Although SML 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 SML, 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 to augment our notion of extensional equivalence and we need to be careful when making substitutions based on referential transparency.

We introduce the main ideas, and develop some examples to illustrate the new concepts and contrast with the purely functional world.

Topics:

- imperative vs. functional programming
- mutable cells
- evaluation and side effects
- reasoning about effects
- persistent vs. ephemeral data structures
- benign effects

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 SML, 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.

For example, in the purely functional subset of SML, 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.

### 1.1 Mutable cells

The SML type system includes reference types of the form $t$ ref, where $t$ is a type. A value of type $t$ ref is a mutable cell capable of holding a value of type $t$.

There are three operations on refs: ref (initialization), ! (read) and := (update/write/assign). ref and ! are essentially functions, := is a binary infix operator.

#### Typing.

- ref $v$: 'a ref if $v$: 'a.
- !$v$: 'a if $v$: 'a ref.
- $v_1 := v_2$: unit if $v_1$: 'a ref and $v_2$: 'a.

#### Evaluation.

- **[Initialization]** ref $v$ creates a fresh mutable cell, distinct from all others created so far, which initially contains $v$, and returns the cell.
- **[Read]** If $v$ is a mutable cell, !$v$ evaluates to its current value.
- **[Update (Write)]** $v_1 := v_2$ updates the mutable cell $v_1$ to contain $v_2$ and evaluates to ()

Caution: Do not make the mistake of writing = for assignment; SML is not C.

Here is some code to illustrate:

We can create a new mutable cell representing a bank account, initialize its contents to 100, and bind acct to the cell:

```ml
val acct = ref 100
(* acct : int ref *)
```

We can then check the balance by reading the cell:

```ml
val bal1 = !acct
(* bal = 100 *)
```

Next, we increment the contents of acct by 100:

```ml
acct := !acct + 100
val bal2 = !acct
(* bal2 = 200 *)
```
It’s important to note that, with mutation, the same program can have different results if it is evaluated multiple times. For example, the two versions of the line `val X = !acct` bind two different numbers, depending on the value currently in the cell. This is characteristic of effects such as mutation and input/output, but not something that a pure functional program will do.

It’s also important to note that the value of the variable `acct` never changed. It is still a mathematical variable, just like always: it always stands for the same cell. However, that cell can have different values at different times, which is a new ingredient.

Now we declare a `new` cell and bind it to `acct2`:

```plaintext
val acct2 = ref 100
(* acct2 : int ref, distinct from the value of acct *)
```

Assigning to `acct2` has no effect on the contents of `acct1`:

```plaintext
acct2 := !acct2 + 50
(* !acct = 200, !acct2 = 150 here *)
```

However, we can introduce an alias for a cell (like a new debit card for the same account), e.g.,

```plaintext
val acct1' = acct1
(* acct1' : int ref, acct1' = acct1, !acct1' = !acct1 = 200 here *)
```

Ref cells can be tested for identity, and `acct1 = acct1'` evaluates to `true`, `acct1 = acct2` evaluates to `false`.

Now consider:

```plaintext
acct1' := !acct1' - 100
(* !acct1' = !acct1 = 100 *)
```

Assigning to `acct1'` also affects the value of `!acct1`, because `acct1'` evaluates to the same cell as `acct1`.

We can define the function:

```plaintext
fun update (f : 'a -> 'a) (r : 'a ref) : unit =
  r := f (!r)
```

It combines a read and a write; `update (fn x => x + 1) r` is kind of like `r++` (except that it returns `()`).

For example:

```plaintext
fun deposit n a = update (fn x => x + n) a
fun withdraw n a = update (fn x => x - n) a
```
2 Pattern Matching and ref Cells

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

- We can use variable patterns and wildcard to match against values of type \( t \) \texttt{ref} for some \( t \).
- And we can use patterns of form \( \texttt{ref} \ p \), where \( p \) is also a pattern, to match against cells whose contents (in the current state) match \( p \).

Here's an example:

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

Here we use the pattern \( \texttt{ref} \ v \) to bind \( v \) to the current contents of the cell denoted by \( r \).

3 Sequential Composition

As you may 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 first evaluating \( e_1 \), and then, if \( e_1 \) has a value, evaluating \( e_2 \) in the state produced by \( e_1 \). The value of \( e_1;e_2 \) is the value of \( e_2 \) thus obtained, if such a value exists. (The value of \( e_1 \) is again irrelevant, but must exist.)

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

\[
e_1; (e_2; e_3) \equiv (e_1; e_2); e_3
\]

and so we can just write \( e_1;e_2;e_3 \).

4 Reasoning about Imperative Code

Let’s prove that \( 1 == 2 \).

