15–150: Principles of Functional Programming

Imperative Programming

Michael Erdmann[*]
Spring 2020

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
- benign effects
- parallel evaluation and 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 looping forever.

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` reduces to a value then the value

[*] Adapted from a document by Stephen Brookes.

of e=e is true. Hence (assuming that e reduces to a value) we can replace e=e by true, without affecting the behavior of the program. But if e=e includes imperative constructs, the value of e=e is no longer so easy to predict.

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 a mutable cell capable of holding a value of type t.

- [**Initialization**] We can create a *fresh* cell, initialized to contain a given value, by using the function ref : 'a -> 'a ref .
- [**Read**] We can evaluate the contents of a cell by using the unary prefix operator ! .
- [**Write**] We can update the contents of a cell using the the infix assignment operator := .

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

Here is some code to illustrate:

First, we create a new mutable cell, initialize its contents to 0, and bind r to the cell:

```
val r = ref 0
(* r : int ref *)
(* !r = 0 here *)
```

Next, we increment the contents of r by 1:

```
r := !r + 1
(* !r = 1 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 content of this cell has been changed to 1.

Now we declare a new cell and bind it to s:

```
val s = ref 1
(* s : int ref, distinct from the value of r *)

s := !s + 1
(* !r = 1, !s = 2 here *)
```

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

```
s := !s + 1
(* !r = 1, !s = 3 here *)
```

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

```
val q = r
(* q : int ref, q = r, !q = !r = 1 here *)
```

Ref cells can be tested for identity, and `q = r` evaluates to `true`, `r = s` evaluates to `false`.

Now consider:

```
q := !q + 1
(* !q = !r = 2 *)
```

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

2 Reasoning about Imperative Code

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 raise an exception or loop forever) and may cause an *effect*, which we interpret as a state change.

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

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

- `r` and `s` are *not* equivalent (at type `int ref`).
- `r` and `q` are equivalent (at type `int ref`).
- `!r` and `!q` are equivalent (at type `int`).
- `(!r) + (!r)` and `(!q) + (!q)` are equivalent (at type `int`).

With this extended notion of extensional 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 extensionally equivalent sub-expression. For example,

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

declares a function `inc : int ref * int -> unit`, and in the context of the above sequence of declarations we get

$$\texttt{inc(q, 42)} \cong \texttt{inc(r, 42)}$$

Indeed, we also get `inc(q, 42)` $\cong$ `(q := !q + 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 extensional equivalence that takes account of effects as well as values. In particular, extensional equivalence 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

```
   local
      val x = ref 0
   in
      fun tick( ) = (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.

3 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 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`.

Here are some examples:

```
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 also have written

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

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

```
   update f r; update g r
```

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

```
   update (g o f) r  ?
```

4 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 `e1` and `e2` are well-typed expressions of types `t1` and `t2`, `e1;e2` is a well-typed expression of type `t2`. (The type of `e1` is irrelevant to the ultimate type of `e1;e2`, but it must exist). From any initial state, the value of `e1;e2` is obtained by first evaluating `e1`, and then, if `e1` has a value, evaluating `e2` in the state produced by `e1`. The value of `e1;e2` is the value of `e2` thus obtained, if such a value exists. (The value of `e1` is again irrelevant, but must exist.)

Semicolon is associative: for all well-typed expressions `e1`, `e2` and `e3`,

$$\mathtt{e1}; (\mathtt{e2}; \mathtt{e3}) \cong (\mathtt{e1}; \mathtt{e2}); \mathtt{e3}$$

and so we can just write `e1;e2;e3`.

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
```

Suppose we execute the following code fragment, which uses sequential composition:

```
  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 (sequentially executed) code fragments are extensionally equivalent.

6 Imperative List Reversal

Consider the following imperative code for reversing a list:

```
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` $\cong$ `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 Beneficial Side 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.

