Deklarative Programmierung...

- Deklarative Programmierung: logisch vs. funktional

Kombinationen funktionaler und logischer Programmierung...

- Monaden

Zum Ausklang...

- Rückblick auf Haskell und seine Entwicklung(sgeschichte)

Literaturhinweis

Die folgende Darstellung orientiert sich an...

- Michael Spivey, Silvija Seres. *Combinators for Logic Programming*. In Jeremy Gibbons, Oege de Moor (Hrsg.), *The Fun of Programming*. Palgrave MacMillan, 2003.
- Simon Peyton-Jones. *Wearing the Hair Shirt: A Retrospective on Haskell*, eingeladener Vortrag, 30th Annual Symposium on Principles of Programming Languages (POPL'03), 2003.

Deklarative Programmierung

- *Kennzeichend* ...betont das *"was"* weniger das *"wie"*
  - *Essenz* ...Programme deklarative Zusicherungen über ein Problem, weniger imperative Lösungsvorschriften
- *Ausprägungen* ...*funktionale* und *logische* Programmierung
- Frage ...lassen sich funktionale und logische Programmierung kombinieren?

Zur Kombination fkt.&log. Programmierung

- *Klassisch* ...Entwurf neuer Programmiersprachen, die Aspekte beider Programmierstile vereinen (z.B. Curry)
- *Einfacher* ...einen Interpretierer für einen Stil im jeweils anderen schreiben
- *Noch einfacher* ..."logische" Programme in Haskell schreiben mit Hilfe einer Bibliothek von Kombinatoren (vgl. Kombinatoransatz für Parsing)

Zum Kombinatoransatz

- Vor- und Nachteile ...weniger ausdruckskräftig, dafür geringer Aufwand

Zentrale Probleme

- Modellierung logischer Programme mit...
  - mehreren Ergebnissen
  - *logischen* Variablen
- Modellierung der inhärenten Auswertungsstrategie logischer Programme

Lfd. Bsp.: Faktorisierung nat. Zahlen

...zerlegen einer positiven ganzen Zahl in Paare von Faktoren

Beispiel:

Zahl	Faktorpaare
24	(1,24), (2,12), (3,8), (4,6), ..., (24,1)

Offensichtliche Lösung:

```
factor   :: Int -> [(Int,Int)]
factor n =  [(r,s) | r <- [1..24], s <- [1..24], r*s == n]

?factor 24
[(1,24),(2,12),(3,8),(4,6),(6,4),(8,3),(12,2),(24,1)]
```

In der Folge machen wir es uns etwas schwerer...
...auffassen als Suchproblem (im Suchraum [1..] $\times$ [1..])

Zum 1. Problem: Mehrere Ergebnisse

Lösung ...Erfolgslisten (lists of successes)
$\leadsto$lazy lists (Phil Wadler)
(vgl. dazu auch Kombinatorparsing)

```
factor   :: Int -> [(Int,Int)]
factor n =  [(r,s) | r <- [1..], s <- [1..], r*s == n]

?factor 24
[(1,24)

...gefolgt von unendlichem Warten.
```


Faire Durchlaufordnungen

Soforthilfe ...Durchlaufen des Suchraums in fairer Ordnung

```
factor n = [(r,s) | (r,s) <- diagprod [1..][1..], r*s == n]

diagprod       :: [a] -> [b] -> [(a,b)]
diagprod xs ys = [(xs!!i, y!!(n-i) | n <- [0..], i <- [0..n]]

...jedes Paar (x,y) wird nach endlicher Zeit erreicht
[(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3),(3,2),...]

Somit:
?factor 24
[(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),(1,24),(24,1)

...alle Ergebnisse, gefolgt wieder von unendlichem Warten.
```

Systematische Abhilfe: Mittels Monaden

Erinnerung:

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

Vereinbarung für diese Vorlesung:

- `Stream a` ...für potentiell unendliche Listen
- `[a]` ...für endliche Listen
- Bemerkung: In Haskell kein Unterschied. Es gilt:
  `type Stream a = [a]`

Listenmonade

Monade (potentiell unendlicher) Listen

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

(>>=)    :: Stream a -> (a -> Stream b) -> Stream b
xs >>= f =  concat (map f xs)
```


Vorteil 1(2)

...`return` und `(>>=)` erlauben Listenkomprehension auszudrücken:

Betrachte

```
[(x,y) | x <- [1..], y <- [10..]]

