Deklarative Programmierung
Vorlesung
Wintersemester 2009

Johannes Waldmann, HTWK Leipzig

25. Januar 2010

Einleitung

Definition

:larativ: jedes (Teil-)Programm/Ausdruck hat einen *Wert*
und keine weitere (versteckte) Wirkung).
rte können sein:

:lassische" Daten (Zahlen, Listen, Bäume. . .)

unktionen (Sinus, . . .)

ktionen (Datei schreiben, . . .)

Softwaretechnische Vorteile

eweisbarkeit: Rechnen mit Programmteilen (= Werten)
'ie in der Mathematik

icherheit: es gibt keine Nebenwirkungen und Wirkungen
eht man bereits am Typ

/iederverwendbarkeit: durch Entwurfsmuster (=
unktionen höherer Ordnung)

ffizienz: durch Programmtransformationen im Compiler,
uch für parallele Hardware.

Gliederung der Vorlesung

irundlagen: algebraische Datentypen, Pattern Matching

unktionales Programmieren:

Fkt. höherer Ordnung, Rekursionsmuster
Funktoren, Monaden (Zustand, Nichtdeterminismus,
Parser, Ein/Ausgabe)
Bedarfsauswertung, unendl. Datenstrukturen
fortgeschrittene Datenstrukturen
Nebenläufigkeit, Parallelität

ogisches Programmieren:

Wiederholung Prolog
(Relationen, Unifikation, Resolution)
Mercury (≈ Prolog mit statischen Typen und Modi)

Organisation der LV

:de Woche eine Vorlesung, eine Übung

'öchentlich (kleine) Übungsaufgaben

rojekte (in Gruppen zu je 2 bis 3 Studenten)

ur Diskussion OPAL-Forum/Wiki benutzen

rüfung mündlich, Prüfungsgespräch beginnt mit
rojektvorstellung.

Literatur

`ttp://haskell.org/` (Sprachdefinition, Werkzeuge,
utorials, . . .)

ntwurfsmuster-Tutorial:
`ttp://www.imn.htwk-leipzig.de/~waldmann/`
`raft/pub/hal4/emu/`

`ttp://www.realworldhaskell.org` (Buch,
eispielprogramme)

`ttp://www.cs.mu.oz.au/research/mercury/`

Daten

Algebraische Datentypen

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

:eichnungen:

`ata Foo` ist Typname

`oo { .. }` ist Konstruktor

`ar, baz` sind Komponenten

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

Datentyp mit mehreren Konstruktoren

spiel (selbst definiert)

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

spiele (in Prelude vordefiniert)

```
:a Bool = False | True
:a Ordering = LT | EQ | GT
```

Fallunterscheidung, Pattern Matching

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

unterscheidung:

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

tern Matching (Bezeichner f, b werden lokal gebunden):

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

Peano-Zahlen

```
:a N = Z | S N
```

efiniere Addition, Multiplikation, Potenz

eweise die üblichen Eigenschaften

Algebren

einer Signatur Σ passende Algebra:

ir jedes Sortensymbol S_k eine nichtleere Menge M_k

ir jedes Funktionssymbol eine Funktion von dem Typ,
er durch Interpretation der Sorten bestimmt wird

spiel (Vektorraum)

$\mapsto \mathbb{R}, V \mapsto \mathbb{R}^3$

ist Addition in $\mathbb{R}$, m ist Multiplikation in $\mathbb{R}$,

ist Vektor-Addition in $\mathbb{R}^3$,

ist Multiplikation (Skalar mal Vektor)

Vergleich Termalgebra/data

ereinstimmungen:

ortensymbol = Typ

unktionssymbol = Konstruktor

veiterungen:

enannte Notation von Konstruktor-Argumenten

olymorphie

Rekursive Datentypen

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

(mehrsortige) Signaturen

orten-Symbole $S = \{S_1, \dots, S_i\}$

unktionssymbole $F = \{f_1, \dots, f_j\}$

weils mit Sorten-Zuordnung $T : F \to S^+$

chreibe $T(f) = [s_1, \dots, s_k]$ als $f : s_1 \times \dots \times s_{k-1} \to s_k$

telligkeit (Arität) von f ist $|T(f)| - 1$.

spiel:

orten K, V

unktionen

$p : [K, K, K]$, d. h. $p : K \times K \to K$
$q : V \times V \to V$, $m : K \times K \to K$, $n : K \times V \to V$

Term-Algebren

gegebener Signatur $\Sigma = (S, F, T)$
rachte Menge der Bäume (Terme) Term(Σ)

der Knoten t ist markiert durch ein $f \in F$

ezeichne $T(f) = [s_1, \dots, s_k, s]$,

ie Sorte von t ist sort$(t) = s =$ last$(T(f))$

hat Kinder $t_1 \dots, t_k$,

nd $\forall i : 1 \le i \le k :$ sort$(t_i) = s_i$

nterpretiere Sortensymbol s_i durch $\{t \mid \text{sort}(t) = s_i\}$

nd Fktssymb. f durch Fkt. $(t_1, \dots, t_k) \mapsto f(t_1, \dots, t_k)$

Polymorphie

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

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

Listen

entlich:

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

ɛr aus historischen Gründen

```
:a [a] = a : [a] | []
```

tern matching dafür:

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

ımme der Elemente einer Liste?

Operationen auf Listen

ppend:

Definition
Beweis Assoziativität, neutrales Element

everse:

Definition
Beweis: `reverse . reverse = id`

Funktionen

Funktionen als Daten

ıer:

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

ıivalent: Lambda-Ausdruck

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

nbda-Kalkül: Alonzo Church 1936, Henk Barendregt
3*, ...

ıktionsanwendung:

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

falls x nicht (frei) in B vorkommt

Ein- und mehrstellige Funktionen

e einstellige Funktion zweiter Ordnung:

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

wendung dieser Funktion:

```
 3) 4 = ...
```

zschreibweisen (Klammern weglassen):

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

ıng:

jeben `t = \ f x -> f (f x)`
stimme `t succ 0, t t succ 0,`
`: t succ 0, t t t t succ 0, ...`

Rekursion über Listen

```
d :: [ Bool ] -> Bool
d l = case l of
  x : xs -> x && and xs ; [] -> True
ıgth :: [ a ] -> Int
ıgth l = case l of
  x : xs ->  1 + length xs ; [] -> 0
ld :: ( a -> b -> b ) -> b -> [a] -> b
ld cons nil l = case l of
  x : xs -> cons x ( fold cons nil xs )
  [] -> nil
d = fold (&&) True
ıgth = fold ( \ x y ->  1 + y) 0
```

Rekursionsmuster (Prinzip)

en Konstruktor durch eine passende Funktion ersetzen.

```
:a N = Z | S N
ld ( z :: b ) ( s :: b -> b ) :: N -> b

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

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

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

Rekursionsmuster (Peano-Zahlen)

```
:a N = Z | S N

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

ıs  = fold ( \ y -> y )
           ( \ f -> \ y -> S ( f y ) )
nes = fold ( \ y -> Z )
           ( \ f y -> plus y ( f y ) )
```

Übungen Rekursionmuster

en: Muster anwenden (append)

ıme: Muster definieren und anwenden

