnung:

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

Anwendung dieser Funktion:

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

Kurzschreibweisen (Klammern weglassen):

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

Übung:

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

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

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

Typen

für nicht polymorphe Typen: tatsächlicher Argumenttyp muß mit deklarierendem 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.

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

$\sigma := \text{mgu}(A, A')$ (allgemeinster Unifikator)
allgemeinster Typ von (fx) ist dann $B\sigma$.

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
```

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
t (f) (3);
```

- ▶ Anwendungen bei Streams (Iterable), später mehr

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

Lambda-Ausdrücke in Javascript

```
$ node
```

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

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

Übung Lambda-Kalkül

- ▶ Wiederholung: konkrete Syntax, abstrakte Syntax, Semantik
- ▶ $S = \lambda xyz.xz(yz)$, $K = \lambda ab.a$, Normalform von $SKKc$
- ▶ $t = \lambda fx.f(fx)$, Normalform von $ttS0$
- ▶ 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
```

Übung Fkt. höherer Ordnung

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

```
compose ::
```

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

- ▶ Implementierung von takeWhile, dropWhile

Rekursion über Bäume (Beispiele)

```

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

summe :: Tree Int -> Int
summe t = case t of
  Leaf {} -> 0 ; Branch {} ->
    summe (left t) + key t + summe (right t)

preorder :: Tree a -> [a]
preorder t = case t of
  Leaf {} -> [] ; Branch {} ->
    key t : preorder (left t) ++ preorder (right t)

```

Rekursion über Bäume (Schema)

```
f :: Tree a -> b
f t = case t of
  Leaf {} -> ...
  Branch {} ->
    ... (f (left t)) (key t) (f (right t))
```

dieses Schema *ist* eine Funktion höherer Ordnung:

```
fold :: ( ... ) -> ( ... ) -> ( Tree a -> ... )
fold leaf branch = \ t -> case t of
  Leaf {} -> leaf
  Branch {} ->
    branch (fold leaf branch (left t))
            (key t) (fold leaf branch (right t))
```

Haskell-Syntax für Komponenten-Namen

- ▶ bisher: positionelle Notation der Konstruktor-Argumente

```
data Tree a = Leaf | Branch (Tree a) a
t = Branch Leaf "bar" Leaf
case t of Branch l k r -> k
```

- ▶ **alternativ: Notation mit Komponentennamen:**

```
data Tree a = Leaf
  | Branch {left::Tree a, key::a, right::Tree a}
t = Branch {left=Leaf, key="bar", right=Leaf}
case t of Branch {} -> key t
```

- ▶ kann auch gemischt verwendet werden:

```
Branch {left=Leaf, key="bar"}
```

Objektinitialisierer in C#

```
class C {  
    public int foo; public string bar;  
}
```

```
C x = new C { bar = "oof", foo = 3 };
```

vgl. <http://msdn.microsoft.com/en-us/library/vstudio/bb384062.aspx>

Das funktioniert nicht für unveränderliche
(readonly) Attribute.

(Dafür wird es gute Gründe geben, aber mir fallen
keine ein.)

Rekursion über Listen

```
and :: [ Bool ] -> Bool
```

```
and xs = case xs of
```

```
    [] -> True ; x : xs' -> x && and xs'
```

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

```
length xs = case xs of
```

```
    [] -> 0 ; x : xs' -> 1 + length xs'
```

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

```
fold nil cons xs = case xs of
```

```
    [] -> nil
```

```
    x : xs' -> cons x ( fold nil cons xs'
```

```
and = fold True (&&)
```

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

```
length = fold 0 ( \ _ 1 -> 1 + 1 )
reverse = fold [] ( \ x ys ->
```

Rekursionsmuster (Merksätze)

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

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

Rekursion über Listen (Übung)

das vordefinierte Rekursionsschema über Listen ist:

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

$$\text{length} = \text{foldr} (\ \backslash x y \rightarrow 1 + y) 0$$

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 { key :: a }
  | Branch { left :: Tree a, right :: 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:
 - ▶ Anzahl der Branch-Knoten ist ungerade (nicht zählen!)

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)
```

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 Schlüsseln  $2^d * (c - 1) .. 2^d * c - 1$   
    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 $\text{Tree}\langle A \rangle$)

entspricht einem Tupel von Funktionen

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

Empfangen eines Besuchers:

durch jeden Teilnehmer des Kompositums

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

Implementierung

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 Binärzahl, Bsp: $(1, 1, 0, 1) \implies 13$

