Deklarative Programmierung Vorlesung Wintersemester 2009

Johannes Waldmann, HTWK Leipzig

25. Januar 2010

1 Einleitung

Definition

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

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

Softwaretechnische Vorteile

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

Gliederung der Vorlesung

- Grundlagen: algebraische Datentypen, Pattern Matching
- 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 ($\approx$ 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, Werkzeuge, 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/`

2 Daten

Algebraische Datentypen

```
data Foo = Foo { bar :: Int, baz :: String }
    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 gebunden):

```
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 Baum
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   {} -> ...
    Branch {} -> 0
```

Peano-Zahlen

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

- definiere Addition, Multiplikation, Potenz
- beweise die üblichen Eigenschaften

(mehrsortige) Signaturen

- Sorten-Symbole $S = \{S_1, \ldots, S_i\}$
- Funktionssymbole $F = \{f_1, \ldots, f_j\}$
- jeweils mit Sorten-Zuordnung $T : F \to S^+$
  schreibe $T(f) = [s_1, \ldots, s_k]$ als $f : s_1 \times \ldots \times s_{k-1} \to s_k$
  Stelligkeit (Arität) von f ist $|T(f)| - 1$.

Beispiel:

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

Algebren

Zu einer Signatur Σ passende Algebra:

- für jedes Sortensymbol S_k eine nichtleere Menge M_k
- für jedes Funktionssymbol eine Funktion von dem Typ, der durch Interpretation der Sorten bestimmt wird

Beispiel (Vektorraum)

- $K \mapsto \mathbb{R}, V \mapsto \mathbb{R}^3$
- p ist Addition in $\mathbb{R}$, m ist Multiplikation in $\mathbb{R}$,
  q ist Vektor-Addition in $\mathbb{R}^3$,
  n ist Multiplikation (Skalar mal Vektor)

Term-Algebren

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

- jeder Knoten t ist markiert durch ein $f \in F$
  bezeichne $T(f) = [s_1, \ldots, s_k, s]$,
  die Sorte von t ist sort$(t) = s =$ last$(T(f))$
- t hat Kinder $t_1 \ldots, t_k$,
  und $\forall i : 1 \leq i \leq k :$ sort$(t_i) = s_i$
- interpretiere Sortensymbol s_i durch $\{t \mid \text{sort}(t) = s_i\}$
  und Fktssymb. f durch Fkt. $(t_1, \ldots, t_k) \mapsto f(t_1, \ldots, t_k)$

Vergleich Termalgebra/data

Übereinstimmungen:

- Sortensymbol = Typ
- Funktionssymbol = Konstruktor

Erweiterungen:

- benannte Notation von Konstruktor-Argumenten
- Polymorphie

Polymorphie

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

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

Listen

eigentlich:

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

aber aus historischen Gründen

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

Pattern matching dafür:

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

Summe der Elemente einer Liste?

Operationen auf Listen

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

3 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

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 Listen

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

Rekursionsmuster (Prinzip)

jeden Konstruktor durch eine passende Funktion ersetzen.

```
data N = Z | S N
fold ( z :: b ) ( s :: b -> b ) :: N -> b

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

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

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

Rekursionsmuster (Peano-Zahlen)

```
data N = Z | S N

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

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

Übungen Rekursionmuster

Listen: Muster anwenden (append)

Bäume: Muster definieren und anwenden

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

4 Bedarfs-Auswertung

Motivation: Datenströme

Folge von Daten:

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

aus softwaretechnischen Gründen diese drei Aspekte im Programmtext trennen,
aus Effizienzgründen in der Ausführung verschränken (bedarfsgesteuerter Transformation/Erzeugung)

Bedarfs-Auswertung, Beispiele

- Unix: Prozesskopplung durch Pipes

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

- OO: Iterator-Muster

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

- 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 in 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
  ```

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 fibonacci )

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

Übungen:

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

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!

5 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 ; ... }
```

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 mzero
```

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, return`
- bei Permutationen benutze:

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

Die IO-Monade

```
data IO -- abstract

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

instance Functor IO ; instance Monad IO
```

Alle „Funktionen“, deren Resultat von der Außenwelt (Systemzustand) abhängt, haben Resultattyp `IO ...`

Am Typ einer Funktion erkennt man ihre möglichen (schädlichen) Wirkungen bzw. deren garantierte Abwesenheit.

