# References

COMP 320

15 REFERENCES

# Imperative Programming

Imperative programming can be seen as  a special case of functional programming

Functional programming is all about transforming data into new data. Imperative programming is all about updating and changing data.

Much of the material in this section is based on Chapter 5 of the text by Prof. Pientka available of MyCourses

In these notes we will see SML versions of some of the Ocaml code there

Unlike some pure functional languages (e.g. Haskell) SML provides a type `'a ref` representing mutable memory cells

# Reference Cells

The type is ‘a ref is equipped with a constructor

```
ref : 'a -> 'a ref
```

Evaluating **`ref v`** creates and returns a new memory cell containing the value v.

Pattern-matching a cell with the pattern **ref p** gets the contents.

Example:

```
val r = ref 0
val s = ref 0
```

# Reference Cells

Reference cells are like all other values

They can be bound to names, passed as arguments, returned as the result of a function, stored in other ref cells, etc

```
val a = r=s
```

The names r and s refer to the reference cell not to the contents

# Accessing Contents

The constructor ! Is used to access the contents of a cell (!r returns 0).

The assignment operator :=, generally written as an infix operator is used to change the contents of a reference cell.

For example `r := 5` replaces the contents of r with 5

Type checking:

```
x := y,  x is of type 'a ref, y is of type 'a
```

# Example

```
val r = ref 0
val s = ref 0
val _ = r := 3 (* discard the result *)
val x = !s + !r
val t = r  (* aliasing *)
val b = s=t
val c = r=t
val _ = t := 5
val y = !s + !r
val z = !t + !r;
```

# Sequencing

Programs now have a temporal aspect

The values of a reference cell can change over time

Up until now, a binding for a variable does not change when we evaluate it within its scope

We can use ; to sequentially compose expressions:

```
e1 ; e2
```

Is an abbreviation of

```
let
  val _ = e1
in
  e2
end
```

# Recall an old example

```
fun test () =
  let
    val pi : real = 3.14                                    (* 1 *)
    val area = (fn (r:real) => pi * r * r)                  (* 2 *)
    val a2 = area  2.0                                      (* 3 *)
    val pi : real = 6.0                                     (* 4 *)
    val a3 = area 2.0
  in
    print ("Area a2 = " ^ Real.toString a2 ^ "\n");
    print ("Area a3 = " ^ Real.toString a3 ^ "\n")
end;
```

# Using References

```
fun test_ref () =
  let
    val pi : real ref = ref 3.14                          (* 1 *)
    val area = (fn (r:real) => !pi * r * r)               (* 2 *)
    val a2 = area (2.0)                                   (* 3 *)
    val _  = pi := 6.0                                    (* 4 *)
    val a3 = area (2.0)
  in
    print ("Area a2 = " ^ Real.toString a2 ^ "\n");
    print ("Area a3 = " ^ Real.toString a3 ^ "\n")
end;
```

# Results

```
- test();
Area a2 = 12.56
Area a3 = 12.56
val it = () : unit

- test_ref ();
Area a2 = 12.56
Area a3 = 24.0
```

# Verification

In languages such as C all variables are implicitly bound to reference cells

They are implicitly dereferenced whenever they are used

The name of a variable always stands for the current contents

Reference cells make reasoning about programs much harder

They are used very sparingly in SML since we want to be able to prove things about programs

# Mutation and Verification

```
fun ++ r = (r := !r + 1; !r)
val r = ref 0
++ r
== 1

(++ r ; ++ r)
== (r := r + 1; !r) ; ++ r
== let val _ = (r := r + 1; !r) in ++ r
== let val _ = !r in ++ r
== ++ r
== 2
```

# Mutation and Verification

```
fun ++ r = (r := !r + 1; !r)
val r = ref 0
++ r
== 1

(++ r ; ++ r)
== (r := r + 1; !r) ; ++ r
== let val _ = (r := r + 1; !r) in ++ r
== let val _ = !r in ++ r
== ++ r
== 2
```

**If referential transparency held,**
**1 == 2**

# Some Remarks

It's important to note that, with mutation, the same program can have different results if it is evaluated multiple times

This is characteristic of effects (sometimes called side effects) such as mutation and input/output, but not something that a pure functional program will do

Note that the value of r never changes. It is bound to a container whose contents change but the values bound to variables do not change.

# Some Functions

```
fun !(ref v) = v
```

This gets the value of a cell, so we can write !r instead of pattern-matching

```
fun update (f : 'a -> 'a) (r : 'a ref) : unit =
           let val (ref cur) = r
           in
              r := f cur
           end
```

This combines getting a value and modifying it

`update (fn x => x + 1) r` is like r++ (except it doesn't return either the before or the after value).

# Mutable Data Structures

We have seen lists, trees, etc

It is not difficult to implement other data structures such as stacks, etc

However immutable structures are not destroyed when we perform operations such as inserting a new value.

