Data Model 03

Thumrongsak Kosiyatrakul
tkosiyat@pitt.edu

A Little Bit about Data Types

- A `tuple` can be used to group a fixed number of components of different types
- A `list` can be used to group an unlimited number of elements of a homogeneous type
- Suppose we want to store in information about a client:
  - Name: Luke
  - Age: 19
  - Vehicle Bought: X-Wing Fighter and X-34 Landspeeder
- One option is as follows:

```
("Luke", 19, ["X-Wing Fighter", "X-34 Landspeeder"])
```

- Without the knowledge about the data and the client, the above expression may have the following meaning:
  - Type of Bird: Luke
  - Wingspan: 19 inches
  - Habitats: X-Wing Fighter and X-34 Landspeeder

Algebraic Data Type (ADT) in Haskell

- We need to introduce a new data type to represent a client
- An ADT in Haskell is defined by two pieces of data:
  - A **name** of the type that will be used to represent its values
  - A **set of constructors**, with zero or more arguments of specified types, that will be used to create new values
- In Haskell, different constructors are used to represent completely different alternatives to construct values
- A Java class can have multiple constructors but they share the same name

Algebraic Data Type (ADT) in Haskell

- Suppose we have three kinds of clients:
  1. Government organizations which are known by their name
  2. Companies with the following information:
     - Name
     - Identification number
     - Contact person
     - Contact person's position within the company
  3. Individual clients with the following information
     - First name
     - Last name
     - Whether they want to receive notifications
- A way to represent these client types in Haskell is as follows:

```
data Client = GovOrg     String
            | Company    String Integer String String
            | Individual String String Bool
```

Algebraic Data Type (ADT) in Haskell

- Again, the data type client:

```
data Client = GovOrg     String
            | Company    String Integer String String
            | Individual String String Bool
```

- The syntax:
  - Starts with the `data` keyword
  - Followed by the type name
  - Followed by a list of constructors separated by `|`
  - Each constructor starts with a constructor name and zero or more types of the arguments
- **Note**: Type names and constructors must start with an **uppercase** letter

ADT

- As usual, put the expression about the data type `Client` and a file, load it into `GHCi`, and test it:

```
ghci> :t GovOrg "First Order"
GovOrg "First Order" :: Client
ghci> :t Company "Jedi" 324 "Yoda" "Master"
Company "Jedi" 324 "Yoda" "Master" :: Client
ghci> :t Individual "Luke" "Skywalker" False
Individual "Luke" "Skywalker" False :: Client
```

- Note that they all have the same type `Client`
- Suppose we just enter one of the above expressions (without the command `:t`)

```
ghci> Individual "Luke" "Skywalker" False

<interactive>:53:1: error:
    • No instance for (Show Client) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
```

- The interpreter does not know how to print the above value on the screen

- Haskell provide the **automatic deriving** facility that allows you to add some functionality to your ADT
- To get a `String` representation of the values, we need to derive `Show` as follows:

```
data Client = GovOrg     String
            | Company    String Integer String String
            | Individual String String Bool
            deriving Show
```

- After adding `deriving Show`

```
ghci> Individual "Luke" "Skywalker" False
Individual "Luke" "Skywalker" False
```

- `Show` is a type class
- We will discuss later how to implement a string representation ourselves instead of using automatic deriving

ADT

- Note that a contact person in a company is also an individual person
- We can create a new ADT for persons and use it in the data type `Client`

```
data Client = GovOrg     String
            | Company    String Integer Person String
            | Individual Person Bool
            deriving Show
data Person = Person String String
            deriving Show
```

- Notes:
    - A data type name can be the same as a constructor name
    - **Convention**: If a type only contains one-alternative value, use the type name as the constructor name
    - Since `Client` derives `Show` and use `Person`, `Person` must derive `Show` as well
    - Two different types cannot share the same constructor name

Enumerate Types in Haskell

- The following is an example of an enumerate type:

```
data Gender = Male
            | Female
            | Unknown
            deriving Show
```

- Here are some tests:

```
ghci> :t Person "Luke" "Skywalker"
Person "Luke" "Skywalker" :: Person
ghci> :t Individual (Person "Luke" "Skywalker") False
Individual (Person "Luke" "Skywalker") False :: Client
ghci> Individual (Person "Luke" "Skywalker") False
Individual (Person "Luke" "Skywalker") False
```

```
ghci> :t Female
Female :: Gender
ghci> Unknown
Unknown
```

Pattern Matching

- Recall our data type `Client` and `Person`

```
data Client = GovOrg     String
            | Company    String Integer Person String
            | Individual Person Bool
            deriving Show
data Person = Person String String
            deriving Show
```

- Given a value of type `Client`, suppose we want to extract the name of a client which can be as follows:
  - the name of a government organization
  - the name of a company
  - the name of an individual person
- We need to create a function to extract the name which can be in one of the three different alternatives
- Each alternative has its own **unique** pattern
- A simple way to handle this in most functional programming is **pattern matching**

Pattern Matching

- Let's look at one of the styles, the `case` keyword

```
clientName :: Client -> String
clientName client = case client of
                          GovOrg name                        -> name
                          Company name id person position -> name
                          Individual person ads              ->
                             case person of
                                Person fName lName ->
                                       fName ++ " " ++ lName
```

- What would happen if we execute the following:

```
clientName (Individual (Person "Luke" "Skywalker") False)
```

  - The system could not match the given client with the first two patterns since the constructor is not the same
  - The system match it with the third pattern and bind the following:
    - `person` to `Person "Luke" "Skywalker"`
    - `ads` to `False`
  - The pattern of the binding `person` is matched with the first case and bind the following:
    - `fName` to `"Luke"`
    - `lName` to `"Skywalker"`
  - Finally, it is evaluated to `"Luke Skywalker"`

