Haskell for Beginners

Stefan Wehr (wehr@cp-med.com)
medilyse GmbH, Freiburg im Breisgau
Universität Freiburg, 2018-07-25

Who am I? What am I doing

- Haskell user for more than 15 years
- Started with Haskell at university
- Since 2010: working in industry, still using Haskell!
- *medilyse research GmbH*, Freiburg im Breisgau
  - Software for hospitals
  - Server-side software 99% in Haskell
  - 7 employees

What to Expect From This Lecture

- Introduction to basic Haskell features
- Not an in-depth coverage of the features
- Practically oriented
- Some details omitted or not covered in full generality

Why functional, why Haskell?

- Control over side-effects

Why functional, why Haskell?

- Control over side-effects
- High degree of modularity, clear interfaces
- Abstraction is easy
- Reusability

Why functional, why Haskell?

- Control over side-effects

- High degree of modularity, clear interfaces
- Abstraction is easy
- Reusability

- Good testability
- Short and readable code
- Statically typed
- Expressive type system, very good support for data modeling

Why functional, why Haskell?

- Control over side-effects

- High degree of modularity, clear interfaces
- Abstraction is easy
- Reusability

- Good testability
- Short and readable code
- Statically typed
- Expressive type system, very good support for data modeling

- Haskell is the "world's finest imperative programming language"
- "Smart people"

Functional vs Imperative: Variables

```haskell
-- Haskell
x :: Int
x = 5
```

- Variable x has value 5 forever

Functional vs Imperative: Variables

```haskell
-- Haskell
x :: Int
x = 5
```

- Variable x has value 5 forever

```java
// Java
int x = 5;
...
x = x+1;
```

- Variable x can change its content over time

Functional vs Imperative: Functions

```
-- Haskell
f :: Int -> Int -> Int
f x y = 2*x + y

f 42 16 // always 100
```

- Return value of a function **only** depends on its inputs

Functional vs Imperative: Functions

```
-- Haskell
f :: Int -> Int -> Int
f x y = 2*x + y

f 42 16 // always 100
```

- Return value of a function **only** depends on its inputs

```
// Java
boolean flag;
static int f (int x, int y) {
  return flag ? 2*x + y , 2*x - y;
}

f (42, 16); // who knows?
```

- Return value depends on non-local variable `flag`

Functional vs Imperative: Laziness

```haskell
-- Haskell
x = expensiveComputation
g anotherExpensiveComputation
```

- The expensive computation will only happen if `x` is ever used.
- Another expensive computation will only happen if `g` uses its argument.

Functional vs Imperative: Laziness

```haskell
-- Haskell
x = expensiveComputation
g anotherExpensiveComputation
```

- The expensive computation will only happen if `x` is ever used.
- Another expensive computation will only happen if `g` uses its argument.

```java
// Java
int x = expensiveComputation;
g (anotherExpensiveComputation)
```

- Both expensive computations will happen anyway.
- Laziness can be simulated, but it's complex!

Haskell Demo

- Let's say we want to buy a game in the USA and we have to convert its price from USD to EUR
- A *definition* gives a name to a value
- Names are case-sensitive, must start with lowercase letter
- Definitions are put in a text file ending in `.hs`

```haskell
-- Example.hs
dollarRate = 1.17047
```

ghci: Haskell interpreter

```
$ stack ghci Examples.hs
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /Users/swehr/.ghci
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> dollarRate
1.17047
*Main> 53 * dollarRate
62.034909999999996
*Main>
```

A function to convert EUR to USD

```haskell
dollarRate = 1.17047

-- |convert EUR to USD
toUsd euros = euros * dollarRate
```

- line starting with `--`: comment
- `toUsd`: function name (defined)
- `euros`: argument name (defined)
- `euros * dollarRate`: expression to compute the result

Using the function

```
*Main> :r
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> toUsd 1
1.17047
*Main> toUsd 53
62.034909999999996
*Main>
```

- `:r` reload file in ghci

Exercise

Now it's your turn: write function `toEuro`

```
*Main> toEuro 1
0.8543576511999454
*Main> toEuro (toUsd 53)
53.0
```

Quick detour: testing

```haskell
prop_euroUsd x = toEuro (toUsd x) == x

-- == is the equality operator
```

```
*Main> prop_euroUsd 1
True
*Main> prop_euroUsd 10
True
```

