Deklarative Programmierung Vorlesung Wintersemester 2009

Johannes Waldmann, HTWK Leipzig

18. Januar 2011

Einleitung

Definition

deklarativ: jedes (Teil-)Programm/Ausdruck hat einen W
(. . . und keine weitere (versteckte) Wirkung).
Werte können sein:

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

Softwaretechnische Vorteile

- Beweisbarkeit: Rechnen mit Programmteilen (= Werte wie in der Mathematik
- Sicherheit: es gibt keine Nebenwirkungen und Wirkun sieht man bereits am Typ
- Wiederverwendbarkeit: durch Entwurfsmuster (= Funktionen höherer Ordnung)
- Effizienz: durch Programmtransformationen im Compi auch für parallele Hardware.

Gliederung der Vorlesung

- Grundlagen: algebraische Datentypen, Pattern Matchi
- Funktionales Programmieren:
  - Fkt. höherer Ordnung, Rekursionsmuster
  - Funktoren, Monaden (Zustand, Nichtdeterminismus, Parser, Ein/Ausgabe)
  - Bedarfsauswertung, unendl. Datenstrukturen
  - fortgeschrittene Datenstrukturen
  - Nebenläufigkeit, Parallelität
- Logisches Programmieren:
  - Wiederholung Prolog (Relationen, Unifikation, Resolution)
  - Mercury (≈ Prolog mit statischen Typen und Modi)

Organisation der LV

- jede Woche eine Vorlesung, eine Übung
- wöchentlich (kleine) Übungsaufgaben
- Projekte (in Gruppen zu je 2 bis 3 Studenten)
- zur Diskussion OPAL-Forum/Wiki benutzen
- Prüfung mündlich, Prüfungsgespräch beginnt mit Projektvorstellung.

Literatur

- `http://haskell.org/` (Sprachdefinition, Werkzeu Tutorials, . . .)
- Entwurfsmuster-Tutorial: `http://www.imn.htwk-leipzig.de/~waldmann draft/pub/hal4/emu/`
- `http://www.realworldhaskell.org` (Buch, Beispielprogramme)
- `http://www.cs.mu.oz.au/research/mercury/`

Daten

Algebraische Datentypen

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

Bezeichnungen:

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

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


Datentyp mit mehreren Konstruktoren

Beispiel (selbst definiert)

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

Bespiele (in Prelude vordefiniert)

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

Fallunterscheidung, Pattern Matching

```
data T = A { foo :: Int }
       | B { bar :: String }
```

Fallunterscheidung:

```
f :: T -> Int
f x = case x of
    A {} -> foo x
    B {} -> length $ bar x
```

Pattern Matching (Bezeichner f, b werden lokal gebunde

```
f :: T -> Int
f x = case x of
    A { foo = f } -> f
    B { bar = b } -> length b
```


Rekursive Datentypen

```
data Tree = Leaf {}
    | Branch { left :: Tree, key :: Int
             , right :: Tree }
full :: Int -> Tree -- vollst. binärer Ba
full h = if h > 0
    then Branch { left = full (h-1)
         , key = h, right = full (h-1) }
    else Leaf { }
leaves :: Tree -> Int
leaves t = case t of
    Leaf   {} -> 1
    Branch {} -> ...
```

Aufgabe: maximal unbalancierte AVL-Bäume

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)

Wiederholung Bäume

```
data Tree = Leaf {}
    | Node { left :: Tree
           , key :: Int, right :: Tree }

nodes :: Tree ->  Int
nodes t = case t of
    Leaf {} -> 0
    Node {} ->
        nodes (left t) + 1 + nodes (right
```

Zusammenhang:

Datentyp	Funktion
zwei Konstruktoren	zwei Zweige
rekursiv (Tree → Tree)	rekursiv (nodes → nodes)

Polymorphie

Container-Datentypen sollten *generisch polymorph* im Inhaltstyp sein

```
data Tree a = Leaf {}
    | Branch { left :: Tree a, key :: a
             , right :: a }
data List a = Nil {}
    | Cons { head :: a, tail :: List a }
```

(Kleinbuchstabe $=$ Typvariable, implizit all-quantifiziert)

eine generisch polymorphe Funktion:

```
append :: List a -> List a -> List a
append xs ys = case xs of
```


Listen

eigentlich:

```
data List a = Nil {}
    | Cons { head :: a, tail :: List a }
```

aber aus historischen Gründen:

```
List a = [a], Nil = [], Cons = (:)
data [a] = [] | (:) { head :: a, tail ::
```

Pattern matching dafür:

```
length :: [a] -> Int
length l = case l of
    []     -> 0
    x : xs -> ...
```


Operationen auf Listen

- append:
  - Definition
  - Beweis Assoziativität, neutrales Element
- reverse:
  - Definition
  - Beweis: `reverse . reverse = id`

Beispiel Suchbäume

t ist Suchbaum $\iff$ Inorder-Reihenfolge ist monoton steigend:

```
inorder :: Tree a -> [a]
inorder t = case t of ...
```

Einfügen eines Elementes in einen Suchbaum:

```
insert :: Ord a => a -> Tree a -> Tree a
insert x t = case t of
inserts :: Ord a => [a] -> Tree a -> Tree
sort xs = inorder $ inserts xs Leaf
```

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:

```
(\ x -> A) B  =  A [x := B]
```

… falls x nicht (frei) in B vorkommt

Typen

für nicht polymorphe Typen: tatsächlicher Argumenttyp muß mit deklariertem Argumenttyp übereinstimmen: wenn $f :: A \to B$ und $x :: A$, dann $(fx) :: B$.

bei polymorphen Typen können der Typ von $f :: A \to B$ und der Typ von $x :: A'$ Typvariablen enthalten. Dann müssen A und A' nicht übereinstimmen, sondern *unfizierbar* sein (eine gemeinsame Instanz besitzen).

