ML isn’t pure

ML supports *functional programming*, but also

- imperative features
- evaluation and effects
- mutable data

“impure” features don’t mesh so nicely with parallel evaluation so we’ll work sequentially for now…
ML

- ML = functional + imperative
- expressions have types
- evaluation produces a value, and causes an effect
- Expressions may evaluate to ( ) : unit when their purpose is to cause an effect
  
  (count := !count + 1) : unit
  print “calling F” : unit
using effects

- When *debugging* code it can be helpful to put *print* statements at *strategic* places.
- But don’t forget to *delete* them before releasing the code to users!
example

- In alpha/beta minimax, insert print statements in F and G, to check for pruning.
- The chatty versions of F and G have the same types, and produce the same results, as the old F and G, but also have an effect.

\[
F, G : \text{Game.state} \rightarrow \text{int} \\
F_1, G_1 : \text{Game.state} \rightarrow \text{int}
\]

- no effect
- prints call information
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 * ’a -> unit

  • to read a cell’s contents, use ! : ’a ref -> ’a
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 $e_2$
ref types

- 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 that $r$ is an alias for $s$
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 y = ref 21;
transfer(x, y);
(!x, !y)

= (0, 42)

val x = ref 21;
val y = x;
transfer(x, y);
(!x, !y)

loops forever
“state” = env + store

• *Declarations* introduce an **environment** binding names to values, which may be cells

  \[[x:r_{99}, \ y:42, \ L:[2,3,4]]\]

• The **store** specifies the *contents* of active cells

  \[[r_{99}:23, \ r_{104} : [5,6,7]]\]

• *Assignment* changes the store (not the environment)
**illustration**

\[
\text{val } x = \text{ref } 0; \\
x := !x + 1
\]

*binds* \(x\) *to new cell containing* 0

*updates contents to* 1

\(x\) *still bound to the same cell, now containing* 1
Frege’s principle for functional programs

• The type of an expression depends on the types of its free variables

• The value of an expression depends on the values of its free variables

\([x:21, y:21]\) \(x+y \Rightarrow^* 42\)

If \(x\) and \(y\) are bound to 21, the value of \(x+y\) is 42
Frege’s principle for imperative programs

• The type of an expression depends on the types of its free variables

• The value of an expression and its effect depend on the values of its free variables, and on the store

\[[x:loc, y:loc] \{!loc = 21\} !x + !y =\Rightarrow* 42 \{!loc = 21\}\]

If x and y are bound to the same cell, containing 21, the value of \(!x + !y\) is 42 and there’s no effect
referential transparency
for functional programs

• Expressions of type \texttt{int} are equal if they evaluate to the same integer value, or fail to terminate, or raise the same exception

• A program is equal to the program obtained by replacing a sub-expression with an equal expression
referential transparency for imperative programs

- Expressions of type `int` are **equal** if, from every environment and 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 **equal** to the program obtained by replacing a sub-expression with an equal expression.
example

fun inc(a : int ref, n : int) : unit =
  if n=0 then () else (a := !a + 1; inc(a, n-1))

inc(x, 42) = x := !x + 42

implies

P[inc(x, 42)] = P[x := !x + 42]

implies

same value, same effect

implies

same value, same effect, in all contexts
patterns

• Can match cells and contents (in current state)
• Variable and wildcard patterns can match cells
• \texttt{ref p} matches a cell whose contents match \( p \)
• Use pattern \texttt{x as ref p} to match \( x \) with a cell and \( p \) with its current contents

- \texttt{fun update (f:'a ->'a)(x as ref v) = (x := f(v));}
  \texttt{val update = fn : ('a -> 'a) -> 'a ref -> unit}
fun update (f : 'a -> 'a) (x : 'a ref) : unit =
let
  val (ref v) = x
in
  x := f(v)
end
fun update (f : 'a -> 'a) (x as ref v) : unit =
  x := f(v)
examples

fun update (f : 'a -> 'a) (x : 'a ref) : unit =
  x := f(!x)
question

• What assumptions about the values of $f$, $g$, and $a$ are sufficient to ensure that

$$(\text{update } f \ a \ ; \ \text{update } g \ a) = \text{update } (g \circ f) \ a$$
example

bank account

fun update (f:'a ->'a)(x as ref v) = (x := f(v))

type account = int ref

fun deposit(n : int) : account -> unit
  = update (fn v => v+n)

fun withdraw(n : int) : account -> unit
  = update (fn v => v-n)

not very realistic!
more realistic

bank account

type account = int ref

exception Penalty

fun create( ) : account = ref 0

(* REQUIRES n ≥ 0 *)

fun deposit(n : int)(r : account) : unit
  = update (fn v => v+n) r

(* REQUIRES n ≥ 0 *)

fun withdraw(n : int)(r : account) : unit
  = if !r < n then raise Penalty else update (fn v => v-n) r
evaluation

• environments  \([x:42, y:\text{loc}_{33}]\)

• stores  \(\{\text{!loc}_{33}=0, \text{!loc}_{99}=2\}\)

• evaluating an expression from a state

\([\text{env}] \{\text{pre-store}\} \ e \Rightarrow^* v \ \{\text{post-store}\}\)

\([x:42, y:\text{loc}] \ \{\text{!loc}=0\} \ y := !y + x \Rightarrow^* ( ) \ \{\text{!loc}=42\}\)
val r: account = ref 100;
withdraw 200 r;
deposit 100 r;

does the order matter?
example

fun update f x = (x := f(!x))