   ...ist aequivalent zu

concat (map (\x -> [(x,y) | y <- [10..]])[1..])

   ...ist aequivalent zu

concat (map (\x -> concat (map (\y -> [(x,y)])[10..]))[1..])
```

`return` und `(>>=)` erlauben dies wie folgt abzukürzen

```
[1..] >>= (\x -> [10..] >>= (\y -> return (x,y)))
```


Vorteil 2(2)

Noch kompakter mittels do-Notation:

```
do x <- [1..]; y <- [10..]; return (x,y)
```

Allgemeine Regel:

```
do x1 <- e1; x2 <- e2; ... ; xn <- en; e
   ...ist gleichbedeutend mit
e1 >>= (\x1 -> e2 >>= (\x2 -> ... >>= (\xn -> e)...))
```

Noch nicht gelöst:
...die Durchlaufordnung ist (nach wie vor) nicht fair.

Fairness: Anpassung der Fkt. (>>=) 1(2)

Idee ...Diagonalisierung einbetten in `(>>=)`

Umsetzung

```
newtype Diag a = MkDiag (Stream a) deriving Show

instance Monad Diag where
  return x       = MkDiag[x]
  MkDiag xs >>= f = MkDiag (concat (diag (map (unDiag . f) xs)))

-- Entfernen des Typkonstruktors MkDiag
unDiag (MkDiag xs) = xs

-- Umgruppierung von Werten zu fairer Folge
diag :: Stream (Stream a) -> Stream [a]
diag []       = []
diag (xs:xss) = lzw (++) [ [x] | x <- xs] ([] : diag xss)
-- lzw entspricht zipwith mit Anhaengen der nichtleeren
-- Restliste, falls eine Argumentliste leer wird
```

Fairness 2(2)

Idee hinter `diag`
...überführt eine unendliche Liste unendlicher Listen
`[[x11,x12,x13,...],[x21,x22,...],[x31,x32,...],...]`
...in unendliche Liste endlicher Diagonalen
`[[x11],[x12,x21],[x13,x22,x31],...]`

Damit

```
?do x <- MkDiag [1..]; y <- MkDiag [10..]; return (x,y)
MkDiag[(1,10),(1,11),(2,10),(1,12),(2,11),(3,10),(1,13),...

-- Paare werden jetzt in fairer Ordnung geliefert
```


Zurück zum Faktorisierungsproblem 1(3)

Probleme:

- Generierung (von Paaren in fairer Ordnung): gelöst
- Selektion (der zur Lösung gehörenden Paare): noch offen

Lösungsansatz: ...Filtern mittels Bedingungen

Dazu...

```
class Monad m => Bunch m where
  zero :: m a              -- leeres Ergebnis, keine Antwort
  alt  :: m a -> m a -> m a  -- alle Antworten aus xm oder ym
  wrap :: m a -> m a         -- von Nebenrechn. gelieferte Antworten
                           -- wrap hier zunaechst identisch zu Id
```


Zurück zum Faktorisierungsproblem 2(3)

Im Detail

```
instance Bunch [] where
  zero      = []
  alt xs ys = xs ++ ys
  wrap xs   = xs
```

Oder auch

```
instance Bunch Diag where
  zero                        = MkDiag[]
  alt (MkDiag xs)(MkDiag ys) = MkDiag (shuffle xs ys)
  wrap xm                     = xm

shuffle [] ys     = ys
shuffle (x:xs) ys = x : shuffle ys xs
```

Zurück zum Faktorisierungsproblem 3(3)

Der Schlüssel zum Filtern...

```
test   :: Bunch m => Bool -> m()
test b =  if b then return() else zero
```

Anwendungsbeispiele

```
?do x <- [1..]; () <- test (x 'mod' 3 == 0); return x
[3,6,9,12,15,18,21,24,27,30,33,...

?do x <- MkDiag [1..]; test (x 'mod' 3 == 0); return x
MkDiag[3,6,9,12,15,18,21,24,27,30,33,...
```


Sind wir fertig?

Nein! Betrachte...