$\sigma := \mathrm{mgu}(A, A')$ (allgemeinster Unifikator)
allgemeinster Typ von (fx) ist dann $B\sigma$.
Typ von x wird dadurch spezialisiert auf $A'\sigma$

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

Rekursion über Bäume (Schema)

gemeinsame Form dieser Funktionen:

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

dieses Schema *ist* eine Funktion höherer Ordnung:

```
fold :: ( ... ) -> ( ... ) -> ( Tree a ->
fold leaf branch = \ t -> case t of
  Leaf {} -> leaf
  Branch {} ->
     branch (f (left t)) (key t) (f (righ
summe = fold 0 ( \ l k r -> l + k + r )
```


Rekursionsmuster (Prinzip)

jeden Konstruktor durch eine passende Funktion ersetz

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

Rekursionsmuster instantiieren = (Konstruktor-)Symbol
interpretieren (durch Funktionen) = eine Algebra angeb

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


Ein- und mehrstellige Funktionen

eine einstellige Funktion zweiter Ordnung:

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

Anwendung dieser Funktion:

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

Kurzschreibweisen (Klammern weglassen):

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

Übung:

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

Rekursion über Bäume (Beispiele)

```
data Tree a    = Leaf
   | Branch { left :: Tree a, key :: a, r

summe :: Tree Int -> Int
summe t = case t of
    Leaf {} -> 0
    Branch {} ->
       summe (left t) + key t + summe (ri
preorder :: Tree a -> [a]
preorder t = case t of
    Leaf {} -> []
    Branch {} ->
       key t : inorder (left t) ++ inorde
```


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


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 { key :: a }
  | Branch { left :: Tree a, right :: Tre

fold :: ...
```

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

Rose Trees

```
data Tree a =
    Node { key :: a
         , children :: [ Tree a ]
         }
```

- vgl. HTML/XML-Dokumente
- Binomialbäume
- Übersetzung in binäre Bäume
- Binomialheaps, leftist heaps

das Rekursionsschema für Rose-Trees?

Rekursionsmuster (Peano-Zahlen)

```
data N = Z | S N

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

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


Strukturerhaltende Folds

elementweise Operation:
Argument und Resultat haben gleiche Struktur, aber (mglw.) verschiedene Elemente:

```
map :: (a -> b) -> (Tree a -> Tree b)
map f = fold Leaf
   ( \ l k r -> Branch l (f k) r )

map :: (a -> b) -> ([a] -> [b])
map f = foldr ( \ x ys -> f x : ys ) []

map length [ "foo", "bar" ] = [ 3, 3 ]
```

Ü: Unterschiede zw. `map reverse` und `reverse`

Programmtransformationen

Komposition von Funktionen:
`(f . g) = \ x -> f (g x)`
Ü: Typ von `(.)`. Bem: Notation leider falschherum.
Satz: (wenn `map` „richtig“ definiert ist, gilt:)

- `map id == id`
- `map (f . g) == map f . map g`

Anwendung: Einsparung von Zwischen-Strukturen.

Programmtransformationen (II)

Satz: (wenn `map` „richtig“ definiert ist, gilt:)

- `foldr nil cons . map f == foldr .. ..`
- desgl. für Bäume

Parallele Folds

nach Definition:

```
  foldr f z [x1, x2, x3]
= f x1 (f x2 (f x3 z)))
```

wenn `f` assoziativ ist, dann

```
= f (f x1 x2) (f x3 z)
```

und das kann man parallel ausrechnen.

Map/Reduce

Dean and Gemawat: *Simplified Data Processing on Lar Clusters*, OSDI, 2004.
Ralf Lämmel: *Google's Map/Reduce Programming Mod Revisited*, in: Science of Computer Programming, 2006.
`http://userpages.uni-koblenz.de/~laemmel/MapReduce/`

```
mapReduce :: ( (k1,v1) -> [(k2,v2)] )
          -> ( k2 -> [v2] -> v3 )
          -> ( Map k1 v1 ) -> ( Map k2 v3
mapReduce m r
  = reducePerKey r -- 3. Apply r to each
  . groupByKey     -- 2. Group per key
  . mapPerKey m    -- 1. Apply m to each
```

Werkzeuge zum Testen

Beispiel

```
import Test.QuickCheck

app :: [a] -> [a] -> [a]
app xs ys = case xs of
    []      -> ys
    x : xs' -> x : app xs' ys
assoc :: [Int] -> [Int] -> [Int] -> Bool
assoc xs ys zs =
    app xs (app ys zs) == app (app xs ys)
main :: IO ()
main = quickCheck assoc
```


QuickCheck, SmallCheck, ...

John Hughes, Koen Classen:
Automatic Specification-Based Testing
`http://www.cs.chalmers.se/~rjmh/QuickChec`

- gewünschte Eigenschaften als Funktion (Prädikat):
  $p :: A \to B \to \ldots \to$ Bool
- Testtreiber überprüft $\forall a \in A, b \in B, \ldots : p\ a\ b \ldots$
- dabei werden Wertetupel $(a, b, \ldots)$ *automatisch* erzeu
  - QuickCheck: zufällig
  - SmallCheck: komplett der Größe nach
  - LazySmallCheck: nach Bedarf
- Generatoren für anwendungsspezifische Datentypen

Einordnung

allgemein:

- Beweisen ist besser als Testen
- Testen ist besser als gar nichts
- das Schreiben von Tests ist eine Form des Spezifizier

Vorteile QuickCheck u.ä. gegenüber JUnit u. ä.

- Test (Property) spezifiziert Eigenschaften, nicht Einzelfälle
- Spezifikation getrennt von Generierung der Testfälle
- Generierung automatisch und konfigurierbar

Polymorphie/Typklassen

Einleitung

```
reverse [1,2,3,4] = [4,3,2,1]
reverse "foobar"  = "raboof"
reverse :: [a] -> [a]
```

`reverse` ist polymorph

```
sort [5,1,4,3] = [1,3,4,5]
sort "foobar"  = "abfoor"

sort :: [a] -> [a] -- ??
sort [sin,cos,log]  = ??
```

`sort` ist *eingeschränkt polymorph*

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, h
`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
```

vgl. Java:

```
interface Comparable<T> { int compareTo (
```


Instanzen

Typen können Instanzen von *Typklassen* sein.
(OO-Sprech: Klassen implementieren Interfaces)
Für vordefinierte Typen sind auch die meisten sinnvoller
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 =
    compare (matrikel s) (matrikel 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 Konstruktor und konkreten Methoden) `data T = ...`
3. Instanz-Deklaration (= Implementierung der abstrakte 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 deklarierter 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
    []        == []
        =  True
    (x : xs) == (y : ys)
        =  (x == y) && ( xs == ys )
    _         == _
        =  False
```


