Although ML is called a *functional programming language*, for pragmatic reasons the language also includes some *imperative* features. These features co-exist with the purely functional subset of ML, and the entire language is still based firmly on principles: well-typed programs don’t go wrong, etc. However, the inclusion of imperative constructs means that we need (again) to revise our notion of what “equality” means, and we need to be careful when substituting “equals for equals”.

We introduce the main ideas, and develop some examples to illustrate the new concepts and contrast with the purely functional world. Afterwards we reflect on whether or not, and how, to use imperative features: what advantages and what potential disadvantages can there be? What happens when we combine imperative features with parallel evaluation?

Topics:

- imperative vs. functional programming
- mutable cells
- evaluation and side effects
- reasoning about effects

Today we will limit attention to sequential programs. It’s easier to reason about imperative code in a sequential setting. Next time we will consider what happens when we use imperative features in a parallel setting.
1 Background

Functional programming is about evaluating expressions to produce values, by transforming data into new data. In contrast, imperative programming is about updating data and changing state. Using the imperative features of ML, we can write expressions that cause a side-effect (e.g. updating some stored value, or printing to a display) in addition to producing a value or raising an exception or failing to terminate.

Imperative features can sometimes be harnessed to improve efficiency (e.g. by avoiding repeated re-evaluation of some expression), but should be used carefully because it is much harder to reason about correctness of imperative code, since we need to take account of side-effects.

As an obvious example, in the purely functional subset of ML, whenever \(e\) is a well-typed expression of type `int`, \(e = e\) is a well-typed expression of type `bool`, and if \(e\) terminates then the value of \(e = e\) is `true`. Hence (assuming that \(e\) terminates) we can replace \(e = e\) by `true`, without affecting the behavior of the program. But if \(e\) includes imperative constructs, the value of \(e = e\) is no longer so easy to predict!

Mutable cells

The ML type system includes reference types of the form `t ref`, where `t` is a type. A value of type `t ref` is a “reference to a value of type `t`”, or a mutable cell capable of holding a value of type `t`. We can create a fresh cell, initialized to contain a given value, by using the function

\[
\text{ref} : 'a -> 'a ref
\]

The contents of a cell is dynamically updatable, by evaluating an update expression (or “assignment”). The assignment operator `:=` is written as an infix.

Do not make the mistake of writing `=` for assignment; ML is not C.

Here is some code to illustrate:

\[
\text{val r = ref 0;}
\]
\[
(* r : int ref *)
\]
\[
(* !r = 0 here *)
\]
This creates a new mutable cell, initializes its contents to 0, and binds \( r \) to the cell.

\[
r := (!r) + 1;
\]

\(* !r = 1 \text{ here } *)

The value of \( r \) (i.e. the cell bound to \( r \)) is unchanged – this is still the same mutable cell as before. However, as a side-effect, the contents of this cell has been changed to 1.

\[
\text{val } s = \text{ref 1};
\]

\(* s : \text{int ref, distinct from the value of } r. \text{ *)}

Now we declare a new cell and bind it to \( s \).

\[
s := (!s) + 1;
\]

\(* !r = 1, !s = 2 \text{ here } *)

Assigning to \( s \) has no effect on the contents of \( r \):

\[
s := !s + 1;
\]

\(* !r = 1, !s = 3 \text{ here } *)

We can introduce an alias for a cell, e.g.

\[
\text{val } x = r;
\]

\(* x : \text{int ref, } x = r, !x = !r = 1 \text{ here } *)

Ref cells can be tested for identity, and \( x = r \) evaluates to \text{true}, \( r = s \) evaluates to \text{false}.

\[
x := !x + 1;
\]

\(* !x = !r = 2 \text{ *)}

Assigning to \( x \) also affects the value of \( !r \), because \( x \) evaluates to the same cell as \( r \).
2 Reasoning about imperative code

The value of an expression in the imperative fragment of ML depends not only on the values of its free variables, but also on the contents of the cells denoted by its free variables. The value of \( !x + !y \) depends on the cells denoted by \( x \) and \( y \), and on the contents of these cells. We use the term *environment* for a value binding that associates the free variables of a program to values (which may include ref cells), and *store* for a mapping from ref cells to values (which, again, may include ref cells!). A *state* is an environment paired with a store.

