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

Overview of Aggregate Data Structures / Types in OCaml

- Despite being an older functional language, OCaml has a wealth of aggregate data types
- The table below describe some of these with some characteristics
- We have discussed Lists and Arrays at some length
- We will now discuss the others

	Elements	Typical Access	Mutable	Example
Lists	Homoegenous	Index/PatMatch	No	`[1;2;3]`
Array	Homoegenous	Index	Yes	`[|1;2;3|]`
Tuples	Heterogeneous	PatMatch	No	`(1,"two",3.0)`
Records	Heterogeneous	Field/PatMatch	No/Yes	`{name="Sam"; age=21}`
Variant	Not Applicable	PatMatch	No	`type letter = A | B | C;`

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 *

```
# 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

```
(* Return the longest string and its
   length from the list given. If the
   list is empty return ("",0) *)
let longest_string strlist =
  let rec help (max_str,max_len) list =
    match list with
    | [] -> (max_str,max_len)
    | str :: tail ->
       let len = String.length str in
       if len > max_len then
         help (str,len) tail
       else
         help (max_str,max_len) tail
  in
  help ("",0) strlist
;;
```

```
# 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`?

Answers: Tuple Warm-up

```
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

```
(* Pattern match a pair of booleans,
   return a relevant string *)
let boolpair_str bpair =
  match bpair with
  | true,true   -> "all true"
  | false,false -> "all false"
  | true,false
  | false,true  -> "mixed bag"
;;

(* 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`

```
# 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")
```

Answers: Min-Max of a List

```
(* Returns min/max of a list as a pair. *)
let rec minmax list =
  match list with
  | [] -> failwith "empty list"      (* empty list fail *)
  | last :: [] -> (last,last)        (* base case: 1 element *)
  | head :: tail ->                  (* recursive case *)
     let (min,max) = minmax tail in  (* recurse, then match results *)
     match (head < min),(head > max) with
     | false,false -> (min,max)      (* head in the middle *)
     | true,false  -> (head,max)     (* head is smaller *)
     | false,true  -> (min,head)     (* head is bigger *)
     | true,true   -> (head,head)    (* both? stranger things... *)
;;
(* Same as above with tail recursiv helper function *)
let rec minmax list =
  match list with
  | [] -> failwith "empty list";     (* empty list fail *)
  | first :: rest ->                 (* peel off first element *)
     let rec help (min,max) lst =    (* define TR helper *)
       match lst with
       | [] -> (min,max)             (* end of list *)
       | head :: tail ->             (* keep going *)
          match (head < min),(head > max) with
          | false,false -> help (min,max)   tail
          | true,false  -> help (head,max)  tail
          | false,true  -> help (min,head)  tail
          | true,true   -> help (head,head) tail
     in
     help (first,first) rest;;       (* call helper *)
;;
```

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

```
# 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 = {                                   (* three fields *)
    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=
# h.name <- "Gollum";; (* assignment *
- : unit = ()
# h;;
- : mut_hob = {name="Gollum"; age=25}
```

Exercise: Define two Record Functions

```
# let hobs = [ {m_name="Frodo";  age=23};       (* list of hobbits *)
               {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

```
(* DEFINE: creates a new list of mut_hob with all ages incremented by 1 *)
let rec hobbit_bdays (list : mut_hob list) =
  match list with
  | [] -> []
  | hob :: tail ->
     {hob with age=hob.age+1} :: (hobbit_bdays tail)
;;

(* DEFINE: name of each hobbit has the string "Fellow" prepended to it so
   that "Frodo" becomes "Fellow Frodo" *)
let rec hobbit_fellowship (list : mut_hob list) =
  match list with
  | [] -> ()
  | hob :: tail ->
     hob.m_name <- "Fellow "^hob.m_name;
     hobbit_fellowship tail;
;;
```

`hobbit_bdays`	`hobbit_fellowship`
Uses `with` : new records	uses `<-` : old records, new field values
Uses cons operator: new list	Does NOT use cons, same list
NOT tail recursive	IS tail recursive

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

  ```
  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

```
(* Establish a type that is either an int or string *)
type age_name =
  | Age  of int                  (* Age constructor takes an int *)
  | Name of string               (* Name constructor takes a string  *)
;;

(* Construction of individual age_name values   *)
let i = Age 21;;                  (* construct an Age with data 21 *)
let s = Name "Sam";;              (* construct a Name with data "Sam" *)
let j = Age 15;;

(* age_name list to demonstrate how they are the same type and can
   therefore be in a list together. *)
let mixed_list = [
    Age 1;
    Name "Two";
    Age 3;
    Name "Four";
];;
```

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

```
(* Establish a type that is either an int or string *)
type age_name =
  | Age  of int                      (* Age constructor takes an int *)
  | Name of string                   (* Name constructor takes a string  *)
;;
```

```
(* Sum all the Age data in the given age_name list *)
let rec sum_ages list =
  match list with
  | [] -> 0                          (* base case *)
  | (Age i)::tail ->                 (* have an age with data i *)
     i + (sum_ages tail)             (* add i onto recursive call *)
  | _ :: tail ->                     (* must be a Name *)
     sum_ages tail                   (* don't add anything *)
;;
```

```
# 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

```
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

```
(* 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.  *)
let rec sum_lengths list =
  match list with
  | [] -> 0
  | (Age _)::tail ->                          (* don't need data for age *)
     1 + (sum_lengths tail)                   (* add 1 onto total *)
  | (Name n) :: tail ->                       (* do need data for name *)
     (String.length n) + (sum_lengths tail)   (* add on length of name *)
;;
```

An Interesting Algebraic Type: 'a option

- Ocaml has a built-in type called option which is defined roughly as

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

- Type is **polymorphic**

```
# 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

```
(* 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

```
(* Return the value associated with query key in association
   list alist.  Raises a Not_found exception if there is no
   association *)
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 *)
;;

(* Find association of query key in given association
   list. Return (Some value) if found or None if not found. *)
let rec assoc_opt query alist =
```

Answers: Implement `assoc_opt`

```
(* Return the value associated with query key in association
   list alist.  Raises a Not_found exception if there is no
   association *)
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 *)
;;

(* Find association of query key in given association
   list. Return (Some value) if found or None if not found. *)
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

```
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 *)

# 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

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


(* Sum all (Some i) options that appear in the list *)
let rec sum_some_ints opt_list =
  match opt_list with
  | [] -> 0
  | None::tail -> sum_some_ints tail
  | (Some i)::tail -> i + (sum_some_ints tail)
;;
```

Options vs Exceptions

- 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`

```
type 'a mylist =                    (* type parameter *)
  | Empty                           (* end of the list *)
  | Cons of ('a * 'a mylist)        (* an element with more list *)
;;

(* construct a string list *)
let list1 = Cons ("a", Cons("b", Cons("c", Empty)));;

(* construct a boolean list *)
let list2 = Cons (true, Cons(false, Cons(true, Cons(true, Empty))));;

(* function that calculates the length of a mylist *)
let rec length_ml list =
  match list with
  | Empty -> 0
  | Cons (_,tail) -> 1 + (length_ml tail)
;;
```

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

```
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

```
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
  `type 'a option = None | Some of 'a;;`
- Shows up less frequently but can use multiple type parameters
  `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

```
type fieldtree =
  | Bot                             (* no fields *)
  | 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
;;
```