

# Haskell

A functional paradigm exemplar and other aspects (cont'd)

L. Thomas van Binsbergen

UvA

March 2, 2023

# Types

## What we have seen about types so far

- Haskell is strong, statically typed
- Concrete types: `Int`, `Char`, `Bool`, `[Integer]`, `(Double, Float)`, ...
- Type variables: `a`, `b`, ... (lower case names, often single characters)
- Function types: `Integer → Integer`, `a → b → b`, `(a → b) → b`, `a → Bool`, ...

## Today

- Type aliases (giving shorthand names to types)
- Datatype definitions (introducing your own types of values)
- Application of type constructors (building types by composing types)  
for example, `[a]` is the application of type constructor `[ ]` to type variable `a`
- Type-inferencing (Haskell's advanced form of type-checking) and polymorphism
- Type classes & advanced type classes (`Functor`, `Monad`)

# ⚙️ Type system

- **Type checker:**

- ▶ static → type-checking without program execution, typically at compile-time  
↳ no runtime type errors (types of branches must be 'unified')
- ▶ strong → strict separation between types

- **Type inference:** algorithmic reasoning about types of variables and parameters

```
Prelude> [1,'a']
```

```
<interactive>:62:2: error:
```

- No instance for (Num Char) arising from the literal '1'
- In the expression: 1  
In the expression: [1, 'a']  
In an equation for 'it': it = [1, 'a']

- **Type classes:** introduce overloading as a principle → polymorphism <sup>1</sup>

- ▶ class definition

```
class (Eq a) => Ord a where  
  (<), (<=), (>=), (>)  :: a -> a -> Bool  
  max, min             :: a -> a -> a
```

- ▶ class constraints:

```
class (Eq a, Show a) => ShowOrd a where ...
```

---

<sup>1</sup>[https://www.researchgate.net/publication/2710954\\_How\\_to\\_Make\\_Ad-Hoc\\_Polymorphism\\_Less\\_Ad-Hoc](https://www.researchgate.net/publication/2710954_How_to_Make_Ad-Hoc_Polymorphism_Less_Ad-Hoc)

## Type signatures

When defining a function, we can also give its type-signature:

Lecture.hs

```
fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n-1)
```

or using an unknown type:

Lecture.hs

```
third :: (a,b,c) -> c
third (_,_,z) = z
```

## Type signatures

When defining a function, we can also give its type-signature:


Lecture.hs

```
fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n-1)
```

or using an unknown type:

Lecture.hs

```
third :: (a,b,c) -> c
third (_,_,z) = z
```

- Type signatures for functions are optional ( but recommended), as the Haskell compiler will infer the types of functions and variables from their usage (type inferencing)

## Type aliases

```
-- a type alias is an alternative (simple) name for a (complex) type
type Row = Int
type Column = Int
type Value = Int
type Grid = [[Value]]           -- hides certain details
type Sudoku = (Row, Column) -> Value -- function type example

-- Function to extend a sudoku with a value at (row, column).
-- 1 + 3 + 2 arguments??
extend :: Sudoku -> (Row, Column, Value) -> Sudoku
extend sud (r, c, v) (i, j)
  | r == i && c == j = v
  | otherwise       = sud (i, j)
```

# Datatype definitions

```
data Direction = North | East | South | West
```

```
type Directions = [Direction]
```

```
rotate :: Direction -> Direction
```

```
rotate North = East
```

```
rotate East  = South
```

```
rotate South = West
```

```
rotate West  = North
```

```
rotate' dir = case dir of
```

```
    North -> East
```

```
    East  -> South
```

```
    South -> West
```

```
    West  -> North
```

# Datatype definitions – with constructor functions

```
data BinIntTree = Leaf Int | Node BinIntTree BinIntTree
```

```
sum :: BinIntTree -> Int
```

```
sum tree = case tree of
```

```
    Leaf i      -> i
```

```
    Node t1 t2 -> sum t1 + sum t2
```

```
depth :: BinIntTree -> Int
```

```
depth tree = case tree of
```

```
    Node t1 t2 -> ...
```

```
    Leaf i      -> ...
```