```
:a Tree a
 = Branch (Tree a) a (Tree a) | Leaf
old ( branch ::             ) (leaf ::   )
 :: Tree a -> b
```

Bedarfs-Auswertung

Motivation: Datenströme

ge von Daten:

rzeugen (producer)

ansformieren

erarbeiten (consumer)

; softwaretechnischen Gründen diese drei Aspekte im grammtext trennen,
; Effizienzgründen in der Ausführung verschränken darfsgesteuerter Transformation/Erzeugung)

Bedarfs-Auswertung, Beispiele

nix: Prozesskopplung durch Pipes

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

)O: Iterator-Muster

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

P: lazy evaluation

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

Bedarfsauswertung in Haskell

er Funktionsaufruf ist *lazy*:

ehrt *sofort* zurück

esultat ist *thunk*

unk wird erst bei Bedarf ausgewertet

edarf entsteht durch Pattern Matching

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

Strictness

edem Typ T betrachte $T_\perp = \{\perp\} \cup T$
ktion f heißt *strikt*, wenn $f(\perp) = \perp$.
laskell:

onstruktoren (Cons,…) sind nicht strikt,

estruktoren (head, tail,…) sind strikt.

Fkt. mit mehreren Argumenten: betrachte Striktheit in em Argument einzeln.
ktheit bekannt $\Rightarrow$ Compiler kann effizienteren Code eugen (frühe Argumentauswertung)

Ändern der Striktheit

urch `seq` Auswertung erzwingen:

`eq x y` wertet `x` aus (bis oberster Konstruktur eststeht) und liefert dann Wert von `y`

nnotation `!` in Konstruktor erzwingt Striktheit

```
ata N = Z | S !N
```

rgument von `S` wird vor Konstruktion ausgewertet

nnotation `~` in Muster entfernt Striktheit:

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

Primzahlen

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

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

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

Rekursive Stream-Definitionen

```
turals = 0 : map succ naturals

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

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

ungen:

```
ncat = foldr ...
```

Traversieren

```
ta Tree a = Branch (Tree a) (Tree a)
          | Leaf a
ld :: ...
rgest :: Ord a => Tree a -> a
place_all_by :: a -> Tree a -> Tree a
place_all_by_largest
  :: Ord a => Tree a -> Tree a
```

offensichtliche Implementierung

```
place_all_by_largest t =
  let l = largest t
  in  replace_all_by l t
```

chquert den Baum zweimal.

Monaden

Motivation (I): Rechnen mit Maybe

```
:a Maybe a = Just a | Nothing
```

sche Benutzung:

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

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

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

Die Konstruktorklasse Monad

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

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

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

Monaden mit Null

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

inition:

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

kung:

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

krete Implementierung:

```
ass Monad m => MonadPlus m where
  mzero :: m a ; ...
stance MonadPlus [] where mzero = []
```

Die IO-Monade

```
:a IO -- abstract

adFile :: FilePath -> IO String
:StrLn :: String -> IO ()

stance Functor IO ; instance Monad IO
```

: „Funktionen“, deren Resultat von der Außenwelt stemzustand) abhängt, haben Resultattyp `IO ...`

Typ einer Funktion erkennt man ihre möglichen hädlichen) Wirkungen bzw. deren garantierte wesenheit.

gen der Monad-Instanz: benutze do-Notation

Motivation (II): Rechnen mit Listen

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

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

iivalent:

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

Do-Notation für Monaden

ginal:

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

Notation (implizit geklammert)

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

statt

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

wende Abkürzung

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

Aufgaben zur List-Monade

ythagoreische Tripel aufzählen

amanujans Taxi-Aufgabe ($a^3 + b^3 = c^3 + d^3$)

lle Permutationen einer Liste

lle Partitionen einer Zahl alle ungeraden, alle aufsteigenden)

weise:

llgemein: Programme mit `do, <-, guard, return`

ei Permutationen benutze:

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

Die Zustands-Monade

nn man nur den Inhalt einer Speicherstelle ändern will, in braucht man nicht `IO`, sondern es reicht `State`.

```
port Control.Monad.State

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

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

gabe: wie könnte die Implementierung aussehen?

```
:a State s a = ??
stance Functor ( State s ) where
stance Monad ( State s ) where
```

Parser als Monaden

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

ɔkentyp `t`, Resultattyp `a`

ustand ist Liste der noch nicht verbrauchten Token

ustandsübergänge sind nichtdeterministisch

usführen: `./Cat *.tex`

yp und Implementierung von `forM_`?

Konkretes Modell für IO: reaktiv

Haskell-)Programm ist eine Funktion

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

eihenfolge ist *kein* Schreibfehler, lazy evaluation!

etriebsystem ist „Funktion“(mit Nebenwirkungen)

```
s :: Befehl -> Antwort
```

rogramm ausführen:

```
et  bs = main $ map os bs
```

```
visit args

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

stel-Aufgabe: soweit ergänzen, daß es sich *wirklich* wie
`-R1` verhält

The “real” world: IO

IO-Beispiel

a = IO-Aktion mit Resultattyp `a`.

```
ɔort System.Environment ( getArgs )
ɔort Control.Monad ( forM_ )

in :: IO ()
in = do
  argv <- getArgs
  forM_ argv $ \ arg -> do
        cs <- readFile arg
        putStr cs
```

bersetzen: `ghc --make Cat`

Konkretes Modell für IO: Zustand

derung des Weltzustandes

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

Welt-Objekt bezeichnet Welt außerhalb des
grammes

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

ungen:

askell: Typ `World` ist *privat*, öffentlich ist nur `IO`

lean: Typ `World` ist öffentlich, aber *unique*

IO-Übung: find

erzeichnis-Inhalt rekursiv ausgeben

enutze `getDirectoryContents`

loral: Haskell als „Skript“-Sprache

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

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

Lazy IO

s ist ein sehr dunkles Kapitel)

```
ɔort System.IO
in = do
  h <- openFile "Lazy.hs" ReadMode
  cs <- hGetContents h
  hClose h
  putStr cs
```

`GetContents` liefert einen lazy String,

rst bei Bedarf wird der Handle gelesen.

. falls er dann noch offen ist

Variablen (IORefs)

```
ɔort Data.IORef
ın :: IO ()
ın = do
  x <- newIORef 7
  writeIORef x 8
  a <- readIORef x
  print a
```

ınge Unterscheidung zwischen

erweis (`x :: IORef Integer`)

/ert (`a :: Integer`)

ꞩen und Schreiben sind IO-Aktionen, weil sie den uptspeicherinhalt ändern.

Variablen (STRefs)

```
ɔort Data.STRef; import Control.Monad.ST
ın :: ST s ()
ın = do
  x <- newSTRef 7
  writeSTRef x 8
  a <- readSTRef x
  return a
```

ꞩen und Schreiben sind ST-Aktionen (nicht IO!), weil sie den Hauptspeicherinhalt ändern.