Wegen der Monad-Instanz: benutze do-Notation

```
do cs <- readFile "foo.bar" ; putStrLn cs
```

Die Zustands-Monade

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

```
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 = ??
instance Functor ( State s ) where
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

6 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 = ( deleteFile "foo" w, putStr "bar" w )
```

Lösungen:

- Haskell: Typ `World` ist *privat*, öffentlich ist nur `IO`
- Clean: Typ `World` ist öffentlich, aber *unique*

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 "." ) sub
        setCurrentDirectory ".."
```

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

Lazy IO

(das ist ein sehr dunkles Kapitel)

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

- `hGetContents` liefert einen lazy String,
- erst bei Bedarf wird der Handle gelesen.
- ... falls er dann noch offen ist
- benutze `seq`, um Bedarf herzustellen

Variablen (IORefs)

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

strenge Unterscheidung zwischen

- Verweis (`x :: IORef Integer`)
- Wert (`a :: Integer`)

Lesen und Schreiben sind IO-Aktionen, weil sie den Hauptspeicherinhalt ändern.

Variablen (STRefs)

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

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

ausführen mit beschränkten Nebenwirkungen

```
runST :: ( forall s . ST s a ) -> a
```

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

Variablen — Ausblick

- IORefs sind nicht thread-sicher

  (die üblichen Probleme mit globalen Variablen)
- benutze Control.Concurrent.{MVar,Channel}
- STM (software transactional memory) für spekulative Ausführung (atomic transactions)

7 Projekte

autotool: Erweiterungen, Reparaturen

- autotool bauen (ghc/cabal, git)
- erweitern:
  - 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`
- reparieren `http://dfa.imn.htwk-leipzig.de/bugzilla/buglist.cgi?component=autotool&bug_status=NEW&bug_status=ASSIGNED&bug_status=REOPENED`

autotool-Server/Happstack

die *Challenger*-Idee (ca. 2001) vgl. `http://www.imn.htwk-leipzig.de/~waldmann/autotool/doc/challenger/node142.html`

Einsender stellen sich gegenseitig Aufgaben(-Instanzen) zu vorgegebenem Typ:

- Menge von publizierten Instanzen
- Aktion: Lösung zu einer Instanz einsenden
- Aktion: neue Instanz einsenden (mit Lösung, die geheim bleibt)

Bewertungskriterien:

- kleine Instanz, lange ungelöst, ...

Implementierung: benutze autotool-Semantik und

- `http://happstack.com/`

Ableitungsmuster

- gegeben: Ersetzungssystem,
- gesucht: untere Schranken für Ableitungskomplexität,
  d. h. Ableitungsmuster mit Parametern und induktiven Beweisen

Beispiel $R = \{ab \to ba\}$
linear: $\forall k \geq 0 : a^k b \to^k ba^k$, quadratisch: $\forall k, l \geq 0 : a^k b^l \to^{k \cdot l} b^l a^k$,
Beispiel $S = \{ab \to baa\}$
linear: $\forall k \geq 0 : a^k b \to^* ba^{2k}$
exponentiell: $\forall l \geq 0 : ab^l \to^* b^l a^{2^l}$

8 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) zs
main :: IO ()
main = quickCheck assoc
```

Quickcheck, Smallcheck, . . .

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

- 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* erzeugt:

  - 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 Spezifizierens

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

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

Beispiel: Test von Sortierverfahren

sinngemäß nach Kap. 11 aus Real World Haskell: `http://book.realworldhaskell.org/read/testing-and-quality-assurance.html`

- zum Formulieren der Spezifikationen:

```
ist_monoton :: Ord a => [a] -> Bool
ist_permutation :: Eq a => [a]->[a]-> Bool
erzeuge_monoton :: [Int] -> [Int]
```

- Spezifikationen von: `merge, msort`
- Implementierung von: `merge, msort`
- Testtreiber
- Testabdeckung

Code Coverage

Haskell Program Coverage `http://projects.unsafeperformio.com/hpc/`

```
ghc -fhpc --make Sort.hs
./Sort
hpc report Sort
hpc markup Sort
```

Beispiel: Partitionen

Bijektion: ungerade ↔ steigende

Hinweis:

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

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

Übungsaufgabe: `odd_to_strict`

9 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` enthält:

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

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

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

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

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

vgl. Java:

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

Instanzen

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

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

weiter Instanzen kann man selbst deklarieren:

```
data Student = Student { vorname  :: String
                       , nachname :: String
                       , 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 Konstruktoren und konkreten Methoden)
   `data T = ...`
3. Instanz-Deklaration (= Implementierung der abstrakten Methoden) `instance C T where { m =`

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

Das ist an einigen Stellen nachteilig und erfordert Bastelei: wenn `class T implements Comparabl` aber man die T-Objekte anders vergleichen will?

Man kann deswegen oft die gewünschte Vergleichsfunktion separat an Sortier-Prozeduren übergeben.

... natürlich nicht die Funktion selbst, Java ist ja nicht funktional, sondern ihre Verpackung als Methode eines Objekts einer Klasse, die

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

implementiert.

Wörterbücher

Haskell-Typklassen/Constraints...

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

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

... sind Abkürzungen für Wörterbücher:

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

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

Für jedes Constraint setzt der Compiler ein Wörterbuch ein.

Wörterbücher (II)

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

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

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

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

Vergleich Polymorphie

- Haskell-Typklassen:

  statische Polymorphie,

  Wörterbuch ist zusätzliches Argument der Funktion

- OO-Programmierung:

  dynamische Polymorphie,

  Wörterbuch ist im Argument-Objekt enthalten.

  (OO-Wörterbuch = Methodentabelle der Klasse)

Klassen-Hierarchien

Typklassen können in Beziehung stehen.

Ord ist tatsächlich „abgeleitet“ von Eq:

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

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

Ord ist Typklasse mit Typconstraint (Eq)

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

Jedes Ord-Wörterbuch hat ein Eq-Wörterbuch.

Die Klasse Show

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

vgl. Java: toString()

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

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

... oder schreiben lassen: `deriving Show`

Kanonische Show/Read-Instanzen

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

- eine Show-Methode (Instanz) heißt *kanonisch*, wenn `show x` gültiger Haskell-Quelltext ist, dessen Auswertung wieder `x` ergibt.
- `deriving Show` liefert kanonische Instanzen.

`class Read a where read :: String -> a -- vereinfacht`

- Read-Instanz heißt kanonisch, wenn `read (show x) == x`
- `deriving Read` liefert kanonische Instanzen

Die Wahrheit über Read

Standard-Haskell:

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

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

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

Siehe `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Text-ParserCombinators-ReadPrec.html`

Automatisches Ableiten von Instanzen (I)

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

Beachte: generische Instanz mit Typconstraint

Das kann der Compiler selbst:

```
data Tree a = ... deriving Show
```

Default-Implementierungen

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

```
class  (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
```

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

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]"
```

Überlappende Instanzen

Wegen `String = [Char]` gilt bisher

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

Erwünscht ist aber:

```
instance Show String where
    show cs = "\"" ++ cs ++ "\""
show "foo" = "\"foo\""
```

Diese Instanz-Deklaration *überlappt* mit generischer.

Für `Show [Char]` gibt es dann zwei Wörterbücher— das ist grundsätzlich verboten:
in Standard Haskell dürfen generische Instanzen keine Typvariablen instanziieren.

Überlappende Instanzen (II)

mit `ghc(i) -XTypeSynonymInstances` (Kommandozeile) oder `{# language TypeSynonym`
(Quelltext, 1. Zeile) sind instantiierte Typvariablen in Instanzen erlaubt.

mit `-XOverlappingInstances` gewinnt bei Überlappung die speziellere Instanz.

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

Typklassen als Prädikate

Man unterscheide *gründlich* zwischen Typen und Typklassen (OO: zwischen Klassen und Schnittstellen).

Eine Typklasse C ist ein (einstelliges) *Prädikat* auf Typen T:

Die Aussagen $C(T_1), C(T_2), \ldots$ sind wahr oder falsch.

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

Multi-Parameter-Klassen

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

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

Oft benötigt man mehrstellige Prädikate (Relationen)

```
class Brett b => Zug b z where ...
instance Zug Havannah Satz where ...
```

diese werden von *Tupeln* von Typen erfüllt (oder nicht).

(geht das in „klassischen“ OO-Sprachen? - Nein.)

Man kann zusichern, daß die Relation eine Funktion ist (*functional dependency*):

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

zu jedem Typ-Paar (p, i) gibt es höchstens ein b mit `Problem p i b`-Wörterbuch.