```
?do r <- MkDiag[1..]; s <- MkDiag[1..]; test(r*s==24); return (r,s)
MkDiag[(1,24)

...gefolgt von unendlichem Warten.
```

Zur Ursachenforschung...

```
do r <- MkDiag[1..]; s <- MkDiag[1..]; test(r*s==24); return (r,s)
   ist aequivalent zu
do x <- MkDiag[1..]
   (do y <- MkDiag[1..]; test(x*y==24); return (x,y))

   d.h. der Generator fuer y wird mit nachfolgendem Test zu
   folgendem Teilausdruck zusammengefasst
do y <- MkDiag[1..]; test(x*y==24); return (x,y)
```


Lösungsansatz

Tieferliegende Ursache für obiges unerwünschtes Verhalten
Fehlende Assoziativität von (>>=) für Diag.

```
(xm >>= f) >>= g = xm >>= (\x -> f x >>= g)
  ...gilt fuer (>>=) und Diag nicht!
```

Abhilfe ...explizite Anordnung

```
?do (x,y) <- (do u <- MkDiag[1..]; v <- MkDiag[1..]; return (u,v))
    test (x*y==24); return (x,y)
MkDiag[(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),(1,24),(24,1)

...alle Ergebnisse, aber wieder gefolgt von unendlichem Warten
```


Anzeige, dass keine Lösung gefunden wird...

Dazu... Neuer Typ und Breitensuche

Intuition

- Typ `Matrix` ...unendliche Liste endlicher Listen
- Ziel ...Programm, das Matrix von Antworten liefert, wobei Reihe i alle Antworten enthält, die mit Kosten $c(i)$ berechnet werden können.
- Lösung des Anzeigeproblems ...durch Rückgabe der leeren Liste in einer Zeile (entspricht "habe nichts gefunden")

Umsetzung... 1(2)

```
newtype Matrix a = MkMatrix (Stream [a]) deriving Show
-- Hilfsfunktion
unMatrix (MkMatrix xm) = xm

-- Funktionen, um Matrix zur Instanz von Bunch zu machen
return x = MkMatrix[[x]]  -- Matrix mit einer Zeile
zero = MkMatrix[]         -- Matrix ohne Zeilen
alt(MkMatrix xm) (MkMatrix ym) = MkMatrix(lzw (++) xm ym)
wrap(MkMatrix xm) = MkMatrix([]:xm) -- in wrap liegt der Clou!

(>>=) :: Matrix a -> (a -> Matrix b) -> Matrix b
(MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix . f))

bindm :: Stream[a] -> (a -> Stream[b]) -> Stream[b]
bindm xm f = map concat (diag (map (concatAll . map f) xm))

concatAll :: [Stream [b]] -> Stream [b]
concatAll = foldr (lzw (++)) []
```

Umsetzung... 2(2)

Damit...

```
instance Monad Matrix where
  return x          = MkMatrix[[x]]
  (MkMatrix xm) >>= f = MkMatrix(bindm xm (unMatrix . f))

instance Bunch Matrix where
  zero                          = MkMatrix[]
  alt(MkMatrix xm)(MkMatrix ym) = MkMatrix(lzw (++) xm ym)
  wrap(MkMatrix xm)             = MkMatrix([]:xm)

intMat = MkMatrix[[n] | n <- [1..]]
```

Anwendungsbeispiel

```
?do r <- intMat; s <- intMat; test(r*s==24); return (r,s)
MkMatrix[[],[],[],[],[],[],[],[],[(4,6),(6,4)],[(3,8),(8,3)],
  [],[],[(2,12),(12,2)],[],[],[],[],[],[],[],[],[],[],
  [(1,24),(24,1)],[],[],[],...
```

Unabhängigkeit v.d. Suchstrat. 1(2)

Breitensuche (`MkMatrix[[n]|n<-[1..]]`), Tiefensuche ([1..]), Diagonalisierung...

Zusätzliche Funktionen, um die Strategie beim Aufruf ("im letzten Moment") festlegen zu können...

Steuerung über den Monadentyp...

