An Overview of Haskell

Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOG
Robertino Martinez, IOG
Karina Lopez, IOG

August, 2023

Goals

- What is Haskell?
- Provide a few examples of Haskell.
- Explain what makes Haskell unique.

What is Haskell?

- Designed by a committee to create a standard lazy and functional language.
- Haskell 1.0 Report released 1990.
- Several iterations up to Haskell 98, released 1999.
- Minor revision of standard in Haskell 2010.
- A lot of development since then, but primarily outside of the standard, in the Glasgow Haskell Compiler (GHC).

Haskell Features

- functional
- statically typed
- algebraic datatypes
- type inference and polymorphism
  (Damas-Hindley-Milner type system)
- type classes
- explicit effects (often called *pure*)
- lazy evaluation

Haskell Features

- functional
- statically typed
- algebraic datatypes
- type inference and polymorphism
  (Damas-Hindley-Milner type system)
- type classes
- explicit effects (often called *pure*)
- lazy evaluation

Datatypes and functions

A type for “block chains”

```haskell
data Chain =
    GenesisBlock
  | Block Chain Txs

type Txs = Int
```

Terminology:

A type for “block chains”

```
data Chain =
    GenesisBlock
  | Block Chain Txs

type Txs = Int
```

Terminology:

- datatype

A type for “block chains”

```
data Chain =
    GenesisBlock
  | Block Chain Txs

type Txs = Int
```

Terminology:

- datatype
- (data) constructor

A type for "block chains"

```
data Chain =
    GenesisBlock
  | Block Chain Txs

type Txs = Int
```

Terminology:

- datatype
- (data) constructor
- constructor arguments / fields

A type for “block chains”

```
data Chain =
    GenesisBlock
  | Block Chain Txs
type Txs = Int
```

Terminology:

- datatype
- (data) constructor
- constructor arguments / fields
- type synonym

Example chains

```
chain1 =
  Block GenesisBlock 2
chain2 =
  Block chain1 4
```

Example chains

```
chain1 =
  Block GenesisBlock 2
chain2 =
  Block chain1 4
```

```
chain2' =
  Block (Block GenesisBlock 2) 4
```

Terminology:

Example chains

```
chain1 =
  Block GenesisBlock 2
chain2 =
  Block chain1 4
```

```
chain2' =
  Block (Block GenesisBlock 2) 4
```

Terminology:

- binding

Example chains

```
chain1 =
  Block GenesisBlock 2
chain2 =
  Block chain1 4
```

```
chain2' =
  Block (Block GenesisBlock 2) 4
```

Terminology:

- binding
- left hand side

Example chains

```
chain1 =
  Block GenesisBlock 2

chain2 =
  Block chain1 4
```

```
chain2' =
  Block (Block GenesisBlock 2) 4
```

Terminology:

- binding
- left hand side
- right hand side

Example chains

```
chain1 =
  Block GenesisBlock 2
chain2 =
  Block chain1 4
```

```
chain2' =
  Block (Block GenesisBlock 2) 4
```

Terminology:

- binding
- left hand side
- right hand side
- expression

Example chains

```
chain1 =
  Block GenesisBlock 2

chain2 =
  Block chain1 4
```

```
chain2' =
  Block (Block GenesisBlock 2) 4
```

Terminology:

- binding
- left hand side
- right hand side
- expression

Example chains

```
chain1 =
  Block GenesisBlock 2

chain2 =
  Block chain1 4
```

```
chain2' =
  Block (Block GenesisBlock 2) 4
```

Terminology:

- binding
- left hand side
- right hand side
- expression

Determine the length of a block chain

```
chainLength GenesisBlock = 0
chainLength (Block c _)   = chainLength c + 1
```

Determine the length of a block chain

```
chainLength :: Chain -> Int
chainLength GenesisBlock = 0
chainLength (Block c _)   = chainLength c + 1
```

Determine the length of a block chain

```
chainLength :: Chain -> Int
chainLength GenesisBlock = 0
chainLength (Block c _)   = chainLength c + 1
```

Terminology:

- type signature (optional)

Determine the length of a block chain

```
chainLength :: Chain -> Int
chainLength GenesisBlock = 0
chainLength (Block c _)   = chainLength c + 1
```

Terminology:

- type signature (optional)
- equations / cases

Determine the length of a block chain

```
chainLength :: Chain -> Int
chainLength GenesisBlock = 0
chainLength (Block c _)  = chainLength c + 1
```

Terminology:

- type signature (optional)
- equations / cases
- left hand sides

Determine the length of a block chain

```
chainLength :: Chain -> Int
chainLength GenesisBlock = 0
chainLength (Block c _)  = chainLength c + 1
```

Terminology:

- type signature (optional)
- equations / cases
- left hand sides
- patterns

Determine the length of a block chain

```
chainLength :: Chain -> Int
chainLength GenesisBlock = 0
chainLength (Block c _)   = chainLength c + 1
```

Terminology:

- type signature (optional)
- equations / cases
- left hand sides
- patterns
- right hand sides

Determine the length of a block chain

```
chainLength :: Chain -> Int
chainLength GenesisBlock = 0
chainLength (Block c _)   = chainLength c + 1
```

Terminology:

- type signature (optional)
- equations / cases
- left hand sides
- patterns
- right hand sides
- recursive call

Determine the length of a block chain

```haskell
chainLength :: Chain -> Int
chainLength GenesisBlock = 0
chainLength (Block c _)   = chainLength c + 1
```

```haskell
data Chain =
     GenesisBlock
   | Block Chain Txs

type Txs = Int
```

Evaluation in GHCi

```
GHCi> 1 + 1
2
GHCi> chainLength chain1
1
GHCi> chainLength chain1 + 1
2
GHCi> chainLength chain2
2
GHCi> chainLength chain1 == chainLength chain2
False
```

Evaluation step by step

```
chainLength chain2
```

Evaluation step by step

```
  chainLength chain2
= chainLength (Block chain1 4)
```

Evaluation step by step

```
  chainLength chain2
= chainLength (Block chain1 4)
= chainLength chain1 + 1
```

Evaluation step by step

```
  chainLength chain2
= chainLength (Block chain1 4)
= chainLength chain1 + 1
= chainLength (Block GenesisBlock 2) + 1
```

Evaluation step by step

```
  chainLength chain2
= chainLength (Block chain1 4)
= chainLength chain1 + 1
= chainLength (Block GenesisBlock 2) + 1
= (chainLength GenesisBlock + 1) + 1
```

Evaluation step by step

```
  chainLength chain2
= chainLength (Block chain1 4)
= chainLength chain1 + 1
= chainLength (Block GenesisBlock 2) + 1
= (chainLength GenesisBlock + 1) + 1
= (0 + 1) + 1
```

Evaluation step by step

```
  chainLength chain2
= chainLength (Block chain1 4)
= chainLength chain1 + 1
= chainLength (Block GenesisBlock 2) + 1
= (chainLength GenesisBlock + 1) + 1
= (0 + 1) + 1
= 1 + 1
```

Evaluation step by step

```
  chainLength chain2
= chainLength (Block chain1 4)
= chainLength chain1 + 1
= chainLength (Block GenesisBlock 2) + 1
= (chainLength GenesisBlock + 1) + 1
= (0 + 1) + 1
= 1 + 1
= 2
```

Evaluation step by step

```
  chainLength chain2
= chainLength (Block chain1 4)
= chainLength chain1 + 1
= chainLength (Block GenesisBlock 2) + 1
= (chainLength GenesisBlock + 1) + 1
= (0 + 1) + 1
= 1 + 1
= 2
```

Also known as equational reasoning.

Currying

Testing for a particular block

```
hasBlock x GenesisBlock = False
hasBlock x (Block c t)   =
  x == t || hasBlock x c
```

The operator `||` implements logical disjuction (“*or*”).

Testing in GHCi

```
chain1 =
  Block GenesisBlock 2
chain2 =
  Block chain1 4
```

```
GHCi> hasBlock 4 chain1
False
GHCi> hasBlock 4 chain2
True
```

Currying

The (inferred) type of `hasBlock` is

```
GHCi> :t hasBlock
hasBlock :: Txs -> Chain -> Bool
```

Currying

The (inferred) type of `hasBlock` is

```
GHCi> :t hasBlock
hasBlock :: Txs -> Chain -> Bool
```

```
 hasBlock              :: Txs -> (Chain -> Bool)
 hasBlock 4            ::         Chain -> Bool
(hasBlock 4) chain2    ::                  Bool
```

