

Rekursionsschemata

Zygohistomorphic prepromorphisms

Christian Höner zu Siederdisen
`christian.hoener.zu.siederdisen@uni-jena.de`

Theoretische Bioinformatik, Bioinformatik Uni Jena

Dec 15th, 2022

Rush-hour

<https://www.thinkfun.de/products/rush-hour/>



Rushhour ist *PSPACE-complete* und damit wahrscheinlich nur in exponentieller Zeit lösbar

Strukturelle Rekursion

Wir beginnen mit `foldr`

```

1  foldr :: (a -> b -> b) -> b -> [a] -> b
2  foldr _ b [] = b
3  foldr f b (x:xs) = f x (foldr f b xs)
4
5
6
7  foldr (-) 0 [1..3] =
8  foldr (-) 0 (1:[2,3]) = (-) 1 (foldr (-) 0 [2,3])
9                          = (-1) 1 (-1) = 2
10 foldr (-) 0 (2:[3]) = (-) 2 (foldr (-) 0 [3])
11                      = (-) 2 3 = -1
12 foldr (-) 0 [3] = (-) 3 (foldr (-) 0 [])
13                  = (-) 3 0 = 3
14 foldr (-) 0 [] = 0

```

Strukturelle Rekursion

Wir beginnen mit `foldr`

```

1  foldr :: (a -> b -> b) -> b -> [a] -> b
2  foldr _ b [] = b
3  foldr f b (x:xs) = f x (foldr f b xs)
4
5
6
7  foldr (:) [] [1..3] =
8  foldr (:) 0 (1:[2,3]) = (:) 1 (foldr (:) [] [2,3])
9                          = (:) 1 [2,3] = [1,2,3]
10 foldr (:) 0 (2:[3]) = (:) 2 (foldr (:) [] [3])
11                      = (:) 2 [3] = [2,3]
12 foldr (:) [] [3] = (:) 3 (foldr (:) [] [])
13                  = (:) 3 [] = [3]
14 foldr (:) [] [] = []

```

Strukturelle Rekursion

foldl

```

1 foldl :: (b -> a -> b) -> b -> [a] -> b
2 foldl _ b [] = b
3 foldl f b (x:xs) = foldl f (f b x) xs
4
5
6 foldl (-) 0 [1,2,3] = foldl (-) ((-) 0 1) [2,3]
7 foldl (-) (-1) [2,3] = foldl (-) ((-) (-1) 2) [3]
8 foldl (-) (-3) [3] = foldl (-) ((-) (-3) 3) []
9 foldl (-) (-6) [] = -6
10
11 -- oder?
```

Strukturelle Rekursion

foldl

```

1 foldl :: (b -> a -> b) -> b -> [a] -> b
2 foldl _ b [] = b
3 foldl f b (x:xs) = foldl f (f b x) xs
4
5
6 foldl (-) 0 [1,2,3] = foldl (-) ((-) 0 1) [2,3]
7 foldl (-) ((-) 0 1) [2,3] = foldl (-) ((-) ((-) 0 1) 2) [3]
8 foldl (-) ((-) ((-) 0 1) 2) [3]
9   = foldl (-) ((-) ((-) ((-) 0 1) 2) 3) []
10 foldl (-) (((-) ((-) ((-) 0 1) 2) 3)) [] = -6

```

Strukturelle Rekursion

foldl'

```

1 foldl' :: (b -> a -> b) -> b -> [a] -> b
2 foldl' _ b [] = b
3 foldl' f b (x:xs) = let z = f b x
4   in z 'seq' foldl' f z xs
5
6
7
8 foldl' (-) 0 [1,2,3] = foldl (-) ((-) 0 1) [2,3]
9 foldl' (-) (-1) [2,3] = foldl (-) ((-) (-1) 2) [3]
10 foldl' (-) (-3) [3] = foldl (-) ((-) (-3) 3) []
11 foldl' (-) (-6) [] = -6

```

Merke: foldl ist eigentlich nie korrekt, immer foldl' nutzen.

Strukturelle Rekursion

foldl

```

1 foldl :: (b -> a -> b) -> b -> [a] -> b
2 foldl _ b [] = b
3 foldl f b (x:xs) = foldl f (f b x) xs
4
5 snoc xs x = x:xs
6
7
8 foldl snoc [] [1,2] = foldl snoc (snoc [] 1) [2]
9 foldl snoc (snoc [] 1) [2]
10 = foldl snoc (snoc (snoc [] 1) 2) []
11 foldl snoc (snoc (snoc [] 1) 2) []
12 = (snoc (snoc [] 1) 2)
13 = snoc (1:[]) 2
14 = 2:(1:[]) = [2,1]

```

(Hier ist es egal ob wir foldl oder foldl' nutzen)

