15-150 Lectures 27 and 28: Imperative Programming

Lectures by Dan Licata

April 24 and 26, 2012

In these lectures, we will show that *imperative programming is a special case of functional programming*. We will also investigate the relationship between imperative programming and parallelism, and think about what happens when the two are combined.

1 Mutable Cells

Functional programming is all about transforming data into new data. Imperative programming is all about updating and changing data.

To get started with imperative programming, ML provides a type `'a ref` representing *mutable memory cells*. This type is equipped with:

- A constructor `ref : 'a -> 'a ref`. Evaluating `ref v` creates and returns a new memory cell containing the value `v`. Pattern-matching a cell with the pattern `ref p` gets the contents.
- An operation `:= : 'a ref * 'a -> unit` that updates the contents of a cell.

For example:

```
val account = ref 100
val (ref cur) = account
val () = account := 400
val (ref cur) = account
```

1. In the first line, evaluating `ref 100` creates a new memory cell containing the value 100, which we will write as a box:

   and binds the variable `account` to this cell.

2. The next line pattern-matches `account` as `ref cur`, which binds `cur` to the value currently in the box, in this case `100`.

3. The next line updates the box to contain the value `400`.

4. The final line reads the current value in the box, in this case `400`.

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 copies of the line `val (ref cur) = account` bind `cur` to two different numbers, depending on the value currently in the box. This is characteristic of effects such as mutation and input/output, but not something that a pure functional program will do.[1]

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

Here are some useful functions on references:

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

The first gets the value of a cell, so we can write `!r` instead of pattern-matching. The second combines a read and a write; e.g. `update (fn x => x + 1) r` is like `r++` (except it doesn't return either the before the after value).

For example:

```
fun deposit  n a = update (fn x => x + n) a
fun withdraw n a = update (fn x => x - n) a
```

1.1 Mutation and Parallelism: Race Conditions

What happens if I do

```
val account = ref 100
val () = deposit 100 account
val () = withdraw 50 account
```

In the first line, `account` is bound to a box, which has the following values

after line 1	100
after line 2	200
after line 3	150

What happens if I do

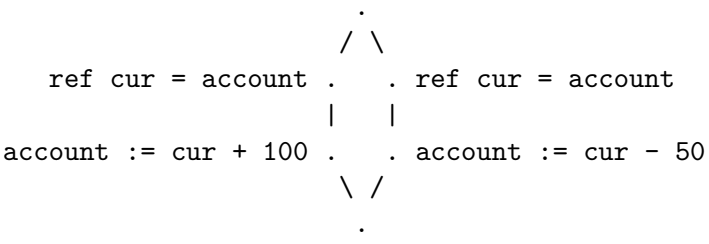
```
val account = ref 100
val _ = Seq.tabulate (fn 0 => deposit 100 account | 1 => withdraw 50 account) 2
```

The `tabulate` runs the deposit and the withdrawal in parallel. So the value in the box at the end is 150, because addition is commutative, so it doesn't matter which order I do the transactions in, right?

Wrong. The reason is that the two transactions might not happen in either sequence: they might be interleaved. In particular, the value in the box at the end might be 200, or 50.

To see this we can make a cost graph and look at the various possible schedules for two processors:

[1] Insanity is doing the same thing over and over again and expecting different results.



If the schedule does first the whole left side, then the right, then we get 150 in the box at the end. Similarly, the schedule does first the whole right side, then the left, then we also get 150. However, what happens with a breadth-first schedule? We read the *initial* value into `cur` on both sides (on the two different processors), and then write the updated values. So one write will clobber the other, and we will get either 200 or 50.

This is called a *race condition*: two processors are accessing the same memory cell at the same time, with at least one doing a write. Consequently, the meaning is dependent on the schedule.

What can we do about race conditions? One possibility is to use *locks* to force the read and write on each side to happen *atomically*, without any other processors accessing the memory. Before doing the left-side, you acquire a lock (no one else touch this for now!); after, you release the lock (okay, I'm done; go!). Programming with locks is pretty tricky, though.

Another possibility is to *avoid mutation in parallel code when possible*. That is, inasmuch as possible, you want to rely on functional programming in parallel code, because that way you inherently avoid race conditions. It is also possible to make careful use of mutation in parallel code, in such a way that you still get a schedule-independent result—see the discussion about benign effects next time.

1.2 Mutation and Verification

Exercise: prove that `1 = 2`.

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

++ r
== 1

(++ r ; ++ r)
== (r := r + 1; !r) ; ++ r
== let val _ = (r := r + 1; !r) in ++ r
== let val _ = !r in ++ r
== ++ r
== 2

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 if you run it twice; in the presence of mutation, that's not true!

Another way to diagnose the problem is that an expression by itself doesn't tell you everything you need to know to evaluate a program. The second `++ r` is really in a different memory than the first one, so it's wrong to assert that they are equal.

