# (Side) Effects

## CS3100 Fall 2019

## Why Side Effects

- We have only used purely functional feature of OCaml
- Our study of lambda calculus used only purely functional features

- The above statements are lies
  - We have used `print_endline` , `printf` and other features to display our results to screen.
- It is sometimes useful to write programs that have **side effects**

## Side effects

Side effects include

- Mutating (i.e., destructively updating) the values of program state.
- Reading from standard input, printing to standard output.
- Reading and writing to files, sockets, pipes etc.
- ...
- Composing, sending and receiving emails, editing documents, writing this slide, etc.

## Side effects in OCaml

- OCaml programs can include side effects
- Features
  - Mutations: Reference cells, Arrays, Mutable record fields
  - I/O of all sorts
- In this lecture, **Mutations**

## Reference cells

- Aka "refs" or "ref cell"
- Pointer to a typed location in memory
- The binding of a variable to a pointer is immutable but the contents of the memory may change.

## Reference cells

In [31]:

```
let r = ref 0
```

Out[31]:

```
val r : int ref = {contents = 0}
```

In [32]:

```
r := !r + 1;
!r
```

Out[32]:

```
- : int = 1
```

## Reference Cells: Types

In [33]:

```
ref
```

Out[33]:

```
- : 'a -> 'a ref = <fun>
```

In [34]:

```
(!)
```

Out[34]:

```
- : 'a ref -> 'a = <fun>
```

In [35]:

```
(:=)
```

Out[35]:

```
- : 'a ref -> 'a -> unit = <fun>
```

# Implementing a counter

In [36]:

```
let make_counter init =
  let c = ref init in
  fun () ->
    (c := !c + 1; !c)
```

Out[36]:

```
val make_counter : int -> unit -> int = <fun>
```

In [37]:

```
let next = make_counter 0
```

Out[37]:

```
val next : unit -> int = <fun>
```

In [38]:

```
next()
```

Out[38]:

```
- : int = 1
```

# Side effects make reasoning hard

- Recall that referential transparency allows replacing $e$ with $v$ if $e \rightarrow_\beta v$.
- Side effects break referential transparency.

# Referential transparency

Consider the function `foo` :

In [39]:

```
let foo x = x + 1
```

Out[39]:

```
val foo : int -> int = <fun>
```

In [40]:

```
let baz = foo 10
```

Out[40]:

```
val baz : int = 11
```

`baz` can now be optimised to

In [41]:

```
let baz = 11
```

Out[41]:

```
val baz : int = 11
```

# Referential transparency

Consider the function `bar` :

In [42]:

```
let bar x = x + next()
```

Out[42]:

```
val bar : int -> int = <fun>
```

In [43]:

```
let qux = bar 10
```

Out[43]:

```
val qux : int = 12
```

Can we now optimise `qux` to:

In [44]:

```
let qux = 12
```

Out[44]:

```
val qux : int = 12
```

**NO.** Referential transparency breaks under side effects.

## Aliases

References may create aliases.

What is the result of this program?

In [45]:

```
let x = ref 10 in
let y = ref 10 in
let z = x in
z := !x + 1;
!x + !y
```

Out[45]:

```
- : int = 21
```

- `z` and `x` are said to be **aliases**
  - They refer to the same object in the program heap.

## Equality

- The `=` that we have been using is known as **structural equality**
  - Checks whether the values' structurally equal.
  - `[1;2;3] = [1;2;3]` evaluates to `true`.
- Because of references, one may also want to ask whether two expressions are **aliases**
  - This equality is known as **physical equality**.
  - OCaml uses `==` to check for physical equality.

## Equality

In [46]:

```
let l1 = [1;2;3];;
let l2 = l1;;
let l3 = [1;2;3];;
let r1 = ref l1;;
let r2 = r1;;
let r3 = ref l3;;
```

Out[46]:

```
val l1 : int list = [1; 2; 3]
```

Out[46]:

```
val l2 : int list = [1; 2; 3]
```

Out[46]:

```
val l3 : int list = [1; 2; 3]
```

Out[46]:

```
val r1 : int list ref = {contents = [1; 2; 3]}
```

Out[46]:

```
val r2 : int list ref = {contents = [1; 2; 3]}
```

Out[46]:

```
val r3 : int list ref = {contents = [1; 2; 3]}
```

# Equality

```
let l1 = [1;2;3];;
let l2 = l1;;
let l3 = [1;2;3];;
let r1 = ref l1;;
let r2 = r1;;
let r3 = ref l3;;
```

which of the following are true?

(1) `l1 = l2` (2) `l1 = l3` (3) `r1 == r2` (4) `l1 == l2`

(5) `r1 == r3` (6) `l1 == l3` (7) `r1 = r2` (8) `r1 = r3`

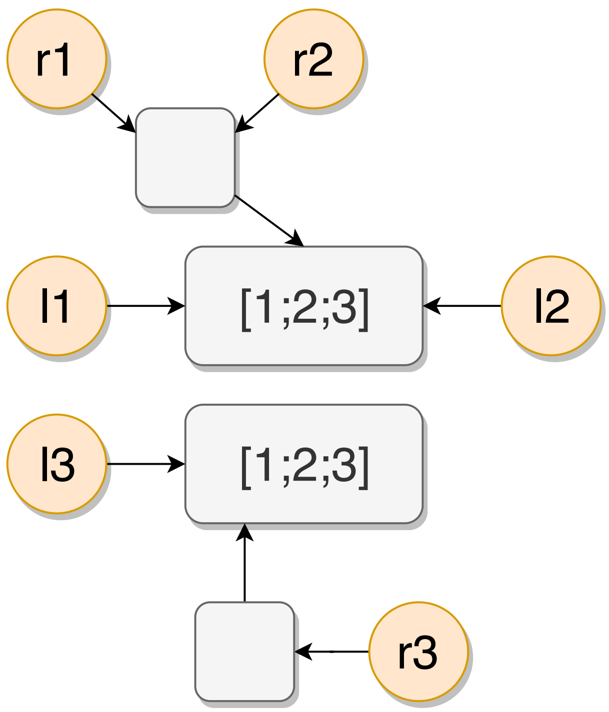
In [47]:

```
l1 = l2
```

Out[47]:

```
- : bool = true
```

## Equality




which of the following are true?

