Introduction to Objective Caml — Part 2

Stefan Wehr

University of Freiburg.

November 8, 2006

Reference Cells

- Remember: Variables are only *names* for values
- Can't assign to variables
- Solution: reference cells
- Constructor: `ref : 'a -> 'a ref`
- Type of reference cell holding values of type `t`: `t ref`
- Assignment operator: `:=`
- Dereference operator: `!`

Example

```
# let i = ref 1;;
val i : int ref = {contents = 1}
# i := 2;;
- : unit = ()
# !i;;
- : int = 2
```


Example

```
# let i = ref 1;;
val i : int ref = {contents = 1}
# i := 2;;
- : unit = ()
# !i;;
- : int = 2
```

We can have two different names for the same reference cell:

```
# let j = i;;
val j : int ref = {contents = 2}
# j := 5;;
- : unit = ()
# !i;;
- : int = 5
```

Pitfall

Don't confuse ! with boolean negation

```
# let flag = ref true;;
val flag : bool ref = {contents = true}
# if !flag then 1 else 2;;
- : int = 1
# if not (!flag) then 1 else 2;;
- : int = 2
```


Example: Imperative Queues

```
type 'a queue = 'a list ref (* enqueue list *)
              * 'a list ref (* dequeue list *)

let create () = (ref [], ref [])

let enqueue (eq, _) x = eq := x :: !eq

let rec dequeue ((eq, dq) as queue) =
  match !dq with
      x :: rest ->
       dq := rest; x
    | [] -> (* Shift the queue *)
      if !eq = [] then
         failwith "cannot dequeue empty queue"
      dq := List.rev !eq;
      eq := [];
      dequeue queue
```


Exceptions

```
# 1 / 0;;
Exception: Division_by_zero.
```

- Similar to exceptions in Java
- Signal a runtime error
- Can be catched
- Throwing an exception: `raise <some exception>`
- Catching an exception: `try ... with ...`
- Defining a new exception:
  `exception <Name> of <type>`

Example

```
# exception Empty_list of string;;
exception Empty_list of string
# let head = function
      [] -> raise (Empty_list "head: the list is empty")
    | x::_ -> x;;
val head : 'a list -> 'a = <fun>
# head [1;2;3];;
- : int = 1
# head [];;
Exception: Empty_list "head: the list is empty".
# let f l = try head l with
              Empty_list s -> print_endline s; 0;;
val f : int list -> int = <fun>
# f [1;2;3];;
- : int = 1
# f [];;
head: the list is empty
- : int = 0
```

Types for Exceptions

```
# Empty_list "head: the list is empty";;
- : exn = Empty_list "head: the list is empty"
# raise;;
- : exn -> 'a = <fun>
# 1 + raise (Empty_list "foo");;
Exception: Empty_list "foo".
```

- An exception has type `exn`. An `exception` definition *extends* this type.
- The `raise` function takes an exception and can produce *any* type because it never returns.

Important Builtin Exceptions

- `Failure : string -> exn`, signals some kind of failure.
- `Not_found : exn`, raised (for example) when a given element is not found in a data structure.
- `Invalid_argument : string -> exn`, raised when the argument to a function does not match the function's precondition.
- `Sys_error : string -> exn`, raised on a system call failure.

Exception Handlers

A `try ... with` expression can handle multiple exceptions:

```
# let some_computation () = ...;;
val some_computation : unit -> unit = <fun>
# try some_computation () with
      Sys_error s ->
        print_endline ("Sys_error: " ^ s)
    | Not_found ->
        print_endline "Not_found"
    | Empty_list s ->
        print_endline ("Empty_list: " ^ s);;
- : unit = ()
```


`finally`

- OCaml doesn't provide a `finally` (as in Java)
- But we can program it as a function

```
# type 'a result = Success of 'a | Failure of exn;;
type 'a result = Success of 'a | Failure of exn
# let finally f x cleanup =
    let result = try Success (f x) with e -> Failure e in
      cleanup();
      match result with
          Success y -> y
        | Failure e -> raise e;;
val finally : ('a -> 'b) -> 'a -> (unit -> 'c) -> 'b = <fun>
# let process in_channel = ...;;
val process : in_channel -> unit = <fun>
# let process_file fname =
    let chan = open_in fname in
      finally process chan (fun () -> close_in chan);;
val process_file : string -> unit = <fun>
```

Modules

- Features:
  - Namespace management
  - Decomposition of large programs into smaller units *(modules)*
  - Abstraction
  - Separate compilation
- Key parts of the OCaml module system:
  - Signature: defines the interface of a module
  - Structure: holds the implementation of a module
  - Functor: function over structures

Signatures

- Define the interfaces of modules
- Contain type definitions, abstract types, value definitions, . . .

```
module type IntSetSig =
  sig
    type elem = int
    type set
    val empty : set
    val member : elem -> set -> bool
    val insert : elem -> set -> set
  end
```


Structures

- Hold the implementations of modules
- Contain type definitions, value definitions, . . .

```
# module IntSet1 =
  struct
    type elem = int
    type set = elem list
    let empty   = []
    let member i s = List.exists ((=) i) s
    let insert i s = if member i s then s else (i :: s)
  end;;
module IntSet1 : sig
                   type elem = int
                   type set = elem list
                   val empty : 'a list
                   val member : 'a -> 'a list -> bool
                   val insert : 'a -> 'a list -> 'a list
                 end
```