There is a large body of work on reasoning about imperative programs with mutation. We won't have time to get to this in 150. But this example shows that reasoning about imperative programs is a lot harder than reasoning about purely functional programs, which is why we've been focusing on pure programming all semester.

2 Persistent vs. Ephemeral Data Structures

Now that we have mutation, we can define interfaces that mimic the ones you're used to from C or Java.

The GAME signature that you're using for homework is persistent: making a move creates a new game board. This is crucial for game tree search, where you explore multiple possible moves independently and in parallel:

```
signature GAME =
sig
    type state
    type move
    val make_move : (state * move) -> state

    ...
end

fun search (depth : int) (s : Game.state) : edge =
    choose (Game.player s)
           (Seq.map
            (fn m => (m , evaluate (depth - 1) (Game.make_move (s,m))))
            (Game.moves s))
```

Here, it is important the `(Game.make_move (s,m))` creates a new state reflecting the result of the move.

What happens if we use an ephemeral interface?

```
signature EPH_GAME =
sig
    type state
    type move
    val make_move : (state * move) -> unit

    ... unchanged otherwise ...
end
```

`make_move` imperatively updates the state, changing it: it actually moves the piece around on the chessboard.

As a first cut, we can adapt MiniMax as follows:

```
fun search (depth : int) (s : Game.state) : edge =
    choose (Game.player s)
           (Seq.map
            (fn m => (m ,
                      (Game.make_move (s,m);
                       evaluate (depth - 1) s)))
            (Game.moves s))
```

That is, we first update the state by the move, and then search recursively. However, this is incorrect! The reason is that the updates we do while searching one branch of the game tree will be visible in the others. E.g. for Nim, with search depth 1, we'll see

```
    15
  /  | \
 14 12  9
```

not

```
    15
  /  | \
 14 13  12
```

because the updates second move (subtract 2) modifies the state resulting from the first move (subtract 1), rather than the original state. That is, rather than experimentally playing out future possibilities, like with the persistent board, we are actually playing out the (single) future.

By doing functional programming, you get persistent interfaces don't need to think about *undoing* your actions. Ephemeral interfaces allow a whole new class of bugs.

One way to fix this is to suppose that the game additionally provides an `undo` function:

```
val undo_last_move : state -> unit
```

that picks up a piece.

Then we can say:

```
fun search (depth : int) (s : Game.state) : edge =
    choose (Game.player s)
           (Seq.map
            (fn m => (m ,
                      (Game.make_move (s,m);
                       evaluate (depth - 1) s) before
                       Game.undo_last_move s))
            (Game.moves s))
```

That is, we play out the future, but we clean up after ourselves when we're done.

This code is correct *sequentially*—if we explore each branch of the tree in turn. So this code works if you want to live in the ephemeral/sequential world.

However, what if we try to use it in parallel? If we try to explore different branches in parallel, then they may incorrectly see each other's moves, because we have not yet undone them. It won't work. You could fix this by searching each branch of the tree atomically, without any other processors acting on the board, using locks. But then there is no parallelism left, and you're right back in the sequential case.

This makes sense, intuitively: If you want to explore multiple futures in parallel, you need multiple chess boards—i.e. a persistent data structure. If all you have is an ephemeral data structure—one chess board—you need to play out the futures sequentially.

Thus, a persistent data structure provides *more functionality* than the corresponding ephemeral one, because it provides the ability to play out multiple futures in parallel. On the other hand, ephemeral data structures allow *more implementations*, e.g. imperative ones. For example, for an ephemeral Connect 4, you could represent the board by a mutable array, so that, to make a move, you just imperatively update the one location in the array, without any copying. However, this comes at the price of having only one future.

Persistent from Ephemeral Above, we said that persistent data structures provide more functionality. I can prove to you that they provide *at least as much functionality* by writing a functor that coverts a persistent data structure into an ephemeral one. That way, any client code that was written with an ephemeral data structure in mind can be used with the (converted) persistent one.

Here's how this looks:

```
signature GAME =
sig
    type state
    type move
    ...
    val make_move : (state * move) -> state
    val player : state -> player
end

signature EPH_GAME =
sig
    type state
    type move

    val make_move : (state * move) -> unit
    val player : state -> player
end

functor EphFromPers (G : GAME) : EPH_GAME =
struct
  type state = G.state ref
  type move = G.move
  ...
  fun make_move (s,m) =
    s := G.make_move(!s,m)
```

```
  fun player s = G.player (! s)
end
```

The idea is that an ephemeral game state is a single memory cell containing a persistent game state. `next_move` updates the contents of the memory cell to be the result of making the move on the persistent game. Operations that access the state, like `player`, read the memory cell and then do the corresponding persistent operation. Thus, a data structure that has only one future is a special case of a dictionary that can have many futures.