For expressions in the imperative fragment of ML, evaluating an expression will produce a value (or fail to terminate, or raise an exception) and cause an *effect*, which we interpret as a state change.

Expressions of type \( \text{int} \) whose syntax uses imperative features are *equal* if in every state they evaluate to equal values *AND* have the same effect. (We can also extend equality to other types, along similar lines.)

For example, given the above sequence of declarations, note that

- \( r \) and \( s \) are *not* equal (at type \( \text{int ref} \))
- \( r \) and \( x \) are equal (at type \( \text{int ref} \))
- \( !r \) and \( !x \) are equal (at type \( \text{int} \))
- \( (!r) + (!r) \) and \( (!x) + (!x) \) are equal (at type \( \text{int} \))

With this extended notion of equality, we can still safely use referential transparency. The value and effect of an imperative code fragment are not affected if we replace a sub-expression by an *equal* sub-expression. For example,

\[
\text{fun inc (a:int ref, n:int) =}
\begin{align*}
& \text{if } n=0 \text{ then ( ) else (} a := !a + 1; \text{ inc(a, n-1)\}) \\
\end{align*}
\]

declares a function \( \text{inc:int ref * int -> unit} \), and in the context of the above sequence of declarations we get

\[
\text{inc(x, 42) = inc(r, 42)}
\]

Indeed, we also get \( \text{inc(x, 42) = (x := !x + 42).} \)
Although we still get referential transparency, reasoning about imperative code can be tricky, because we can only substitute safely based on a notion of equality that takes account of effects as well as values. In particular, equational laws familiar from the functional world may fail in the imperative setting, so you cannot use them without checking that they work! For example, in the functional world, whenever \( e \) is a purely functional expression of type \( \text{int} \), \( e = e \) holds (obviously — both sides evaluate to equal integer values, or both diverge, or both raise the same exception). Further, if \( e \) is a purely functional expression of type \( \text{int} \) which we know terminates without raising an exception, we also know that

\[
(e = e) = \text{true}
\]

holds, both mathematically (i.e. according to our notion of equality for expressions of type \( \text{bool} \)) and in ML, since ML evaluates \( e = e \) to \text{true} under the given assumptions. So it would be OK to replace \( e = e \) by \text{true} in any purely functional piece of code where \( e \) has the assumed properties. However, in the imperative setting this goes wrong. Suppose we declare:

\[
\text{val x = ref 0;}
\]

Let \( e \) be the expression \((x := !x +1; !x)\); this expression is well-typed and has type \( \text{int} \). ML evaluates \((x := !x +1; !x) = (x := !x +1; !x)\) from left to right, so the second expression gets evaluated in the state produced by the update done by the first. Consequently this expression of form \( e = e \) will evaluate to \text{false}.

One more example to convince you that imperative code is harder to deal with. Consider the function \( \text{tick} : \text{unit} \rightarrow \text{int} \) defined by

\[
\text{local}
\begin{align*}
\text{val x = ref 0} \\
\text{fun tick( ) = (x := !x +1; !x)}
\end{align*}
\text{end}
\]

Even though \( \text{tick} \) only affects a locally created ref cell, every time we call this function we get a different value. It's not possible to say any more that the meaning of a value of type \( \text{unit} \rightarrow \text{int} \) is representable mathematically as just a function from (values of type) \( \text{unit} \) to (values of type) \( \text{int} \), because the result of a function call depends on the private, invisible, piece of local
state inside the `tick` code. Even though the `binding` (of `x` to this cell) is no longer in scope when we leave the function body, the cell stays alive in the background, holding an integer that actually represents the number of times `tick` has been called!

### 3 Pattern matching and ref cells

We can use patterns to match cells, in a way that takes account of state.

- As usual, we can use variable patterns and wildcard to match against values of type `t ref` for some `t`.
- And we can use patterns of form `ref p`, where `p` is also a pattern, to match against cells whose contents (in the current state) match `p`.
- Sometimes we want to match a ref cell and bind one name to the cell and other names to components of its contents; we can use patterns of form `x as p` to do this, where `x` is the name we want to bind to the cell and `p` is to be used to match the contents.