```
choose        :: Bunch m => Stream a -> m a
choose (x:xs) =  wrap (return x 'alt' choose xs)

factor   :: Bunch m => Int -> m(Int, Int)
factor n =  do r <- choose[1..]; s <- choose[1..];
                         test(r*s==n); return (r,s)
```


Unabhängigkeit v.d. Suchstrat. 2(2)

Damit möglich...

- Benutzung von `factor` mit verschiedenen Suchstrategien
- Der spezifizierte Typ von `factor` bestimmt die Suchmonade (und damit die Suchstrategie)

```
?factor 24 :: Stream(Int,Int)
[(1,24)

?factor 24 :: Matrix(Int, Int)
Matrix[[],[],[],[],[],[],[],[],[],[],[(4,6),(6,4)],
  [(3,8),(8,3)],[],[],[(2,12),(12,2)],[],[],[],[],[],[],
  [],[],[],[],[(1,24),(24,1)],[],[],[],...
```

Zwischenfazit

Erinnerung...

Zentrale Probleme

- Modellierung logischer Programme mit...
  - mehreren Ergebnissen: **gelöst** (i.w. mittels lazy lists)
  - *logischen* Variablen: **noch offen**
    - * Üblich bei logischen Programmen: keine pure Vereinfachung eines anfangs vollständig gegebenen Ausdrucks, sondern variablenbehafteter Ausdrücke, für die geeignete Belegungen zu bestimmen sind. Variablen können im Zuge der Lösung durch andere variablenbehaftete Teilausdrücke ersetzt werden, für die dann ihrerseits geeignete Belegungen zu finden sind.
  - Modellierung der inhärenten Auswertungsstrategie logischer Programme: **gelöst**
    - * implizite Suche logischer Programmiersprachen explizit gemacht
    - * mittels Typklassen von Haskell sogar unterschiedliche Suchstrategien einfach realisierbar

Zum letzten Problem: Terme, Substitutionen und Prädikate 1(3)

Zur Modellierung in Haskell...

```
-- Terme beschreiben Werte logischer Variablen
data Term = Int Int | Nil | Cons Term Term | Var Variable
  deriving Eq

-- Benannte Variablen, um Anfragen zu formulieren.
-- Generierte Variablen: im Zuge der Berechnung
data Variable = Named String | Generated Int
  deriving (Show, Eq)
```

Hilfsfunktionen

```
-- Ueberfuehren einer Zeichenreihe in eine benannte Variable
var   :: String -> Term
var s =  Var (Named s)
-- Konstruktion der Termdarstellung einer Liste ganzer Zahlen
list    :: [Int] -> Term
list xs =  foldr Cons Nil (map Int xs)
```

Zum letzten Problem: Terme, Substitutionen und Prädikate 2(3)

Substitution und Unifikation

```
-- Substitution: i.w. eine Abbildung von Variablen auf Terme
-- Details dazu spaeter
newtype Subst

-- Weiters benoetigte Funktionen
apply :: Subst -> Term -> Term
idsubst :: Subst
unify :: (Term, Term) -> Subst -> Maybe Subst
```


Zum letzten Problem: Terme, Substitutionen und Prädikate 3(3)

Logische Programme (in unserer Haskell-Umgebung)

```
-- Logische Programme haben Typ Pred m
type Pred m = Answer -> m Answer

-- Antworten; die integer-Komponente steuert die Generierung
-- neuer Variablen
newtype Answer = MkAnswer (Subst, Int)

-- "Initiale Antwort"
initial :: Answer
initial =  MkAnswer (idsubst, 0)
run   :: Bunch m => Pred m -> m Answer
run p = p initial

-- "Programmlauf eines Praedikats als Anfrage", wobei
-- p auf die initiale Antwort angewandt wird
run p :: Stream Answer
```

Schreiben logischer Programme

Ein Beispiel...

`append(a,b,c)` mit `a,b` Listen und `c` Konkatenation von `a` und `b`

Implementierung als Funktion von Termen auf Prädikate...