This shows that, if you implement a persistent data structure, you can use it as an ephemeral one, so anything you can do with an ephemeral one, you can do with the persistent one. On the other hand, there are more ways to implement the ephemeral signature—e.g. for Connect 4 you could use a mutable array, where updates actually change the state. So persistent and ephemeral data structures are just different—both are good tools, but it's important to know the advantages and disadvantages of each.

2.1 Summary

Thus, we now have two independent choices for a data structure: (1) do we want to use it in sequential code or in parallel code? and (2) do we want a persistent or ephemeral interface? Note that these choices are independent: there are times when you want a persistent data structure for a sequential algorithm (e.g. for backtracking), and times when you want to an ephemeral data structure in a parallel programming (e.g. to implement an operating system). Here's a summary of what happens when you use mutation, depending on these choices:

	persistent	ephemeral
parallel	effects must be benign	need to think about concurrency
sequential	effects must be benign	OK

If you want to use effects to implement a persistent interface, then those effects must be *benign*: the code must be deterministic, in the sense that an expression has the same behavior no matter how many times it is run, or what is run in parallel with it. That is, even though the code uses effects under the good, it must behave like a functional program. Of course, one way to achieve this is to eschew effects and program functionally! But there are times when you can hide uses of mutation and still achieve a persistent interface, either sequentially or in parallel. We'll see examples below. On the other hand, if you use an ephemeral data structure, then if you want sequential execution, this is OK—you are doing traditional sequential imperative programming. But if you want parallel execution, then life gets tricky: you need to think about the interaction of concurrent processes that share state. This is not something we'll cover in this course.

Thus, the persistent `GAME` interface that we used in HW is purely functional, and therefore lands in the top-left quadrant, whereas the `EPH_GAME` interface is ephemeral, and can only be used sequentially—so it falls into the bottom-right quadrant.

3 Persistent vs. Ephemeral Dictionaries

Note: Not covered in lecture, Spring 2012. This might be worth reading if you were confused by the mutable list lab.

As a second example, the purely functional implementation of dictionaries that we've seen before satisfies what we will now call a *persistent* interface:

```
signature DICT =
sig
  structure Key : ORDERED
  type 'v dict

  val empty  : 'v dict
  val insert : 'v dict -> (Key.t * 'v) -> 'v dict
  val lookup : 'v dict -> Key.t -> 'v option
  val delete : 'v dict -> Key.t -> 'v dict
end
```

The key point is that a value of type `dict` exists for all time, and operations like `insert` create new dictionaries. This means that a value of type `dict` can have multiple futures, in which different things happen to it.

On the other hand, here is an *ephemeral* dictionary signature:

```
signature EPH_DICT =
sig
  structure Key : ORDERED
  type 'v dict

  val mk_empty  : unit -> 'v dict
  val insert : 'v dict -> (Key.t * 'v) -> unit
  val lookup : 'v dict -> Key.t -> 'v option
  val delete : 'v dict -> Key.t -> unit
end
```

Here, operations like `insert` modify a dictionary, so that a dictionary can only have one future. `empty` is replaced by `mk_empty`, a function that generates a new empty dictionary. `insert` returns `unit` (the type with exactly one inhabitant, `()`); this type tells you that we are running `insert` for its effects, modifying the dictionary, not for its value, which is trivial.

Here's an example illustrating the difference:

Persistent dictionary `Dict : DICT`:

```
val d = Dict.empty
val d1 = Dict.insert d (1,"a")
val NONE = Dict.lookup d 1
```

In this case, `d` can have many independent futures, one where you insert 1, another where you insert 2, ..., all of which have nothing to do with each other.

Ephemeral dictionary `EphDict : EPH_DICT`:

```
val d  = EphDict.mk_empty ()
val () = EphDict.insert d (1,"a")
val SOME "a" = EphDict.lookup d 1
```

In this case, `d` has exactly one future, which consists of all of the modifications that you ever make to it.

Next, we'll illustrate programming in the bottom-right quadrant, where it is OK to use mutation, by implementing ephemeral dictionaries.