We just rebuild the entire structure as we process it

They are called **persistent**

However sometimes we want to be able to update a data structure (e.g. a dictionary) without having to make a copy of it

Using references we can create **ephemeral** data structures

# Ephemeral Lists

```
datatype 'a rlist =
                  Empty
               |  Rcons of 'a * (('a rlist) ref)
```

# Circular Lists

```
val L1 = ref (RCons(4, ref Empty))
val L2 = ref (RCons(5, l1))

(* this will create a circular list *)
val _ =  L1 := !L2
```

We can visualize this by drawing box diagrams

# Traversing the list

```
(* Given a reference list L and a bound n,

observe(l,n) prints the first n values. Without this
bound, observe loops indefinitely for circular lists *)

(* observe: 'a rlist * int -> unit *)

fun observe L n =

  case L of

      Empty => print “0\n"

    | RCons(x, L) =>

      if n = 0 then print "STOP\n"

      else (print (Int.toString (x) ^ " ");

            observe !L (n-1))
```

# Example

- `observe (!L1, 4);`

  `5 5 5 STOP`

# Append

We want to append two reference lists.

The first list is a pointer to a reference list.

Intuitively, we traverse this list and re-assign the reference to the second list

Nothing is returned from this function since it just modifies the first list (destructively)

# Destructive Append

```
(* rapp : 'a refList * 'a refList -> unit *)
fun rapp r1 r2 =
  case r1 of
    ref Empty =>  r1 := !r2
  | ref RCons (h,t) => rapp t r2
```

As a side effect, reference list r1 is updated

We can only observe this change by going through r1 after appending

# Example

```
- val rlist1 = ref (RCons (1, ref Empty));
val rlist1 = ref (RCons (1,ref Empty)) : int rlist ref
- val rlist2 = ref (RCons (4,ref (RCons (6, ref Empty))));
val rlist2 = ref (RCons (4,ref (RCons (6,ref Empty)))) :int rlist ref
- rapp(rlist1, rlist2);
val it = () : unit
- (!rlist1);
val it = RCons (1,ref (RCons (4,ref(RCons(6,ref Empty))))):int rlist
```

# Reversing a List

Methodology:

- Create a temporary reference to an empty list
- Recursively traverse the list
- Pop the elements from the list and push them onto the temporary reference list
- When done, reassign the original list to the new one

# Destructive Reverse

```
(* This is a destructive reverse function.  *)
(* reverse : 'a refList -> ' a refList *)
fun rev L =
  let
    val r = ref Empty
    fun rev' L =
      case L of
        ref Empty => L := !r
      | RCons (h,t) => (r := RCons(h, ref (!r));rev' t)
  in
    rev' L ; L := !r
  end
```

# Implementing a Counter

We can create a counter which is

- incremented by the function tick
- reset by the function reset

```
local
  val counter = ref 0
in
  fun tick () = (counter := !counter + 1; !counter)
  fun reset() = (counter := 0)
end
```

The two functions tick:unit -> int and reset:unit -> unit share a private variable *counter* that is bound to a reference cell containing the current value of a shared counter.

The tick operation increments the counter and returns its new value, and the reset operation resets its value to zero.

# Records

We make use of records in the following example

Record syntax: `{f1 = e1, …, fn = en}`

- Records have fields with names, f1
- An expression is associated with each fieldname

Record type: `{f1 : t1, …, fn : tn}` where tn is the type of en

Record values:

- Evaluate ei to vi
- Value of record is {f1 = v1, …, fn = vn}

There is no ordering of fields

- REPL orders fieldnames alphabetically by convention

Components can be accessed by `#fieldname e`

# A Counter Generator

We can define a counter generator to produce several instances of a counter each with its own tick and reset functions

```
(* newCounter: unit -> {tick : unit -> int, reset: (unit -> unit)} *)

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}
```

# A Counter Generator

We can define a counter generator to produce several instances of a counter each with its own tick and reset functions

```
(* newCounter: unit -> {tick : unit -> int, reset: (unit -> unit)} *)
fun newCounter () =
  let
    val counter = ref 0
    fun tick () = (counter := !counter + 1; !counter)
    fun reset() = (counter := 0)
  in
    {tick = tick, reset = reset}
  end
```

# Object Oriented Model

We've packaged the two operations into a record containing two functions that share private state.

There is an obvious analogy with class-based object-oriented programming.

The function newCounter may be thought of as a constructor for a class of counter objects.

Each object has a private instance variable counter that is shared between the methods tick and reset

```
val c1 = newCounter()
val c2 = newCounter()
```

Notice, that c1 and c2 are distinct counters!

# Example

```
#tick c1 ();
(* 1 *)
#tick c1 ();
(* 2 *)
#tick c2 ();
(* 1 *)
#reset c1 ();
#tick c1 ();
(* 1 *)
#tick c2 ();
(* 2 *)
```

# Bank Account

```
datatype transactions =
         Withdraw of int
       | Deposit of int
       | Check_balance
```

Define a function

```
make_account : int  ->
                          (transaction -> int)
```

# Bank Account

```
fun make_account (opening_balance: int) =
  let val balance = ref opening_balance in
    fn (trans: transactions) =>
        case trans of
          Withdraw(a)
              => ((balance := !balance-a); !balance)
        | Deposit(a)
              => ((balance := !balance+a); !balance)
        | Check_balance
              => (!balance)
    end
```

# Examples

```
- val Alice = make_account 500);
val Alice = fn : transactions -> int
- val Bob = make_account 400;
val bob = fn :  transactions -> int
- Bob (Deposit(75));
val it = 475 : int
- Alice(Withdraw(75));
val it = 425 : int
- Alice(Check_balance);
val it = 425 : int
```