Benutzung von Typklassen bei Smallcheck

Colin Runciman, Matthew Naylor, Fredrik Lindblad:
SmallCheck and Lazy SmallCheck: automatic exhaustive 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 Tiefe d
    series :: Int -> [a]
```

jedes Objekt hat endliche Tiefe, zu jeder Tiefe nur endliche viele Objekte
Die „Tiefe“ von Objekten:

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

Kombinatoren für Folgen

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

(\/) :: Series a -> Series a -> Series a
s1 \/ s2 = \ d -> s1 d ++ s2 d
(><) :: Series a -> Series b -> Series (a,b)
```

```
s1 >< s2 = \ d ->
    do x1 <- s1 d; x2 <- s2 d; return (x1, x2)

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 ) where
    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 t
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 d

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

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

Kategorien (Definition II)
... und erfüllt Bedingungen:

- $\mathrm{id}.$ sind neutral (auf beiden Seiten)
  für alle $a \xrightarrow{m} b$:
  $\mathrm{id}_a \circ m = m = m \circ id_b$
- 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 Mengen 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* von A 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: $F = (F_{\mathrm{Obj}}, F_{\mathrm{Mor}})$ mit

- Wirkung auf Objekte: $F_{\mathrm{Obj}} : \mathrm{Obj}(C) \to \mathrm{Obj}(D)$
- Wirkung auf Morphismen: $F_{\mathrm{Mor}} : \mathrm{Mor}(C) \to \mathrm{Mor}(D)$ mit $g : A \to B \Rightarrow F_{\mathrm{Mor}}(g) : F_{\mathrm{Obj}}(A) \to F_{\mathrm{Obj}}(B)$
- 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 von 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 -> m c)
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 Kategorie

```
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 xs
  ```

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

11 Arrows

Einleitung

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

- Ausgaben beliebig (M anwenden)
- Eingabe ist immer Funktionsargument

Damit geht also folgendes *nicht*:

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

Lösung:

```
instance Arrow SF where ...
```

```
  http://www.haskell.org/arrows/
```

Stream Functions

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

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

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

Kombinatoren für Stream Functions

sequentiell:

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

parallel:

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

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

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

Die Arrow-Klassen

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

proc-Notation für Arrows

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

wird übersetzt in entsprechende punktfreie Notation

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

. . . und ggf. weiter vereinfacht
`http://www.haskell.org/arrows/syntax.html`

Anwendung von Arrows in HXT
`http://www.fh-wedel.de/~si/HXmlToolbox/` `http://www.haskell.org/haskellwiki/HXT/` Eingabe:

```
<problem type="termination">
<trs> <rules> <rule> <lhs>
   <funapp> <name>a</name> <arg> ...
```

Programm:

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

XML-Serialisierung
. . . für algebraische Datentypen

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

Wie sollte das als XML aussehen? — So:

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

XML-Serialisierung (II)

Prinzipien sollten sein:

- vollständige Information:

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

- elementare Werte in Attributen

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

Probleme bei Abweichen von diesem Prinzipien (z. B. Weglassen von „offensichtlichen“ Tags)

12 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

Syntax

- *Symbol*: Variable beginnt mit Großbuchstaben, sonst Funktions- oder Prädikatsymbol.
- *Regel* besteht aus

  Kopf (Konlusion) :: 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 Fakten) 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 Rumpfes nicht in C vorkommen.

Bedeutung eines Programms P mit Regeln $R_1, \ldots, R_i$ und 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 Ableitungen (Resolution) bestimmt.

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

Dann gibt es (wenigstens) eine Belegung der Variablen der 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 probieren

Substitutionen (Definition)

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

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

  so daß
  - für alle $v \in \mathrm{dom}\,\sigma : v\sigma \neq v$
  - kein $v \in \mathrm{dom}\,\sigma$ kommt in $\mathrm{img}\,\sigma$ als Teilterm vor
- Substitution σ auf Term t anwenden: $t\sigma$

Substitutionen (Produkt, Ordnung)

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!):
$\sigma_1 = \{X \mapsto Y\}, \sigma_2 = \{Y \mapsto X\}, \sigma_1 \circ \sigma_2 = \sigma_2$

Substitution σ_1 ist *allgemeiner als* Substitution σ_2:
$\sigma_1 \underset{\sim}{<} \sigma_2 \iff \exists \tau : \sigma_1 \circ \tau = \sigma_2$
Beispiele: $\{X \mapsto Y\} \underset{\sim}{<} \{X \mapsto a, Y \mapsto a\}$,
$\{X \mapsto Y\} \underset{\sim}{<} \{Y \mapsto X\}$ und $\{Y \mapsto X\} \underset{\sim}{<} \{X \mapsto Y\}$.
Relation $\underset{\sim}{<}$ ist Prä-Ordnung (reflexiv, transitiv, aber nicht antisymmetrisch)

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. $\underset{\sim}{<}$)
Satz: jedes Unifikationsproblem ist

- entweder gar nicht
- oder bis auf Umbenennung eindeutig

lösbar.

(σ ist Umbenennung: $\mathrm{img}\,\sigma \subseteq$ Variablen)

Unifikation—Algorithmus

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

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

Bemerkungen:

- Modellierung in Haskell: Data.Map, Maybe
- korrekt, übersichtlich, aber nicht effizient,
- es gibt Unif.-Probl. mit exponentiell großer Lösung,
- eine komprimierte Darstellung davon kann man aber in 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, backtracking, and pattern matching in lazy functional languages.* 1985. `http://homepages.inf.ed.ac.uk/wadler/`

Ein einfacher Prolog-Interpreter

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

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 Korrektheit

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 Modi deklariert
- für jede Benutzung eines Prädikates wird (vom Compiler) 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.

13 Constraint-Programmierung

Formeln, Modelle

- Aussagenlogik
  - Formel: (in konjunktiver Normalform)
  - Modell: Belegung Variable → Boolean
- Prolog
  - Formel: (Regelmenge, Programm)
  - Modell: Belegung Variable → Term

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

(CNF-)SAT

Das Problem (CNF)-SAT:

- Eingabe: aussagenlog. Formel (in CNF)
- Frage: gibt es erfüllende Belegung?

Eigenschaften

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

vergleiche `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/stdout

import Control.Monad ( forM)
import System.Environment

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

pos n i j = n * i + j + 1
neg n i j = negate $ pos n i j
```

```
in_jeder_zeile_höchstens_eine :: Int -> CNF
in_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 ]

main = do
   [ arg ] <- System.Environment.getArgs
   let n  = read arg :: Int
       cls = -- in_jeder_zeile_wenigstens_eine n
             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)

- lineare Gleichungen

  Gauss-Elimination

- lineare Ungleichungen

  Simplex-Verfahren, Ellipsoid-Methode

- Polynom(un)gleichungen

  - über reellen Zahlen: Satz von Tarski, zylindrische algebraische Zerlegung (QEP-CAD)
  - über ganzen Zahlen: 10. Hilbertsches Problem, Satz von Matiasevich/Roberts

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

Bitblasting

Lösen von Constraint-System über ganzen Zahlen:

- Festlegen einer maximalen Bitbreite
- unbekannte Zahl ⇒ Folge von unbekannten Bits
- arithmetische Operationen ⇒ Bit-Operationen (entspr. Schaltkreis-Entwurf)
- Lösung durch SAT-Solver

Beispiel: `http://dfa.imn.htwk-leipzig.de/satchmo/`

Zusammenfassung Constraint-Programmieren

- Viele Aufgaben *sind* Constraint-Systeme (die Spezifikation *ist* eine prädikatenlogische Formel)
- herkömmlicher Ansatz: man erfindet und implementiert einen *anwendungsspezifischen* Algorithmus, der das Constraint-System löst
- Constraint-Programmierung: man schreibt das Constraint-System in einer formalen Sprache und benutzt einen *anwendungsunabhängigen* Löser.
- Solche Constraint-Systeme sind deklarative Programme.

(Master-Vorlesung Constraint-Programmierung)

14 Nebenläufige und parallele Programme

Definition, Motivation

- nebenläufig (concurrent):

  Nichtdeterminismus, Kommunikation, Synchronisation

  auch auf single-Core sinnvoll (Multi-Agenten-Systeme)
- parallel (data parallelism):

  Auswertungsstrategien, die Multiprozessor/core-Hardware ausnutzten

  Idealfall: Kommunikation/Synchronisation im Programmtext unsichtbar

Threads

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

- kompilieren: `ghc --make Foo -threaded -O2`
- ausführen: `./Foo +RTS -N2` (benutzt 2 Cores)

forkIO erzeugt Haskell-Thread, das RTS verteilt diese auf Ausführungskontexte (OS/-Cores)

main thread fertig $\Rightarrow$ Programm fertig (geforkte Threads werden abgebrochen)

Kommunikation: MVar

MVar = mutable variable

- erzeugen:

  `newEmptyMVar :: IO (MVar a)`

- schreiben:

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

  blockiert, solange Variable belegt ist

- lesen:

  `takeMVar :: MVar a -> IO a`

  blockiert, solange Variable leer ist

Beispiel MVar

Hauptprogramm wartet, bis ein Thread fertig ist:

```
main = 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

Channel = Ringpuffer, Queue; ohne Kapazitätsschranke

- erzeugen:

  `newChan :: IO (Chan a)`

- schreiben:

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

  (blockiert nicht)

- lesen:

  `readChan :: Chan a -> IO a`

  blockiert, solange Puffer leer ist

Beispiel Chan

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

```
ch :: Chan Integer <- newChan

-- mehrere Threads starten:
forM [ 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:
ss <- forM [1 .. threads] $ \ t -> readChan ch
print $ sum ss
```

Programmieren mit Locks

. . . ist fehleranfällig:

- zuwenig Locking: riskiert Inkonsistenzen der Daten

Beispiel (Vorsicht, fehlerhafter Code):

```
überweise ( 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 ->
            return $ z + betrag
```

Programmieren mit Locks

. . . ist fehleranfällig:

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

Beispiel (Vorsicht, fehlerhafter Code):

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

Transaktionen anstelle von Locks

Transaktion ist eine Folge von Anweisungen:

- als Ganzes (atomar) erfolgreich (commit) oder nicht
- bekannt von Implementierung von Datenbanken
- hier angewendet auf Speicherzugriffe (software transactional memory)
- *spekulative* Ausführung: vor commit prüfen, ob eine Variable gelesen wurde, die inzwischen von anderer Transaktion geändert wurde (ja → kein commit, retry)

- Transaktionen dürfen keine sichtbaren Nebenwirkungen haben (Typ ist nicht `IO a`, sondern `STM a`)
- *retry*: Transaktion abbbrechen, später neu starten (wenn eine der Konfliktvariablen geschrieben wurde)

STM-Beispiel

```
überweise ( 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
```

benutzt Funktionen:

```
readTVar   :: TVar a -> STM a
writeTVar  :: TVar a -> a -> STM a
atomically :: STM a -> IO a
```

STM-Literatur

- Josef Svenningsson: `http://computationalthoughts.blogspot.com/2008/03/some-examples-of-software-transactional.html`
- `http://www.haskell.org/haskellwiki/Software_transactional_memory`

Parallelität durch Annotationen
(Bibliothek Concurrent.Parallel)

- `x `par` y`:
  - Berechnung von (whnf von) x starten
  - Resultat (sofort) ist y
- `x `pseq` y`:

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

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

Beispiel: Quicksort mit par/pseq

(vgl. Abschnitt in Real World Haskell)

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

rekursive Aufrufe sollten gleichzeitig ausgewertet werden:

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

Par/Pseq-Beispiel

```
{-# language PatternSignatures #-}

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

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

main = 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 links)
sort1 [] = []
sort1 (x:xs) =
  let (lo,hi) = Data.List.partition (<x) xs
      slo = sort1 lo ; shi = sort1 hi
  in  slo ++ x : shi

-- parallele Auswertung
sort2 [] = []
sort2 (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
sort3 d xs | null xs || d <= 0 = sort1 xs
sort3 d (x: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
sort4 d xs | null xs || d <= 0 = sort1 xs
sort4 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 )
```

```
force xs = xs `using` rdeepseq
```

Par/PSeq-Zusammendfassung

- Vorteil:

  Programm wird nur annotiert, Form bleibt erhalten,

  keine expliziten Threads, MVars etc.

- zu 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
`http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`

- data parallelism: verwende Datenstrukturen mit implizit parallelen Konstruktoren

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

- flat DP: Vektoren, Arrays
- nested DP: geschachtelte/rekursive Typen (Bäume)
  Implementierung durch *Vektorisierung* (= Transformation zu flat DP und zurück)

Weiterentwicklung (...CUDA-Backend) `http://hackage.haskell.org/package/accelerate`