ꞩführen mit beschränkten Nebenwirkungen

`ıST :: ( forall s . ST s a ) -> a`

gleiche: es gibt kein `runIO :: IO a -> a`

Variablen — Ausblick

ↄRefs sind nicht thread-sicher

die üblichen Probleme mit globalen Variablen)

enutze Control.Concurrent.{MVar,Channel}

TM (software transactional memory) für spekulative usführung (atomic transactions)

Projekte

autotool: Erweiterungen, Reparaturen

utotool bauen (ghc/cabal, git)

rweitern:

Intercal-Operationen (Bug 107)
Malbolge-Interpreter (Bug 174)
Datenstrukturen (binäre Bäume → AVL) vgl. `https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?topic=SuchbaumBinary-Quiz`

eparieren `http://dfa.imn.htwk-leipzig.de/ugzilla/buglist.cgi?component=utotool&bug_status=NEW&bug_status=SSIGNED&bug_status=REOPENED`

autotool-Server/Happstack

Challenger-Idee (ca. 2001) vgl.
`:p://www.imn.htwk-leipzig.de/~waldmann/:otool/doc/challenger/node142.html`

sender stellen sich gegenseitig Aufgaben(-Instanzen) zu gegebenem Typ:

lenge von publizierten Instanzen

ktion: Lösung zu einer Instanz einsenden

ktion: neue Instanz einsenden (mit Lösung, die geheim leibt)

vertungskriterien:

eine Instanz, lange ungelöst, …

ıplementierung: benutze autotool-Semantik und

`ttp://happstack.com/`

Ableitungsmuster

egeben: Ersetzungssystem,

esucht: untere Schranken für Ableitungskomplexität,

. h. Ableitungsmuster mit Parametern und induktiven eweisen

spiel $R = \{ab \to ba\}$
ar: $\forall k \ge 0 : a^k b \to^k ba^k$, quadratisch:
$l \ge 0 : a^k b^l \to^{k \cdot l} b^l a^k$,
spiel $S = \{ab \to baa\}$
ar: $\forall k \ge 0 : a^k b \to^* ba^{2k}$
ıonentiell: $\forall l \ge 0 : ab^l \to^* b^l a^{2^l}$

Werkzeuge zum Testen

Beispiel

```
›ort Test.QuickCheck

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

Quickcheck, Smallcheck, ...

ın Hughes, Koen Classen:
omatic Specification-Based Testing
`:p://www.cs.chalmers.se/~rjmh/QuickCheck/`

ewünschte Eigenschaften als Funktion (Prädikat):

:: $A \to B \to \ldots \to$ Bool

esttreiber überprüft $\forall a \in A, b \in B, \ldots : p\ a\ b \ldots$

abei werden Wertetupel $(a, b, \ldots)$ *automatisch* erzeugt:

- QuickCheck: zufällig
- SmallCheck: komplett der Größe nach
- LazySmallCheck: nach Bedarf

ieneratoren für anwendungsspezifische Datentypen

Einordnung

emein:

- eweisen ist besser als Testen
- esten ist besser als gar nichts
- as Schreiben von Tests ist eine Form des Spezifizierens

teile QuickCheck u.ä. gegenüber JUnit u. ä.

- est (Property) spezifiziert Eigenschaften, nicht inzelfälle
- pezifikation getrennt von Generierung der Testfälle
- ienerierung automatisch und konfigurierbar

Beispiel: Test von Sortierverfahren

ıgemäß nach Kap. 11 aus Real World Haskell:
`:p://book.realworldhaskell.org/read/`
`sting-and-quality-assurance.html`

- ım Formulieren der Spezifikationen:

```
st_monoton :: Ord a => [a] -> Bool
st_permutation :: Eq a => [a]->[a]-> Bool
rzeuge_monoton :: [Int] -> [Int]
```

- pezifikationen von: `merge, msort`
- nplementierung von: `merge, msort`
- esttreiber
- estabdeckung

Code Coverage

skell Program Coverage
`:p://projects.unsafeperformio.com/hpc/`

```
: -fhpc --make Sort.hs
Sort
: report Sort
: markup Sort
```

Beispiel: Partitionen

ıktion: ungerade $\leftrightarrow$ steigende

weis:

```
lit :: Int -> (Int, Int)
lit = undefined
›p_split n =
  let (o, e) = split n
  in  odd o && n == o * 2^e

:ict_to_odd :: Partition -> Partition
:ict_to_odd xs = Data.List.sort $ do
  x <- xs ; let ( o, e ) = split x
  k <- [ 1 .. 2^e ] ; return o
```

ıngsaufgabe: `odd_to_strict`

Polymorphie/Typklassen

Einleitung

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

`verse` ist polymorph

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

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

:t ist *eingeschränkt polymorph*

Der Typ von sort

Erinnerung: `sort` enthält:

`: ( low, high ) = partition ( < ) xs in ...`

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

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

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

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

Java:

```
:erface Comparable<T>
  { int compareTo (T o); }
```

Instanzen

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

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

ter Instanzen kann man selbst deklarieren:

```
:a Student = Student { vorname  :: String
                     , nachname :: String
                     , matrikel :: Int
                     }
stance Ord Student where
compare s  t =
  compare (matrikel s) (matrikel t)
```

Typen und Typklassen

Haskell sind diese drei Dinge *unabhängig*

eklaration einer Typklasse (= Deklaration von
bstrakten Methoden)
`lass C where { m :: ... }`

eklaration eines Typs (= Sammlung von Konstruktoren
nd konkreten Methoden) `data T = ...`

nstanz-Deklaration (= Implementierung der abstrakten
Methoden) `instance C T where { m = ... }`

Java sind 2 und 3 nur *gemeinsam* möglich
`ass T implements C { ... }`

s ist an einigen Stellen nachteilig und erfordert Bastelei:
n `class T implements Comparable<T>`, aber
n die T-Objekte anders vergleichen will?
n kann deswegen oft die gewünschte Vergleichsfunktion
arat an Sortier-Prozeduren übergeben.
natürlich nicht die Funktion selbst, Java ist ja nicht
ktional, sondern ihre Verpackung als Methode eines
ekts einer Klasse, die

```
:erface Comparator<T>
 { int compare(T o1, T o2); }
```

lementiert.

Wörterbücher

skell-Typklassen/Constraints. . .

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

:: C a => a -> Int
< = m x x + 5
```

sind Abkürzungen für Wörterbücher:

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

:: C a -> a -> Int
dict x = ( m dict ) x x + 5
```

jedes Constraint setzt der Compiler ein Wörterbuch

Wörterbücher (II)

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

:t_C_Bar :: C Bar
:t_C_Bar = C { m = \ x y -> ... }
```

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

```
:: Bar ; ... f b ...
     ==> ... f dict_C_bar b ...
```

Vergleich Polymorphie

askell-Typklassen:

tatische Polymorphie,

Wörterbuch ist zusätzliches Argument der Funktion

)O-Programmierung:

ynamische Polymorphie,

Wörterbuch ist im Argument-Objekt enthalten.

OO-Wörterbuch = Methodentabelle der Klasse)

Klassen-Hierarchien

klassen können in Beziehung stehen.
l ist tatsächlich „abgeleitet" von Eq:

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

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

l ist Typklasse mit Typconstraint (Eq)
) muß man erst die Eq-Instanz deklarieren, dann die
l-Instanz.
les Ord-Wörterbuch hat ein Eq-Wörterbuch.

Die Klasse Show

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

Java: toString()
Interpreter Ghci/Hugs geben bei Eingab `exp`
rmalerweise) `show exp` aus.
n sollte (u. a. deswegen) für jeden selbst deklarierten
entyp eine Show-Instanz schreiben.
oder schreiben lassen: `deriving Show`

Kanonische Show/Read-Instanzen

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

ine Show-Methode (Instanz) heißt *kanonisch*, wenn `how x` gültiger Haskell-Quelltext ist, dessen uswertung wieder `x` ergibt.

