Part 1: Introduction

Introduction to Haskell

Andres Löh

Well-Typed
The Haskell Consultants

In this part

Getting started with Haskell:

- What is Haskell?
- Haskell tooling overview
- Expressions
- Type inference
- Parametric polymorphism and overloading
- IO and explicit effects
- Datatypes and functions
- Pattern matching
- Lazy evaluation

In this part

Getting started with Haskell:

- What is Haskell?
- Haskell tooling overview
- Expressions
- Type inference
- Parametric polymorphism and overloading
- IO and explicit effects
- Datatypes and functions
- Pattern matching
- Lazy evaluation

Not yet a detailed introduction to these topics – everything will be covered in more detail later.

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 tooling

GHCi

- Interactive component shipped with GHC.
- Use to quickly try out things, evaluate Haskell **expressions**, obtain (type) information, and test your programs.

Haskell tooling

GHCi

- Interactive component shipped with GHC.
- Use to quickly try out things, evaluate Haskell **expressions**, obtain (type) information, and test your programs.

Editor (potentially with haskell-language-server)

- Use to define Haskell **modules** containing **declarations**.
- Test these developments using GHCi again.

Haskell tooling

GHCi

- Interactive component shipped with GHC.
- Use to quickly try out things, evaluate Haskell **expressions**, obtain (type) information, and test your programs.

Editor (potentially with haskell-language-server)

- Use to define Haskell **modules** containing **declarations**.
- Test these developments using GHCi again.

Note

You cannot just enter any Haskell program line-by-line into GHCi. Use an editor for any more complex development.

Expressions

Expressions in context

- Haskell programs are structured into **modules** (think: files).
- Modules contain **declarations**.
- The most important form of declarations are **bindings** for new constants and functions.
- In such bindings, **expressions** play the central role.

Therefore we are going to look at expressions before everything else.

Expressions

A Haskell **expression** is a (possibly nested) terms built up from constants and function calls.

Expressions

A Haskell **expression** is a (possibly nested) terms built up from constants and function calls.

- An important property of expressions is that they can be **evaluated**, yielding a **value**.
- Values are themselves expressions that cannot be evaluated any further.
- You can type expressions into GHCi. GHCi will then try to evaluate the expression and print its resulting value.

Examples of values

```
GHCi> 2
2
```

```
GHCi> 'x'
'x'
```

```
GHCi> "Haskell"
"Haskell"
```

```
GHCi> True
True
```

```
GHCi> [1,2,4]
[1,2,4]
```

Examples of function calls

```
GHCi> not True
False
```

```
GHCi> min 7 2
2
```

```
GHCi> 2 + 3
5
```

```
GHCi> 3 : [10,99]
[3,10,99]
```

```
GHCi> take 2 [1,3,9,27,81]
[1,3]
```

```
GHCi> map odd [1,2,3,4,5]
[True,False,True,False,True]
```

Operators are functions

Only syntactic differences between symbolic and alphanumeric function names.

Operators are functions

Only syntactic differences between symbolic and alphanumeric function names.

```
GHCi> 6 + 9
15
GHCi> (+) 6 9
15
```

Symbolic identifiers (operators) are **infix** by default, and can be made prefix by enclosing them in parentheses.

Operators are functions

Only syntactic differences between symbolic and alphanumeric function names.

```
GHCi> 6 + 9
15
GHCi> (+) 6 9
15
```

Symbolic identifiers (operators) are **infix** by default, and can be made prefix by enclosing them in parentheses.

```
GHCi> max 12 20
20
GHCi> 12 `max` 20
20
```

Alphanumeric identifiers are **prefix** by default, and can be made infix by enclosing them in **back**quotes.

Well-Typed

Function application syntax

"Space" is function application:

```
min 7 2  -- function applied to two arguments
```

Function application syntax

“Space” is function application:

```
min 7 2   -- function applied to two arguments
```

Parentheses are used for grouping:

```
GHCi> min 7 (2 + 6)
7
GHCi> min 7 2 + 6
8
```

Function application binds stronger than operators.