Quelltexte aus Vorlesung:

```
GIT_SSL_NO_VERIFY=true git clone https://
```

(bis ich dort ein ordentlich signiertes Zertifikat habe)

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 Sp`

`l1 = Cons 4 Nil ; g l1 = False -- nach Sp`

`g (Cons 2 l0) = False -- nach Spezifikati`

`g (Cons 2 l0) = cons 2 (g l0) = cons 2 Fa`

`g (Cons 2 l1) = True -- nach Spezifikatio`

`g (Cons 2 l1) = cons 2 (g l1) = cons 2 Tr`

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 (bedarfsgesteuerter 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)
```

ersetze Daten durch Unterprogr., die Daten produzieren

- ▶ FP: lazy evaluation

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

Realisierung: Termersetzung →

Beispiel Bedarfsauswertung

```
nats = nf 0 ; nf n = n : nf (n+1)
head (x:xs) = x ; tail (x:xs) = xs
```

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

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

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

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

in Haskell:

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

Strictness (Beispiele)

- ▶ `length :: [a] -> Int` ist strict:

`length undefined ==> exception`

- ▶ `(:)` :: `a -> [a] -> [a]` ist nicht strict 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`

Realisierung der Bedarfsauswertung

Begriffe:

- ▶ *nicht strikt*: nicht zu früh auswerten
- ▶ *lazy*: höchstens einmal auswerten

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

Bedarfsauswertung in Scala

```
object L {  
  def F (x : Int) : Int = {  
    println ("F", x) ; x*x  
  }  
  def main (args : Array[String]) {  
    lazy val a = F(3);  
    println ("here")  
    println (a);  
  } }  
}
```

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

Diskussion

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

Primzahlen

```
primes :: [ Int ]  
primes = sieve $ enumFrom 2
```

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

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

Rekursive Stream-Definitionen

```
naturals = 0 : map succ naturals
```

```
fibonacci = 0
```

```
  : 1
```

```
  : zipWith (+) fibonacci ( tail fibonacci )
```

```
bin = False
```

```
  : True
```

```
  : concat ( map ( \ x -> [ x, not x ] )  
            ( tail bin ) )
```

Die Thue-Morse-Folge

$t := \lim_{n \rightarrow \infty} \tau^n(0)$ für $\tau : 0 \mapsto 01, 1 \mapsto 10$

$t = 0110100110010110 \dots$

t ist kubikfrei

Abstandsfolge $v := 210201210120 \dots$

ist auch Fixpunkt eines Morphismus

v ist quadratfrei

Traversieren

```

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

fold :: ...

largest :: Ord a => Tree a -> a
replace_all_by :: a -> Tree a -> Tree a
replace_all_by_largest
  :: Ord a => Tree a -> Tree a

```

die offensichtliche Implementierung

```

replace_all_by_largest t =
  let l = largest t
  in  replace_all_by l t

```

Implementierung des `replace_all_by_largest`

Motivation (Wdhlg.)

Unix:

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

Haskell:

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

C#:

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

- ▶ logische Trennung:
Produzent → Transformator(en) → Konsument
- ▶ wegen Speichereffizienz: verschränkte Auswertung.
- ▶ gibt es bei *lazy* Datenstrukturen geschenkt wird

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 (); ...
}
```

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>() {
            int s = 0;
            public Integer next() {
                int res = s ; s++; return res;
            }
            public boolean hasNext() { return true; }
        };
    }
};

for (int x : nats) { System.out.println(x); }
```

Aufgabe: implementiere (und benutze) eine Methode

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

Enumerator (C#)

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

typische Benutzung: ...

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

Iteratoren mit yield

```
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:

Streams in C#: funktional, Linq

Funktional

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

Typ von Select? Implementierung?

Linq-Schreibweise:

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

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

Streams in Java8

API-Definition: <http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>,

Kommentar: Dave Hartveld <http://blog.hartveld.com/2013/03/jdk-8-33-stream-api.html>

```
List<Block> blocks = ...;
```

```
int sumOfWeights = 0;
for (Block block : blocks) {
    if (block.getColor() == Color.RED) {
        sumOfWeights += block.getWeight();
    }
}
```

```
int sumOfWeights = blocks.stream()
    .filter(b -> b.getColor() == Color.F
```

Motivation: Sortieren/Vergleichen)

