



Lesson 19

IMPERATIVE PROGRAMMING

July 27, 2023

- 1 Mutability
- 2 Reference Cells
- 3 Using Refs
- 4 Aliasing
- 5 Applications of Mutability (Bonus)

1 - Mutability

So far in this course, we've spent a great deal of time emphasizing the importance of safety in programming.

We want to avoid footguns like mutability, which quickly leads us into needing to reason profusely about the entire history of our program, as well as introducing the possibility of messing ourselves up by causing code which worked previously to no longer work.

Our solution was to simply not to play, by only dealing with **pure** code which eschewed side effects for predictable, safe behavior.

We've tried our best, but effects are kind of hard to get away from!

For instance, we saw in an earlier lecture that the presence of exceptions makes addition no longer commutative, in general, for arbitrary expressions. We would like to be able to say that independent computations can be freely reordered, but we can't reorder these two declarations of x and y :

```
fun foo () =  
  let  
    val x = loop ()  
    val y = raise Div  
  in  
    ()  
  end
```

Relatedly, our notion of extensional equivalence is not necessarily preserved, either.

We want to say that extensionally equivalent values can be freely substituted for each other wherever we see them. Unfortunately, there also exists the `print` function, which has type `string -> unit`, which prints a string to the outside world.

We can't necessarily say that `print "hi"` and `()` are extensionally equivalent, because replacing all instances of `()` will definitely make a program with differing extensional behavior. So we need to update our definition of extensional equivalence, in the presence of side effects.

Life is cruel. Unfortunately, we live in the physical world.

We've shyed away from it thus far, but at some point we have to be able to read from files, which is a side effect of its own. At some point, we need to be able to interface with the real world. We can't get away from reality forever.

So what can we do, while maintaining our ability to *generally* write safe code, and avoid the footguns of the real world?

The key term here will be that we want to *avoid* footguns.

Like many other concepts in programming, immutability and purity are concepts which serve our purposes – we do not serve theirs. That means that we do not need to bend over backwards to achieve immutability and purity, if the alternative is genuinely more useful in a particular circumstance.¹

That being said, we still don't want to use mutable code all over the place, but we want to simply offer the *option* to. The key behind immutability is not to forbid mutability, but to make mutability *opt-in*.

¹Worth noting that there *are* programming languages which do achieve *complete* purity, like Haskell, and can still do things like interface with the real world, paradoxically. But it's kind of complex to understand how it does so.

We've seen this in several different contexts this semester. For instance, consider the phenomenon of **null pointers**, which are when values in other programming languages can always possibly be some unsafe `NULL` value.

This is commonly known as Tony Hoare's "**billion-dollar mistake**"²

Our solution to this problem has been to make **optional** values intentional, by requiring them to be explicitly used when some **NONE** case is necessary. In other words, **optional** values are *opt-in*, in that they only show up when they are chosen to.

We are going to take a similar approach with mutability.

²His words, not mine.

2 - Reference Cells

Similarly to how the type `t option` is the type of values of type `t`, plus the possibility of not having such a value at all, we are going to have a brand new type which denotes **mutable storage** of values of a certain type.

Def The type `t ref`, for any type `t`, is the type of mutable references to values of type `t`.

In other words, the type `t ref` is the type of *boxes*, which contain a single value of type `t`.



The key thing is that the contents of the box are allowed to change, as the program evolves!



For anyone familiar with imperative programming, this is just a pointer to some mutable state. What's the difference?

The key innovation here is that mutability is *opt-in*, since we don't *need* to use `ref` types unless we want to. In other words, we now have a **type-level distinction** between immutability and mutability. If we have a bug with some mutability going on in our code, all we need to do is look for the explicit places where `refs` are used, rather than literally every assignment ever, as in an imperative language.

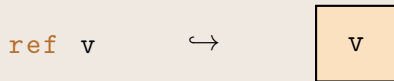
This is just another example of how types will guide the structure of our programs.

There are a few primitives that we will use to manipulate values of `ref` type. These are the `ref`, `!`, and `:=` operators.

```
val ref : 'a -> a ref
val !    : 'a ref -> 'a
val :=   : 'a ref * 'a -> unit
```

These primitives are used for creation, modification, and access for `ref` boxes.

The `ref` function takes in a value and puts it into a mutable box.



Key Fact The value denoted by `ref v` is the box itself, not the contents of the box!

In terms of what SML is doing, the act of calling the `ref` function allocates a single box, capable of storing a single value of type τ , for some type τ .

Note A `ref` box can never be empty! Because it is given an initial value to store inside of it, if the `ref` function is called at all, it is initially full, and there is no way to remove what's inside. No "null".

It's also worth noting that `ref` creation is something which occurs **once per call to `ref`**.

Each box created by a different call to **`ref`** is unique, and does not share the same space. This means that if you want storage that is separate from pre-existing `refs`, you should call `ref` again, rather than reuse existing ones.

