CSci 450: Org. of Programming Languages
Algebraic Data Types

H. Conrad Cunningham

10 October 2017

Contents

Acknowledgements: In Summer 2016, I adapted and revised this chapter from chapter 8 of my *Notes on Functional Programming with Haskell* and from my notes on *Functional Data Structures* from the Spring 2016 Scala variant of CSci 555. The latter was based, in part, on chapter 3 of the book *Functional Programming in Scala* by Paul Chiusano and Runar Bjarnason (Manning, 2015).

In Fall 2017, I added the discussion of the `Maybe` and `Either` types. For this work, I examined the `Data.Maybe` and `Data.Either` documentation and the Wikipedia article on "Option Type".

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of October 2017 is a

recent version of Firefox from Mozilla.

Code: The Haskell module for this chapter is in file `Chap08.hs`.

TODO:

- add chapter goals and outcomes
- convert and test functions from Scala
- clean and add exercises
- clean up and document associated module

8 Algebraic Data Types

8.1 Chapter Introduction

TODO

8.2 Definition

An *algebraic data type* is a type formed by combining other types, that is, it is a *composite* data type. The data type is created by an algebra of operations of two primary kinds:

- a *sum* operation that constructs values to have one variant among several possible variants. These sum types are also called *tagged*, *disjoint union*, or *variant* types. The combining operation is the alternation operator, which denotes the choice of one but not both between two alternatives.
- a *product* operation that combines several values (i.e., *fields*) together to construct a single value. These are *tuple* and *record* types. The combining operation is the Cartesian product from set theory.

We can combine sums and products recursively into arbitrarily large structures.

An *enumerated type* is a sum type in which the constructors take no arguments. Each constructor corresponds to a single value.

Although sometimes the acronym ADT is used for both, an *algebraic data type* is a different concept from an *abstract data type*.

- We specify an algebraic data type with its *syntax* (i.e., structure)–with rules on how to compose and decompose them.
- We specify an abstract data type with its *semantics* (i.e., meaning)–with rules about how the operations behave in relation to one another.

Perhaps to add to the confusion, in functional programming we sometimes use an algebraic data type to help define an abstract data type.

8.3 Haskell Algebraic Data Types

8.3.1 Declaring `data` types

In addition to the built-in data types we have discussed, Haskell also allows the definition of new data types using declarations of the form:

> `data` **Datatype a*1* a*2* ⋯ a*n* = Constr*1* | Constr*2* | ⋯ | Constr*m***

where:

- **Datatype** is the name of a new type constructor of *arity* n *($n \geq 0$)*. As with the built-in types, the name of the data type must begin with an uppercase letter.
- **a*1***, **a*2***, ⋯ **a*n*** are distinct type variables representing the n parameters of the data type. These begin with lowercase letters (by convention at the beginning of the alphabet).
- **Constr*1***, **Constr*2***, ⋯, **Constr*m*** are the m *($m \geq 1$)* data constructors that describe the ways in which the elements of the new data type are constructed. These begin with uppercase letters.

8.3.2 Example type `Color`

For example, consider a new data type `Color` whose possible values are the colors on the flag of the USA. The names of the data constructors (the color constants in this case) must also begin with capital letters.

```
data Color = Red | White | Blue
             deriving (Show, Eq)
```

`Color` is an example of an *enumerated type*, a *sum* type that consists of a finite sequence of *nullary* (i.e., the arity—number of parameters—is zero) data constructors.

We can use the type and data constructor names defined with `data` in declarations, patterns, and expressions in the same way that the built-in types can be used.

```
isRed :: Color -> Bool
isRed Red = True
isRed _   = False
```

Data constructors can also have associated values. For example, the constructor `Grayscale` below takes an integer value.

```
data Color' = Red' | Blue' | Grayscale Int
              deriving (Show, Eq)
```

Constructor `Grayscale` implicitly defines a constructor function with the type.

8.3.3 Deriving class instances

The optional `deriving` clauses above are very useful. They declare that these new types are automatically added as instances of the classes listed. In the above cases, `Show` and `Eq` enable objects of type `Color` to be converted to a `String` and compared for equality, respectively.

The Haskell compiler derives the body of an instance syntactically from the data type declaration. It can derive instances for classes `Eq`, `Ord`, `Enum`, `Bounded`, `Read`, and `Show`.

