
CS 421 – Spring 2007

Lecture Notes Set 11:

User Defined Recursive Types

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 06-userdef (slides 22 - end)

Made available: February 12, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Recursive Types

So far, we have seen variant types as an enumeration type, which keep track of different categories of things. We have seen the disjoint type, which enables one to mix together integers, strings, floats, and even another set of integers and still be able to keep the information separate. Then, one can write functions over them with cases. We also saw that we could make polymorphic types. Now, we will take a look at recursive types. You have already seen a list, and lists fall under the category of variants. It has two constructors, `nil` for the empty list and `cons`, which takes an element and a list and returns a new list.

2.1 Creating a recursive type (slide 22)

You can create a type by putting together some existing types, possibly different ones, possibly the same. Then in your mix, have one or more instances of a recursive type (and it doesn't have to be at top level, it can be inside of another type constructor). So for example, if you were making a type called `splat`. Then one of your constructs can be for a `splat` list. Lists are recursive, and your constructor for it calls for the type `splat`. So you are creating a type that requires itself be complete. And then you can continue to have some simple items just like in enumeration types.

2.2 An example (slides 23-25)

This is an example of a typical recursive data type. We can create an integer binary tree by saying that it has two constructors. The first is for a leaf that contains an integer. Then there is a node. And the nodes simply carries two branches, the left and the right. Internally, it is a pair of integer binary trees, and that makes sense, because when looking at a binary tree, each node has its subtree with a connection from two branches. Notice that we can easily change this code to be of floats instead of integers, or instead of just two branches, we can leave the binary tree behind and make a general tree.

```
# type int_Bin_Tree =  
  Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree);  
type int_Bin_Tree = Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
```

We have a tree that has nodes and leads to leaves at the bottom of the tree. So if we wanted to use this type and wanted to make a binary tree, this is how we might create one.

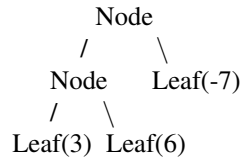
```
# let bin_tree =
```

¹© 2007, Share and Enjoy

```
Node(Node(Leaf 3, Leaf 6),Leaf (-7));
val bin_tree : int_Bin_Tree = Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```

What we have here is going to be called `bin_tree`. It starts with a node, the root, that has its two branches. One of the branches leads to another node while the other leads to a leaf (that holds -7). The inner node's two branches both lead to leaves (one that nodes 3 and the other holds 6). By paying attention to the commas one can see what leaf corresponds to which node.

So, visually, `bin_tree` may look like this:



3 Recursive Functions

3.1 Slight Overview

Now that we can write these recursive types, we need to be able to recurse over it. To do that, we write a recursive function.

We use two things we need for these recursive functions. First is pattern matching. So in this particular case, we need to tell if what we are looking at is a leaf or a node. The other thing we need is to know how to recurse. What should we do when we see a node? What should we do when we see a leaf?

Basically, the idea here is, when we see a node, we will need to check the nodes left side and the node's right side. When we see a leaf, we know there is nothing left to see.

3.2 An example (slide 26)

The first example we will look at is a function that finds the first left most leaf in the integer binary tree type, we created.

```
# let rec first_leaf_value tree =
  match tree with (Leaf n) → n
  | Node (left_tree, right_tree) →
    first_leaf_value left_tree;;
val first_leaf_value : int_Bin_Tree → int = <fun>
```

What happens here is the function takes a tree in as an argument and we check to see if that tree is a leaf or a node. Since we are looking for the left most leaf (the 'first' leaf in the tree) once we do find it, we can simply return the integer value it holds. If we see a node, we match its branches with the recursive call on the left branch (since that is what we care about. And because of that, we could have easily rewritten this case with an `_` instead of matching the right subtree with `'right_tree'`).

So if we were to use the tree we made earlier with this function, the function would travel down until it saw the left most leaf which held the value 3, and the 3 would be returned.

3.3 Mapping over Recursive Types (slides 27-28)

Now, as with lists and options, we can map with recursive types. So, with the tree as an example, we would want to take the leaves and perform some function on them to change their values, but not their structure. So we can write a recursive function that will take in a function and a tree and for every leaf it finds, it will perform that argument function on the value held by the leaf and then give the leaf back. The leaf is still a leaf, it just holds a new value now.

