CS421 Lecture 5: User-Defined Datatypes in OCaml[1]

Mark Hills
mhills@cs.uiuc.edu

University of Illinois at Urbana-Champaign

June 9, 2008

[1] Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

1. Records
2. Variants
3. Recursive Types

Objectives

User-defined datatypes in OCaml will be heavily used in upcoming MPs and are a fundamental feature of the ML family of languages.

As a result of this lecture (plus MP2), you should...

- understand the distinctions between different user-defined data types
- know how to create new data types
- be able to manipulate data types with functions and pattern matching

Records

- Records serve then same programming purpose as tuples
- Provide better documentation, more readable code
- Allow components to be accessed by label instead of position
  - Labels (aka *field names*) must be unique
  - Fields accessed by suffix dot notation

Record Types

Record types must be declared before they can be used

```
1  # type person = {name : string; ss : (int * int * int);
2                   age : int};;
3  type person = { name : string; ss : int * int * int;
4                  age : int; }
```

- **person** is the type being introduced
- **name**, **ss** and **age** are the *labels*, or *fields*

Record Values

Records built with labels; order does not matter

```
1  # let teacher = {name = "Mark Hills";
2                   age = 102;
3                   ss = (123,45,6789)};;
4  val teacher : person =
5    {name = "Mark Hills"; ss = (123, 45, 6789); age = 102}
```

Record Values

```
1  # let student = {ss=(987,65,4321);
2                   name="Rebecca Hills";
3                   age=5};;
4  val student : person =
5    {name = "Rebecca Hills"; ss = (987, 65, 4321); age = 5}
6
7  # student = teacher;;
8  - : bool = false
```

Record Pattern Matching

```
1 # let {name = mark; age = age; ss = (_,_,s3)} = teacher;;
2 val mark : string = "Mark Hills"
3 val age : int = 102
4 val s3 : int = 6789
```

Record Field Access

```
1 # let soc_sec = teacher.ss;;
2 val soc_sec : int * int * int = (123, 45, 6789)
```

New Records from Old

```
1 # let birthday person = {person with age = person.age + 1};;
2 val birthday : person -> person = <fun>
3
4 # birthday teacher;;
5 - : person = {name = "Mark Hills"; ss = (123, 45, 6789);
6               age = 103}
```

New Records from Old

```
1 # let new_id name soc_sec person =
2   { person with name = name; ss = soc_sec };;
3   val new_id : string -> int * int * int ->
4                person -> person = <fun>
5
6 # new_id "John Smith" (321,54,9876) student;;
7 - : person = {name = "John Smith"; ss = (321, 54, 9876);
8               age = 5 }
```

Variants

- Variants, or variant records, provide for a single datatype with multiple forms
- Gives us a way to represent different types of data with one type name

Common examples:

- lists
- trees
- syntactic elements in a programming language

Variant Syntax

Variants are declared as *types* with *constructors*, which take zero or more data items; constructor names always start with caps

- no data with constructor name, constructor is a *constant*
- data types can include *polymorphic types* (more on this later)
- each constructor essentially a function from constructor's data to variant type

  $$\texttt{fun } \texttt{x}_1 \cdots \texttt{x}_n \texttt{-> C } \texttt{x}_1 \cdots \texttt{x}_n \ (\text{type } \texttt{t}_1 \cdots \texttt{t}_n \texttt{->} \ \textit{name})$$

- constructors the basis of almost all pattern matching

Variant Syntax

General syntax:

`type` *name* =	$\texttt{C}_1$	(* Constant *)
	$\texttt{C}_2$ `of` $\texttt{ty}_2$	(* Single data item *)
	$\texttt{C}_3$ `of` $\texttt{ty}_3 * \cdots * \texttt{ty}_n$	(* Multiple data items *)

Enumerations

- Enumerations are variants where all constructors are constants
- Essentially forms a collection of distinct elements (days of the week, months of the year, etc)
- Unlike in C, enumerations in OCaml are *unordered*

Enumerations: An Example

```
1  # type weekday = Monday | Tuesday | Wednesday
2      | Thursday | Friday | Saturday | Sunday;;
3  type weekday =
4        Monday
5      | Tuesday
6      | Wednesday
7      | Thursday
8      | Friday
9      | Saturday
10     | Sunday
```

Functions over Enumerations

```
1 # let day_after day = match day with
2       Monday -> Tuesday
3    | Tuesday -> Wednesday
4    | Wednesday -> Thursday
5    | Thursday -> Friday
6    | Friday -> Saturday
7    | Saturday -> Sunday
8    | Sunday -> Monday;;
9  val day_after : weekday -> weekday = <fun>
```

