FPV Cheatsheet

Funktionale Programmierung (Technische Universität München)

Type Inference

:t …. in haskell

Algorithm

1. Give the variables $x_1, \dots, x_n$ in e the types $a_1, \dots a_n$ where the a_i are distinct type variables.
2. Give *each occurrence* of a function $f :: \tau$ in e a new type τ' that is a copy of τ with fresh type variables.
3. For each subexpression $f\ e_1 \dots e_n$ of e where $f :: \tau_1 \to \cdots \to \tau_n \to \tau$ and where e_i has type σ_i generate the equations $\sigma_1 = \tau_1, \dots, \sigma_n = \tau_n$.
4. Simplify the equations with the following rules as long as possible:
   - $a = \tau$ or $\tau = a$: replace type variable a by τ everywhere (if a does not occur in τ)
   - $T\ \sigma_1 \dots \sigma_n = T\ \tau_1 \dots \tau_n \rightsquigarrow \sigma_1 = \tau_1, \dots, \sigma_n = \tau_n$ (where T is a type constructor, e.g. `[.]`, `.->.`, etc)
   - $a = T \dots a \dots$ or $T \dots a \dots = a$: type error!
   - $T \dots = T' \dots$ where $T \neq T'$: type error!

170

List Comprehension

1. **[x | x <- xs] = xs**
2. **[x | x <- xs, x > 1] = filter (>1)**
3. **[(xs !! i, ys !! i) | i <- [0..(min length xs length ys)]] = zip xs ys**
4. **[(x,y) | x <- xs, y <- ys] = kreuzprodukt**
   a. first [1,2,4] [5,6,7]
      i. =>> [(1,5),(1,6),(1,7),(2,5),(2,6),(2,7),(4,5),(4,6),(4,7)]
5. **[[(x, y) | x <-xs] | y <- ys]**
   a. second [1,2,4] [5,6,7]
      i. =>>[[(1,5),(2,5),(4,5)],[(1,6),(2,6),(4,6)],[(1,7),(2,7),(4,7)]]

Higher Order Functions

1. **map :: (a -> b) -> [a] -> [b]**
   a. map (>1)
   b. map (recip . negate)
   c. map f . map g = map (f.g)
2. **zip :: [a] -> [b] -> [(a,b)]**   **(zip3 [a] -> [b] -> [c] -> [(a,b,c)])**

3. **unzip :: [(a,b)] -> ([a], [b])** **(unzip3 [(a,b,c)] -> ([a],[b],[c]))**
4. **zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]** **(unzipWith)**
    a. zipWith (\x y -> 2*x + y) [1..4] [5..8] => Output: [7,10,13,16]

5. **foldr :: (a -> b -> b) -> b -> [a] -> b**

```
a. foldr (&&) True [1>2,3>2,5==5] => TRUE
b. foldr (\x y -> (x+y)/2) 54 [12,4,10,6] => 12.0
c. foldr max 18 [3,6,12,4,55,11]  => 55
```

6. **curry** :: ((a,b) -> c) -> (a -> b -> c)
    a. **curry f = \ x y -> f (x , y)**

7. **uncurry** :: (a -> b -> c) -> ((a,b) -> c)
    a. **uncurry f = \(x,y) -> f x y**

```
b. f6 f (x,y) = f x y    ⇔   f6 = uncurry
```

8. **iterate :: (a -> a) -> a -> [a]**
    a. creates an infinite list where the first item is calculated by applying the function on the second argument, the second item by applying the function on the previous result and so on.
    b. take 10 (iterate (2*) 1)

9. **takeWhile :: (a -> Bool) -> [a] -> [a] bzw. dropWhile**

```
a. takeWhile p [] =  []
b. takeWhile p (x:xs)
c.     | p x        = x : takeWhile p xs
d.     | otherwise = []
```

10. **cycle :: [a] -> [a]**
    a. take 10 (cycle [1,2,3]) => [1,2,3,1,2,3,1,2,3,1]

Coole Funktions

1. **recip** :: Fractonal a => a -> b (Fractional =>)
    a. a/b => b/a
2. **and** :: [Bool] -> Bool
3. **any** / **all** :: (a -> Bool) -> [a] -> Bool
4. **filter** :: (a -> Bool) -> [a] -> [a]
5. **elem** / **notElem** :: a -> [a] -> Bool
6. **reverse** :: [a] -> [a]
7. **splitAt** :: int -> [a] -> ([a],[a])
8. gcd :: Integral a => a -> a -> a (grosste gemeinsame Teiler)
9. **flip** : (a -> b -> c) -> b -> a -> c (flip (>) 3 5 = True)

10. **nubby** : (a -> a -> Bool) -> [a] -> [a]
    (nubBy (\x y -> x+y == 10) [2,3,5,7,8] = [2,3,5]
      nubBy (\x y -> x+y == 10) [8,7,5,3,2] = [8,7,5]

Strukturen

1. type
    a. type Name = String
2. data
    a. data Tree a = Leaf | Node (Tree a) a (Tree a)
       deriving (Eq, **Show**, Ord, Read …..)
3. class

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

4. instance

```
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False

instance Eq a => Eq [a] where
  [] == [] = True
  (x:xs) == (y:ys) = x == y && xs == ys
  _ == _ = False
```

5. hints:
    a. alle Methoden aus einer Klasse müssen in der instance implementiert werden.