Curried functions and operators

```
(||)  :: Bool -> Bool -> Bool
(&&)  :: Bool -> Bool -> Bool

Block :: Chain -> Txs -> Chain
```

Curried functions and operators

```haskell
(||)  :: Bool -> Bool -> Bool
(&&)  :: Bool -> Bool -> Bool
Block :: Chain -> Txs -> Chain
```

Operators are just identifiers:

```haskell
True || False       -- symbolic infix
(||) True False     -- symbolic prefix
Block chain1 4      -- alphanumeric prefix
chain1 `Block` 4    -- alphanumeric infix
```

Operator priorities

```
GHCi> :i (||)
(||) :: Bool -> Bool -> Bool
    -- Defined in ‘GHC.Classes‘
infixr 2 ||
```

Operator priorities

```
GHCi> :i (||)
(||) :: Bool -> Bool -> Bool
    -- Defined in ‘GHC.Classes‘
infixr 2 ||
```

Defining a symbolic version of ‘Block‘:

```
(|>) :: Chain -> Txs -> Chain
(|>) = Block
infixl 5 |>
```

Operator priorities

```
GHCi> :i (||)
(||) :: Bool -> Bool -> Bool
    -- Defined in ‘GHC.Classes‘
infixr 2 ||
```

Defining a symbolic version of ‘Block‘:

```
(|>) :: Chain -> Txs -> Chain
(|>) = Block
infixl 5 |>
```

```
chain2'' :: Chain
chain2'' = GenesisBlock |> 2 |> 4
```

Higher-order functions

Finding a block with a property

Finding a block with a property

Let's model a property as a function of type `Txs -> Bool`.

Finding a block with a property

Let's model a property as a function of type `Txs -> Bool`.

```
hasBlockProp :: (Txs -> Bool) -> Chain -> Bool
hasBlockProp prop GenesisBlock = False
hasBlockProp prop (Block c t)   =
  prop t || hasBlockProp prop c
```

Anonymous functions and operator sections

```
GHCi> hasBlockProp (\ x -> x > 10) chain2
False
GHCi> hasBlockProp even chain2
True
GHCi> hasBlockProp (\ x -> 4 == x) chain2
True
```

Anonymous functions and operator sections

```
GHCi> hasBlockProp (\ x -> x > 10) chain2
False
GHCi> hasBlockProp even chain2
True
GHCi> hasBlockProp (\ x -> 4 == x) chain2
True
```

```
GHCi> hasBlockProp (> 10) chain2
False
GHCi> hasBlockProp (4 ==) chain2
True
```

Several styles to write one function

```haskell
hasBlockProp :: (Txs -> Bool) -> Chain -> Bool
hasBlockProp prop GenesisBlock = False
hasBlockProp prop (Block c t)   =
  prop t || hasBlockProp prop c
```

vs.

```haskell
hasBlockProp :: (Txs -> Bool) -> Chain -> Bool
hasBlockProp = \ prop chain ->
  case chain of
    GenesisBlock -> False
    Block c t    -> prop t || hasBlockProp prop c
```

Similarity of functions / type-directed programming

```haskell
chainLength :: Chain -> Int
chainLength GenesisBlock = 0
chainLength (Block c _)   = chainLength c + 1
```

```haskell
hasBlock :: Txs -> Chain -> Bool
hasBlock x GenesisBlock = False
hasBlock x (Block c t)   =
  x == t || hasBlock x c
```

```haskell
hasBlockProp :: (Txs -> Bool) -> Chain -> Bool
hasBlockProp prop GenesisBlock = False
hasBlockProp prop (Block c t)   =
  prop t || hasBlockProp prop c
```

Types, overloading and polymorphism

Type inference

```
hasBlockProp ::
hasBlockProp prop GenesisBlock = False
hasBlockProp prop (Block c t)   =
  prop t || hasBlockProp prop c
```

Type inference

```
hasBlockProp :: … -> … -> …
hasBlockProp prop GenesisBlock = False
hasBlockProp prop (Block c t)   =
  prop t || hasBlockProp prop c
```

- Two arguments.

Type inference

```
hasBlockProp :: … -> … -> Bool
hasBlockProp prop GenesisBlock = False
hasBlockProp prop (Block c t)   =
  prop t || hasBlockProp prop c
```

- Two arguments.
- Result of first case is `False`.