Pattern Matching

- Again, the function `clientName`

```
clientName client = case client of
                      GovOrg name                     -> name
                      Company name id person position -> name
                      Individual person ads           ->
                         case person of
                            Person fName lName ->
                                   fName ++ " " ++ lName
```

- There are a lot of unused bindings:
  - `id`, `person`, and `position` in the pattern started with `Company`
  - `ads` in the pattern started with `Individual`
- We can replace unused bindings by underscores (_)

```
clientName client = case client of
                      GovOrg name         -> name
                      Company name _ _ _  -> name
                      Individual person _ ->
                         case person of
                            Person fName lName ->
                                   fName ++ " " ++ lName
```

Pattern Matching

- Instead of using the `case` keyword for the last pattern, we can also match inner part of pattern as well

```
clientName client = case client of
                      GovOrg name            -> name
                      Company name _ _ _     -> name
                      Individual (Person fName lName) _ ->
                         fName ++ " " ++ lName
```

- Again, we need to test:

```
ghci> :t clientName
clientName :: Client -> String
ghci> clientName (GovOrg "First Order")
"First Order"
ghci> clientName (Company "Jedi" 342 (Person "Minch" "Yoda") "Master")
"Jedi"
ghci> clientName (Individual (Person "Luke" "Skywalker") False)
"Luke Skywalker"
```

Non-exhaustive

- Suppose we want to create a function to extract just the company name:

```
companyName :: Client -> String
companyName client = case client of
                        Company name _ _ _ -> name
```

- In some interpreters, you may get a warning about non-exhaustive
- Obviously, we will get an exception for some patterns:

```
ghci> companyName (Company "Jedi" 342 (Person "Minch" "Yoda") "Master")
"Jedi"
ghci> companyName (GovOrg "First Order")
"*** Exception: Pattern01.hs:(18,22)-(19,50): Non-exhaustive
                                               patterns in case
```

- The above is an example of a **partial** function
- If possible, return a default value:

```
companyName :: Client -> String
companyName client = case client of
                        Company name _ _ _ -> name
                        _                  -> "Unknown"
```

Non-exhaustive

- The default value `"Unknown"` may work in some situation if we do not actually have a company named `"Unknown"`
- Haskell provides a parameterized type `Maybe a`
  - `Maybe Integer`, `Maybe String`, `Maybe [Int]`
  - There are two kinds of values
    - `Nothing` with no argument
    - `Just v` with a single value `v` of a specific type
- Here is an example using `Maybe`

```
companyName :: Client -> Maybe String
companyName client = case client of
                        Company name _ _ _ -> Just name
                        _                  -> Nothing
```

- In the above example, `Nothing` behaves almost like `null` in Java which can be checked:

```
ghci> companyName (GovOrg "First Order")
Nothing
ghci> Nothing == companyName (GovOrg "First Order")
True
```

Constants are Patterns

- Consider Fibonacci and factorial:

```
fibonacci :: Integer -> Integer
fibonacci n = case n of
                0 -> 0
                1 -> 1
                _ -> fibonacci (n - 1) + fibonacci (n - 2)

factorial :: Integer -> Integer
factorial n = case n of
                0 -> 1
                _ -> n * factorial (n - 1)
```

- Some tests:

```
ghci> fibonacci 10
55
ghci> factorial 10
3628800
```

No Backtracking

- Once a pattern is matched, no backtracking
- Consider the following functions:

```
f :: Client -> String
f client = case client of
             Company _ _ (Person name _) "Boss" -> name ++ " is the boss"
             _                                  -> "There is no boss"

g :: Client -> String
g client = case client of
             Company _ _ (Person name _) pos ->
                case pos of
                   "boss" -> name ++ " is the boss"
             _                               -> "There is no boss"
```

- Some tests:

```
ghci> f (Company "Jedi" 342 (Person "Obi-Wan" "Kenobi") "Master")
"There is no boss"
ghci> g (Company "Jedi" 342 (Person "Obi-Wan" "Kenobi") "Master")
"*** Exception: Pattern01.hs:(46,17)-(47,51): Non-exhaustive patterns in case
```

- **Pattern matching does not backtrack**

Another Style of Pattern Matching

- This style is allowed in Haskell

```
clientName :: Client -> String
clientName (GovOrg name)                           = name
clientName (Company name _ _ _)                    = name
clientName (Individual (Person fName lName) _)     = fName ++ " " ++ lName
```

```
fibonacci :: Integer -> Integer
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n =  n * factorial (n - 1)
```

Pattern Matching with Lists

- List constructors are `[]` and `:`
- Our version of concatenation can be defines as???

```
(+++) :: [a] -> [a] -> [a]
lst1 +++ lst2 = case lst1 of
                     []     -> lst2
                     (x:xs) -> x : (xs +++ lst2)
```

  or

```
(+++) :: [a] -> [a] -> [a]
[] +++ lst2      = lst2
(x:xs) +++ lst2 = x : (xs +++ lst2)
```

- Check whether a list of integers is sorted???

```
sorted :: [Integer] -> Bool
sorted [] = True
sorted [_] = True
sorted (x:y:ys) = x < y && sorted (y:ys)
```

- Another version of our `maxmin`???

```
maxmin [x] = (x, x)
maxmin (x:xs) = (if x > xs_max then x else xs_max,
                 if x < xs_min then x else xs_min)
                where (xs_max, xs_min) = maxmin xs
```