Proofs

1. ÜL Cheatshit
    a. https://github.com/lukasstevens/cyp/blob/master/cheatsheet.md
2. cd 'C:\Users\nensi\Desktop\fpv exam\cyp\cyp\'
3. stack run cyp ‘C:\Users\nensi\Desktop\fpv exam\EXAM\thy.cthy’ ‘C:\Users\nensi\Desktop\fpv exam\EXAM\proof.cprf’
4. stack run cyp ‘C:\Users\nensi\Desktop\fpv exam\EXAM\thy_2.cthy’ ‘C:\Users\nensi\Desktop\fpv exam\EXAM\proof_2.cprf’

(Proof mirror1 : mirror (mirror t) = id t)

Lemma mirror2 : (mirror . mirror) .=. id
Proof by extensionality with t

To show : (mirror . mirror) t .=. id t
Proof
(mirror . mirror) t
(by def .) .=. mirror (mirror t)
(by mirror1) .=. id t
QED

QED

IO

Basic actions

- **getChar :: IO Char**
  Reads a Char from standard input, echoes it to standard output, and returns it as the result
- **putChar :: Char -> IO ()**
  Writes a Char to standard output, and returns no result
- **return :: a -> IO a**
  Performs no action, just returns the given value as a result
- **getLine :: IO String**
- **print :: IO () (Show a => a -> String)**
- **putStrLn :: String -> IO String**
- **Example of do and terminating**

```
ioLoop :: (a -> Maybe b) -> IO a -> IO b
ioLoop f i = do
                xs <-  i
                case f xs of
                    Nothing -> ioLoop f i
                    Just s -> return s
```

In ReadMode

•**hGetChar :: Handle -> IO Char**

•hGetLine :: Handle -> IO String
•hGetContents :: Handle -> IO String
Reads the whole file lazily

In **WriteMode**

•hPutChar :: Handle -> Char -> IO ()
•hPutStr :: Handle -> String -> IO ()
•hPutStrLn :: Handle -> String -> IO ()
•hPrint :: Show a => Handle -> a -> IO ()

The simple way

•type FilePath = String
•readFile :: FilePath -> IO String Reads file contents lazily,only as much as is needed
•writeFile :: FilePath -> String -> IO () Writes whole file
•appendFile :: FilePath -> String -> IO () Appends string to file

Files and handles

•data IOMode = ReadMode | WriteMode| AppendMode | ReadWriteMode
•openFile :: FilePath -> IOMode -> IO Handle
Creates handle to file and opens file
•hClose :: Handle -> IO () **Closes file**

- `type FilePath = String`
- `readFile :: FilePath -> IO String`

  Reads file contents *lazily*,
  only as much as is needed
- `writeFile :: FilePath -> String -> IO ()`

  Writes whole file
- `appendFile :: FilePath -> String -> IO ()`

  Appends string to file

stdin and stdout

- stdin :: Handlestdout :: Handle
- getChar = hGetChar stdinputChar = hPutChar stdout

Example (interactive cp: icp.hs)

```
main :: IO()
main =
  do fromH <- readOpenFile "Copy from: " ReadMode
     toH <- readOpenFile "Copy to: " WriteMode
     contents <- hGetContents fromH
     hPutStr toH contents
     hClose fromH
     hClose toH

readOpenFile :: String -> IOMode -> IO Handle
readOpenFile prompt mode =
  do putStrLn prompt
     name <- getLine
     handle <- openFile name mode
     return handle
```

Lazy Evaluation

1. Beispiel:
   (\ f -> \ g -> g . map f) (+1) head odds
   (\ g -> g . map (+1)) head odds
   (head . map (+1)) odds
   (\ x -> head (map (+1) x)) odds head
   (map (+1) odds) head
   (map (+1) (1 : map (+2) odds)) head
   (((+1) 1) : map (+1) (map (+2) odds))
   (+1) 1
   2

Abkürzungen Syntax (Beispiele)

```
f1 xs = map (\x -> x + 1) xs
f1' = map (+1)

f2 xs = map (\x -> 2 * x) (map (\x -> x + 1) xs)
f2' = map (2*) . map (+1)
```

```haskell
f2'' = map ((2*) . (+1))

f3 xs = filter (\x -> x > 1) (map (\x -> x + 1) xs)
f3' = filter (>1) . map (+1)

f4 fs = foldr (\f acc -> f acc) 0 (map (\f -> f 5) fs)
f4' fs = foldr (\f acc -> f acc) 0 (map ($5) fs)
f4'' = foldr (\f acc -> f acc) 0 . map ($5)
-- Type inference fails after this
f4''' :: (Num a,Num b) => [(a->b->b)] -> b
f4''' = foldr (\f acc -> (f 5) acc) 0
f4'''' :: (Num a,Num b) => [(a->b->b)] -> b
f4'''' = foldr (\f -> (f 5)) 0
f4''''' :: (Num a,Num b) => [(a->b->b)] -> b
f4''''' = foldr ($5) 0

f5 f g x = f (g x)
f5' f g = f . g
f5'' f = (.) f
f5''' = (.)

f6 f (x,y) = f x y
f6' = uncurry

f7 f x y z = f z y
f7' f x = flip f
f7'' f = const (flip f)
f7''' = const . flip


f8 f g x y = f (g x y)
f8' f g x = f . g x
f8'' f g x = (f.) (g x)
-- f (g x) = (f . g) x for any functions f, g
f8''' f g = ((f.) . g)
f8'''' f g = (.) (f.) g
f8''''' f = (.) (f.)
f8'''''' f = (.) ((.) f)
f8''''''' f = ((.) . (.)) f
-- It's the Haskell owl!
f8'''''''' = (.).(.)
```

- Type - zeigt ein Synonym (z.B : type String = [Char]   ,   type Student = (String,Int))

- new type - kann nur ein attribut nehmen
- data - beliebig

----

curing -> verienfachen -> functions : http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Function.html