1 Chapter 1 - Introduction

> [...] functional programming can be viewed as a style of programming in which the basic method of computation is the application of functions to arguments. – Graham Hutton in Programming in Haskell

2 Chapter 2 - First Steps

Lazy Evaluation Lazy Evaluation is an evaluation strategy which delays the execution of an expression until its value is needed. The programming language Haskell is one language which use this strategy.

Haskell Standard Prelude The standard prelude is the standard module in Haskell which is always included unless you have the extension `NoImplicitPrelude` enabled. In this module are the most common functions, types and monads. This module is required to be able to implement any Haskell application.

Haskell Naming Requirements Every Haskell function has to start with a lower case. After that zero or more letters (upper and lowercase), digits, underscore and forward single quotes. Additionally, the keywords (Haskell Keywords) can not be used for function names or their arguments.

Haskell Naming Convetions List arguments have in Haskell normally the suffix `s`. Therefor, a list of numbers might be names `ns`. A list of arbitrary elements `xs` and a list of chars `css`.

3 Chapter 3 - Types and classes

Currying Currying is the process of turning a multidimensional function into a function which takes one argument and returns a function which takes again one argument. In Haskell and many other Functional Programming languages (What is functional programming?) this is a common practice.

```
add :: Int -> Int -> Int
add x y = x + y


a = add 3 -- type Int -> Int
b = a 4   -- type Int (7)
```

Type Variables Type Variables are something like a placeholder for a real type. See the type of the `length` function:

```
length :: [a] -> Int
--
--          |
-- type variable
```

The function `length` can be applied to any list of any type. This is indicated with the type variable `a`. A type variable must start with a lower-case letter. Typically names are `a`, `b` and so on.

Polymorphic types In Haskell a type (functions are also types) is called polymorphic when it contains one or more type variables (What are type variables in Haskell?).

Classes Classes in Haskell are not quite the same as in OO languages. A class is a collection of types that support a certain overloaded operation called *method*. Therefore, classes are more similar to Interfaces (Interfaces in C#).

Class Constraints Using type variables (What are type variables in Haskell?) you can create polymorphic types (Polymorphic types in Haskell). These type can than applied to any other types. However, sometimes you want some restriction for example the `+` operation should only take numbers. This can be achieved using class constraints.

```
(+) :: Num a => a -> a -> a
--          ^
--          |
-- class constraint
```

Now the type variable `a` must be an instance of the class (Classes in Haskell) `Num`.

Type Overloads A type is called overloaded when it constrains one or more class constraints (What are class constraints in Haskell?).

4 Chapter 4 - Defining Functions

Guarded Equations In Haskell you have multiple ways to choose between a number of possible results. The function `abs` returns the absolute value of a number `n`. You could write the `abs` function with guarded equations.

```
abs n | n >= 0 = n
      | otherwise = -n
```

```
-- in std prelude
-- otherwise = True
```

Haskell reads from top to bottom. The first equation which evaluates to true is taken. `otherwise` is nothing else as `True`. The `|` symbol is read as *such as*.

Pattern Matching Pattern Matching is very useful tool in Haskell. It can also be used to define functions.

```
fst :: (a,b) -> a
fst (x, _) = x

snd :: (a,b) -> b
snd (_, y) = y
```

The `_` symbol matches everything.

List and Cons Lists are no primitives as `Int` for example. They are constructed from the empty list `[]` using the `:` cons operator. The following resulting always in the same list:

```
[1,2,3]
1 : [2,3]
1 : 2 : [3]
1 : 2 : 3 : []
```

This can be really useful to define functions:

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

In the function definition the `()` are required. Otherwise, it would cause syntax error (it would try to call tail / head with an argument).

