This Lecture

• ML-style imperative programming
  – Mutable memory cells

• Persistent versus ephemeral data structures
Functional programming is all about evaluation

- `fun g x = (* some expression *)`
  `val g = fn : int -> int`

- `g 3;`
  `val it = 6 : int`

- `g 3;`
  `val it = 6 : int`

Repeated evaluation of the same expression will yield the same result
functional = pure

Functional programming is all about **evaluation**

- `g 3 + g 4 + g 5;`
  `val it = 24 : int`

- `Seq.mapreduce g 0 (op +) (upto 3 6);`
  `val it = 24 : int`

**Sequential and parallel evaluation of independent sub-expressions will produce the same value**
imperative = impure

Imperative programming is all about **execution**

Executing code produces a **value** and causes an **effect**

- The effect of one evaluation may change the value produced by another evaluation and may change its effect

*Repeated evaluation of the same expression may not yield the same result*

*Sequential and parallel evaluation of independent sub-expressions may not produce the same value*
ML isn’t really pure

ML supports functional programming but also imperative features such as references, arrays, and commands for I/O.
ML

- ML = functional + imperative
  - Expressions have types
  - Evaluation produces a value, and causes an effect

- Expressions may have a value of type `unit` when their purpose is to cause an effect

```ml
- print ("Hello world\n");
Hello world
val it = () : unit

- fun f x = (print "Calling f\n"; x + 1);
val f = fn : int -> int
- f 3;
Calling f
val it = 4 : int
```
Creating, reading, updating reference cells

MUTABLE MEMORY CELLS
Mutable cells

• Type constructor 'a ref

• A value of type int ref is a cell for storing an integer
  – To create a fresh cell
    use ref:'a -> 'a ref
  – To update a cell’s contents,
    use infix := of type 'a ref -> unit
  – To read a cell’s contents,
    use ! : 'a ref -> 'a
Expressions for creating and using references

(initialize) $\text{ref } e$

(read) $!e$

(write) $e_1 := e_2$
Typing

\textbf{ref} \ e : t \ \textbf{ref}

\textbf{!e} : t

\ e\_1 := e\_2 : \text{unit}

\textbf{if} \ e : t

\textbf{if} \ e : t \ \textbf{ref}

\textbf{if} \ e\_1 : t \ \textbf{ref}

\textbf{and} \ e\_2 : t
Sequential composition

• When \( e_1 : t_1 \) and \( e_2 : t_2 \), \( e_1 ; e_2 \) has type \( t_2 \)

• To evaluate \( e_1 ; e_2 \)
  
  evaluate \( e_1 \), then
  
  evaluate \( e_2 \) and
  
  return the value of \( e_2 \)

• The effect of \( e_1 ; e_2 \) is the effect of \( e_1 \) followed by the effect of \( e_2 \)
Example: sequential composition

\texttt{val s = ref 20}

\texttt{(s := !s + 10; !s )}

\hspace{1cm} evaluates to 30

\texttt{(s := !s + 10; s)}

\hspace{1cm} evaluates to \texttt{ref 40}
Aliasing

• **ref** types are *equality* types
• $r = s$ evaluates to **true** if $r$ and $s$ denote the same cell, **false** otherwise
• When $r = s$ evaluates to **true**, we say $r$ is an alias for $s$
fun transfer(x : int ref, y : int ref) : unit =
  if !x = 0 then () else
  (x := !x - 1; y := !y + 1; transfer(x,y))

- val x = ref 21;
val x = ref 21 : int ref
- val y = ref 21;
val y = ref 21 : int ref
- transfer(x,y);
val it = () : unit
- (!x, !y)
(0, 42)

x and y are two different cells
Alias issues

fun transfer(x : int ref, y : int ref) : unit =
  if !x = 0 then () else
  (x := !x - 1; y := !y + 1; transfer(x,y))

- val x = ref 21;
val x = ref 21 : int ref
- val y = x;
val y = ref 21 : int ref
- transfer(x,y);

x and y are the same cell.

loops forever
Referential Transparency for functional programs

- Expressions of type `int` are *equivalent* if
  - they *evaluate to* the same *integer value*, or
  - *fail to terminate*, or
  - *raise the same exception*

- A program is *equivalent* to the program obtained by replacing a sub-expression with an *equivalent* expression
Referential Transparency
for imperative programs

- Expressions of type `int` are equivalent if, from every state (environment + store),
  - they evaluate to the same integer value, or
  - fail to terminate, or
  - raise the same exception,
  - and they have the same effect

- A program is equivalent to the program obtained by replacing a sub-expression with an equivalent expression
Example 1

(* update : ('a -> 'a) -> 'a ref -> unit
  REQUIRES: true
  ENSURES: update(f,r) changes the contents of r to f(!r).*
*)

fun update (f: 'a -> 'a) (r : 'a ref) : unit =
  let
    val ref (cur) = r
  in
    r := f (cur)
  end
fun deposit (n: int) (a: int ref) : unit = a := !a + n

fun withdraw (n: int) (a: int ref) : unit = a := !a - n
Do both sequentially

```plaintext
val r: account = ref 100
(deposit 100 r; withdraw 50 r)
```