Einfügen (in monotone Liste)

```
insert :: Int -> [Int] -> [Int]
```

```
insert x ys = case ys of
```

```
  [] -> [x] ; y : ys' -> if x < y then ..
```

Sortieren durch Einfügen:

```
sort :: [Int] -> [Int]
```

```
sort xs = foldr insert [] xs
```

Einfügen/Sortieren für beliebige Typen:

mit Vergleichsfunktion als zusätzlichem Argument

```
insert :: (a->a-> Bool) -> a -> [a] -> [a]
```

```
insert lt x ys = ... if lt x y then ...
```

Motivation: Numerik

Skalarprodukt von zwei Vektoren von Zahlen:

```
skal :: [Int] -> [Int] -> Int
skal xs ys = case (xs,ys) of
  ( [], [] ) -> 0
  ( x:xs', y:ys' ) -> x*y + skal xs' ys'
```

Skalarprodukt für beliebige Vektoren:
mit *Wörterbuch (dictionary)* als Argument

```
data Num_Dict a = Num_Dict { zero :: a
  , plus :: a -> a -> a
  , times :: a -> a -> a }
skal :: Num_Dict a -> [a] -> [a] -> a
skal d xs ys = ...
```

Typklassen (Definition, Verwendung)

- ▶ jede Typklasse definiert einen Wörterbuchtyp

```
class Num a where plus :: a -> a -> a
```

- ▶ Instanzen definieren Wörterbücher

```
instance Num Int where plus = ...
```

- ▶ Wörterbücher werden *implizit* übergeben

- ▶ Benutzung von Wörterbüchern steht *explizit* als *Constraint* (Einschränkung) im Typ

```
skal :: Num a => [a] -> [a] -> a
skal xs ys = ...
```

Durch Typklassen erhält man *eingeschränkt polymorphe* Funktionen.

Der Typ von sort

zur Erinnerung:

```
sort = inorder . foldr insert Leaf mit
insert x t = case t of
  Branch {} -> if x < key t then ...
```

Für alle a , die für die es eine Vergleichs-Funktion gibt, hat `sort` den Typ `[a] -> [a]`.

```
sort :: Ord a => [a] -> [a]
```

Hier ist `Ord` eine *Typklasse*, so definiert:

```
class Ord a where
  compare :: a -> a -> Ordering
data Ordering = LT | EQ | GT
```

Instanzen

Typen können Instanzen von *Typklassen* sein.
 (OO-Sprech: Klassen implementieren Interfaces)
 Für vordefinierte Typen sind auch die meisten
 sinnvollen Instanzen vordefiniert

```
instance Ord Int ; instance Ord Char ; ..
```

weiter Instanzen kann man selbst deklarieren:

```
data Student = Student { vorname    :: Stri
                        , nachname  :: Stri
                        , matrikel  :: Int
                        }
}
```

```
instance Ord Student where
  compare s t =
```

Typen und Typklassen

In Haskell sind diese drei Dinge *unabhängig*

1. Deklaration einer Typklasse (= Deklaration von abstrakten Methoden)

```
class C where { m :: ... }
```

2. Deklaration eines Typs (= Sammlung von Konstruktoren und konkreten Methoden)

```
data T = ...
```

3. Instanz-Deklaration (= Implementierung der abstrakten Methoden)

```
instance C T where { m = ... }
```

In Java sind 2 und 3 nur *gemeinsam* möglich

```
class T implements C { ... }
```

Wörterbücher

Haskell-Typklassen/Constraints...

```
class C a where m :: a -> a -> Foo
```

```
f :: C a => a -> Int
```

```
f x = m x x + 5
```

...sind Abkürzungen für Wörterbücher:

```
data C a = C { m :: a -> a -> Foo }
```

```
f :: C a -> a -> Int
```

```
f dict x = ( m dict ) x x + 5
```

Für jedes Constraint setzt der Compiler ein
Wörterbuch ein

Wörterbücher (II)

```
instance C Bar where m x y = ...
```

```
dict_C_Bar :: C Bar
```

```
dict_C_Bar = C { m = \ x y -> ... }
```

An der aufrufenden Stelle ist das Wörterbuch *statisch* bekannt (hängt nur vom Typ ab).

```
b :: Bar ; ... f b ...
```

```
==> ... f dict_C_bar b ...
```

Vergleich Polymorphie

- ▶ Haskell-Typklassen:
statische Polymorphie,
Wörterbuch ist zusätzliches Argument der
Funktion
- ▶ OO-Programmierung:
dynamische Polymorphie,
Wörterbuch ist im Argument-Objekt enthalten.
(OO-Wörterbuch = Methodentabelle der Klasse)

Klassen-Hierarchien

Typklassen können in Beziehung stehen.