Lambdas In Haskell lambdas are defined using the `\` symbol (should represent the λ symbol).

```
\x -> x + x
(\x -> x + x) 2


add :: Int -> Int -> Int
add = \x -> \y -> x + y

odds :: Int -> [Int]
odds n = map (\x -> x*2 + 1) [0..n-1]
```

Operators A function which takes two arguments are called *operators* in Haskell. Mathematical functions are often written like the following:

```
8 `div` 2
```

But the typical way is also possible:

```
(+) 8 2 -- the parenthese are required
-- >>> :t (+)
-- (+) :: Num a => a -> a -> a

(1+) 2 -- valid
(+2) 1 -- valid
```

Operator Sections Operator Sections are the partial evaluated parts of an operator (What are operators in Haskell?). The following are examples are operator sections:

```
(+)
(1+)
(+2)

(op) = \x -> (\y -> x op y)
(x op) = \y -> x op y
(op y) = \x -> x op y
```

Operator Sections are required because an operator alone is not a valid expression in Haskell. Therefore, for functions like `foldl` operation sections are required:

```
sum :: [Int] -> Int
sum xs = foldl (+) 0 xs
```

5 Chapter 5 - List comprehensions

List comprehension In Math you can create from one set of numbers a new set of numbers:

$$\{x^2 | x \in \{1 \dots 5\}\} \tag{1}$$

In Haskell you can do the same:

```
[x^2 | x <- [1..5]]
--   ^    ^    ^
--   |    |    |--- |---------- Generator
-- "such as" |
--           "is drawn from"
```

The list comprehension requireds at least one generator but multiple generators are allowed.

```
[(x, y) | x <- [1,2,3], y <- [4,5]]
[(x, y) | x <- [1..3], y <- [x..3]]
```

Usage of list comprehensions When you are creating new lists using list comprehension (What is list comprehension in Haskell?) you dont always want to take all values. When creating a list using list comprehension (What is list comprehension in Haskell?) you have sometimes certain criterias which must be true. For example you want a list with only even numbers. This can be achived using a *guard*.

Evaluates the guard to `True` the value is taken. Otherwise, the value is discareded.

```
[x | x <- [1..10], even x] -- generates
    a list of even numbers
--                      ^
--                      |
--                    Guard
```

6 Chapter 6 - Recursive Functions

Recursion It exists different kind of recursion:

- linear & tail recursion (What is linear recursion?)
- multiple recursion (What is multiple recursion?)
- mutual recursion (What is mutual recursion?)

Recursive Functions

1. Define type
2. Enumerate the cases
3. Define the simple case
   - Often the base case
4. Define the other cases
   - Often the recursive cases
5. Generalize and simplify

```
-- Step 1
product :: [Int] -> Int

-- Step 2
product :: [Int] -> Int
product :: [] =
product :: (n:ns) =

-- Step 3
product :: [Int] -> Int
product :: [] = 1
product :: (n:ns) =

-- Step 4
product :: [Int] -> Int
product :: [] = 1
product :: (n:ns) = n * product ns


-- Step 5
product :: Num a => [a] -> a
product = foldr (*) 1
```

7 Chapter 7 - Higher-order Functions

Higher-Order Functions A higher-order function is a function which takes another function as an argument and / or returns a function. However, returning a function is often only called a curried function (What is curring in terms of math?) and higher-order means often only taking a function as argument.

```
isEven :: [Int] -> [Bool]
isEven xs = map even xs

isEven [1,2,3,4,5,6,7]
[False,True,False,True,False,True,False]
```

foldr and foldl *foldr* and *foldl* are the abbreviation for fold right / left. Many functions can be defined using the following pattern:

```
f [] = v -- some value
f (x:xs) = x # f xs -- for example x + f
    xs
```

The function *foldr* can be used to encapsulate this pattern. You can think that *foldr* replaces the `:` with the operation `#`.

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs


sum_foldr :: Num a => [a] -> a
sum_foldr = foldr (+) 0

{-
sum_foldr [1,2,3] =
sum_foldr 1 : (2 : (3 : [])) =
1 + (2 + (3 + 0)) =
6
-}

sum_foldl :: Num a => [a] -> a
sum_foldl = foldl (+) 0
{-
sum_foldl [1,2,3] =
sum_foldl 1 : (2 : (3 : [])) =
((0 + 1) + 2) + 3 =
6
-}
```

Composition Operator The composition operator takes two function as argument and returns the composition of both. Therefore, is `(f . g) x = f (g x)` true. This is very useful to simplify nested function calls.

```
odd :: Int -> Bool
odd n = not (even n)
odd n = not . even n

twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

```
sumsqreven :: [Int] -> [Int]
sumsqreven xs = sum (map (^2) (filter
    even xs))
sumsqreven = sum . map (^2) . filter
    even

sumsqreven [1,2,3,4,5] -- 20
```

8 Chapter 8 - Declaring types / classes

Create new types In Haskell, you have multiple ways to create an new type.