The derived instances of `Eq` include the `(==)` and `(/=)` methods. The derived instances of `Ord` also include the `compare`, `(<)`, `(<=)`, `(>)`, `(>=)`, `max`, and `min` methods. The ordered comparison operators use the order of the constructors given in the `data` statement, from smallest to largest, left to right. These comparison operators are strict in both arguments.

Similarly, the `Enum` instance assigns integers to the constructors increasing from 0 at the left;`Bounded` assigns `minBound` to the leftmost and `maxBound` to the rightmost.

`Show` enables the function `show` to convert the data type to a syntactically correct Haskell expression consisting of only the constructor names, parentheses, and spaces. Similarly, `Read` enables the function `read` to parse such a string into a value of the data type.

For example, the data type `Bool` might be defined as:

```
data Bool = False | True
            deriving (Ord, Show)
```

Thus `False < True` evaluates to `True` and `False > True` evaluates to `False`. If `x == False`, then `show x` yields the string `False`.

8.3.4 More example types

Consider a data type `Point` that has a type parameter. The following defines a polymorphic type; both of the values associated with the constructor `Pt` must be of type `a`. Constructor `Pt` implicitly defines a constructor function of type `a -> a -> Point a`.

```
data Point a = Pt a a
               deriving (Show, Eq)
```

As another example, consider a polymorphic set data type that represents a set as a list of values as follows. Note that the name `Set` is used both as the type constructor and a data constructor. In general, do not use a symbol in multiple ways. It is acceptable to double use only when the type has only one constructor.

```
data Set a = Set [a]
             deriving (Show, Eq)
```

Now we can write a function `makeSet` to transform a list into a `Set`. This function uses the function `nub` from the `Data.List` module to remove duplicates from a list.

```
makeSet :: Eq a => [a] -> Set a
makeSet xs = Set (nub xs)
```

As we have seen previously, programmers can also define type synonyms. As in user-defined types, synonyms may have parameters. For example, the following might define a matrix of some polymorphic type as a list of lists of that type.

```
type Matrix a = [[a]]
```

We can also use special types to encode error conditions. For example, suppose we want an integer division operation that returns an error message if there is an attempt to divide by 0 and returns the quotient otherwise. We can define and use a union type `Result` as follows:

```
data Result a = Ok a | Err String
                deriving (Show, Eq)

divide :: Int -> Int -> Result Int
divide _  0 = Err "Divide by zero"
divide x  y = Ok (x `div` y)
```

Then we can use this operation in the definition of another function `f` that returns the maximum `Int` value `maxBound` when a division by 0 occurs.

```
f :: Int -> Int -> Int
f x y  = return (divide x y)
         where return (Ok z)  = z
               return (Err s) = maxBound
```

The auxiliary function `return` can be avoided by using the Haskell `case` expression as follows:

```
f' x y =
    case divide x y of
        Ok z  -> z
        Err s -> maxBound
```

This case expression evaluates the expression `divide x y`, matches its result against the patterns of the alternatives, and returns the right-hand-side of the first matching patter.

Later in this chapter we discuss the `Maybe` and `Either` types, two polymorphic types for handling errors defined in the Prelude.

8.3.5 Recursive types

Types can also be recursive.

For example, consider the user-defined type `BinTree`, which defines a binary tree with values of a polymorphic type.

```
data BinTree a = Empty | Node (BinTree a) a (BinTree a)
               deriving (Show, Eq)
```

This data type represents a binary tree with a value in each node. The tree is either "emptyD" (denoted by `Empty`) or it is a "node" (denoted by `Node`) that consists of a value of type `a` and "left" and "right" subtrees. Each of the subtrees must themselves be objects of type `BinTree`.

Thus a binary tree is represented as a three-part "record" in which the left and right subtrees are represented as nested binary trees. There are no explicit "pointers".