3.1 Persistent Deletion

It's instructive to do ephemeral dictionaries with deletion, so let's first review how to do deletion for a persistent dictionary, implemented as a tree:

```
signature DICT =
sig
  ...
  val delete : 'v dict -> Key.t -> unit
end

functor TreeDict(Key : ORDERED) : DICT =
struct

  structure Key : ORDERED = Key

  datatype 'v tree =
      Leaf
    | Node of 'v tree * (Key.t * 'v) * 'v tree

  type 'v dict = 'v tree

  ...

  fun merge (l , r) =
      case r of
          Leaf => l
        | Node (r1 , x , r2) => Node (merge (l , r1) , x , r2)

  fun delete d k =
      case d of
          Leaf => Leaf
        | Node (l , (k',v'), r) =>
              (case Key.compare (k,k') of
                   LESS => Node (delete l k , (k',v'), r)
                 | GREATER => Node (l , (k',v'), delete r k)
                 | EQUAL => merge (l , r))

end
```

To delete a key, we walk down until we find it, deleting on the appropriate side if we don't. When we do, we're left with two trees, `l` and `r`, where everything in `l` is less than or equal to

everything in `r`, and we need to put them together. One simple way to do this is to put the entire left tree as the leftmost child of the leftmost leaf in `r` (the minimum element)—this is well-sorted, because everything in `l` is less than the minimum element of `r`. (There are fancier ways of doing this that preserve balance.) `merge` walks left in `r` and returns `l` at the leaf.

3.2 Persistent Data Structures are More General

The first thing I'd like to do is to prove to you that persistent data structures are more general. I can do this with the following piece of code, which creates an ephemeral dictionary from a persistent one:

```
functor EphFromPers (PersDict : DICT) : EPH_DICT =
struct
  structure Key : ORDERED = PersDict.Key

  datatype 'v eph_dict = Dict of 'v PersDict.dict ref
  type 'v dict = 'v eph_dict

  fun mk_empty () = Dict (ref (PersDict.empty))

  fun insert (Dict (d as ref t)) (k, v) =
      (d := PersDict.insert t (k, v))

  fun lookup (Dict (d as ref t)) k =
      PersDict.lookup t k

  fun delete (Dict (d as ref t)) k =
      (d := PersDict.delete t k)
end
```

The idea is that an ephemeral dictionary is a single memory cell containing a persistent dictionary. `mk_empty` creates a new cell containing the empty dictionary. `insert` and `lookup` read the contents of the memory cell, perform the corresponding operation on the persistent dictionary, and write the result back. `lookup` reads the memory cell and does a persistent lookup.

Thus, a dictionary that has only one future is a special case of a dictionary that can have many futures.

Note that you cannot implement an ephemeral dictionary directly as a persistent dictionary, as opposed to a memory cell containing one, because the ephemeral interface does not allow you to return a new dictionary from `insert` and `lookup`.

The Value Restriction Why do we need `mk_empty : unit -> 'v dict` rather than simply `empty : 'v dict`? We could try to define

```
val empty : 'v dict = Dict(ref PersDict.empty)
```

But there are a couple of problems with this. First, recall that, when you evaluate a module, you run each declaration in it. So, when you run

```
structure E = EphFromPers(TreeDict(IntLt))
```

you run the declarations in the body of the module. Running a function declaration doesn't do anything, because functions are values, but to run a `val` declaration, you evaluate the RHS, which means you do any effects. Thus, when evaluating `E`'s RHS, you run `empty`'s RHS, and evaluate the `ref PersDict.empty`, which is what creates a new cell in memory. Thus, `E` only has *one* empty dictionary, and all `insert`s will modify this one dictionary. This probably isn't what you want: in client code, you will likely want to make many independent dictionaries with the same type of key.

However, even if you only wanted one shared dictionary, the above code is problematic because `empty` is polymorphic. Recall that a declaration

```
val empty : 'v dict
```

means "for any type `'v`, `empty` has type `'v dict`." So suppose we have made this declaration. Thus, `empty : int dict` and `empty : string dict`. But if `empty` stands for one memory cell, this means that we choose a key (say the number `1`), insert a mapping from that key to an integer into the one shared dictionary, and then look it up, in that one shared dictionary, as a string. Since the lookup will actually return the integer, this will allow an integer to be used as a string, subverting the type system and making your program crash! More concretely:

```
val d : int E.dict = E.empty
val () = E.insert d (1,2)

val d' : string E.dict = E.empty
val (SOME 2)  : string option = E.lookup d' 1
```

`d'` and `d` are actually the same memory cell, so when you lookup `1` in `d'`, it will return the value it is associated with in `d`. This allows you to return `2` in a position that has type `string`, which will crash.

For this reason, ML cannot allow such a declaration. ML prevents this using something called the *value restriction*: only values are given polymorphic types. This includes functions, and datatype constructors (like `Empty` in the persistent case), but not non-value expressions like `ref`, which can have effects. So the above declaration is not even allowed, preventing this bug.

That's why we need to replace `empty` with `mk_empty` in the ephemeral signature: `mk_empty` delays the effect of creating a new memory cell inside the function body, so that it gets run every time you call the function. Two different applications `mk_empty()` make two different dictionaries, so it's okay that `mk_empty() : int dict` and `mk_empty() : string dict`, because updates to these two dictionaries will not be shared. Thus, you can't subvert the type system, and you can make multiple independent dictionaries using the same structure.