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 Smallchec

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

`http://www.cs.york.ac.uk/fp/smallcheck/`

- Properties sehen aus wie bei QuickCheck,
- anstatt zu würfeln (QuickCheck): alle Werte der Größe nach benutzen

Typgesteuertes Generieren von Werten

```
class Testable t where ...

test :: Testable t => t -> IO ()

instance Testable Bool where ...

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

test ( \ (xs :: [Bool] ) ->
         xs == reverse ( reverse xs ) )
```

erfordert in ghci: `:set -XPatternSignatures`

Generieren der Größe nach

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

jedes Objekt hat endliche Tiefe, zu jeder Tiefe nur endli
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)
      => (a -> b -> c) -> Series c
```


Anwendung I: Generierung von Bäumen

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

instance Serial a => Serial ( Tree a ) wh
    series = cons0 Leaf \/ cons3 Branch
```


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

test ( \ (Ordered t :: Ordered Int) -> ..
```


Projekte

Heapgeordente Bäume f. Autotool

- `Baum/Such/Class => Baum/Heap/Class`
- voll balancierte Binärbäume (wie in Heapsort)
- leftist Heaps, • Binomialheaps

Literatur:

- `https://autolat.imn.htwk-leipzig.de/building_autotool.html`
- Chris Okasaki: Purely Functional Data Structures, `http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#cup98`

Monaden

Motivation (I): Rechnen mit Maybe

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

typische Benutzung:

```
case ( evaluate e l ) of
    Nothing -> Nothing
    Just a  -> case ( evaluate e r ) of
        Nothing -> Nothing
        Just b  -> Just ( a + b )
```

äquivalent (mit passendem `(>>=)` und `return`)

```
evaluate e l >>= \ a ->
    evaluate e r >>= \ b ->
        return ( a + b )
```


Motivation (II): Rechnen mit Listen

Kreuzprodukt von `xs :: [a]` mit `ys :: [b]`

```
cross xs ys =
    concat ( map ( \ x ->
              concat ( map ( \ y ->
                          [ (x,y) ]
                       ) ) ys
           ) ) xs
```

äquivalent:

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


Die Konstruktorklasse Monad

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

instance Monad Maybe where
    return = \ x -> Just x
    m >>= f = case m of
        Nothing -> Nothing
        Just x  -> f x

instance Monad [] where
    return = \ x -> [x]
    m >>= f = concat ( map f m )
```

Do-Notation für Monaden

Original:

```
evaluate e l >>= \ a ->
    evaluate e r >>= \ b ->
        return ( a + b )
```

do-Notation (implizit geklammert)

```
do a <- evaluate e l
   b <- evaluate e r
   return ( a + b )
```

anstatt

```
do { ... ; () <- m ; ... }
```

verwende Abkürzung

```
do { ... ; m ; ... }
```


LINQ

LINQ = Language Integrated Query

```
using System; using System.Linq;
using System.Collections.Generic;
public class bar { public static void Mai
Func<int,bool> odd = ( x => (x & 1) == 1
var result =
    from x in new int [] { 1,2,3 }
    from y in new int [] { 4,5,6 }
    where odd (x+y)
    select x*y;
foreach (var r in result)
  { System.Console.WriteLine (r); } } }
```


Die Zustands-Monade

```
import Control.Monad.State

tick :: State Integer ()
tick = do c <- get ; put $ c + 1

evalState ( do tick ; tick ; get ) 0
```

Aufgabe: wie könnte die Implementierung aussehen?

```
data State s a = ...
evalState = ... ; get = ... ; put = ...
instance Monad ( State s ) where ...
```


Parser als Monaden

```
data Parser t a =
     Parser ( [t] -> [(a,[t])] )
```

- Tokentyp `t`, Resultattyp `a`
- Zustand ist Liste der noch nicht verbrauchten Token
- Zustandsübergänge sind nichtdeterministisch
- Kombination von Listen- und Zustandsmonade
- Anwendung: Parser-Kombinatoren

Monaden mit Null

```
import Control.Monad ( guard )
do a <- [ 1 .. 4 ]
   b <- [ 2 .. 3 ]
   guard $ even (a + b)
   return ( a * b )
```

Definition:

```
guard f = if f then return () else mzero
```

Wirkung:

```
guard f >>= \ () -> m = if f then m else
```

konkrete Implementierung:

```
class Monad m => MonadPlus m where
    mzero :: m a ; ...
instance MonadPlus [] where mzero = []
```


Aufgaben zur List-Monade

- Pythagoreische Tripel aufzählen
- Ramanujans Taxi-Aufgabe ($a^3 + b^3 = c^3 + d^3$)
- alle Permutationen einer Liste
- alle Partitionen einer Zahl
  (alle ungeraden, alle aufsteigenden)

Hinweise:

- allgemein: Programme mit `do, <-, guard, retur`
- bei Permutationen benutze:

```
import Data.List ( inits, tails )
    (xs, y:ys ) <- zip (inits l) (tails
```


Die IO-Monade

Modell: `type IO a = State World a`
aber ohne `put` und `get`.

```
readFile :: FilePath -> IO String
putStrLn :: String -> IO ()

main :: IO ()
main = do
  cs <- readFile "foo.bar" ; putStrLn cs
```

Alle „Funktionen“, deren Resultat von der Außenwelt (Systemzustand) abhängt oder diesen ändert, haben Resultattyp `IO ...`
Am Typ einer Funktion erkennt man ihre möglichen Wirkungen bzw. deren garantierte Abwesenheit.