Ord ist tatsächlich „abgeleitet“ von Eq:

```
class Eq a where
  (==) :: a -> a -> Bool
```

```
class Eq a => Ord a where
  (<)  :: a -> a -> Bool
```

Ord ist Typklasse mit Typconstraint (Eq)

also muß man erst die Eq-Instanz deklarieren, dann die Ord-Instanz.

Jedes Ord-Wörterbuch hat ein Eq-Wörterbuch.

Die Klasse Show

```
class Show a where  
  show :: a -> String
```

vgl. Java: toString()

Die Interpreter Ghci/Hugs geben bei Eingab `exp`
(normalerweise) `show exp` aus.

Man sollte (u. a. deswegen) für jeden selbst
deklarierten Datentyp eine Show-Instanz schreiben.

... oder schreiben lassen: `deriving Show`

Generische Instanzen (I)

```
class Eq a where
    (==) :: a -> a -> Bool
```

Vergleichen von Listen (elementweise)
wenn a in Eq, dann [a] in Eq:

```
instance Eq a => Eq [a] where
    l == r = case (l,r) of
        ( [], [] ) -> True
        ( x : xs, y : ys )
            -> ( x == y ) && ( xs == ys )
        ( _, _ ) -> False
```

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

Generische Instanzen (II)

```
class Show a where
  show :: a -> String
instance Show a => Show [a] where
  show [] = "[]"
  show xs = brackets
           $ concat
           $ intersperse ", "
           $ map show xs

show 1 = "1"
show [1,2,3] = "[1,2,3]"
```

Benutzung von Typklassen bei Smallcheck

Colin Runciman, Matthew Naylor, Fredrik Lindblad:
SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values

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

<http://hackage.haskell.org/package/smallcheck>

Smallcheck—Beispiel

```
import Test.SmallCheck
```

```
assoc op = \ a b c ->  
    op a (op b c) == op (op a b) c
```

```
main = smallCheck 3  
    (assoc ((++) :: [Bool] -> [Bool] -> [Bo
```

Übung: Kommutativität

Typgesteuertes Generieren von Werten

```
class Testable t where ...
```

```
test :: Testable t => t -> Depth -> [Test
```

```
instance Testable Bool where ...
```

```
instance ( Serial a, Testable b )  
    => Testable ( a -> b ) where ...
```

```
test ( \ (x::Int) (y::Int) -> x+y == y+x
```

Generieren der Größe nach

```
class Serial a where
  -- | series d : alle Objekte mit Tiefe d
  series :: Int -> [a]
```

jedes Objekt hat endliche Tiefe, zu jeder Tiefe nur endliche viele Objekte

Die „Tiefe“ von Objekten:

- ▶ algebraischer Datentyp: maximale Konstruktortiefe
- ▶ Tupel: maximale Komponententiefe
- ▶ ganze Zahl n : absoluter Wert $|n|$
- ▶ Gleitkommazahl $m \cdot 2^e$: Tiefe von (m, e)

Kombinatoren für Folgen

```
type Series a = Int -> [a]
```

```
(\/) :: Series a -> Series a -> Series a
```

```
s1 \/ s2 = \ d -> s1 d ++ s2 d
```

```
(><) :: Series a -> Series b -> Series (a
```

```
s1 >< s2 = \ d ->
```

```
do x1 <- s1 d; x2 <- s2 d; return (x1
```

```
cons0 :: a -> Series a
```

```
cons1 :: Serial a
```

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

```
cons2 :: (Serial a, Serial b)
```

Anwendung I: Generierung von Bäumen

Variante A (explizite Implementierung)

```
data Tree a = Leaf | Branch { left :: Tree a
                             , key :: a , right :: Tree a }
instance Serial a => Serial ( Tree a ) where
    series = cons0 Leaf \/ cons3 Branch
```

Variante B (automatische Implementierung)

```
{-# LANGUAGE DeriveGeneric #-}
import Test.SmallCheck
import GHC.Generics
data Tree a = Leaf | Branch { left :: Tree a
                             , key :: a , right :: Tree a }
```