3.3 Refs All The Way Down

You can also implement an ephemeral dictionary the hard way, with references for the children of each node (Figure 1).

A `eph_dict` is again a memory cell, this time containing a `dict_contents`. A `dict_contents` is either empty or a node, and the children of a node are again `eph_dict`s. This means that each node has an updatable memory cell for each child. Consequently, operations on dictionaries can

```
functor EphDictHard (Key : ORDERED) : EPH_DICT =
struct
  structure Key : ORDERED = Key

  datatype 'v eph_dict = Dict of 'v dict_contents ref
       and 'v dict_contents = Empty
                            | Node of 'v eph_dict * (Key.t * 'v) * 'v eph_dict
  type 'v dict = 'v eph_dict

  fun mk_empty () = Dict (ref Empty)

  fun insert (Dict t) (k, v) =
      case t of
  ref Empty => t := Node (mk_empty (), (k, v), mk_empty ())
| ref (Node (l, (k', v'), r)) =>
  case Key.compare (k, k') of
      LESS => insert l (k, v)
    | EQUAL => t := Node (l, (k, v), r)
    | GREATER => insert r (k, v)

  fun lookup (Dict t) k =
      case t of
  ref Empty => NONE
| ref (Node (l, (k', v'), r)) =>
  case Key.compare (k, k') of
      LESS => lookup l k
    | EQUAL => SOME v'
    | GREATER => lookup r k

  (* merge left into the right *)
  fun merge (l as Dict (ref lctnts) : 'v eph_dict) ((Dict r) : 'v eph_dict) =
      case r of
          ref Empty => r := lctnts
        | ref (Node (rl , x , rr)) => merge l rl

  fun delete (Dict t) k =
      case t of
          ref Empty => ()
        | ref (Node (l , (k',v') , r)) =>
              (case Key.compare (k,k') of
                   LESS => delete l k
                 | GREATER => delete r k
                 | EQUAL => let val () = merge l r
                                val Dict (ref rctnts) = r
                                val () = t := rctnts
                            in () end)
end
```


Figure 1: Ephemeral dictionary using references at each node

walk down and update the relevant part of the tree in place, with no need to reconstruct the tree afterward. (I.e. you can write C code (only prettier!) in ML).

Let's look at each function:

For `insert`, when we reach a leaf, we create a new node, and *assign* the input cell `t` to contain this new node—rather than returning the new node as the answer. Similarly, when we find the key, we update that cell in place. When we don't find it, we recursively insert left or right—but note that, unlike in functional code, there is no need to reconstruct the tree after the recursive call. Note the use of `ref` patterns nested with other patterns, like `ref Leaf` and `ref(Node (...))` to explore the contents of the cell.

`lookup` is the same as the persistent version from previous lectures, except it uses `ref` patterns to peek inside of memory cells.

`delete` has the same overall structure as the above persistent version. However, there are differences: when we reach a leaf without finding the key, we simply return the value of unit type, `()`—there is no work to do!—rather than returning a `Leaf`. Next, we walk down to where the key should be, and, like with `insert`, we can inductively assume that the recursive calls make the appropriate modifications, and there is no need to reconstruct the tree. When we find the key, our strategy is to `merge` the left tree into the right, which will mutate the right cell so that it contains the left tree at the appropriate leaf. This will update the memory cell `r`. Next, we read the updated value out of `r` and assign it into the overall cell `t`.

To `merge` into a cell containing a leaf, we update the cell to contain the contents of `l`. Otherwise, we walk leftwards.

4 Benign Effects

Sometimes you can use mutation under the hood to implement a persistent interface—from the outside, the code behaves like a functional program. This is called a *benign effect*. Often, the use of mutation to implement benign effects is not safe for parallelism—the code behaves functionally to sequential, but not parallel, observers—so you land in the bottom-left quadrant. However, there are times when mutation can be used to implement a persistent interface in a way that is safe for parallelism, in which case a little mutation sneaks into the top-left quadrant.

To illustrate this, we'll use graph reachability.

First, how do we represent a graph? For today, all we need is that given a graph, we can compute the set of successor nodes of any node.

```
signature GRAPH =
sig
    type graph
    structure NodeSet : SET
    type node = NodeSet.El.t
    val successors : graph -> node -> NodeSet.set
end
```

Here's the set interface we'll use; you saw some of this in lab a while back:

```
signature SET =
sig
```

```
  structure El : ORDERED

  type set

  val empty  : set
  val insert : set -> El.t -> set
  val member : set -> El.t -> bool

  datatype lview = Nil | Cons of El.t * set
  val show : set -> lview

  val mapreduce : (El.t -> 'b) -> 'b -> ('b * 'b -> 'b) -> set -> 'b

  val exists : (El.t -> bool) -> set -> bool
  val all : (El.t -> bool) -> set -> bool
  val fromList : El.t list -> set
end

functor Set(E : ORDERED) : SET = ...
```

