1 Topics

Last time we introduced some imperative features of ML, and discussed their use in a sequential setting. We assumed that code is executed according to a simple sequential evaluation strategy. Although we did not say so explicitly, it is important when writing imperative code to be aware of the evaluation order of sub-expressions, to keep track of the effects of one sub-expression evaluation on subsequent stages of evaluation. This is reasonably easy to keep track of when the control flow is sequential, and we stressed the importance of sequential composition (semi-colon) in designing imperative code.

Today we consider some more sophisticated examples, and we reflect on what advantages and disadvantages are inherent in imperative programming. We consider in particular what happens when we try to combine imperative features with parallel evaluation?
2 Introduction

To begin, recall that we defined an imperative function for reversing lists:

```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`, and the runtime for `fastrev L` is linear in length of `L`. For all list values `L`, `fastrev L = rev L`, where `rev` is the usual list-reversal function.

We proved that the imperative function `fastrev` is extensionally equivalent to a function `rev` written in the purely functional subset of ML. We say that `fastrev` uses effects in a benign manner: its behavior (when applied to a list value) doesn’t depend on the store from which it is executed (because it introduces a fresh cell for local use only), it returns a result value independent of the initial store, and it causes no observable side-effect (again, the only cell that gets updated is the local one, and this ceases to be accessible or visible when the function returns). So for a given list value `L`, evaluating `fastrev L` multiple times yields the same result value every time, and evaluating `fastrev L` has no effect on the value of any other expression.

It should be fairly obvious that it’s OK to use benign side effects like this in parallel, as in:

```ml
fun revALL (S: 'a list seq) : 'a list seq = Seq.map fastrev S;
```

Even if we use multiple processors to evaluate the applications of `fastrev` to the lists in the sequence `S`, the end result is just the sequence whose elements are the reverse of the corresponding elements of `S`.

```
revALL ⟨L₁,...,Lₙ⟩ = ⟨rev(L₁),...,rev(Lₙ)⟩
```

and the order of evaluation is irrelevant. Note also that (assuming that the `Lᵢ` are values already) the work done by this code is $O(\sum_{i=1}^{n} \text{length}(Lᵢ))$ and the span is $O(\max\{\text{length}(Lᵢ) \mid 1 \leq i \leq n\})$. 

2
3 Beneficial side effects, in action

The `fastrev` function used side effects not just benignly, but for a purpose: to obtain linear runtime rather than the quadratic runtime of the naïve `rev` function. So we can also characterize this as an instance of *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.

Graph reachability

A simple way to represent a directed graph with nodes that contain data is as a successor function:

```plaintext
type 'a graph = 'a -> 'a list
```

A value `G` of type `int 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:

```plaintext
val G0 = fn 1 => [2,3]
| 2 => [1,3]
| 3 => [4]
| _ => []
```

Note that `G0` has cycles!

The reachability problem for graphs is to define a function

```plaintext
reachable : ''a graph -> ''a * ''a * ''a -> 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`, `false` otherwise.
First attempt

The obvious solution is to design a function that walks down the graph, starting from node \( x \), looking for node \( y \). Here is an attempt to do this, using purely functional code. We’ll use the ML functions

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

defined previously; we assume you are familiar with their specifications.

\[
\text{fun reachable (G:'a graph) (x:'a, y:'a) : bool =}
\]
\[
\text{let}
\]
\[
\text{fun dfs (n:'a) : bool = (n=y) orelse (exists dfs (G n))}
\]
\[
in
\]
\[
\text{dfs x}
\]
\[
\text{end;}
\]

It’s easy to see how we thought of this code: \text{reachable G (x, y)} calls the local function \text{dfs x}; this checks if \( x = y \), in which case \( y \) is (trivially) reachable from \( x \); otherwise, if \( x \) is not the same node as \( y \), it (makes recursive calls and) checks if \( y \) is reachable from one of the successor nodes of \( x \). However, the recursive call pattern here may get side-tracked along a cycle. For example, if we evaluate

\[
\text{reachable G0 (1, 4)}
\]

we’ll begin with a call to \text{dfs 1}, which steps to \text{exists dfs [2,3]}; this calls \text{dfs 2}, which in turn calls \text{exists dfs [1,3]}; this calls \text{dfs 1}, and we’re in a loop! We can show this more rigorously using the evaluation notation, as follows:

\[
(i) \quad \text{dfs 1 =>* exists dfs [2,3]}
\]
\[
=>* (dfs 2) orelse (exists dfs [3])
\]

\[
(ii) \quad \text{dfs 2 =>* exists dfs [1,3]}
\]
\[
=>* (dfs 1) orelse (exists dfs [3])
\]

