M21274 – MATHFUN
Discrete Mathematics and Functional Programming

Worksheet 7: Algebraic Types

Introduction

This worksheet aims to give you practice in defining algebraic types and functions that compute over them. Start by saving `Week7.hs` to your usual folder and read it through. `Week7.hs` includes some definitions from the lecture. Experiment with these definitions. For example, get the area of a `Rectangle` with:

```
area (Rectangle 10 20)
```

Worked example

Exercise: Your task is to write a program that stores and processes the details of a collection of cars. The data we store for each car is the name (make & model), details of the engine (petrol, diesel, or electric and its horsepower), and price.

Start by defining an appropriate data structure for the scenario. Then write the following functions defined over your type:

1. `totalPrice` that returns the total price of a list of cars;
2. `filterByMake` that filters a list of cars by a manufacturer;
3. `updatePriceAt` that updates the price of a car in the list at a specified index.
4. `formatCar` that takes an index and returns a formatted string representation of that car, for example:

   ```
   "3- Vauxhall Corsa costs 8000.00 pounds"
   ```

Solution: Let's start with some **type synonyms**. Add the following to `Week7.hs`.

```
type Make = String
type Model = String
type HorsePower = Int
type Price = Float
```

These are simply convenient nicknames that we give to existing types. For example, a function that has an input type of `Price` receives a `Float` as an argument.

The type of engine can be one of 3 options. Therefore, we define `EngineType` using a enumerated data type:

```
data EngineType = Petrol | Diesel | Electric
  deriving (Show)
```

Add the above lines to `Week7.hs`. The last line adds our data type to the `Show` class (which will allow us to display `EngineType` values). If you want to compare two values of this type as well, you need to also derive from the `Eq` class by adding the highlighted text to the deriving line:

```
  deriving (Show, Eq)
```

Load `Week7.hs` into GHCi. Test it by creating an `EngineType` and checking its type:

```
:t Petrol
```

We can now define the type `Engine` as a product data type consisting of an `EngineType` value and a value of type `HorsePower`. Add the following to `Week7.hs`:

```
data Engine = Engine EngineType HorsePower
  deriving (Show)
```

Reload the script and check the type of the following expression:

```
:t Engine Petrol 55
```

Similarly, `CarName` can be a product type. Add the following to `Week7.hs`:

```
data CarName = CarName Make Model
  deriving (Show)
```

Note that the `CarName` after the = here is the **constructor** of the data type `CarName`. (Note that the constructor does not need to have the same name as the data type.) Run the following where we are creating a value of type `CarName` and checking its type with the '`:t`' command:

```
:t CarName "Ford" "Fiesta"
```

We can now create a data type `Car` as a product type that uses `CarName`, `Engine` and `Price` (`Float`) as parameters. Add this to `Week7.hs`:

```
data Car = Car CarName Engine Price
  deriving (Show)
```

Reload your script and test it by checking the type of a `Car` value:

```
:t Car (CarName "Ford" "Fiesta") (Engine Petrol 55) 10000.0
```

Here is a list of `Car` values over which we can write our functions. Add this to `Week7.hs`:

```
testCars :: [Car]
testCars =
  [ Car (CarName "Ford" "Fiesta") (Engine Petrol 55) 10000.0,
    Car (CarName "Ford" "Focus") (Engine Diesel 85) 15000.0,
    Car (CarName "Vauxhall" "Corsa") (Engine Petrol 55) 8000.0,
    Car (CarName "Vauxhall" "Astra") (Engine Diesel 81) 12000.0,
    Car (CarName "Vauxhall" "Astra") (Engine Diesel 96) 14000.0,
    Car (CarName "VolksWagen" "Golf") (Engine Electric 81) 20000.0
  ]
```

Before we start with the required functions, let us define some helper functions. Add the following to `Week7.hs`. Note the use of wildcards (to ignore what we are not interested in):

```
getMake :: Car -> Make
getMake (Car (CarName make _) _ _) = make

getModel :: Car -> Model
getModel (Car (CarName _ model) _ _) = model

getPrice :: Car -> Price
getPrice (Car _ _ price) = price
```

Let's start with the `totalPrice` function. We use recursion with pattern matching on the `Car` value at the head of the list. Notice that we don't care about the cars' names and engines when summing up their prices, so we use wildcards for these parts of the pattern. Note the position of the parentheses: since the constructor `Car` is actually a kind of function, it has higher precedence than the ':' operator and so we don't need parentheses around the '`Car _ _ price`' expression.

```
totalPrice :: [Car] -> Float
```

```haskell
totalPrice [] = 0
totalPrice (Car _ _ price : cs) = price + totalPrice cs
```

Test it by evaluating the following expression (it should return 79000.0):

```
totalPrice testCars
```

For `filterByMake`, we can use a list comprehension to return a filtered list:

```haskell
filterByMake :: String -> [Car] -> [Car]
filterByMake manufacturer cs = [c | c <- cs, getMake c == manufacturer]
```