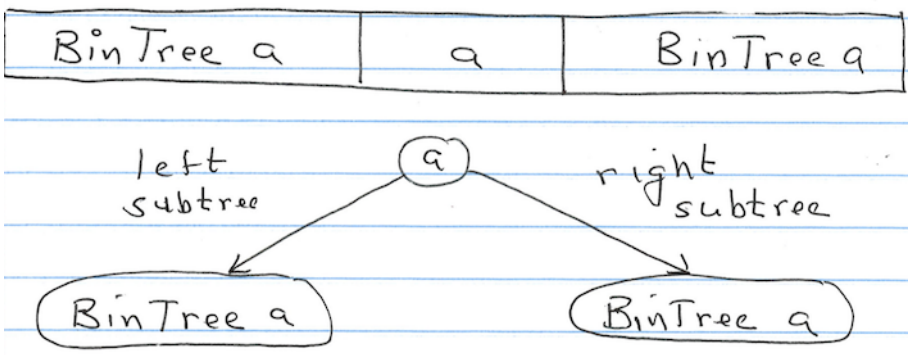

Figure 8-1: Binary Tree `BinTree`

Consider a function `flatten` to return the list of the values in binary tree in the order corresponding to a left-to-right in-order traversal. Thus expression

```
flatten (Node (Node Empty 3 Empty) 5 (Node (Node Empty 7 Empty) 1 Empty))
```

yields `[3,5,7,1]`.

```
flatten :: BinTree a -> [a]
flatten Empty        = []
flatten (Node l v r) = flatten l ++ [v] ++ flatten r
```

The second leg of `flatten` requires two recursive calls. However, as long as the input tree is finite, each recursive call receives a tree that is simpler (e.g., shorter) than the input. Thus all recursions eventually terminate when `flatten` is called with an `Empty` tree.

Function `flatten` can be rendered more efficiently using an accumulating parameter and cons as in the following:

```
flatten' :: BinTree a -> [a]
flatten' t = inorder t []
             where inorder Empty xs        = xs
                   inorder (Node l v r) xs =
                       inorder l (v : inorder r xs)
```

Auxiliary function `inorder` builds up the list of values from the right using cons.

To extend the example further, consider a function `treeFold` that folds an associative operation `op` with identity element `i` through a left-to-right in-order traversal of the tree.

```
treeFold :: (a -> a -> a) -> a -> BinTree a -> a
treeFold op i Empty        = i
treeFold op i (Node l v r) = op (op (treeFold op i l) v)
                                (treeFold op i r)
```

Now let's consider a slightly different formulation of a binary tree: a tree in which values are only stored at the leaves.

```
data Tree a = Leaf a | Tree a :^: Tree a
              deriving (Show, Eq)
```

This definition introduces the constructor function name `Leaf` as the constructor for leaves and the infix construction operator "`:^:`" as the constructor for internal nodes of the tree. (A constructor operator symbol must begin with a colon.)

These constructors allow such trees to be defined conveniently. For example, the tree

```
((Leaf 1 :^: Leaf 2) :^: (Leaf 3 :^: Leaf 4))
```

generates a complete binary tree with height 3 and the integers 1, 2, 3, and 4 at the leaves.

Suppose we want a function `fringe`, similar to function `flatten` above, that displays the leaves in a left-to-right order. We can write this as:

```
fringe :: Tree a -> [a]
fringe (Leaf v)  = [v]
fringe (l :^: r) = fringe l ++ fringe r
```

As with `flatten` and `flatten'` above, function `fringe` can also be rendered more efficiently using an accumulating parameter as in the following:

```
fringe' :: Tree a -> [a]
fringe' t = leaves t []
            where leaves (Leaf v)  = ((:) v)
                  leaves (l :^: r) = leaves l . leaves r
```

Auxiliary function `leaves` builds up the list of leaves from the right using cons.

8.4 Error-handling with `Maybe` and `Either`

Before we examine `Maybe` and `Either`, let's consider a use case.

An *association list* is a list of pairs in which the first component is some *key* (e.g., a string) and the second component is the *value* associated with that key. It is a simple form of a *map* or *dictionary* data structure.

Suppose we have an association list that maps the name of a student (a key) to the name of the student's academic advisor (a value). The following function `lookup'` carries out the search recursively.

```
lookup' :: String -> [(String,String)] -> String
lookup' key ((x,y):xys)
    | key == x  =  y
    | otherwise =  lookup' key xys
```

But what do we do when the key is not in the list (e.g., the list is empty)? How do we define a leg for `lookup' key []` ?

1. Leave the function undefined for that pattern?

   In this case, evaluation will halt with a "non-exhaustive pattern" error message.

