CSCI 2041: Data Types in OCaml

Chris Kauffman

Last Updated:
Thu Oct 18 22:43:05 CDT 2018
Logistics

Reading
- OCaml System Manual: Ch 1.4, 1.5, 1.7
- Practical OCaml: Ch 5

Goals
- Tuples
- Records
- Algebraic / Variant Types

Assignment 3 multimanager
- Manage multiple lists
- Records to track lists/undo
- option to deal with editing
- Higher-order funcs for easy bulk operations
- Post tomorrow
- Due in 2 weeks

Next week
First-class / Higher Order Functions
Despite being an older functional language, OCaml has a wealth of aggregate data types.

The table below describes some of these with some characteristics.

We have discussed Lists and Arrays at some length.

We will now discuss the others.

<table>
<thead>
<tr>
<th>Elements</th>
<th>Typical Access</th>
<th>Mutable</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>Homoegenous</td>
<td>Index/PatMatch</td>
<td>No</td>
</tr>
<tr>
<td>Array</td>
<td>Homoegenous</td>
<td>Index</td>
<td>Yes</td>
</tr>
<tr>
<td>Tuples</td>
<td>Heterogeneous</td>
<td>PatMatch</td>
<td>No</td>
</tr>
<tr>
<td>Records</td>
<td>Heterogeneous</td>
<td>Field/PatMatch</td>
<td>No/Yes</td>
</tr>
<tr>
<td>Variant</td>
<td>Not Applicable</td>
<td>PatMatch</td>
<td>No</td>
</tr>
</tbody>
</table>

Note: Data types can be nested and combined in any way:

- Array of Lists, List of Tuples
- Record with list and tuple fields
- Tuple of list and Record
- Variant with List and Record or Array and Tuple
Tuples

- Potentially mixed data
- Commas separate elements
- Tuples: pairs, triples, quadruples, quintuples, etc.
- Parentheses conventional but not required
- **No general indexing functions**: only `fst` and `snd` for pairs
- Generally use Pattern Matching to extract elements
- Type notation: separate types by asterisk *

```ocaml
# let int_pair = (1,2);;
val int_pair : int * int = (1, 2)

# let mixed_triple = (1,"two",3.0);;
val mixed_triple : int * string * float =
         (1, "two", 3.)

# let mixed_pair = ("a",5);;
val mixed_pair : string * int = ("a", 5)
# fst mixed_pair;;
- : string = "a"
# snd mixed_pair;;
- : int = 5

#fst mixed_triple;;
Error: This expression has type int * string * float but an expression was expected of type 'a * 'b

# match mixed_triple with
     | (a,b,c) -> a;;
- : int = 1

# match mixed_triple with
     | (a,b,c) -> c;;
- : float = 3.
```
Why Tuples?

- Arrays and Lists require homogeneous elements (all same kind)
- Records / Variants require declaration ahead of time
- Tuples are heterogeneous (different kinds) and built-in
- Useful for functions to return multiple items with differing types
- Ex: Returns mixed pair of string * int
- Ex: Pattern matches func arg as a pair

```ocaml
1 (* Return the longest string and its length from the list given. If the list is empty return ("",0) *)
2 let longest_string strlist =
3   let rec help (max_str,max_len) list =
4     match list with
5     | [] -> (max_str,max_len)
6     | str :: tail ->
7     let len = String.length str in
8     if len > max_len then
9     help (str,len) tail
10    else
11     help (max_str,max_len) tail
12   in
13   help ("",0) strlist
14 ;;

# longest_string ["Mario"; "Toad"; "Princess"; "Luigi"];;
- : string * int = ("Princess", 8)

# longest_string ["Bowser"; "Wario"; "Boo"; "Waluigi"; "Koopa"];;
- : string * int = ("Waluigi", 7)
```
Tuple Restrictions

- Tuples ALWAYS have a known cardinality: 2 or 3 or 8 etc.
- Lists/Arrays do not have a known length
- A function cannot take a pair OR a triple: must be one or the other, same with return values
- Cannot expand or grow tuples: a ref to a pair will always refer to a pair
- Cannot index tuples by number: must pattern match them so impractical for more than 4-5 items
Exercise: Tuple Warm-up

