Typen, Konstruktoren, Typklassen, Typinferenz

something about types ...

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

Okt 27st, 2022

Datentypen: Sammlung verwandter Werte

```
1  data Bool = False | True
2  data Maybe a = Nothing | Just a
3  data [] a = [] | a : [a]              -- "eingebaut"
4  data Either a b = Left a | Right b
5  data Tree a = Tip | Node (Tree a) a (Tree a)
6
7  data Int = Int# I#          -- I# ist Black Magic
8
9  type AliasInt = Int
```

- Typ- und Datenkonstruktoren
- Typkonstruktoren haben 0 oder mehr Argumente
- Datenkonstruktoren haben Typen [ghci, info, type, kind]
- Pattern matching zur Dekonstruktion [wb, Maybe]

Infix-Konstruktoren

```
1  infixl 6 `Plus` (:+)        -- infix, infixr
2  data Parser a = Plus a a | a :+ a
```

Typen in Haskell

- Jede (wohlgeformte) Expression hat einen Typ!
- wenn *e* einen Wert vom Typ *t* produziert, dann hat *e* Typ *t*:
  `e :: t`
- *Alle* Typen sind zur Kompilierzeit bekannt, dort wo kein Typ steht wird der Compiler *Typinferenz* nutzen
- Typkonstruktoren und Funktionen koennen Variablen enthalten
- Type constraints schraenken den Raum dieser Variablen ein
- Typfehler werden zur Laufzeit gefunden (kein $1 +$ "hallo")

```
1  Just :: a -> Maybe a
2  sort :: Ord a => [a] -> [a]
```

Typklassen

- Typklassen stellen generische Operationen bereit (zB $(+)$ soll auf allen numerischen Typen funktionieren)
- Soweit moeglich sollten innerhalb einer Typklasse nur Operationen zu finden sein, die logisch Sinn machen ($(==)$ und $(/=)$, aber nicht $(+)$)
- werden Funktionen einer Typklasse innerhalb eines Ausdrucks / Funktion benutzt, so gibt es automatisch einen Constraint das die Funktion nur auf Mitglieder der Typklasse angewandt werden kann (`sort` sortiert nur Listen mit Constraint `Ord`)

Eq

```
1  class Eq a where
2    (==), (/=) :: a -> a -> Bool
3    x == y = not (x /= y)
4    x /= y = not (x == y)
5
6  instance Eq Bool where
7    True  == True  = True
8    False == False = True
9    _     == _     = False
```

Eq

```
1  data Failure a = Error String | Success a
2
3  instance Eq (Failure a) where
4    Error _   == Error _   = True   -- Semantik!
5    Success l == Success r = l == r
6    _         == _         = False
```

Deriving

```
:info Int
type Int :: *
data Int = GHC.Types.I# GHC.Prim.Int#
         -- Defined in ‘GHC.Types’
instance Eq Int -- Defined in ‘GHC.Classes’
instance Ord Int -- Defined in ‘GHC.Classes’
instance Enum Int -- Defined in ‘GHC.Enum’
instance Num Int -- Defined in ‘GHC.Num’
instance Real Int -- Defined in ‘GHC.Real’
instance Show Int -- Defined in ‘GHC.Show’
instance Read Int -- Defined in ‘GHC.Read’
instance Bounded Int -- Defined in ‘GHC.Enum’
instance Integral Int -- Defined in ‘GHC.Real’
```

Deriving

```
1  data Konto = Konto Int     -- oder Konto Rational
2    deriving (Eq,Ord,Enum, ...) -- PROFIT!
3
4  instance Eq Konto where
5    -- NO THANK You
```

Polymorphe Funktionen: Constraints in Action

```
1  quicksort :: Ord a => [a] -> [a]
2  quicksort [] = []
3  quicksort (x:xs) =
4    let ls = [y | y <- xs, y <= x]
5        rs = [y | y <- xs, y >  x]
6    in  quicksort ls ++ [x] ++ quicksort rs
```

Sie koennen alle Listen sortieren deren Elemente einen Ord-constraint haben [:info Ord]

