Tutorium_3

November 25, 2020

1 Tutorium 3

Gruppen 2 und 4
Samuel Teuber
propa@teuber.dev
https://teuber.dev/propa

2 Übungsblätter

- **Fragen? Probleme?** (Praktomat?)

UMFRAGE

3 Vorbemerkung 0

Haskell ist **anders**, aber **lasst euch davon nicht verwirren!** 1. Problem lesen 2. Algorithmus überlegen/nachschauen 3. Evtl. Pseudocode 4. Wie implementiere ich **diesen Mechanismus** in Haskell?

Mergesort: Nutzt die Algorithmen, die angegeben sind!

4 Vorbemerkung 1

```
f (x:xs) a
    | a = [x]
    | otherwise = x:xs
```

Wie kann man das schöner schreiben?
Zwei Optimierungsmöglichkeiten:

```
f (x:xs) True = [x]
f l False = l
```

...oder...

```
f l@(x:xs) a
    | a = [x]
    | otherwise = l
```

```
In [1]: f l@(x:xs) a
            | a = [x]
            | otherwise = l
        f [1,2,3] True
        f [1,2,3] False
```

```
[1]


[1,2,3]
```

5 Vorbemerkung 2

```
In [1]: f x
          | even x = x `div` 2
          | odd x = (x-1) `div` 2

        f' x = x `div` 2
```

```
In [2]: f 5
        f' 5
```

```
2


2
```

6 Aufgabe 1: Polynome

```
type Polynom = [Double]
```

- Polynom Multiplikation mit Konstante

```
cmult :: Polynom -> Double -> Polynom
cmult p c = map (*c) p
```

- Polynom Auswertung mit dem Horner Schema

$$a_0 + x * (a_1 + x * (a_2 + \ldots (a_{n-1} + x * a_n) \ldots))$$

```
eval :: Polynom -> Double -> Double
eval p x = foldr (\a v -> a + v*x) 0 p
```

- Ableitung

```
deriv :: Polynom -> Polynom
deriv [] = []
deriv p  =zipWith (*) [1..] (tail p)
```

7 Aufgabe 2: Collatz

- Unendliche Liste der Folgenmitglieder für Startwert `a0`

```
collatz :: Int -> [Int]
collatz m = iterate next m
    where
            next :: Int -> Int
            next an
                | an `mod` 2 == 0 = an `div` 2
                | otherwise = 3 * an + 1
```

- Zählen bis zur ersten 1

```
num m = length (takeWhile (/= 1) (collatz m)))
```

- Suche nach maximalem `num` in Interval

```
maxNum a b = foldl maxPair (0,0) (map (\m -> (m, num m)) [a..b])
    where maxPair (m,n) (m',n') = if n > n' then (m,n) else (m',n')
```

8 Aufgabe 3: Prime Powers

Klausuraufgabe - Merge auf unendlichen Listen

```
merge [] b = b
merge a [] = a
merge l1@(x:xs) l2@(y:ys)
    | (x<y) = x:merge xs l2
    | otherwise = y:merge l1 ys
```

- Erste n Potenzen aller Primzahlen

```
primepowers n = foldr merge [] [map (^i) primes | i <- [1..n]]
```

9 B-Seite: Termsprachen

Umfrage

10 Noch ein Wort zur Frage von letzter Woche...

Typ [I]: (a -> a) -> (a -> a)
Typ [II]: a -> a

> Wenn eine Funktion Typ [II] hat, "dann hat sie eigentlich auch Typ [I]"
>
> Wenn eine Funktion Typ [II] hat, dann kann man sie verwenden als hätte sie Typ [I]

11 Typen

...schon wieder... - Jeder Ausdruck in Haskell hat einen Typ - Haskell ist damit statisch typisiert - Jeder Ausdruck wertet zu Werten des jeweiligen Typs aus - Typen werden **automatisch inferiert**

12 Polymorphe Typen

...schon wieder...

Beispiel: Listen-Typ - `[t]` ist der Typ von Listen mit Elementen vom Typ `t` - *Typvariable* `t` beliebig

Sichtweisen: - Typvariablen parametrisieren polymorphe Typen - Typkonstruktoren wie `[ ]` erzeugen neue Typen aus bestehenden

Vergleichbares in Java: Generics

```
LinkedList<T>
```

12.1 Funktionstypen

... sind polymorph, denn...

```
s -> t
```

... beschreibt den Typ aller Funktionen von `s` nach `t` (`s`,`t` beliebig)

13 Tupel

...sind polymorph, denn...

```
(,) :: s -> t -> (s,t)
```

...und umgekehrt...

```
fst :: (s,t) -> s
snd :: (s,t) -> t
```

14 Typsynonyme

Verbessert die Lesbarkeit:

```
type Polynom = [Double]
```

15 Explizite Typisierung

...kennen wir auch schon...