## Datatype definitions – with constructor functions

```
data BinIntTree = Leaf Int | Node BinIntTree BinIntTree
```

```
sum :: BinIntTree -> Int
```

```
sum tree = case tree of
```

```
  Leaf i      -> i
```

```
  Node t1 t2 -> sum t1 + sum t2
```

```
depth :: BinIntTree -> Int
```

```
depth tree = case tree of
```

```
  Node t1 t2 -> 1 + max (depth t1) (depth t2)
```

```
  Leaf _     -> 1
```

## Datatype definitions – with type arguments

```
data BinIntTree = Leaf Int | Node BinIntTree BinIntTree
data BinTree a  = Leaf a   | Node (BinTree a) (BinTree a)
```

```
-- depth :: BinIntTree -> Int
depth :: BinTree a -> Int
depth tree = case tree of
  Leaf _      -> 1
  Node t1 t2 -> 1 + max (depth t1) (depth t2)
```

Note the application of `BinTree` to type variable `a`. `BinTree` is a *type constructor*

## Datatype definitions – with type arguments

```
sum :: BinIntTree -> Int
```

How to update the old signature of sum to the more general type BinTree?

## Datatype definitions – with type arguments

```
sum :: BinIntTree -> Int
```

How to update the old signature of `sum` to the more general type `BinTree`?

```
sum :: BinTree Int -> Int
```

or

```
type BinIntTree = BinTree Int -- replace original definition with alias
```

## Datatype definitions – with type arguments

```
sum :: BinIntTree -> Int
```

How to update the old signature of sum to the more general type BinTree?

```
sum :: BinTree Int -> Int
```

or

```
type BinIntTree = BinTree Int -- replace original definition with alias
```

recall

```
data BinIntTree = Leaf Int | Node BinIntTree BinIntTree
data BinTree a  = Leaf a   | Node (BinTree a) (BinTree a)
```

## Type inferencing

### Direction.hs

```
data Direction = North | East | South | West
-- or returns whether any Boolean in the given list is True
findNorth dirs = or (map isNorth dirs)
  where isNorth dir = case dir of North -> True
                                _      -> False
```

- What is the type of *dir*?

## Type inferencing

### Direction.hs

```
data Direction = North | East | South | West
-- or returns whether any Boolean in the given list is True
findNorth dirs = or (map isNorth dirs)
  where isNorth dir = case dir of North -> True
                                _      -> False
```

- What is the type of *dir*?
- What is the type of *isNorth*?

## Type inferencing

### Direction.hs

```
data Direction = North | East | South | West
-- or returns whether any Boolean in the given list is True
findNorth dirs = or (map isNorth dirs)
  where isNorth dir = case dir of North -> True
                                _      -> False
```

- What is the type of *dir*?
- What is the type of *isNorth*?
- What is the type of *dirs*?



## Type inferencing

### Direction.hs

```
data Direction = North | East | South | West
-- or returns whether any Boolean in the given list is True
findNorth dirs = or (map isNorth dirs)
  where isNorth dir = case dir of North -> True
                                _      -> False
```

- What is the type of *dir*?
- What is the type of *isNorth*?
- What is the type of *dirs*?
- What is the type of *or*?

## Type inferencing

### Direction.hs

```
data Direction = North | East | South | West
-- or returns whether any Boolean in the given list is True
findNorth dirs = or (map isNorth dirs)
  where isNorth dir = case dir of North -> True
                                _      -> False
```

- What is the type of *dir*?
- What is the type of *isNorth*?
- What is the type of *dirs*?
- What is the type of *or*?
- What is the type of *findNorth*?

# Type inferencing

Alternative definitions:

```
findNorth dirs = any isNorth dirs
  where isNorth ...
findNorth = any isNorth
  where isNorth ...
```

# Type inferencing

Alternative definitions:

```
findNorth dirs = any isNorth dirs
  where isNorth ...
findNorth = any isNorth
  where isNorth ...
```

Shortest solution:

Direction.hs

```
data Direction = North | East | South | West
findNorth :: [Direction] -> Bool
findNorth = any (==North)
```

# Type inferencing

Alternative definitions:

```
findNorth dirs = any isNorth dirs
  where isNorth ...
findNorth = any isNorth
  where isNorth ...
```

Shortest solution:

## Direction.hs

```
data Direction = North | East | South | West
findNorth :: [Direction] -> Bool
findNorth = any (==North)
```

Direction.hs:2:18: error:

- No instance for (Eq Direction) arising from a use of ‘==’
- In the first argument of ‘any’, namely ‘(== North)’  
In the expression: any (== North)

## Typeclasses

Types can be part (members) of typeclasses<sup>2</sup>:

```
Prelude> :t elem
```

```
elem :: Eq a => a -> [a] -> Bool    -- a type constraint on type variable 'a'
```

- Eq: the type supports == and /=
- Ord: the type is ordered, supports > and <= etc.
- Show: values of the type can be represented as strings, supports show
- Read: values of the type can be read from strings, supports read
- Enum: the values of the type can be enumerated, supports succ, pred and ranges
- Num: numerical types, supports + and \* etc.
- Integral: type is an integer number (Int/Integer)
- Floating: type is a floating point number (Float/Double)

---

<sup>2</sup><https://www.haskell.org/tutorial/classes.html>

# Typeclasses – Eq

## Direction.hs

```
data Direction = North | East | South | West
```

```
findNorth :: [Direction] -> Bool
```

```
findNorth = any (==North)
```

```
instance Eq Direction where
```

```
  d1 == d2 = case (d1, d2) of
```

```
    (North, North) -> True
```

```
    (East, East)   -> True
```

```
    (South, South) -> True
```

```
    (West, West)   -> True
```

```
    _              -> False
```

# Typeclasses – Eq

## Direction.hs

```
data Direction = North | East | South | West
findNorth :: [Direction] -> Bool
findNorth = any (==North)

instance Eq Direction where
  d1 == d2 = case (d1, d2) of
    (North, North) -> True
    (East, East)   -> True
    (South, South) -> True
    (West, West)   -> True
    _              -> False
```

Definition of equality can be *derived*:

```
data Direction = North | East | South | West deriving Eq
```



## Typeclasses – Show

```
data Direction = North | East | South | West deriving Eq
```

```
*Main> rotate North
```

- No instance for (Show Direction) arising from a use of ‘print’

This requires Direction to be an instance of Show, which can also be derived:

```
data Direction = North | East | South | West deriving (Show, Eq)
```

```
*Main> rotate North
```

```
East
```

# Polymorphism

A function is called *polymorph* if it can be applied to values of varying types

Parameteric polymorphism – type of parameter can vary

```
depth :: BinTree a -> Int
depth tree = case tree of
  Leaf _      -> 1
  Node t1 t2 -> 1 + max (depth t1) (depth t2)
```

Ad-hoc polymorphism – type of parameter supports overloaded operation, here +

```
sum :: Num a => BinTree a -> a
sum tree = case tree of
  Leaf n      -> n
  Node t1 t2 -> sum t1 + sum t2
```

## ✂ Input/Output

The Haskell function definitions we have seen so far are all of *pure functions*: functions whose behaviour is only determined by their parameters (and not by mutable variables, or files, or databases, etc.) and whose only effect is the result value

To interact with the machine, for example printing text and reading files, we need to use IO: IO a is an effectful input/output computation, producing a result of type a

```
-- write a string to the standard output device, followed by a newline
putStrLn :: String -> IO ()
-- reads a file and returns the contents of the file as a String
readFile :: FilePath -> IO String
-- returns a list of the program's command line arguments
getArgs  :: IO [String]
```

## ✂ Input/Output

```
-- write a string to the standard output device, followed by a newline
putStrLn :: String -> IO ()
-- reads a file and returns the contents of the file as a String
readFile :: FilePath -> IO String
-- returns a list of the program's command line arguments
getArgs  :: IO [String]
```

### FileTest.hs

```
import System.Environment -- required for getArgs
main :: IO()
main = do
    args <- getArgs
    if null args then putStrLn "please provide command line arguments"
    else do text <- readFile (head args)
            putStrLn text
```

## ✂ Compiling and running FileTest.hs

```
$ ghc FileTest.hs -o filetest
[1 of 1] Compiling Main                ( FileTest.hs, FileTest.o )
Linking filetest ...
```

```
$ ./filetest
please provide command line arguments
```

```
$ ./filetest /tmp/valentine_letter.txt
Dear Valentine,
```

```
With this letter I would like to inform you that ..
```

```
All the best,
  Yours truly
```