Functions over Enumerations

```
1 # let rec days_later n day =
2     match n with 0 -> day
3   | _ -> if n > 0
4           then day_after (days_later (n - 1) day)
5          else days_later (n + 7) day;;
6 val days_later : int -> weekday -> weekday = <fun>
```

Functions over Enumerations

```
# days_later 2 Tuesday;;
- : weekday = Thursday

# days_later (-1) Wednesday;;
- : weekday = Tuesday

# days_later (-4) Monday;;
- : weekday = Thursday
```

Disjoint Union Types

- Variants with constructors including one or more data items are *disjoint unions* of data types
- Data types can be repeated between constructors
- Some constructors can be constants (singletons)

Disjoint Union Types

```
1  # type id = DriversLicense of int
2             | SocialSecurity of int
3             | Name of string;;
4  type id = DriversLicense of int
5          | SocialSecurity of int
6          | Name of string
7
8  # let check_id id = match id with
9        DriversLicense num ->
10        not (List.mem num [13570; 99999])
11     | SocialSecurity num -> num < 900000000
12     | Name str -> not (str = "John Doe");;
13  val check_id : id -> bool = <fun>
```

Polymorphism in Variants

The type 'a option gives us something to represent non-existence or failure

```
1 # type 'a option = Some of 'a | None;;
2 type 'a option = Some of 'a | None
```

- Used to encode partial functions
- Often can replace the raising of an exception

Functions over Option

```
1  # let rec first p list =
2      match list with [ ] -> None
3      | (x::xs) -> if p x then Some x else first p xs;;
4  val first : ('a -> bool) -> 'a list -> 'a option = <fun>
5
6  # first (fun x -> x > 3) [1;3;4;2;5];;
7  - : int option = Some 4
8
9  # first (fun x -> x > 5) [1;3;4;2;5];;
10 - : int option = None
```

Mapping over Variants

```
1  # let optionMap f opt =
2      match opt with
3      | None -> None
4      | Some x -> Some (f x);;
5  val optionMap : ('a -> 'b) -> 'a option -> 'b option = <fun>
6
7  # optionMap
8    (fun x -> x - 3)
9    (first (fun x -> x > 3) [1;3;4;2;5]);;
10 - : int option = Some 1
```

Folding over Variants

```
1 # let optionFold someFun noneVal opt =
2     match opt with
3     | None -> noneVal
4     | Some x -> someFun x;;
5 val optionFold : ('a -> 'b) -> 'b -> 'a option -> 'b = <fun>
6
7 # let optionMap f opt =
8     optionFold (fun x -> Some (f x)) None opt;;
9 val optionMap : ('a -> 'b) -> 'a option -> 'b option = <fun>
```

Recursive Types

The type being defined may be a component of itself

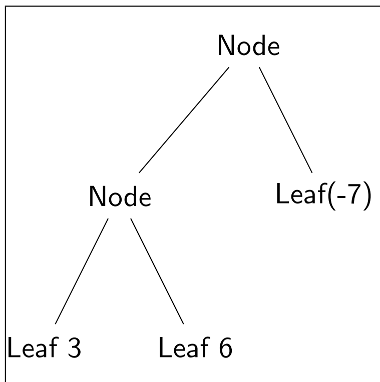
```
1 # type mylist = Nil
2         | Cons of int * mylist;;
3   type mylist = Nil | Cons of int * mylist
```

Recursive Data Types

```
1 # type int_Bin_Tree =
2  Leaf of int
3  | Node of (int_Bin_Tree * int_Bin_Tree);;
4
5 type int_Bin_Tree = Leaf of int
6   | Node of (int_Bin_Tree * int_Bin_Tree)
```

Recursive Data Type Values

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



Recursive Functions

```
1  # let rec first_leaf_value tree =
2      match tree with
3      | (Leaf n) -> n
4      | Node (left_tree, right_tree) ->
5       first_leaf_value left_tree;;
6  val first_leaf_value : int_Bin_Tree -> int = <fun>
7
8  # let left = first_leaf_value bin_tree;;
9  val left : int = 3
```

