Introduction to Haskell

Functional programming in Haskell

Ivan Trepakov

NSU Sys.Pro

What is Haskell?

- Pure, lazy and functional programming language
- Designed by a committee of researchers
- Haskell 1.0 Report released 1990
- Haskell 98 Language Report
- Haskell 2010 Language Report (*current standard*)
- Actively developed on top of standard via extensions to Glasgow Haskell Compiler (GHC)
- Most changes in GHC are accompanied by research paper
    - Compiler and language research platform
    - Production-ready compiler and runtime

Installing Haskell toolchain

Official installer GHCup

- GHC (Glasgow Haskell Compiler)
- GHCi — interactive REPL-like environment
- HLS (Haskell Language Server) — integration with editors
- `cabal` and `stack` — tools for package management and development

```
$ ghc --version
The Glorious Glasgow Haskell Compilation System,
version 9.4.8
```

Note: any version 9.x.x or above will be fine

https://www.haskell.org/ghcup/

GHC interactive

Using GHCi

- `:?` — help
- `:quit` or `:q` — quit
- `:load` or `:l` — load module
- `:reload` or `:r` — reload modules
- `:info` or `:i` — information about identifier
- `:type` or `:t` — type of expression
- `:set` / `:unset` — set or unset options

```
$ ghci
GHCi, version 9.4.8:
https://www.haskell.org/ghc/ :? for help
ghci> 2
2
ghci> True
True
ghci> 'a'
'a'
ghci> "Hello"
"Hello"
ghci> [1,2,3]
[1,2,3]
ghci> (12, True)
(12, True)
ghic> :q
Leaving GHCi.
```

Evaluating expressions

Arithmetic

```
ghci> 2 + 3
5
ghci> 2 + 3 * 2
8
ghci> (-2) * 4
-8
ghci> 5.0 / 2.0
2.5
ghci> 5 `div` 2
2
ghci> 5 `mod` 2
1
```

Booleans and comparisons

```
ghci> True && False
False
ghci> True || False
True
ghci> not True
False
ghci> 5 == 2 + 3
True
ghci> 5 /= 2 + 3
False
ghci> True > False
True
```

Operators are functions

```
ghci> (+) 2 3
5
ghci> div 5 2
2
ghci> max 5 2
5
ghci> 5 `max` 2
5
```

Associativity and precedence

Symbolic operators

- Any non-alphanumeric identifier is considered operator and *infix* by default
- But can be made *prefix* by enclosing in parentheses
- Associativity and precedence must be explicitly specified

Alphanumeric functions

- Any alphanumeric identifier is *prefix* by default
- But can be made *infix* by enclosing in backticks
- Function application has highest precedence and always left-associative

```
ghci> 2 + 3 * 2
8
ghci> :i (+)
type Num :: * -> Constraint
class Num a where
  (+) :: a -> a -> a
  ...
    -- Defined in `GHC.Num'
infixl 6 +
ghci> :i (*)
type Num :: * -> Constraint
class Num a where
  ...
  (*) :: a -> a -> a
  ...
    -- Defined in `GHC.Num'
infixl 7 *
```

Associativity and precedence

Symbolic operators

- Any non-alphanumeric identifier is considered operator and *infix* by default
- But can be made *prefix* by enclosing in parentheses
- Associativity and precedence must be explicitly specified

Alphanumeric functions

- Any alphanumeric identifier is *prefix* by default
- But can be made *infix* by enclosing in backticks
- Function application has highest precedence and always left-associative

```
ghci> max 2 3 + 2
4
ghci> (max 2 3) + 2
4
ghci> max 2 (3 + 2)
5
ghci> min 4 (max 2 3)
3
```

Lists and tuples

Lists

- *Homogeneous* linked lists
  - `[]` — empty list
  - `(:)` — constructor "cons"
  - `(++)` — concatenation
- Enumeration notation `[1..10]`

```
ghci> [1,2,3]
[1,2,3]
ghci> []
[]
ghci> 1 : []
[1]
ghci> [3,4] ++ [1,2]
[3,4,1,2]
ghci> 1 : 2 : 3 : []
[1,2,3]
ghci> 1 : 2 : 3 : [] == [1,2,3]
True
ghci> [1..5]
[1,2,3,4,5]
ghci> [1,3..10]
[1,3,5,7,9]
```