## ✂ do-notation

The **do**-notation lets you write ‘instruction-like’ expressions:

- what you can write in `ghci` you can write in a `do`-block of type `IO a` (and vice versa)
- available for expressions/functions of the type `IO a` (for any `a`)
- or, actually, for any ‘monad<sup>3</sup>’: `Monad m => m a`
- is syntactic sugar: implicit application of either `>>` or `>>=` between every “instruction”

```
main :: IO () -- the statements of the do block have equal indentation
main = do
  forM_ [1..10] $ \x -> do -- discouraged, use: mapM_ print [1..10]
    print x

  x <- return 1           -- highly discouraged, use: let x = 1
  x <- return (x + 1)      -- , and: let y = x + 1

  return ()
```

---

<sup>3</sup><https://wiki.haskell.org/Monad>

## ✂ When (not) to use IO?

- A module with the function `main :: IO ()` is a 'Main' module
- Best practice:
  - ▶ only use IO in your Main modules
  - ▶ and only for the main function itself + a few helpers
  - ▶ the 'logic' of your program should be expressed fully within pure functions
- There are different perspectives on whether non-IO Monads are considered pure
- The function `unsafePerformIO :: IO a -> a` can be used 'run' an IO computation
- Highly discouraged! You should only allow the OS to run the IO computation `main`
- When to use **do**?

# Functors

The members of the type-class Functor are ‘containers’ that can be ‘mapped’.

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

Does this seem familiar?



# Functors

The members of the type-class Functor are 'containers' that can be 'mapped'.

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

Does this seem familiar?

'fmap' generalises 'map'

```
instance Functor [] where
    fmap f []      = []
    fmap f (a:as) = f a : fmap f as
```

# Functors

Our binary tree type can also be made an instance of Functor

```
data BinTree a = Leaf a | Node (BinTree a) (BinTree a)
```

```
instance Functor BinTree where
```

```
    fmap f (Leaf a)      = Leaf (f a)
```

```
    fmap f (Node t1 t2) = Node (fmap f t1) (fmap f t2)
```

```
*Main> fmap (+1) (Node (Leaf 1) (Leaf 4))
```

```
Node (Leaf 2) (Leaf 5)
```

# Functors

Nog een voorbeeld, van 'rose trees'

```
data Rose a = Node a [Rose a] deriving (Show)
```

```
instance Functor Rose where
```

```
    fmap f (Node a ts) = Node (f a) (fmap (fmap f) ts)
```

```
*Main> fmap (\x -> x * x) my_rose
```

```
Node 1 [Node 4 [],Node 9 [Node 16 []],Node 25 []]
```

Merk op dat `fmap` meerdere keren en met een verschillende definitie wordt toegepast.  
Welke aanroep is recursief?

# Applicative functors

Applicative functors are functors that contain the results of computations

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

Vergelijk met (\$) :: (a -> b) -> a -> b

```
instance Applicative [] where
    pure a  = [a]
    p <*> q = [ f a | f <- p, a <- q]
```

```
Prelude> let my_functions = [(+1), (*2), (`div` 3)]
```

```
Prelude> my_functions <*> [1..10]
```

```
[2,3,4,5,6,7,8,9,10,11,2,4,6,8,10,12,14,16,18,20,0,0,1,1,1,2,2,2,3,3]
```

# Applicative functors

Een expressie van type `Maybe a` is een berekening die kan 'falen':

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

```
instance Applicative Maybe where
```

```
    pure      = Just
```

```
    p <*> q = case p of Nothing -> Nothing
                  Just f  -> case q of
                      Nothing -> Nothing
                      Just a  -> Just (f a)
```

Bijvoorbeeld een (niet langer) oplosbare Sudoku,  
vanwege een constraint `(r, c, [])` met geen mogelijkheden

# Safe head

```
safeHead :: [a] -> Maybe a  
safeHead []      = Nothing  
safeHead (a:_)  = Just a
```

# Safe head

```
safeHead :: [a] -> Maybe a
safeHead []      = Nothing
safeHead (a:_)  = Just a
```

From any Functor instance, the operator `<$>` is derived (infix version of `fmap`)

```
*Main> (+) <$> safeHead [] <*> safeHead []
Nothing
*Main> (+) <$> safeHead [1..5] <*> safeHead []
Nothing
*Main> (+) <$> safeHead [] <*> safeHead [1..5]
Nothing
*Main> (+) <$> safeHead [1..5] <*> safeHead [5..10]
Just 6
```

# Monads

Monads are applicative functors that contain the results of computations with effects. That is, computations can be placed in sequence and can affect each other.