(iii) So, from (i) and (ii), we have
dfs 1 =>* exists dfs [2,3]
    =>* (dfs 2) orelse (exists dfs [3])
    =>* ((dfs 1) orelse (exists dfs [3])) orelse (exists dfs [3])

(iv) It follows from (iii) that there is an infinite sequence of evaluation steps from dfs 1.

In order for the search to terminate we’ll need to keep track of which nodes we’ve seen earlier; if we don’t, we may get side-tracked along an infinite cycle that doesn’t contain the target node, even though there may be other paths in the graph that do contain it. Moreover, it’s a good idea to share information about visited nodes from exploring one path when exploring later paths! So it is natural to use mutable state here. This leads us to...

Second attempt

fun reachable (G:'a graph) (x:'a, y:'a) : bool =
  let
    val visited = ref [ ]
    fun dfs (n:'a) : bool =
      (n = y) orelse
      let
        val V = !visited
      in
        if mem n V then false else
        (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, we’ve already visited this node and there’s no point looking further, so it returns false; if not, update the visited list (insert n) and check if there is some successor of n from which y is reachable.

• The visited list is used to prevent fruitless search along cycles.
• The work for checking reachability is linear in the number of nodes + the number of edges.

• It’s not possible to exploit parallelism inside the body of dfs by working with a sequence of visited nodes rather than a list (and using a reduce instead of the implicit fold inside the exists function). The reasons: two processors might find themselves simultaneously updating the cell holding the visited information, which would be a race condition; and we wouldn’t be able to ensure that each processor knows accurate information about the visited list.

• 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 repeat it. There are no visible effects, since only locally bound cells get updated.

Using the above code on the graph G0, we get

reachable G0 (1,4) = true
reachable G0 (3,2) = false
reachable G0 (1,5) = false

We can also build an instrumented version of this code, in which we return the final contents of the visited list. This is useful for testing.

fun reachable_plus (G:'a graph) (x:'a, y:'a) : bool * 'a list =
let
  val visited = ref [ ]
  fun dfs (n:'a) : bool =
    (n = y) orelse
    let
      val V = !visited
    in
      if mem n V then false else
      (visited := n::V; exists dfs (G n))
    end
  in
    (dfs x, !visited)
  end
Some results:

reachable_plus G0 (1,4) = (true, [3,2,1])
reachable_plus G0 (3,2) = (false, [4,3])
reachable_plus G0 (1,5) = (false, [4,3,2,1])

Since each call to reachable creates its own fresh cell to use for a list of visited nodes, and this cell is no longer accessible after the call returns, again this is an example of benign side-effects. It is perfectly safe to perform parallel calls to reachable. However, as we noted in the comments, it’s not safe to try to use parallelism inside the dfs code, because we want to explore the graph depth-first but always using the most up-to-date visited list. If we explored from more than one successor node of n in parallel, it wouldn’t be possible to ensure that each always sees the accurate visited list.

Specifications

What is an appropriate spec for reachable? We’d like to prove that, for an equality type t and a value G of type t graph, and values x,y of type t,

reachable G (x, y) returns true if there is a path from x to y in the graph represented by G, returns false otherwise.

Clearly this isn’t plausible if G has infinitely many nodes, because it would then be impossible to check non-existence of a path. So we’ll assume that G is a finite graph, i.e. that G is a total function and that there are only finitely many arguments n such that G(n) is non-empty. (Our definition of finite graphs does allow cycles to exist.)

As usual, we need a spec for the helper function dfs. Since this function is defined locally, inside the body of reachable G (x, y), the spec for dfs needs to mention G and y, and the ref cell denoted by visited; here it is:

For all values V : t list and x:t, when evaluated in a state with !visited = V, dfs x returns true if there is a path in the graph represented by G from x to y that does not pass through any node in V; returns false otherwise.