QuickCheck: automatic and random testing

- Does `prop_euroUsd` hold in general?
- Use QuickCheck to find out!
  - QuickCheck generates random testcases
  - QuickCheck checks your properties for the testcases generated
  - QuickCheck tries to find a minimal counterexample

```
$ stack install QuickCheck
$ stack ghci Examples.hs
*Main> import Test.QuickCheck
*Main Test.QuickCheck> quickCheck prop_euroUsd
*** Failed! Falsifiable (after 5 tests and 8 shrinks):
15.0
*Main Test.QuickCheck> prop_euroUsd 15
False
```

The Problem: Floating Point Arithmetic

```
*Main Test.QuickCheck> toEuro (toUsd 15)
14.999999999999998
```

- A better property:

```
prop_euroUsd' x =
    abs (toEuro (toUsd x) - x) < 10e-10
```

```
*Main Test.QuickCheck> quickCheck prop_euroUsd'
+++ OK, passed 100 tests.
```

Function definition by cases

- Example: absolute value

```
absolute x | x >= 0 = x
absolute x | x < 0  = -x

absolute' x | x >= 0     = x
            | otherwise  = -x

absolute'' x = if x >= 0 then x else -x
```

Recursion

Standard approach to define functions in functional languages *(no loops!)*

- Reduce a problem (e.g., `power x n`) to a smaller problem of the same kind
- Eventually reach a "smallest" base case
- Solve base case separately
- Build up solutions from smaller solutions

Example: power

Compute `x^n` without using the built-in operator

```haskell
-- compute x to n-th power
power :: Int -> Int -> Int
power x 0          = 1
power x n | n > 0 = x * power x (n - 1)
```

```
*Main> power 1 0
1
*Main> power 2 4
16
*Main> power 2 (-1)
*** Exception: code/Examples.hs:(24,1)-(25,39): Non-exhaustive patterns
```

- `power` is a partial function: it crashes for negative numbers

Function Types

- `Bool -> Int`
  - takes a `Bool` as argument
  - returns an `Int`
- `Int -> Int -> Int` is short for `Int -> (Int -> Int)`
  - takes an `Int`
  - returns another function `Int -> Int`
- `(Bool -> Int) -> Int`
  - takes a function `Bool -> Int`
  - returns an `Int`
  - such functions are called *higher-order functions*

Examples for Function Types

```haskell
boolToInt :: Bool -> Int
boolToInt True = 1
boolToInt False = 0

sumTheBools :: (Bool -> Int) -> Int
sumTheBools fun = fun True + fun False
```

```
*Main> sumTheBools boolToInt
1
*Main> sumTheBools (\b -> if b then 41 else 1)
42
```

- `(\b -> if b then 41 else 1)` is a lambda-expression / anonymous functions

Haskell and Types

- Haskell is a statically typed language.
- Prevents errors like this:

```
$ python
Python 2.7.10 (default, Oct  6 2017, 22:29:07)
>>> "STEFAN".toLowerCase()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'toLowerCase'
```

```
-- Javascript
> x.fun()
=> undefined is not a function
```

- Types are inferred automatically, no need to write them.
- But it is good practice to write a type signature for every top-level function.

Predefined types

- `Bool` (`True`, `False`)
- `Char` (`'x'`, `'?'`, `'x'`, `...`)
- `Double`, double precision floating points
- `Float`, single precision floating points
- `Integer`, arbitrary-precision integers
- `Int`, machine integers (at least 30 bits signed integer, but typically 64 bits)
- `()`, the unit type, single value `()`
- Function types
- Tuples
- Lists
- `String` (`"xyz"`, `"Stefan"`, `...`): just a list of `Char`, better use `Data.Text.Text` for production
- `Maybe` for optional values
- Names of concrete types start with uppercase letters.

Tuples

```haskell
-- example tuples
examplePair :: (Double, Bool)  -- Double x Bool
examplePair = (3.14, False)

exampleTriple :: (Bool, Int, String) -- Bool x Int x String
exampleTriple = (False, 42, "Answer")

exampleFunction :: (Bool, Int, String) -> Bool
exampleFunction (b, i, s) = not b && length s < i
```

- Syntax for tuple type like syntax for tuple values.
- Tuples are *immutable* once a tuple value is defined it cannot change!

Lists

- The "duct tape" of functional programming
- Collections of things of the same type
- For any type `a`, the type `[a]` is the type of lists of `a`s
  - e.g. `[Bool]` is the type of lists of `Bool`
- Syntax for list type like syntax for list values
- Lists are *immutable* : once a list value is defined it cannot change!

Constructing Lists

Tye values of type `[a]` are ...