Lists and tuples

Lists

- *Homogeneous* linked lists
  - `[]` — empty list
  - `(:)` — constructor “cons”
  - `(++)` — concatenation
- Enumeration notation `[1..10]`

Tuples

- Cartesian product of several types
- Except for pairs should not be used anywhere[1]
  - `fst` and `snd` are only for pairs

```
ghci> (1,2)
(1,2)
ghci> (True,2)
(True,2)
ghci> fst (True,2)
True
ghci> snd (True,2)
2
ghci> (True,[1,2],42)
(True,[1,2],42)
```

[1] Haskell provides better ways via custom data structures

Strings

Strings are lists

- Strings are lists of Unicode characters[1]
- Characters can be enumerated
- Strings can be compared lexicographically
- In real world more efficient implementations are used (see text and bytestring)

[1] Actually Unicode code points

```
ghci> 'a'
'a'
ghci> 'λ'
'\120582'
ghci> putStrLn "λ"
λ
ghci> "abc123"
"abc123"
ghci> ['a','b','c']
"abc"
ghci> 'a' : "bc" == "abc"
True
ghci> ['a'..'f']
"abcdef"
ghci> "Haskell" > "C++"
True
```

More functions

List functions

```
ghci> length "Haskell"
7
ghci> reverse "Haskell"
"lleksaH"
ghci> take 2 "Hello" ++ drop 5 "Haskell"
"Hell"
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> sum (filter even [1..10])
30
ghci> map odd [1..5]
[True,False,True,False,True]
```

More functions

List functions

```
ghci> length "Haskell"
7
ghci> reverse "Haskell"
"lleksaH"
ghci> take 2 "Hello" ++ drop 5 "Haskell"
"Hell"
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> sum (filter even [1..10])
30
ghci> map odd [1..5]
[True,False,True,False,True]
```

Anonymous functions[1]

```
ghci> (\x -> 3 * x + 2) 2
8
ghci> map (\x -> 3 * x + 2) [1..5]
[5,8,11,14,17]
ghci> (\x y -> x + y) 2 3
5
ghci> zipWith (\x y -> x + y) [1..5] [6..10]
[7,9,11,13,15]
ghci> zipWith (+) [1..5] [6..10]
[7,9,11,13,15]
```

[1] Also known as *lambda functions*

Types

Inspecting types in GHCi

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t [True,False]
[True,False] :: [Bool]
ghci> :t (True,'a')
(True,'a') :: (Bool, Char)
ghci> :t ('a',True)
('a',True) :: (Char, Bool)
ghci> :t not
not :: Bool -> Bool
```

- `::` reads as “has type”

Types

Inspecting types in GHCi

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t [True,False]
[True,False] :: [Bool]
ghci> :t (True,'a')
(True,'a') :: (Bool, Char)
ghci> :t ('a',True)
('a',True) :: (Char, Bool)
ghci> :t not
not :: Bool -> Bool
```

- `::` reads as “has type”

Parametric polymorphism

```
ghci> :t reverse
reverse :: [a] -> [a]
ghci> reverse [1,2,3]
[3,2,1]
ghci> reverse "Haskell"
"lleksaH"
ghci> :t fst
fst :: (a, b) -> a
```

- Lower-case identifiers in type signatures are *type variables*
- Concrete types always start with upper-case letter

Types

Currying

- Functions with multiple parameters are always *curried*[1]
  - Accept exactly one argument and return another function
- `->` is *right-associative*, so following type signatures are the same

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

- Allows *partial application* of function to the first argument(s)

```
ghci> :t take
take :: Int -> [a] -> [a]
ghci> :t take 2
take 2 :: [a] -> [a]
ghci> :t take 2 "abc"
take 2 "abc" :: [Char]
ghci> :t map
map :: (a -> b) -> [a] -> [b]
ghci> :t map (take 2)
map (take 2) :: [[a]] -> [[a]]
ghci> map (take 2) ["abc", "def"]
["ab","de"]
```

[1] This idea was first introduced by *Moses Schönfinkel* and then further developed and popularized by *Haskell Curry*