- `type` (The type keyword in Haskell)
- `data` (The data keyword in Haskell)
- `newtype` (The newtype keyword in Haskell)

type Using the keyword `type` you create basically an alias for the type.

```
type String = [Char]
```

Recursive definition for a new type using `type` is **not** allowed:

```
type Tree = (Int, [Tree]) -- Not allowed
```

Typed declared using the `type` mechanism can not be made into new instances of classes (Classes in Haskell).

data

Using the keyword `data` you create real new types.

```
data Bool = False | True
data Move = North | South | East | West
```

The new values (North, South, ...) must be unique and are called **constructors**.

```
data Shape = Circle Float | Rect Float
    Float


area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y


{-
:t Circle
Circle :: Float -> Shape
:t Rect
Rect :: Float -> Float -> Shape
-}
```

The `Circle` and `Rect` from the example are called **constructor functions**.

newtype Different to the `type` keyword (The type keyword in Haskell) with `newtype` you create a real new type.

```
newtype Nat = N Int

someFunc :: Nat -> Nat
someFunc x = x
```

For the function `someFunc` you can **not** use an `Int` (type safety). However, at compile time the `Nat` is replaced with `Int`. Therefore, I have type safety during development without performance impact (`data Nat = N Int` would be slower).

class

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

implement class

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

data T a = Leaf | Node (T a) a (T a)

instance Show a => Show (T a) where
  show x = show (toList x)

instance Eq a => Eq (T a) where
  (==) x y = toList x == toList y

instance Ord a => Semigroup (T a) where
  x <> y = merge x y

instance Ord a => Monoid (T a) where
  mempty = empty
```

9 Chapter 10 - Interactive Programming

Interactive Programming Haskell is a pure functional programming language. Therefore, side effects are not allowed and because of this interactive programming should not be possible. Haskell overcomes this problem using IO actions (IO actions in haskell).

IO Action `IO` can be seen as a data type which represents the whole world and is built-in into Haskell. The most important actions for interactive programming are:

- `getChar`
- `putChar`
- `return`

```
getChar :: IO Char
putChar :: Char -> IO ()
return :: a -> IO a
```

`return` is a very important action which produces from a normal value an action with this value.

```
myFunc :: IO a
myFunc = do v1 <- a1
            v2 <- a2
            v3 <- a3
            vn <- an
            return (f v1 v2 v3 vn)
```

10 Chapter 12 - Monads and more

Functor A functor is a class (Classes in Haskell) which provides the `fmap` function. To be a valid functor the restrictions from listening 1.

> Functors are types that wrap values of other types and allow us to map functions over the wrapped value. – Fahrad Metha

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

fmap id = id
fmap (g . h) = fmap g . fmap h
```

Listing 1: Functor laws

Applicatvie Functors `fmap` takes a function which takes only one argument (What is a functor in haskell?).

```
fmap :: Functor f => (a -> b) -> f a ->
    f b
```

Sometimes it is required to provide a function with two or more input arguments. For this use case the `Applicative` typeclass (Classes in Haskell) exists (see 2).

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b


g <*> x <*> y <*> z == ((g <*> x) <*> y)
    <*> z
pure g <*> x1 <*> x2 <*> xn
```

Listing 2: Applicative Class

The `pure` function returns a functor with a value, noramally a function (see 3). The function must be a curried (What is curring in terms of math?) one and takes n arguments. The `<*>` operator is now used to perform partial application and is used n times. The `<*>` operator uses `fmap` to apply the function to the elemens.

```
instance Applicative Maybe where
  -- pure :: a -> Maybe a
  pure = Just
  -- (<*>) :: Maybe (a -> b) -> Maybe a
     -> Maybe b
  --                  |          |
  --    ______________           |
  --    |      ___________________
  --    |      |
  (Just g) <*> mx = fmap g mx
  Nothing <*> _ = Nothing


myVar :: Maybe (Int -> Int)
myVar = pure (+)
pure (+) <*> Just 1 <*> Just 2
-- Just 3
```

Listing 3: Maybe as an Applicative example

Monad Monad is a typeclass in Haskell with the definition in listening 4.