(The call to dfs x also updates the contents of visited, but we’ll not try to say anything specific here about that.)
We haven’t proven this result! But assuming we did, it would then be obvious that reachable satisfies its spec: simply let \( V \) be the empty list and simplify what the spec for \( \text{dfs} \ x \) says.

### 4 Memoizing intermediate results

As another demonstration of the utility of imperative features, let’s revisit lazy lists. We’ll generalize in a way that allows us to exploit side-effects to avoid re-calculating items. This amounts to a kind of *memoization*.  

We will obtain implementations of lazy lists that behave as if they are purely functional, in that if we ask for the first 100 items, then ask again for the first 100 items, we’ll get back the same list; but the first computation causes the results to get stored (memoized, or remembered) and the second computation merely retrieves these results without performing any significant effort.

First, a signature with a type constructor for building “suspended values”, or “computations”, that can be “forced” to return a value; the signature also specifies a way to turn a thunk into a suspended value.

```plaintext
signature LAZY =
  sig
    type 'a susp
    val delay : (unit -> 'a) -> 'a susp
    val force : 'a susp -> 'a
  end
```

Now we’ll develop 3 different implementations of this signature.

---

1It’s also possible to implement functions that use private state to store argument-result pairs and avoid re-calculation; this technique is also known as memoization.
(i) Thunks

We can implement this signature very simply, by representing a suspended value as a thunk! The delay function is then trivial (the identity function) and forcing is simply applying the think to its dummy argument:

```haskell
structure Thunk : LAZY =
struct
  type 'a susp = unit -> 'a
  fun delay f = f
  fun force f = f( )
end
```

In essence this idea here is very similar to how we did lazy lists previously.

(ii) Thunks and references to values

Another implementation is to use a thunk together with a reference to a value of option type. When the contents of the ref cell is NONE the suspended value is still suspended; the first time we force it, we update the cell to contain SOME v, where v is the value obtained from the thunk body; in subsequent forces we get back this value immediately, without re-using the thunk.

```haskell
structure ThunkOptionRef : LAZY =
struct
  type 'a susp = (unit -> 'a) * 'a option ref
  fun delay f = (f, ref NONE)
  fun force (f, r) =
    case (!r) of
      SOME v => v
    | NONE => let val v = f( ) in (r := SOME v; v) end
end
```

The semi-colon in the code for force is very necessary! We need to return the value v ultimately, but we also need to cause an update to the contents of r. Using ; as shown will do both of these things. Also note that the v in the SOME branch of the case has nothing to do with the v in the NONE branch!
(iii) References to thunks

Finally we can implement LAZY using a reference to a thunk; to delay a thunk \( f \) we create a fresh ref cell initialized to a thunk designed so that the first time it gets used it calls \( f \) to obtain a value \( v \), then updates this cell so that in subsequent forces we’ll get back the value \( v \); the force operation is obvious: dereference the cell and apply its contents to the dummy argument.

```ml
structure RefThunk : LAZY =
struct
  type 'a susp = (unit -> 'a) ref
  exception Bad
  fun force r = (!r) ()
  fun delay f =
    let
      val memo = ref (fn () => raise Bad)
    in
      memo := (fn () => let val v = f () in memo := (fn () => v); v end);
      memo
    end
end
```

Again the semi-colons are vital! Their use allows us to combine the desired effects with delivery of the right value. This code is quite tricky! Make sure you understand how it works. It might be a good idea for you to walk through the evaluation steps that occur when you evaluate (assuming you are working inside the body of `RefThunk`) the following code fragment:

```ml
delay (fn () => fib 10);
```

where you can assume that `fib 10` evaluates to the integer value 89.
Constructing lazy lists from suspensions

Now we can encapsulate a construction for lazy lists of data over an arbitrary LAZY structure. This construction makes sense for any choice of an implementation of suspended values, embodied as a structure that implements the signature LAZY. Thus we will define a functor that allows us to re-use the same construction over and over.

