CS51 **Spring 2024**

Code Review 3: Algebraic Data Types

1 Algebraic Data Types

Definition 1.1 (Algebraic Data Types)**.** We have seen that data types can either be *atomic* types or *composite* types.

We can use **algebraic data types** to construct **composite** data structures.

To construct algebraic data types, we use a combination of these two **methods**:

1. **Conjunction**: Constructing a composite value by **combining all components**. For example, a tuple of type `int * float` is formed as a **conjunction** of an `int` component and a `float` component.
2. **Alternation**: Constructing a composite value by **choosing one alternative from a set of alternatives**. For example, an `int list` is formed as an **alternation** of two components: the `[]` (empty list) and a **conjunction** of an `int` (the head of the list) and an `int list` (the tail of the list).

Algebraic data types are data types **built by conjunction and alternation**.

Example 1.2 (Colors)**.** In lab 5, we defined the following type to represent a set of colors:

Listing 1: color_label

```
type color_label =
 | Red
 | Orange
 | Yellow
 | Green
 | Blue
 | Indigo
 | Violet ;;
```

`color_label` is a **variant type**, where we define a set of alternatives, and we *choose* one of those alternatives. Here, we make use of **alternation**. For instance, we could define `let c1 : color_label = Red` or `let c2 : color_label = Indigo`. The alternatives (i.e. `Red`, `Orange`, etc.) are called **variants**.

We defined the type `color` in the following manner:

Listing 2: color

```
type color =
 | Simple of color_label
 | RGB of int * int * int
```

Defining **value constructors** `Simple` and `RGB` makes use of conjunction (and also alternation). We can construct a color either using a value of `color_label` or a conjuction of three integers representing the red, green, and blue values of the color.

Definition 1.3 (Invariants)**.** An **invariant** is a logical condition that must hold true in a certain situation. **Algebraic data types** allow us to **enforce invariants**.

Suppose, we defined `color_label` to be of type string (i.e. `type color_label = string`).

This would allow us to define values of type `color_label` that are not valid color labels. By using alternation, we ensure that a user can only define a valid color label.

This approach satisfies the **edict of prevention**: *making the illegal inexpressible*.

1.1 Pattern Matching

To decompose algebraic data types, we can use **pattern matching**.

Example 1.4 (ADT Pattern Matching)**.** In lab, we defined a function `rgb_of_color` that takes in a value of type `color` and returns its rgb values.

```
let rgb_of_color (c : color) : int * int * int =
  match c with
  | RGB (r, g, b) -> (r, g, b)
  | Simple x ->
    match x with
     | Red -> (255, 0, 0)
     | Orange -> (255, 165, 0)
     | Yellow -> (255, 255, 0)
     | Green -> (0, 255, 0)
     | Blue  -> (0, 0, 255)
     | Indigo -> (75, 0, 130)
     | Violet -> (240, 130, 240) ;;
```

1.2 Built-In Algebraic Data Types

We have already worked with several composite data structures that are implemented as algebraic data types, particularly *lists* and *options*.

Here's how we would define lists and options as an algebraic data type:

Listing 3: Lists and Options

```
type 'a list =
 | Nil
 | Cons of 'a * 'a list ;;

type 'a option =
| None
| Some of 'a ;;
```

Note that we can define **polymorphic ADTs** by writing our type in terms of **type variables,** as we did with `'a list` and `'a option`.

Also note that the definition of algebraic data types can be **recursive**. A list is either an empty list (denoted as `Nil`) or the **conjunction** of a value of type `'a` (the head of a list) and another list of type `'a list` (the tail of the list).

2 Edict of Prevention: Dictionaries

Example 2.1 (Dictionaries)**.** A **dictionary** is a data structure for storing a group of objects that has a set of *keys* each associated with a single *value*. The data type of the dictionary depends on the types of the keys and values.

Suppose we decide to define the type as polymorphic: `('key, 'value) dictionary`. One possible implementation for our dictionary data structure is to store the keys and values as equally-sized list such that the key at index i in the `'key` list corresponds to the value at index i in the `'value` list.

Thus, we can define a dictionary as follows[1]:

Listing 4: Dictionary Type

```
type ('key, 'value) dictionary =
 { keys : 'key list; values : 'value list } ;;
```

However, this representation of a dictionary is problematic, as it allows us to create structures such that the key list and value list are not equally-sized (e.g. `{keys = [1; 2; 3]; values = ["first"; "second"]}`. This contradicts the entire point of a dictionary where each key is associated with a single value.

Definition 2.2 (Edict of Prevention)**.** The **edict of prevention**, as coined from the textbook, is the idea that we can "make the illegal inexpressable."

In the context of the example, we can structure the dictionary type to ensure it is impossible to create a value that does not conform to the standard definition of a dictionary. One possible solution is to represent a dictionary as a *list of pairs* of keys and values rather than a *pair of lists*.