```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

  return = pure
```

Listing 4: Monad Class definition

The `>>=` operator takes a container with a value (`Just 2`), unwraps the value (2) and feed into the function, which returns again an container.

```
myAddOne :: Int -> Maybe Int
myAddOne x = Just (x+1)

myValue :: Maybe Int
myValue = Just 2 >>= myAddOne
```

Monad Example The `>>=` operator for `Maybe` could look like in 5.

```
-- (>>=) :: Maybe a -> (a -> Maybe b) ->
     Maybe b
mx >>= f = case mx of
             Nothing -> Nothing
             Just x -> f x
```

Listing 5: The bind operator for Maybe

In 6 the `eval` function is declared 3 times. Every definition does the exact same thing. The latter two variants are using the bind operator (»=) from the Monad type class.

```haskell
data Expr = Val Int | Div Expr Expr

eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
                   Nothing -> Nothing
                   Just n -> case eval y
                        of
                     Nothing -> Nothing
                     Just m -> safediv n
                         m

eval (Div x y) = eval x >>= \n ->
                 eval y >>= \m ->
                 safediv n m

eval (Div x y) = do n <- eval x
                    m <- eval y
                    safediv n m


safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n m = Just (n `div` m)
```

Listing 6: Example of the monad typeclass in action

let expression The `let` expression is just syntactical sugar for immediate lambda evaluation.