More examples of expressions

```
GHCi> reverse (reverse [1,2,3])
[1,2,3]
```

```
GHCi> sum (filter odd [1,2,3,4,5])
9
```

```
GHCi> take 1 "Haskell" ++ drop 4 "Haskell"
"Hell"
```

Anonymous functions (or lambda terms)

Referring to functions without giving them a name:

```
GHCi> (\ x -> x + 3) 4
7
```

```
GHCi> (\ list n -> take n (reverse list)) "hello" 3
"oll"
```

The `\` is pronounced “lambda”.

Well-Typed

Anonymous functions (or lambda terms)

Referring to functions without giving them a name:

```
GHCi> (\ x -> x + 3) 4
7
```

```
GHCi> (\ list n -> take n (reverse list)) "hello" 3
"oll"
```

The `\` is pronounced “lambda”.

Particularly useful as an argument to another function:

```
GHCi> map (\ x -> 3 * x + 1) [1,2,3]
[4,7,10]
```

Functions taking other functions as arguments are called **higher-order functions**.

First recap

We have learned about:

- Expressions and values,
- Functions and operators,
- Numbers, characters, Booleans, lists (and strings),
- Lambda terms.

In particular, try to get used to the function application syntax (using space) and the use of parentheses only for grouping.

Types

Expressions have types

- Every expression (and subexpression) has a type.
- Types are checked statically.
- Types can be inferred.

Expressions have types

- Every expression (and subexpression) has a type.
- Types are checked statically.
- Types can be inferred.

You can use GHCi (or your editor with haskell-language-server) to infer types.

Inferring types in GHCi

```
GHCi> :t 'x'
'x' :: Char
```

```
GHCi> :t False
False :: Bool
```

```
GHCi> :t [True,False]
[True,False] :: [Bool]
```

```
GHCi> :t not
not :: Bool -> Bool
```

Inferring types in GHCi

```
GHCi> :t 'x'
'x' :: Char
```

```
GHCi> :t False
False :: Bool
```

```
GHCi> :t [True,False]
[True,False] :: [Bool]
```

```
GHCi> :t not
not :: Bool -> Bool
```

- `:t` is a **GHCi command** – it's not a part of Haskell.
- The `::` symbol reads "is of type".
- Types are different from expressions. You cannot use a type as an expression.

Parametric polymorphism

```
GHCi> :t reverse
reverse :: [a] -> [a]
```

```
GHCi> reverse [True,False]
[False,True]
```

```
GHCi> reverse [1,2,3]
[3,2,1]
```

```
GHCi> reverse "Haskell"
"lleksaH"
```

Parametric polymorphism

```
GHCi> :t reverse
reverse :: [a] -> [a]
```

```
GHCi> reverse [True,False]
[False,True]
```

```
GHCi> reverse [1,2,3]
[3,2,1]
```

```
GHCi> reverse "Haskell"
"lleksaH"
```

- Lower-case identifiers in types are **type variables**.
- Types involving variables are called **parametrically polymorphic**.
- We can choose at which concrete type to use the expression.
- Strings are lists of characters.

Currying

```
GHCi> :t take
take :: Int -> [a] -> [a]
```

Currying

```
GHCi> :t take
take :: Int -> [a] -> [a]
```

Two views:

- A function that takes an `Int` and a `[a]` and returns a `[a]`.

Currying

```
GHCi> :t take
take :: Int -> [a] -> [a]
```

Two views:

- A function that takes an `Int` and a `[a]` and returns a `[a]`.
- A function that takes an `Int` and returns another function, which then expects a `[a]` and returns a `[a]`.

Currying

```
GHCi> :t take
take :: Int -> [a] -> [a]
```

Two views:

- A function that takes an `Int` and a `[a]` and returns a `[a]`.
- A function that takes an `Int` and returns another function, which then expects a `[a]` and returns a `[a]`.