`eriving Show` liefert kanonische Instanzen.

```
ıss Read a where read :: String -> a -- verei
```

ead-Instanz heißt kanonisch, wenn

```
ead (show x) == x
```

`eriving Read` liefert kanonische Instanzen

Die Wahrheit über Read

ndard-Haskell:

```
ıss Read where
  readsPrec :: Int -> ReadS a
ɔe ReadS a = String -> [(a,String)]
```

ist der monadische Parsertyp, aber die Monad-Instanz lt (deswegen keine Do-Notations usw.) — Repariert in C:

```
ıss Read where     ...
  readPrec :: ReadPrec a
```

he `http://www.haskell.org/ghc/docs/latest/` `ıl/libraries/base/` `ɛt-ParserCombinators-ReadPrec.html`

Automatisches Ableiten von Instanzen (I)

```
a Tree a = Branch { key :: a
                  , left :: Tree a
                  , right :: Tree a
                  }
         | Leaf
tance Show a => Show (Tree a) where
 show t @ (Branch {}) =
     "Branch{" ++ "key="  ++ show (key  t) ++ ","
             ++ "left=" ++ show (left t) ++ ","
             ++ "left=" ++ show (left t) ++ "}"
 show Leaf = "Leaf"
```

ıchte: generische Instanz mit Typconstraint

ꞩ kann der Compiler selbst:

```
:a Tree a = ... deriving Show
```

Default-Implementierungen

zielle Definition von `class Ord a`, siehe `http:` `ww.haskell.org/onlinereport/basic.html#sect6.3.2`

```
ss  (Eq a) => Ord a  where
 -- Deklarationen:
 compare              :: a -> a -> Ordering
 (<), (<=), (>=), (>) :: a -> a -> Bool
 -- (gegenseitige) Default-Implementierungen:
 compare x y | x == y    = EQ
             | x <= y    = LT
             | otherwise = GT
 x <= y  = compare x y /= GT
 x <  y  = compare x y == LT
```

ɜicht: Man implementiert entweder `compare` oder `(<=)`, I erhält restliche Methoden durch Defaults.

Generische Instanzen (I)

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

gleichen von Listen (elementweise)

ɲn a in Eq, *dann* [a] in Eq:

```
ɜtance Eq a => Eq [a] where
  []        == []
      =  True
  (x : xs) == (y : ys)
      =  (x == y) && ( xs == ys )
  _         == _
      =  False
```

Generische Instanzen (II)

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

ɜtance Show a => Show [a] where
  show [] = "[]"
  show xs = brackets
          $ concat
          $ intersperse ","
          $ map show xs

ɔw 1 = "1"
ɔw [1,2,3] = "[1,2,3]"
```

Überlappende Instanzen

gen `String = [Char]` gilt bisher

```
ɔw 'f' = "'f'"
ɔw "foo" = "['f','o','o']"
```

ʋünscht ist aber:

```
ɜtance Show String where
  show cs = "\"" ++ cs ++ "\""
ɔw "foo" = "\"foo\""
```

se Instanz-Deklaration *überlappt* mit generischer.

`Show [Char]` gibt es dann zwei Wörterbücher— das ɟrundsätzlich verboten:

Ꞩtandard Haskell dürfen generische Instanzen keine variablen instanziieren.

Überlappende Instanzen (II)

`ghc(i) -XTypeSynonymInstances`

mmandozeile) oder

`language TypeSynonymInstances #-}`

ıelltext, 1. Zeile) sind instantiierte Typvariablen in ɫanzen erlaubt.

`-XOverlappingInstances` gewinnt bei Überlappung speziellere Instanz.

`:` `instance Show [Char]` gewinnt gegen `ɜtance Show [a]`.

Typklassen als Prädikate

n unterscheide *gründlich* zwischen Typen und
klassen (OO: zwischen Klassen und Schnittstellen).

e Typklasse C ist ein (einstelliges) *Prädikat* auf Typen T:
Aussagen $C(T_1), C(T_2), \dots$ sind wahr oder falsch.

ch mehrstellige Prädikate (Typklassen) sind möglich und
nvoll. (Haskell: multi parameter type classes, Java: ?)

Multi-Parameter-Klassen

e Typklasse (Interface) ist ein einstelliges Prädikat. ein
erfüllt es (ist Instanz, implementiert es), oder nicht.

```
ss Ord a where ... ; instance Ord Student where ...
```

benötigt man mehrstellige Prädikate (Relationen)

```
ass Brett b => Zug b z where ...
stance Zug Havannah Satz where ...
```

se werden von *Tupeln* von Typen erfüllt (oder nicht).
ht das in „klassischen“ OO-Sprachen? - Nein.)

n kann zusichern, daß die Relation eine Funktion ist
nctional dependency):

```
ass Problem p i b | (p, i) -> b
```

edem Typ-Paar (p, i) gibt es höchstens ein b mit
`blem p i b`-Wörterbuch.

Benutzung von Typklassen bei Smallcheck

in Runciman, Matthew Naylor, Fredrik Lindblad:
*allCheck and Lazy SmallCheck: automatic exhaustive
ting for small values*

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

roperties sehen aus wie bei QuickCheck,

nstatt zu würfeln (QuickCheck): alle Werte der Größe
ach benutzen

Typgesteuertes Generieren von Werten

```
ass Testable t where ...

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

stance Testable Bool where ...

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

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

ordert in ghci: `:set -XPatternSignatures`

Generieren der Größe nach

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

es Objekt hat endliche Tiefe, zu jeder Tiefe nur endliche
e Objekte

„Tiefe“ von Objekten:

lgebraischer Datentyp: maximale Konstruktortiefe

upel: maximale Komponententiefe

anze Zahl n: absoluter Wert $|n|$

ileitkommazahl $m \cdot 2^e$: Tiefe von (m, e)

Kombinatoren für Folgen

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

/) :: Series a -> Series a -> Series a
 \/ s2 = \ d -> s1 d ++ s2 d
<) :: Series a -> Series b -> Series (a,b)
 >< s2 = \ d ->
  do x1 <- s1 d; x2 <- s2 d; return (x1, x2

ns0 :: a              -> Series a
ns1 :: Serial a
    => (a -> b)       -> Series b
ns2 :: ( Serial a, Serial b)
    => (a -> b -> c) -> Series c
```

Anwendung I: Generierung von Bäumen

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

stance Serial a => Serial ( Tree a ) where
  series = cons0 Leaf \/ cons3 Branch
```