```haskell
let name =  value in body
( \name -> body ) value
```

11 Lambda Calculus

Lambda Calculus The lambda calculus is a programming language and the base for all functional programming languages. In the lambda calculus you only have the "λ", ".", "(", and ")" as reserved words. Everything else is achieved using function application.
The lambda calculus has only two concepts: Predicates (P) and $λ$-terms (M).

$$\begin{aligned} P &::= M = M \\ M &::= x \mid \lambda x.\ M \mid MM \end{aligned} \tag{2}$$

Free / Bound Variables In the lambda calculus you can use variables as placeholder. It exists two types of variables: bound and free variables. Bound variables are bound to a lambda term (y in equation (3)). Bound variables are only valid inside the lambda term. They are only placeholders and therefore can be renamed.
The free variables (a, b in (3)) are globaly valid.

$$(\lambda y.\ y) a\ b \tag{3}$$

Alpha Conversion The α conversion is used to rename bound variables (What are free and bound variables in the lambda calculus?). This is sometimes required during application because otherwise the free variable may be captured.

$$\begin{aligned} &\lambda x.\ x\ z \\ &= \lambda a.\ a\ z \quad |\alpha \\ &= \lambda b.\ b\ z \quad |\alpha \\ &\neq \lambda z.\ z\ z \quad |\alpha \end{aligned} \tag{4}$$

Beta Reduction The β reduction is nothing else than one function application. In the equation (5) the "a" is applied to the lambda term. Therefore, the "y" is replaced with "a".

$$\begin{aligned} &\underline{(\lambda y.\ y)\ a}\ b \\ &= a\ b \end{aligned} \tag{5}$$

Normal Form A λ-term is then in the normal form if no further reduction (Beta Reduction) is possible. Equation (6) and (7) are evaluated using two different strategies. Howerver, both are at the end in the normal form.
Every λ-term has at most one normal form. Some λ-terms do not have a normal form (see (8)).

$$\begin{aligned} &(\lambda x.\ square\ x)\ \underline{((\lambda y.quare\ y)5)} \\ &= \underline{(\lambda x.\ square\ x)\ (square\ 5)} \\ &= square\ (square\ 5) \end{aligned} \tag{6}$$

$$\begin{aligned} &\underline{(\lambda x.\ square\ x)\ ((\lambda y.\ square\ y)5)} \\ &= \underline{squre\ ((\lambda y.\ square\ y)5)} \\ &= square\ (square\ 5) \end{aligned} \tag{7}$$

$$\begin{aligned} &\underline{(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)} \\ &= \underline{(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)} \\ &= \underline{(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)} \\ &\vdots \end{aligned} \tag{8}$$

Delta Reduction The δ-reduction replaces the free variable (What are free and bound variables in the lambda calculus?) with its definition.

$$
\begin{aligned}
&square = \lambda x.\ *\ x\ x \\
&\underline{(\lambda x.\ square\ x)\ 5} \\
&= (square\ 5) \quad |\ \beta \\
&= (\underline{(\lambda x.\ *\ x\ x)\ 5}) \quad |\ \delta \\
&= (*\ 5\ 5) \quad |\ \beta
\end{aligned} \tag{9}
$$

Evaluation Strategies
Evaluation strategies describe in which order redexs (reducible expression, any β, δ reducible sub-term) are reduced.
A few of them are:

- leftmost innermost / applicative order / innermost first
- leftmost outermost / normal order / outermost first
- The lazy evaluation / call by need

Leftmost innermost Using this evaluation strategy the innermost redex is reduced first. In case there is more than one innermost redex, the leftmost innermost redex is reduced first. A redex is innermost, if there is no other redex inside it.

Using this strategy the functions arguments are reduced exactly once. It is also called **Call by value**.

$$
\begin{aligned}
&\underline{square}\ (square\ 5) \\
&= (\lambda x.\ *\ x\ x)(\underline{square}\ 5) \\
&= (\lambda x.\ *\ x\ x)(\underline{(\lambda x.\ *\ x\ x)5}) \\
&= \underline{(\lambda x.\ *\ x\ x)(*\ 5\ 5)} \\
&= (*\ (*\ 5\ 5)\ (*\ 5\ 5))
\end{aligned} \tag{10}
$$

Leftmost outermost Using this evaluation strategy the outermost redex is reduced first. In case there is more than one outermost redex, the leftmost outermost redex is reduced first. A redex is outermost, if there is no other redex outside it.

A functions arguments are reduced as often as they are needed. This startegy will always find the normal form if one exits. It is also called call by name.

$$
\begin{aligned}
&\underline{square}\ (square\ 5) \\
&= \underline{(\lambda x.\ *\ x\ x)\ (square\ 5)} \\
&= (*\ (\underline{square}\ 5)\ (square\ 5)) \\
&= (*\ (\underline{(\lambda.x\ *\ x\ x)5))}\ (square\ 5)) \\
&= (*\ (*\ 5\ 5)\ (\underline{square}\ 5)) \\
&= (*\ (*\ 5\ 5)\ (\underline{(\lambda.x\ *\ x\ x)\ 5})) \\
&= (*\ (*\ 5\ 5)\ (*\ 5\ 5))
\end{aligned} \tag{11}
$$

Lazy Evaluation The lazy evaluation is a combination of leftmost innermost and leftmost outermost. It uses the property from leftmost innermost (exactly once) and from leftmost outermost (when needed, finds normal form).
The lazy evaluation performs the leftmost outermost strategy. However, it uses memoization (caching). If an expression was already evaluated it will use the same evaluated result again. Haskell uses this strategy for evaluation.

12 Property Based Testing

13 Reasoning

Vollständige Induktion

1. Verankerung
2. Induktionsschritt
   (a) Induktionsannahme
   (b) Induktionsbehauptung
   (c) Induktionsbeweis

1. Beispiel Beweise dass folgendes gilt:

$$\sum_{k=1}^{n}(k) = \frac{1}{2} \cdot n(n+1)$$

   (a) Verankerung ($n_0 = 1$) Linke Seite $\sum_{k=1}^{1} k = 1$, Rechte Seite $\frac{1}{2} \cdot 1(1+1) = 1$
   (b) Induktionsschirtt $n \Rightarrow n+1$
      i. Induktionsanhame:

$$\sum_{k=1}^{n}(k) = \frac{1}{2} \cdot n(n+1)$$

      ii. Induktionsbehauptung

$$\sum_{k=1}^{n+1}(k) = \frac{1}{2} \cdot (n+1)(n+2)$$

      iii. Induktionsbeweis
         A.

$$\sum_{k=1}^{n+1}(k) = \left(\sum_{k=1}^{n} k\right) + n + 1$$

         B.

$$\sum_{k=1}^{n+1}(k) = \frac{1}{2} \cdot n \cdot (n+1) + n + 1$$

         C.

$$\sum_{k=1}^{n+1}(k) = \frac{1}{2} \cdot (n^2 + n + 2n + 2) = \frac{1}{2} \cdot (n^2 + 3n + 2)$$

         D.

$$\sum_{k=1}^{n+1}(k) = \frac{1}{2} \cdot (n+1) \cdot (n+2)$$