2. Put in an explicit `error` call with a custom error message?

3. Return some default value of the advisor such as `"NONE"`?

4. Return a *null reference*?

The first two approaches either halt the entire program or require use of the exception-handling mechanism. However, in any language, both abnormal termination and exceptions should be avoided except in cases in which the program is unable to continue. The lack of an assignment of a student to an advisor is likely not such an extraordinary situation.

Exceptions break referential transparency and, hence, negate many of the advantages of purely functional languages such as Haskell. In addition, Haskell programs can only catch exceptions in the outer layers that handle input/output.

The third approach only works when there is some value that is not valid. This is not a very general approach.

The fourth approach, which is not available in Haskell, can be an especially unsafe programming practice. British computing scientist Tony Hoare, who introduced the null reference into the Algol type system in the mid-1960s, calls that his "billion dollar mistake" because it "has led to innumerable errors, vulnerabilities, and system crashes".

What is a safer, more general approach than these?

Haskell includes the union type `Maybe` (from the Prelude and `Data.Maybe`) which can be used to handle such cases.

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord)
```

The `Maybe` algebraic data type encapsulates an optional value. A value of type `Maybe a` either contains a value of type `a` (represented by `Just a`) or it is empty (represented by `Nothing`).

The`Maybe` type is a good way to handle errors or exceptional cases without resorting to an `error` call.

Now we can define a general version of `lookup'` using a `Maybe` return type. (This is essentially function `lookup` from the Prelude.)

```
lookup'' :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup'' key [] =  Nothing
lookup'' key ((x,y):xys)
   | key == x   =  Just y
   | otherwise  =  lookup'' key xys
```

Suppose `advisorList` is an association list pairing students with their advisors and `defaultAdvisor` is the advisor the student should consult if no advisor is officially assigned. We can look up the advisor with a call to `lookup` and then pattern match on the `Maybe` value returned. (Here we use a `case` expression.)

```
whoIsAdvisor :: String -> String
whoIsAdvisor std =
    case lookup std advisorList of
        Nothing   -> defaultAdvisor
        Just prof -> prof
```

The `whoIsAdvisor` function just returns a default value in place of `Nothing`. The function

```
fromMaybe :: a -> Maybe a -> a
```

supported by the `Data.Maybe` library has the same effect. Thus we can rewrite `whoIsAdvisor` as follows:

```
whoIsAdvisor' std =
    fromMaybe defaultAdvisor $ lookup std advisorList
```

Alternatively, we could use `Data.Maybe` functions such as:

```
isJust    :: Maybe a -> Bool
isNothing :: Maybe a -> Bool
fromJust  :: Maybe a -> a    -- error if Nothing
```

This allows us to rewrite `whoIsAdvisor` as follows:

```
whoIsAdvisor'' std =
    let ad = lookup std advisorList
    in if isJust ad then fromJust ad else defaultAdvisor
```

If we need more fine-grained error messages, then we can use the union type `Either` defined as follows:

```
data Either a b = Left a | Right b
                  deriving (Eq, Ord, Read, Show)
```

The `Either a b` type represents values with two possibilities: a `Left a` or `Right b`. By convention, a `Left` constructor usually contains an error message and a `Right` constructor a *correct* value.

As with `fromMaybe`, we can use similar `fromRight` and `fromLeft` functions from the `Data.Either`library to extract the `Right` or `Left` values or to return a default value when the value is represented by the other constructor.

```
fromLeft  :: a -> Either a b -> a
fromRight :: b -> Either a b -> b
```

Library module `Data.Either` also includes functions to query for the presence of the two constructors.