The function arrow associates to the right. These two types are the same:

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

Partial application

```
GHCi> :t take
take :: Int -> [a] -> [a]
GHCi> :t take 2
take 2 :: [a] -> [a]
GHCi> :t take 2 "currying"
take 2 "currying" :: [Char]
```

```
GHCi> take 2 "currying"
"cu"
```

Partial application in use

```
GHCi> :t map
map :: (a -> b) -> [a] -> [b]
GHCi> :t take 2
take 2 :: [a] -> [a]
GHCi> :t map (take 2)
map (take 2) :: [[a]] -> [[a]]
```

Partial application in use

```
GHCi> :t map
map :: (a -> b) -> [a] -> [b]
GHCi> :t take 2
take 2 :: [a] -> [a]
GHCi> :t map (take 2)
map (take 2) :: [[a]] -> [[a]]
```

```
GHCi> map (take 2) [[1,2,3],[4,5,6],[7,8,9]]
[[1,2],[4,5],[7,8]]
```

Partial application in use

```
GHCi> :t map
map :: (a -> b) -> [a] -> [b]
GHCi> :t take 2
take 2 :: [a] -> [a]
GHCi> :t map (take 2)
map (take 2) :: [[a]] -> [[a]]
```

```
GHCi> map (take 2) [[1,2,3],[4,5,6],[7,8,9]]
[[1,2],[4,5],[7,8]]
```

Partial application can be seen as an abbreviation for a lambda term:

```
GHCi> map (\ x -> take 2 x) [[1,2,3],[4,5,6],[7,8,9]]
[[1,2],[4,5],[7,8]]
```

Overloading

Some functions work for many, but not all types:

```
GHCi> :t enumFromTo
enumFromTo :: Enum a => a -> a -> [a]
```

Overloading

Some functions work for many, but not all types:

```
GHCi> :t enumFromTo
enumFromTo :: Enum a => a -> a -> [a]
```

The part of to the left of the `=>` is a **constraint** which restricts the choice of the type variable `a` to types that are an **instance** of the **type class** `Enum`.

Many types are an instance of `Enum`, but not all types are.

Overloading in use

```
GHCi> enumFromTo 1 5
[1,2,3,4,5]
```

```
GHCi> enumFromTo 'a' 'c'
"abc"
```

```
GHCi> enumFromTo False True
[False,True]
```

Overloading in use

```
GHCi> enumFromTo 1 5
[1,2,3,4,5]
```

```
GHCi> enumFromTo 'a' 'c'
"abc"
```

```
GHCi> enumFromTo False True
[False,True]
```

```
GHCi> enumFromTo "abc" "def"
```

No instance for (`Enum [Char]`) arising from a use of ‘`enumFromTo`’

Lists are not an instance of the `Enum` class.

Well-Typed

Overloading is used in lots of places

```
GHCi> :t max
max :: Ord a => a -> a -> a
```

```
GHCi> :t (+)
(+) :: Num a => a -> a -> a
```

Overloading is used in lots of places

```
GHCi> :t max
max :: Ord a => a -> a -> a
```

```
GHCi> :t (+)
(+) :: Num a => a -> a -> a
```

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

The “container” is restricted, but not the element type:

```
GHCi> length [(+),(-),\ x y -> x * x + y]
3
```

For now, if you see `Foldable` , think “list”.