Die Zustands-Monade

Motivation

Für Datentyp

```
data Tree a = Leaf
    | Branch { left :: Tree a, key :: a,
instance Functor Tree where fmap f t = ..
```

schreibe Funktion

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

mit Spezifikation

```
let ( u, ys ) = relabel t xs
in   fmap (const ()) u == fmap (const ())
  && inorder u ++ ys == xs
```

Motivation (II)

Lösung etwa so:

```
relabel t xs = case t of
  Leaf -> ( Leaf, xs )
  Branch {} ->
    let (l, ys) = relabel (left t) xs
        (k, zs) = ( head ys, tail ys)
        (r, ws) = relabel (right t) zs
    in  (Branch {left=l,key=k,right=r} ,
```

Die Teilrechnungen als Aktionen auffassen, die jeweils e
Resultat liefern `l,k,r` und einen Zustand ändern
`xs -> ys -> zs -> ws`.
Verkettung der Zustände durch `>>=` einer geeigneten Monade.

Zustands-Transformatoren

```
data State s a = State ( s -> (a, s) )

next :: State [b] b
next = State $ \ xs -> (head xs, tail xs)

instance Monad ( State s ) where
    return x = State $ \ s -> ( x, s )
    State f >>= g = State $ \ s ->
        let (a, t) = f s ; State h = g a
        in  h t

evalState :: State s a -> s -> a
evalState (State f) s = let (a,t) = f s i
```


Zustands-Transformatoren (Anwendung)

```
relabel :: Tree a -> State [b] (Tree b)
relabel t = case t of
  Leaf -> return Leaf
  Branch {} -> do
    l <- relabel $ left t
    k <- next
    r <- relabel $ right t
    return $ Branch {left=l,key=k,right=r
```

- Zustands(transformator)monade ist mathematisches Modell (Nebenwirkung findet nicht statt, sondern wird modelliert)
- die (modellierte) Nebenwirkung erkennt man am Typ

The "real" world: IO

IO-Beispiel

`IO a` = IO-Aktion mit Resultattyp `a`.

```
import System.Environment ( getArgs )
import Control.Monad ( forM_ )
main :: IO ()
main = do
    argv <- getArgs
    forM_ argv $ \ arg -> do
         cs <- readFile arg ; putStr cs
```

- übersetzen: `ghc --make Cat`
- ausführen: `./Cat *.tex`
- Typ und Implementierung von `forM_`?

Konkretes Modell für IO: Zustand

Änderung des Weltzustandes

```
data World = ...
data IO a = IO ( World -> (a, World) )
```

das Welt-Objekt bezeichnet Welt außerhalb des Programmes

```
f :: World -> ( World, World )
f w = ( putStr "foo" w, putStr "bar" w )
```

Lösungen:

- Haskell: Typ `World` ist *privat*, öffentlich ist nur `IO`
- Clean: Typ `World` ist öffentlich, aber *unique* (nicht verdoppeln, nicht löschen)

Konkretes Modell für IO: reaktiv

- (Haskell-)Programm ist eine Funktion

  `main :: [ Antwort ] -> [ Befehl ]`

- Reihenfolge ist *kein* Schreibfehler, lazy evaluation!
- Betriebsystem ist „Funktion"(mit Nebenwirkungen)

  `os :: Befehl -> Antwort`

- Programm ausführen:

  `let  bs = main $ map os bs`

IO-Übung: find

- Verzeichnis-Inhalt rekursiv ausgeben
- benutze `getDirectoryContents`
- Moral: Haskell als „Skript"-Sprache

```
import System.Directory
import System.Environment
import Control.Monad ( forM_, when )
import Data.List (isPrefixOf)

main :: IO ()
main = do
    args <- getArgs
```



```
    visit args

visit :: [ FilePath ] -> IO ()
visit files = forM_ files $ \ file -> do
    putStrLn file
    d <- doesDirectoryExist file
    when d $ do
        sub <- getDirectoryContents file
        setCurrentDirectory file
        visit $ filter ( not . isPrefixOf
        setCurrentDirectory ".."
```

Bastel-Aufgabe: soweit ergänzen, daß es sich *wirklich* w
`ls -R1` verhält

Bibliotheken und Werkzeuge

Übersicht

Werkzeuge (Auswahl):

- `+RTS -p` Profiling, hpc (Code/Test-Überdeckung)
- haddock (≈ javadoc)
- `http://haskell.org/hoogle/` (API-Suchmaschi
- cabal (Quelltext-Paketmanager), `http://hackage.haskell.org/`

Bibliotheken (Auswahl):

- Control.Monad (guard, forM), Data.List (partition, sort)
- System.Random, Data.Sequence, Data.Set, Data.Ma
- parsec (Kombinator-Parser), snap (Web-App-Server),

– Typeset by FoilTEX –

Haddock

- Quelltext-Annotationen an Deklarationen
- Ausgabe: HTML-Dokumente

Annotationen beziehen sich auf

- folgende (`-- | blah`)
- vorhergehende (`-- ^ blah`)

(Teil-)Deklaration.

Beispiel

```
-- | Die Funktion 'sqr' quadriert ihr Arg
sqr :: Int -- ^ Eingabe
    -> Int -- ^ Ausgabe
```

– Typeset by FoilTEX –

Pseudozufallszahlen (I)

mit explizitem Zustand des Generators:

```
randomR :: ( RandomGen g, Random a )
        => (a,a) -> g -> (a, g)
split :: RandomGen g => g -> (g,g)
```

mit Zustand in der globalen IO-Monade:

```
randomRIO :: Random a
        => (a,a) -> IO a
```

Übungen:

- zufälliger Binärbaum (gegebener Größe)
- zufällige Permutation

– Typeset by FoilTEX –

Pseudozufallszahlen (II)

Liste von Zufallszahlen aus gegebenem Bereich:

```
import System.Random ; import Control.Mon
zufs :: Int -> (Int,Int) -> IO [Int]
zufs n (lo,hi) =
    forM [ 1 .. n ] $ \ k -> randomRIO (l
```

eine zufällige Permutation:

```
perm :: [a] -> IO [a]
perm xs = if null xs then return [] else
    i <- randomRIO (0, length xs - 1)
    let ( pre, this : post ) = splitAt i
    ys <- perm $ pre ++ post
    return $ this : ys
```

– Typeset by FoilTEX –

Container-Datentypen (I)

- Folge: Data.Sequence
- Menge: Data.Set
- Abbildung: Data.Map

Übungen:

- Wörter-Statistik
- Textwürfeln mit Markov-Ketten

– Typeset by FoilTEX –

Container-Datentypen (I)