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

instance Monad [] where
    return a  = [a]
    ma >>= f  = [ b | a <- ma, b <- f a ]
    -- p <*> q = [ f a | f <- p, a <- q]    {for comparison}

instance Monad Maybe where
    return = Just
    ma >>= f = case ma of Just a  -> f a
                      Nothing -> Nothing
```



# Monads

Monads are applicative functors that contain the results of computations with effects. That is, computations can be placed in sequence and can affect each other.

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

instance Monad [] where
    return a  = [a]
    ma >>= f  = [ b | a <- ma, b <- f a ]
    -- p <*> q = [ f a | f <- p, a <- q]    {for comparison}

instance Monad Maybe where
    return = Just
    ma >>= f = case ma of Just a  -> f a
                      Nothing -> Nothing
```

For a gentle introduction to Functors, Applicatives and Monads go to [http://adit.io/posts/2013-04-17-functors,\\_applicatives,\\_and\\_monads\\_in\\_pictures.html](http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html)

# MaybeHeadEven

```
maybeEven :: Int -> Maybe Int
maybeEven i | even i      = Just i
              | otherwise = Nothing
```

```
maybeHeadEven :: [Int] -> Maybe Int
maybeHeadEven xs = safeHead xs >>= maybeEven
```

```
*Main> maybeHeadEven [1..5]
```

```
Nothing
```

```
*Main> maybeHeadEven [2..5]
```

```
Just 2
```

```
*Main> maybeHeadEven [3..5]
```

```
Nothing
```

# The State Monad

## Definition (in Prelude)

```
data State s a = State (s -> (s,a)) -- s is the type of state, a of result
runState (State comp) = comp
```

```
instance Monad (State s) where
  -- return :: a -> State s a
  return x = State $ \s -> (s, x)
  -- (>>=) :: State s a -> (a -> State s b) -> State s b
  m >>= f = State $ \s -> case runState m s of
    (s', x) -> runState (f x) s'
```

## In Control.Monad.State

```
get :: State s s
get = State $ \s -> (s,s)
```

```
modify :: (s -> s) -> State s ()
modify f = State $ \s -> (f s, ())
```

# The State Monad – example

```
type Seed = Int
type SeedGen = State Seed

fresh_seed :: SeedGen Seed
fresh_seed = modify (+1) >> get

test = do
  x <- fresh_seed
  y <- fresh_seed
  z <- fresh_seed
  return [x,y,z]

*Main> print (runState test 0)
(3,[1,2,3])
```

## do-notation

The **do**-notation lets you write ‘instruction-like’ expressions:

- available for expressions/functions of the type `Monad m => m a`
- is syntactic sugar: there is an implicit application of either `>>` or `>>=` between every “instruction”

```
maybeHeadEven :: [Int] -> Maybe Int
```

```
-- maybeHeadEven xs = safeHead xs >>= maybeEven
```

```
-- maybeHeadEven xs = safeHead xs >>= (\i -> maybeEven i)
```

```
maybeHeadEven xs = do i <- safeHead xs -- attention to indentation
                      maybeEven i
```

```
--print123 = print 1 >>= (\_ -> print 2 >>= (\_ -> print 3))
```

```
--print123 = print 1 >> print 2 >> print 3
```

```
print123 = do print 1
              print 2
              print 3
```

## ✂ When to use **do**?

Only use **do** when:

- The expression cannot be written in 'Applicative-style' using `<$>` and `<*>`
- Using `>>` or `>>=` would result in a hard to read expression
- Do not write a **do** block with only one instruction
- When writing IO computations: the above applies, and see [earlier slide](#)

# Enjoy Haskell!