Type errors

```
GHCi> not 'x'
```

Couldn't match expected type '`Bool`' with actual type '`Char`'

Type errors

```
GHCi> not 'x'
```

Couldn't match expected type '`Bool`' with actual type '`Char`'

Numeric literals are overloaded, and cause somewhat confusing errors:

```
GHCi> :t 1
1 :: Num p => p
GHCi> not 1
```

No instance for (`Num Bool`) arising from the literal '`1`'

GHCi pitfall

This looks like a type error, but the error is not in your code:

```
GHCi> take
No instance for (Show (Int -> [a0] -> [a0]))
  arising from a use of 'print'
```

The expression is in fact type-correct:

```
GHCi> :t take
take :: Int -> [a] -> [a]
```

GHCi pitfall

This looks like a type error, but the error is not in your code:

```
GHCi> take
No instance for (Show (Int -> [a0] -> [a0]))
  arising from a use of 'print'
```

The expression is in fact type-correct:

```
GHCi> :t take
take :: Int -> [a] -> [a]
```

GHCi implicitly tries to call `print` on the expressions you type, to print the result of evaluation on screen, and this fails ...

Explicit effects

```
GHCi> :t print
print :: Show a => a -> IO ()
```

- The type `()` is a type containing just one value `()`.
- The type `IO ()` denotes an IO **action** yielding no interesting result, but having the **side effect** of printing the argument to the screen.
- The argument is flexible, but constrained to be an instance of the `Show` class.
- Functions are not an instance of the `Show` class, hence the GHCi error when typing in anything of a functional type.
- In Haskell, all side-effecting operations are explicitly marked by being elements of the `IO` type.

Second recap

We have learned about:

- Inferring types with GHCi,
- Currying and partial application,
- Parametric polymorphism (types containing unconstrained variables),
- Overloading (types containing constrained variables),
- Explicit effects (the `IO` type).

The primary goals for now are to be aware of `:t` in GHCi and to be able to make some sense of the reported types.

Datatypes and functions

Bindings

A **binding** is a **declaration** that gives a name to an expression so that it can be reused.

```
five = 2 + 3
ten = five + five
aList = [1,2,3,4,five]
```

Such bindings are typically put into a Haskell file in an editor, with the extension `.hs`.

(One can make bindings directly in GHCi, but certain restrictions are then in place.)

Using new bindings in GHCi

One can then load the source file into GHCi and use the bindings:

```
GHCi> five
5
GHCi> ten
10
GHCi> map (\ x -> x * ten) aList
[10,20,30,40,50]
```

Function bindings

```
double = \ x -> x + x
```

A different way to define the same function:

```
double x = x + x
```

Function bindings

```
double = \ x -> x + x
```

A different way to define the same function:

```
double x = x + x
```

```
GHCi> double 3
6
GHCi> double (-10)
-20
```

Function bindings

```
double = \ x -> x + x
```

A different way to define the same function:

```
double x = x + x
```

```
GHCi> double 3
6
GHCi> double (-10)
-20
```

Careful with negative numbers!

```
GHCi> double - 10
```

Non type-variable argument in the constraint: `Num (a -> a)`

(A slightly strange way to complain that there is no `Num` instance for function types.)

Type signatures

Type signatures are optional, but strongly encouraged:

```
distance :: Num a => a -> a -> a
distance x1 x2 = abs (x1 - x2)
```

- Type signatures are checked and enforced.
- Types provide guidance for defining functions.
- Typically type errors are better with explicit type signatures.

The `Prelude`

Very little is built into Haskell. E.g., all of

```
(+)
min
not
```

are **library** functions.

The `Prelude`

Very little is built into Haskell. E.g., all of

```
(+)
min
not
```

are **library** functions.

- Code is organized into **modules**.
- One special module `Prelude` is implicitly available in any other Haskell module.

Defining a datatype

```
data Choice = Rock | Paper | Scissors
  deriving Show
```

Defines a **new** datatype `Choice` with just three values: `Rock`, `Paper`, and `Scissors`.

Defining a datatype

```
data Choice = Rock | Paper | Scissors
  deriving Show
```

Defines a **new** datatype `Choice` with just three values: `Rock`, `Paper`, and `Scissors`.

```
GHCi> :t Rock
Choice
GHCi> :t Paper
Choice
GHCi> Rock
Rock
```

The last command works only because `deriving Show` instructs GHC to create an "obvious" `Show` instance for the new datatype.