Overloading

- Type variables of polymorphic functions can have additional constraints[1] denoted by `=>` clause
- In that case we say that they are *overloaded*
- Overloaded functions use some specific API provided by those constraints in their implementation
- `Ord` means something *comparable*
- `Num` is any number-like type (`Int`, `Integer`, `Double`)
- `Foldable` is a generalization of any container-like type[2]

```
ghci> :t max
max :: Ord a => a -> a -> a
ghci> max "Haskell" "C++"
"Haskell"
ghci> max 3 5
5
ghci> :t (>)
(>) :: Ord a => a -> a -> Bool
ghci> :t (+)
(+) :: Num a => a -> a -> a
ghci> :t length
length :: Foldable t => t a -> Int
```

[1] Such constraints are called *type classes* and we will encounter them a lot during semester

[2] For now consider it to be simply list type

Types

Built-in types

- Numeric literals are overloaded
- We can explicitly specify type for any expression
- `Int` — fixed-precision integer type
  - Guaranteed to be at least $[-2^{29}, 2^{29} - 1]$[1], but usually is machine word sized
- `Integer` — arbitrary-precision integer type
  - Implemented internally via GNU Multiple Precision Arithmetic Library (GMP)[2]
- `Float` — single-precision floating point type
- `Double` — double-precision floating point type
- `Char` — Unicode code point (character)
- `()` — Unit type

[1]See Haskell 2010 Language Report, Section 6.4 Numbers

[2]See integer-gmp package

```
ghci> :t 2
2 :: Num a => a
ghci> :t maxBound
maxBound :: Bounded a => a
ghci> maxBound
()
ghci> maxBound :: Int
9223372036854775807
ghci> maxBound :: Char
'\1114111'
ghci> 2^100
1267650600228229401496703205376
ghci> 2^100 :: Int
0
ghci> 2^100 :: Integer
1267650600228229401496703205376
```

Types

Explicit effects

- All functions in Haskell are *pure* by default
- *Impure* functions explicitly marked with `IO` type
- `IO ()` represents action that does not yield any result but produces some *side effect*
- Side effects include
  - Interacting with `stdin/stdout`
  - Mutating global program state
  - Reading and writing files
  - Accessing database
  - Sending or receiving TCP/IP requests
- `Show` constraint provides conversion from given type to `String` via `show` function
- Under the hood GHCi uses `print` to show expressions on screen

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> putStrLn "Hello"
Hello
ghci> "Hello"
"Hello"
ghci> :t print
print :: Show a => a -> IO ()
ghci> print "Hello"
"Hello"
ghci> show "Hello"
"\"Hello\""
ghci> print [1,2,3]
[1,2,3]
ghci> show [1,2,3]
"[1,2,3]"
```

Program structure

Modules

- Haskell program consists of *modules*
- Each module corresponds to single `.hs` or `.lhs` (literate Haskell) file
- Each module contains *declarations*:
  - Function declarations (*bindings*)
  - Type signatures
  - Fixity declarations (associativity and precedence of operators)
  - *Type declarations*
  - And many others...
- Order of declarations does not matter

Prelude

- Prelude is an implicitly imported module containing standard function and type declarations
- Most of the functions we have seen so far come from Prelude module
- Very little is actually built into Haskell language itself

https://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html

Functions

Bindings

Examples

Functions

Bindings

- Type signature
  - Optional but recommended
  - Improves type error messages
- Zero or more arguments
  - Binding without arguments is *constant*

Examples

```haskell
e :: Double
e = exp 1

square :: Int -> Int
square x = x * x

squareSum :: Int -> Int -> Int
squareSum x y = square (x + y)

sumSquare :: Int -> Int -> Int
sumSquare x y = square x + square y
```

Functions

Bindings

- Type signature
  - Optional but recommended
  - Improves type error messages
- Zero or more arguments
  - Binding without arguments is *constant*
- `if p then x else y`

Examples

```haskell
max' :: Int -> Int -> Int
max' x y = if x > y then x else y
```

Functions

Bindings

- Type signature
  - Optional but recommended
  - Improves type error messages
- Zero or more arguments
  - Binding without arguments is *constant*
- `if p then x else y`
- Guards

Examples

```haskell
max' :: Int -> Int -> Int
max' x y = if x > y then x else y

max'' :: Int -> Int -> Int
max'' x y
  | x > y     = x
  | otherwise = y
```

Functions

Bindings

- Type signature
  - Optional but recommended
  - Improves type error messages
- Zero or more arguments
  - Binding without arguments is *constant*
- `if p then x else y`
- Guards
  - `otherwise = True`

Examples

