Learning to Learn OCaml

Alexander Berenbeim

2016-10-25 Tuesday

Outline

Learning To Learn OCaml: Lecture III Review of extra work The `unit` Type Learning To Program With The Unit Type

Tail Recursive Functions

One way to write a *tail-recursive* function counting the number of `true` elements in a list is as follows:

```
let rec count_true l =
   match l with
     [] -> 0
   | true :: t -> 1+ count_true t
   | false :: t -> count_true t;;
```

member Function

- First, we recognize that member is of type α -> (α list -> bool)
- So we to define a function that takes a witness of α to a function from α list to a boolean value, i.e. member x is of type α list -> bool

```
let rec member elt l =
   match l with
     [] -> false
   | h :: t ->  ( h = elt ) || ( member elt t );;
```

make_set function

Now we make our first not-so-trivial application of a definable predicate to turn a list into a set

```
let rec make_set l =
   match l with
     []-> []
   | h :: t -> if member h t then make_set t
                             else h :: make_set t;;
```

mergeSort Function

```
# let rec merge x y =
    match x, y with
    [], l -> l
    | l, [] -> l
    | hx::tx, hy::ty -> if hx < hy
                        then hx :: ( merge tx (hy :: ty) )
                        else hy :: ( merge (hx :: tx) ty);;
let rec msort l =
    match l with
    [] -> []
    | [x] -> [x]
    | _ -> let left = take (length l / 2) l in
                let right = drop (length l / 2) l in
                     merge (msort left) (msort right);;
```

apply Function

```
let rec map f l =
match l with
[] -> []
| h :: t -> f h :: map f t;;
let rec apply f n x =
if n = 0 then x
else f ( apply f (n - 1) x );;
```

filter and mapl Functions

```
let rec filter f l =
   match l with
   [] -> []
   | h :: t -> if f h then h :: filter f t
                      else filter f t;;
```

Then, since a map which takes lists of α lists as an argument is of type (α -> β) -> α list list -> β list list

```
let rec mapl f l = map (map f) l;;
```

What is the unit Type?

- When OCaml prints to the screen with functions like print_int x or print_string x, the output will look like x- : unit = ()
- This function takes an integer as its argument, *prints* an integer *on* the screen, and then closes; there is no output
- That is, there is nothing that occurs to the arguments; they're mapped to the only witness of the unit type, ()
- The outputs on the screen are called side effects

What is the unit in relation to void?

- The unit type is similar, but distinct from the void type found in C.
- Recall the void type from C is the type for a result of functions that return, but do not provide, a result value to its caller.
- Both are used for their side effects, but the void type only simulates some of the properties of the unit type, as the void type can not be a type of argument in C, whereas functions like print_newline in OCaml are of type unit -> unit.
- There is also a difference in that the void is never stored in a record type, while the unit type can be stored.

Example 1 : Writing To The Screen

- We can produce several side effects by using the ; symbol, which evaluates the expression on the left handside, then tosses the result (so that an expression like `x;y` is thus of the same type as `y`).
- For example, to print an entire dictionary (say $d : \mathtt{int} \times \mathtt{string}$) to the screen::

```
let print_dict_entry ( k , v ) =
    print_int k ; print_newline () ;
    print_string v ; print_newline ();;
let rec iter f l =
    match l with
        [] -> ()
      | h :: t  -> f h ; iter f t;;
let print_dict d = iter print_dict_entry;;
```

Example 2 : Reading From The Keyboard

- Of course, printing things isn’t the only thing we do with computers; we want to input information from time to time as well!
- OCaml has built in functions that allow us to input values of int and string, read_int and read_string respectively, of type unit to int (or string).

```
let rec read_dict () =
   try
      let i = read_int () in
         if i = 0 then [] else
             let name = read_line () in
                (i, name) :: read_dict ()
   with
     Failure "int_of_string" ->
        print_string "This is not an integer. We’ll be here all
        print_newline ();
        read_dict () ;;
```

Example 3 : Using Files

- It is obviously inefficient to have to manually enter a new data set every time you want to call upon it.
- OCaml has basic functions that help to read and write from *places* that data is stored on our computer
- If a *place* is of the type in_channel, we can read from it; if a *place* is of the type out_channel, we can write to it.
- Importantly, OCaml does not pass types as types when data is being read or written, so OCaml passes integers as character arrays.
- We work around the lack of an output_int function by using the built in string_of_int function.
- There is also no output_newline function, so we use the special character '\n'
- The function open_out gives an output channel for a filename given by the input string and whenever called, must eventually be followed by close_out, if we ever want to close the file.
- On the next slide, we'll enter and store a dictionary

Example 3 : Writing To Files (Cont'd)

```
let entry_to_channel ch (k , v) = outout_string ch (string_of_i
                                  output_char ch '\n';
                                  output_string ch v;
                                  output_char ch '\n';;
let dictionary_to_channel ch d = iter (entry_to_channel ch) d;;
let dictionary_to_file filename dict =
     let ch = open_out filename in
         dictionary_to_channel ch dict;
         close_out ch;;
```

Example 4rgrg : Reading From Files (Cont'd)

```
let entry_of_channel ch =
    let number = input_line ch in
       let name = input_line ch in
          (int_of_string number, name);;
let rec dictionary_of_channel ch =
   try
      let e = entry_of_channel ch in
          e :: dictionary_of_channel ch
   with
     End_of_file -> [];;
let dictionary_of_file filename =
    let ch = open_in filename in
        let dict = dictionary_of_channel ch in
           close_in ch; dict;;
```

Table of Useful Functions

Function	Type	Description
print_int	int -> unit	prints an integer to the screen
print_string	string -> unit	prints a string
print_newline	unit -> unit	prints a new line to the screen
read_line	unit -> string	read a string from the user
read_int	unit -> unit	read an integer from the user
int_of_string	string -> int	makes an integer from a string raising =Failure "int$_{ofstring}$" in the event of an error
open_out	string -> out_channel	given a file name, opens a channel for output, rasing Sys_error if the file cannot be opened
close_out	string -> unit	closes the output channel (never forget)
open_in	string -> in_channel	opens a channel for input named by the given string
close_in	in_channel -> unit	close the input channel (never forget)
output_string	out_channel -> string -> unit	write a string to an output channel
output_char	out_channel -> char -> unit	write a character to an output channel