Here are some examples:

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

Here we use the pattern `ref v` to bind `v` to the current contents of the cell denoted by `r`. We could have written instead:

```ocaml
fun update (f : 'a -> 'a) (r as ref v) : unit = (r := f(v))
```

Here of course we really do need a name for the cell and a name for its contents!

We could also have written

```ocaml
fun update (f : 'a -> 'a) (r : 'a ref) : unit = (r := f(!r))
```
4 Sequential composition

As you must have noticed(!), we can use semicolons to cause expressions or declarations to be evaluated in a specific sequential order. This can be especially important when there are non-trivial effects, as in our examples above.

In general, when \( e_1 \) and \( e_2 \) are well-typed expressions of types \( t_1 \) and \( t_2 \), \( e_1; e_2 \) is a well-typed expression of type \( t_2 \). (The type of \( e_1 \) is irrelevant to the ultimate type of \( e_1; e_2 \), but it must exist!). From any initial state, the value of \( e_1; e_2 \) is obtained by evaluating \( e_1 \) then returning the value of \( e_2 \) in the state produced by \( e_1 \). (The value of \( e_1 \) is thrown away.)

Semicolon is associative: for all well-typed expressions \( e_1, e_2 \) and \( e_3 \),

\[ e_1; (e_2; e_3) = (e_1; e_2); e_3. \]

And \( () \) is a “left unit” for sequential composition, in that for all well-typed expressions \( e \),

\[ ()e = e \]

5 An example: bank accounts

Consider the following functions:

fun deposit (n:int , a:int ref):unit = update (fn v => v+n) a;
fun withdraw (n:int, a:int ref):unit = update (fn v => v-n) a;

If we execute the following code fragment, which uses sequential composition, it’s easy to explain what happens:

val r = ref 100;
deposit(100, r);
withdraw(50, r);

The first line binds \( r \) to a new cell with initial contents 100. The second line changes the contents to 200. And the third line changes the contents to 150 and returns \( () \). The overall result is: binds \( r \) to a cell, returns \( () \), contents of cell set to 150. If we execute instead using a different sequential order, as in:
val r = ref 100;
withdraw(50, r);
deposit(100, r);

we get the same overall result: binds r to a cell, returns ( ), contents of cell set to 150. These two code fragments are equivalent.

We can make the above analysis more precise by using =>* (adjusted to take account of state), or by equational reasoning. We need to extend our notation for evaluation to allow us to talk about the state before and after evaluating a piece of code. We will use and around assertions about the contents of relevant cells. For example,

{!r = 100} deposit(100, r) =>* ( ) {!r = 200}

Recall that we also use [ ] around assertions about the values bound to relevant variables, to enable us to talk about value bindings produced by declarations and pattern matches.

Here is a more detailed summary of what happens when we evaluate deposit(100, r) from a state in which r is bound to cell l and the contents of l is initially 100:

\[
[r:l] \{!l = 100\} \text{deposit}(100, r) \\
= \Rightarrow \ [n:100, a:l] \{!l = 100\} (\text{update} (\text{fn} v => v+n) a) \\
= \Rightarrow \ {!l = 100} \text{update} (\text{fn} v => v+100) l \\
= \Rightarrow \ {!l = 100} \text{let \ val \ (ref v) = l in} (l := (\text{fn} v => v+100)(v)) \text{end} \\
= \Rightarrow \ [v:100] \{!l = 100\} (l := (\text{fn} v => v+100)(v)) \\
= \Rightarrow \ {!l = 100} (l := (\text{fn} v => v+100)100) \\
= \Rightarrow \ {!l = 100} (l := 100+100) \\
= \Rightarrow \ ( ) \{!l = 200\}
\]

In terms of equational reasoning, we have:

\[
\text{deposit}(100, r) \\
= \text{update} (\text{fn} v => v+100) r \\
= \text{let \ val \ (ref v) = r in} r := (\text{fn} v => v+100)(v) \text{end} \\
= \text{let \ val \ (ref v) = r in} r := v+100 \text{ end}
\]
6 List reversal, imperatively

Consider the following imperative code for reversing a list:

```ML
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;
```

- `fastrev` is a function of type `'a list -> 'a list`.
- The runtime for `fastrev L` is linear in length of `L`. The proof relies on an analysis of the runtime behavior of the function `loop`.
- For all lists `L`, `fastrev L = rev L`, where `rev` is the usual list-reversal function.

