MATHFUN Lecture FP7
Algebraic Types

Matthew Poole
moodle.port.ac.uk

School of Computing
University of Portsmouth

2014/15

Introduction to Lecture

- Up to this point, we have been able to use Haskell to define:
  - functions;
  - modules; and
  - type synonyms (using the `type` keyword).
- However, every modern programming language needs a mechanism for defining completely **new types**:
  - in object-oriented languages, classes define types;
  - in Haskell, we can define our own **algebraic types**.
- Here we see how algebraic types are defined, and give examples.
- This lecture is supported by Chapter 14 of Thompson's book.

Defining types in Haskell

- So far we have seen the following data types in Haskell:
  - the basic types; e.g., `Int`, `Float`, `Bool`, and `Char`;
  - tuple types; for example `(Int, Int, Char)`; and
  - list types; for example `[Int]` and `[(Int, Char)]`.
- We have also seen that we can give convenient names to types using type synonyms; for example:

```
type HouseNumber = Int
type StreetName = String
type Address = (HouseNumber, StreetName)
```

- However, more complex structures (e.g. binary trees) are difficult to model using these types.
- Algebraic types allow us to define arbitrarily complex types.

Algebraic Types

- Algebraic types are introduced with the keyword `data` followed by (both beginning with capital letters):
  - the name of the type being defined; and
  - a list of **constructors** (with their argument types).
- The simplest algebraic types are those where the constructors don't take any arguments; for example:

```
data Day = Mon | Tue | Wed | Thur | Fri | Sat | Sun
```

- Here, constructors are the data values, or members of the type.
- We call a type defined in this way an **enumerated type**.
- Haskell's Boolean type can be defined as an enumerated type:

```
data Bool  =  False | True
```

Functions on enumerated types

- The simplest way to define a function on an enumerated type is by pattern matching.
- (In an earlier lecture we defined the Boolean operators && and || in this way.)
- For the Day data type, an example function is:

```
isWeekend :: Day -> Bool
isWeekend Sat  = True
isWeekend Sun  = True
isWeekend _    = False
```

- Note that our Day data type doesn't (yet) include any (clearly useful) operators such as ==.
- We could define == ourselves but this would be tedious—why?

Algebraic types and type classes

- We can ask Haskell to provide an == operator by declaring that we want our type `Day` to be a member of the `Eq` **type class**.
- We'll see more about type classes in Lecture FP10; for now, we need to know that `Eq` includes all types that include == and /=.
- To do this, we need to define `Day` as follows:

```
data Day = Mon | Tue | Wed | Thur | Fri | Sat | Sun
           deriving (Eq)
```

- Haskell provides us with the 'obvious' implementations of == and /= and we can now, if we wished, redefine `isWeekend` by:

```
isWeekend day  =  day == Sat || day == Sun
```

Algebraic types and type classes

- We can include an type such as `Day` in several other type classes to automatically add other useful functions and operators:
  - `Ord` to provide us with <, <=, > and >=;
  - `Show` to provide a function `show :: Day -> String` that converts `Day` values into strings;
  - `Read` to provide a function `read :: String -> Day` to convert strings (like `"Mon"`) into `Day` values.
- Typically then, we might write:

```
data Day = Mon | Tue | Wed | Thur | Fri | Sat | Sun
           deriving (Eq,Ord,Show,Read)
```

and we are now able to redefine `isWeekend` thus:

```
isWeekend day  =  day >= Sat
```

Product types

- In an earlier lecture, we defined student "records" (names and marks) using a type synonym giving a name to a tuple:

```
type StudentMark = (String, Int)
```

- An alternative is to use an algebraic type with a constructor that has two arguments, one for the name, and one for the mark:

```
data StudentMark = Student String Int
```

- Here, the constructor `Student` is followed by its argument types.
- Example values of the **product** type `StudentMark` are:

```
Student "Sam" 44
Student "Jill" 64
```

Product types

- An example function defined on this data type is:

```
betterStu :: StudentMark -> StudentMark -> String
betterStu (Student s1 m1) (Student s2 m2)
    | m1 >= m2              = s1
    | otherwise             = s2
```

- For example,

```
ghci> betterStu (Student "Sam" 44) (Student "Jo" 73)
"Jo"
```

- The main advantage of using an algebraic type is that every data value has an explicit label of its purpose (e.g. `Student`).

Alternatives

- Combining the ideas of enumerated types and product types leads to types whose elements can be built in different ways.
- For example, a shape might be:
  - a circle, specified by its radius; or
  - a rectangle, specified by its height and width.
- This can be represented by the following type:

```
data Shape = Circle Float |
             Rectangle Float Float
```

- Example values of type `Shape` are:

```
Circle 9.0
Rectangle 4.5 6.0
```

Alternatives

- A function for the `Shape` data type, using pattern matching:

```
area :: Shape -> Float
area (Circle r)      = pi * r * r
area (Rectangle h w) = h * w
```

- As another example, consider the problem of representing addresses: some buildings have numbers; others have names.
- A data type for representing (the first line of) an address is:

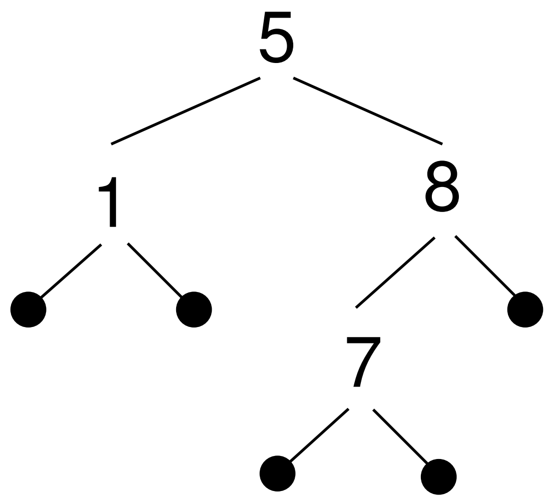
```
data Address = Address Building String
data Building =  Number Int | Name String
```

- Example values of the `Address` data type are:

```
Address (Number 42) "High Street"
Address (Name "Seaview") "Uplands Road"
```

Recursive Types

- Types can be described in terms of themselves. Consider for example binary trees of integers:



- A binary tree is defined recursively as:
  - a null node; or
  - a node with a value, a left sub-tree and a right sub-tree.

Recursive Types

- We can define a data type directly from this definition:

```
data Tree = Null |
            Node Int Tree Tree
```

- Three examples values of this type are:

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

- (The final value is represented by the diagram on the previous page.)

Recursive Types

- Many functions on trees will mirror the recursive structure of the type (i.e. use `Null` for the base case, & `Node` for the recursion).
- E.g, the following function returns the height of a binary tree:

```
height :: Tree -> Int
height Null = 0
height (Node _ st1 st2)
       = 1 + max (height st1) (height st2)
```

- The following function sums the values in a binary tree:

```
sumValues :: Tree -> Int
sumValues Null = 0
sumValues (Node n st1 st2)
       = n + sumValues st1 + sumValues st2
```