Wörter-Statistik:

```
import Data.Map (Map)
import qualified Data.Map as M

count :: FilePath -> IO ()
count f = do
    cs <- readFile f
    let m = statistik $ words cs
    putStrLn $ show m
statistik :: [ String ] -> Map String Int
statistik ws = M.fromListWith (+) $ do
     w <- ws
     return ( w, 1 )
```

– Typeset by FoilTEX –

Container: Folgen (I)

- `Prelude.[]` : einfach verkettet
  - head: konstant,
  - last, (!!), (++): linear
- `Data.Sequence` : Fingerbaum
  - head, last: konstant,
  - (!!), (++): logarithmisch

Übung:

- binäres Einfügen,
- damit Sortieren

– Typeset by FoilTEX –

Container: Folgen (II)

benutzt `Prelude.[]`:

```
binsert :: Ord a => a -> [a] -> [a]
binsert x xs =
    if null xs then [x] else
    let ( pre, mid : post ) =
            splitAt ( div (length xs) 2 )
    in  if x < mid
        then binsert x pre ++ mid : post
        else pre ++ mid : binsert x post

bisort :: Ord a => [a] -> [a]
bisort = foldr binsert []
```

– Typeset by FoilTEX –

benutzt Data.Sequence.Seq:

```
import Data.Sequence ( Seq )
import qualified Data.Sequence as S
binsert :: Ord a => a -> Seq a -> Seq a
binsert x xs =
    if S.null xs then S.singleton x else
    let ( pre, midpost ) =
            S.splitAt ( div (S.length xs)
        mid S.:< post = S.viewl midpost
    in  if x < mid
        then binsert x pre S.>< midpost
        else pre S.>< mid S.<| binsert x
bisort :: Ord a => [a] -> Seq a
bisort = foldr binsert S.empty
```


Bemerkungen:

- Testfall: `bisort $ reverse [1 .. 10000]`
- diese Funktion `bisort` ist wirklich nur ein Test. Wenn nur um das Einfügen in geordnete Listen geht, dann sollte man von Anfang an einen Suchbaum verwende

Bedarfs-Auswertung

Motivation: Datenströme

Folge von Daten:

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

aus softwaretechnischen Gründen diese drei Aspekte ir 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
```

- OO: Iterator-Muster

```
Sequence.Range(0,10).Select(n => n*n).Su
```

- FP: lazy evaluation

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


Bedarfsauswertung in Haskell

jeder Funktionsaufruf ist *lazy*:

- kehrt *sofort* zurück
- Resultat ist *thunk*
- thunk wird erst bei Bedarf ausgewertet
- Bedarf entsteht durch Pattern Matching

```
data N = Z | S N
positive :: N ->  Bool
positive n = case n of
    Z -> False ; S {} -> True
x = S ( error "err" )
positive x
```


Strictness

zu jedem Typ T betrachte $T_\perp = \{\perp\} \cup T$
Funktion f heißt *strikt*, wenn $f(\perp) = \perp$.
in Haskell:

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

für Fkt. mit mehreren Argumenten: betrachte Striktheit i jedem Argument einzeln.
Striktheit bekannt $\Rightarrow$ Compiler kann effizienteren Code erzeugen (frühe Argumentauswertung)

Ändern der Striktheit

- durch `seq` Auswertung erzwingen:

  `seq x y` wertet `x` aus (bis oberster Konstruktur feststeht) und liefert dann Wert von `y`

- Annotation `!` in Konstruktor erzwingt Striktheit

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

  Argument von `S` wird vor Konstruktion ausgewertet

- Annotation `~` in Muster entfernt Striktheit:

  ```
  case error "huh" of   (a,b) -> 5
  case error "huh" of ~ (a,b) -> 5
  ```


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/

Primzahlen

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

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

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


Rekursive Stream-Definitionen

```
naturals = 0 : map succ naturals

fibonacci = 0
    : 1
    : zipWith (+) fibonacci ( tail fibona

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

Übungen:

```
concat = foldr ...
map f  = foldr ...
```


Die Thue-Morse-Folge

$t := \lim_{n\to\infty} \tau^n(0)$ für $\tau : 0 \mapsto 01, 1 \mapsto 10$
$t = 0110100110010110\ldots$

t ist kubikfrei

Abstandsfolge $v := 210201210120\ldots$
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
```

durchquert den Baum zweimal.
Eine Durchquerung reicht aus!

Logisches Programmieren

Einleitung

- funktionales Programmieren:
  LISP (John McCarthy, 1957)
  benutzerdefinierte Funktionen,
  definiert durch Gleichungen (Ersetzungsregeln)
  Rechnen = Normalform bestimmen
- logisches Programmieren:
  Prolog (Alain Colmerauer, 1972)
  benutzerdefinierte Relationen (Prädikate),
  definiert durch Schlußregeln (Implikationen).
  Rechnen = Schlußfolgerung (Widerspruch) ableiten

Implementierung (Motivation)

einfacher Prolog-Interpreter in Haskell, benutzt:

- `Maybe`-Monade (bei Unifikation)
- `[]`-Monade (Nichtdeterminismus bei Klausel-Auswahl
- `StateT`-Monaden-Transformator
- `Parser`-Monade

Syntax

- *Symbol*: Variable beginnt mit Großbuchstaben, sonst Funktions- oder Prädikatsymbol.
- *Regel* besteht aus
  Kopf (Konklusion) :: Term, Rumpf (Prämisse) :: [Term]
  `p(X,Z) :- p(X,Y) , p(Y,Z).`
- *Fakt*: Regel mit leerer Prämisse. `p(a,b).   p(b,c)`
- *Anfrage* (Query) :: [Term]   `?- p(X,Y).`
  auffassen als Regel mit falscher Konklusion
  `false :- p(X,Y).`
- *Programm* besteht aus Menge von Regeln (und Fakte und einer Anfrage.

Denotationale Semantik

Bedeutung einer Regel C :- $P_1, \ldots, P_n$
mit Variablen $X_1, \ldots, X_k$ ist:
$\forall X_1 \cdots \forall X_k : (P_1 \wedge \ldots \wedge P_n) \rightarrow C$

beachte: äquiv. Umformung, falls Variablen des Rumpfe nicht in C vorkommen.

Bedeutung eines Programms P mit Regeln $R_1, \ldots, R_i$ u Anfrage Q ist Konjunktion aller Bedeutungen
$[P] := [R_1] \wedge \ldots \wedge [R_i] \wedge [Q]$

beachte: Negation in Bedeutung der Anfrage Q
d. h. $[P]$ = false $\Leftrightarrow$ Anfrage folgt aus Programm.

Operationale Semantik

Bedeutung eines Programmes P wird durch Ableitunge
(Resolution) bestimmt.

Wenn $[P] =$ false abgeleitet werden kann, dann heißt di
Anfrage des Programms *erfolgreich*:

Dann gibt es (wenigstens) eine Belegung der Variablen
Anfrage, mit denen der Widerspruch begründet wird.