First, a signature LAZYLIST specifying a type constructor and functions for lazy mapping a function and for showing the first few items in a lazy list.

signature LAZYLIST =

sig
    structure S : LAZY;
    datatype 'a Lazylist = Cons of 'a * 'a Lazylist S.susp;
    val Lazymap : ('a -> 'b) -> 'a Lazylist S.susp -> 'b Lazylist S.susp
    val Show : int -> 'a Lazylist S.susp -> 'a list
end;

Next a functor Listify that builds an implementation of LAZYLIST when supplied with an implementation of LAZY. In order to test this code we’ve deliberately omitted the signature ascription :LAZYLIST from the functor header, and we included some functions for generating lazy lists of natural numbers and the lazy list of Fibonacci numbers. The function loudfib is for testing – and for demonstrating that indeed the imperative features can avoid re-calculation.

functor Listify (S : LAZY) =

struct

    structure S = S

    datatype 'a Lazylist = Cons of 'a * 'a Lazylist S.susp

    fun Lazymap f s =
        S.delay(fn () =>
            let
                val Cons(x, t) = S.force s
            in
                Cons(f x, Lazymap f t)
            end)

end;
fun Show 0 s = [ ]
  | Show n s = let val Cons(x, t) = S.force s in x :: Show (n-1) t end

fun natgen n = fn ( ) => Cons(n, S.delay(natgen (n+1)))

val natsusp = S.delay (natgen 0)

val nats = S.force natsusp

fun fib 0 = 1 | fib 1 = 1 | fib(n) = fib(n-1) + fib(n-2)

fun loudfib n = 
( print ( "Calling fib " ^ Int.toString n ^ "\n" );
  fib n )

val Fibs = Lazymap loudfib natsusp
end

We can use this functor to build 3 different representations for lazy lists:

structure S1 = Listify(Thunk);
structure S2 = Listify(ThunkOptionRef);
structure S3 = Listify(RefThunk);

The implementation defined in S1 is very similar to our direct hand-coded treatment of lazy infinite lists in class.

Here is a transcript of the results:

- S1.Show 10 S1.Fibs;
Calling fib 0
Calling fib 1
Calling fib 2
Calling fib 3
Calling fib 4
Calling fib 5
Calling fib 6
Calling fib 7
Calling fib 8
Calling fib 9
val it = [1,1,2,3,5,8,13,21,34,55] : int list

- S1.Show 10 S1.Fibs;
Calling fib 0
Calling fib 1
Calling fib 2
Calling fib 3
Calling fib 4
Calling fib 5
Calling fib 6
Calling fib 7
Calling fib 8
Calling fib 9
val it = [1,1,2,3,5,8,13,21,34,55] : int list

The first evaluation calls fib 10 times. The second evaluation of the same expression does all the same work again; this is all purely functional, no values are being stored for later retrieval! And the second computation returns exactly the same list as the first one did.

On the other hand, if we use S2, we get

- S2.Show 10 S2.Fibs;
Calling fib 0
Calling fib 1
Calling fib 2
Calling fib 3
Calling fib 4
Calling fib 5
Calling fib 6
Calling fib 7
Calling fib 8
Calling fib 9
val it = [1,1,2,3,5,8,13,21,34,55] : int list

- S2.Show 10 S2.Fibs;
val it = [1,1,2,3,5,8,13,21,34,55] : int list

The first time, again we have to call fib 10 times. But the values obtained in these calls are stored inside the lazy data structure, so the second call
merely retrieves these values without incurring any calls to \texttt{fib}. Again we get exactly the same list the second time; but (as the printed information – or lack thereof) shows, the results of the first round of calls to \texttt{fib} were stored and retrieved.

Results for S3 are the same as for S2.