Mapping over Recursive Types

```
1   # let rec ibtreeMap f tree =
2       match tree with
3       | (Leaf n) -> Leaf (f n)
4       | Node (left_tree, right_tree) ->
5        Node (ibtreeMap f left_tree,
6              ibtreeMap f right_tree);;
7   val ibtreeMap :
8     (int -> int) -> int_Bin_Tree -> int_Bin_Tree = <fun>
9
10  # ibtreeMap ((+) 2) bin_tree;;
11  - : int_Bin_Tree = Node (Node (Leaf 5, Leaf 8), Leaf (-5))
```

Folding over Recursive Types

```
1 # let rec ibtreeFoldRight leafFun nodeFun tree =
2     match tree with
3     | Leaf n -> leafFun n
4     | Node (left_tree, right_tree) ->
5       nodeFun
6       (ibtreeFoldRight leafFun nodeFun left_tree)
7       (ibtreeFoldRight leafFun nodeFun right_tree);;
8 val ibtreeFoldRight :
9   (int -> 'a) -> ('a -> 'a -> 'a) -> int_Bin_Tree -> 'a = <fun>
```

Folding over Recursive Types

```
1 # let tree_sum =
2     ibtreeFoldRight (fun x -> x) (+);;
3 val tree_sum : int_Bin_Tree -> int = <fun>
4
5 # tree_sum bin_tree;;
6 - : int = 2
```

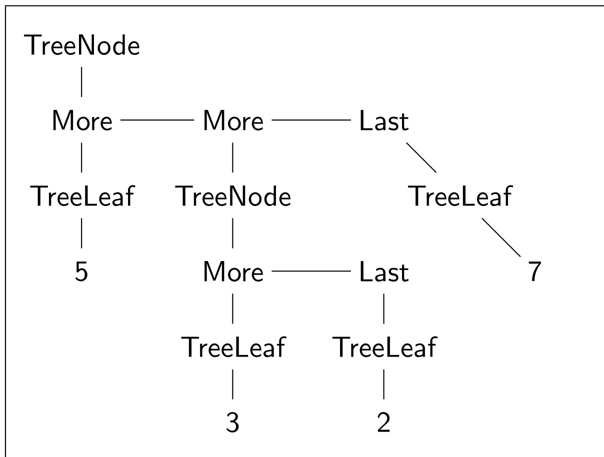
Mutually Recursive Types

```
1 # type 'a tree = TreeLeaf of 'a
2     | TreeNode of 'a treeList
3 and 'a treeList = Last of 'a tree
4     | More of ('a tree * 'a treeList);;
5
6 type 'a tree = TreeLeaf of 'a
7              | TreeNode of 'a treeList
8 and 'a treeList = Last of 'a tree
9                 | More of ('a tree * 'a treeList)
```

Values of Mutually Recursive Types

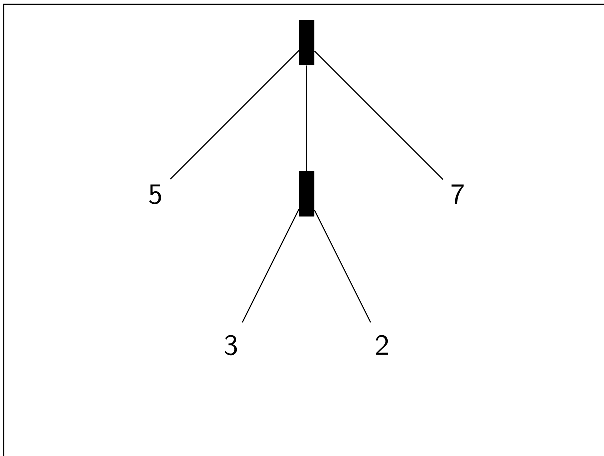
```
1  # let tree =
2     TreeNode
3      (More (TreeLeaf 5,
4            (More (TreeNode
5                  (More (TreeLeaf 3,
6                        Last (TreeLeaf 2))),
7                  Last (TreeLeaf 7)))));;
8   val tree : int tree =
9    TreeNode
10    (More
11      (TreeLeaf 5,
12       More
13        (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
14                              Last (TreeLeaf 7))))
```

Values of Mutually Recursive Types



Values of Mutually Recursive Types

A more conventional picture...