For example, a really simple way to represent a graph is as the successor function:

```
structure IntGraph : GRAPH =
struct
    structure NodeSet = Set (IntLt)
    type graph = int -> NodeSet.set
    type node = NodeSet.El.t
    fun successors f x = f x
end
```

For example, here's a graph:

```
val testg = fn 1 => fromlist [2,3]
             | 2 => fromlist [1,3]
             | 3 => fromlist [4]
             | 4 => fromlist []
```

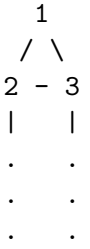
The reachability problem for a graph `G` is to give a function `reachable`, which, given any two nodes in a graph, tells you whether or not there is a path from the first (the start node) to the second (the goal):

```
signature REACHABILITY =
sig
    structure G : GRAPH
    val reachable : G.graph -> G.node -> G.node -> bool
end
```

All semester, we've been posing problems as types, and solutions to them as programs of that type. Now, we're starting to lift that up to the module level: a problem is a signature, and a solution is a module implementing that signature.

The idea for reachability is to walk down the graph looking for the goal node. For this to terminate, we need to keep track of which nodes we've seen before: otherwise, if we end up on a cycle that doesn't contain the goal node, we'll walk around it forever.

In fact, we can do this in time linear in the number of nodes + edges if we share the same visited list across all calls. For example, for the graph



we want to remember all of the nodes we visited while searching from `2` when we search from `3`, so we don't redo all of that work. Consequently, the same visited list flows "down" the graph as we search, but also comes back "up" so that siblings see the modifications to it. This flow of information is characteristic of mutable state.

4.1 Benign in Sequential Contexts

Here's a first implementation of reachability:

```
functor ReachBenignSeq (G : GRAPH) : REACHABILITY =
struct
    structure G = G
    structure Visited = Set(G.NodeSet.El)

    val visited : Visited.set ref = ref Visited.empty

    fun reachable g start goal =
        let
          val () = visited := Visited.empty

          fun dfs (cur : G.node) : bool =
              case G.NodeSet.El.compare (cur, goal) of
                  EQUAL => true
                | _ => case Visited.member (!visited) cur of
                       true => false
                     | false => (visited := (Visited.insert (! visited) cur);
                                 (G.NodeSet.exists dfs (G.successors g cur)))
        in
            dfs start
        end
```

```
end
```

We create a memory cell containing the visited list. At the beginning of each call to `reachable`, we clear the visited list, and then run a loop over nodes starting with `start`. For each node we visit, we check if it's the goal node `goal`, and if so, succeed. Otherwise, we read the current value of the visited set, and check whether we've already been to `cur`. If so, we fail—we just walked around a cycle without finding the goal. Otherwise, we update the visited set to include the current node, then then check whether `goal` is reachable from some successor of `cur` (using the `exists` function on `sets`). The expression `e1 ; e2` evaluates `e1` for its effects and then evaluates and returns the value of `e2`—it's like `let val () = e1 in e2 end`

Benignity. From the outside, `reachable` behaves like a functional program: for a fixed graph `g` and nodes `x` and `y`, any two calls to `reachable g x y` will return the same value, and none of the effects of running `reachable` can be observed. From the outside, it is as if `reachable` doesn't have any effects at all. The reason no effects can be observed is (1) the memory cell `visited` is not exported from the module, so no code besides `reachable` can access it, and (2) the function clears the visited list at the beginning, so one call to `reachable` will not influence the value of another.

At least, not if the calls are made sequentially! If the calls are made sequentially, then whatever updates I make to the visited list are thrown away as you soon as you start. However, if the calls are made in parallel

```
Seq.map (fn g => reachable g 1 3) (seq [g1,g2,g3,...])
```

then you might clear the visited list while I'm working, in which case things will go wrong.

Thus, this implementation is *benign in sequential contexts*: if everything is run sequentially, then it has deterministic behavior and does not have any observable effects. In general, if a piece of code is independent of the initial value of a memory cell (which it is here, because it clears it), and no one else can observe the updates it makes to that cell (which is true here, because *all* functions that use the memory cell ignore its initial value), then that use of mutation will be benign in sequential contexts.

Note that it is not possible to exploit parallelism inside `dfs`—i.e. we are assuming that `NodeSet.exists` is sequential. There are two reasons this doesn't work: (1) Two processors may try to update the visited list in parallel, which is a race condition. This could result in a node being forgotten from the visited set (if it was added in between when one processor reads `visited` and when it writes the result back). Moreover, it could crash the program, because there is no reason to believe that the cell updates `visited := (Visited.insert curVisited x)` happen atomically—i.e. two processors could interfere with each other, resulting in `visited` containing half of the pointer that internally represents one set and half of another. (2) We could fix this using locks, but, even then, we might explore a node more than once, if two processors decide to start exploring it (because it's not in the visited set) simultaneously, before either marks it as visited. I.e. the naïve parallel version would do more work overall than the sequential version, so it is not clear that the parallelism is a win.

