COMP105 Lecture 21

Parameterised Custom Types

Outline

Today

- Parameterized custom types
- The `Maybe` type
- The `Either` type
- `case` expressions

Relevant book chapters

- Programming In Haskell Chapter 8
- Learn You a Haskell Chapter 8

Recap: Custom Types

```
data Point = Point Int Int deriving(Show, Read, Eq)
```

```
ghci> Point 1 2
Point 1 2
```

```
ghci> Point 1 2 /= Point 3 4
True
```

```
ghci> read "Point 1 1" :: Point
Point 1 1
```

Type Variables in Custom Types

We can use **type variables** in custom types

```
data Point a = Point a a
```

```
ghci> :t Point (1::Int) (2::Int)
Point Int
```

```
ghci> :t Point "hello" "there"
Point [Char]
```

Type Variables in Custom Types

We can use **multiple** variables in the same type

```
data Things a b c = Things a b c   deriving(Show)
```

```
ghci> Things "string" 1 True
Things "string" 1 True
```

```
ghci> Things [] 1.5 'a'
Things [] 1.5 'a'
```

Type Variables in Custom Types

We can write **functions** using these types

```
first_thing (Things x _ _) = x
```

```
ghci> first_thing (Things 1 2 3)
1
```

```
ghci> :t first_thing
first_thing :: Things a b c -> a
```

The Maybe type

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

```
ghci> :t Just "hello"
Maybe [Char]
```

```
ghci> :t Just False
Maybe Bool
```

```
ghci> :t Nothing
Maybe a
```

The Maybe type

The Maybe type is used in pure functional code **that might fail**

```
safe_head [] = Nothing
safe_head (x:_) = Just x
```

```
ghci> safe_head [1,2,3]
Just 1
```

```
ghci> safe_head []
Nothing
```

Case expressions

`case` expressions can do pattern matching **in functions**

```
head_or_zero list =
    let
        h = safe_head list
    in
        case h of Just x  -> x
                  Nothing -> 0
```

```
ghci> head_or_zero [1,2,3]
1
```

Case expressions

The **syntax** for a case expression is

```
case [expression] of [pattern 1] -> [expression]
                     [pattern 2] -> [expression]
                     ...
                     [pattern k] -> [expression]
```

You can use _ (the wildcard) as a catch-all

Case expressions

You can write all the patterns on **one line**

```
case h of {Just x  -> x; Nothing -> 0}
```

Case is an **expression**

```
ghci> (case 1 of 1 -> 1) + (case 2 of 2 -> 1)
2
```

Maybe example

```
safe_get_heads list =
    let
        mapped = map safe_head list
        filtered = filter (/=Nothing) mapped
        unjust = (\ x -> case x of Just a -> a)
    in
        map unjust filtered
```

```
ghci> safe_get_heads [[], [1], [2,3]]
[1,2]
```

Exercise

What do these functions do?

```
mystery x 0 = Nothing
mystery x y = Just (div x y)
```

```
mystery2 x = case x of Just _  -> False
                       Nothing -> True
```

```
mystery3 (Just x) (Just y) = Just (x+y)
mystery3  _         _      = Nothing
```

Exceptions in Haskell

Haskell does include support for **exceptions**

```
ghci> head []

*** Exception: Prelude.head: empty list
```

Exceptions are **not** pure functional

- Every function returns exactly one value
- You can't catch exceptions in pure functional code
- Exceptions are mostly used in IO code

Exceptions in Haskell

The `Maybe` type provides a way to do **exception-like** behaviour in pure functional code

Can this function fail for some inputs?

- use the `Maybe` type

Exceptions should only be used in **IO** code

- File not found, could not connect to server, etc.
- These are unpredictable events

The Either type

```haskell
data Either' a b = Left a | Right b
```

```
ghci> :t Left 'a'
Either Char b
```

```
ghci> :t Right 'b'
Either a Char
```

The Either type

The either type is useful if you want to store **different types** in the same list

```
ghci> let list = [Left "one", Right 2,
                                  Left "three", Right 4]
```

```
is_left (Left _) = True
is_left _        = False
```

```
ghci> map is_left list
[True,False,True,False]
```

The Either type

```
get_lefts list =
    let
        filtered = filter is_left list
        unleft = (\ (Left x) -> x)
    in
        map unleft filtered
```

```
ghci> get_lefts list
["one","three"]
```

Example: squaring mixed number types

```
ghci> let nums = [Left pi, Right (4::Int), Left 2.7182]
```

```
square (Left x)  = Left (x ** 2)
square (Right x) = Right (x ^ 2)
```

```
ghci> map square nums
[Left 9.86,Right 16,Left 7.38]
```

Meaningful error messages

Either can be used to give **detailed errors**

```
safe_head_either []     = Right "empty list"
safe_head_either (x:_) = Left x
```

```
ghci> safe_head_either []
Right "empty list"

ghci> safe_head_either [1,2,3]
Left 1
```

Exercise

What are the **types** of these functions?

```
is_nothing x = case x of Just _  -> False
                         Nothing -> True
```

```
second_thing (Things _ y _) = y
```

```
half (Left x)  = Left  (x / 2)
half (Right x) = Right (x `div` 2)
```

Exercises

1. Create a type `FourThings` that has a single constructor with four parameters, each of which can be any type.

2. Write a function `middle_two` that takes one input of type `FourThings` and returns a tuple containing the middle two elements.

3. Write a type annotation for `middle_two`. Check that it compiles. Comment out your annotation, and use the `:t` command in ghci to see if your annotation was the most general.

4. Create a type `ThreeSameThings` that has a single constructor with three parameters, each of which has the same type.

Exercises

5. Write a function `safe_tail :: [a] -> Maybe [a]` that returns the tail of the list if the list is non-empty, and `Nothing` otherwise.

6. Write a function `safe_tails :: [[a]] -> [Maybe [a]]` that takes a list of lists, and returns a list containing the output of `safe_tail` for each element of the input list.

7. Use `safe_tails` to write a function `tails :: Eq a => [[a]] -> [[a]]`, which returns the tail of each non-empty list from the input.

Exercises

8. Use a case expression to write a function `second_head :: [a] -> a` that returns the second element of a list

9. Use a case expression and `safe_tail` to write a function `is_empty :: [a] -> Bool` that returns True if the input list is empty.

10. Write a function `one_over :: [Float] -> Maybe Float` which computes the sum of $1/x$ for every element x in the input list. But, if there is a zero in the input list, the function should return `Nothing`

Summary

- Parameterized custom types
- The `Maybe` type
- The `Either` type
- `case` expressions

Next time: Recursive custom types