

Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 5 - Sequence 6: *Variables*, also known as References



Variables in imperative programming languages

Consider this program fragment in C or Java:

```
int i = 0;  
i = i+1;
```

What is really going on

- ▶ the *variable* `i` is a name
- ▶ that refers to a *memory cell* (left of = sign)
- ▶ ... or to its *contents* (right of the = sign)
- ▶ this leads to complex notions, like *L-values* and *R-values*

Simulating variables in OCaml

We can use *mutable* records to get the same effect

```
1 int i = 0;  
2 i = i+1;
```

may be written

```
type refcell = {mutable content: int};;
```

```
let i = {content=0};;          (* line 1 *)
```

```
i.content <- i.content +1;;  (* line 2 *)
```

No ambiguities

The update operator `<-` clearly indicates what is read and what is written.

The predefined ref type

In *OCaml* there is a *predefined* `ref` type that works exactly this way. We call the type, as well as the instances, *reference*.

```
type 'a ref = {mutable contents:'a}
```

It comes with convenient syntactic support

- ▶ a function `ref: 'a -> 'a ref`
to **create** the reference
- ▶ a prefix operator `!`
to **read** the content of the reference
- ▶ an infix operator `r := v`
to **update** the content of the reference `r` with `v`

A simple variable I

```
let i = ref 0;;  
# val i : int ref = {contents = 0}
```

```
i;;  
# - : int ref = {contents = 0}
```

```
i := !i + 1;;  
# - : unit = ()
```

```
i;;  
# - : int ref = {contents = 1}
```

Computing the integer log in base two I

```
let log2int n =  
  let count = ref 0 and v = ref n in  
  while !v > 1 do  
    count := !count + 1;  
    v := !v/2  
  done;  
  !count;;  
# val log2int : int -> int = <fun>
```

```
log2int 16;;
```

```
# - : int = 4
```

```
log2int 1024;;
```

```
# - : int = 10
```

```
log2int 1000000;;
```

```
# - : int = 19
```

Reading a list of integers I

```
(* read a list of integers , and stop *)  
(* when a non integer is entered      *)
```

```
let read_intlist () =  
  (* a reference to hold the results *)  
  let l = ref [] in  
  (* the reading loop *)  
  let doread() =  
    try  
      while true do  
        l := (read_int ()):: !l  
      done  
    with _ -> ()  
  in
```

Reading a list of integers II

```
doread();  
List.rev !l  
;;  
# val read_intlist : unit -> int list = <fun>
```


Summary

- ▶ The usual *variables* of imperative languages can be implemented via *mutable* records
- ▶ The notion of memory cell, and of contents of memory cell, are clearly distinguished
- ▶ A special syntax is available for writing more concise programs using *references*

This concludes our short exploration of the imperative features of the *OCaml* language.