- How does one declare a tuple generally?
- Declare the following tuples
  - Pair hitch of int 42 and string "life"
  - Quadruple nums of 1 2 3 4
  - Triple of thresh float 1.23 boolean false int 123
- How do you access the first/second element of hitch?
- How do you access the third element of thresh?
let hitch = (42,"life") in
let nums = (1,2,3,4) in
let thresh = (1.23,false,123) in
let first = fst hitch in
let second = snd hitch in
let third =
    match thresh with
    | (a,b,c) -> c
in
();;
Pattern Matching Tuples

1 (* Pattern match a pair of booleans, return a relevant string *)
2 let boolpair_str bpair =
3     match bpair with
4         | true,true      -> "all true"
5         | false,false    -> "all false"
6         | true,false     |
7         | false,true     |
8         | false,true     -> "mixed bag"
9     ;;

(* Pattern match a pair of lists to determine which is longer *)
let rec longer_list lista listb =
    match lista, listb with
    | [], []         -> "same length"
    | _, []         -> "a is longer"
    | [], _         -> "b is longer"
    | (a::atail), (b::btail) ->
    |     longer_list atail btail
    ;;

▶ Extremely useful for destructuring multiple inputs together (like two sorted lists being merged)

▶ Can be exhaustive on tuple (boolean example)

▶ Or can use catch-alls / underscore to match anything for a tuple element (list example)
Exercise: Min-Max of a List

- Write minmax, returns the minimum and maximum elements of an arbitrary list
- Returns min/max as a pair (2-tuple)
- On empty list inputs, use failwith "empty list" to raise an exception
- Exploit pattern matching as much as possible, likely 2 layers deep
  - List structure
  - Relation of element to min/max
- Tail Recursive OR Not are both fine

REPL demo of minmax

```ocaml
# minmax;;
- : 'a list -> 'a * 'a = <fun>

# minmax [3];;
- : int * int = (3, 3)

# minmax [5;3];;
- : int * int = (3, 5)

# minmax [5;3;4;1;2;9;7];;
- : int * int = (1, 9)

# minmax ["c";"x"];;
- : string * string = ("c", "x")

# minmax ["v";"c";"x";"r";"q"];;
- : string * string = ("c", "x")

# minmax ["v";"c";"r";"x";"q";"y"];;
- : string * string = ("c", "y")
```
1 (* Returns min/max of a list as a pair. *)
2 let rec minmax list =
3     match list with
4     | [] -> failwith "empty list" (* empty list fail *)
5     | last :: [] -> (last,last) (* base case: 1 element *)
6     | head :: tail -> (* recursive case *)
7     let (min,max) = minmax tail in (* recurse, then match results *)
8     match (head < min),(head > max) with
9     | false,false -> (min,max) (* head in the middle *)
10    | true,false -> (head,max) (* head is smaller *)
11    | false,true -> (min,head) (* head is bigger *)
12    | true,true -> (head,head) (* both? stranger things... *)
13 ;;
14 (* Same as above with tail recursiv helper function *)
15 let rec minmax list =
16     match list with
17     | [] -> failwith "empty list"; (* empty list fail *)
18     | first :: rest -> (* peel off first element *)
19     let rec help (min,max) lst = (* define TR helper *)
20         match lst with
21         | [] -> (min,max) (* end of list *)
22         | head :: tail -> (* keep going *)
23         match (head < min),(head > max) with
24         | false,false -> help (min,max) tail
25         | true,false -> help (head,max) tail
26         | false,true -> help (min,head) tail
27         | true,true -> help (head,head) tail
28         in
29         help (first,first) rest;; (* call helper *)
30 ;;
Records