```
# let rec ibtreeMap f tree =
  match tree with (Leaf n) → Leaf (f n)
  | Node (left_tree, right_tree) →
    Node (ibtreeMap f left_tree,
```

```

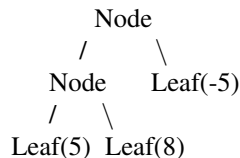
    ibtreeMap f right_tree);;
    val ibtreeMap : (int → int) → int.Bin_Tree → int.Bin_Tree = <fun>

```

So, in `ibtreeMap`, `f` is the argument function that takes an integer as its argument and returns an integer (usually, we would want our trees to be more general, aka polymorphic, but here we are simply looking at integers). Then there is the argument tree. In the match function, when we see a leaf, we match the leaf's value with the new value we get after performing the function `f` and put that value into the leaf. In the case we see a node, we match its branches with a recursive call on both sides of the node. So we call `ibtreeMap` again, giving it the function `f` and the subtrees, and we put those results back into node.

So if you were to use the addition operation with the integer 2, as your function and we gave `ibtreeMap` the `bin_tree` we made earlier, we would get the results with all the leaves having 2 added to the values. The leaf with 3 now holds 5, the leaf with 6 now holds 8, and the leaf with -7 now holds -5.

So, visually, `bin_tree` may look like this:



The structure of the tree has stayed the same, the only difference is the values in the leaves have been altered.

3.4 Folding over Recursive Types (slides 29-30)

Just like we have done in the past, we can define folding operations over other recursive types. So the tree we have can have a folding operation. Basically we will be taking the recursive result from the left subtree and putting it together with the recursive result from the right subtree.

```

# let rec ibtreeFoldRight leafFun nodeFun tree =
  match tree with Leaf n → leafFun n
  | Node (left_tree, right_tree) →
    nodeFun
      (ibtreeFoldRight leafFun nodeFun left_tree)
      (ibtreeFoldRight leafFun nodeFun right_tree);;
val ibtreeFoldRight : (int → 'a) → ('a → 'a → 'a) → int.Bin_Tree → 'a = <fun>

```

More specifically, in folding right over a tree, we need a function that tells us what to do when we see a leaf. We need a function that tells us what to do when we see a node. And then we need the tree itself. In side the fold function, when we do see a leaf, we match the value held in the leaf with the leaf function being performed with the value as the argument. So the type of `leafFun` is a function that takes an integer and returns something of type `'a`. Basically say that the function can do anything with that integer as long as the the `nodeFun` can take it as an argument.

Now when we see a node, we match that with applying the `nodeFun` with two arguments of type `'a`. We get those two `'a` arguments by getting the left and right recursive results. To get those results we do the recursive call on the left subtree with the same argument functions and then do the same with the right subtree.

Note that the match case for the node requires a pair of the subtrees but what it matches to, does not. That is because of types. A node is a pair of subtrees, but the `nodeFun` is a function that takes in two separate arguments, so no comma is needed between the two recursive calls.

If you were to use this fold function on `bin_tree` with the leaf case returning the value inside the leaf and the node case adding the values returned from the subtrees together, we are basically saying to add all the values in the leaves together. So with `bin_tree` $3 + 6 + (-7)$ turns out to be 2.

4 Mutually Recursive Types

The end level that we can push variants to involves two things. First of all, with recursive types, there was always the option of using polymorphism. We could have made a generic `bin_tree` instead of the `int_bin_tree`. This generic tree would have had its leaves hold `'a` instead of the specific type of `int`.

There is one more thing we would like to be able to do, and that is write family of mutually recursive types.

4.1 Another tree example (slide 31)

So an example of this would be if you wanted to make a tree but not have it restricted to only having two branches per node. You want this tree to have an arbitrary number of branches per node. Then you could do this by combining it with a list.

