Optionals and typeclasses

FIT2102 Programming Paradigms

S2 2017 Week 7

Preamble

In this week's tutorial we will see better ways to handle undefined behaviours. Last week, we realised that in some cases you cannot reliably define the behaviour of a function.[1]

[1] How do you define the minimum of an empty binary tree? The maximum integer, an error, 0?

One of your tasks, now, will be to write the type of your functions. We will start using polymorphism in our code instead of fixed types. Polymorphism is one of the strong suits of Haskell, allowing users to write a single function that will work on multiple types instead of having to define one per possible type. [2]

[2] By convention, a polymorphic type is called `a`, but any (lower case) letter would work.

Finally, you will learn to define your own *typeclasses*. A way to look at typeclasses is as parent objects, a typeclass in Haskell is a guarantee that a type enforces certain properties. The most common examples are:

- `Show`: enable the display of the element.
- `Read`: allow an object to be created from its string representation.
- `Eq`: enable equality between objects.
- And many more that you can explore online, or see the end.

Reminder

Once you have downloaded and extracted the code, set it up by running:

```
$ stack setup
```

To run, debug, or test the code, enter Haskell's interactive environment: GHCi[3]:

[3] GHCi is a REPL (Read-Eval-Print-Loop), which means that whatever code you type in it is interpreted as if it were source code; loading a file is equivalent to copy/pasting it.

```
$ stack ghci
```

It should load all the source files in the current project. If you modify your code and want to test it, you need to `:load/:l` or `:reload/:r` your code, the former will load the file you indicate, while the latter will reload the whole project:

```
*Main Lib> :l src/Maybes.hs
[1 of 1] Compiling Maybes           ( src/Maybes.hs, interpreted )
Ok, modules loaded: Maybes
*Maybes> :r
```

```
[1 of 1] Compiling Maybes           ( src/Maybes.hs, interpreted )
Ok, modules loaded: Maybes.
```

The interactive environment gives you a number of tools to explore the code, typing `:type/:t` will give you the type of a function; to have more information, you can use `:info/:i`.

```
*Maybes> :t isJust
isJust :: Maybe a -> Bool
*Maybes> :i isJust
isJust :: Maybe a -> Bool       -- Defined at src/Maybes.hs:16:1
```

Using GHCi makes debugging faster as each time you reload your file(s) it will inform you of any syntax or type error.

You can also type code directly within GHCi and use its results either by copy/pasting into your code, or using `it` to recall the last results.

```
Prelude> map (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
Prelude> map (*2) it
[4,6,8,10,12,14,16,18,20,22]
```

Once you have finished writing the code for a file, you can test it using:

```
$ stack exec doctest src/<file>.hs
```

If you want to test the whole project, you can use:

```
$ stack test
```

Exercise 1: Safe Lists

Haskell offers an alternative to the issue of undefined behaviours by providing an *optional value* type: `Maybe`. A `Maybe` is either *just* a value or an empty result.

```
data Maybe a = Nothing | Just a
```

Your task is to implement *safe* functions on lists [4]:

1. A safe version of `head` and `tail` which return an empty result instead of an error if the list is empty.
2. A *better* sum which differentiates empty lists and nil sums.[5]

[4] This time you are also tasked with writing the *type signature* of the functions.

[5] Hint: you cannot just sum any types together, use the `Num` typeclass.

Optional

Rewrite the `minTree` function from last week's binary tree using a `Maybe`.

Exercise 2: Maybes

This exercise is about type manipulation rather than actual computation, your first foray into functional exercises.[6] You need to implement a number of helper functions around `Maybe` constructs.

[6] A general piece of advice in FP: "follow the types."

Question

Can you write a function with the following signature?

```
mystery :: Maybe a -> a
```

Optional (hard)

Write the following function:[7]

[7] Hint: use <$> (fmap) and <*> (apply).

```
-- | Turns a list of @Maybe@ values into @Maybe@ a list of values.
--
-- >>> flipMaybe ljust
-- Just [1,7,3]
--
-- >>> flipMaybe lnaught
-- Nothing
flipMaybe :: [Maybe a] -> Maybe [a]
```

Exercise 3: Rock Paper Scissors

The goal of this exercise is to write typeclasses yourself. A typeclass is a *property* of a type. By default, types in Haskell do not do anything, you cannot even compare two instances of the same type together! However, you can derive a number of typeclasses by default.[8]

[8] E.g., you can derive `Eq` on all sum types without any custom code.

To create an instance of `Ord` it is necessary to define a *complete definition* using either: `<=` or `>=` but it requires `Eq`; or use `compare`. The latter is the most compact way to define the ordering of a type, it has the following definition:

```
data Ordering =  EQ | LT | GT
compare :: Ord a => a -> a -> Ordering
```

You will implement a (simple) game of Rock-Paper-Scissors, first define the necessary typeclasses then two functions:

1. `whoWon` takes two hands and return a results.

2. `competition` takes two series of hands and a number, and return whether a player has won more than `n` times.

   An important feature of this exercise is to try to write *elegant* code.[9]

[9] Elegant code leverages functional constructs as much as possible, think: `map`, `filter`, etc.

Optional

Generalise `competition` to return which player won.

```
competition' :: [RockPaperScissors] -> [RockPaperScissors] -> Result
```

Appendix

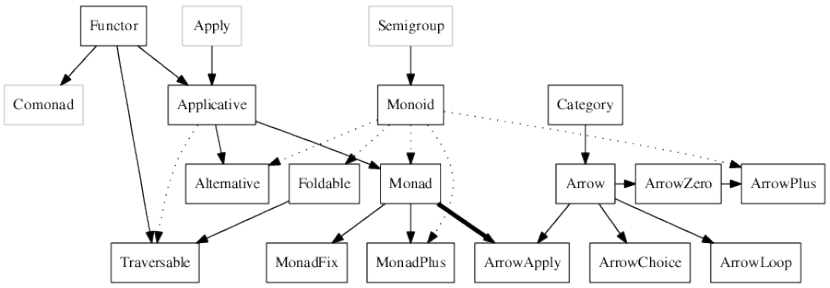

Figure 1: Typeclasses diagram