7.1 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 graph represents a graph with integer nodes; for each n: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

```
reachable : graph -> int * int -> bool
```

such that given a graph g and two nodes x and y in g, reachable g (x, y) evaluates to true if y is reachable from x and evaluates to 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

```
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:

```
(* mem : int -> int list -> bool *)
fun mem (n:int) = List.exists (fn x => x=n)
```

Now for the function reachable:

```
fun reachable (g:graph) (x:int, y:int) : bool =
      let
         fun dfs (n:int) : bool = (n=y) orelse (exists dfs (g n))
      in
         dfs x
      end
```

(Aside: "dfs" stands for "depth first search.")

Intuitively, reachable g (x, y) calls the local function 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

```
reachable G (1, 4)
```

the code begins with a call to `dfs 1`, which steps to `exists dfs [2,3]`. This calls `dfs 2`, which in turn calls `exists dfs [1,3]`. That calls `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 it is cumbersome to share information functionally between different paths through the graph. Doing so becomes reasonably straightforward using mutable state.

Second Attempt:

```
fun reachable (g:graph) (x:int, y:int) : bool =
     let
        val visited = ref [ ]
        fun dfs (n:int) : bool =
              (n = y) orelse
                       let
                          val V = !visited
                       in
                         (not (mem n V))
                             andalso
                         (visited := n::V; exists dfs (G n))
                       end
     in
        dfs x
     end
```

- A call to `reachable g (x, y)` creates a local cell initialized to the empty list. A call to `dfs n` checks if `n` is the target node `y`, and returns `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 `false`.

- The `visited` list is used to prevent fruitless search along cycles.

- The use of cells and mutation here is *benign*: the function call `reachable g (x, y)` always produces the same truth value, even if we evaluate `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

```
reachable G (1,4) ==> true
reachable G (3,2) ==> false
reachable G (1,5) ==> false
```

Side comment: A more efficient implementation might use a binary search tree in place of a list to keep track of visited nodes. We could also represent neighbors by sequences and perform parallel depth-first searches from neighbors. Some of these would then be redundant and visit nodes more than once, depending on the particular concurrency issues associated with updating the visited data structure. See Section 8 for more such issues.

7.2 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 an I/O stream repeatedly.

8 Effects and Parallel Evaluation

Let us return to the topic of parallelism. Previously, we defined some imperative functions for manipulating bank accounts.

What happens if we use parallelism, as in:

```
val r = ref 100;
val _ = Seq.tabulate(fn 0 => deposit(100, r)
                      | 1 => withdraw(50, r)
                      | _ => raise Range)
                     2
```

The deposit and withdrawal are run in parallel. But the deposit code actually *reads* the contents of `r` first, then *writes* the updated value. Similarly the withdraw code *reads* the contents of `r` before writing the updated value. In each case, the updated value gets computed as an increment or decrement of whatever value was read. So it is possible (depending on the process scheduling) for the two reads to happen *before* either of the two writes. And then the order in which the writes get scheduled will determine who gets to set the final contents of `r`. The possible final values for the contents of `r` include `50, 150`, and `200`. (There could be yet other values, if individual bits of memory may be written in parallel asynchronously by multiple processors.) This is probably *NOT* what a bank or an account holder would desire.

This kind of problem, in which it is possible for multiple processors to access the same cell simultaneously, with one or more processors trying to write, is called a *race condition.* The value produced, and the effect caused, is hard to predict and may depend on scheduling quirks. It is extremely hard to keep track of side-effects if they could come from multiple processors. So it is a bad idea to write code that may be susceptible to race conditions.

Solutions to this problem include:

- Use *locks* to enforce *atomic* execution of updates to shared state, so no other processor can be trying to read or write; however, programming with locks is hard to get right. We will not take this path in this class.
- Avoid side effects in parallel code. Rely on purely functional programming whenever you want to exploit parallelism. There are never any race conditions when you program functionally (because there are no side effects). This is the approach that we advocate. It is also OK to use *benign* or *beneficial* side effects in parallel code, because benignity means again that there won't be any race conditions.