```haskell
max' :: Int -> Int -> Int
max' x y = if x > y then x else y

max'' :: Int -> Int -> Int
max'' x y
  | x > y     = x
  | otherwise = y
```

Functions

Bindings

- Type signature
  - Optional but recommended
  - Improves type error messages
- Zero or more arguments
  - Binding without arguments is *constant*
- `if p then x else y`
- Guards
  - `otherwise = True`
- Recursion

Examples

```
fib :: Integer -> Integer
fib n
  | n == 0    = 0
  | n == 1    = 1
  | otherwise = fib (n - 1) + fib (n - 2)
```

Functions

Bindings

- Type signature
  - Optional but recommended
  - Improves type error messages
- Zero or more arguments
  - Binding without arguments is *constant*
- `if p then x else y`
- Guards
  - `otherwise = True`
- Recursion
- Pattern matching
  - Literals

Examples

```
fib :: Integer -> Integer
fib n
  | n == 0    = 0
  | n == 1    = 1
  | otherwise = fib (n - 1) + fib (n - 2)

fib' :: Integer -> Integer
fib' 0 = 0
fib' 1 = 1
fib' n = fib' (n - 1) + fib' (n - 2)
```

Functions

Bindings

- Type signature
  - Optional but recommended
  - Improves type error messages
- Zero or more arguments
  - Binding without arguments is *constant*
- `if p then x else y`
- Guards
  - `otherwise = True`
- Recursion
- Pattern matching
  - Literals

Examples

```haskell
rating :: String -> Int
rating "Haskell" = 10
rating "Scala"   = 8
rating "C"       = 6
rating "C++"     = 2
rating _         = 0
```

Functions

Bindings

- Type signature
  - Optional but recommended
  - Improves type error messages
- Zero or more arguments
  - Binding without arguments is *constant*
- `if p then x else y`
- Guards
  - `otherwise = True`
- Recursion
- Pattern matching
  - Literals
  - *Constructors*

Examples

```haskell
rating :: String -> Int
rating "Haskell" = 10
rating "Scala"   = 8
rating "C"       = 6
rating "C++"     = 2
rating _         = 0

sumPair :: (Int, Int) -> Int
sumPair (x, y) = x + y
```

Functions

Bindings

- Type signature
  - Optional but recommended
  - Improves type error messages
- Zero or more arguments
  - Binding without arguments is *constant*
- `if p then x else y`
- Guards
  - `otherwise = True`
- Recursion
- Pattern matching
  - Literals
  - *Constructors*

Examples

```haskell
rating :: String -> Int
rating "Haskell" = 10
rating "Scala"   = 8
rating "C"       = 6
rating "C++"     = 2
rating _         = 0

sumPair :: (Int, Int) -> Int
sumPair (x, y) = x + y

isEmpty :: [a] -> Bool
isEmpty []     = True
isEmpty (x:xs) = False
```

Factorial

$$n! = 1 \cdot 2 \cdot ... \cdot n$$

Factorial

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

$$F_1(n) = \begin{cases} 1 & n = 0 \\ n \cdot F_1(n-1) & \text{otherwise} \end{cases}$$

Factorial

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

$$F_1(n) = \begin{cases} 1 & n = 0 \\ n \cdot F_1(n-1) & \text{otherwise} \end{cases}$$

```
fact1 :: Integer -> Integer
fact1 n
  | n == 0    = 1
  | otherwise = n * fact1 (n - 1)
```

Factorial

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

$$F_1(n) = \begin{cases} 1 & n = 0 \\ n \cdot F_1(n-1) & \text{otherwise} \end{cases}$$

$$F_2(0) = 1$$
$$F_2(n) = n \cdot F_2(n-1)$$

```
fact1 :: Integer -> Integer
fact1 n
  | n == 0    = 1
  | otherwise = n * fact1 (n - 1)
```

Factorial

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

$$F_1(n) = \begin{cases} 1 & n = 0 \\ n \cdot F_1(n-1) & \text{otherwise} \end{cases}$$

$$F_2(0) = 1$$
$$F_2(n) = n \cdot F_2(n-1)$$