Anwendung II: geordnete Bäume

```
inorder :: Tree a -> [a]
```

```
ordered :: Ord a => Tree a -> Tree a
ordered t =
```

```
    relabel t $ Data.List.sort $ inorder
```

```
relabel :: Tree a -> [b] -> Tree b
```

```
data Ordered a = Ordered ( Tree a )
```

```
instance ( Ord a, Serial a )
```

```
    => Serial (Ordered a ) where
```

```
    series = \ d -> map ordered $ series
```

Weitere Werkzeuge zur Testfallgenerierung

Haskell (typgesteuert, statisch)

- ▶ Smallcheck (Objekte der Größe nach)
- ▶ Lazy-Smallcheck (bedarfsweise)
- ▶ Quickcheck (zufällige Objekte)

OO-Sprachen (typgesteuert, dynamisch—runtime reflection)

- ▶ JCheck

`http://www.jcheck.org/tutorial/`

```
@RunWith(org.jcheck.runners.JCheckRunner.class)
```

```
class SimpleTest {
```

```
    @Test public void m(int i, int j) {
```

üblich für weitere Sprachen, ...

Übung Typklassen und Smallcheck

- ▶ **definiere:** Liste ist monoton steigend

```
increasing :: Ord a => [a] -> Bool
```

(a) explizite Rekursion, (b) mit zipWith

- ▶ teste mit Test.SmallCheck, ob jede Liste monoton ist

- ▶ Einfügen und Suchen in unbal. Suchbaum (mit Tests):

```
insert :: Ord a => a -> Tree a -> Tree
```

```
contains :: Ord a => ...
```

- ▶ **schreibe als fold:**

```
inorder :: Tree a -> [a]
```

- ▶ damit sortieren und Tests dafür

```
instance Show a => Show (Tree a) where
```

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`, `Drop`, `Where`
- ▶ verbrauchen (konsumieren):
 - ▶ `Aggregate`
 - ▶ **Spezialfälle:** `All`, `Any`, `Sum`, `Count`

Strom-Transformationen (1)

elementweise (unter Beibehaltung der Struktur)

Vorbild:

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

Realisierung in C#:

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

Rechenregeln für `map`:

$$\text{map } f \ [] = \dots$$

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

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, ...

Strom-Transformationen (3)

neue Struktur, neue Elemente

Vorbild:

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

Realisierung:

`SelectMany`

Rechenregel (Beispiel):

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

Übung:

Definition des Operators $\gg=$ durch

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

Typ von $\gg=$? Assoziativität? neutrale Elemente?

Strom-Verbraucher

„Vernichtung“ der Struktur

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

Vorbild:

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

in der Version „von links“

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

Realisierung:

```
B Aggregate<A, B>
  (this IEnumerable<A> source,
   ... seed, ... func)
```

Zusammenfassung: C#(Linq) (Semantik)

C# (Linq)	Haskell
Select	map
SelectMany	>>=
Where	filter
Aggregate	foldl

mehr zu Linq: <http://msdn.microsoft.com/en-us/library/bb336768>

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`)
- ▶ Kompilation: `dmcs Foo.cs -r:System.Core`

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>

Iterierte assoziative Operationen

Prinzip: wenn \oplus assoziativ, dann sollte man

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_8 \oplus$$

so auswerten:

$$((x_1 \oplus x_2) \oplus (x_3 \oplus x_4)) \oplus ((x_5 \oplus x_6) \oplus (x_7 \oplus x_8))$$

Beispiel: carry-lookahead-Addierer

(die assoziative Operation ist die Verkettung der Weitergabefunktionen des Carry)

- ▶ beweise Assoziativität von $++$
- ▶ welches ist die assoziative Operation für „(parallele) maximale Präfix-Summe“?

Map/Reduce-Algorithmen

`map_reduce`

```

:: ( (ki, vi) -> [(ko, vm)] ) -- ^ map
-> ( (ko, [vm]) -> [vo] ) -- ^ reduce
-> [(ki, vi)] -- ^ eingabe
-> [(ko, vo)] -- ^ ausgabe

```

Beispiel (word count)

`ki` = Dateiname, `vi` = Dateiinhalt

`ko` = Wort, `vm` = `vo` = Anzahl

- ▶ parallele Berechnung von `map`
- ▶ parallele Berechnung von `reduce`
- ▶ verteiltes Dateisystem für Ein- und Ausgabe

Map-Reduce: Literatur

- ▶ Jeffrey Dean and Sanjay Ghemawat:
MapReduce: Simplified Data Processing on Large Clusters, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
<http://labs.google.com/papers/mapreduce.html>
- ▶ Ralf Lämmel: *Google's MapReduce programming model - Revisited*, Science of Computer Programming - SCP , vol. 70, no. 1, pp. 1-30, 2008 <http://www.systems.ethz.ch/education/past-courses/hs08/map-reduce/reading/>

Übung Stream-Operationen

- ▶ die Funktion `reverse :: [a] -> [a]` als `foldl`

- ▶ die Funktion

`fromBits :: [Bool] -> Integer`,

Beispiel

`fromBits [True, False, False, True, False ...]` 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](http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.7401)

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.