```
# type 'a tree = TreeLeaf of 'a
  | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree
  | More of ('a tree * 'a treeList);;
type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList)
```

You can say that node took a list of subtrees. To make sure that the empty list was not an option you can combine a mutually recursive type where we have a tree that is a leaf or a node of a treeList. And we can specify that treeList to be either a singleton of a tree or a More of a tree paired with a treeList. Basically, the treeList is encoded as a list that has one or more elements in it. So in other words, your tree is using the type treeLists and the treeList is using the type tree. So they are recursive together and apart, do not make much sense.

This is very useful when writing BNF grammars. When writing a grammar, there are usually many syntactic categories and there is a recursion between them. For example, in OCaml, you saw that a local declaration which contained an expression. But to make a that declaration, you had to bind an identifier to an expression. So to know what an expression is, you need to know what a declaration is. And to know what a declaration is, you need to know what an expression is. So you have a mutual dependency between the two.

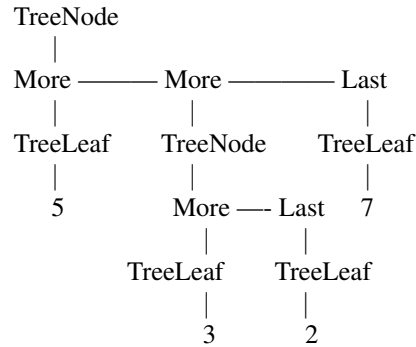
4.2 Using a mutually recursive type (slides 32-35)

Here we are going to declare a tree that has the type of int tree.

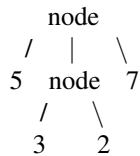
```
# let tree =
  TreeNode
  (More (TreeLeaf 5,
    (More (TreeNode
      (More (TreeLeaf 3,
        Last (TreeLeaf 2))),
      Last (TreeLeaf 7)))));;
val tree : int tree =
  TreeNode
  (More (TreeLeaf 5,
    More (TreeNode
      (More (TreeLeaf 3,
        Last (TreeLeaf 2))),
      Last (TreeLeaf 7))))
```

If you wanted to see how this was drawn, well you start out with the first treeNode. That treeNode has a More that has a branch leading to a treeLeaf with value 5 and a branch that leads to another More. That second More, contains a treeNode with a branch leading to a third More. The third More has a branch that leads to a treeLeaf with the value 3 and has another branch that leads to a Last, that has a treeLeaf with value 2. Now going back to the second More, that second More has a branch leading to a Last that is a treeLeaf and holds the value of 7.

If taken literally, it may look like this:



If taken like a conventional graph, we may have something like this:



4.3 Functions over mutually recursive types (slides 36-37)

Now the question is, how are we to write functions over these mutually recursive types. Well, the answer is with mutually recursive functions. For example, with the tree we created, we will write a recursive function over the tree and over the treeList. That way, we will be able to recurse over the whole structure. So the one over the tree will call the one over treeLists and the one over treeLists is going to call the one over trees.

So let's say we want to find all the values held in the leaves of the tree. We write a function that takes a tree and starts out with matching. If it finds a leaf, it returns a list with just the value of the leaf inside the list. If we find a node, we call the function on lists. Then in the list function, if we see a last then we know we need to go back to the tree function. If we see a More, we will recursively call the tree function and append what that returns onto the list we have been accumulating by recursively calling the list_fringe function. Basically, the More has the head of the list, called tree in this case, and the rest of the elements of the list, called list and we match against that. The tree is like a subtree of the node we were at when we called list_fringe.

```

# let rec fringe tree =
  match tree with (TreeLeaf x) → [x]
                 | (TreeNode list) → list_fringe list
and list_fringe tree_list =
  match tree_list with (Last tree) → fringe tree
                      | (More (tree,list)) →
                        (fringe tree) @ (list_fringe list);;
val fringe : 'a tree → 'a list = <fun>
val list_fringe : 'a treeList → 'a list = <fun>

```

After compiling, OCaml tells us that we created two functions, one that goes from a tree to a list and then one that goes from a treeList to a list.

Note the "and" keyword. It allows us to declare two things at once.

You cannot just have one function that tries to match for every case because a tree has types leaf and node while the list has last and more. Those are not interchangeable, so you have to make two functions to deal with different types. If you were to attempt to write one single function, it would be like trying to match a string to a float...it just won't work.

If we were to run this function on the tree we made earlier, we would get [5;3;2;7]

5 Nested Recursive Types

Any time you have mutual recursion, that can be replaced with nested recursion. When we created the tree type, what we could have done is made the `treeList` be completely polymorphic by instead of saying last was a 'a tree, we could have just put 'a. And then in the more statement, we could have put 'a * 'a `treeList`. If we did that, then later, we could have put in 'a tree, or something else if we wanted. Then we would have been able to recurse over the tree and the `treeList`.

With that we could have combined the polymorphism with nested recursion. That is, you break the types out one at a time, but have most of them be polymorphic.

5.1 Creating a nested recursive type (slide 38)

So here we have a type declaration for `labeled_tree`.