Anwendung II: geordnete Bäume

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

dered :: Ord a => Tree a -> Tree a
dered t =
  relabel t $ Data.List.sort $ inorder t
label :: Tree a -> [b] -> Tree b

ta Ordered a = Ordered ( Tree a )
stance ( Ord a, Serial a )
=> Serial (Ordered a ) where
  series = \ d -> map ordered $ series d
```

Theorems . . . for Free

Kategorien

thematisches Beschreibungsmittel für
meinsamkeiten von) Strukturen
wendung in Haskell: Typkonstruktoren als . . .

. Funktoren (fmap)

. Monaden (Kleisli-Kategorie)

. Arrows

eitung von Regeln:

stanzen müssen diese erfüllen,

nwendbar bei Programmtransformationen

Kategorien (Definition I)

egorie C besteht aus:

bjekten $\mathrm{Obj}(C)$

lorphismen $\mathrm{Mor}(C)$, jedes $m \in \mathrm{Mor}(C)$ besitzt:

Quelle (source) $\mathrm{src}(m) \in \mathrm{Obj}(C)$
Ziel (target) $\mathrm{tgt}(m) \in \mathrm{Obj}(C)$

chreibweise: $\mathrm{src}(m) \xrightarrow{m} \mathrm{tgt}(m)$

peration $\mathrm{id} : \mathrm{Obj}(C) \to \mathrm{Mor}(C)$, so daß für alle
$\in \mathrm{Obj}(C)$: $a \xrightarrow{\mathrm{id}_a} a$

perator $\circ$: wenn $a \xrightarrow{f} b \xrightarrow{g} c$, dann $a \xrightarrow{f \circ g} c$

Kategorien (Definition II)

und erfüllt Bedingungen:

l. sind neutral (auf beiden Seiten)

ir alle $a \xrightarrow{m} b$:

$l_a \circ m = m = m \circ id_b$

erkettung von Morphismen $\circ$ ist assoziativ:

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

Kategorien: einfache Beispiele

egorie der Mengen:

bjekte: Mengen

lorphismen: Funktionen

egorie der Datentypen:

bjekte: (Haskell-)Datentypen

lorphismen: (Haskell-definierbare) Funktionen

egorie der Vektorräume (über gegebenem Körper K)

bjekte: Vektorräume über K

lorphismen: K-lineare Abbildungen

ung: Eigenschaften nachrechnen)

sp: Kategorie, deren Objekte keine Mengen sind

gegebener Halbordnung $(M, \leq)$:

bjekte: die Elemente von M

lorphismen: $a \to b$, falls $a \leq b$

jenschaften überprüfen)

erscheide von:
egorie der Halbordnungen:

bjekte: halbgeordnete Mengen, d. h. Paare $(M, \leq_M)$

lorphismen: monotone Abbildungen

Punktfreie Definitionen: injektiv

lls B, C Mengen:

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

beliebiger Kategorie:

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

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

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

aler Begriff (alle Pfeile umdrehen) ist *epimorph* (epic).
ung: was heißt das für Mengen?

Punktfreie Definitionen: Produkt

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

ung:

as bedeutet Produkt in der Kategorie einer
albordnung?

elcher Begriff ist dual zu Produkt? (alle Pfeile
mdrehen)

Funktoren zwischen Kategorien

egorien C, D,
eißt *Funktor* von C nach D, falls: $F = (F_{\mathrm{Obj}}, F_{\mathrm{Mor}})$ mit

irkung auf Objekte: $F_{\mathrm{Obj}} : \mathrm{Obj}(C) \to \mathrm{Obj}(D)$

irkung auf Morphismen: $F_{\mathrm{Mor}} : \mathrm{Mor}(C) \to \mathrm{Mor}(D)$ mit
$: A \to B \Rightarrow F_{\mathrm{Mor}}(g) : F_{\mathrm{Obj}}(A) \to F_{\mathrm{Obj}}(B)$

ir alle passenden $f, g \in \mathrm{Mor}(C)$ gilt:
$F_{\mathrm{Mor}}(f \circ g) = F_{\mathrm{Mor}}(f) \circ F_{\mathrm{Mor}}(g)$

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

: *Endofunktor*: Funktor von C nach C

(Endo-)Funktoren in Haskell

Erinnerung:

ɔbjekte: Haskell-Typen

lorphismen: Haskell-Funktionen

ɔo-Funktor F:

$_{\mathrm{Obj}}$: bildet Typ auf Typ ab,

. h: ist *Typkonstruktor* (Beispiel: List-of, Tree-of)

$_{\mathrm{Mor}}$: bildet Funktion auf Funktion ab (vom passenden ɣp)

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

Funktoren als Typklasse

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

stance Functor [] where
  fmap = map

:e Tree a
  = Branch ( Tree a ) a ( Tree a )
  | Leaf

stance Functor Tree where ...
```

Theorems for free

ɘr „free“ = kostenlos)

l Wadler, ICFP 1989: `http://homepages.inf.ed.` `.uk/wadler/topics/parametricity.html`

spiele:

ɾenn `f :: forall a . [a] -> [a]`,

ann gilt für alle `g :: a -> b, xs :: [a]`

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

ɾenn `f :: forall a . [a] -> a`,

ann gilt für alle `g :: a -> b, xs :: [a]`

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

Theorems for free (II)

ɘ Haskell-Funktion „weiß nichts“ über Argumente von ɣmorphem Typ.

les solche Argument kann vor oder nach ıktionsanwendung transformiert werden.

zu ggf. die richtige Funktor-Instanz benötigt.

eies Theorem für `f :: a -> a`

eies Theorem für `foldr`

eies Theorem für `sort :: Ord a => [a] -> [a]`

rhält man nach Übersetzung in uneingeschränkt olymorphe Funktion (mit zusätzlichem Vörterbuch-Argument)

Hintergrund zu Monaden

isli-Kategorie K zu einem Endo-Funktor F einer egorie C:

ɔbjekte von K = Objekte von C

lorphismen von K: Morphismen in C der Form $\to F_{\mathrm{Obj}}(B)$

ɜ wird eine Kategorie, wenn man definiert:

omposition $:: (A_1 \to FA_2) \times (A_2 \to FA_3) \to (A_1 \to FA_3)$

lentitäten in K: $\mathrm{id}_A : A \to F_{\mathrm{Obj}}A$

daß die nötigen Eigenschaften gelten (Neutralität, ɜoziativität)

Monaden

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

nn `m` ein Endo-Funktor ist, dann gilt in der isli-Kategorie von `m`:

ntität id_a ist `return :: a -> m a`

nposition ist:

```
ɔort Control.Monad

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

Rechenregeln für Monaden

isli-Kategorie ist wirklich eine Kategorie

l_a ist neutral bzgl. Komposition

omposition ist assoziativ

ɘgeln hinschreiben)

konstruktor ist Funktor auf zugrundeliegender Kategorie

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

ɾweisen, daß das richtigen Typ und richtige enschaften hat)

Rechenregeln (Beispiele)

achrechnen für `Maybe`, für `[]`

t das eine Monade?

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

esgl. für „2“ statt „1“?

lonad-Instanzen für binäre Bäume mit Schlüsseln ...

in Verzweigungen

in Blättern

Arrows

Einleitung

nade M, zur Modellierung einer Rechnung: $c :: a \to Mb$

usgaben beliebig (M anwenden)

ingabe ist immer Funktionsargument

mit geht also folgendes *nicht*:

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

ung:

```
stance Arrow SF where ...
:p://www.haskell.org/arrows/
```

Stream Functions

```
:a SF a b = SF { run :: [a] -> [b] }

: :: ( a -> b ) -> SF a b
: f = SF $ map f

:egrate :: SF Integer Integer
:egrate = SF $ \ xs ->
  let int s [] = []
      int s (x:xs) = (s+x) : int (s+x) xs
  in  int 0 xs
```

Kombinatoren für Stream Functions

quentiell:

```
>>) :: SF a b -> SF b c -> SF a c
 f >>> SF g = SF ( \ xs -> g (f xs) )
```

allel:

```
&&) :: SF a b -> SF a c -> SF a (b,c)
**) :: SF a c -> SF b d -> SF (a,b) (c,d)
:st :: SF a b -> SF (a,c) (b,c)
```

definiere `***` und `&&&` nur durch `first` (und `arr`)