Das Inverse eines fold?

```
1  unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
2  unfoldr f b = case f b of
3    Nothing -> []
4    Just (a,b') -> a : unfoldr f b'
5
6
7  go x = if x<0 then Nothing else Just (x,x-1)
8
9
10 unfoldr go 2 = case go 2 of
11   Just (2,1) -> 2 : unfoldr go 1
12   = 2 : 1 : []
13 unfoldr go 1 = case go 1 of
14   Just (1,0) -> 1 : unfoldr go 0
15   = 1 : []
16 unfoldr go 0 = case go 0 of
17   Nothing = []
```

Deforestation

```
1 foldl' (+) 0 (unfoldr go 10) =
2 foldl' (+) 0 (10 : unfoldr go 9) =
3 foldl' (+) (0+10 = 10) (unfoldr go 9) =
```

- Ersetze 10 durch 10^{100}
- `unfoldr` wird nur soweit aufgerollt wie nötig ist um das nächste Element zu nehmen
- Haskell kann die "temporären" Datenstrukturen sogar "wegrechnen" (2023)

Rekursions-Schemata

- Ersetze rekursive Funktionen auf rekursiven Datentypen
- durch nicht-rekursive Funktionen auf verwandten nicht-rekursiven Datentypen
- Diese Strukturen nennen wir Basis-Funktoren (base functor)

```

1 type family Base t :: * -> *
2
3 type instance Base Natural = Maybe -- ???
4
5 data ListF a b = NilF | ConsF a b
6
7 instance Functor (ListF a) where
8     fmap f NilF = NilF
9     fmap f (ConsF a b) = Cons a (f b)
10
11 type instance Base [a] = ListF a

```

Base ist eine Funktion von Typ- nicht Datenargumenten!
 (das werden wir jetzt an der Tafel mal genauer anschauen)

Und für Bäume

```
1 data Tree a = Node a [Tree a]
2
3 type ForestF a b = [b]
4 data TreeF a b = NodeF a (ForestF a b)
5
6 instance Functor (TreeF a) where
7     fmap f (NodeF x xs) = NodeF x (fmap f xs)
8
9 type instance Base (Tree a) = TreeF a
```

Recursive

```
1 class Functor (Base t) => Recursive t where
2   project :: t -> Base t t
3
4 instance Recursive Natural where
5   project 0 = Nothing
6   project n = Just (n-1)
7
8 instance Recursive [a] where
9   project [] = NilF
10  project (x:xs) = ConsF x xs
11
12 instance Recursive (Tree a) where
13  project (Node x xs) = NodeF x xs
```

Hinweis: durch (momentane) Magie kann Recursive automatisch generiert werden.

fold / catamorphism

Layer um Layer wird die Struktur zu einem Wert zusammengefaltet

```
1 cata :: Recursive t => (Base t a -> a) -> t -> a
2 cata f = c where c = f . fmap c . project
3
4 t = cata (\x -> case x of { NilF -> 0; ConsF a b -> a+b})
5           [1..4]
```

Paramorphismus

Layer um Layer wird die Struktur zu einem Wert zusammengefaltet;
in jedem Layer steht der noch zu verarbeitende Teil der Struktur
zur Verfügung

```

1  para :: Recursive t => (Base t (t,a) -> a) -> t -> a
2  para t = p where p x = t . fmap ((,) <*> p) $ project x
3
4  testpara1 = para (\x -> case x of
5    { NilF -> 0; ConsF a (bs,b) -> a+b}) [1..4]
6
7  10
8
9  testpara2 = para (\x -> case x of
10   { NilF -> [[]]; ConsF a (rs,as) -> [rs,[a]] : as}) [1..4]
11
12 [ [[2,3,4] , [1]]
13   , [[3,4]   , [2]]
14   , [[4]     , [3]]
15   , [[]      , [4]]
16   , [ ]

```

Histomorphismen

- Variante eines catamorphismus
- innerhalb jedes Layers stehen alle cata-Ergebnisse aller Layer zur Verfügung

```
1 histo :: Recursive t => (Base t (Cofree (Base t) a) -> a) -> a
2 histo = gcata
3   . fmap extract fc :< fmap (distHisto . Cofree.unwrap) fc
4
5 gcata k g = g . extract . c where
6   c = k . fmap (duplicate . fmap g . c) . project
```

(wir geniessen jetzt nur noch)

Zygo Histo und weitere Varianten

- Die Komplexität weiterer Algorithmen nimmt jetzt stark zu
- Zygohistomorphic prepromorphisms sind eigentlich ein Witz
- *aber* Ed Kmett dachte sich: why not?
- wenn man Rekursion, alle vorigen Layer, und eine rekursiv angewandte natürliche Transformation will, warum nicht?

Was soll das Ganze?

- Wir koennen rekursive Algorithmen einmal generisch entwickeln
- Die Entwicklung konzentriert sich auf einzelne Ebenen, die *nicht rekursiv* sind
- Diese Idee ist fuer einige effiziente Compilerkonstruktionen wichtig
- Insbesondere Histomorphismen haben (unerwartet) praktische Anwendung in der dynamischen Optimierung
- die Basetypen koennen vom Compiler automatisch generiert werden

```
1 newtype Fix f = In (f (Fix f))
```