(1) `l1 = l2` (2) `l1 = l3` (3) `r1 == r2` (4) `l1 == l2`

(5) `r1 == r3` (6) `l1 == l3` (7) `r1 = r2` (8) `r1 = r3`

References are structurally equal iff their contents are structurally equal.

## Mutable Record Fields

Ref cells are essentially syntactic sugar:

```
type 'a ref = { mutable contents: 'a }
let ref x = { contents = x }
let ( ! ) r = r.contents
let ( := ) r newval = r.contents <- newval
```

- That type is declared in `Pervasives`
- The functions are compiled down to something equivalent

# Doubly-linked list

In [48]:

```
(* The type of elements *)
type 'a element = {
  value : 'a;
  mutable next : 'a element option;
  mutable prev : 'a element option
}

(* The type of list *)
type 'a dllist = 'a element option ref
```

Out[48]:

```
type 'a element = {
  value : 'a;
  mutable next : 'a element option;
  mutable prev : 'a element option;
}
```

Out[48]:

```
type 'a dllist = 'a element option ref
```

# Double-linked list

In [49]:

```
let create () : 'a dllist = ref None
let is_empty (t : 'a dllist) = !t = None

let value elt = elt.value

let first (t : 'a dllist) = !t
let next elt = elt.next
let prev elt = elt.prev
```

Out[49]:

```
val create : unit -> 'a dllist = <fun>
```

Out[49]:

```
val is_empty : 'a dllist -> bool = <fun>
```

Out[49]:

```
val value : 'a element -> 'a = <fun>
```

Out[49]:

```
val first : 'a dllist -> 'a element option = <fun>
```

Out[49]:

```
val next : 'a element -> 'a element option = <fun>
```

Out[49]:

```
val prev : 'a element -> 'a element option = <fun>
```

# Doubly-linked list

In [50]:

```
let insert_first (t:'a dllist) value =
  let new_elt = { prev = None; next = !t; value } in
  begin match !t with
  | Some old_first -> old_first.prev <- Some new_elt
  | None -> ()
  end;
  t := Some new_elt;
  new_elt
```

Out[50]:

```
val insert_first : 'a dllist -> 'a -> 'a element = <fun>
```

# Doubly-linked list

In [51]:

```
let insert_after elt value =
  let new_elt = { value; prev = Some elt; next = elt.next } in
  begin match elt.next with
  | Some old_next -> old_next.prev <- Some new_elt
  | None -> ()
  end;
  elt.next <- Some new_elt;
  new_elt
```

Out[51]:

```
val insert_after : 'a element -> 'a -> 'a element = <fun>
```

# Doubly-linked list

In [52]:

```
let remove (t:'a dllist) elt =
  let { prev; next; _ } = elt in
  begin match prev with
  | Some prev -> prev.next <- next
  | None -> t := next
  end;
  begin match next with
  | Some next -> next.prev <- prev;
  | None -> ()
  end;
  elt.prev <- None;
  elt.next <- None
```

Out[52]:

```
val remove : 'a dllist -> 'a element -> unit = <fun>
```

# Doubly-linked list

In [53]:

```
let iter (t : 'a dllist) f =
  let rec loop = function
    | None -> ()
    | Some el -> f (value el); loop (next el)
  in
  loop !t
```

Out[53]:

```
val iter : 'a dllist -> ('a -> 'b) -> unit = <fun>
```

# Doubly-linked list

In [54]:

```
let l = create ();;
let n0 = insert_first l 0;;
let n1 = insert_after n0 1;;
insert_after n1 2
```

Out[54]:

```
val l : '_weak2 dllist = {contents = None}
```

Out[54]:

```
val n0 : int element = {value = 0; next = None; prev = Non
e}
```

Out[54]:

```
val n1 : int element =
  {value = 1; next = None;
   prev = Some {value = 0; next = Some <cycle>; prev = Non
e}}
```

Out[54]:

```
- : int element =
{value = 2; next = None;
 prev =
  Some
   {value = 1; next = Some <cycle>;
    prev = Some {value = 0; next = Some <cycle>; prev = Non
e}}}
```

# Doubly-linked list

In [55]:

```
iter l (Printf.printf "%d\n%!")
```

```
0
1
2
```

Out[55]:

```
- : unit = ()
```

# Arrays

In [56]:

```
let a = [| 1;2;3 |]
```

Out[56]:

```
val a : int array = [|1; 2; 3|]
```

In [57]:

```
a.(2)
```

Out[57]:

```
- : int = 3
```

In [58]:

```
a.(1) <- 0;
a
```

Out[58]:

```
- : int array = [|1; 0; 3|]
```

In [59]:

```
a.(4)
```

```
Exception: Invalid_argument "index out of bounds".
Raised by primitive operation at unknown location
Called from file "toplevel/toploop.ml", line 180, character
s 17-56
```

## Benefits of immutability

- Programmer doesn't have to think about aliasing; can concentrate on other aspects of code
- Language implementation is free to use aliasing, which is cheap
- Often easier to reason about whether code is correct
- Perfect fit for concurrent programming

But

- Some data structures (hash tables, arrays, ...) are more efficient if imperative

# Fin.