5 Persistent or ephemeral?

Using imperative features, we can define signatures that resemble the kind of interfaces you typically see in languages like Java and C. For example, let’s reconsider the \texttt{GAME} signature from before. We defined:

\begin{verbatim}
signature GAME =
sig
type state
  type move
  val start : state
  val step : state * move -> state
  val moves : state -> move Seq.seq
...
end
\end{verbatim}

A structure with this signature must contain a function \texttt{step} that builds a new state from a state and a move, the new state representing the result of making the given move from the given state. It’s obvious from the type (and because we were doing purely functional programming then) that the “old” state doesn’t get “updated” – the new state is constructed and returned by \texttt{step}. We refer to this as a \textit{persistent} signature, because it requires that states persist; this persistence was crucial in what we wanted to do – it allowed us to design a \texttt{search} function and an \texttt{evaluate} function that performed backtracking. We were able to explore the possible moves from a given state and then come back and re-examine the same game state that we started with, knowing that it hadn’t been overwritten or destroyed by the search so far. We defined a structure \texttt{Nim} with signature \texttt{GAME}, by:

\begin{verbatim}
structure Nim : GAME =
struct
  exception Fail of string
\end{verbatim}
type state = int
type move = int
fun step (p, n) =
    if p >= n then p - n
    else raise Fail "illegal move";

Here is a signature that specifies an ephemeral interface: making a move has a side effect and (presumably) updates the state imperatively. To make it possible to implement suitable searching and game state evaluation functions, we also include an undo_step function!

signature EPH_GAME =
    sig
        type state
        type move
        val score : state -> int
        val step : state * move -> unit
        val undo_step : state * move -> unit
        val moves : state -> move Seq.seq
    end

And here is an imperative implementation of Nim:

structure Eph_Nim : EPH_GAME =
    struct
        type state int ref
        type move int
        fun score s = if !s = 0 then 1 else 0
        fun step (r as ref p, n) =
            if p >= n then r := p - n
            else raise Fail "illegal move"

        fun undo_step (r as ref p, n) = (r := p + n)

        fun moves s = case !s of ....
    end
Let’s now attempt to define a functor for building imperative game players that use the minimax algorithm for exploring game states. We must rewrite the old minimax functor, which worked with the persistent GAME signature, to work with the ephemeral one. In particular, we cannot just call \( G(\text{step}(s,m)) \) to compute the integer value of the state reached by making move \( m \) from game position \( s \). Instead we’ll need to make the move, compute the integer, then undo the move and return the integer.

```ocaml
signature EPH_PLAYER =
  sig
    structure Game : EPH_GAME;
    val player : Game.state -> Game.move
  end

functor Eph_MiniMax (Game : EPH_GAME) : EPH_PLAYER =
  struct
    structure Game = Game
    open Game; ...
    fun F (s : state) : int =
      let
        val M = moves s
      in
        if (null M) then (score s) else
          reduce1 Int.max (Seq.map (fn m => (step(s,m);
            let
              val v = G(s)
            in
              undo_step(s,m);
              v
            end)) M)
      end
    and G (s : state) : int = ...
  end
```
The main difference is that we replaced the code fragment \( G(\text{step}(s,m)) \) from the persistent version with

\[
(\text{step}(s,m);
  \text{let}
  \quad \text{val } v = G(s)
  \text{ in }
  \quad \text{undo}_\text{Step}(s,m);
  \quad v
\text{ end})
\]

But there is a problem here! This code works with sequences, and

\[
\text{Seq.map } (\text{fn } m \mapsto \ldots) \ M
\]

may apply this function (which calls \text{step}(s,m)) in parallel, to all moves in \( M \). These calls to \text{step} have a side effect, and these parallel calls may interfere with each other! For example, the undo part of one move calculation might occur in the middle of another move’s code, causing the other move calculation to call \( G \) on a “wrong” state!

It’s better to redesign the ephemeral minimax functor to be \textit{sequential}, using lists rather than sequences, to avoid this kind of problem.

First we’ll adjust the game signature:

\begin{verbatim}
signature EPH_GAME =
  sig
    type state
    type move
    val start : state
    val step : state * move -> unit
    val undo_step : state * move -> unit
    val moves : state -> move list (* not sequences! *)
    ...
  end
\end{verbatim}

Of course we’ll need to adjust the Nim game, and any other games we want to play, to work with this new signature.