Listing 5: Dictionary Type (Version 2)

```
type ('key, 'value) dict_entry =
 { key : 'key; value : 'value }
and ('key, 'value) dictionary =
 ('key, 'value) dict_entry list ;;
```

In the above, the `dict_entry` is a record type with with a key and value field. We then define a dictionary as a list of `dict_entry` elements, which ensures that each key is associated with one value.

Read Section 11.3 in the textbook for more detail.

3 Recursive Algebraic Data Types

Example 3.1 (Binary Trees)**.** A **binary tree** is a tree data structure in which each node (i.e. element) has at most two children, which are referred to as the left and right child.

[1]Note that here we decide to use a record type for consistency with the textbook, but we could also implement a dictionary using another data structure, such as a tuple.

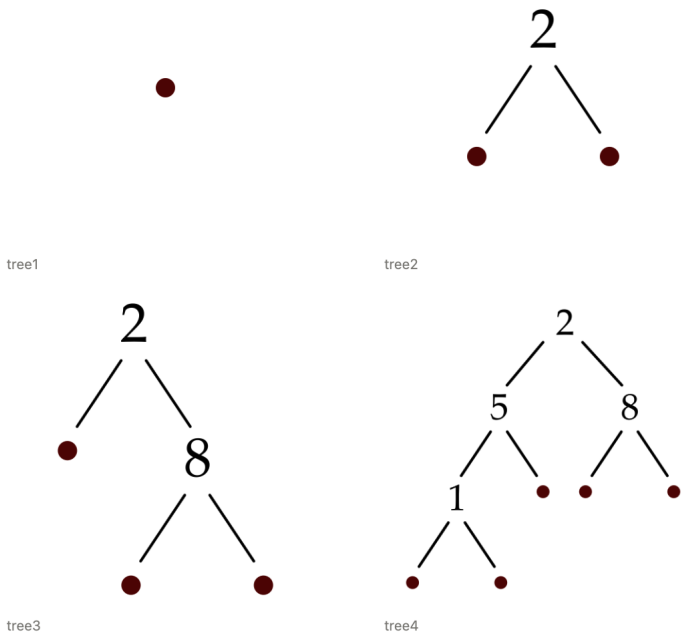


Figure 1: Example Binary Trees

Nodes which have no children are called **leaf** nodes. In the figure above, **leaf** nodes are denoted as dots. Formally, we can think of a binary tree as a tuple (r, L, R) where L and R are binary trees (called subtrees) and r is the root node of the tree. A binary tree can also be "empty," such that it has no nodes.

Algebraic data types can recursively refer to themselves, which allow us to represent data structures like binary trees:

Listing 6: Binary Tree Representation in OCaml

```
type 'a bintree =
  | Empty
  | Node of 'a * 'a bintree * 'a bintree ;;
```

We see that the type definition above naturally follows the formal definition stated above. Using this definition, we can define values for the binary trees listed in Figure 1 as follows:

Listing 7: Binary Tree Examples in OCaml

```
let tree1 = Empty ;;

let tree2 = Node (2, Empty, Empty) ;;

let tree3 =
  Node (2, Empty,
        Node (8, Empty, Empty)) ;;

let tree4 =
  Node (2, Node (5, Node (1, Empty, Empty), Empty),
        Node (8, Empty, Empty)) ;;
```

In lab 6, we defined a function `node_count` that takes in a tree and returns the number of nodes.

Listing 8: Node Count

```
let rec node_count (tree : 'a bintree) : int =
  match tree with
  | Empty -> 0
  | Node (_, left, right) -> 1 + node_count left
                               + node_count right ;;
```

To think about the logic behind this function, we can follow the process outlined in **Section 2.2 of Code Review 1 (How to Think About Recursion)**. First, consider the base cases. An empty binary tree has zero nodes, so if the tree passed in is `Empty`, we return 0. For the recursive step, we assume our `node_count` function is correct and we can use it as a helper.

Recall that we can think of a binary tree as a tuple consisting of the root node r, a left subtree L, and a right subtree R. The number of nodes that make up the tree are all the nodes in L, all the nodes in R, and the root node. L and R are binary trees, so we can use `node_count` to count the number of nodes in each subtree. We then add 1 to account for the root node. Thus, in our recursive step, we return the size of the tree as `1 + node_count left + node_count right`.

Example 3.2 (Binary Search Trees)**.** A **binary search tree** is a special kind of binary tree that satisfies the *BST Property*: for all nodes in the tree, all values in its left subtree are less than the node and all values in its right subtree are greater than the node.

