Entwurfsmuster vom Kopf auf die Füße gestellt (Haskell für Umsteiger)

Johannes Waldmann, HTWK Leipzig

Einleitung, Überblick

Konzept	Realisierung FP	Realisierung (
freie Algebra	algebraischer Datentyp, pattern matching	Kompositum
Algebra	Rekursionschema (fold)	Besucher
Strom	laziness	Iterator
Funktion	Funktion	Befehl/Strateg
referentiell transparent	pure	immutable

Haskell benutzen

- `http://leksah.org/`
- `http://eclipsefp.sourceforge.net/`
- `http://www.haskell.org/visualhaskell/`

Haskell benutzen

- `http://leksah.org/`
- `http://eclipsefp.sourceforge.net/`
- `http://www.haskell.org/visualhaskell/`

Real Programmers (`http://xkcd.com/378/`)

- Quelltext im Editor `emacs Foo.hs`
- Interpreter `ghci Foo.hs`
- Interpreter: Ausdrücke auswerten
- Editor: speichern, Interpreter: neu laden (`:r`)

Algebraische Datentypen (1)

```
data Foo = Foo { bar :: Int, baz :: String
    deriving Show
x :: Foo
x = Foo { bar = 3, baz = "hal" }
```


Pattern Matching

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

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

Bespiel:

```
data Bool = False | True
```


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
```


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?

Theorie

- jeder Haskell-`data`-Typ ist: (mehrsortige) Signatur = Menge von Funktionssymbolen mit Sorten (Typen)
- die Objekte des Typs sind die korrekt zusammengesetzten Terme, die aus diesen Symbolen bestehen

Rekursions-Schemata (1)

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

Codeverdopplung $\longrightarrow$ Refaktoring!

Rekursions-Schemata (2)

das ist der Plan…

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

f = inorder
g = [] ; h x y z = x ++ [y] ++ z
```


Rekursions-Schemata (3)

…und den kann man tatsächlich hinschreiben!

```
fold :: ... -> ... -> Tree a  -> b
fold g h t = case t of
    Leaf {} ->  g
    Branch {} ->
        h (fold g h $ left t)
          (key t) (fold g h $ right t))
inorder = fold [] ( \ x y z -> x ++ [y] ++
sum     = fold ... ...
```

Anzahl der Blätter? Verzweigungen? Tiefe?

Rekursions-Schemata (4)

…für Listen?

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

foldr g z l = case l of
    x : xs -> ...
    []     -> ...

length  :: [a] -> Int ; length = foldr ...
reverse :: [a] -> [a]
map     :: (a -> b) -> [a] -> [b]
filter  :: (a -> Bool) -> [a] -> [a]
```


Rekursions-Schemata

Datentyp $\rightarrow$ Schema: Konstruktor $\rightarrow$ Funktion

ein Rekursionsschema für ein Objekt eines algebraischen Datentyps auswerten:
jeden Konstruktor (= Funktionssymbol) durch die entsprechende Funktion ersetzen

Datentyp = Signatur Σ,
Rekursionsschema = Σ-Algebra.

Streams

für bessere Programmstruktur: Trennung von

- Produzent
- Transformator(en)
- Konsument

```
sum $ map ( \ x -> x ^3 ) $ [ 1 .. 100 ]
```

aus Effizienzgründen *verschränkte* Auswertung

- Pipes zwischen Prozessen,
- Enumeratoren in OO.

Laziness

in Haskell: Listen sind Streams, der Tail wird erst bei Bedarf ausgewertet.

jeder Wert wird erst bei Bedarf ausgewertet.

wann genau? Kann man nicht beobachten. Die Auswertung hat keine Nebenwirkungen.

eine unendliche Datenstruktur:

```
naturals :: [ Integer ]
naturals = from 0 where
    from x = x : from (x+1)
```

Liste aller Quadratzahlen? Primzahlen?

Laziness: Anwendungen

Bindungsreihenfolge in let-Blöcken ist egal:

```
let  xs = 'a' : ys
     ys = 'b' : zs
     zs = 'c' : xs
in   xs
```

If-then-else könnte als Funktion geschrieben werden (kein Makro nötig)

```
ite b y n =
    case b of { True -> y ; False -> n }
```


Aufgabe

schreibe eine Funktion, die jeden Schlüssel eines Baumes durch die Summe aller Schlüssel ersetzt

```
sum = fold 0 ( \ x y z -> x + y + z )
replace r =
      fold Leaf ( \ x y z -> Branch x r z )

sumrep :: Tree Int -> Tree Int
sumrep t = replace ( sum t ) t
```

und den Baum nur einmal durchquert.

Hinweis

mit einer so spezifizierten Hilfsfunktion

```
help :: Int -> ( Tree Int )
     -> ( Tree Int, Int )
help r t = ( replace r t, sum t )
```

kann man die Aufgabe lösen:

```
sumrep t =
    let ( u, s ) = help s t  in  u
```

. . . wenn man `help` als ein `fold` schreibt!