Computer Science 312

Defining New Data Types

Type Synonyms

```
Prelude> type Fruit = String

Prelude> fruits = ["Apple", "Banana"] :: [Fruit]

Prelude> :type fruits
[Fruit]

Prelude> "buttermilk":fruits
["buttermilk", "Apple", "Banana”]
```

A type synonym is convenient and readable, but it does not provide ***type safety***

If a fruit is just a string, we can make any old string a fruit, and perform string operations on fruits

Abstract Data Type (ADT)

- An ***abstract data type*** is a set of values and operations on those values; no other operations are allowed

- Examples: integer, list, string, text file, dictionary, stack, queue, BST

- The implementation of an ADT is hidden behind an interface (the allowable operations)

- One sees the values and operations, but not the underlying data representations and algorithms

Defining New Data Types

- In object-oriented languages like C++, Java, and Python, new data types are ***classes***, which provide type names, constructors for values of those types, and methods for operating on those values

- In functional languages, new data types are ***algebraic types***, which provide type names and constructors for values of those types

- Values of algebraic types, or the component parts of these values, can be accessed but not mutated

Algebraic Data Types

3 broad categories:

- Enumerated – a set of symbolic values
- Product – a structure with component parts
- Union – a set of enumerated and product values

Defining Enumerated Types

Syntax:

```
data <type name> = <value-1> | <value-2> | … | <value-n>
```

Examples (in the module **Algebraic.hs**):

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

```
data Color = Red | Green | Blue
```

```
data Fruit = Apple | Banana | Cherry | Orange
```

```
data WeekDay = Monday| Tuesday| Wednesday | Thursday | Friday
```

Using Enumerated Types

```
:load Algebraic
Ok, one module loaded.

*Algebraic> day = Monday

*Algebraic> :type day
day :: WeekDay

*Algebraic> day

<interactive>:15:1: error:
    • No instance for (Show WeekDay) arising from a use of
      ‘print’
    • In a stmt of an interactive GHCi command: print it
```

The GHCI repl tries to run **`show`** to get the day’s string for output, but can’t find this function

Deriving Some Type Classes

Syntax:

```
data <type name> = <value-1> | <value-2> | … | <value-n>
    deriving (<type class-1>, … , <type class-n>)
```

Example (in the module **Algebraic.hs**):

```
data WeekDay = Monday| Tuesday| Wednesday | Thursday | Friday
    deriving (Enum, Eq, Ord, Read, Show)
```

`Enum`	supports ranges using `..` to create lists
`Eq`	supports equality tests using **`==`** and **`/=`**
`Ord`	supports comparisons using **`<`**, **`>`**, **`<=`**, **`>=`**
`Read`	supports converting from string using **`read`**
`Show`	supports converting to string using **`show`**

Using Enumerated Types

```
:load Algebraic
Ok, one module loaded.

*Algebraic> day = Monday

*Algebraic> day
Monday

*Algebraic> show day
"Monday"

*Algebraic> day < Tuesday
True

*Algebraic> [Monday..Friday]
[Monday, Tuesday, Wednesday, Thursday, Friday]

*Algebraic> read "Monday" :: WeekDay
Monday
```

Defining Product Types

Syntax:

```
data <type name> = <constructor name> <component type-1> …

                                       <component type-n>
```

Examples (in the module **Algebraic.hs**):

```
data Student = Student String [Int]
```

```
data HoursWorked = HoursWorked AssociationList
```

```
data Employee = Employee String Float HoursWorked
```

The constructor name is usually the same as the type name

Using Product Types

```
*Algebraic> student = Student "Pepe" [100, 88, 77]

*Algebraic> student
Student "Pepe" [100,88,77]

*Algebraic> hr = HoursWorked (zip [Monday, Tuesday, Friday]
                                  [8, 7, 4.5])

*Algebraic> hr
HoursWorked [(Monday,8.0),(Tuesday,7.0),(Friday,4.5)]

*Algebraic> emp = Employee "Ken" 15.00 hr

*Algebraic> emp
Employee "Ken" 15.0 (HoursWorked [(Monday,8.0),(Tuesday,7.0),
                                  (Friday,4.5)])
```

The type “tags” each value of that type, leading to safety

Access with Pattern Matching

```
*Algebraic> student = Student "Pepe" [100, 88, 77]

*Algebraic> student
Student "Pepe" [100,88,77]

*Algebraic> Student name scores = student

*Algebraic> name
"Pepe"

*Algebraic> scores
[100,88,77]
```

Must supply the type tag in the pattern – enforces safety!