4.2 Benign in Parallel Contexts

It is easy to change the above code so that multiple copies of reachability can be run in parallel (though we still do not use parallelism inside the implementation of `reachable`, for the reasons just

discussed). The idea is that, instead of having one visited list for all calls to reachable, we generate a new visited list for each call:

```
functor ReachBenignPar (G : GRAPH) : REACHABILITY =
struct
    structure G = G
    structure Visited = Set(G.NodeSet.El)

    fun reachable g start goal =
     let
       val visited = ref Visited.empty

       fun dfs (cur : G.node) : bool =
           case G.NodeSet.El.compare (cur, goal) of
               EQUAL => true
             | _ => case Visited.member (!visited) cur of
                    true => false
                  | false => (visited := (Visited.insert (! visited) cur);
                              (G.NodeSet.exists dfs (G.successors g cur)))
     in
         dfs start
     end
end
```

The change is simply to create a new visited list by running `ref Visited.empty` inside of each call to `reachable`.

This use of mutation is *benign in parallel contexts*: it has a deterministic result and no observable effects, even if its run in parallel with any other code. The reason is that different calls to `reachable` can no longer interfere through shared state. In general, if a piece of code allocates a memory cell, and ensures that the cell doesn't *escape* (either in the return value, or by being stored into some other memory cell), then that use of mutation will be benign in parallel contexts.

4.3 Store passing

If "benign effect" means "equivalent to a functional program", then it stands to reason that we should be able to implement reachability without using mutation. Here's how: we pass the visited list along as an extra argument, and return the visited list as an extra result:

```
functor ReachExplicitSP (G : GRAPH) : REACHABILITY =
struct
    structure G = G
    structure NodeSet = G.NodeSet
    structure Node = G.NodeSet.El
    structure Visited = Set(Node)

    type dfsresult = bool * Visited.set (* nodes you've visited so far *)
```

```
    fun reachable g start goal =
        let
            fun dfs (cur : Node.t) (visited : Visited.set) : dfsresult =
                case Node.compare (cur, goal) of
                    EQUAL => (true , visited)
                  | _ => (case Visited.member visited cur of
                              true => (false , visited)
                            | false => dfs_children (G.successors g cur) (Visited.insert visit
            and dfs_children (cs : NodeSet.set) (visited : Visited.set) : dfsresult =
                case NodeSet.show cs of
                    NodeSet.Nil => (false , visited)
                  | NodeSet.Cons(c1, cs) =>
                        case dfs c1 visited of
                            (true , visited) => (true, visited)
                          | (false , visited) => dfs_children cs visited

            val (b , _) = dfs start Visited.empty
        in
            b
        end
end
```

`dfs` takes a `Visited.set` as an extra argument, and returns it in addition to the `bool`. In each base case, we return the current visited list in addition to `true`/`false`. In place of `Set.exists`, we write a function `dfs_children` that checks each child in turn. We cannot use `exists` because we need to pass and return the visited list—in the imperative version, this was happening behind the scenes. When `dfs` calls `dfs_children`, it first "updates" the visited list—i.e. it passes the updated list. When `dfs_children` calls `dfs` to search one node `c1`, this produces a new visited list, which is used to explore the remaining children `cs`. This way, the updates made while exploring one child are visible in its siblings. At the end of the day, we run `dfs` on the empty visited list, and return the boolean result, ignoring the final visited list.

Because `dfs` calls `dfs_children` and vice verse, these two functions are said to be *mutually recursive*. To define mutually recursive functions, you use `and` in place of the second (and third, ...) `fun` binding.

4.4 Store-Passing in Monadic Style

One knock against the functional version, as compared to the imperative version, is that the code is longer and harder to read, because the "plumbing" of passing the visited list along is explicit. We can solve this problem, and write a purely functional program that is line-by-line equivalent to the imperative version, by abstracting out the process of passing the visited list along.

4.4.1 Monads

To do this, we'll use the following signature:

```
signature STATE_MONAD =
sig
    type state
    type 'a comp = state -> 'a * state
    val return : 'a -> 'a comp
    val >>= : 'a comp * ('a -> 'b comp) -> 'b comp
    val get  : state comp
    val set  : state -> unit comp
    val run  : 'a comp -> state -> 'a
end
```

The type `state` represents the state that is passed along; below, we will choose it to be a `Visited.set`. The type `'a comp` represents a computation that can "read from" and "write to" this state. Such a computation is modeled by a function that takes an argument of type `state` and returns both a value of type `'a` and a new `state`.