Mutually Recursive Functions

```
# 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>

# fringe tree;;
: int list = [5; 3; 2; 7]
```

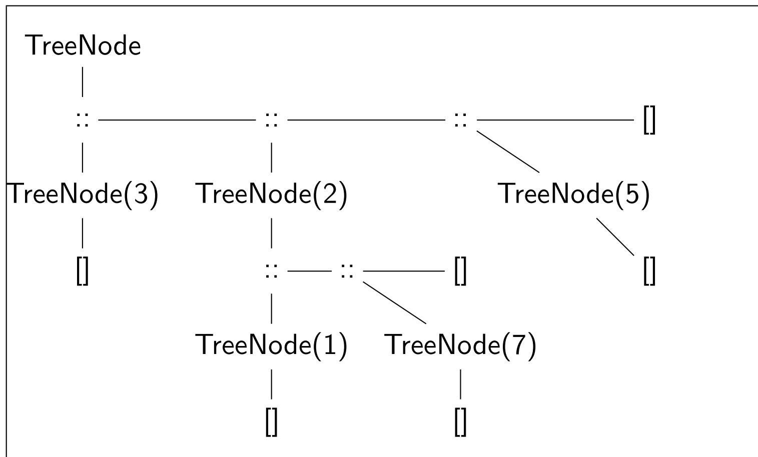
Nested Recursive Types

```
1 # type 'a labeled_tree =
2  TreeNode of ('a * 'a labeled_tree list);;
3 type 'a labeled_tree =
4   TreeNode of ('a * 'a labeled_tree list)
```

Nested Recursive Type Values

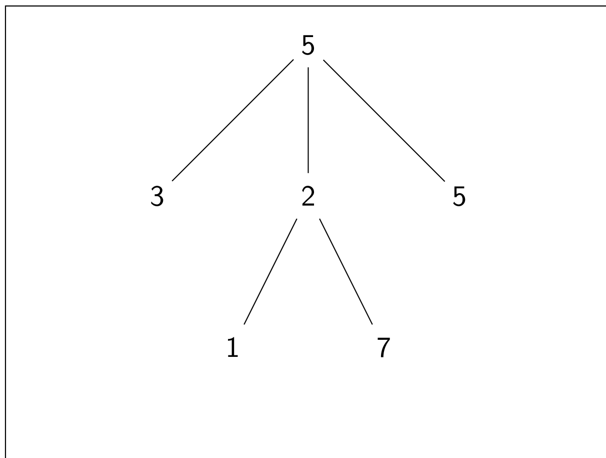
```
1  # let ltree =
2    TreeNode(5,
3      [TreeNode (3, []);
4       TreeNode (2, [TreeNode (1, []);
5                     TreeNode (7, [])]);
6       TreeNode (5, [])]);;
7  val ltree : int labeled_tree =
8    TreeNode
9     (5,
10     [TreeNode (3, []);
11      TreeNode (2, [TreeNode (1, []); TreeNode (7, [])]);
12      TreeNode (5, [])])
```

Values of Nested Recursive Types



Values of Nested Recursive Types

A more conventional picture...



Mutually Recursive Functions

```
1  # let rec flatten_tree labtree =
2      match labtree with
3        | TreeNode (x,treelist)
4          -> x::flatten_tree_list treelist
5    and flatten_tree_list treelist =
6      match treelist with
7        | [] -> []
8        | labtree::labtrees
9          -> flatten_tree labtree
10            @ flatten_tree_list labtrees;;
```

Mutually Recursive Functions

```
val flatten_tree :
  'a labeled_tree -> 'a list = <fun>
val flatten_tree_list :
  'a labeled_tree list -> 'a list = <fun>

# flatten_tree ltree;;
- : int list = [5; 3; 2; 1; 7; 5]
```

Nested recursive types lead to mutually recursive functions

Infinite Recursive Values

```
1  # let rec ones = 1::ones;;
2  val ones : int list =
3    [1; 1; 1; 1; ...]
4
5  # match ones with x::_ -> x;;
6  Characters 0-25:
7  Warning: this pattern-matching is not exhaustive.
8  Here is an example of a value that is not matched:
9  []
10   match ones with x::_ -> x;;
11   ^^^^^^^^^^^^^^^^^^^^^^^^^
12 - : int = 1
```

Infinite Recursive Values

```
1  # let rec lab_tree = TreeNode(2, tree_list)
2      and tree_list = [lab_tree; lab_tree];;
3  val lab_tree : int labeled_tree =
4    TreeNode (2, [TreeNode(...); TreeNode(...)])
5  val tree_list : int labeled_tree list =
6    [TreeNode (2, [TreeNode(...); TreeNode(...)]);
7     TreeNode (2, [TreeNode(...); TreeNode(...)])]
8
9  # match lab_tree
10     with TreeNode (x, _) -> x;;
11 - : int = 2
```