Introduction to Haskell

Functional programming in Haskell

Ivan Trepakov

NSU Sys.Pro

What is Haskell?

What is Haskell?

Functional

- Functions as first-class citizens
- Higher order functions
- Declarative style

What is Haskell?

Functional

- Functions as first-class citizens
- Higher order functions
- Declarative style

Pure

- Side-effect separation
- Equational reasoning
- Simplified parallelism

What is Haskell?

Functional

- Functions as first-class citizens
- Higher order functions
- Declarative style

Pure

- Side-effect separation
- Equational reasoning
- Simplified parallelism

Lazy

- Infinite data structures
- Compositional programming style
- Tricky to evaluate complexity

What is Haskell?

Functional

- Functions as first-class citizens
- Higher order functions
- Declarative style

Pure

- Side-effect separation
- Equational reasoning
- Simplified parallelism

Lazy

- Infinite data structures
- Compositional programming style
- Tricky to evaluate complexity

Statically typed

- “If a program compiles, it probably works”
- Expressive type system
- Type inference

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 above 9.x.x 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 (Haskell provides better ways via custom data structures)
  - `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)
```

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

Basics

Anatomy of declaration

Here is sample Haskell declaration:

```haskell
x :: Int    -- Type declaration
x = 42      -- Value declaration
```

`name = expression` is a *binding* (not assignment)

- `::` reads as “has type”
- `=` reads as “defined to be”

Anatomy of declaration

Here is sample Haskell declaration:

```haskell
x :: Int    -- Type declaration
x = 42      -- Value declaration
```

`name = expression` is a *binding* (not assignment)

- `::` reads as “has type”
- `=` reads as “defined to be”

Multiple declarations with the same name are not allowed!
Compiler will let us know about it with error:

```
Multiple declarations of 'x'
```

Anatomy of declaration

Here is sample Haskell declaration:

```
x :: Int    -- Type declaration
x = 42      -- Value declaration
```

`name = expression` is a *binding* (not assignment)

- `::` reads as “has type”
- `=` reads as “defined to be”

Multiple declarations with the same name are not allowed!

Compiler will let us know about it with error:

```
Multiple declarations of 'x'
```

What does this declaration mean?

And what is its type if any?

```
y = y + 1
```

Anatomy of declaration

Here is sample Haskell declaration:

```haskell
x :: Int    -- Type declaration
x = 42      -- Value declaration
```

`name = expression` is a *binding* (not assignment)

- `::` reads as “has type”
- `=` reads as “defined to be”

Multiple declarations with the same name are not allowed!
Compiler will let us know about it with error:

```
Multiple declarations of 'x'
```

What does this declaration mean?
And what is its type if any?

```haskell
y = y + 1
y :: Int
```

Built-in types

Built-in types

```
-- Fixed-precision integer
i :: Int
i = 12
```

Guaranteed[1] to be at least $[-2^{29}, 2^{29}-1]$, but usually is machine word sized

```
-- Actual bounds
minInt, maxInt :: Int
minInt = minBound
maxInt = maxBound
```

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

Built-in types

```haskell
-- Fixed-precision integer
i :: Int
i = 12
```

Guaranteed[1] to be at least $[-2^{29}, 2^{29} - 1]$, but usually is machine word sized

```haskell
-- Actual bounds
minInt, maxInt :: Int
minInt = minBound
maxInt = maxBound
```

```haskell
-- Arbitrary-precision integer
n :: Integer
n = 2 ^ (2 ^ (2 ^ (2 ^ 2)))

numDigits :: Int
numDigits = length (show n)

-- >>> numDigits
-- 19729
```

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

Built-in types

```
-- Double-precision floatint point
d1, d2 :: Double
d1 = 3.1415
d2 = 6.2831e-4

-- Boolean
b1, b2 :: Bool
b1 = True
b2 = False
```

Built-in types

```haskell
-- Double-precision floatint point
d1, d2 :: Double
d1 = 3.1415
d2 = 6.2831e-4

-- Boolean
b1, b2 :: Bool
b1 = True
b2 = False
```

```haskell
-- Unicode code point (character)
c1, c2, c3 :: Char
c1 = 'A'
c2 = 'λ'
c3 = '🌎'

-- String (list of characters)
s :: String
s = "Hello world! 🌎"
```

Q&A