We will use the following operations on this type: `return` says that if you have a value of type `'a` that doesn't read or write the state, you can make a (trivial) computation. `>>=` (pronounced `bind`) says that you can sequence two computations. A special case is the `;` of imperative programming:

```
; : unit comp -> unit comp -> unit comp
```

which runs two computations in sequence. Here, a computation can return a result, and `>>=` feeds the result of the first computation as input to the second. In general, the word *monad* is used to describe a type of computations with `return` and `>>=`. I.e. we can define a monad type class:

```
signature MONAD =
sig
    type 'a comp
    val return : 'a -> 'a comp
    val >>= : 'a comp * ('a -> 'b comp) -> 'b comp
end
```

The implementation of `STATE_MONAD` will be a `MONAD`, with extra types and operations for working with the state.

`get` makes a computation that returns the current state. `set` makes a computation that updates the state to a new value. `run` runs a computation on a given initial state, returning its value and discarding the final state.

4.4.2 Reachability, Monadically

This is all kind of abstract, so let's make it concrete by showing how to implement reachability:

```
functor ReachMonadicSP (G : GRAPH) : REACHABILITY =
struct
    structure G = G
    structure Visited = Set(G.NodeSet.El)

    structure SP = StorePassing (struct type state = Visited.set end)
    structure SPUtils = SetMonadUtils (struct structure S = G.NodeSet structure T = SP end)
```

```
    open SP
    infix 7 >>=

    fun reachable g start goal =
        let
            fun dfs (cur : G.node) : bool comp =
                case G.NodeSet.El.compare (cur, goal) of
                    EQUAL => return true
                  | _ => get >>= (fn curVisited =>
                         case Visited.member curVisited cur of
                              true => return false
                            | false => set (Visited.insert curVisited cur) >>= (fn _ =>
                                       (SPUtils.existsM dfs (G.successors g cur))))
        in
            run (dfs start) Visited.empty
        end
end
```

This code matches up line for line with the second imperative version given above (the one that is benign in all contexts), but is simply a more abstract version of the functional store-passing implementation.

First, we name some structures (`NodeSet`, `Node`) for convenience, and apply the `Set` functor to create a visited set. Next, we apply the `StorePassing` functor to create a `STATE_MONAD` whose state is `Visited.set`. `SPUtils` will be described below.

The keyword `open M` makes all of the declarations in `M` available in the current scope, so we don't need to use qualified names like `SP.comp` and `SP.>>=` for components of `SP`.

`dfs` produces a `bool comp`, which is shorthand for saying that it takes a `Visited.set` as an argument and returns one as a result, as above. If we've found the goal, we `return true`. If not, we `get` the current state, and name it `curVisited`. If we've seen `x` before, then we `return false`. Otherwise we update the state to be `Visited.insert curVisited x` and check whether `y` is reachable from some successor node.

The `SetMonadUtils` functor creates a function

```
val existsM : (El.t -> bool comp) -> set -> bool comp
```

that is just like the usual `Set.exists` except that it passes the state along, and each call to the predicate may read/write it. The implementation is a parametrized version of the function `dfs_children` above. (In the imperative version, we were able to use `Set.exists` for this purpose, which worked because the updates to the state happened behind the scenes, using actual mutable memory cells.) In fact, `SetMonadUtils` implements `existsM` for any `MONAD`, not just the `STATE_MONAD`.

In the end, we `run` the computation `dfs x` on the initial state `Visited.empty`. This corresponds to (1) allocating the memory cell `visited` in the imperative version, and (2) throwing it away when the call to `reachable` is over. If the state persisted across calls, like in the first version above, we would not be able to "escape" the monad here using `run`—because once we have escaped the monad, the code can make no more use of the state.

4.4.3 Implementation of Store Passing

The implementation of each store passing operation is simple, but it's a nice use of higher-order functions:

```
functor StorePassing(A : sig type state end) : STATE_MONAD =
struct
 type state = A.state
 type 'a comp = state -> 'a * state
 fun return x = fn s => (x,s)
 fun >>= (c : state -> 'a * state, f : 'a -> (state -> 'b * state))
   : state -> 'b * state =
     fn s => let val (v,s) = c s in f v s end
 val get : state -> state * state = fn s => (s , s)
 fun set (news : state) : state -> unit * state = fn _ => (() , news)
 fun run f s = case (f s) of (v , _) => v
end
```

`return` gives back the designated value, leaving the given state unchanged. Bind runs the two computations in sequence, passing the value and state resulting from the first to the second. `get` returns the current state, and leaves it unchanged. `set news` returns `()`, ignores the input state, and returns `news` as output. `run` runs the computation and ignores the final state.

Overall, what we have seen is that we can *simulate* effects using pure functional programming, in such a way that the code looks just like the imperative code. Many other effects, such as exceptions and input/output, can be modeled in the same way.