Type inference

```
hasBlockProp :: ... -> Chain -> Bool
hasBlockProp prop GenesisBlock = False
hasBlockProp prop (Block c t)  =
  prop t || hasBlockProp prop c
```

- Two arguments.
- Result of first case is `False`.
- Pattern for second arg in first case is `GenesisBlock`.

Type inference

```
hasBlockProp :: ... -> Chain -> Bool
hasBlockProp prop GenesisBlock = False
hasBlockProp prop (Block c t)   =
  prop t || hasBlockProp prop c
```

- Two arguments.
- Result of first case is `False`.
- Pattern for second arg in first case is `GenesisBlock`.
- We know `t :: Txs` due to the type of `Block`.

Type inference

```haskell
hasBlockProp :: (Txs -> Bool) -> Chain -> Bool
hasBlockProp prop GenesisBlock = False
hasBlockProp prop (Block c t)  =
  prop t || hasBlockProp prop c
```

- Two arguments.
- Result of first case is `False`.
- Pattern for second arg in first case is `GenesisBlock`.
- We know `t :: Txs` due to the type of `Block`.
- We know `prop :: Txs -> Bool` due to `prop t || ...`.

(Parametric) Polymorphism

```
data Chain =
      GenesisBlock
    | Block Chain Txs
```

(Parametric) Polymorphism

```
data Chain =
    GenesisBlock
  | Block Chain Txs
```

Abstract from the type of transactions:

```
data Chain txs =
    GenesisBlock
  | Block (Chain txs) txs
```

Polymorphic types

```
GenesisBlock :: Chain txs
Block :: Chain txs -> txs -> Chain txs
```

Polymorphic types

```haskell
GenesisBlock :: Chain txs
Block :: Chain txs -> txs -> Chain txs
```

```haskell
chainLength :: Chain txs -> Int
hasBlockProp :: (txs -> Bool) -> Chain txs -> Bool
```

These work for any choice of type `txs`!

Overloading

```
hasBlock x GenesisBlock = False
hasBlock x (Block c t)   =
  x == t || hasBlock x c
```

Not entirely independent of the type of `x` and `t`.

Overloading

```
hasBlock x GenesisBlock = False
hasBlock x (Block c t)  =
  x == t || hasBlock x c
```

Not entirely independent of the type of `x` and `t`.

```
(==) :: Eq a => a -> a -> Bool
```

Overloading

```
hasBlock x GenesisBlock = False
hasBlock x (Block c t)  =
  x == t || hasBlock x c
```

Not entirely independent of the type of `x` and `t`.

```
(==) :: Eq a => a -> a -> Bool
```

This type has a (class) constraint.

It works for any choice of type `a` that is in the `Eq` class.

Overloading

```
hasBlock :: Eq txs => txs -> Chain txs -> Bool
hasBlock x GenesisBlock = False
hasBlock x (Block c t)   =
  x == t || hasBlock x c
```

Not entirely independent of the type of `x` and `t`.

```
(==) :: Eq a => a -> a -> Bool
```

This type has a (class) constraint.

It works for any choice of type `a` that is in the `Eq` class.

Chains vs. Lists

Polymorphic chains are quite similar to built-in lists:

```
GenesisBlock :: Chain txs
Block        :: Chain txs -> txs -> Chain txs

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

Chains vs. Lists

Polymorphic chains are quite similar to built-in lists:

```
GenesisBlock :: Chain txs
Block        :: Chain txs -> txs -> Chain txs

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

```
anotherChain :: Chain Int
anotherChain =
  Block (Block GenesisBlock 1) 2
```

```
aList :: [Int]
aList = 2 : (1 : [])
```

Syntactic sugar for lists

These are all equivalent:

```
2 : (1 : [])
2 : 1 : []
[2, 1]
```

Syntactic sugar for lists

These are all equivalent:

```
2 : (1 : [])
2 : 1 : []
[2, 1]
```

Lists of characters are called strings:

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

These are all equivalent for strings:

```
'x' : 'y' : []
['x', 'y']
"xy"
```

Overloaded length

Corresponds to `chainLength`:

```
length :: Foldable t => t a -> Int
```

Here, `Foldable`

- is another type class
- contains container types, such as lists

Overloaded length

Corresponds to `chainLength`:

```
length :: Foldable t => t a -> Int
```

Here, `Foldable`

- is another type class
- contains container types, such as lists

Specializations:

```
length :: [a] -> Int
length :: [Int] -> Int
length :: String -> Int
```