```
Programm : p(a,b). p(b,c).
           p(X,Z) :- p(X,Y), p(Y,Z).
Anfrage  : ?- p(a,X).
Antworten: X = b; X = c.
```


Beispiele

Programm:

```
append(nil,Y,Y).
append(cons(X,Y),Z,cons(X,W)) :-
    append(Y,Z,W).
```

Anfragen:

```
?- append (cons(a,nil),cons(b,nil),Z).
?- append (X,Y,nil).
?- append (X,Y,cons(a,nil)).
?- append (X,X,cons(a,cons(a,nil))).
```


Implementierung

Prinzipien:

- teilweise unbestimmte Terme (Terme mit Variablen)
- Unifikation:

  Terme in Übereinstimmung bringen durch (teilweise) Belegung von Variablen

  angewendet für Anfrageterm und Regelkopf
- Backtracking (Nichtdeterminismus):

  alle Regeln, deren Kopf paßt, der Reihe nach probiere

Substitutionen (Definition)

- Signatur $\Sigma = \Sigma_0 \cup \ldots \Sigma_k$,
- $\mathrm{Term}(\Sigma, V)$ ist kleinste Menge T mit $V \subseteq T$ und $\forall 0 \le i \le k, f \in \Sigma_i, t_1 \in T, \ldots, t_i \in T : f(t_1, \ldots, t_i) \in T$.
- Substitution: partielle Abbildung $\sigma : V \to \mathrm{Term}(\Sigma, V)$,

  Definitionsbereich: $\operatorname{dom} \sigma$, Bildbereich: $\operatorname{img} \sigma$.
- Substitution σ auf Term t anwenden: $t\sigma$
- σ heißt *pur*, wenn kein $v \in \operatorname{dom} \sigma$ als Teilterm in $\operatorname{img} \sigma$ vorkommt.

Substitutionen: Produkt

Produkt von Substitutionen: $t(\sigma_1 \circ \sigma_2) = (t\sigma_1)\sigma_2$

Beispiel 1:
$\sigma_1 = \{X \mapsto Y\}, \sigma_2 = \{Y \mapsto a\}, \sigma_1 \circ \sigma_2 = \{X \mapsto a, Y \mapsto a$
Beispiel 2 (nachrechnen!):
$\sigma_1 = \{X \mapsto Y\}, \sigma_2 = \{Y \mapsto X\}, \sigma_1 \circ \sigma_2 = \sigma_2$
Eigenschaften:

- σ pur $\Rightarrow \sigma$ idempotent: $\sigma \circ \sigma = \sigma$
- σ_1 pur $\wedge$ σ_2 pur impliziert nicht $\sigma_1 \circ \sigma_2$ pur

Implementierung:

```
import Data.Map
type Substitution = Map Identifier Term
times :: Substitution -> Substitution ->
```


Substitutionen: Ordnung

Substitution σ_1 ist *allgemeiner als* Substitution σ_2:
$\sigma_1 \lesssim \sigma_2 \iff \exists \tau : \sigma_1 \circ \tau = \sigma_2$
Beispiele:

- $\{X \mapsto Y\} \lesssim \{X \mapsto a, Y \mapsto a\}$,
- $\{X \mapsto Y\} \lesssim \{Y \mapsto X\}$,
- $\{Y \mapsto X\} \lesssim \{X \mapsto Y\}$.

Eigenschaften

- Relation $\lesssim$ ist Prä-Ordnung (…, …, aber nicht …)
- Die durch $\lesssim$ erzeugte Äquivalenzrelation ist die …

Unifikation—Definition

Unifikationsproblem

- Eingabe: Terme $t_1, t_2 \in \mathrm{Term}(\Sigma, V)$
- Ausgabe: eine allgemeinster Unifikator (mgu): Substitution σ mit $t_1\sigma = t_2\sigma$.

(allgemeinst: minimal bzgl. $\lesssim$)
Satz: jedes Unifikationsproblem ist

- entweder gar nicht
- oder bis auf Umbenennung eindeutig

lösbar.

Unifikation—Algorithmus

$\mathrm{mgu}(s, t)$ nach Fallunterscheidung

- s ist Variable: …
- t ist Variable: symmetrisch
- $s = f(s_1, s_2)$ und $t = g(t_1, t_2)$: …

```
mgu :: Term -> Term -> Maybe Substitution
```

Unifikation—Komplexität

Bemerkungen:

- gegebene Implementierung ist korrekt, übersichtlich, a nicht effizient,
- es gibt Unif.-Probl. mit exponentiell großer Lösung,
- eine komprimierte Darstellung davon kann man aber i Polynomialzeit ausrechnen.

Suche in Haskell

Modellierung von Suche/Nichtdeterminismus in Haskell: Liste von Resultaten, vgl.

```
permutationen :: [a] -> [[a]]
permutationen [] = return []
permutationen (x:xs) = do
    ys <- perms xs
    (pre, post) <-
       zip (inits xs) (tails xs)
    return $ pre ++ x : post
```

Phil Wadler: *How to replace failure by a list of successes—a method for exception handling, backtrack and pattern matching in lazy functional languages.* 1985
http://homepages.inf.ed.ac.uk/wadler/

Ein einfacher Prolog-Interpreter

```
query  :: [Clause] -> [Atom] -> [Substitu
query cs [] = return M.empty
query cs (a : as) = do
    u1 <- single cs a
    u2 <- query cs $ map ( apply u1 ) as
    return $ u1 `times` u2
single :: [Clause] ->  Atom  -> [Substitu
single cs a = do
    c <- cs
    let c' = rename c -- VORSICHT
    u1 <- maybeToList $ unify a $ head c'
    u2 <- query cs $ map ( apply u1 ) $ b
    return $ u1 `times` u2
```


Global eindeutige Namen

bei jeder Benutzung jeder Klausel müssen deren Variab umbenannt werden (= durch „frische“ Namen ersetzt).
Globalen Zähler hinzufügen =
Zustands-Monaden-Transformator anwenden.