```
# type 'a labeled_tree =  
  TreeNode of ('a * 'a labeled_tree list);;  
type 'a labeled_tree = TreeNode of ('a * 'a labeled_tree list)
```

What this allows us to do is have values directly in our nodes. This type has one constructor. And that one constructor is recursive. This might seem odd at first, because it may bring one to wonder, where is the base case? How does the recursion stop?

For the base case, well that can be easily solved with the empty list. So any time you had a value in a node with the empty list, then that is the `labeled_tree`'s version of a leaf.

5.2 Using nested recursive types (slides 39-42)

```
# let ltree =  
  TreeNode(5,  
    [TreeNode (3, []);  
     TreeNode (2, [TreeNode (1, []);  
                       TreeNode (7, [])]);  
     TreeNode (5, [])]);;  
val ltree : int labeled_tree =  
  TreeNode(5,  
    [TreeNode (3, []); TreeNode (2, [TreeNode (1, []); TreeNode (7, [])]);  
     TreeNode (5, [])])
```

So what this is, is a node, the root, with the value 5, that has a list of nodes, a leaf of 3, a node of 2 that has two leaves of 1 and 7 and then a leaf of 5 at the end. Each leaf and node also has a branch leading to the empty list.

Conventionally, it looks like (for the unconventional graph, see slide 41):

```
  5  
 / | \  
3 2 5  
 / \  
1 7
```

5.3 Example function (slide 43-44)

Because we have the case, again, where we have two different types, in side of a structure, we will again, look at mutually recursive functions.

So here we have a function that finds all the values in a tree and puts them in a list. We start out with a tree and if we need a node we take the value we found and then recursively call the second function on the list of the node. In the second function, we are back to basic list matching; the empty list matches with the empty list, the head of the list is sent back to the first function while the tail list is recursed over by the second function.

```
# let rec flatten_tree labtree =  
  match labtree with TreeNode (x,treelist)
```

```

→ x::flatten_tree_list treelist
and flatten_tree_list treelist =
  match treelist with [] → []
  | labtree::labtrees
  → flatten_tree labtree
  flatten_tree_list labtrees;;
val flatten_tree : 'a labeled_tree → 'a list = <fun>
val flatten_tree_list : 'a labeled_tree list → 'a list = <fun>

```

So, running the function on the labeled tree we made, we get the list [5;3;2;1;7;5].

Notice, we could have written the second function with fold right. It is forward recursive. And because of that we could have used that as an inline function during the node matching in the first function, instead of the recursive call.

6 Infinite recursive Values

It is possible in OCaml, to make infinitely recursive objects. That is, a type that contains itself an infinite number of times.

6.1 A simple example (slide 49)

Here we have an infinite list of the integer 1.

```

# let rec ones = 1::ones;;
val ones : int list =
  [1; 1; 1; 1; ...]

```

Now, note that this does not mean that all of a computer's memory has 1 stored in it. This is saying that there is a listNode that contains the value 1. And the pointer from the node, points to itself. So every time you try to go to it's next, you end up with the same value and node.

So, if one wanted to do a match with this list, you can try: match ones with x::_ → x;; This will return 1. You may also get a warning saying that you matching isnt exhaustive, since you don't take into account the empty list. But that doesn't matter, since you will never have the empty list.

6.2 A little more complicated example (slides 46-end)

Here, we are making another infinitely recursive structure. And what this is doing is saying that we have a node with the value 2 and has a list. That list has two of the original nodes in it, which those new nodes have the original nodes, etc. And all of a sudden, we have an exponentially growing tree.

```

# let rec lab_tree = TreeNode(2, tree_list)
  and tree_list = [lab_tree; lab_tree];;

```

But again, we don't take up that much space. All it is, is a node with a 2, and then two branches that point back to the original two. So we can pattern match with this as well.

```

# match lab_tree
  with TreeNode (x, _) → x;;
- : int = 2

```

Here, we are asking for the value in the root, and that is what we get, the 2.