Other functions on lists

```haskell
elem :: (Eq a, Foldable t) => a -> t a -> Bool
any  :: Foldable t => (a -> Bool) -> t a -> Bool
```

Specializations:

```haskell
elem :: (Eq a) => a -> [a] -> Bool
elem :: Txs -> [Txs] -> Bool
any  :: (a -> Bool) -> [a] -> Bool
any  :: (Txs -> Bool) -> [Txs] -> Bool
```

Various other predefined functions

```
reverse :: [a] -> [a]
(++)    :: [a] -> [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
map     :: (a -> b) -> [a] -> [b]
```

Various other predefined functions

```haskell
reverse :: [a] -> [a]
(++)    :: [a] -> [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
map     :: (a -> b) -> [a] -> [b]
```

```haskell
id    :: a -> a
const :: a -> b -> a
```

Various other predefined functions

```
reverse :: [a] -> [a]
(++)    :: [a] -> [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
map     :: (a -> b) -> [a] -> [b]
```

```
id    :: a -> a
const :: a -> b -> a
```

```
(+)  :: Num a => a -> a -> a
(/)  :: Fractional a => a -> a -> a
(<=) :: Ord a => a -> a -> Bool
```

Various other predefined functions

```
reverse :: [a] -> [a]
(++)    :: [a] -> [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
map     :: (a -> b) -> [a] -> [b]
```

```
id    :: a -> a
const :: a -> b -> a
```

```
(+)  :: Num a => a -> a -> a
(/)  :: Fractional a => a -> a -> a
(<=) :: Ord a => a -> a -> Bool
```

```
show :: Show a => a -> String
```

Some important type classes

`Eq`	for types that support an equality test
`Ord`	for types that can be compared
`Num`	for numeric types
`Fractional`	for fractional numeric types
`Show`	for types that have a representation as string
`Read`	for types that can be parsed from strings
`Enum`	for types that can be enumerated
`Bounded`	for types that have a smallest and largest value
`Foldable`	for container types

Deriving instances

For some classes, we can automagically derive instances[1]:

```
data Chain txs =
    GenesisBlock
  | Block (Chain txs) txs
  deriving (Eq, Show, Foldable)
```

[1] For `Foldable`, a language extension is required.

Deriving instances

For some classes, we can automagically derive instances[1]:

```
data Chain txs =
    GenesisBlock
  | Block (Chain txs) txs
 deriving (Eq, Show, Foldable)
```

Generally, we have to manually provide `instance` declarations.

[1] For `Foldable`, a language extension is required.

Explicit effects

Pure functions

- Function results depend only on their inputs.
- Functions have no side effects.

Pure functions

- Function results depend only on their inputs.
- Functions have no side effects.

Side effects are expressed by the `IO` type:

```
f :: Int -> Int       -- result depends only on argument
g :: Int -> IO Int    -- result is not an Int, but an action
```

Reading a file from disk

```haskell
readFile :: FilePath -> IO String

type FilePath = String
```

Reading a file from disk

```
readFile :: FilePath -> IO String

type FilePath = String
```

The following is not type correct:

```
lengthOfFile file =
  length (readFile file)  -- type error!
```

Reading a file from disk

```haskell
readFile :: FilePath -> IO String

type FilePath = String
```

The following is not type correct:

```haskell
lengthOfFile file =
  length (readFile file)  -- type error!
```

This is correct:

```haskell
lengthOfFile :: FilePath -> IO Int
lengthOfFile file =
  length <$> readFile file
```

Constructing complex IO actions

```
(<$>) :: (a -> b) -> IO a -> IO b
(>>)  :: IO a -> IO b -> IO b
(<*>) :: IO (a -> b) -> IO a -> IO b
(>>=) :: IO a -> (a -> IO b) -> IO b
```

(These all have more general types, but that does not matter now.)

No escape

- Results that depend on effects have an `IO` marker in their types.
- We cannot[2] lie about this.
- Effectful programs have stronger requirements than non-effectful ones.
- Encourages a programming style that separates effectful wrapper from pure kernels.