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

- Übersichtlichkeit
- Hilft beim Debugging

16 Wollen wir statt Klassen einfach Tupel benutzen?

```
type Name = (String, String)
type MNumber = Integer -- Für die Addition von Matrikelnummern...
type Address = (String, Integer, String) -- Strae, PLZ und Ort
type Grades = [(String, Double)] -- Fach Name und Note
type Student = (Name,MNumber, Address, Grades)
```

Ist das eine gute Idee?

17 NEIN

- Tupel könnte für beliebige Werte benutzt werden
- Kaum/Keine explizite Bedeutung

18 Algebraische Datentypen

Definition neuer Typen durch Auflistung aller *Konstruktoren*:

```
data Address = Address String Integer String
data UniversityPerson = Student String Integer Address
                      | Professor String Address
```

Ergibt die folgenden Konstruktoren:

```
Address :: String -> Integer -> String -> Address
Student :: String -> Integer -> Adress -> UniversityPerson
Professor :: String -> Adress -> UniversityPerson
```

Jede*r `Student` und jede*r `Professor` ist eine `UniversityPerson`

19 Pattern-Matching

```
In [1]: data UniversityPerson = Student String Integer
                              | Professor String
        getName :: UniversityPerson -> String
        getName (Student name _) = name
        getName (Professor name)  = name

        getName (Student "Samuel" 424242)
        getName (Professor "Prof. Dr. Snelting")
```

```
"Samuel"
```

```
"Prof. Dr. Snelting"
```

20 Polymorphe Algebraische Datentypen

z.B. Realisierung von **optionalen Werten**:

```
data Maybe t = Nothing | Just t
Just True :: Maybe Bool
```

21 Polymorphe Algebraische Datentypen (2)

```
In [1]: data Either t s = Option1 t | Option2 s

In [2]: -- Übungsaufgabe 1 (2min)
        process :: (t -> a) -> (s -> a) -> Either t s -> a
        process f _ (Option1 a) = f a
        process _ g (Option2 b) = g b

In [3]: printBool :: Bool -> String
        printBool False = "Aww, it's false"
        printBool True = "Yay, it's true"

        printInt :: Int -> String
        printInt x
            | x<=0 = "Nope, not gonna spread this negativity"
            | (x>0) && (x<100) = "This number is pretty small"
            | otherwise = "Woah that's a big number"

        print' = process printBool printInt

        :t print'
        print' (Option1 False)
        print' (Option2 (-101))
```

```
print' :: Either Bool Int -> String
```

```
"Aww, it's false"
```

```
"Nope, not gonna spread this negativity"
```

22 Polymorphe Rekursive Algebraische Datentypen

```
data Tree t = Leaf t | Node (Tree t) t (Tree t)
```

```
In [4]: -- Übungsaufgabe 2:
        -- Definiere einen polymorphen Datentyp für einfach verkettete Listen
        -- Definiere eine Operation zur Berechnung der Listenlänge
```

```
        data List a = End | Part a (List a)

        listLength :: List a -> Int
        listLength End = 0
        listLength (Part a l) = 1 + (listLength l)

        listLength (Part 2 End)
```

```
Line 8: Redundant bracket
Found:
1 + (listLength l)
Why not:
1 + listLength l
```

```
1
```

23 Typklassen

...endlich... - Zusammenfassen von Typen anhand von definierter Operationen - Java: ~Interfaces **Beispiel** `Eq`:

```
class Eq t where
    (==) :: t -> t -> Bool
    (/=) :: t -> t -> Bool
```

Default Implementierungen:

```
    x /= y = not (x == y)
    x == y = not (x /= y)
```

24 Typklassen implementieren

```
instance Eq Bool where
    True == True = True
    False == False = True
    x == y = False
```

Auch für Polymorphe Typen:

```
In [14]: data Maybe t = Nothing | Just t

         class Defaultable t where
             getDefault :: t a -> a -> a

         instance Defaultable Maybe where
             getDefault (Just x) c = x
```