```
isLeft    :: Either a b -> Bool
isRight   :: Either a b -> Bool
```

Most recently designed languages include a maybe or option type. For example, Java 8 added the final class `Optional<T>`. Scala supports the `Option[T]` case class hierarchy, and Rust includes the `Option<T>` enum (algebraic data type). The functional languages Idris, Elm, and PureScript all have Haskell-like `Maybe` algebraic data types.

When programming in an object-oriented language that does not provide an option/maybe type, a programmer can often use the Null Object design pattern to achieve a similar result. Instead of returning a null reference to denote the absence of a valid result, the function can return an object that implements the expected interface but which "does nothing" – at least nothing harmful.

8.5 Exercises

1. For trees of type `Tree`, implement a tree-folding function similar to `treeFold`.
2. For trees of type `BinTree`, implement a version of `treeFold` that uses an accumulating parameter. (Hint: `foldl`.)
3. In a binary search tree all values in the left subtree of a node are less than the value at the node and all values in the right subtree are greater than the value at the node.

Given binary search trees of type `BinTree`, implement the following Haskell functions:

a. `makeTree` that takes a list and returns a perfectly balanced (i.e., minimal height) `BinTree` such that `flatten (makeTree xs) = sort xs`. Prelude function `sort` returns its argument rearranged into ascending order.

b. `insertTree` that takes an element and a `BinTree` and returns the `BinTree` with the element inserted at an appropriate position.

c. `elemTree` that takes an element and a `BinTree` and returns `True` if the element is in the tree and `False` otherwise.

d. `heightTree` that takes a `BinTree` and returns its height. Assume that height means the number of levels in the tree. (A tree consisting of exactly one node has a height of 1.)

e. `mirrorTree` that takes a `BinTree` and returns its mirror image. That is, it takes a tree and returns the tree with the left and right subtrees of every node swapped.

f. `mapTree` that takes a function and a `BinTree` and returns the `BinTree` of the same shape except each node's value is computed by applying the function to the corresponding value in the input tree.

g. `showTree` that takes a `BinTree` and displays the tree in a parenthesized, left-to-right, in-order traversal form. (That is, the traversal of a tree is enclosed in a pair of parentheses, with the traversal of the left subtree followed by the traversal of the right subtree.)

Extend the package to support both insertion and deletion of elements. Keep the tree balanced using a technique such the AVL balancing algorithm.

4. Implement the package of functions described in the previous exercise for the data type `Tree`.

5. Each node of a general (i.e., multiway) tree consists of a label and a list of (zero or more) subtrees (each a general tree). We can define a general tree data type in Haskell as follows:

```
data Gtree a = Node a [Gtree a]
```

For example, tree `(Node 0 [ ])` consists of a single node with label 0; a more complex tree, , consists of root node with three single-node subtrees.

Implement a "map" function for general trees, i.e., write Haskell function

```
mapGtree :: (a -> b) -> Gtree a -> Gtree b
```

that takes a function and a `Gtree` and returns the `Gtree` of the same shape such that each label is generated by applying the function to the corresponding label in the input tree.

6. We can introduce a new Haskell type for the natural numbers (i.e., nonnegative integers) with the statement

```
data Nat = Zero | Succ Nat
```

where the constructor `Zero` represents the value 0 and constructor `Succ` represents the "successor function" from mathematics. Thus `(Succ Zero)` denotes 1, `(Succ (Succ Zero))` denotes 2, and so forth. Implement the following Haskell functions.

   a. `intToNat` that takes a nonnegative `Int` and returns the equivalent `Nat`, for example, returns .

   b. `natToInt` that takes a `Nat` and returns the equivalent value of type `Int`, for example, returns 1.

   c. `addNat` that takes two `Nat`s and returns their sum as a `Nat`. This function cannot use integer addition.

   d. `mulNat` that takes two `Nat`s and returns their product as a `Nat`. This function cannot use integer multiplication or addition.

   e. `compNat` that takes two `Nat`s and returns the value `-1` if the first is less than the second, `0` if they are equal, and `1` if the first is greater than the second. This function cannot use the integer comparison operators.

7. Consider the following Haskell data type for representing sequences (i.e., lists):