```haskell
fact1 :: Integer -> Integer
fact1 n
  | n == 0    = 1
  | otherwise = n * fact1 (n - 1)

fact2 :: Integer -> Integer
fact2 0 = 1
fact2 n = n * fact2 (n - 1)
```

Factorial

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

$$F_1(n) = \begin{cases} 1 & n = 0 \\ n \cdot F_1(n-1) & \text{otherwise} \end{cases}$$

$$F_2(0) = 1$$

$$F_2(n) = n \cdot F_2(n-1)$$

$$F_3(n) = \prod_{x=1..n} x$$

```
fact1 :: Integer -> Integer
fact1 n
  | n == 0    = 1
  | otherwise = n * fact1 (n - 1)

fact2 :: Integer -> Integer
fact2 0 = 1
fact2 n = n * fact2 (n - 1)
```

Factorial

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

$$F_1(n) = \begin{cases} 1 & n = 0 \\ n \cdot F_1(n-1) & \text{otherwise} \end{cases}$$

```
fact1 :: Integer -> Integer
fact1 n
  | n == 0    = 1
  | otherwise = n * fact1 (n - 1)
```

$$F_2(0) = 1$$

$$F_2(n) = n \cdot F_2(n-1)$$

```
fact2 :: Integer -> Integer
fact2 0 = 1
fact2 n = n * fact2 (n - 1)
```

$$F_3(n) = \prod_{x=1..n} x$$

```
fact3 :: Integer -> Integer
fact3 n = product [1..n]
```

Factorial

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

$$F_1(n) = \begin{cases} 1 & n = 0 \\ n \cdot F_1(n-1) & \text{otherwise} \end{cases}$$

$$F_2(0) = 1$$
$$F_2(n) = n \cdot F_2(n-1)$$

$$F_3(n) = \prod_{x=1..n} x$$

```
fact1 :: Integer -> Integer
fact1 n
  | n == 0    = 1
  | otherwise = n * fact1 (n - 1)

fact2 :: Integer -> Integer
fact2 0 = 1
fact2 n = n * fact2 (n - 1)

fact3 :: Integer -> Integer
fact3 n = product [1..n]
```

[1] Checkout more factorials in "The Evolution of a Haskell Programmer"

Collatz sequence

Collatz function

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod{2} \\ 3n+1 & \text{if } n \equiv 1 \pmod{2} \end{cases}$$

Collatz sequence

Collatz function

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2 \end{cases}$$

Collatz conjecture[1]

A sequence obtained by consecutive application of f to *any* positive number eventually reaches number 1.

$$n, f(n), f(f(n)), ..., 1$$

[1] Wikipedia: Collatz conjecture

Collatz sequence

Collatz function

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2 \end{cases}$$

Collatz conjecture[1]

A sequence obtained by consecutive application of f to *any* positive number eventually reaches number 1.

$$n, f(n), f(f(n)), \ldots, 1$$

[1] Wikipedia: Collatz conjecture

```haskell
collatz :: Integer -> Integer
collatz n
  | even n    = n `div` 2
  | otherwise = 3 * n + 1
```

Collatz sequence

Collatz function

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2 \end{cases}$$

Collatz conjecture[1]

A sequence obtained by consecutive application of f to *any* positive number eventually reaches number 1.

$$n, f(n), f(f(n)), ..., 1$$

[1]Wikipedia: Collatz conjecture

```
collatz :: Integer -> Integer
collatz n
  | even n      = n `div` 2
  | otherwise = 3 * n + 1

collatzSeq :: Integer -> [Integer]
collatzSeq 1 = [1]
collatzSeq n = n : collatzSeq (collatz n)
```

Exercise

Sum consecutive pairs of elements in the list

Exercise

Sum consecutive pairs of elements in the list

```
-- >>> sumPairwise []
-- []
-- >>> sumPairwise [1]
-- [1]
-- >>> sumPairwise [1,2]
-- [3]
-- >>> sumPairwise [1,2,3]
-- [3,3]
```

Exercise

Sum consecutive pairs of elements in the list

```
-- >>> sumPairwise []
-- []
-- >>> sumPairwise [1]
-- [1]
-- >>> sumPairwise [1,2]
-- [3]
-- >>> sumPairwise [1,2,3]
-- [3,3]

sumPairwise :: [Int] -> [Int]
sumPairwise []       = []
sumPairwise [x]      = [x]
sumPairwise (x:y:xs) = (x + y) : sumPairwise xs
```

Q&A