Week 2

Ch 1: Starting Out
Ch 2: Believe the Type

University of the Fraser Valley

Dr. Russell Campbell

Russell.Campbell@ufv.ca

COMP 481: Functional and Logic Programming

1

Terminology

- referential transparency—guarantees a function returns the same result when called with the same parameter values
- lazy—Haskell will not calculate values until necessary (allows you to make seemingly infinite data structures)
- statically typed—all things are determined at compile time
- type inference—can let Haskell determine what type something belongs to, but can still state the type of something if you want

2

— Math Operations —

3

Operators

Typical addition, subtraction, multiplication, division

- `+, -, *`
- floating point division `/`
- integer division, e.g.:
  `div 10 4` evaluates to `2` and not `2.5`
- `12 mod 7`

Boolean

- `True && False`
- `True || False`
- `not True`

Comparison

- `5 == 5`
- `1 == 0`
- `5 /= 5`
- `"hello" == "hello"`

4

Inline vs Prefix

Take a moment to read the error message when attempting to sum two values of different types, e.g.: `5 + "llama"`.

- the operations written between two values (such as `*`):
- are considered functions with two parameters
- are described as "inline" when written in between its arguments
- can be written in prefix style, i.e.: `(*) 2 3` evaluates to `6`

5

More Math Functions

A few different kinds of functions:

- `succ`
- `pred`
- `min`
- `max`

6

— Lists —

7

List Functions

List functions:

- `head`
- `tail`
- `init`
- `last`
- `length`
- `null`
- `reverse`
- `take`
- `drop`
- `maximum`
- `minimum`
- `sum`
- `product`
- `elem`
- `cycle`
- `repeat`
- `replicate`
- `zip`

8

List Creation with Ranges

Examples:

- `[1..20]`
- `[2,4..20]`
- `[20,19..1]`
- `take 24 [13,26..]`
- `take 10 (cycle [1,2,3])`
- `take 12 (cycle "LOL ")`
- `take 10 (repeat 5)`
- `replicate 3 10`
- watch out with using ranges and floating-point accuracy
- `[0.1, 0.3 .. 1]`

9

List Comprehensions

- `[2*x | x <- [1..10]]`
- you can add predicates after a comma, and have as many predicates as you want to filter your list
- `[2*x | x <- [1..10], odd x]`
- you can have multiple variables, each assigned with a list, so the result is like a cross product
- `[x*y | x <- [3, 5], y <- [2, 4, 6]]`
- can involve more complex expressions
-
```
boombangs xs =
  [if x < 10 then “BOOM!” else “BANG!” | x <- xs, odd x]
```

10

List Access

- use `!!` to access one element using an index
- `[1, 2, 3] !! 0`

- again, strings are also lists of characters:
- `“hello” !! 4`

We can also have lists inside of lists:

- `[ [1,2,3], [4,5,6], [7,8,9] ]`

11

— Tuples —

12

Tuple Types

- different numbers of elements are treated as distinct types
  - so, a 2-tuple is considered different type from a 3-tuple
- for tuples with different types in corresponding elements are altogether different types as well
  - e.g.: `(1, 'a')` different type from `('a', 1)`
- can compare elements of the same type
- cannot compare tuples of different lengths

13

Tuple Functions

- `fst` returns the first element in a pair
- `snd` returns the second element in a pair
  - the above only work on 2-tuples

- `zip` function takes two input lists and creates a list of tuples with corresponding values from the input lists
  - `zip [1..] ["apple", "orange", "cherry", "mango"]`

14

— Chapter 2 —

15

GHCI Multiline

It is just easier to use multiline statements in `ghci` than to write scripts in files for examples that are only a few lines at a time

- use `:{` `:}` for multiline
- use `:set +m` for multiline without braces

Different multiline has different ways to exit:

- for a mistake with `:{` just close with `:}` and start over
- for a `let` multiline, to exit just press `ENTER` twice

Without multiline braces, code must follow layout syntax:

- mostly, just indent the next line to the same column as the variable name on its previous line
- this is only for some expressions that can span multiple lines

16

Tuple Types

Every thing in Haskell has a type determined at compile time:

- you do not need to explicitly declare types
  if there is enough other information for Haskell
- `::` reads as "has type of"
  - e.g.: `(True, 'a') :: (Bool, Char)`
- `:t` lets us check the type of whatever follows the command

17

Common Types

- `` `Int` ``—bounded by min and max values
  - depending on your architecture, for a 64-bit CPU, likely the bounds are -2^{63} and 2^{63}
- `` `Integer` ``—unbounded, but less efficient than `` `Int` ``
- `` `Float` ``—real numbers to single floating-point precision
- `` `Double` ``—real numbers to double floating-point precision
- `` `Bool` ``—only the two expected values `` `True` `` or `` `False` ``
- `` `Char` ``—a single Unicode character
  - `` `[Char]` `` is the same type as the `` `String` `` type
- we saw tuples—their types also depend on their length
  - note that an empty tuple is also a type `` `()` ``
  - tuples can have at most 62 elements, but theoretically there are an infinite number of types

18

Type Variables

The type of a function could involve lists, but lists themselves could contain some arbitrary type:

- so, the declaration will involve a placeholder for the element type, such as `a`
  - e.g.: `:t head;   head :: [a] -> a`
  - e.g.: `:t fst;   fst :: (a, b) -> a`

Note that the type of `fst` function describes the first element in the pair `a` to have the same type as the return value of the `fst` function.