- Hetergeneous with named fields, Like C struct / Java object
- Introduced via the type keyword, each field is given a type
- Constructed with {...}, assign each field

```ocaml
# type hobbit = {name : string; age : int};; (* two fields *)
type hobbit = { name : string; age : int; }

# let bilbo = {name="Bilbo Baggins"; age=111};;
val bilbo : hobbit = {name = "Bilbo Baggins"; age = 111}

# let sam = {name="Samwise Gamgee"; age=21};;
val sam : hobbit = {name = "Samwise Gamgee"; age = 21}

# type ring = {
  number : int;
  power : float;
  owner : string;
};;
type ring = { number : int; power : float; owner : string; }

# let nenya = {number=3; power=5000.2; owner="Galadriel"};;
val nenya : ring = {number = 3; power = 5000.2; owner = "Galadriel"}

# let one = {number=1; power=9105.6; owner="Sauron"};;
val one : ring = {number = 1; power = 9105.6; owner = "Sauron"}
```
Basic Record Use

- Dot notation is used to access record field values

  ```
  # sam.age;;
  - : int = 21
  # sam.name;;
  - : string = "Samwise Gamgee"
  # nenya.power;;
  - : float = 5000.2
  ```

- Records and their fields are immutable by default

  ```
  # sam.age <- 100;;
  Characters 0-14:
  sam.age <- 100;;
  ^^^^^^^^^^^^^^^
  Error: The record field age is not mutable
  # sam.age = 100;;
  - : bool = false
  # sam;;
  - : hobbit =
  {name = "Samwise Gamgee"; age = 21}
  ```

- Create new records using with syntax to replace field values

  ```
  # let old_sam = {sam with age=100};;
  val old_sam : hobbit =
  {name = "Samwise Gamgee"; age = 100}
  # let lost_one = {one with
  owner="Bilbo";
  power=1575.1};;
  val lost_one : ring =
  {number = 1; power = 1575.1;
  owner = "Bilbo"}
  ```

- Fields declared mutable are changeable using <- operator

  ```
  # type mut_hob = {
  mutable name : string; (*changable*)
  age : int (*not*)
  };;
  # let h = {name="Smeagol"; age=25};;
  val h: mut_hob = {name="Smeagol"; age=25};
  # h.name <- "Gollum"; (* assignment *)
  - : unit = ()
  # h;;
  - : mut_hob = {name="Gollum"; age=25}
  ```
Exercise: Define two Record Functions

```ml
# let hobs = [ {m_name="Frodo"; age=23};
   {m_name="Merry"; age=22};
   {m_name="Pippin"; age=25}; ];;;

val hobbit_bdays : mut_hob list -> mut_hob list = <fun>
(* DEFINE: creates a new list of mut_hob with all ages incremented by 1 *)

# let older_hobs = hobbit_bdays hobs;;
val older_hobs : mut_hob list =
  [{m_name = "Frodo"; age = 24}; (* new list; ages updated *)
   {m_name = "Merry"; age = 23}; (* distinct from old list *)
   {m_name = "Pippin"; age = 26}]

val hobbit_fellowship : mut_hob list -> unit = <fun>
(* DEFINE: name of each hobbit has the string "Fellow" prepended to it so that "Frodo" becomes "Fellow Frodo" *)

# hobbit_fellowship hobs;; (* changes original list of hobs *)
- : unit = ()

# hobs;; (* show changed names *)
- : mut_hob list =
  [{m_name = "Fellow Frodo"; age = 23};
    {m_name = "Fellow Merry"; age = 22};
    {m_name = "Fellow Pippin"; age = 25}]```
Answers: Define two Record Functions

1 (* DEFINE: creates a new list of mut_hob with all ages incremented by 1 *)
2 let rec hobbit_bdays (list : mut_hob list) =
3   match list with
4     | [] -> []
5     | hob :: tail ->
6       {hob with age=hob.age+1} :: (hobbit_bdays tail)
7   ;;
8
9 (* DEFINE: name of each hobbit has the string "Fellow" prepended to it so
10   that "Frodo" becomes "Fellow Frodo" *)
11 let rec hobbit_fellowship (list : mut_hob list) =
12   match list with
13     | [] -> ()
14     | hob :: tail ->
15       hob.m_name <- "Fellow "^hob.m_name;
16       hobbit_fellowship tail;
17   ;;

<table>
<thead>
<tr>
<th>hobbit_bdays</th>
<th>hobbit_fellowship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses with : new records</td>
<td>uses &lt;- : old records, new field values</td>
</tr>
<tr>
<td>Uses cons operator: new list</td>
<td>Does NOT use cons, same list</td>
</tr>
<tr>
<td>NOT tail recursive</td>
<td>IS tail recursive</td>
</tr>
</tbody>
</table>
Refs are Just Mutable Records

- Have seen that OCaml’s ref allows for mutable data
- These are built from Records with a single mutable field
- Examine myref.ml which constructs the equivalent of standard refs in a few lines of code