[r:loc] {!loc = 100} update (fn v => v+100) r
  => update (fn v => v+100) loc {!loc = 100}
  => loc := (fn v => v+100)(!loc) {!loc = 100}
  => loc := (fn v => v+100)(100) {!loc = 100}
  => loc := 100+100 {!loc = 100}
  => loc := 200 {!loc = 100}
  => ( ) {!loc = 200}

the derivation shows intermediate states

[r:loc] {!loc = 100} update (fn v => v+100) r =>* ( ) {!loc = 200}
specifications

\[
\text{fun transfer}(x, y) = \\
\quad \text{if } !x = 0 \text{ then } () \text{ else } (x := !x - 1; y := !y + 1; \text{transfer}(x, y))
\]

- If \( \text{loc}_1 \neq \text{loc}_2 \), for all \( m \geq 0 \) and all \( n : \text{int} \) we have

\[
[x: \text{loc}_1, y: \text{loc}_2] \{!\text{loc}_1 = m, !\text{loc}_2 = n\} \\
\text{transfer}(x, y) \Rightarrow^\ast () \{!\text{loc}_1 = 0, !\text{loc}_2 = m + n\}
\]

- For all \( m \neq 0 \),

\[
[x: \text{loc}, y: \text{loc}] \{!\text{loc} = m\} \text{transfer}(x, y) \Rightarrow^\omega
\]

loops forever

Need to specify value \textit{and} effect
fast reverse

fastrev : ’a list -> ’a 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
properties

fastrev : 'a list -> 'a list
loop : 'a list -> 'a list

• Runtime for fastrev L is $O(\text{length } L)$

• For all types $t$ and all values $L : t$ list,
  fastrev $L = \text{rev } L$

• For all values $A, B : t$ list and $r : (t$ list$)$ ref
  $[R:r]\{!r = B\}$ loop $A \Rightarrow* !R \{!r = (\text{rev } A)@B\}$
proof

fun loop [ ] = !R
  | loop (x::xs) = (R := x :: (!R); loop xs)

For all values A, B : t list,
[R:r]{!r = B} loop A  =>* !R {!r = (rev A)@B}

Proof:
By induction on (length of) A

Base case: For A = [ ]

[R:r]{!r = B} loop [ ] =>* !R {!r = B} by def of loop
  =>* !R {!r = (rev [ ]))@B}
since rev [ ] = [ ] and [ ] @ B = B
proof

Inductive case: $A = x::xs$

Induction hypothesis: For all values $B'$, 
$[R:r] \{!r = B' \} \text{ loop } xs \implies \ast \! R \{!r = (\text{rev } xs)@B' \}$

$[R:r] \{!r = B \} \text{ loop}(x::xs)$

$\implies \ast (R := x :: (!R); \text{ loop } xs) \{!r = B \} \text{ by def}$

$\implies \ast (R := x :: B; \text{ loop } xs) \{!r = B \}$

$\implies \ast \text{ loop } xs \{!r = x::B \}$

$\implies \ast \! R \{!r = (\text{rev } xs) @ (x::B) \} \text{ by IH}$

$\implies \ast \! R \{!r = (\text{rev } (x::xs))@B \}$

since $(\text{rev } xs) @ (x::B) = (\text{rev}(x::xs)) @ B$
fun fastrev (L : 'a list) : 'a list =
  let
    val R = ref [ ]
    fun loop [ ] = !R
      l loop (x::xs) = (R := x :: (!R); loop xs)
  in
    loop L
  end

fun rev [ ] = [ ]
  l rev (x::L) = (rev L) @ [x]

- We proved that fastrev = rev
- Even though fastrev uses imperative features, its only use of mutable state is local
- Calling fastrev [1,2,3] has no visible effect and returns [3,2,1], same as rev [1,2,3]
• Imperative programs have value and effect
• Can reason about both using an appropriately generalized notion of equality
• Specifications and proofs must account for both and involve environment and store
• For purely functional code, effect is trivial and store is irrelevant, so analysis is easier
**functional = pure**

it’s all about evaluation

Repeated evaluation of the same expression will yield the same result

Sequential and parallel evaluation of independent sub-expressions will produce the same value

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

- g 3;
  val it = 6 : int

(* WHAT HAPPENS HERE? *)
- (g 3, g 3);
  val it = (6, 6) : int * int

- g 3 + g 4 + g 5 + g 6;
  val it = 870 : int

(* WHAT HAPPENS HERE? *)
- Seq.mapreduce g 0 (op +) (upto 3 6);
  val it = 870 : int
```
imperative = impure
it’s all about execution

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

the effect of one evaluation may change
the value produced by another evaluation and may affect its effect
ML isn’t really pure

- val flag = ref 0;
val flag = ref 0 : int ref

- fun g n = (flag := 1 - !flag; n * !flag);
val g = fn : int -> int

- g 3;
val it = 3 : int

- (g 3, g 3);

(* WHAT HAPPENS? *)
val it = (0, 3) : int * int

repeated evaluation, different result
ML isn’t really pure

- val flag = ref 0;
  val flag = ref 0 : int ref

- fun g n = (flag := 1 - !flag; n * !flag);
  val g = fn : int -> int

- g 3 + g 4 + g 5 + g 6;
  val it = 10 : int

- Seq.mapreduce g 0 (op +) (upto 3 6);
  (* WHAT HAPPENS? *)
  val it = 7 : int
  or val it = 8 : int
  or val it = 9 : int
  or val it = 10 : int
  or val it = 11 : int

*outcome could vary, depending on evaluation order*