Pattern matching

```
improve :: Choice -> Choice
improve Rock     = Paper
improve Paper    = Scissors
improve Scissors = Rock
```

The terms `Rock`, `Paper` and `Scissors` are called the **(data) constructors** of the `Choice` type.

Data constructors can be used for **pattern matching**.

If a binding has multiple equations, then the patterns on the left hand sides determine which equation applies for a given argument.

Pattern matching

```
improve :: Choice -> Choice
improve Rock     = Paper
improve Paper    = Scissors
improve Scissors = Rock
```

The terms `Rock`, `Paper` and `Scissors` are called the **(data) constructors** of the `Choice` type.

Data constructors can be used for **pattern matching**.

If a binding has multiple equations, then the patterns on the left hand sides determine which equation applies for a given argument.

```
GHCi> improve Paper
Scissors
```

Logical or

The actual definitions of `Bool` and `(|)` :

```
data Bool = False | True
  deriving Show   -- and other classes
```

```
(||) :: Bool -> Bool -> Bool
False || y = y
True  || y = True
```

Logical or

The actual definitions of `Bool` and `(|)` :

```
data Bool = False | True
  deriving Show   -- and other classes
```

```
(||) :: Bool -> Bool -> Bool
False || y = y
True  || y = True
```

```
GHCi> True || False
True
GHCi> False || False
False
```

Our own list type

```
data List a = Nil | Cons a (List a)
  deriving Show
```

Our own list type

```haskell
data List a = Nil | Cons a (List a)
  deriving Show
```

```
GHCi> :t Nil
List a
GHCi> :t Cons
a -> List a -> List a
GHCi> :t Cons 1 (Cons 2 (Cons 3 Nil))
Cons 1 (Cons 2 (Cons 3 Nil)) :: Num a => List a
GHCi> Cons 1 (Cons 2 (Cons 3 Nil))
Cons 1 (Cons 2 (Cons 3 Nil))
```

Built-in lists

Special syntax:

```
data [a] = [] | a : [a]
  deriving Show   -- and other classes
infixr 5 :      -- the “cons” operator is right-associative
```

Built-in lists

Special syntax:

```
data [a] = [] | a : [a]
  deriving Show   -- and other classes

infixr 5 :      -- the "cons" operator is right-associative
```

```
GHCi> :t []
[a]
GHCi> :t (:)
a -> [a] -> [a]
GHCi> :t 1 : 2 : 3 : []
1 : 2 : 3 : [] :: Num a => [a]
GHCi> 1 : 2 : 3 : []
[1,2,3]
```

The notation `[1,2,3]` is actually **syntactic sugar** for `1 : 2 : 3 : []`.

Pattern matching on lists

```
elem x []        = ...
```

What if we are looking for an element in the empty list?

Pattern matching on lists

```
elem x []       = False
```

If a list is not `[]`, it must be of shape `y : ys` for a suitable “head” `y` and “tail” `ys` …

Pattern matching on lists

```
elem x []       = False
elem x (y : ys) = ...
```

One option is that `x` is equal to `y` ...

Pattern matching on lists

```
elem x []       = False
elem x (y : ys) = x == y
```

Here, `(==)` tests two expressions for equality.

This definition works, but it is not correct. We also need to consider `ys` …

Pattern matching on lists

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

Recursion is the answer!

The list datatype definition is recursive. Functions on lists are typically recursive as well.

What about a type signature?