```
single :: [Clause] -> Atom -> [Substituti
single cs a = do
    c <- cs

import Control.Monad.State
single :: [Clause] -> Atom
    -> StateT Int [] Substitution
single cs a = do
    c <- lift cs
```


Monaden-Transformator StateT

`data StateT s m a`

- s Zustandstyp
- m zugrundeliegende Monade
- a Resultattyp

Operationen

- `evalStateT :: StateT s m a -> s -> m a`
- `get :: StateT s m s`
- `put :: s -> StateT s m ()`
- `lift :: m a -> StateT s m a`

vgl. S. 36 ff in: Mark P. Jones: *Functional Programming*

Overloading and Higher-Order Polymorphism,
http://web.cecs.pdx.edu/~mpj/pubs/springschool.html

Ideales und Reales Prolog

wie hier definiert (ideal):

- Semantik ist deklarativ
- Reihenfolge der Regeln im Programm und Atome in Regel-Rumpf beeinflußt Effizienz, aber nicht Korrekthe

reales Prolog:

- *cut* (!) zum Abschneiden der Suche
  - green cut: beeinflußt Effizienz
  - red cut: ändert Semantik

  merke: cut $\approx$ goto, grün/rot schwer zu unterscheiden
- Regeln mit Nebenwirkungen (u. a. für Ein/Ausgabe)

für beides: keine einfache denotationale Semantik

Erweiterungen

- eingebaute Operationen (Maschinenzahlen)
- effiziente Kompilation (für Warren Abstract Machine)
- *Modi:* Deklaration von In/Out und Determinismus (Mercury)
- Funktionen/Prädikate höherer Ordnung:

  Lambda-Prolog (Dale Miller) http://www.lix.polytechnique.fr/~dale/lProlog
- statisches Typsystem: Mercury (Fergus Henderson) http://www.mercury.csse.unimelb.edu.au/

Modus-Deklarationen für Prädikate

```
:- mode append (in,in,out) is det.
:- mode append (in,out,in) is semidet.
:- mode append (out,out,in) is multi.
```

Bedeutung Det:

- det: genau eine Lösung
- semidet: höchstens eine Lösung
- multi: unbestimmt (0, 1, mehr)

Bedeutung In/Out:

- In: Argument ist *voll instantiiert* (d.h.: enthält keine Variablen)
- Out: Argument ist *frei* (d.h.: ist Variable)

Verwendung von Modi

- für jedes Prädikat wird eine nichtleere Menge von Mo deklariert
- für jede Benutzung eines Prädikates wird (vom Compi ein passender Modus festgelegt
- Implementierung: Matching statt Unifikation.

Matching-Problem:

- Eingabe: Terme $t_1 \in \mathrm{Term}(\Sigma, V), t_2 \in \mathrm{Term}(\Sigma, \emptyset)$
- Ausgabe: Substitution σ mit $t_1\sigma = t_2$

Motivation: Lesbarkeit, Effizienz — aber:
es gibt Prolog-Programme/Queries,
für die *keine* Modus-Deklarationen existieren.

Theorems ... for Free

Kategorien

mathematisches Beschreibungsmittel für
(Gemeinsamkeiten von) Strukturen
Anwendung in Haskell: Typkonstruktoren als ...

- ... Funktoren (fmap)
- ... Monaden (Kleisli-Kategorie)
- ... Arrows

Ableitung von Regeln:

- Instanzen müssen diese erfüllen,
- anwendbar bei Programmtransformationen

Kategorien (Definition I)

Kategorie C besteht aus:

- Objekten $\mathrm{Obj}(C)$
- Morphismen $\mathrm{Mor}(C)$, jedes $m \in \mathrm{Mor}(C)$ besitzt:
  - Quelle (source) $\mathrm{src}(m) \in \mathrm{Obj}(C)$
  - Ziel (target) $\mathrm{tgt}(m) \in \mathrm{Obj}(C)$

  Schreibweise: $\mathrm{src}(m) \stackrel{m}{\to} \mathrm{tgt}(m)$
- Operation $\mathrm{id} : \mathrm{Obj}(C) \to \mathrm{Mor}(C)$, so daß für alle $a \in \mathrm{Obj}(C)$: $a \stackrel{\mathrm{id}_a}{\to} a$
- Operator $\circ$: wenn $a \stackrel{f}{\to} b \stackrel{g}{\to} c$, dann $a \stackrel{f \circ g}{\to} c$

Kategorien (Definition II)

... und erfüllt Bedingungen:

- $\mathrm{id}.$ sind neutral (auf beiden Seiten)

  für alle $a \stackrel{m}{\to} b$:

  $\mathrm{id}_a \circ m = m = m \circ id_b$
- Verkettung von Morphismen $\circ$ ist assoziativ:

  $(f \circ g) \circ h = f \circ (g \circ h)$

Kategorien: einfache Beispiele

Kategorie der Mengen:

- Objekte: Mengen
- Morphismen: Funktionen

Kategorie der Datentypen:

- Objekte: (Haskell-)Datentypen
- Morphismen: (Haskell-definierbare) Funktionen

Kategorie der Vektorräume (über gegebenem Körper K

- Objekte: Vektorräume über K
- Morphismen: K-lineare Abbildungen

(Übung: Eigenschaften nachrechnen)

Bsp: Kategorie, deren Objekte keine Meng sind

Zu gegebener Halbordnung $(M, \leq)$:

- Objekte: die Elemente von M
- Morphismen: $a \to b$, falls $a \leq b$

(Eigenschaften überprüfen)

unterscheide von:
Kategorie der Halbordnungen:

- Objekte: halbgeordnete Mengen, d. h. Paare $(M, \leq_M)$
- Morphismen: monotone Abbildungen

Punktfreie Definitionen: injektiv

- falls B, C Mengen:

  $g : B \to C$ heißt *injektiv*, wenn
  $\forall x, y \in B : g(x) = g(y) \Rightarrow x = y$.
- in beliebiger Kategorie:

  $g : B \to C$ heißt *monomorph*, (engl.: monic), wenn

  für alle $f : A \to B, f' : A' \to B$:

  aus $f \circ g = f' \circ g$ folgt $f = f'$

Dualer Begriff (alle Pfeile umdrehen) ist *epimorph* (epic
Übung: was heißt das für Mengen?

Punktfreie Definitionen: Produkt

Gegeben $A, B \in \mathrm{Obj}(C)$:
$(P \in \mathrm{Obj}(C), \pi_A : P \to A, \pi_B : P \to B)$ heißt *Produkt* vo
mit B, falls:
für jedes $Q \in \mathrm{Obj}(C), f : Q \to A, g : Q \to B$:
existiert genau ein $h : Q \to P$ mit $f = h \circ \pi_A, g = h \circ \pi_B$.

Übung:

- was bedeutet Produkt in der Kategorie einer Halbordnung?
- welcher Begriff ist dual zu Produkt? (alle Pfeile umdrehen)

Funktoren zwischen Kategorien

Kategorien C, D,
F heißt *Funktor* von C nach D, falls: $F = (F_{\mathrm{Obj}}, F_{\mathrm{Mor}})$ m

- Wirkung auf Objekte: $F_{\mathrm{Obj}} : \mathrm{Obj}(C) \to \mathrm{Obj}(D)$
- Wirkung auf Morphismen: $F_{\mathrm{Mor}} : \mathrm{Mor}(C) \to \mathrm{Mor}(D)$ m
  $g : A \to B \Rightarrow F_{\mathrm{Mor}}(g) : F_{\mathrm{Obj}}(A) \to F_{\mathrm{Obj}}(B)$
- für alle passenden $f, g \in \mathrm{Mor}(C)$ gilt:
  $F_{\mathrm{Mor}}(f \circ g) = F_{\mathrm{Mor}}(f) \circ F_{\mathrm{Mor}}(g)$

Bsp: $C =$ Vektorräume über K, $D =$ Mengen.
Bsp: Funktor von Mengen nach Vektorräumen?

Def: *Endofunktor*: Funktor von C nach C
Bsp: Endofunktoren in der Kategorie einer Halbordnung

(Endo-)Funktoren in Haskell

zur Erinnerung:

- Objekte: Haskell-Typen
- Morphismen: Haskell-Funktionen

Endo-Funktor F:

- F_{Obj}: bildet Typ auf Typ ab,
  d. h: ist *Typkonstruktor* (Beispiel: List-of, Tree-of)
- F_{Mor}: bildet Funktion auf Funktion ab (vom passenden Typ)