Using Structures

- Access structure components through the dot notation

```
# let singleton_set =
    IntSet1.insert 1 IntSet1.empty;;
val singleton_set : int list = [1]
# IntSet1.member 1 singleton_set;;
- : bool = true
# IntSet1.member 0 singleton_set;;
- : bool = false
# IntSet1.member 0 [1;2;3];;
- : bool = false
```

Sealing

- Structure `IntSet1` reveals that sets are implemented as lists!
- Goal: Make the `set` type abstract
- Solution: Seal the structure with a signature where `set` is abstract

```
# module IntSet2 = (IntSet1 : IntSetSig);;
module IntSet2 : IntSetSig
# IntSet2.member 0 [1;2;3];;
This expression has type 'a list but is here
used with type IntSet2.set
```


Functors

- Until now: set implementation only works for integers
- Wanted: generic set implementation that abstracts over the element type and the equality comparison
- Solution: use functors which map structures to structures

```
# module MkSet =
  functor (E : sig type t val eq : t -> t -> bool end) ->
   (struct
      type elem = E.t
      type set = elem list
      let empty = []
      let member x s = List.exists (E.eq x) s
      let insert x s = if member x s then s else (x :: s)
    end : sig type elem =  E.t
              type set
              val empty : set
              val member : elem -> set -> bool
              val insert : elem -> set -> set end);;
```


Signatures for Functors

Signature inferred by the toplevel loop for the `MkSet` functor:

```
module MkSet :
  functor (E : sig type t val eq : t -> t -> bool end) ->
    sig
      type elem = E.t
      type set
      val empty : set
      val member : elem -> set -> bool
      val insert : elem -> set -> set
    end
```


Using Functors (1)

```
# module IntEq =
  struct
    type t = int
    let eq = (=)
  end;;
module IntEq : sig type t = int
                   val eq : 'a -> 'a -> bool end
# module IntSet3 = MkSet(IntEq);;
module IntSet3 :
  sig
    type elem = IntEq.t
    type set = MkSet(IntEq).set
    val empty : set
    val member : elem -> set -> bool
    val insert : elem -> set -> set
  end
```

Using Functors (2)

```
# module IntEqMod13 = struct
                         type t = int
                         let eq i j = i mod 13 = j mod 13
                       end;;
module IntEqMod13 : sig type t = int
                         val eq : int -> int -> bool end
# module IntSetMod13 = MkSet(IntEqMod13);;
module IntSetMod13 : sig
                        type elem = IntEqMod13.t
                        type set = MkSet(IntEqMod13).set
                        val empty : set
                        val member : elem -> set -> bool
                        val insert : elem -> set -> set
                      end
# let s = IntSetMod13.insert 1 IntSetMod13.empty;;
val s : IntSetMod13.set = <abstr>
# IntSetMod13.member 14 s;;
- : bool = true
```


Using Functors (3)

```
# module StringEqCase = struct
   type t = string
   let eq s s' = String.lowercase s = String.lowercase s'
  end;;
module StringEqCase : sig type t = string
             val eq : string -> string -> bool end
# module StringSetCase = MkSet(StringEqCase);;
module StringSetCase :
  sig type elem = StringEqCase.t
      type set = MkSet(StringEqCase).set
      val empty : set
      val member : elem -> set -> bool
      val insert : elem -> set -> set end
# let s = StringSetCase.insert "STEFAN"
            StringSetCase.empty;;
val s : StringSetCase.set = <abstr>
# StringSetCase.member "stefan" s;;
- : bool = true
```


Sharing Constraints (1)

Suppose we define `MkSet2` as follows:

```
# module type SetSig =
  sig type  elem
      type  set
      val   empty   :  set
      val   member  :  elem -> set -> bool
      val   insert  :  elem -> set -> set end;;
...
# module MkSet2 = functor
   (E : sig type t val eq : t -> t -> bool end) -> (struct
      type elem = E.t
      type set = elem list
      let empty = []
      let member x s = List.exists (E.eq x) s
      let insert x s = if member x s then s else (x :: s)
    end : SetSig)
module MkSet2 : functor
  (E: sig type t val eq : t -> t -> bool end) -> SetSig
```


Sharing Constraints (2)

Then we have a problem:

- Type `elem` in the result signature is abstract.
- Functor is useless.

```
# module IntSet4 = MkSet2(IntEq);;
module IntSet4 :
  sig
    type elem = MkSet2(IntEq).elem
    type set = MkSet2(IntEq).set
    val empty : set
    val member : elem -> set -> bool
    val insert : elem -> set -> set
  end
# IntSet4.insert 1 IntSet4.empty;;
This expression has type int but is here used with type
  IntSet4.elem = MkSet2(IntEq).elem
```

Sharing Constraints (3)

Solution: use a sharing constraint `with type` to propagate the `elem` type from the functor argument to the result signature.

```
# module MkSet3 = functor
  (E : sig type t val eq : t -> t -> bool end) -> (struct
      type elem = E.t
      type set = elem list
      let empty = []
      let member x s = List.exists (E.eq x) s
      let insert x s = if member x s then s else (x :: s)
    end : SetSig with type elem = E.t);;
module MkSet3 : functor
  (E : sig type t val eq : t -> t -> bool end) -> sig
      type elem = E.t
      type set
      val empty : set
      val member : elem -> set -> bool
      val insert : elem -> set -> set
    end
```


Sharing Constraints (4)

```
# module IntSet5 = MkSet3(IntEq);;
module IntSet5 :
  sig
    type elem = IntEq.t
    type set = MkSet3(IntEq).set
    val empty : set
    val member : elem -> set -> bool
    val insert : elem -> set -> set
  end
# IntSet5.insert 1 IntSet5.empty;;
- : IntSet5.set = <abstr>
```