Pattern matching on lists

```
elem :: Eq a => a -> [a] -> Bool
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

We make no assumptions about the type of list elements except that we can perform the equality test, which comes from the `Eq` class.

Haskell's evaluation model

- Expressions are "reduced" to values.
- For function calls, find matching equations.
- Replace left hand sides by right hand sides.
- Stop once no more reduction is possible (a value is reached).
- This process is called **equational reasoning**.

Equational reasoning example

```
    elem 9 [6,9,42]
```

Remember:

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

```
False || y = y
True  || y = True
```

Equational reasoning example

```
   elem 9 [6,9,42]
⇝ elem 9 (6 : (9 : (42 : [])))
```

Remember:

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

```
False || y = y
True  || y = True
```

Well-Typed

Equational reasoning example

```
   elem 9 [6,9,42]
⇝ elem 9 (6 : (9 : (42 : [])))
⇝ 9 == 6 || elem 9 (9 : 42 : [])
```

Remember:

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

```
False || y = y
True  || y = True
```

Equational reasoning example

```
   elem 9 [6,9,42]
⇝ elem 9 (6 : (9 : (42 : [])))
⇝ 9 == 6 || elem 9 (9 : 42 : [])
⇝ False  || elem 9 (9 : 42 : [])
```

Remember:

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

```
False || y = y
True  || y = True
```

Equational reasoning example

```
   elem 9 [6,9,42]
⇝ elem 9 (6 : (9 : (42 : [])))
⇝ 9 == 6 || elem 9 (9 : 42 : [])
⇝ False  || elem 9 (9 : 42 : [])
⇝ elem 9 (9 : 42 : [])
```

Remember:

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

```
False || y = y
True  || y = True
```

Equational reasoning example

```
   elem 9 [6,9,42]
⇝ elem 9 (6 : (9 : (42 : [])))
⇝ 9 == 6 || elem 9 (9 : 42 : [])
⇝ False  || elem 9 (9 : 42 : [])
⇝ elem 9 (9 : 42 : [])
⇝ 9 == 9 || elem 9 (42 : [])
```

Remember:

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

```
False || y = y
True  || y = True
```

Equational reasoning example

```
   elem 9 [6,9,42]
 ⇝ elem 9 (6 : (9 : (42 : [])))
 ⇝ 9 == 6 || elem 9 (9 : 42 : [])
 ⇝ False  || elem 9 (9 : 42 : [])
 ⇝ elem 9 (9 : 42 : [])
 ⇝ 9 == 9 || elem 9 (42 : [])
 ⇝ True   || elem 9 (42 : [])
```

Remember:

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

```
False || y = y
True  || y = True
```

Equational reasoning example

```
   elem 9 [6,9,42]
⇝ elem 9 (6 : (9 : (42 : [])))
⇝ 9 == 6 || elem 9 (9 : 42 : [])
⇝ False  || elem 9 (9 : 42 : [])
⇝ elem 9 (9 : 42 : [])
⇝ 9 == 9 || elem 9 (42 : [])
⇝ True   || elem 9 (42 : [])
⇝ True
```

Remember:

```
elem x []       = False
elem x (y : ys) = x == y || elem x ys
```

```
False || y = y
True  || y = True
```

Well-Typed

Lazy evaluation

Let's look at the definition of "or" again:

```
False || y = y
True  || y = True
```

- We can make a decision without looking at the second argument (and indeed we did, while reducing `elem`).
- This definition of `(||)` has "shortcut behaviour".
- Unlike in many languages, this does not require a special hack, but follows from the definition and Haskell's **evaluation strategy** that essentially says **"only evaluate things once they are needed"**.

Recap

The most fundamental concept to define functions on datatypes is **pattern matching**:

- We distinguish multiple cases, usually one per data constructor of the datatype of the function argument we analyze.
- We often use recursion when the underlying datatype is recursive (such as lists are).

It is very easy to define new datatypes. New datatypes start out with **no** operations (except pattern matching), but for some classes, we can use `deriving` to obtain instances automatically.

Outlook

In the next part of the course, we take a much more detailed look at how to define functions systematically using pattern matching, and discuss various datatypes.

We also discuss the expression language in more detail.

In the other parts we will discuss:

- parametric polymorphism and overloading,
- higher-order functions and abstraction,
- explicit effects (the `IO`) type,
- more advanced abstraction patterns such as applicative functors and monads.