```ocaml
type 'a myref = {mutable contents : 'a};;
```

- **Notable:** a polymorphic record
  - Field `contents` can be any type
    - `int ref` or `string list ref` etc.
- File includes `make_ref`, `deref`, `assign` functions which are `ref x, !x, x := y`
- Shows how to bind symbols like `:=` to functions though not how to determine if they are infix/prefix
Algebraic / Variant Data Types

Following strange construct appeared in week 1

type fruit = (* create a new type *)
  Apple | Orange | Grapes of int;; (* 3 value kinds possible *)

let a = Apple;; (* bind a to Apple *)
let g = Grapes(7);; (* bind g to Grapes *)

let count_fruit f = (* function of fruit *)
  match f with (* pattern match f *)
  | Apple -> 1 (* case of Apple *)
  | Orange -> 1 (* case of Orange *)
  | Grapes(n) -> n (* case of Grapes *)
;;

▶ As with records, type introduces a new type
▶ fruit is an **Algebraic** or **Variant** type
▶ Has exactly 3 kinds of values
  ▶ Apple and Orange which have no additional data
  ▶ Grapes which has an additional int of data
▶ Closest C/Java equivalent: **enumerations** (i.e. enum)
▶ OCaml's take on this is different and more powerful
Algebraic Types Allow Mixtures

- An algebraic type is just one type however its variants may have different kinds of data associated with them
- Allows mixed list/array as data is housed in a unified type

```plaintext
1 (* Establish a type that is either an int or string *)
2 type age_name =
3   | Age of int (* Age constructor takes an int *)
4   | Name of string (* Name constructor takes a string *)
5 ;;
6
7 (* Construction of individual age_name values *)
8 let i = Age 21;; (* construct an Age with data 21 *)
9 let s = Name "Sam";; (* construct a Name with data "Sam" *)
10 let j = Age 15;;
11
12 (* age_name list to demonstrate how they are the same type and can therefore be in a list together. *)
13 let mixed_list = [
14   Age 1;
15   Name "Two";
16   Age 3;
17   Name "Four";
18 ];;
```
Pattern Matching and Algebraic Types

- Pattern matching is used extensively with algebraic types
- The below function pattern matches on a age_name list
- Note use of list AND variant destructuring

```ocaml
1 (* Establish a type that is either an int or string *)
2 type age_name =
3 | Age of int (* Age constructor takes an int *)
4 | Name of string (* Name constructor takes a string *)
5 ;;

6 (* Sum all the Age data in the given age_name list *)
7 let rec sum_ages list =
8   match list with
9     | [] -> 0 (* base case *)
10    | (Age i)::tail -> (* have an age with data i *)
11       i + (sum_ages tail) (* add i onto recursive call *)
12    | _ :: tail -> (* must be a Name *)
13       sum_ages tail (* don’t add anything *)
14 ;;

# sum_ages;;
- : age_name list -> int = <fun>
# sum_ages [Age 1; Name "Two"; Age 3; Name "Four"; Age 5];;
- : int = 9
```
Exercise: Sum Lengths of age_name

Define the following function