```
||) :: SF a c -> SF b c
    -> SF (Either a b) c
ft  :: SF a b
    -> SF (Either a c) (Either b c)
```

Die Arrow-Klassen

```
ass Category cat where
  id :: cat a a
  (>>>) :: cat a b -> cat b c -> cat a c
ass Category a => Arrow a where
  arr :: (b -> c) -> a b c
  first, second, (&&&), (***)
ass Arrow a => ArrowChoice a where
  left, right, (|||), (+++)
```

proc-Notation für Arrows

```
{ language Arrows #-}
dA :: Arrow a
   => a b Int -> a b Int -> a b Int
dA f g = proc x -> do
      y <- f -< x ; z <- g -< x
      returnA -< y + z
```

d übersetzt in entsprechende punktfreie Notation

```
dA f g = arr (\ x -> (x, x)) >>>
  first f >>> arr (\ (y, x) -> (x, y)) >>>
  first g >>> arr (\ (z, y) -> y + z)
```

und ggf. weiter vereinfacht

```
:p://www.haskell.org/arrows/syntax.html
```

Anwendung von Arrows in HXT

```
:p://www.fh-wedel.de/~si/HXmlToolbox/
:p://www.haskell.org/haskellwiki/HXT/
```

gabe:

```
:oblem type="termination">
:s> <rules> <rule> <lhs>
 <funapp> <name>a</name> <arg> ...
```

gramm:

```
oort Text.XML.HXT.Arrow
:Problem = atTag "problem" >>> proc x -> do
  ty <- getType <<< getAttrValue "type" -<
  rs <- getTRS <<< getChild "trs" -< x
  ...
  returnA -< case st of ...
```

XML-Serialisierung

für algebraische Datentypen

```
:a Tree
  = Branch { left :: Tree, right :: Tree }
  | Leaf { key :: Int }
= Branch { left = Leaf { key = 2 }
         , right = Leaf { key = 3 } }
```

e sollte das als XML aussehen? — So:

```
:ee><branch>
<left><tree><leaf><key><int val="2"/>
      </key></leaf></tree></left> ..
oranch></tree>
```

XML-Serialisierung (II)

nzipien sollten sein:

ollständige Information:

```
Typ><Konstruktor><Attribut>...
```

lementare Werte in Attributen

```
int val="1234"/>
```

bleme bei Abweichen von diesem Prinzipien (z. B.
glassen von „offensichtlichen" Tags)

Logisches Programmieren

Einleitung

nktionales Programmieren:
ISP (John McCarthy, 1957)

enutzerdefinierte Funktionen,
efiniert durch Gleichungen (Ersetzungsregeln)

echnen = Normalform bestimmen

gisches Programmieren:
rolog (Alain Colmerauer, 1972)

enutzerdefinierte Relationen (Prädikate),
efiniert durch Schlußregeln (Implikationen).

echnen = Schlußfolgerung (Widerspruch) ableiten

Syntax

ymbol: Variable beginnt mit Großbuchstaben, sonst unktions- oder Prädikatsymbol.

egel besteht aus

opf (Konlusion) :: Term, Rumpf (Prämisse) :: [Term]

`(X,Z) :- p(X,Y) , p(Y,Z).`

akt: Regel mit leerer Prämisse. `p(a,b).   p(b,c).`

nfrage (Query) :: [Term]   `?- p(X,Y).`

uffassen als Regel mit falscher Konklusion
`alse :- p(X,Y).`

rogramm besteht aus Menge von Regeln (und Fakten) nd einer Anfrage.

Denotationale Semantik

eutung einer Regel C :– $P_1, \ldots, P_n$
Variablen $X_1, \ldots, X_k$ ist:
$\cdots \forall X_k : (P_1 \wedge \ldots \wedge P_n) \to C$

chte: äquiv. Umformung, falls Variablen des Rumpfes t in C vorkommen.

eutung eines Programms P mit Regeln $R_1, \ldots, R_i$ und rage Q ist Konjunktion aller Bedeutungen
$:= [R_1] \wedge \ldots \wedge [R_i] \wedge [Q]$

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

Operationale Semantik

eutung eines Programmes P wird durch Ableitungen solution) bestimmt.

nn $[P]$ = false abgeleitet werden kann, dann heißt die rage des Programms *erfolgreich*:

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

```
gramm : p(a,b). p(b,c).
        p(X,Z) :- p(X,Y), p(Y,Z).
rage  : ?- p(a,X).
worten: X = b; X = c.
```

Beispiele

gramm:

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

ragen:

```
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

nzipien:

ilweise unbestimmte Terme (Terme mit Variablen)

nifikation:

erme in Übereinstimmung bringen durch (teilweise) elegung von Variablen

ngewendet für Anfrageterm und Regelkopf

acktracking (Nichtdeterminismus):

lle Regeln, deren Kopf paßt, der Reihe nach probieren

Substitutionen (Definition)

ignatur $\Sigma = \Sigma_0 \cup \ldots \Sigma_k$,

$\mathrm{erm}(\Sigma, V)$ ist kleinste Menge T mit $V \subseteq T$ und
$0 \le i \le k, f \in \Sigma_i, t_1 \in T, \ldots, t_i \in T : f(t_1, \ldots, t_i) \in T$.

ubstitution: partielle Abbildung $\sigma : V \to \mathrm{Term}(\Sigma, V)$,

efinitionsbereich: $\mathrm{dom}\,\sigma$, Bildbereich: $\mathrm{img}\,\sigma$,

o daß

für alle $v \in \mathrm{dom}\,\sigma : v\sigma \ne v$
kein $v \in \mathrm{dom}\,\sigma$ kommt in $\mathrm{img}\,\sigma$ als Teilterm vor

Substitutionen (Produkt, Ordnung)

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

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

stitution σ_1 ist *allgemeiner als* Substitution σ_2:
$\sigma_2 \iff \exists \tau : \sigma_1 \circ \tau = \sigma_2$
spiele: $\{X \mapsto Y\} \lesssim \{X \mapsto a, Y \mapsto a\}$,
$\mapsto Y\} \lesssim \{Y \mapsto X\}$ und $\{Y \mapsto X\} \lesssim \{X \mapsto Y\}$.
ation $\lesssim$ ist Prä-Ordnung

Unifikation—Definition

fikationsproblem

ingabe: Terme $t_1, t_2 \in \mathrm{Term}(\Sigma, V)$

usgabe: eine allgemeinster Unifikator (mgu):
ubstitution σ mit $t_1\sigma = t_2\sigma$.

gemeinst: minimal bzgl. $\lesssim$)
z: jedes Unifikationsproblem ist

ntweder gar nicht

der bis auf Umbenennung eindeutig

oar.
st Umbenennung: $\mathrm{img}\,\sigma \subseteq$ Variablen)

Suche in Haskell

dellierung von Suche/Nichtdeterminismus in Haskell:
e von Resultaten, vgl.