```
f :: A -> B;  map f :: [A] -> [B]
map :: (A -> B) -> ([A] -> [B])
```


Funktoren als Typklasse

```
class Functor f where
    fmap :: ( a -> b ) -> ( f a -> f b )

instance Functor [] where
    fmap = map

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

instance Functor Tree where ...
```


Theorems for free

(hier „free“ $=$ kostenlos)
Phil Wadler, ICFP 1989: `http://homepages.inf.ed.ac.uk/wadler/topics/parametricity.html`
Beispiele:

- wenn `f :: forall a . [a] -> [a]`,
  dann gilt für alle `g :: a -> b, xs :: [a]`
  `f (map g xs) == map g (f xs)`
- wenn `f :: forall a . [a] -> a`,
  dann gilt für alle `g :: a -> b, xs :: [a]`
  `f (map g xs) == g (f xs)`

Theorems for free (II)

eine Haskell-Funktion „weiß nichts“ über Argumente vor
polymorphem Typ.
Jedes solche Argument kann vor oder nach Funktionsanwendung transformiert werden.
Dazu ggf. die richtige Funktor-Instanz benötigt.

- freies Theorem für `f :: a -> a`
- freies Theorem für `foldr`
- freies Theorem für `sort :: Ord a => [a] -> [a`
  erhält man nach Übersetzung in uneingeschränkt polymorphe Funktion (mit zusätzlichem Wörterbuch-Argument)

Hintergrund zu Monaden

Kleisli-Kategorie K zu einem Endo-Funktor F einer Kategorie C:

- Objekte von $K =$ Objekte von C
- Morphismen von K: Morphismen in C der Form $A \to F_{\mathrm{Obj}}(B)$

Das wird eine Kategorie, wenn man definiert:

- Komposition
  $\circ_k :: (A_1 \to FA_2) \times (A_2 \to FA_3) \to (A_1 \to FA_3)$
- Identitäten in K: $\mathrm{id}_A : A \to F_{\mathrm{Obj}}A$

so daß die nötigen Eigenschaften gelten (Neutralität, Assoziativität)

Monaden

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

Wenn `m` ein Endo-Funktor ist, dann gilt in der Kleisli-Kategorie von `m`:
Identität id_a ist `return :: a -> m a`
Komposition ist:

```
import Control.Monad

(>=>) :: Monad m
    => (a -> m b) -> (b -> m c) -> (a ->
f (>=>) g = \ x -> ( f x ) >>= g
```

Rechenregeln für Monaden

Kleisli-Kategorie ist wirklich eine Kategorie

- id_a ist neutral bzgl. Komposition
- Komposition ist assoziativ

(Regeln hinschreiben)
Typkonstruktor ist Funktor auf zugrundeliegender Kateg

```
instance Monad m => Functor m where
    fmap f xs = xs >>= ( return . f )
```

(Beweisen, daß das richtigen Typ und richtige Eigenschaften hat)

Rechenregeln (Beispiele)

- Nachrechnen für `Maybe`, für `[]`
- ist das eine Monade?

```
instance Monad [] where -- zweifelhaft
    return x = [x]
    xs >>= f = take 1 $ concat $ map f x
```

- desgl. für „2“ statt „1“?
- Monad-Instanzen für binäre Bäume mit Schlüsseln ...
  - in Verzweigungen
  - in Blättern