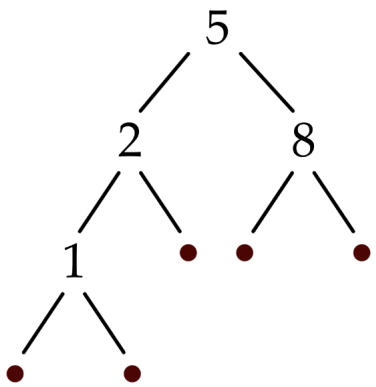


Figure 2: Binary Search Tree Example

In lab 6, we wrote a function `insert_bst` that inserts a value into a binary search tree (if the value is already in the tree, the tree remains unchanged).

Listing 9: Insert in BST

```
let rec insert_bst (value : 'a)
                   (tree : 'a bintree) : 'a bintree =
  match tree with
  | Empty -> Node (value, Empty, Empty)
  | Node (stored, left, right) ->
     if value = stored then tree
     else if value < stored then
       Node (stored, insert_bst value left, right)
     else
       Node (stored, left, insert_bst value right) ;;
```

Again, we first consider the base cases. If the tree is `Empty`, the result should be a tree with a single node (the one that is being inserted).

For the recursive step, assume `insert_bst` works (inserts the node into the correct spot in the tree). Suppose we start at the root of the tree, and check the value of the node being inserted with the value of the root. By the *BST property*, if the inserted node is less than the root, then the node should be inserted into the left subtree. If the node is greater than the root, then we insert the node into the right subtree. If it is equal to the root, then the node is already in the tree, so the tree should remain unchanged.

4 Practice Problems

Problem 4.1 (Multiway Trees)**.**

1. Define a type for a multiway tree, where each node can have any number of children.
2. Define a function `mapbt` that given a function `f` and a mulitway tree `tr`, applies `f` to each node in the tree and returns a transformed binary tree.

3. Define a function `foldbt` that walks the entire tree and performs an operation on each node and the accumulator, returning the final value of the accumulator.
4. A leaf is a node with no children. Define a function `leaves` that collects the leaves of a tree, returning them as a list.
5. Define a function `zip_trees` that takes two multiway trees and merges them into a new tree. The trees should be merged in a way that corresponding nodes are combined into pairs. If two trees are not the same shape, raise an Exception.

Problem 4.2 (File System)**.** Suppose we want to represent a file system using algebraic data types as follows:

```
type fileObj = File of string | Folder of string * fileObj list
```

A file system contains *objects*, which can be folders or files. Folders can contain files or other folders, which we'll refer to as *subfolders*. The folder in which a *subfolder* is contained is called the *parent* folder.

Define a function `list_all` that prints the name of all the **files** (each on a new line) contained in a given `fileObj` (including those in all subfolders).

Problem 4.3 (Syntax Trees)**.** Often, linguists use a tree-like structure to model sentence structure. Linguists find that such structures help them understand hierarchical structures which often times lead to ungrammatical or ambiguous speech[2].

For our purposes of modeling these sentences, we find it useful to represent different words as members of different classes: nouns, verbs, adjectives, etc. A simplistic way of representing a words class is as follows:

```
type word =
    | Noun of string
    | Verb of string
    | Adjective of string
    | Determiner of string
    | Adjunct of string
```

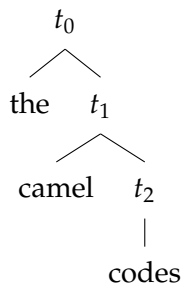
Here, we've defined a simplified structure: we define Determiners here as a member of the closed set (in English) {the, that, this}, and Adjuncts as any other kind of modifier (prepositions, adverbs, etc).

1. In English, simple declarative sentences always require a verb and at least a subject. Create a syntax tree type with a **flat-tree** structure, that allows for simple sentences like "The camel codes". Under the flat tree structure, all words should be 'sisters'—that is, no word should be nested under another (i.e, not recursive).
2. What drawbacks does this method of representing a sentence have? Why might a flat tree structure not make sense for representing a sentence? What kind of sentences does this structure make sense for?

[2]Such cases go beyond this course. If you are interested, Ling 102 goes into this very topic!

3. Rather than the flat tree structure, modern theoretical syntax advocates for a recursive structure, known as X-bar Theory. Under this structure, Linguists recognize that certain relationships between words are hierarchical: for example, there are certain effects that the subject in a sentence has over the verb and object (Linguists call these binding constraints but you don't need to worry about that).

   We can reorganize our answer from (1.) to implement these hierarchical structures by adding in the concept of **complements**—other tree structures which are joined with a word at the same level. Under this structure, a sentence like "The camel codes" might look like:



   Complements can be thought of as our recursive step! Restructure your answer from (1) to match the diagram above.

4. What does this model do better than in (1.)? What implications does it have for hierarchy? Are there any sentences that aren't possible under this model that should be? Give an example if one exists!