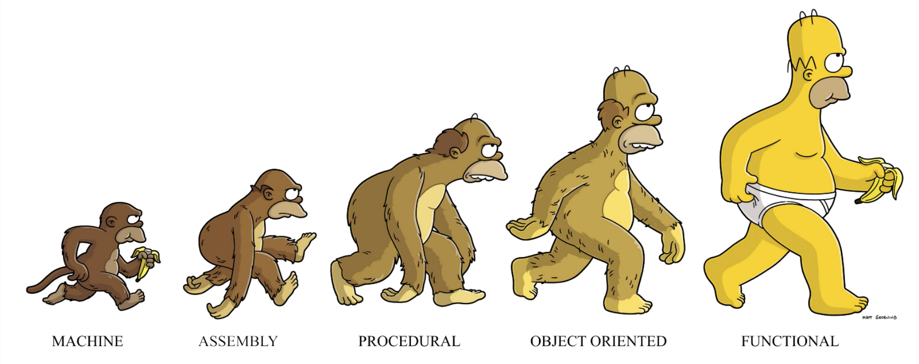
COMP302: Programming Languages and Paradigms

Week 5: References

Prof. Brigitte Pientka <bpientka@cs.mcgill.ca>

School of Computer Science, McGill University



Computation and Effects

So far:

Expressions in OCaml:

- An expression has a type.
- An expression evaluates to a value (or diverges).

Today:

Expressions in OCaml may also have an *effect*.

Recall: Variable Bindings and Overshadowing

```
# let (k : int) = 4;;
```

```
# let (k : int) = 3 in k * k;;
```

int 9

k	4
~~k~~	~~3~~

temp

remove

What is the value of `k`?

```
# k;;
```

4

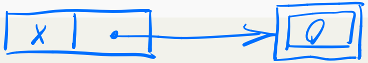
Easy as Pi

```
let pi   = 3.14 ;;
let area = (fun r -> pi *. r *. r) ;;
let a2   = area (2.0) ;;   12.56
let pi   = 6.0 ;;
let a3 = area 2.0 ;;   12.56
```

How to program with state? – Allocate and Compare

- How to allocate state?

```
let x = ref 0
```



Allocates a reference cell with the name `x` in memory and initializes it with `0`.

- How to compare two reference cells?

Compare their address: `r == s`

Succeeds if both `r` and `s` refer to the same location in memory

Compare their content: `r = s`

Succeeds if both reference cells store the same value.
(Equivalent to `!r = !s`.)

How to program with state? – Read

How to read value stored in a reference cell?

int

```
!x
```

int ref

Read value that is stored in the reference cell with name `r`.

int

```
let {contents = x} = r
```

int ref

Pattern match on value that is stored in the reference cell with name `r`.

How to program with state? – Write

How to update the value stored in a reference cell?

int ref int

```
x := 3
(* or *)
x.contents <- 3
```

unit

element of type unit : ()
called "unit"

Writes the value in the reference cell with the name `r`

The previously stored value is overwritten.

x := 2+3

(5 is stored)

Imperative vs Declarative Programming in OCaml

```ocaml
let imperative_fact n =
  let result = ref 1 in
  for i = 1 to n do
    result := !result * i
  done;
  !result
```

computes an uninteresting value BUT an interesting effect

sequencing <exp> ; <exp>

- More complicated than the purely functional version; harder to reasont about
- Considered bad style in a functional language

```ocaml
let fact n =
let rec f n acc =
  if n = 0 then acc
  else f (n-1) (n*acc)
in
  f n 1
```

```ocaml
let rec fact n =
  if n = 0 then 1
  else  n * f (n-1)
```

Good Uses of State: Global Counter

```
let counter = ref 0

(* newName () ===> a,  where a is a new name *)
(* Names are described by strings and an nat. *)
let newName () =
  (counter := !counter + 1;
   "a" ^ string_of_int (!counter))
```

global counter

concat

Good Uses of State: Objects with shared state

Combine higher-order functions and shared state.

```
type counter_object =
  {tick : unit -> int ;
   reset: unit -> unit}

let newCounter () =
  let counter = ref 0 in
    {tick = (fun () -> counter := !counter + 1; !counter) ;
     reset = fun () -> counter := 0}
```

let C1 = newCounter ()   C1 will be bound to a record with 2 fields,
tick and reset, which share the variable counter

different

let C2 = newCounter ()   C1 will be bound to a record with 2 fields,
tick and reset, which share another variable counter

What are immutable data structures?

- Examples: Lists, Trees, etc.
- It is impossible to change the structure of the list without building a modified copy of that structure
- Immutable data structures are *persistent*, i.e. operations performed on them donot destroy the original structure.
- Implementations using immutable data structures are easier to understand and reason about.

What are mutable data structures?

- Examples: Linked lists, arrays, etc.
- Update in place and modify an existing structure without rebuilding it
- Mutable data structures are *ephemeral*, i.e. operations performed on them do modify directly the original structure.

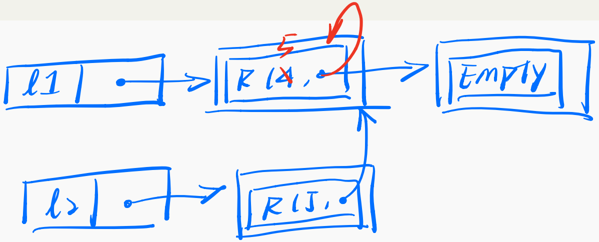
Circular Lists

```
type 'a rlist = Empty | RCons of 'a * 'a rlist ref
```

Let's define some lists ...

```
# let l1 = ref (RCons(4, ref Empty));;
# let l2 = ref (RCons(5, l1));;
# l1 := !l2;;
```

circular list



Programming with Linked Lists

Imperative Append on Linked Lists

(α list) ref (α list) ref

```
let rec rapp (r1, r2) = match !r1 with
  | Empty -> r1 := !r2
  | RCons (x,xs) -> rapp (xs, r2)
```

rapp : (α list) ref * (α list) ref -> unit

In contrast to our former declarative list append:

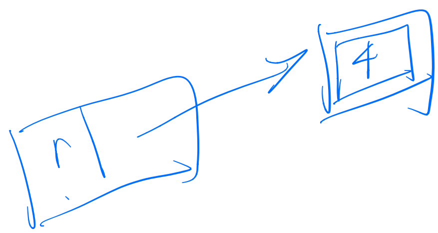
```
let rec app l1 l2 = match l1 with
  | [] -> l2
  | x::xs -> x::(app xs l2)
```

Take-Away

- Programming with mutable state using references
  (allocate, read, and write to a location in memory)
- Understanding the difference between variable bindings and mutable state
- Declarative programming is less error prone
- Mutable state is useful for certain data structures (for example linked lists), global variables, etc.
- Model Objects using records where each field is a function and shared state

$f(6)$

$y = x$



$y = 3 \times 4 = 12$

$6 + 3 + 6 + 4$

9 25

$24 + 3 + 24 + 4$

$48 + 7$