

Algebraic Data Types

COMP1100 — Introduction to Programming and Algorithms

Reading: Thompson Ch.14

Ben Lippmeier

Australian National University
Semester 2, 2006

Defining Our Own Data Types

So far we have only used the pre-defined and constructed types of Haskell.

For representing “real-world” data types, we can do a lot with just those types, possibly re-naming them with synonyms.

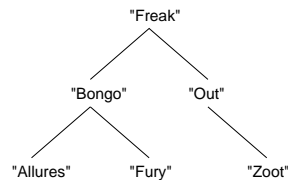
However, some “real-world” data types do not *naturally* correspond to standard types and requires some kind of *coding*.

For example, to represent the months of the year or the days of the week, we could use the integers 1–12 and 1–7 respectively.

In that case, it is not clear whether a literal 3 represents March, Tuesday, Wednesday, the value 3, . . .

Other examples where data type “coding” is inconvenient, unclear and unnatural:

- A type which is *either a number or a string* (for example, in some areas houses may have names rather than numbers).
- A *tree* data structure:



Any type *can* be represented using the pre-defined types, but the representation may not be very *natural*.

Haskell's **Algebraic Data Types** let us define *new types* to more naturally model types like those above.

Enumerated Types

The simplest algebraic type definitions are just an enumeration of the *elements* or *values* of that new type.

For example, the months of the year and the days of the week could be defined as follows:

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

```
data Month = Jan | Feb | Mar | Jun | Jul | Aug | Sep | Oct | Nov | Dec
```

Another example:

```
data Temp = Cold | Hot
```

```
data Season = Spring | Summer | Autumn | Winter
```

These are *new types*.

Type `Temp` consists of only two values: `Cold` and `Hot`.

`Cold` and `Hot` are called **constructors** of type `Temp`.

Note: Constructors must begin with a **capital letter** (to distinguish them from variables). Remember that type names also begin with a capital letter.

Constructors May Appear in Patterns

That is the standard way to define functions over algebraic types. For example:

```
weather :: Season -> Temp
weather Summer = Hot
weather other  = Cold
```

```
weekend :: Day -> Bool
weekend Sat    = True
weekend Sun    = True
weekend other  = False
```

Algebraic types do not automatically have operators such as equality, ordering, `show` and so on. For example, the following definition:

```
weather :: Season -> Temp
weather s
  | s == Summer = Hot
  | otherwise   = Cold
```

gives the error message:

```
AlgDT.hs:37:8:
  No instance for (Eq Season)
    arising from use of ‘==’ at AlgDT.hs:37:8-9
  ...
```

Defining Instances

If you need some overloaded operator such as `(==)` to apply to an algebraic type, you can declare it as an instance of class `Eq` and define the equality function yourself:

```
instance Eq Temp where
  Hot == Hot    = True
  Cold == Cold  = True
  _ == _        = False
```

This can get tedious. (Try defining `Month` as an instance of `Ord`.)

Deriving Instances

Haskell has a standard way of deriving instances for algebraic types. In most cases it is satisfactory but sometimes you need to define your own operators.

```
data Season = Spring | Summer | Autumn | Winter
           deriving Eq
```

```
data Temp   = Cold | Hot
           deriving Eq
```

The clause `deriving Eq` makes the type an instance of type class `Eq` in the “obvious” way.

If we want to show the new values as strings, we also need to derive an instance of class `Show`. Otherwise:

```
> weather Autumn
No instance for (Show Temp)
Probable fix: add an instance declaration for (Show Temp)
```

We need:

```
data Season = Spring | Summer | Autumn | Winter
           deriving (Eq, Show)
data Temp   = Cold | Hot
           deriving (Eq, Show)
```

Often we also want to derive an instance of classes `Ord` and `Enum` so that we can use the relational operators and progressions, such as `[Mon..Fri]`.

Union Types

Example: A program may be concerned with various geometrical shapes. Using an *enumerated type* we can distinguish between different kinds of shapes, but we cannot carry other information such as dimensions.