```ocaml
let rec sum_lengths list = <fun>
(* Sum the "lengths" of Ages and Names. Length of an Age is 1; Length of a Name is the string length of the associated data. *)
```

# sum_lengths [];;
- : int = 0
# sum_lengths [Age 4];;
- : int = 1
# sum_lengths [Name "bugger"];;
- : int = 6
# sum_lengths [Age 4; Name "bugger"];;
- : int = 7
# sum_lengths [Age 4; Name "bugger"; Age 2];;
- : int = 8
# sum_lengths [Age 4; Name "bugger"; Age 2; Name "bug"];;
- : int = 11

▶ In match/with destructure both list and data variants Age and Name to deal with them separately
▶ Age a elements contribute 1
▶ Name n elements contribute String.length n
Answers: Sum Lengths of age_name

15 (* Sum the "lengths" of Ages and Names. Length of an Age is 1; Length
16 of a Name is the string length of the associated data. *)
17 let rec sum_lengths list =
18 match list with
19 | [] -> 0
20 | (Age _)::tail -> (* don’t need data for age *)
21 | (Name n) :: tail -> (* do need data for name *)
22 | (String.length n) + (sum_lengths tail) (* add on length of name *)
23 ;;
An Interesting Algebraic Type: 'a option

- Ocaml has a built-in type called option which is defined roughly as
  ```plaintext
  type 'a option = None | Some of 'a;;
  ```
- Type is **polymorphic**
  ```plaintext
  # let iopt = Some 5;;
  val iopt : int option = ...
  # let bopt = Some false;;
  val bopt : bool option = ...
  # let stropt_list = [None; Some "dude"; Some "sweet"];;
  val stropt_list : string option list = ...
  ```

- option used to indicate presence or absence of something, often in function return values

- Compare assoc and assoc_opt operations on association lists
  ```plaintext
  (* An association list *)
  # let alist = [("a",5); ("b",10)];;
  val alist : (string * int) list = ...
  (** assoc: return element or raise exception *)
  # List.assoc "b" alist;;
  - : int = 10
  # List.assoc "z" alist;;
  Exception: Not_found.
  (** assoc_opt: return Some or None to indicate failure *)
  # List.assoc_opt "a" alist;;
  - : int option = Some 5
  # List.assoc_opt "z" alist;;
  - : int option = None
  ```
Exercise: Implement assoc_opt

Below is code for assoc from Lab04. Alter it to fulfill the requirements of assoc_opt

1 (* Return the value associated with query key in association list alist. Raises a Not_found exception if there is no association *)
2 let rec assoc query alist =
3   match alist with
4   | [] -> raise Not_found (* not found *)
5   | (k,v)::tail when query=k -> v (* found *)
6   | _::tail -> assoc query tail (* recurse deeper *)
7 ;;
8 (* Find association of query key in given association list. Return (Some value) if found or None if not found. *)
9 let rec assoc_opt query alist =
Answers: Implement assoc_opt

```ocaml
let rec assoc query alist =
  match alist with
  | [] -> raise Not_found (* not found *)
  | (k,v)::tail when query=k -> v (* found *)
  | _::tail -> assoc query tail (* recurse deeper *)

let rec assoc_opt query alist =
  match alist with
  | [] -> None (* not found *)
  | (k,v)::tail when query=k -> Some v (* found *)
  | _::tail -> assoc_opt query tail (* recurse deeper *)
```

- Change empty list case to `None` rather than exception
- Change found case to `Some v`
Exercise: Counting Some

- Implement the following two functions on option lists
- Both solution have very similar recursive structure

```ocaml
count_some : 'a option list -> int = <fun>
(* Count how many times a (Some _) appears in the 'a option list *)

sum_some_ints : int option list -> int = <fun>
(* Sum i’s in all (Some i) that appear in the int option list *)
```

```ocaml
# count_some [];;
- : int = 0
# count_some [None; None];;
- : int = 0
# count_some [Some 5];;
- : int = 1
# count_some [Some "a"; None; Some "b"; None; None; Some "c"];;
- : int = 3

# sum_some_ints [];;
- : int = 0
# sum_some_ints [None; None];;
- : int = 0
# sum_some_ints [Some 2];;
- : int = 2
# sum_some_ints [Some 2; None; Some 4; Some 9; Some 3; None];;
- : int = 18
```
Answers: Counting Some

1 (* Count how many times a (Some _) appears in a list of options *)
2 let rec count_some opt_list =
3     match opt_list with
4         | [] -> 0
5         | None::tail -> count_some tail
6         | (Some _):tail -> 1 + (count_some tail)
7     ;;

10 (* Sum all (Some i) options that appear in the list *)
11 let rec sum_some_ints opt_list =
12     match opt_list with
13         | [] -> 0
14         | None::tail -> sum_some_ints tail
15         | (Some i):tail -> i + (sum_some_ints tail)
16     ;;
Consider code in `opt_v_exc.ml` which underscores the differences in style between `assoc` and `assoc_opt`.

Exception version crashes when something is not found.

Many built-in operators/functions have these two alternatives:

1. Return an option: found as `Some v`, not found as `None`.
2. Return found value directly or raise a `Not_found` exception.

Will contrast these more later when discussing exception handling.
Lists are Algebraic Types

- OCaml's built-in list type is based on Algebraic types
- The file `alg_lists.ml` demonstrates how one can re-create standard lists with algebraic types (but don't do that)
- Note the use of type parameter in `a mylist`: can hold any type of data so it is a polymorphic data type
- Note also the **type is recursive** referencing itself in `Cons`