The proof of the third property requires a lemma about the `loop` function:

**Theorem**
For all types `t` and all list values `A` and `B` of type `t list`, evaluating `loop A` in a state where `!R = B` sets `!R` to the value of `rev(A) @ B` and returns this value.

This can be proven by induction on the length of `A`.

- **Base case**: For `A = [ ]`. Need to show that evaluating `loop [ ]` in a state where `!R = B` sets `!R` to the value of `rev [ ] @ B` and returns this value. But `rev [ ] = [ ]` and `[ ] @ B = B`, and according to the function definition we have

  `loop [ ] = !R`

So, evaluated from this state, `loop [ ]` returns `B` without changing anything. In this state, `!R` is equal to the value of `rev(A) @ B`, as required.
- Inductive case: For \( A = x::xs \). Assume as induction hypothesis that for all lists \( B' \), evaluating \( \text{loop } xs \) in a state where \(!R = B' \) sets \(!R \) to the value of \( \text{rev } xs \circ B' \) and returns this value. Need to show that for all lists \( B \), evaluating \( \text{loop}(x::xs) \) from a state where \(!R = B \) sets \(!R \) to the value of \( \text{rev}(x::xs) \circ B \) and returns this value.

So let \( B \) be a list and suppose we evaluate \( \text{loop}(x::xs) \) in a state where \(!R = B \). By definition,

\[
\text{loop}(x::xs) = (R := x :: (!R); \text{loop } xs)
\]

Given the assumptions about the current state, we have

\[
\{!R = B\} \text{loop}(x::xs) \\
\Rightarrow* (R := x :: (!R); \text{loop } xs) \\
\Rightarrow* (R := x :: B; \text{loop } xs) \\
\Rightarrow* \{!R = x::B\} \text{loop } xs \\
\Rightarrow* \{!R = (\text{rev } xs) \circ (x::B)\} !R \text{ by ind hyp}
\]

and the desired result follows because

\( (\text{rev } xs) \circ (x::B) = \text{rev } (x::xs) \circ B \).

Even though during the evaluation of \( \text{fastrev } L \) a fresh cell gets created and updated, since the cell is only used in a local binding inside the function body, and this binding is no longer in scope when the function call returns, and there’s no other way to access this cell, ML will “garbage collect” the cell. So \( \text{fastrev } L \) actually behaves just like a purely functional piece of code, in that it returns a list and has no observable side-effect. Using imperative features like this, in the background, can be beneficial: here, we obtained a linear reversal function rather than the naïve quadratic version. In a sense, the side effects in this example are benign.
7 Self-test

1. Check that for all well-typed expressions $e_1, e_2$ and $e_3$, the following equation is valid (i.e. in each case the two expressions have the same evaluation behavior and the same effect):

$$e_1; (e_2; e_3) = (e_1; e_2); e_3.$$ 

In other words, sequential composition is associative.

2. What assumptions about the values of $f$, $g$ and $r$ are sufficient to ensure that $\text{update } f \ r; \text{update } g \ r$ has the same effect and evaluation behavior as $\text{update } (f \circ g) \ r$? In other words, when is it the case that $\text{update } f \ r; \text{update } g \ r = \text{update } (f \circ g) \ r$?

3. Refer to the helper function $\text{loop}$ used in $\text{fastrev}$. Using our extended notation for evaluation of imperative code, show that for all list values $A$ and $B$, when $\text{loop } A$ is evaluated from a state in which the contents of the cell $R$ is $B$, the number of evaluation steps is $k \ast \text{length}(A) + c$, for some constants $c$ and $k$.

4. Consider the following code fragment:

```plaintext
local
  val a = ref 1
  val b = ref 1
in
  fun fib ( ) =
    let val t = !a in (a := !b; b := t + !b; !a) end
end
```

(a) What is the type of $\text{fib}$?

(b) What happens when we evaluate the following code, immediately after the above?

```plaintext
val (x, y, z) = (fib ( ), fib ( ), fib ( ))
```