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

++ r 
== 1

(++) r ; ++ r
```
By reflexivity,
++ r == ++ r

Therefore, by transitivity,
1 == 2

Where did we go wrong?

One way to diagnose the problem is that reflexivity doesn’t hold: ++ r == ++ r implies that an expression means the same thing when you run it twice, but that’s not true in the presence of mutation!

Another way to think about this is that the value of an expression in the imperative fragment of SML 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 may themselves include ref cells). A state is an environment paired with a store.

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

Expressions of type int whose syntax uses imperative features are equivalent if in every state they evaluate to equal values (or both fail to terminate or both raise the same exception) AND have the same effect. (We can also extend equivalence to other types, along similar lines.)

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

• acct1 and acct2 are not equivalent (at type int ref).
• acct1 and acct1’ are equivalent (at type int ref).
• !acct1 and !acct1’ are equivalent (at type int).

With this extended notion of equivalence, 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 equivalent sub-expression. For example, in the context of the above sequence of declarations we get

\[
\text{deposit 100 acct1} \equiv \text{deposit 100 acct1’}
\]

Although we still get referential transparency, reasoning about imperative code can be tricky, because we can only substitute safely based on a notion of equivalence 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 the particular imperative program at hand.

As another example, consider the function tick : unit -> int defined by
val tick = 
  let val x = ref 0
  in
  fn () => (x := !x + 1; !x)
  end

Even though tick only affects a locally created ref cell, every time we call this function we get a different value. It is not possible to say any more that the meaning of a value of type unit -> int is representable mathematically as just a function from (values of type) unit to (values of type) 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.

Remark: A variation of the tick example may be used to encapsulate code for pseudo-random number generators. The local state in such a generator holds some integer or real value that is manipulated with every call to produce a very different value.

Another remark: You can rewrite this code as

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

This allows tick to be visible outside, but not x.

Question: What assumptions about the values of f, g and r are sufficient to ensure that

update f r; update g r

is equivalent to (has the same effect and produces the same value, when executed from the same state as)

update (f o g) r ?

5 Persistent vs. Ephemeral Data Structures

All of the data structures we’ve seen so far in the course are persistent: functions that operate on them return a new data structure rather than modifying the old one. This has to be the case because, until now, we haven’t seen any way of modifying a data structure. Consider the cons function for lists:

fun cons (x : 'a) (xs : 'a list) : 'a list =
  x::xs

The cons function returns a new list. The old one is still around and hasn’t changed:
With references, we can create *ephemeral* data structures which are actually modified in-place by operations.

```ocaml
datatype 'a mlist = MList of 'a mfront ref
and    'a mfront = Nil | Cons of 'a * 'a mlist

fun empty () : 'a mlist = MList (ref Nil)
fun cons (x : 'a) (MList rxs : 'a mlist) : unit =
    rxs := Cons (x, MList (ref (!rxs)))

fun toList (MList (ref Nil)) = []
    | toList (MList (ref (Cons (x, xs)))) = x::(toList xs)
```

It's easy to make destructive mistakes with ephemeral data structures:

```ocaml
fun length (MList r) =
    case !r of
        Nil => 0
    | Cons (x, MList xs) => (r := !xs; 1 + length (MList r))
```

Oops.

Persistent data structures are nice for other reasons too. Think about how much harder your Checkers alpha-beta code would have been to write if `make_move` had the signature:

`make_move : Game.state -> Game.move -> unit`
6 Imperative List Reversal

Consider the following imperative code for reversing a list:

```sml
fun fastrev (L : 'a list) : 'a list = 
  let
    val R = ref [ ]
    fun rloop [ ] = !R
    | rloop (x::xs) = (R := x :: (!R); rloop xs)
  in
    rloop L
  end

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

A proof of this last property would make use of the following theorem about the rloop function:

Theorem
For all types t and all list values A and B of type t list, evaluating rloop A in a state where !R is B sets the contents of R to the value of List.rev(A) @ B and returns that value.

This can be proven by structural induction on A.

Remark: Evaluation of fastrev L creates a fresh ref cell and then updates that cell within rloop. Since the cell is only used in a local binding inside the function body of fastrev, and since this binding is no longer in scope when the function call returns, and since there is no other way to access this cell, SML will ultimately “garbage collect” the cell. So 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. The side effects in this example are said to be benign.

7 Benign Effects

There are many other ways in which side effects may yield benefits, either in improving efficiency or in achieving correct extensional behavior. With care, we can use side-effects to communicate useful information from one computation to subsequent computations. In the example considered next, we actually achieve termination by avoiding fruitless search, through the judicious use of side effects.
Graph Reachability

A simple way to represent a directed graph is as a successor function:

    type graph = int -> int list

A value \( g \) of type \( \text{graph} \) represents a graph with integer nodes; for each \( n : \text{int} \), \( g(n) \) is the list of successor nodes in the graph. A leaf node has the empty list of successors.

For example:

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

Observe that \( G \) has a directed cycle between nodes 1 and 2.

The reachability problem for graphs is to define a function

\[
    \text{reachable : graph} \rightarrow \text{int} \times \text{int} \rightarrow \text{bool}
\]

such that given a graph \( g \) and two nodes \( x \) and \( y \) in \( g \), \( \text{reachable} \ g \ (x, y) \) evaluates to \text{true} if \( y \) is reachable from \( x \) and evaluates to \text{false} otherwise.