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

The alternatives in a `data` definition can include *other types*, rather than being simple constants like `Hot` and `Cold`.

```
data Shape
  = Circle    Float
  | Rectangle Float Float
  deriving (Eq, Show)
```

Now `Circle` is a **constructor function**.

```
> :type Circle
Circle :: Float -> Shape
```

In other words, `Circle` *constructs* a `Shape` from any `Float` value.

similarly, `Rectangle` *constructs* a `Shape` from two `Float` values:

```
> :type Rectangle
Rectangle :: Float -> Float -> Shape
```

Constructor Functions May Appear in Patterns

Constructor functions are the *only* functions that can appear in patterns.

For example:

```
isRound :: Shape      -> Bool
isRound (Circle r)    = True
isRound (Rectangle l b) = False

area :: Shape         -> Float
area (Circle r)       = pi * r^2
area (Rectangle l b) = l * b
```

Recursive Algebraic Types

It is possible to use the algebraic type being defined in a *data* definition within its own definition. That means the type itself is *recursive*.

Lists are a common example of a recursive type. A list is either:

- **empty**, or
- it consists of a **head** and a **tail** where the tail is also a *list*.

An algebraic type declaration that mirrors the built-in type of lists:

```
data List
  = Empty
  | Cons Int List
  deriving (Eq, Ord, Show)
```

Another standard example of a recursive data type is the **Tree** structure shown at the beginning of this section.

A **Binary Tree** is either:

- **empty**, or
- it consists of a **node** containing some value and **left and right subtrees** where the subtrees are also *binary trees*.

An example where the node values are strings:

```
data Tree
  = Null
  | Node String Tree Tree
```

Polymorphic Algebraic Types

The last two examples are constrained in the sense that type *List* represents only lists of *integers* and *Tree* only represents binary trees of *strings*.

It is possible to define lists and trees of any component type:

```
data Tree a
  = Null
  | Node a (Tree a) (Tree a)
```

The type variable 'a' can be instantiated to any type.

```
Node 3    Null (Node 4 Null Null)    :: Tree Int
Node 'a' (Node 'b' Null Null) Null  :: Tree Char
```

Tuples and lists as ADTs

The 'inbuilt' tuple and list types are really ADTs, but with some syntactic sugar to make the source code easier to read.

We could represent a two element tuple with the following data type:

```
data Tuple2 a b
  = T2 a b
  deriving (Eq, Ord, Show)
```

The tuple types and values could be translated as follows:

`(String, Int) ⇒ Tuple2 String Int`

`("Catch", 22) ⇒ T2 "Catch" 22`

We could also represent lists with an ADT:

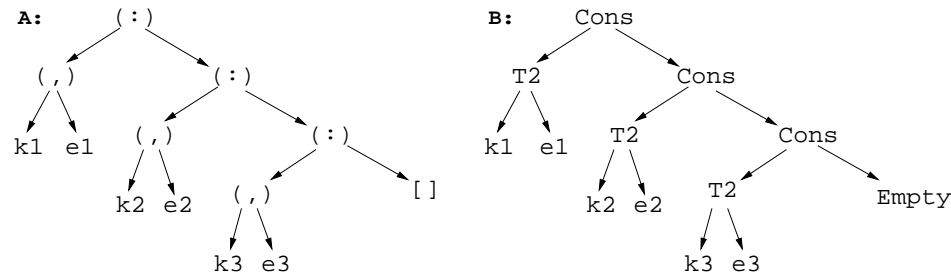
```
data List a
  = Empty
  | Cons a (List a)
  deriving (Eq, Ord, Show)
```

The translations would then be:

```
[Int]
⇒ List Int

1 : 2 : 3 : []
⇒ Cons 1 (Cons 2 (Cons 3 Empty))
```

Here is the table example from the last lecture - first represented with lists and tuples, and then with an ADT. Only the names of the constructors have changed, not the structure.



A: `(k1,e1) : (k2,e2) : (k3, e3) : []`
of type `[(a, b)]`

B: `Cons (T2 k1 e1) (Cons (T2 k2 e2) (Cons (T2 k3 e3) Empty))`
of type `List (Tuple2 a b)`

where `a, b` are the types of the `kn, en` elements respectively