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

l Wadler: *How to replace failure by a list of
ccesses—a method for exception handling, backtracking,
d pattern matching in lazy functional languages.* 1985.
`tp://homepages.inf.ed.ac.uk/wadler/`

Ideales und Reales Prolog

hier definiert (ideal):

emantik ist deklarativ

eihenfolge der Regeln im Programm und Atome in
egel-Rumpf beeinflußt Effizienz, aber nicht Korrektheit

les Prolog:

ut (!) zum Abschneiden der Suche

green cut: beeinflußt Effizienz
red cut: ändert Semantik

nerke: cut $\approx$ goto, grün/rot schwer zu unterscheiden

egeln mit Nebenwirkungen (u. a. für Ein/Ausgabe)

beides: keine einfache denotationale Semantik

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.
```

deutung Det:

et: genau eine Lösung

emidet: höchstens eine Lösung

nulti: unbestimmt (0, 1, mehr)

deutung In/Out:

n: Argument ist *voll instantiiert* (d.h.: enthält keine
ariablen)

Unifikation—Algorithmus

u(s, t) nach Fallunterscheidung

ist Variable: ...

ist Variable: symmetrisch

$= f(s_1, s_2)$ und $t = g(t_1, t_2)$: ...

nerkungen:

lodellierung in Haskell: Data.Map, Maybe

orrekt, übersichtlich, aber nicht effizient,

s gibt Unif.-Probl. mit exponentiell großer Lösung,

ine komprimierte Darstellung davon kann man aber in
olynomialzeit ausrechnen.

Ein einfacher Prolog-Interpreter

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

Erweiterungen

ingebaute Operationen (Maschinenzahlen)

ffiziente Kompilation (für Warren Abstract Machine)

Modi: Deklaration von In/Out und Determinismus
Mercury)

unktionen/Prädikate höherer Ordnung:

ambda-Prolog (Dale Miller) `http:
/www.lix.polytechnique.fr/~dale/lProlog/`

tatisches Typsystem: Mercury (Fergus Henderson)
`ttp://www.mercury.csse.unimelb.edu.au/`

Verwendung von Modi

ir jedes Prädikat wird eine nichtleere Menge von Modi
eklariert

ir jede Benutzung eines Prädikates wird (vom Compiler)
in passender Modus festgelegt

nplementierung: Matching statt Unifikation.

tching-Problem:

ingabe: Terme $t_1 \in \mathrm{Term}(\Sigma, V), t_2 \in \mathrm{Term}(\Sigma, \emptyset)$

usgabe: Substitution σ mit $t_1\sigma = t_2$

tivation: Lesbarkeit, Effizienz — aber:
gibt Prolog-Programme/Queries

Constraint-Programmierung

Formeln, Modelle

ussagenlogik

- Formel: (in konjunktiver Normalform)
- Modell: Belegung Variable → Boolean

rolog

- Formel: (Regelmenge, Programm)
- Modell: Belegung Variable → Term

ndere Bereiche (z. B. Zahlen)

- Formel: Constraint-System (z. B. Gleichungssystem, Ungleichungssystem; linear, polynomiell)
- Modell: Belegung Variable → Bereich

(CNF-)SAT

s Problem (CNF)-SAT:

- ingabe: aussagenlog. Formel (in CNF)
- rage: gibt es erfüllende Belegung?

enschaften

- t praktisch wichtig (viele Anwendungsaufgaben lassen ch nach SAT übersetzen)
- t schwer (NP-vollständig) ⇒ wahrscheinlich nicht ffizient lösbar
- s gibt erstaunlich gute Heuristiken (Probleme mit 1.000 . 10.000 Variablen lösbar in 1 . . . 100 Sekunden)

gleiche `http://www.satcompetition.org/`

Beispiel SAT-Codierung

```
 | Programm druckt aussagenlogische Formel,
 die das N-Damen-Problem modelliert.
   ghc --make Queens
   ./Queens 8 | minisat /dev/stdin /dev/stdou

ort Control.Monad ( forM)
ort System.Environment

e CNF = [ Clause ] -- verknüpft mit UND
e Clause = [ Literal ] -- verknüpft mit ODER
e Literal = Int -- Bsp: -3 = neg x3, +4 = x4

 n i j = n * i + j + 1
```

```
 n i j = negate $ pos n i j

_jeder_zeile_höchstens_eine :: Int -> CNF
_jeder_zeile_höchstens_eine n = do
 i <- [ 0 .. n-1 ]
 j <- [ 0 .. n-1 ]
 k <- [ j + 1 .. n-1 ]
 return [ neg n i j, neg n i k ]

n = do
 [ arg ] <- System.Environment.getArgs
 let n  = read arg :: Int
     cls = -- in_jeder_zeile_wenigstens_ein
           in_jeder_zeile_höchstens_eine n
```

```
           -- Spalten
           -- Diagonalen
 putStrLn $ unwords [ "cnf", show (n*n)
                    , show ( length cls )
                    ]
 forM cls $ \ cl ->
     putStrLn $ unwords $ map show $ cl ++ [0
 return ()
```

SMT (SAT modulo Theories)

- neare Gleichungen
  - auss-Elimination
- neare Ungleichungen
  - implex-Verfahren, Ellipsoid-Methode
- olynom(un)gleichungen
  - über reellen Zahlen: Satz von Tarski, zylindrische algebraische Zerlegung (QEP-CAD)
  - über ganzen Zahlen: 10. Hilbertsches Problem, Satz von Matiasevich/Roberts

`http://www.smtcomp.org/2009/`

Bitblasting

en von Constraint-System über ganzen Zahlen:

- estlegen einer maximalen Bitbreite
- nbekannte Zahl ⇒ Folge von unbekannten Bits
- rithmetische Operationen ⇒ Bit-Operationen (entspr. chaltkreis-Entwurf)
- ösung durch SAT-Solver

spiel:
`p://dfa.imn.htwk-leipzig.de/satchmo/`

Zusammenfassung Constraint-Programmieren

- iele Aufgaben *sind* Constraint-Systeme (die pezifikation *ist* eine prädikatenlogische Formel)
- erkömmlicher Ansatz: man erfindet und implementiert inen *anwendungsspezifischen* Algorithmus, der das onstraint-System löst
- onstraint-Programmierung: man schreibt das onstraint-System in einer formalen Sprache und benutzt inen *anwendungsunabhängigen* Löser.
- olche Constraint-Systeme sind deklarative Programme.

Nebenläufige und parallele Programme

Definition, Motivation

ebenläufig (concurrent):

ichtdeterminismus, Kommunikation, Synchronisation

uch auf single-Core sinnvoll (Multi-Agenten-Systeme)

arallel (data parallelism):

uswertungsstrategien, die
lultiprozessor/core-Hardware ausnutzten

lealfall: Kommunikation/Synchronisation im
rogrammtext unsichtbar

Threads

```
port Control.Concurrent ( forkIO )
port Control.Monad ( forM_ )
in = do
  forkIO $ forM_ [ 1, 3 .. 100 ] print
  forM_ [ 0, 2 .. 100 ] print
```

ompilieren: `ghc --make Foo -threaded -O2`

usführen: `./Foo +RTS -N2` (benutzt 2 Cores)

IO erzeugt Haskell-Thread, das RTS verteilt diese
Ausführungskontexte (OS/Cores)
in thread fertig ⇒ Programm fertig
forkte Threads werden abgebrochen)

Kommunikation: MVar

ar = mutable variable

rzeugen:

```
ewEmptyMVar :: IO (MVar a)
```

chreiben:

```
utMVar :: MVar a -> a -> IO ()
```

lockiert, solange Variable belegt ist

sen:

```
akeMVar :: MVar a -> IO a
```

lockiert, solange Variable leer ist

Beispiel MVar

ptprogramm wartet, bis ein Thread fertig ist:

```
in = do
  -- nur zum Synchronisieren,
  -- Inhalt ist egal:
  synch :: MVar () <- newEmptyMVar

  forkIO $ do
      forM_ [ 1, 3 .. 100 ] print
      putMVar synch () -- fertig

  forM_ [ 0, 2 .. 50 ] print

  takeMVar synch -- warten
```

Kommunikation: Chan

annel = Ringpuffer, Queue; ohne Kapazitätsschranke

rzeugen:

```
ewChan :: IO (Chan a)
```

chreiben:

```
riteChan :: Chan a -> a -> IO ()
```

lockiert nicht)

sen:

```
eadChan :: Chan a -> IO a
```

lockiert, solange Puffer leer ist

Beispiel Chan

echnet Summe der Zahlen `1..n]` mit mehreren Threads

```
 :: Chan Integer <- newChan

 mehrere Threads starten:
cM [ 1 .. threads ] $ \ t -> forkIO $ do
  let s = sum [ t, t + threads .. n ]
  seq s -- Auswertung erzwingen
        $ writeChan ch s -- dann schreiben

 auf die Werte warten:
 <- forM [1 .. threads] $ \ t -> readChan c
int $ sum ss
```

Programmieren mit Locks

st fehleranfällig:

uwenig Locking: riskiert Inkonsistenzen der Daten

spiel (Vorsicht, fehlerhafter Code):

```
erweise ( betrag :: Geld )
( quelle :: MVar Geld )
( ziel :: MVar Geld ) = do
  q <- readMVar quelle
  when ( q >= betrag ) $ do
      modifyMVar_ quelle $ \ q ->
          return $ q - betrag
      modifyMVar_ ziel   $ \ z ->
```

Programmieren mit Locks

st fehleranfällig:

uviel Locking: riskiert Deadlock wg. *mutual exclusion*,
der wartet auf eine Ressource, die der andere festhält

spiel (Vorsicht, fehlerhafter Code):

```
erweise ( betrag :: Geld )
( quelle :: MVar Geld ) ( ziel :: MVar Geld
  q <- takeMVar quelle
  when ( q >= betrag ) $ do
      z <- takeMVar ziel
      putMVar quelle $ q - betrag
      putMVar ziel   $ z + betrag
```

Transaktionen anstelle von Locks

nsaktion ist eine Folge von Anweisungen:

ls Ganzes (atomar) erfolgreich (commit) oder nicht

ekannt von Implementierung von Datenbanken

ier angewendet auf Speicherzugriffe
oftware transactional memory)

oekulative Ausführung: vor commit prüfen, ob eine
ariable gelesen wurde, die inzwischen von anderer
ansaktion geändert wurde (ja → kein commit, retry)

ansaktionen dürfen keine sichtbaren Nebenwirkungen
aben (Typ ist nicht `IO a`, sondern `STM a`)

etry: Transaktion abbbrechen, später neu starten (wenn
ine der Konfliktvariablen geschrieben wurde)

STM-Beispiel

```
erweise ( betrag :: Geld )
( quelle :: TVar Geld ) ( ziel :: TVar Geld
atomically $ do
  q <- readTVar quelle
  if ( q >= betrag ) then do
      writeTVar quelle $ q - betrag
      z <- readTVar ziel
      writeTVar ziel $ z + betrag
  else retry
```

utzt Funktionen:

```
adTVar    :: TVar a -> STM a
iteTVar   :: TVar a -> a -> STM a
omically :: STM a -> IO a
```

STM-Literatur

osef Svenningsson:
`ttp://computationalthoughts.blogspot.com/`
`008/03/`
`ome-examples-of-software-transactional.`
`tml`

`ttp://www.haskell.org/haskellwiki/`
`oftware_transactional_memory`

Parallelität durch Annotationen

oliothek Concurrent.Parallel)

`` `par` y``:

Berechnung von (whnf von) x starten
Resultat (sofort) ist y

`` `pseq` y``:

Berechnung von (whnf von) x starten
Resultat (wenn fertig) ist y

sche Benutzung: `` a `par` b `pseq` f a b ``

Beispiel: Quicksort mit par/pseq

l. Abschnitt in Real World Haskell)

```
:t [] = []
:t (x:xs) =
let (lo,hi) = Data.List.partition (<x) xs
    slo = sort lo ; shi = sort hi
in  slo ++ x : shi
```

ursive Aufrufe sollten gleichzeitig ausgewertet werden:

```
let ...
in  slo `par` (shi `pseq` (slo ++ x : shi))
```

Par/Pseq-Beispiel

```
{ language PatternSignatures #-}

port Data.List (partition)
port Control.Parallel
port Control.Parallel.Strategies

port System.Environment
port System.Random
port System.IO
port Data.Time.Clock

in = do
  [ n ] <- getArgs
```

```
  gen <- newStdGen
  let xs :: [ Int ] = take ( read n )
         $ randomRs ( 0, 1000 ) gen
  print $ sum xs
  start <- getCurrentTime
  print $ sum $ sort4 2 xs
  end <- getCurrentTime
  print $ diffUTCTime end start


 standard (naives Quicksort, Pivot steht link
:t1 [] = []
:t1 (x:xs) =
```

```
    slo = sort1 lo ; shi = sort1 hi
in  slo ++ x : shi

 parallele Auswertung
:t2 [] = []
:t2 (x:xs) =
let (lo,hi) = Data.List.partition (<x) xs
    slo = sort2 lo ; shi = sort2 hi
in  slo `par` shi `pseq`
        ( slo ++ x : shi )

 ... nur für Rekursionen am Anfang
:t3 d xs | null xs || d <= 0 = sort1 xs
```

```
  let (lo,hi) = partition (<x) xs
      slo = sort3 (d-1) lo
      shi = sort3 (d-1) hi
  in  slo `par` shi `pseq`
               ( slo ++ x : shi )

 ... mit kompletter Forcierung der Resultate
:t4 d xs | null xs || d <= 0 = sort1 xs
:t4 d (x:xs) =
  let (lo,hi) = partition (<x) xs
      slo = sort4 (d-1) lo
      shi = sort4 (d-1) hi
  in  force slo `par` force shi `pseq`
               ( slo ++ x : shi )
```

```
:ce xs = xs `using` rdeepseq
```

Par/PSeq-Zusammendfassung

orteil:

rogramm wird nur annotiert, Form bleibt erhalten,

eine expliziten Threads, MVars etc.

ı beachten:

Steuerung der *Granularität* nötig
(zu fein ⇒ Verwaltung dauert länger als Ausführung)
Erzwingen der Auswertung nötig
par/pseq werten linkes Arg. nur zu whnf aus,
d. h. nur den obersten Konstruktor
stärkere Strategien ⇒ tiefere Auswertung
nützlich:
`http://code.haskell.org/ThreadScope/`

(Nested) Data Parallelism

`:p://www.haskell.org/haskellwiki/GHC/`
`:a_Parallel_Haskell`

ata parallelism: verwende Datenstrukturen mit implizit
arallelen Konstruktoren

```
s :: [: Double :]  ;  [: x * x | x <- xs :]
```

at DP: Vektoren, Arrays

ested DP: geschachtelte/rekursive Typen (Bäume)

nplementierung durch *Vektorisierung* (= Transformation
ı flat DP und zurück)

iterentwicklung (…CUDA-Backend) `http:`
`ıackage.haskell.org/package/accelerate`