```
append :: Bunch m => (Term, Term, Term) -> Pred m

-- Implementierung von append vorausgesetzt (kommt spaeter)
?run(append(list[1,2],list[3,4],var "z")) :: Stream Answer
[{z=[1,2,3,4]}]


-- beziehungsweise genauer und aequivalent zu obiger Liste
Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```


Kombinatoren für log. Programme 1(4)

Einfache Prädikate mithilfe des Operators `(=:=)` :

```
?run(var "x" =:= Int 3) :: Stream Answer
```

Implementierung

```
(=:=) :: Bunch m => Term -> Term -> Pred m
(t=:=u)(MkAnswer(s,n)) =
   case unify(tu) s of
     Just s' -> return(MkAnswer(s',n))
     Nothing -> zero
```


Kombinatoren für log. Programme 2(4)

Konjunktion von Prädikaten mithilfe des Operators `(&&&)`:

```
?run(var "x" =:= Int 3 &&& var "y" =:= Int 4) :: Stream Answer
[{x=3,y=4}]

?run(var "x" =:= Int 3 &&& var "x" =:= Int 4) :: Stream Answer
[]
```

Implementierung

```
(&&&) :: Bunch m => Pred m -> Pred m -> Pred m
(p &&& q) s = p s >>= q

-- gleichbedeutend und die Sequentialitaet unterstreichend waere
do t <- p s; u <- q t; return u
```


Kombinatoren für log. Programme 3(4)

Disjunktion von Prädikaten mithilfe des Operators `(|||)`:

```
?run(var "x" =:= Int 3 ||| var "x" =:= Int 4) :: Stream Answer
[{x=3,x=4}]
```

Implementierung

```
(|||) :: Bunch m => Pred m -> Pred m -> Pred m
(p ||| q) s = alt (p s) (q s)
```

Kombinatoren für log. Programme 4(4)

Einführung neuer Variablen in Prädikate (unter Ausnutzung der Integer-Komponente von Antworten)

...zur Konstruktion lokaler Variablen in rekursiven Prädikaten

```
exists :: Bunch m => (Term -> Pred m) -> Pred m
exists p (MkAnswer (s,n)) =
  p (Var(Generated n)) (MkAnswer(s,n+1))
```

Ebenfalls zur Behandlung rekursiver Prädikate

...stellt sicher, dass im Zshg. mit `Matrix` die Kosten pro Rekursionsauffaltung um 1 steigen

```
step     :: Bunch m => Pred m -> Pred m
step p s =  wrap (p s)
```


Beispiel

Anwendungsbeispiele für `wrap` und `step`

```
?run (var "x" =:= Int 0) :: Matrix Answer
MkMatrix[[{x=0}]]

?run(step(var "x" =:= Int 0)) :: Matrix Answer
MkMatrix[[],[{x=0}]]
```


Rekursive Programme 1(2)

Damit kann jetzt die Implementierung von `append` angegeben werden:

```
append(p,q,r) =
  step(p =:= Nil &&& q =:= r
       ||| exists (\x -> exists (\a -> exists (\b ->
             p =:= Cons x a &&& r =:= Cons x b
             &&& append(a,q,b)))))
```


Rekursive Programme 2(2)

Auch folgende (für logische Programme typische) Anwendung ist möglich:

Welche Listen ergeben konkateniert die Liste `[1,2,3]`?

```
?run(append(var "x", var "y", list[1,2,3])) :: Stream Answer
[{x = Nil, y = [1,2,3]},
  {x = [1], y = [2,3]},
  {x = [1,2], y = [3]},
  {x = [1,2,3], y = Nil}]
```


Ein komplexeres Beispiel 1(2)

Konstruktion "guter" Sequenzen von Nullen und Einsen

Festlegung

1. Die Sequenz [0] ist gut
2. Wenn die Sequenzen s1 und s2 gut sind, dann auch die Sequenz [1] ++ s1 ++ s2
3. Außer den nach 1. und 2. guten Sequenzen gibt es keine weiteren

Ein komplexeres Beispiel 2(2)

Implementierung als Prädikat

```
good(s) =
  step (s =:= Cons(Int 0) Nil
        ||| exist (\t -> exists (\q -> exists (\r ->
              s =:= Cons (Int 1) t &&& append(q,r,t)
              &&& good(q) &&& good(r)))))
```


Beispiele 1(4)

