(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

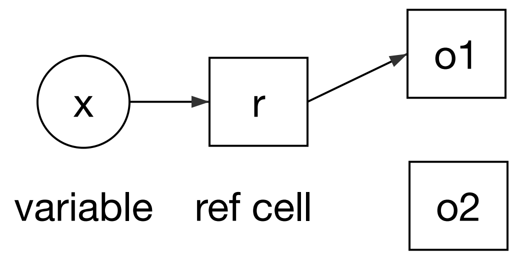
- Mutating (i.e., destructively updating) the values of program state.
- Reading from standard input, printing to standard output.
- Reading and writing to files, sockets, pipes etc.
- ...
- Composing, sending and receiving emails, editing documents, writing this slide, etc.

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 ref cell is immutable
  - but the contents of the ref cell may change.



Reference cells

In [ ]:

```
let r = ref 0
```

In [ ]:

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

Reference Cells: Types

In [ ]:

```
ref
```

In [ ]:

```
(!)
```

In [ ]:

```
(:=)
```

Implementing a counter

In [ ]:

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

In [ ]:

```
let next = make_counter 0
```

In [ ]:

```
next()
```

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 [ ]:

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

In [ ]:

```
let baz = foo 10
```

`baz` can now be optimised to

In [ ]:

```
let baz = 11
```

Referential transparency

Consider the function `bar` :

In [ ]:

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

In [ ]:

```
let qux = bar 10
```

Can we now optimise `qux` to:

In [ ]:

```
let qux = 12
```

NO. Referential transparency breaks under side effects.

Aliases

References may create aliases.

What is the result of this program?

In [ ]:

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

- `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 [ ]:

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

Equality

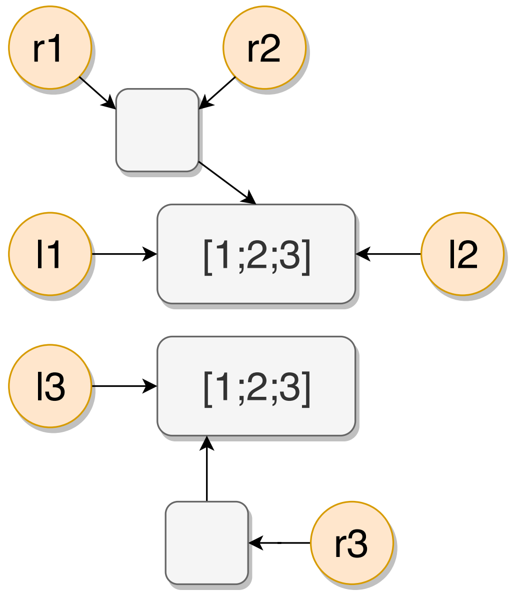
```
let l1 = [1;2;3];;
let l2 = l1;;
let l3 = [1;2;3];;
let r1 = ref l1;;
let r2 = r1;;
let r3 = ref l3;;
```

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`

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.

In [ ]:

```
l1 = l2
```

Value Restriction

Consider the following program:

In [ ]:

```
let r = [] in
let r1 : int list = r in
let r2 : string list = r in
(r1,r2)
```

`r` has type `'a list`. But otherwise, nothing surprising here.

Value Restriction

Consider a modified program:

In [ ]:

```
let r = ref [] in
let r1 : int list ref = r in
let r2 : string list ref = r in
(r1,r2)
```

Value Restriction

Let's look at the type of `ref []`

In [ ]:

```
let r = ref []
```

- The `'_weak1' says that r` is only *weakly polymorphic*.
  - `r` can be used with only one type.
  - This is known as **value restriction**.
- But why does value restriction exist?

Why does value restriction exist?

If value restriction does not exist, the following program would be well-typed.

```
let r = ref [] in
let r1 : int list ref = r in
let r2 : string list ref = r in
r1 := [1];
print_endline (List.hd !r2)
```

- We are storing an int list in `r1` and reading it out as a string list through `r2`.
- In OCaml, value restriction is implemented as a syntactic check of RHS + some typing checks.
  - Details are beyond the scope of this course.

Partial Application and Value restriction

Since value restriction is implemented as a syntactic check, it can sometimes be restrictive.

For example, here is a function that swaps the elenents of a pair in a list of pairs.

In [ ]:

```
let swap_list = List.map (fun (a,b) -> (b,a))
```

The type inferred is a weakly polymorphic type.

In [ ]:

```
(swap_list [(1,"hello")],
 swap_list [(1,1)])
```

Partial Application and Value restriction

In many cases, the unnecessary value restriction can be fixed by eta expansion.

In [ ]:

```
let swap_list l = List.map (fun (a,b) -> (b,a)) l
```

In [ ]:

```
(swap_list [(1,"hello")],
 swap_list [(1,1)])
```

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 [ ]:

```
(* 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
```

Double-linked list

In [ ]:

```
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
```

Doubly-linked list

In [ ]:

```
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
```

Doubly-linked list

In [ ]:

```
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
```

Doubly-linked list

In [ ]:

```
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
```

Doubly-linked list

In [ ]:

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

Doubly-linked list

In [ ]:

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

Doubly-linked list

In [ ]:

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

Arrays

Collection type with efficient random access.

In [ ]:

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

In [ ]:

```
a.(2)
```

Arrays

In [ ]:

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

In [ ]:

```
a.(4)
```

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.