And we can adapt the functor from above, which was racy because of its use of parallel operations having side effects, so that the side effects all occur in a predictable sequential order:
functor Eph_MiniMax (Game : EPH_GAME) : EPH_PLAYER =
  struct
    structure Game = Game;
    open Game;
    ...
    fun reduce1 g (x::L) = foldr g x L
    (* sequential combination using g *)

    fun F (s : state) : int =
      let
        val M = moves s
      in
        if (null M) then (score s) else
          reduce1 Int.max
          (List.map (fn m => (step(s,m);
            let
              val v = G(s)
            in
              undo_step(s,m);
              v
            end)) M)
      end
    and G (s : state) : int =
      ...(*) similarly *)
  end

  With this implementation we can build imperative/ephemeral minimax players for games, and their move selections should be exactly the same as those made by the corresponding persistent player.
6 Effects and parallel evaluation

Now let’s return to the topic of parallelism: what happens when you combine effects and parallel evaluation. We defined some functions for manipulating bank accounts. These use effects in a non-trivial way, obviously! What happens if we use parallel evaluation, as in:

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

The deposit and withdrawal can be done 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’s possible (if the scheduling is sufficiently perverse) 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. It should be clear from this discussion that the possible final values for the contents of r include 50, 150, and 200. This is probably NOT what a bank (or even an account holder) would approve of!

This kind of problem, in which it is possible for two processors to access the same cell simultaneously, one or more trying to write, is called a race condition. The value produced, and the effect caused, by racy code 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; 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’s also OK to use benign or beneficial side effects in parallel code, because benignity means again that there won’t be any race conditions.
7 Self-test

1. When we introduced graph reachability earlier we annotated the `reachable` function with type information to help you to read the code. Here is it again, with most of the type annotations omitted:

   ```ml
   type 'a graph = 'a -> 'a graph;

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

   The ML type inference algorithm can figure out that the most general type of `reachable` is `''a graph -> ''a * ''a -> bool`. Here are some simple questions (about type inference) to help explain how.

   - Why does ML insist that the type variable is `''a` and not `'a`?
   - Assuming that `G` has type `''a graph` and `x` and `y` have type `''a`, and that the most general type of `exists` is

     ```ml
     (''b -> bool) -> 'b list -> bool
     ```

     what is the most general type of the function `dfs` declared by

     ```ml
     fun dfs a = (n=y) orelse (exists dfs (G n))
     ```

   - Why is this also the (most general) type of `reachable`?

   Your answers should all be based entirely on the syntactic structure of the above code!

2. Let `G` be a finite graph with `n` nodes and `e` edges. Give a big-O estimate of the work for `reachable G (x, y)`. How about the span?

3. Using the notation for evaluation of imperative expressions, let `dfs` be the function used by `reachable G (1,4)`, where `G0` is the graph given earlier, and show that
4. What happens if we execute the following code?

```ocaml
fun G 1 = [1,2]
| G _ = []

reachable G (1, 2)
```

Give an intuitive explanation first, and then write a brief more formal explanation using the notation we introduced for evaluation of imperative expressions, showing the effects as well as values. This will give you practice in using formal notation to express (accurately) the ideas behind informal intuition.

5. Suppose we represent a graph as an \textit{adjacency list}: a list of node-node pairs, each pair \((x, y)\) representing an edge from node \(x\) to node \(y\).

Define a version of the \texttt{reachable} function for graphs represented as adjacency lists.

HINT: Keep the imperative code the same and make a minimal change to the rest of the code.

If you did it right, your new version should satisfy the same spec as the original version, because that spec was agnostic about the way that graphs are being represented.
6. Consider the following version of the reachability function.

```ml
fun reachable (G:'a graph) (x:'a, y:'a) : bool = 
  let
    val visited = ref [ ]
    fun dfs (n:'a) : bool =
      (n = y) orelse
      let
        val V = !visited
        val v = if mem n V then false else exists dfs (G n)
      in
        (visited := n::V; v)
      end
  in
    dfs x
  end
```

Explain how the flow of control (and the effects) of this version differ from the original.

7. What happens if we execute the following code, using the revised version of `reachable`?

```ml
fun G 1 = [1,2]
| G _ = [ ]

reachable G (1, 2)
```

Again give an intuitive explanation first, and then write a brief more formal explanation using the notation we introduced for evaluation of imperative expressions, showing the effects as well as values.