19

Intro to Type Classes

For now, think of a type class as mostly the same concept as an abstract interface in object-oriented programming

- type class declares, but not an implementation for its functions
  - any class belonging to the type class will need to implement function behaviour
- equality is a good example of a function that makes use of type classes in its pattern
  - try `:t (==)`
  - everything before the `=>` is called a class constraint
  - any of the values in place of `a` must be of the `Eq` type class
- for our first example of a type class is `Eq`, of which everything in Haskell is an instance (except for input/output types)

20

Eq & Ord Type Classes

The `Eq` type class supports equality testing.

- so if a function has an `Eq` class constraint for one of its type variables, then that function must implement BOTH `==` and `/=` within its definition
- "definition" means the function's statements of execution

The `Ord` type class is used by types that need arrange their values in some order

  - try `:t (>)`

- the `compare` function takes two input values both with type an instance of `Ord`
  - the return type is `Ordering`
  - `Ordering` is a type with values `GT`, `LT`, `EQ`

21

Show and Ord Type Class

All types we have seen so far except functions are instances of the `Show` type class

- the `show` function will print its input as a string

The `Ord` type class is used by types that need to arrange their values in some order

- try `:t (>)`

22

Read Type Class

All types we have seen so far except functions are instances of the `Read` type class as well.

- the `read` function (inverse of the `show` function)
- `read` takes a String type value as input and returns what value would be expected when used in context

For example: `read "True" || False`

- the above context would expect a value of type `Bool` in place of the `read "True"` expression
- `read "4"` will result in an error, because the expression is not used in any context, so it does not know what type to expect

23

Type Annotations

Now we will need type annotations sometimes when we want to specify the resulting type of an expression.

- append `::` with a corresponding type to the expression
  - e.g.: `read "5" :: Int`
  - e.g.: `(read "5" :: Float) * 4`
- we may avoid an annotation when Haskell can infer the type
  - e.g.: `[read "True", False, True]`

24

`Enum` Type Class

The `Enum` type class describes any type that has values which are totally ordered:

- the advantage is to be able to specify list ranges with `..`

- its `pred` function will return the value that directly precedes its input value in the total order
- its `succ` function will return the next value directly after it input value in the total order

Examples of types in this type class:

- `(), Bool, Char, Ordering, Int, Integer, Float, Double`
- try the above in creating a few lists

25

The `Bounded` Type Class

It is more helpful to look at an example function that uses the `Bounded` type class:

- the `maxBound` function has an interesting type
  - `:t maxBound;  maxBound :: Bounded a => a`
  - the textbook describes this kind of function as polymorphic constants
  - i.e.: `maxBound` has no input parameters, but specifies the return type
- tuples with all element types as instances of `Bounded` are altogether considered an instance of `Bounded` as well

26

The Num Type Class

Look at `:t 20`

- similar type as `maxBound`, but with type class `Num` instead
- so a value such as `20` is also a polymorphic constant
  - "can act like any type that's an instance of the `Num` type class"
  - `Int, Integer, Float, Double`

Try `t: 20 :: Double`

- consider the multiplication operator `:t (*)`
  - accepts two numbers and returns a number of the same type
  - e.g.: `(5 :: Int) * (6 :: Integer)` will cause a type error
  - e.g.: `5 * (6 :: Integer)` has no such error
  - `5` can act like either an `Int` or an `Integer`, but not both at once
- to be an instance of `Num`, a type must also be an instance of `Show` and `Eq`

27

The Floating Type Class

- envelopes the `Float` and `Double` types
- examples of functions to try the type `:t` are
  - `sin`,
  - `cos`, and
  - `sqrt`

28

The `Integral` Type Class

- envelopes the `Int` and `Integer` types
- only whole numbers

An example with more than one class constraint:

```
:t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

- returns a more general type for the same value
- you can use `fromIntegral` to smoothly combine expressions that use mixed numeric types
  - e.g.: `fromIntegral (length [1,2,3,4]) + 3.2`

29

Converting Float Values to Int

To convert a Floating instance value to an Integral one:

- `floor 3.2`
- `ceiling 3.2`

Just consider how these functions work with negative numbers:

- `floor (-3.2)`
- `ceiling (-3.2)`

Try with alternative notation to parentheses:

- `floor $ -3.2`
- `ceiling $ -3.2`

30

— Finishing Up —

31

Final Remarks

- because type classes are essentially abstract interfaces, a type can be an instance of many different type classes
- sometimes a type must be an instance of one type class in order to be an instance of another type class
- e.g.: an instance of `Ord` must first be an instance of `Eq`
- this makes sense from our experience in math
  - comparing two things for ordering
  - we should also be able to check whether two of those things are equal
- we call this implication between type classes a prerequisite

32

Thank You!

Questions?

33