Typisierung von Daten

von untypisiert bis statisch typisiert:

- ▶ Zeichenketten (String), Bytefolgen
- ▶ XML-Baum (DOM) ohne Schema
- ▶ Objekt eines bestimmten Typs, mit bestimmten Attributen

beachte:

- ▶ statische Typisierung ist das anzustrebende Ziel (Typprüfung durch Compiler)
- ▶ wird das nicht erkannt oder nicht erreicht, dann: Typprüfung durch Programm, zur Laufzeit (unsicher, ineffizient)

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", name)  
        .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 (λ custName \rightarrow ...)

Nachteile (vgl. auch <http://xkcd.com/327/>)

- ▶ drei Namensbereiche
- ▶ keine statische Typisierung
- ▶ keine Syntaxprüfung

Noch mehr Sprachen: HTML, Javascript

<http://weblogs.java.net/blog/driscoll/archive/2009/09/26/ajax-tag-events-and-listeners>

```
<h:body> <h:form id="form">
```

```
Echo test: <h:outputText id="out" value="#{listen
```

```
String Length: <h:outputText id="count" value="#{
```

```
<h:inputText id="in" value="#{listenBean.hello}"
```

```
<f:ajax event="keyup" render="out count eventcour
```

```
listener="#{listenBean.update}"/></h:inputText>
```

grundsätzliche Probleme werden dadurch noch verstärkt:

- ▶ jede Sprache: eigene Abstraktionen, eigenes
TypeSystem:

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
```

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
using nwind;
var conn = new SqliteConnection
    ("DbLinqProvider=Sqlite; Data Source=No
Main db = new Main (conn);
```

Datenquelle benutzen:

```
var rs = from c in db.Customers
        select new { c.City, c.Address
foreach (var r in rs) { Console.WriteLine
break; LINQ Notation (C#) ...
```

Übung LINQ und SQLmetal

Quellen:

<https://code.google.com/p/dblinq2007/>

- ▶ **Beispiel-Datenbank herstellen:**

```
sqlite3 Northwind.db3 < create_Northwind.sql
```

...und ansehen (sqlite3 Northwind.db3)

- ▶ **Schnittstellenbeschreibung herstellen**

```
(sqlmetal /namespace:nwind ...) und  
ansehen
```

- ▶ **Hauptprogramm kompilieren und ausführen**

```
mcs db.cs nwind.cs -r:Mono.Data.Sqlite
```

```
mono db.exe
```

Überblick

übliche OO-Sichtweise:

- ▶ Def.: *Entwurfsmuster*: häufige, typische, nützliche Beziehung zw. Klassen, ausgedrückt durch interfaces)

die (von mir) empfohlene („funktionale“) Sicht:

- ▶ Entwurfsmuster realisieren Konzepte der (fktl.) Programmierung (algebraischer Datentyp \Rightarrow Kompositum, fold \Rightarrow Besucher, Bedarfsauswertung \Rightarrow Iterator)
- ▶ simuliert werden Funktionen höherer Ordnung (Unterprogramm als Argument \Rightarrow Objekt mit 1 Methode)

Überblick (II)

es gibt weitere Muster, die spezielle Probleme der OO/imperativen Programmierung lösen:

- ▶ keine Konstruktoren in Interfaces \Rightarrow Fabrik-Muster
- ▶ Objekte mit veränderlichen Attributen
 - ▶ ersetze durch unveränderliche Objekte \Rightarrow Zustands-Muster
 - ▶ externe Reaktion auf Zustandsänderungen \Rightarrow Beobachter-Muster
 - ▶ spätere/mehrmalige Ausführung von Zustandsänderungen \Rightarrow Befehls-Muster