Test auf "gut":

```
?run (good (list[1,0,1,1,0,0,1,0,0])) :: Stream Answer
[{}]  -- leere Antwort bei guter Liste

?run (good (list[1,0,1,1,0,0,1,0,1])) :: Stream Answer
[]    -- keine Antwort bei nicht guter Liste
```


Beispiele 2(4)

Konstruktion guter Listen

```
-- Unfairer bunch-Typ: Antworten fehlen
?run(good(var "s")) :: Stream Answer
[{s=[0]},
  {s=[1,0,0]},
  {s=[1,0,1,0,0]},
  {s=[1,0,1,0,1,0,0]},
  {s=[1,0,1,0,1,0,1,0,0]},...
```

Beispiele 3(4)

```
-- Zum Vergleich: Fairer Bunch-Typ
?run(good(var "s")) :: Diag Answer
Diag[{s=[0]},
  {s=[1,0,0]},
  {s=[1,0,1,0,0]},
  {s=[1,0,1,0,1,0,0]},
  {s=[1,1,0,0,0]},
  {s=[1,0,1,0,1,0,1,0,0]},
  {s=[1,1,0,0,1,0,0]},
  {s=[1,0,1,1,0,0,0]},
  {s=[1,1,0,0,1,0,1,0,0]},...
```


Beispiele 4(4)

```
-- Zum Vergleich: Breitensuche-Bunch-Typ
-- Ergebnisausgabe "vorhersagbarer"
?run(good(var "s")) :: Matrix Answer
MkMatrix[[],
   [{s=[0]}],[],[],[],
   [{s=[1,0,0]}],[],[],[],
   [{s=[1,0,1,0,0]}],[],
   [{s=[1,1,0,0,0]}],[],
   [{s=[1,0,1,0,1,0,0]}],[],
   [{s=[1,0,1,1,0,0,0]}],{s=[1,1,0,0,1,0,0]}],[],
   ...
```


Zu guter Letzt: Nachträge 1(5)

Neue Infixoperatoren

```
infixr 4 =:=
infixr 3 &&&
infixr 2 |||
```

Substition

```
newtype Subst = MkSubst [(Var, Term)]
unSubst(MkSubst s) = s

idsubst = MkSubst[]
extend x t (MkSubst s) = MkSubst ((x,t):s)
```


Zu guter Letzt: Nachträge 2(5)

Substitutionsanwendung

```
apply :: Subst -> Term -> Term
apply s t =
       case deref s t of
         Cons x xs -> Cons (apply s x) (apply s xs)
         t'        -> t'

deref :: Subst -> Term -> Term
deref s (Var v) =
       case lookup v (unSubst s) of
         Just t    -> deref s t
         Nothing   -> Var v
deref s t = t
```


Zu guter Letzt: Nachträge 3(5)

Unifikation

```
unify :: (Term, Term) -> Subst -> Maybe Subst
unify (t,u) s =
  case (deref s t, deref s u) of
    (Nil, Nil) -> Just s
    (Cons x xs, Cons y ys)  -> unify (x,y) s >>= unify (xs, ys)
    (Int n, Int m) | (n==m) -> Just s
    (Var x, Var y) | (x==y) -> Just s
    (Var x, t)              -> if occurs x t s then Nothing
                                    else Just (extend x t s)
    (t, Var x)              -> if occurs x t s then Nothing
                                    else Just (extend x t s)
    (_,_)                   -> Nothing
```


Zu guter Letzt: Nachträge 4(5)

```
occurs :: Variable -> Term -> Subst -> Bool
occurs x t s =
  case deref s t of
    Var y     -> x == y
    Cons y ys -> occurs x y s || occurs x ys s
    _         -> False
```


Zu guter Letzt: Nachträge 5(5)

Was weiters fehlt...

- Funktionen für's Schöndrucken von Termen und Anworten

...der Kreis schließt sich (vgl. Vorlesungsteil 5)

Zum Ausklang

16 Jahre Haskell: Ein Rückblick zum 15. Geburtstag

von

Simon Peyton-Jones

Wearing the Hair Shirt: A Retrospective on Haskell

http://research.microsoft.com/users/simonpj/papers/haskell-retrospective/