```ocaml
1  type 'a mylist =         (* type parameter *)
2   | Empty                   (* end of the list *)
3   | Cons of ('a * 'a mylist) (* an element with more list *)
4  ;;
5
6 (* construct a string list *)
7 let list1 = Cons ('a', Cons('b', Cons('c', Empty))));;
8
9 (* construct a boolean list *)
10 let list2 = Cons (true, Cons(false, Cons(true, Cons(true, Cons(true, Empty))))));;
11
12 (* function that calculates the length of a mylist *)
13 let rec length_ml list =
14     match list with
15     | Empty -> 0
16     | Cons (_,tail) -> 1 + (length_ml tail)
17  ;;
```
Uses for Algebraic Types: Tree Structures

- In the future we will use Algebraic Types in several major ways
- Will study functional data structures, rely heavily on trees
- Algebraic types give nice null-free trees

```ocaml
type strtree =
  | Bottom (* no more tree *)
  | Node of string * strtree * strtree (* data with left/right tree *)
;;
let empty  = Bottom;;
let single = Node ("alone",Bottom,Bottom);;
let small  = Node ("Mario",
                  Node("Bowser",
                      Bottom,
                      Node("Luigi",
                          Bottom,
                          Bottom)),
                  Node("Princess",
                      Bottom,
                      Bottom));
```
Uses for Algebraic Types: Lexer/Parser Results

- In the future we will use Algebraic Types in several major ways
- Will study converting a text stream to an executable program
- Usually done in 2 phases: lexing and parsing
- Both usually employ algebraic types

```ocaml
let input = "5 + 9*4 + 7*(3+1)";;  (* Lexing: convert this string.. *)
let lexed = [Int 5; Plus; Int 9;  (* Into this stream of tokens *)
    Times; Int 4; Plus;
    Int 7; Times;
    OParen; Int 3; Plus;
    Int 1; CParen];;
let parsed = (* Parsing: convert lexed tokens.. *)
    Add(Const(5),  (* Into a semantic data structure, *)
        Add(Mul(Const(9),  (* in this case a tree reflecting the *)
            Const(4)),  (* order in which expressions should *)
        Mul(Const(7),  (* be evaluated. Intrepretation involves *)
            Add(Const(3),  (* walking the tree to compute a *)
                Const(1))))))  (* result. Compilation converts the tree *)
    ;;  (* into a linear set of instructions. *)
```
Algebraic Extras

Multiple Type Params

- Records and Algebraic types can take type parameters as in
  \[
  \text{type 'a option = None | Some of 'a;};
  \]
- Shows up less frequently but can use multiple type parameters
  \[
  \text{type ('a, 'b) thisthat = This of 'a | That of 'b;};
  \]
- File thisthat.ml explores this a little but is not required reading
- Will make use of multiple type params for polymorphic Maps and Hashtables

Anonymous Records in Algebraic Types

- Algebraic types can have any kind of data, typically tuples of different kinds
- Anonymous records with named fields are also allows
- Relatively new feature of OCaml, helps to document data in type

\[
\text{type fieldtree =}
| \text{Bot} (* no fields *)
| \text{Nod of \{data : string; (* anonymous record with data *)}
  left : fieldtree; (* left and *)
  right : fieldtree} (* right fields *)
\]

let tree = ...;;

let rootdata = match tree with (* assign data from root node *)
  | Bot -> "" | Nod(n) -> n.data
;;