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 ();
```

Entwurfsmuster: Zustand

Zustand eines Objektes = Belegung seiner Attribute

Zustand erschwert Programm-Benutzung und -Verifikation (muß bei jedem Methodenaufruf berücksichtigt werden).

Abhilfe: Trennung in

- ▶ Handlungsobjekt (nur Methoden)
- ▶ Zustandsobjekt (nur Daten)
- ▶ Zustands-Objekte sind unveränderlich

daraus ergibt sich:

- ▶ jede Methode bekommt Zustandsobjekt als Argument
- ▶ ... und liefert jeweils neues Zustandsobjekt

Impliziter und expliziter Zustand, Bsp. 1

- ▶ Zustand implizit

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

- ▶ Zustand explizit

```
class C1 {  
    public int step (int z) { return z -  
}
```

Impliziter und expliziter Zustand, Bsp. 2

implizit:

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

explizit:

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

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

- ▶ *unveränderliche* Zustandsobjekte:
- ▶ Verwendung früherer Zustandsobjekte (undo, reset, test)

wiederverwendbare Komponenten („Software as Service“) dürfen *keinen* Zustand enthalten.

(Thread-Sicherheit, Load-Balancing usw.)

(vgl.: Unterprogramme dürfen keine globalen Variablen benutzen)

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() >= threshold)
            System.out.println ("alarm");
    }
public static void main(String[] args) {
    Counter c = new Counter (); Watcher w = new Watcher (c);
}

```

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

Befehl

Beispiel:

```
interface ActionListener {  
    void actionPerformed( ActionEvent e);  
}  
JButton b = new JButton ();  
b.addActionListener (new ActionListener()  
    public void actionPerformed (ActionEvent  
    } );
```

trennt Befehls-Erzeugung von -Ausführung,
ermöglicht Verarbeitung von Befehlen (auswählen,
speichern, wiederholen)

Kompositum, Wiederholung

Gestaltung von zusammengesetzten Layouts:

- ▶ `Container extends Component`
- ▶ `JButton extends Component`
- ▶ `Container { add (Component c); }`

Modellierung als algebraischer Datentyp ist:

```
data Component
  = JButton { ... }
  | Container [ Component ]
```

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/](http://www.imn.htwk-leipzig.de/~waldmann/edu/ws06/informatik/manage/)

Strategie

≈ öfter benutzter Befehl, mit Parametern

Beispiel:

```
interface Comparator<T> { int compare (T  
List<Integer> xs = ...;  
Collections.sort  
    (xs, new Comparator<Integer>() { ...
```

Übung:

- ▶ sortiere Strings länge-lexikografisch, ...
- ▶ wo wird Transitivität, Linearität der Relation benutzt?

Strategie (Beispiel II)

```
public class Strat extends JApplet {
    public void init () {
        JPanel p = new JPanel
            (new GridLayout(8,0)); // Strategie
        for (int i=0; i<40; i++) {
            p.add (new JButton ());
        }
        this.getContentPane().add(p);
    }
}
```

Bemerkungen: Kompositum (Wdhlg), MVC (später)

Definition

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/> und Stefan Buchholz: Refactoring (Seminarvortrag)

<http://www.imn.htwk-leipzig.de/~waldmann/edu/ss05/se/talk/sbuchhol/>

Refactoring: Herkunft

Kent Beck: *Extreme Programming*, Addison-Wesley 2000:

- ▶ Paar-Programmierung (zwei Leute, ein Rechner)
- ▶ test driven: erst Test schreiben, dann Programm implementieren
- ▶ Design nicht fixiert, sondern flexibel

Grundlagen: semantikerhaltende Transformationen

- ▶ von Daten (Mengenlehre)
- ▶ von Unterprogrammen (Lambda-Kalkül)

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)

Guter und schlechter Code

- ▶ 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 [http:](http://)

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, vgl.

http://en.wikiquote.org/wiki/Donald_Knuth:

- ▶ Premature optimization is the root of all evil.

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)

Plan

(für restl. Vorlesungen)

- ▶ code smells und Refactoring für Klassen
- ▶ ... für Methoden, Anweisungen
- ▶ Leistungsmessungen und -verbesserungen
- ▶ Zusammenfassung

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 ext;  
};
```

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)
```

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, verstecke 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 (); }
```

Das Fabrik-Muster

```
interface I {}  
class C implements I {}  
class D implements I {}
```

Bei Konstruktion von I-Objekten muß ein konkreter Klassenname benutzt werden.

Wie schaltet man zwischen C- und D-Erzeugung um?

Benutze Fabrik-Objekt: Implementierung von

```
interface F { I create () }
```

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.

Typsichere Aufzählungen

Definition (einfach)