```
data Seq a = Nil | Att (Seq a) a
```

`Nil` represents the empty sequence. `Att xz y` represents the sequence in which *last element* `y` is "attached" at the right end of the *initial sequence* `xz`.

Note that `Att` is similar to the ordinary "cons" (`:`) for Haskell lists except that elements are attached at the opposite end of the sequences. `(Att (Att (Att Nil 1) 2) 3)` represents the same sequence as the ordinary list `(1:(2:(3:[])))`.

Implement Haskell functions for the following operations on type `Seq`. The operations are analogous to the similarly named operations on the built-in Haskell lists.

   a. `lastSeq` takes a nonempty `Seq` and returns its last (i.e., rightmost) element.

   b. `initialSeq` takes a nonempty `Seq` and returns its initial sequence (i.e., sequence remaining after the last element removed).

   c. `lenSeq` takes a `Seq` and returns the number of elements that it contains.

d. `headSeq` takes a nonempty `Seq` and returns its head (i.e., leftmost) element.

e. `tailSeq` takes a nonempty `Seq` and returns the `Seq` remaining after the head element is removed.

f. `conSeq` that takes an element and a `Seq` and returns a `Seq` with the argument element as its head and the `Seq` argument as its tail.

g. `appSeq` takes two arguments of type `Seq` and returns a `Seq` with the second argument appended after the first.

h. `revSeq` takes a `Seq` and returns the `Seq` with the same elements in reverse order.

i. `mapSeq` takes a function and a `Seq` and returns the `Seq` resulting from applying the function to each element of the sequence in turn.

j. `filterSeq` that takes a predicate and a `Seq` and returns the `Seq` containing only those elements that satisfy the predicate.

k. `listToSeq` takes an ordinary Haskell list and returns the `Seq` with the same values in the same order (e.g., `headSeq (listToSeq xs) = head xs` for nonempty `xs`.)

l. `seqToList` takes a `Seq` and returns the ordinary Haskell list with the same values in the same order (e.g., `head (seqToList xz) = headSeq xz` for nonempty `xz`.)

8. Consider the following Haskell data type for representing sequences (i.e., lists):

```
data Seq a = Nil | Unit a | Cat (Seq a) (Seq a)
```

The constructor `Nil` represents the empty sequence; `Unit` represents a single-element sequence; and `Cat` represents the "concatenation" (i.e., append) of its two arguments, the second argument appended after the first.

Implement Haskell functions for the following operations on type `Seq`. The operations are analogous to the similarly named operations on the built-in Haskell lists. (Do not convert back and forth to lists.)

a. `toSeq` that takes a list and returns a corresponding `Seq` that is balanced.

b. `fromSeq` that takes a `Seq` and returns the corresponding list.

c. `appSeq` that takes two arguments of type `Seq` and returns a `Seq` with the second argument appended after the first.

d. `conSeq` that takes an element and a `Seq` and returns a `Seq` with the argument element as its head and the `Seq` argument as its tail.

e. `lenSeq` that takes a `Seq` and returns the number of elements that it contains.

f. `revSeq` that takes a `Seq` and returns a `Seq` with the same elements in reverse order.

g. `headSeq` that takes a nonempty `Seq` and returns its head (i.e., leftmost or front) element. (Be careful!)

h. `tailSeq` that takes a nonempty `Seq` and returns the `Seq` remaining after the head is removed.

i. `normSeq` that takes a `Seq` and returns a `Seq` with unnecessary embedded `Nil`s removed. (For example, `normSeq (Cat (Cat Nil (Unit 1)) Nil)` returns `(Unit 1)`.)

j. `eqSeq` that takes two `Seq`s and returns `True` if the sequences of values are equal and returns `False` otherwise. Note that two `Seq` "trees" may be structurally different yet represent the same sequence of values.

   For example, `(Cat Nil (Unit 1))` and `(Cat (Unit 1) Nil)` have the same sequence of values (i.e., `[1]`). But `(Cat (Unit 1) (Unit 2))` and `(Cat (Unit 2) (Unit 1))` do not represent the same sequence of values (i.e., `[1,2]` and `[2,1]`, respectively).

   Also `(Cat (Cat (Unit 1) (Unit 2)) (Unit 3))` has the same sequence of values as `(Cat (Cat (Unit 1) (Unit 2)) (Unit 3))` (i.e., `[1,2,3]`).

In general what are the advantages and disadvantages of representing lists this way?

8.6 References

[**Bird-Wadler 1998**] Richard Bird and Philip Wadler. *Introduction to Functional Programming*, Second Edition, Addison Wesley, 1998. [First Edition, 1988]

[**Chiusano-Bjarnason 2015**] Paul Chiusano and Runar Bjarnason, *Functional Programming in Scala*, Manning, 2015.

[**Cunningham 2014**] H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1994-2014.

[**Wikipedia 2017**] Wikipedia articles on "Algebraic Data Type" and "Abstract Data Type".

8.7 Terms and Concepts

Types, algebraic data types (composite), sum (tagged, disjoint union, variant, enumerated), product (tuple, record), nullary,recursive types, algebraic data

types versus abstract data types, pattern matching, safe error handling, `Maybe` and `Either` “option” types