Imperative Programming in OCaml

CS496

Imperative Features in OCaml

OCaml (as seen so far) is purely functional

- every expression is evaluated solely for its value

This lack of side-effects has an important consequence

- purely functional languages are said to enjoy referential transparency
  - This means that the order in which subexpressions are evaluated, in some large expression, is irrelevant

As a result, one

- can use standard algebraic equations (eg. $a + b = b + a$) to reason about programs
- can easily parallelize

Imperative Features in OCaml

However sometimes imperative features are needed

- variable assignment and destructive update of data structures (specially for efficiency reasons)
- I/O: communication with some external device

Therefore, OCaml is enriched with expressions that are evaluated solely for their effects

- Reference types, assignment
- Mutable fields in records
- Arrays
- I/O: communication with some external device:

  `scanf`, `printf`, ...
- Many more

Examples

```
# let x=2
val x : int = 2

# let x=ref 2;;
val x : int ref = {contents = 2}
```

- `ref 2` denotes a location in memory (i.e. a memory address)
- ref is similar to malloc in C

Examples

```
# let x = ref 2 in !x;;
- : int = 2
# let x=ref 2 in x:=!x+1; !x;;
- : int = 3
# let x=ref 2;;
val x : int ref = {contents = 2}
# x;;
- : int ref = {contents = 2}
# !x;;
- : int = 2
# x:=!x+1;;
- : unit = ()
# !x;;
- : int = 3
# x:=!x+1;;
- : unit = ()
# !x;;
- : int = 4
```

Modeling a Counter Object

- Hidden state: value of counter
- First we declare the type of such an object, namely a record type

```
# type counter = { get : unit -> int;
                   set : int -> unit;
                   inc: unit->unit};;
type counter = { get : unit -> int; set : int ->
    ↪ unit; inc : unit -> unit; }
```

- Models the public interface of the object

Modeling a Counter Object

Then we define the object `c` itself:

```
# let c =
  let s = ref 0
  in { get = (fun () -> !s);
       set = (fun x -> s:=x);
       inc = (fun () -> s:=!s+1)};;
val c : counter = {get = <fun>; set = <fun>; inc = <
    ↪ fun>}
```

and interact with it

```
# c.get ();;
- : int = 0
# c.inc ();;
- : unit = ()
# c.get ();;
- : int = 1
# c.set 4;;
- : unit = ()
# c.get ();;
- : int = 4
```

Modeling a Function that Creates Counter Objects

```
# let newCounter n =
      let s = ref n
  in { get = (fun () -> !s);
       set = (fun x -> s:=x);
       inc = (fun () -> s:=!s+1)};;
val newCounter : int -> counter = <fun>
# let c1 = newCounter 1;;
val c1 : counter = {get = <fun>; set = <fun>; inc = <fun>}
# let c2 = newCounter 2;;
val c2 : counter = {get = <fun>; set = <fun>; inc = <fun>}
# c1.get();;
- : int = 1
# c2.get();;
- : int = 2
# c1.inc();;
- : unit = ()
# c2.get();;
- : int = 2
```

Modeling a Counter Object with `this`

```
# let newCounter n =
let s = ref n
in let rec this =
  { get = (fun () -> !s);
    set = (fun x -> s:=x);
    inc = (fun () -> s:=this.get ()+1)}
  in this;;
val newCounter : int -> counter = <fun>
# let c= newCounter 4;;
val c : counter = {get = <fun>; set = <fun>; inc = <
    ↪ fun>}
# c.get ();;
- : int = 4
# c.inc ();;
- : unit = ()
# c.get();;
- : int = 5
```

Example – Linked List of Integers

```ocaml
type node = { data: int;
                    mutable next: (node ref) option }

type llist = { mutable head: (node ref) option;
               mutable size :int }
```

- `option` allows a field to be `None`, representing a `null` reference

Example – Linked List of Integers

```
let create () =
  { head = None;
    size=0}

let add x ll =
  ll.head <- Some (ref {data=x; next=ll.head});
  ll.size<-ll.size+1;
  ll

let string_of_list ll =
  let rec string_of_node = function
    | None -> ""
    | Some r -> string_of_node (!r.next) ^ string_of_int (!r.data)
  in string_of_node (ll.head)
```

Exercises

Implement

- `get (i:int)(l:llist): int`
- `append (l1:llist)(l2:llist): llist`
- `is_circular (l:llist): bool`
- binary trees

Arrays.

```
# let a = [|1;2;3|];;
val a : int array = [|1; 2; 3|]
# a.(1);;
- : int = 2
# a.(1)<-4;;
- : unit = ()
# a;;
- : int array = [|1; 4; 3|]
```