Does the order matter?
Do both in parallel

fun deposit (n: int) (a: int ref) : unit = a := !a + n
fun withdraw (n: int) (a: int ref) : unit = a := !a - n

val r: account = ref 100

Seq.tabulate (fn 0 => deposit 100 r | 1 => withdraw 50 r) 2

The order matters. You could end up with 50, 150 or 200.
In the presence of mutation

<table>
<thead>
<tr>
<th></th>
<th>Persistent data</th>
<th>Ephemeral data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>Effects must be benign</td>
<td>Need to worry about concurrency, scheduling</td>
</tr>
<tr>
<td>Sequential</td>
<td>Effects must be benign</td>
<td>OK but can complicate reasoning</td>
</tr>
</tbody>
</table>

traditional imperative programming
BENIGN SIDE EFFECTS
Naive rev

\[
\text{fun } \text{rev} \ [\ ] = [\ ] \\
| \ \text{rev} \ (x::xs) = (\text{rev} \ xs) \ @ \ [x]
\]

Quadratic work: recall that append does work linear in the size of the first list.
fun fastrev (L : 'a list) : 'a list = 
  let
    val R = ref []
    fun loop [] = !R
    | loop (x::xs) = (R := x::(!R); loop xs)
  in
    loop L
  end
Graph Reachability

```plaintext
type graph = int -> int list

val G : graph = fn 1 => [2,3]
  | 2 => [1,3]
  | 3 => [4]
  | _ => []

Define a function that satisfies the following specification.

(* reachable : graph -> int * int -> bool
  REQUIRES:  true
  ENSURES:   reachable g (x,y) returns true
             if y is reachable from x in g,
             and returns false otherwise.
  *)
```
Graph Reachability

type graph = int -> int list

val G : graph = fn 1 => [2,3]
  | 2 => [1,3]
  | 3 => [4]
  | _ => []

fun mem (n:int) = List.exists (fn x => n = x)
Graph Reachability

type graph = int -> int list

val G : graph = fn 1 => [2,3]
   | 2 => [1,3]
   | 3 => [4]
   | _ => []

fun mem (n:int) = List.exists (fn x => n = x)

fun reachable1 (g:graph) (x:int, y:int) : bool =
  let
    fun dfs n = (n=y) orelse (List.exists dfs (g n))
  in
    dfs x
  end

  did we reach y?

neighbors of n
Graph Reachability

type  graph = int -> int list

val  G : graph = fn 1 => [2,3]  
    | 2 => [1,3]  
    | 3 => [4]  
    | _ => [ ]

fun mem (n:int) = List.exists (fn x => n = x)

fun reachable1 (g:graph) (x:int, y:int) : bool =  
let  
    fun dfs n = (n=y) orelse (List.exists dfs (g n))  
  in  
    dfs x  
  end

val  true = reachable1 G (3,4)

val  true = reachable1 G (1,4) ← loops forever
(* reachable : graph -> int * int -> bool
  REQUIRES:  true
  ENSURES:   reachable g (x,y) returns true
            if y is reachable from x in g,
            and returns false otherwise.  
*)

fun reachable (g:graph) (x:int, y:int) : bool =
  let
    val visited = ref []
    fun dfs n = (n=y) orelse
      (not (mem n (!visited))
       andalso
       (visited := n::(!visited);
        List.exists dfs (g n)))
  in
    dfs x
  end
In the presence of mutation

<table>
<thead>
<tr>
<th></th>
<th>Persistent data</th>
<th>Ephemeral data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parallel</strong></td>
<td>Effects must be benign</td>
<td>Need to worry about concurrency, scheduling</td>
</tr>
<tr>
<td><strong>Sequential</strong></td>
<td>Effects must be benign</td>
<td>OK but can complicate reasoning</td>
</tr>
</tbody>
</table>

traditional imperative programming
A puzzle?

• For all expressions $e : \text{int}$, if $e$ terminates, then according to our definition of equality, $e$ is equal to $e$ even if $e$ causes an effect.

• But in ML, $e = e$ may evaluate to $\text{false}$.

For example, let $e$ be the expression

$$(x := !x + 1; !x)$$
Not a puzzle

• ML’s equality operator = uses left-to-right *sequential* evaluation

• If e has a side effect, the second e gets evaluated in a different state, so may return a different value

• In such a case, \((e = e) \implies false\) in ML

• But it is still the case that \(e = e\) holds, mathematically

And ML = is not the same as our notion of equivalence
Functions with private state

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

The type of \texttt{tick} is \texttt{unit -> int}

The function’s applicative behavior depends on the contents of the \texttt{int} \texttt{ref} cell holding an integer

When we call \texttt{tick()} the cell gets updated
Salvador Dali’s masterpiece, *The Persistence of Memory*, is one of the most recognizable works of art in the world. The critic Dawn Ades described it as “a Surrealist meditation on the collapse of our notions of a fixed cosmic order” induced by Einstein’s penetrating analysis of the concepts of time and space in the physical world. Just as Dali’s Persistence of Memory demarcated the transition from age-old conceptions of time and space in physics, so does the computational concept of persistence of memory mark the transition from sequential time and mutable space to parallel time and immutable space.

Bob Harper