```
public enum Figur { Bauer, Turm, König }
```

Definition mit Attribut (aus JLS)

```
public enum Coin {  
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);  
    Coin(int value) { this.value = value;  
    private final int value;  
    public int value() { return value; }  
}
```

Definition mit Methode:

```
public enum Figur {
```

Vererbung bricht Kapselung

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

Problem: `class C extends B` \Rightarrow

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

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

\Rightarrow 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(HashSet<E>)` mittels Wrapper

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.

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

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

Typenprüfung: gut

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.

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 Anrühigkeiten entwickeln (und für perfekten Code).
- ▶ jede Faktorisierung hat ein Inverses.
(neue Methode deklarieren \leftrightarrow 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);
        System.out.println (5.5 + " " + 6);
    }
}
```

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

Aufgaben zu Refactoring (II)

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

... 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

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%  
}
```

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

Antwort: keines, gcc erzeugt identischen
Assemblercode.

Das funktioniert immer für *Endrekursion* (= die letzte

Java ist langsam? Falsch!

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

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

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

Weitere Diskussionen

z. B. Effizienz von Haskell-Code

vgl. <http://article.gmane.org/gmane.comp.lang.haskell.cafe/99002>

selbst wenn dort noch ein Faktor 2 (zu C/Java) zu sehen ist ...

- ▶ Laufzeiteffizienz ist *ein* Kriterium,
- ▶ ein anderes ist, was man durch Beschäftigung mit Sprache/Paradigma *lernt*.
- ▶ Antwort: Abstraktionen.

Übersicht

- ▶ hier: 6 Aufgaben je 10 ... 20 min
- ▶ Prüfung (wahrscheinlich): 4 Aufgaben je 30 min
- ▶ Lösung allein und ohne Unterlagen
- ▶ Korrektur durch Kommilitonen
- ▶ Bewertung: je Aufgabe ca. 3 Teilaufgaben,
je Teilaufgabe: richtig (1P), vielleicht (0.5P),
falsch (0P);
gesamt 18 P, Zensurengrenzen (%): 40, 60, 80,
95

Datentypen

- ▶ Gegeben sind:

```
data Foo = Bar
```

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

```
data Pair a b =
```

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

Geben Sie jeweils alle Elemente dieser Typen an:

```
Maybe (Maybe Bool)      -- 1 P
```

```
Pair Bool (Pair Foo Bool) -- 1 P
```

- ▶ Geben Sie die `Pair a b` entsprechende Java-Klasse an.

1 P

Termersetzung

für das Termersetzungssystem (mit Variablen X, Y)

$$R = \{f(a, b, X) \rightarrow_1 f(X, X, X), g(X, Y) \rightarrow_2 X, g(X, Y) \rightarrow_3 Y\}:$$

- ▶ geben Sie die Signatur an.
- ▶ geben Sie alle Positionen in $f(g(a, b), g(a, b), g(a, b))$ an, auf denen ein b steht
- ▶ für welchen Term t gilt $t \rightarrow_1 f(g(a, b), g(a, b), g(a, b))$?
- ▶ geben Sie eine R -Ableitung von $f(g(a, b), g(a, b), g(a, b))$ nach t (aus der vorigen Teilaufgabe) an.

Pattern Matching, Rekursion

- ▶ Programmieren Sie das logische Oder: 1 P

`oder :: Bool -> Bool -> Bool`

`oder x y = case x of ...`

- ▶ Für den Datentyp `data N = Z | S N` implementieren Sie den Test auf Gleichheit 2 P

`eq :: N -> N -> Bool`

`eq x y = case x of ...`

Rekursionsmuster (Bäume)

Zu dem Datentyp für nichtleere binäre Bäume

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

gehört ein Rekursionsmuster (`fold`).

Geben Sie dessen Typ an.

Geben Sie Typ und Bedeutung dieser Funktion an:

```
fold (\ k -> if k then 1 else 0)
      (\ x k y -> x + y)
```

Schreiben Sie mittels `fold` die Funktion, die einen Baum spiegelt. Test:

Rekursionsmuster (Zahlen)

- ▶ Geben Sie das Rekursionsschema für den Datentyp an:

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

```
fold ::          -- 1 P
```

```
fold           =  -- 1 P
```

Hinweis: Es soll gelten

```
plus x y = fold y S x
```

- ▶ Implementieren Sie `mal x y = ...` 1 P

(Bearbeiten Sie dazu die autotool-Aufgaben!)