[2] not easily

Lazy evaluation

Building a chain

```haskell
build :: Int -> Chain Int
build n =
  if n <= 0
    then GenesisBlock
    else Block (build (n - 1)) n
```

Building a chain

```haskell
build :: Int -> Chain Int
build n =
  if n <= 0
    then GenesisBlock
    else Block (build (n - 1)) n
```

```
GHCi> build 2
Block (Block GenesisBlock 1) 2
```

Building a long chain

This takes a while:

```
GHCi> length (build 10000000)
10000000
```

Building a long chain

This takes a while:

```
GHCi> length (build 10000000)
10000000
```

What about this?

```
GHCi> hasBlockProp even (build 10000000)
```

Building a long chain

This takes a while:

```
GHCi> length (build 10000000)
10000000
```

What about this?

```
GHCi> hasBlockProp even (build 10000000)
True
```

This is nearly immediate.

Multiple options to reduce

```
hasBlockProp even (build 10000000)
```

Multiple options to reduce

```
  hasBlockProp even (build 10000000)
= hasBlockProp even
    (if 10000000 <= 0
      then GenesisBlock
      else Block (build (10000000 - 1)) 10000000
    )
```

Multiple options to reduce

```
  hasBlockProp even (build 10000000)
= hasBlockProp even
    (if 10000000 <= 0
      then GenesisBlock
      else Block (build (10000000 - 1)) 10000000
    )
= hasBlockProp even
    (if False
      then GenesisBlock
      else Block (build (10000000 - 1)) 10000000
    )
```

Multiple options to reduce

```
  hasBlockProp even (build 10000000)
= hasBlockProp even
    (if 10000000 <= 0
      then GenesisBlock
      else Block (build (10000000 - 1)) 10000000
    )
= hasBlockProp even
    (if False
      then GenesisBlock
      else Block (build (10000000 - 1)) 10000000
    )
= hasBlockProp even
    (Block (build (10000000 - 1)) 10000000)
```

Multiple options to reduce

```
  hasBlockProp even (build 10000000)
= hasBlockProp even
    (if 10000000 <= 0
      then GenesisBlock
      else Block (build (10000000 - 1)) 10000000
    )
= hasBlockProp even
    (if False
      then GenesisBlock
      else Block (build (10000000 - 1)) 10000000
    )
= hasBlockProp even
    (Block (build (10000000 - 1)) 10000000)
```

Multiple options to reduce

```
  hasBlockProp even (build 10000000)
= hasBlockProp even
    (if 10000000 <= 0
      then GenesisBlock
      else Block (build (10000000 - 1)) 10000000
    )
= hasBlockProp even
    (if False
      then GenesisBlock
      else Block (build (10000000 - 1)) 10000000
    )
= hasBlockProp even
    (Block (build (10000000 - 1)) 10000000)
```

Leftmost outermost

```
= hasBlockProp even
    (Block (build (10000000 - 1)) 10000000)
```

Leftmost outermost

```
= hasBlockProp even
    (Block (build (10000000 - 1)) 10000000)
= even 10000000
     || hasBlockProp even (build (10000000 - 1))
```

Leftmost outermost

```
= hasBlockProp even
    (Block (build (10000000 - 1)) 10000000)
= even 10000000
     || hasBlockProp even (build (10000000 - 1))
= True
     || hasBlockProp even (build (10000000 - 1))
```

Leftmost outermost

```
= hasBlockProp even
    (Block (build (10000000 - 1)) 10000000)
= even 10000000
     || hasBlockProp even (build (10000000 - 1))
= True
     || hasBlockProp even (build (10000000 - 1))
= True
```

Lazy evaluation

- The leftmost outermost expressions are reduced.
- Expressions are only evaluated when needed.
- This implies we can work with potentially infinite values.
- More importantly, it allows for better separation of concerns and the definition of “control-flow constructs”.
- The disadvantage is that it becomes more difficult to reason about performance.

Summary

What we have seen

- Being able to define higher-order functions easily is a feature all functional languages share.
- Polymorphism, data types and pattern matching are common to statically typed functional languages.
- Type classes are rather unique to Haskell, but other languages have other features instead.
- Explicit effects and lazy evaluation make Haskell truly special (although some languages derived from Haskell take a similar approach).

What is next?

First exercises!

We will focus on two fundamental points of Haskell programming:

- How to define datatypes.
- How datatypes shape functions (type-directed programming).