- either `[]`, the empty list
- or `x:xs` where `x` has type `a` and `xs` has type `[a]`. The `:` operator is pronounced "cons"
- `(:) :: a -> [a] -> [a]`

Quiz: which of the following values have type `[Bool]`?

```
[]
True : []
True:False
False:(False:[])
(False:False):[]
(False:[]):[]
(True : (False : (True : []))) : (False:[]):[]
```

List Shorthands

- `1:(2:(3:[]))` – standard, fully parenthesized
- `1:2:3:[]` – (:) associates to the right
- `[1,2,3]`

Functions on Lists

- Definition by *pattern matching*

```haskell
-- function over lists - examples
summerize :: [String] -> String
summerize []  = "None"
summerize [x] = "Only " ++ x
summerize [x,y] = "Two things: " ++ x ++ " and " ++ y
summerize [_,_,_] = "Three things: ???"
summerize _   = "Several things."   -- wild card pattern
```

- `++`: list concatenation
  - Associates to the right because it's more efficient
  - `[1,2,3,4,5] ++ ([6,7,8,9] ++ [])` — 10 copy operations
  - `([1,2,3,4,5] ++ [6,7,8,9]) ++ []` — 14 copy operations, because `[1,2,3,4,5]` is copied twice

Pattern Matching on Lists

- patterns are checked in sequence
- variables in patterns are bound to the values in corresponding position in the argument
- each variable may occur at most once in a pattern
- wild card pattern _ matches everything, no binding, may occur multiple times

Maybe

- Haskell's safe alternative to null references
- Null references: the billion dollar mistake (Tony Hoare)

```
data Maybe a = Nothing | Just a
```

- `Maybe` expresses optional values. For example, as the result of a lookup:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

- `Eq a` is a *type class constraint*. More on this topic later.
- `a` and `b` are *type variables*. Type variables are instantiated with concrete types.
  - E.g. `lookup "key" assocs` where `assocs` has type `[(String, Int)]` yields a value of type `Int`
  - `a` is instantiated with `String`, `b` with `Int`
- Names of type variables start with a lowercase letter

Recursion on Lists

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter pred (x:xs)
    | pred x    = x : filter pred xs
    | otherwise = filter pred xs
```

```
*Main> filter (\x -> x > 5) [1,2,3,4,5,6,7,8]
[6,7,8]
*Main> filter (\x -> False) [1,2,3,4,5,6,7,8]
[]
```

More Recursion on Lists

```haskell
lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup _ [] = Nothing
lookup k ((x, y) : rest)
    | k == x    = Just y
    | otherwise = lookup k rest
```

```
*Main> lookup "one" [("one", 1), ("two", 2)]
Just 1
*Main> lookup "three" [("one", 1), ("two", 2)]
Nothing
```

Library Functions on Lists

```haskell
map :: (a -> b) -> [a] -> [b]
-- map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]

foldl :: (a -> b -> a) -> a -> [b] -> a
-- foldl f z [x1, x2, ..., xn] ==
   (...((z `f` x1) `f` x2) `f`...) `f` xn

foldr :: (a -> b -> b) -> b -> [a] -> b
-- foldr f z [x1, x2, ..., xn] ==
   x1 `f` (x2 `f` ... (xn `f` z)...)
```

User-defined Data Types

- We can define our own data types in Haskell
- The syntax is concise and expressive
- Haskell is a very good tool for data modelling!

A Definition for Shapes

- A circle is defined by its radius

```
data Circle = Circle Double
```

- We support three different colors: red, green, and blue

```
data Color = Red | Green | Blue
```

- A rectangle is defined by its width and height
- `r_width` and `r_height` are *record labels*. The prefix `r_` is not required but often used because record labels have module scope.

```
data Rectangle = Rectangle { r_width :: Double, r_height :: Double }
```

Putting It All together

```haskell
data Shape
    = ShapeRectangle Rectangle
    | ShapeCircle Circle
    | ShapeAbove Shape Shape
    | ShapeBeside Shape Shape
    | ColoredShape Color Shape
```

- Shape is a *recursive datatype* because it appears on the left- and right-hand side of the definition

```haskell
sampleShape :: Shape
sampleShape =
    ShapeAbove
        (ShapeCircle (Circle 10))
        (ColoredShape Red (ShapeRectangle
            (Rectangle { r_width = 10, r_height = 20 })))
```

Functions on Data Types

Here's a function for counting the number of circles in a shape.