You can write a more elegant definition using a `filter` and a lambda:

```haskell
filterByMake manufacturer = filter (\c -> getMake c == manufacturer)
```

Add the definition that you prefer to `Week7.hs` and check your function by evaluating:

```
filterByMake "Ford" testCars
```

For `updatePriceAt`, we will use a helper function `updatePrice` that updates the price of a car. Here is a recursive definition of `updatePriceAt`:

```haskell
updatePriceAt :: Int -> Float -> [Car] -> [Car]
updatePriceAt _ _ [] = []
updatePriceAt 0 amount (c : cs) = updatePrice amount c : cs
updatePriceAt index amount (c : cs) = c : updatePriceAt (index - 1) amount cs

updatePrice :: Float -> Car -> Car
updatePrice newPrice (Car name engine _) = Car name engine newPrice
```

As an alternative, we can use the list index operator (`!!`) to access an element in a list. Next you can stick the pieces of the list back together using the functions `drop` and `take`:

```haskell
updatePriceAt index price cars = take index cars ++ [newCar] ++ drop (index + 1) cars
  where
    newCar = updatePrice price (cars !! index)
```

Test `updatePriceAt` by updating the price of the Vauxhall Corsa:

```
updatePriceAt 2 9999.9 testCars
```

For `formatCar`, we are going to use the `printf` function to format different values of a Car (e.g., to give the price to 2 decimal places). First, add the following import at the top of `Week7.hs`

```haskell
import Text.Printf (printf)
```

Now add `formatCar` shown below to `Week7.hs`. Note the use of index operator (`!!`) and refer to the above link to the documentation page to understand the first parameter of the `printf` function:

```haskell
formatCar :: [Car] -> Int -> String
formatCar [] _ = ""
formatCar cars i = printf "%d- %s %s costs %.2f pounds" (i+1) (getMake c) (getModel c)
    (getPrice c)
  where
    c = cars !! i
```

Programming exercises

Enumerated types

1. Define an algebraic type `Month` that represents the twelve months of the year and `Season` that represents the four seasons. See the test cases for exercise 2.

2. Define a function `season` that maps months onto (meteorological) seasons. See the test cases showing the outputs for February and March. (All seasons are all three months long).

Test	Output
`season February`	`Winter`
`season March`	`Spring`

   Try to make your definition as short as possible.

3. Define a function `numberOfdays` that takes a month and a year and returns the number of days in that month. Assume all years divisible by four are leap years.

Test	Output
`numberOfDays February 2023`	28
`numberOfDays February 2024`	29
`numberOfDays May 2023`	31

Points and Shapes

4. Define an algebraic type `Point` which should represent the coordinates of points in two-dimensional space.

5. Using `Shape` and `Point`, define a data type called `PositionedShape` which combines a shape and with its centre point.

6. Define a function `move` with the signature below that moves a given `PositionedShape` by the given x and y distances.

```
move :: PositionedShape -> Float -> Float -> PositionedShape
```

   Note that you will need to add `deriving (Show)` to the data types from exercises 4 and 5.

Functions for the binary tree type

7. `Week7.hs` defines a recursive type `Tree` for a binary tree and a `testTree` value of this type. Define a function `numberOfNodes` that returns the number of nodes in a given tree.

Test	Output
`numberOfNodes testTree`	7

8. Define a function with the signature below that determines if a value exists in a tree.

```
isMember :: Int -> Tree -> Bool
```

Test	Output
`isMember 7 testTree`	`True`
`isMember 5 testTree`	`False`

9. Define a function `leaves` with the signature below that returns the list of all the leaves of the tree. Leaves are nodes that have `Null` as both subtrees (see the test case).

```
leaves :: Tree -> [Int]
```

Test	Output
`leaves testTree`	`[12,7,6]`

10. Define a function `inOrder` that lists the elements of a tree according to the **in-order** traversal.

Test	Output
`inOrder testTree`	`[12,3,7,20,6,4,8]`

If the tree is a valid **binary search tree**, this function will give a list of the tree's elements in ascending numerical order. Try it on the `testBinaryTree` below.

```
testBinaryTree = Node 5 (Node 1 Null Null) (Node 8 (Node 7 Null Null) Null)
```

11. Define a function `insert` which inserts a new value into a tree. The function should assume that the tree is a binary search tree and should preserve this property.

```
insert :: Int -> Tree -> Tree
```

Test your function by inserting 2 to `testBinaryTree` defined in exercise 10. It should produce the following tree:

```
Node 5 (Node 1 Null (Node 2 Null Null)) (Node 8 (Node 7 Null Null) Null)
```

If you struggle with the concept, try this visualisation tool in your browser and insert the following values one after another: 5, 1, 7, 8, 2.

12. Define a function `listToSearchTree` which creates a binary search tree from a list of integers. You need to insert elements in the order that they appear in the list.

Test	Output
`listToSearchTree [2,1,3]`	`Node 2 (Node 1 Null Null) (Node 3 Null Null)`

Finally, using `listToSearchTree` and `inOrder`, write another function `binaryTreeSort` that sorts a lit of integers.