Warum Typklassen und Typvariablen

- Typklassen sind nuetzlich
- Typklassen und deren Constraints geben dem Leser Hinweise was die Funktion tun kann
- Funktiontypen koennen viele, aber nicht alle Fehler abfangen
- (ueberladene) Funktionen mit Constraints gehoeren *nicht* zur Typklasse, sondern nutzen nur deren Funktionen

```
1  destroyWorld :: a -> a
2  destroyWorld x = ? -- Was kann diese Funktion tun?
3
4  wrsum :: Num a => [a] -> a
5  wrsum [] = 3 -- semantische Fehler koennen passieren
6  wrsum (x:y:zs) = x + sum zs -- strukturelle Fehler
7  wrsum [x] = x
```

Jeder fliegt 1. Klasse

```
1  :type ['a','b'] ==> [Char]
2  :type [sum, minimum] ==> (Num a, Ord a) => [[a] -> a]
3  :type [Left, Right] ==> [b -> Either b b]
```

[ghci, selbst machen, verwirrt sein?]

Lambda-Expressionen

- Haskell kennt natuerlich auch Lambda-Ausdruecke mittels \ (was ein λ sein soll)
- Sie koennen hier quasi beliebig komplizierte Funktionen hinschreiben
- Sie sollten aber nur sehr kurze Ausdruecke nutzen
- Partiell angewandte Funktionen sind stattdessen haeufig besser

```
1  :type (\x -> x+2) :: Num a => a -> a
2  :type (+2)
3
4  :type filter (/='Z')
5
6  gerade = filter even
```

Hausaufgabe

```
1  tail :: [a] -> [a]
2  tail [1,2,3] == [2,3]
3  tail [] = error ...
4
5  ohneAnfang :: [a] -> Maybe [a]
6  ohneAnfang [1,2,3] == Just [2,3]
7  ohneAnfang [] == Nothing
```

Schreiben Sie `ohneAnfang` auf verschiedene Weisen, if-then-else, guards, Pattern Matching

Strings

- Haskell kennt einen Stringtyp `type String = [Char]`
- Dieser ist ausreichend fuer unsere Beispiele
- Fuer "echte" Programme gibt es: `ByteString`, `Text`, ...

Einfache Funktionskombinatoren

```
1  sum (map (*2) (filter even [1..100]))
2
3  sum . map (*2) $ filter even [1..100]
4
5  :type sum . map (*2) . filter even -- ??
6
7  infixr 9 .
8  (.)     :: (b -> c) -> (a -> b) -> a -> c
9  (.) f g = \x -> f (g x)
10
11 infixr 0 $
12 ($) :: (a -> b) -> a -> b
13 f $ x =  f x
```

Kompliziertere Listengeneratoren

```
1  [(x,y) | x <- [1..10], y <- [x..10]]
2  concat :: [[a]] -> [a]
3  concat xss = [x | xs <- xss, x <- xs]
4  cceteven xss = [x | xs <- xss, even (length xs)
5                    , x <- xs, even x]
6
7  faktoren n = [x|x <- [1..n], n `mod` x == 0]
8  prime n = faktoren n == [1,n]
9  primes = [x | x <- [2..], prime x]
10 -- take 100 primes
```

Der Reisverschluss

```
1  zipWith :: (a->b->c) -> [a] -> [b] -> [c]
2  zipWith f = go
3    where
4      go [] _ = []
5      go _ [] = []
6      go (x:xs) (y:ys) = f x y : go xs ys
7
8  zip = zipWith (,) -- testen!
```

Mehr Rekursion auf Listen

```
 1  filter :: (a->Bool) -> [a] -> [a]
 2  filter f [] = []
 3  filter f (x:xs) = if f x then x : filter f xs else filter f
 4
 5  map :: (a->b) -> [a] -> [b]
 6  map f [] = []
 7  map f (x:xs) = f x : map f xs
 8
 9  (++) :: [a] -> [a] -> [a]
10  [] ++ ys = ys
11  (x:xs) ++ ys = x : xs ++ ys    -- Achtung: infixr 5 : ++
12
13  reverse [] = []
14  reverse (x:xs) = reverse xs ++ [x]
15
16  reverse' xs = let
17      rev [] acc = acc
18      rev (x:xs) acc = rev xs (x:acc)
19    in go xs []
```