`Student` as an ADT

```
newStudent name numberOfScores -> Student

getNumberOfScores student -> Int

getName student -> String

setName newName student -> Student

getScore index student -> Int

setScore index newScore student -> Student

getHighScore student -> Int

getAverageScore student -> Float
```

This is the interface to the **`Student`** type

Creating and Using a **`Student`** Value

```
Prelude> :load Student
Ok, one module loaded.

*Student> student = newStudent "Nora" 10

*Student> student
Student "Nora" [0,0,0,0,0,0,0,0,0,0]

*Student> getName student
"Nora"

*Student> getNumberOfScores student
10

*Student> student2 = setScore 0 100 student  -- Mutator returns a new
                                              -- Student value

*Student> student2
Student "Nora" [100,0,0,0,0,0,0,0,0,0]

*Student> getAverageScore student2
10.0
```

Implementation of **`Student`**

```
module Student where

data Student = Student String [Int]
  deriving (Show)

newStudent :: String -> Int -> Student
newStudent name numberOfScores =
  let scores = map (\x -> 0) [1..numberOfScores]
  in Student name scores
```

`newStudent` builds the scores list and returns a new **`Student`** value

Implementation of **Student**

```
module Student where

data Student = Student String [Int]
  deriving (Show)

newStudent :: String -> Int -> Student
newStudent name numberOfScores =
  let scores = map (\x -> 0) [1..numberOfScores]
  in Student name scores

getName :: Student -> String
getName (Student name _) = name

setName :: String -> Student -> Student
setName newName (Student _ scores) = Student newName scores
```

`setName` transforms a **`Student`** value into another **`Student`** value

Generic AlgebraicTypes

```
module Stack where

data Stack a = Stack [a]
  deriving (Show)

newStack :: (Stack a)
newStack = Stack []

pushStack :: a -> (Stack a) -> (Stack a)
pushStack newItem (Stack items) = Stack (newItem:items)

popStack :: (Stack a) -> (Stack a)
popStack (Stack items) = Stack (tail items)

topStack :: (Stack a) -> a
topStack (Stack items) = head items
```

Defining Union Types

Syntax:

```
data <type name> = <constructor name> <component type-1> …
                                      <component type-n> |
                   <value>
```

Examples:

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

```
data BST k v = EmptyNode | InteriorNode k v (BST k v) (BST k v)
```

Combines a structured type with an atomic type as alternatives

Type name is now different from constructor names

Can be recursive!

A Binary Search Tree (BST)

```
module BST where

data BST k v = EmptyNode | InteriorNode k v (BST k v) (BST k v)
  deriving (Eq, Show)

newBST :: (BST k v)
newBST = EmptyNode
```

Two types of nodes:

- **`EmptyNode`** is like an empty link
- **`InteriorNode`** has a key, a value, a left BST and a right BST

Can represent a sorted map (tree map)

A Lookup Function for BST

```
module BST where

data BST k v = EmptyNode | InteriorNode k v (BST k v) (BST k v)
  deriving (Eq, Show)

newBST :: (BST k v)
newBST = EmptyNode

bstLookup :: Ord k => k -> (BST k v) -> v
bstLookup key EmptyNode = error "Key not found"
bstLookup key (InteriorNode k v left right)
  | key < k = bstLookup key left
  | key > k = bstLookup key right
  | key == k = v
```

Type of key is now constrained by **`Ord`**, for comparisons

A Insertion Function for BST

```
module BST where

data BST k v = EmptyNode | InteriorNode k v (BST k v) (BST k v)
  deriving (Eq, Show)

newBST :: (BST k v)
newBST = EmptyNode

bstAdd :: Ord k => k -> v -> (BST k v) -> (BST k v)
bstAdd key value EmptyNode = InteriorNode key value EmptyNode EmptyNode
bstAdd key value (InteriorNode k v left right)
    | key < k = InteriorNode k v (bstAdd key value left) right
    | key > k = InteriorNode k v left (bstAdd key value right)
    | key == k = error "Duplicate key"
```

- Base case: tree is empty, so return a new interior node
- Otherwise, if key is less, go left to insert
- Otherwise, if key is greater, go right to insert
- Otherwise, duplicate key is found, so raise an error

For next time

Record structures

Type classes