First Attempt

One solution is to design a function that walks down the graph, starting from node \( x \), looking for node \( y \). Here is a first attempt to do this, using purely functional code.

Recall (or look up) the function

\[
    \text{List.exists : ('a -> bool) -> 'a list -> bool}
\]

for determining whether a list contains an element satisfying a predicate. We also define the following function that tests whether an integer is present in a list:

\[
    (* \text{mem : int -> int list -> bool} *)
\]

\[
    \text{fun mem (n:int) = List.exists (fn x => x=n)}
\]

Now for the function \( \text{reachable} \):

\[
    \text{fun reachable (g:graph) (x:int, y:int) : bool =}
\]

\[
    \begin{align*}
    \quad & \text{let} \\
    \quad & \quad \text{dfs (n:int) : bool = (n=y) orelse (exists dfs (g n))} \\
    \quad & \quad \text{in} \\
    \quad & \quad \text{dfs x} \\
    \quad & \text{end}
    \end{align*}
\]

(Aside: “dfs” stands for “depth first search.”)

Intuitively, \( \text{reachable} \ g \ (x, y) \) calls the local function \( \text{dfs} \ x \); this checks whether \( x = y \). If so, \( y \) is (trivially) reachable from \( x \). If not, the function recursively checks whether \( y \) is reachable from one of the successor nodes of \( x \). And so forth.
Problem: The recursive call pattern may get stuck in a cycle. For example, if we evaluate

\[ \text{reachable } G \ (1, 4) \]

the code begins with a call to \( \text{dfs} \ 1 \), which steps to \( \text{exists} \ \text{dfs} \ [2,3] \). This calls \( \text{dfs} \ 2 \), which in turn calls \( \text{exists} \ \text{dfs} \ [1,3] \). That calls \( \text{dfs} \ 1 \), and now the evaluation is in a loop.

A solution: In order for the search to terminate, the code should keep track of previously encountered nodes.

We could write purely functional code to keep track of such information, but the code is ugly. In particular, it is cumbersome to share information functionally between different paths through the graph. Doing so becomes reasonably straightforward using mutable state.

Second Attempt

\[
\begin{align*}
\text{fun reachable } \ (g:\text{graph}) \ (x:\text{int}, y:\text{int}) \ : \ \text{bool} = \\
\text{let} \\
\quad \text{val visited} = \text{ref} \ [\] \\
\quad \text{fun dfs } \ (n:\text{int}) \ : \ \text{bool} = \\
\quad \quad (n = y) \ \text{orelse} \\
\quad \quad \text{let} \\
\quad \quad \quad \text{val } V = !\text{visited} \\
\quad \quad \quad \text{in} \\
\quad \quad \quad \quad \text{if } \text{mem } n \ V \ \text{then false else} \\
\quad \quad \quad \quad (\text{visited} := n::V; \ \text{exists} \ \text{dfs} \ (g \ n)) \\
\quad \quad \text{end} \\
\quad \text{in} \\
\quad \text{dfs } x \\
\text{end}
\end{align*}
\]

- A call to \( \text{reachable } g \ (x, y) \) creates a local cell initialized to the empty list. A call to \( \text{dfs} \ n \) checks if \( n \) is the target node \( y \), and returns \text{true} if it is. Otherwise, it checks if \( n \) is in the current visited list; if so, the search has already visited this node and there is no point looking further. If further search is required, then the code first updates the visited list by adding node \( n \) and then checks whether there is some successor of \( n \) from which \( y \) is reachable. If there are no further successor nodes, this exploration terminates with \text{false}.

- The \text{visited} list is used to prevent fruitless search along cycles.

- The use of cells and mutation here is \textit{benign}: the function call \( \text{reachable } g \ (x, y) \) always produces the same truth value, even if we evaluate \( \text{reachable } g \ (x, y) \) multiple times. There are no visible effects, since only locally bound cells get updated.

Using the above code on the graph \( G \), we get

\[
\begin{align*}
\text{reachable } G \ (1,4) & \Rightarrow \text{true} \\
\text{reachable } G \ (3,2) & \Rightarrow \text{false} \\
\text{reachable } G \ (1,5) & \Rightarrow \text{false}
\end{align*}
\]
Memoization

Another potentially beneficial use of mutation is to remember values that were difficult to compute. For instance, it might take years to compute the two-millionth prime. We might want to remember it, rather than compute it again the next time we need it. It is possible to modify our Stream implementation from last time so as to remember values once they have been computed, in a way that is transparent to users except for the time of computation and any side effects produced by the stream itself. Reducing such side-effects by remembering values may actually be beneficial in certain situations, for instance when doing I/O. If a stream represents input from a user, we may not want to ask the user re-input data needed during computations that access a stream repeatedly.

We’d need a different technique for doing memoization than what we used above for reachable. In our code above, each call to reachable generates a fresh, new cell which is then inaccessible upon termination of reachable. For memoization, the whole point is that the value sticks around for future calls. How could we make sure that the ref cell stays around for future calls to a memoized function?