```
countCircles :: Shape -> Int
countCircles shape =
    case shape of
      ShapeRectangle _ -> 0
      ShapeCircle _ -> 1
      ShapeAbove top bottom ->
          countCircles top + countCircles bottom
      ShapeBeside left right ->
          countCircles left + countCircles right
      ColoredShape _ shape ->
          countCircles shape
```

- The `case ... of ...` expression is another form for writing pattern matching.
- Other pattern matching forms can be rewritten as case-expressions.

Exercise

Now it's your turn:

- Write a function collecting all colors of a shape:
  `collectColors :: Shape -> [Color]`
- Write a function to compute the bounding box of a shape:
  `boundingBox :: Shape -> Rectangle`

IO and Referential Transparency and Substitutivity

- Every variable and expression has just one value: *referential transparency*
- Every variable can be replaced by its definition: *substitutivity*
- Enables reasoning

```
-- sequence of function calls does not matter
f () + g () == g () + f ()

-- number of function calls does not matter
f () + f ( ) == 2 * f ()
```

How does IO fit in?

- Bad example: Suppose we had `input :: () -> Integer`
- Consider `let x = input () in x + x`
  - Expect to read one input and use it twice
  - By substitutivity, this expression must behave like `input () + input ()` which reads two inputs!
  - VERY WRONG!!!

The dilemma

- Haskell is a pure language, but IO is a side effect
- A contradiction?
- No!
  - Instead of performing IO operations directly, there is an abstract type of *IO actions*
  - Some actions (e.g., read from a file) return values, so the abstract IO type is parameterized over their type
  - Keep in mind:
    - actions are just values like any other
    - they can be passed around, stored in collections, executed conditionally, ...
- Haskell is the "world's finest imperative programming language"

Haskell IO

- Top-level result of a program is an IO “action”.

```haskell
main :: IO ()
main = putStrLn "Hello World!"
```

- an action describes the *effect* of the program
- effect = IO action, imperative state change, ...

Basic IO Operations

```haskell
putStr :: String -> IO ()      -- writes to stdout, without \n
putStrLn :: String -> IO ()    -- writes to stdout, with trailing \n
getLine :: IO String           -- reads from stdin

readFile :: FilePath -> IO String             -- reads contents of file
writeFile :: FilePath -> String -> IO ()      -- writes string to file

getArgs :: IO String     -- Zugriff auf Kommandozeilenargumente
```

The do-Notation

- IO-operations are sequenced via do
- Use <- to bind result values to names

```
printName :: IO ()
printName =
    do putStr "What's your name? "
       s <- getLine
       putStrLn ("Ah, your name is " ++ s)
```

```
*Main> printName
What's your name? Stefan
Ah, your name is Stefan
```

IO Actions Are Just Values

You can easily abstract over IO actions.

```
sequence_ :: [IO ()] -> IO ()
sequence_ []     = return ()
sequence_ (a:as) =
    do a
       sequence_ as
```

```
*Main> sequence_ [putStr "Hello", putStr " World\n"]
Hello World
```

- `return` has type `a -> IO a` (more general in reality)

Example: `if` With Condition Living in IO

```haskell
ioIf :: IO Bool -> IO a -> IO a -> IO a
ioIf condAction ifAction thenAction =
    do cond <- condAction
       if cond then ifAction else thenAction
```

Example: `if` With Condition Living in IO

```haskell
ioIf :: IO Bool -> IO a -> IO a -> IO a
ioIf condAction ifAction thenAction =
    do cond <- condAction
       if cond then ifAction else thenAction
```

```
*Main> :{
*Main| let condAction =
*Main|         do x <- getLine
*Main|            return (x == "yes")
*Main| :}
*Main> ioIf condAction (putStrLn "Got a yes") (putStrLn "No yes")
yes
Got a yes
```

A Short Look at Concurrent Programming With Haskell

- Threads are extremely cheap
  - `forkIO :: IO () -> IO ThreadId`
  - Green threads, not OS-level threads
  - Millions of green threads can run on this laptop in a single Haskell program
- Classic means for synchronization: semaphore, locks (not covered in this lecture)
- Modern synchronization technique: Software transactional Memory (STM, not covered in this lecture)
- Fast and convenient
  - Under the hood, most system-level IO calls are async.
  - But the programmer uses blocking calls and mutliple threads.
  - Much more convenient then event-driven programming

Forking a Thread

```haskell
threadExample :: IO ()
threadExample =
    do forkIO (doWork "x" 10)
       forkIO (doWork "y" 10)
       return ()
    where
      doWork x i =
          do putStr x
             threadDelay 100000
             if i < 1 then return () else doWork x (i - 1)
```

Forking a Thread

```haskell
threadExample :: IO ()
threadExample =
    do forkIO (doWork "x" 10)
       forkIO (doWork "y" 10)
       return ()
    where
      doWork x i =
          do putStr x
             threadDelay 100000
             if i < 1 then return () else doWork x (i - 1)
```