Hausaufgabe

```haskell
data Ausdruck
  = Wert Int
  | Ausdruck :+ Ausdruck
  | Ausdruck :* Ausdruck
  deriving (Show)

instance Num (Ausdruck) where
  fromInteger x = Wert x     -- oh dear

ausdruck = Wert 1 :+ (Wert 2 :* Wert 3) -- 1 :+ (2 :* 3)

tiefe :: Ausdruck -> Int
tiefe = error "Tiefe des Ausdrucksbaumes"
wert :: Ausdruck -> Int
wert = error "Wert dieses Ausdruck"
```

Binaerbaeume

```
 1  data Tree a = Tip | Node (Tree a) a (Tree a)
 2
 3  empty = Tip :: Tree a
 4
 5  insert Tip a = Node Tip a Tip
 6  insert (Node ls x rs) a
 7    | a == x = Node ls x rs
 8    | a < x = Node (insert ls a) x rs
 9    | a > x = Node ls x (insert rs a)
10
11  exists Tip a = False
12  exists (Nodes ls x rs) a
13    | a==x = True
14    | a< x = exists ls a
15    | a> x = exists rs a
16
17  -- data Tree k v = ...
```

Deep Magic!

```
1  data V (k :: Nat) a where
2    Nul :: V 0
3    (:>) :: a -> V k -> V (k+1)
4
5  length :: forall k . V (k::Nat) -> Int
6  length _ = fromIntegral (natVal Proxy :: Proxy k)
```

Datentypen

- Neue Datentypen `data TyCon a b = DataCon a b`
- TypAliase `type T = S`
- `newtype TyCon a = DataCon a`

```
1  data TyData a b c = HereD a b | ThereD c
2
3  type MyInt = Int
4
5  newtype NewTy X = NewTy Int
```

The Roller Coaster

```
 1  data Person = P { age :: Int, height :: Int }
 2  guard :: Person -> Bool
 3  guard (P a h) = not (a < 12 || h < 100)
 4  coaster = map guard
 5
 6  guys =
 7    coaster [P 11 110, P 8 90, P 117 13, P 13 130]
 8
 9  newtype Age = Age Int
10  newtype Height = Height Int
11  ... P { age :: Age, height :: Height }
```

Der Guard lässt nur das dritte und vierte Kind fahren

Haskell Prelude

```
 1  head [1..5] == 1
 2  tail [1..5] == [2..5]
 3  [1..5] !! 2 == 3
 4  take 2 [1..5] == [1..2]
 5  drop 2 [1..5] == [3..5]
 6  length [1..5] == 5
 7  take 2 [1..10^100] == [1..2]
 8  take 2 [1..] == [1..2]
 9  sum [] == 0
10  sum [1..5] == 15
11  product [] == 1
12  product [1..5] == 120
13  reverse [1..5] == [5,4..1]
```

Implementation nach `data List`

Funktionen anwenden

Mathematisch: $f(a, b) = a + b$, mit zB. $f(1, 2) = 3$.

```
1  f :: (Int,Int) -> Int
2  f (a,b) = a+b              -- f(1,2) == 3
```

In Haskell, wir bevorzugen allerdings Funktionen die *curried* sind:

```
1  f :: Int -> Int -> Int
2  {- f :: Int -> (Int -> Int) -}
3  f a b = a+b
4
5  f 1 2 + 3    == (f 1 2) + 3 == 3 + 3 == 6
```

Hilfreich: im Interpreter `:type Funktionsname` eingeben!

Kompliziertere Funktionen

```
1  fakultaet x = product [1..x]
2  mittelwertI :: ?
3  mittelwertI xs
4    = sum xs `div` (length xs)
5  mittelwertR :: ?
6  mittelwertR xs = s / l
7    where s = fromIntegral (sum xs)
8          l = fromIntegral (length xs)
```

Zum Nachdenken!

```
1  malZwei :: Int -> Int
2  malZwei x = 2 * x
3
4  malVier :: Int -> Int
5  malVier x = malZwei (malZwei x)
6
7  potenz :: Int -> Int -> Int
8  potenz potenz x = ???
```