```
*Main> threadExample
xy*Main> yxxyxyyxxyyxyxxyyxyx
```

Overloading

So far:

- Equality (==) and comparison (<) for different types
- Arithmetic operations for different types

Overloading: The *same operator* can be used to execute *different code* at *many different types*.

Ad-hoc or Restricted Polymorphism

- Some functions work on parametric types, but are restricted to specific instances
- Types contain type variables and *constraints*

Examples:

```
-- lookup requires an equality comparison
lookup :: Eq a => a -> [(a, b)] -> Maybe b

-- The absolute value can be computed for anything that supports
-- numeric operations and comparison
*Main> :t absolute
absolute :: (Ord a, Num a) => a -> a
```

- `Eq a` and `Ord a` and `Num a` are *constraints*.
- The type variable a can be instantiated with any type that implements the type class `Eq` and `Ord` and `Num`.

Type Classes

- Each constraint mentions a *type class* like `Eq`, `Ord`, `Num`, ...
- A type class specifies a set of operations for a type e.g. `Eq` requires == and /=
- Type classes form a hierarchy e.g. `Ord a => Eq a`, that is `Eq` is a superclass of `Ord`
- Many classes are predefined, but you can roll your own

Classes and Instances

- A class declaration *only* specifies a signature (i.e., the class members and their types)

```
class Num a where
  (+), (*), (-) :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger :: Integer -> a
```

- An instance declaration specifies that a type belongs to a class by giving definitions for all class members

```
instance Num Int where ...
instance Num Integer where ...
instance Num Double where ...
instance Num Float where ...
```

- This info can be obtained from GHCI by

```
:i Num
```

Equality

```haskell
-- predefined type class Eq
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x /= y = not (x == y) -- default definition
```

An instance only has to provide (==).

```haskell
instance Eq Rectangle where
    r1 == r2 =
        r_width r1 == r_width r2 &&
        r_height r1 == r_height r2
```

Important Type Classes

```haskell
-- Show is used to convert a value into a string
class Show a where
    show :: a -> String

class Eq a where ...

class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min :: a -> a -> a
    compare :: a -> a -> Ordering
data Ordering = LT | EQ | GT

class Num a where ...
```

- Other standard type classes (investigate on your own): Bounded and Enum

Deriving Type Classes

- The compiler can automically derive instance of standard type classes

```
data Rectangle = Rectangle { r_width :: Double, r_height :: Double }
    deriving (Show, Eq, Ord)
```

```
*Main> show (Rectangle 10 20)
"Rectangle {r_width = 10.0, r_height = 20.0}"
```

Monads and Other Advanced features

- There is another important type class: `Monad`
- `IO` is an instance of `Monad`
- All monads support the do-notation
- We won't cover monads in this lecture (for lack of time)
- Type classes allow for many powerful features such as generic programming or type-level programming. We won't cover these topics in this lecture.

Sample Project

- Meme generator: https://github.com/agrafix/meme-tutorial

libgd for Graphics Manipulations

```haskell
loadJpegByteString :: ByteString -> IO Image
saveJpegByteString :: Int -> Image -> IO ByteString
useFontConfig :: Bool -> IO Bool
measureString ::
  String -> Double -> Double -> Point ->
  String-> Color -> IO (Point, Point, Point, Point)
drawString ::
  String -> Double -> Double -> Point ->
  String -> Color -> Image -> IO (Point, Point, Point, Point)
```

Getting Started

- Haskell API search
  - http://hayoo.fh-wedel.de/
  - https://www.haskell.org/hoogle/
- `stack`
  - Installs GHC and the required packages
  - Builds the project
  - `stack.yaml` describes which distribution of packages to use
  - http://haskellstack.org
- `meme-tutorial.cabal`: project definition
  - which libraries, which executables are provided
  - which packages are required
- `app/Main.hs` defines the `main` function
- `src/MemeGen.hs` defines a stub for the `createMeme` function
- One-time setup:
  - Install `libgd`
  - Then: `stack setup`
- Compiling: `stack build`
- Running: `stack exec memegen`

Resources

- Paper by the original developers of Haskell in the conference on History of Programming Languages (HOPL III): http://dl.acm.org/citation.cfm?id=1238856
- The Haskell home page: http://www.haskell.org
- Haskell libraries repository: https://hackage.haskell.org/
- Haskell Tool Stack: https://docs.haskellstack.org/en/stable/README/
- School of Haskell: https://www.fpcomplete.com/
- Real World Haskell, *Bryan O'Sullivan*, *Don Stewart* und *John Goerzen*, O'Reilly 2008
  - http://book.realworldhaskell.org/
- Learn you a Haskell for a great good: http://learnyouahaskell.com/