So for instance, for the following code:

```
val r1 = ref 1
val r2 = ref "5"
val r3 = ref 0
```

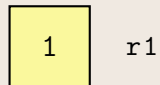
THE STORE

```
val r1 = ref 1
val r2 = ref "5"
val r3 = ref 0
```



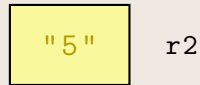
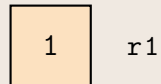
```
val r1 = ref 1  
val r2 = ref "5"  
val r3 = ref 0
```

THE STORE



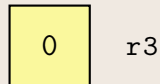
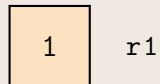
```
val r1 = ref 1
val r2 = ref "5"
val r3 = ref 0
```

THE STORE



```
val r1 = ref 1
val r2 = ref "5"
val r3 = ref 0
```

THE STORE



What if we want to change the contents of the box, though? That's where the `:=` operator³ comes in, which is an infix operator of type `'a ref * 'a -> unit`.

For instance, take the following code, in the context of the previous three boxes:

```
val () = v1 := 1
val () = v2 := "1"
```

Note how the `:=` operator returns a `()`, because it is strictly used for its side effects, and computes no meaningful values.

³Which can be pronounced "walrus", "assignment", or "colon equals".

```
val () = v1 := 2  
val () = v2 := "1"
```

THE STORE

1 r1

"5" r2

0 r3

```
val () = v1 := 2  
val () = v2 := "1"
```

THE STORE

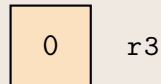
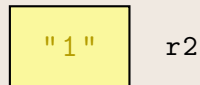
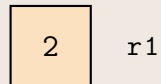
2 r1

"5" r2

0 r3

```
val () = v1 := 2  
val () = v2 := "1"
```

THE STORE



When we have boxes, it seems like we should be able to do anything!

However, if our code had contained the following line, then our code would never have run, because it would not have type-checked:

```
val () = v2 := 1
```

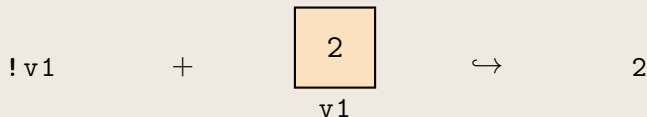
This is because `v2` is of type `string ref`, and `:=` is `'a ref * 'a -> unit`, so it doesn't type-check!

In principle, this is because when we make a box, it is only able to store values of a certain type. So we don't have complete freedom, here.

Finally, we have discussed how to make boxes, and how to put stuff into boxes. Now we need to discuss how to take things out.

This is achieved via the `!` operator, pronounced as the "bang" operator, of type `'a ref -> 'a`, which simply returns whatever value is currently in the box.

So in the current example:



We simply unpack the contents of the box `v1`, which is 2.

Note that we can also do this ! access by pattern matching.

For instance, for the refs `v1 : int ref` and `v2 : string ref` that we have been working with, we could write:

```
case (v1, v2) of
  (ref 2, ref "1") => true
| _ => false
```

which would successfully return `true`. We might say that this kind of pattern matching includes an *implicit deference*, because we access the contents of each `ref` without needing to call `!` explicitly.

Note that this works because `ref` is not just a function of type `'a -> 'a ref`, it is also a constructor!

So far, we've been using `val` bindings, which don't actually bind anything interesting, but merely serve to evaluate some expression for side-effecting purposes.

This might look something like this, for instance:

```
let
  val () = r := 150
in
  e
end
```

if we wanted to set some `ref` to the value 150, prior to evaluating some expression `e`.

This is a lot to type out, though! Luckily, there's a shorthand.

We can use the `;`, which is the **sequencing operator**, to evaluate some expressions in a sequence, while disregarding their value.

So for instance, for the previous example, we could instead write out the expression `(r := 150; e)`. This evaluates both expressions from left-to-right to values, but only returns the value of the last entry. This would be extensionally equivalent to the previous expression.

We can also nest these arbitrarily deep. So we could write `(r := 1; r := 5; r := 0; 150)`⁴ for the expression which cannot make up its mind and constantly reassigns the ref `r`, and then eventually reduces to the value 150.

⁴Generally, you need parentheses around the whole thing, whenever you have a sequence of expressions.

3 - Using Refs

Now that we know the three fundamental operators for working with `ref` cells, we can start looking at some examples of us actually using `refs`.

Now we can define some functions that we previously also could, but now we can do it with `refs`. Let's go back to our roots. Here's `fact`.⁵

```
val store = ref 1

fun fact 0 = !store
  | fact n =
    ( store := n * (!store);
      fact (n - 1)
    )
```

Seems legit, right?

⁵Disclaimer that this is, of course, strictly educational, and in practice a terrible idea. You should never introduce mutability for the sake of mutability.

At least, it would be, if we didn't end up in the entirely predictable circumstance where writing imperative code ended up causing a bug.

This `fact` implementation is wrong. Consider what happens when you call `fact 2`.

```
fun fact 0 = !store
| fact n =
  ( store := n * (!store);
    fact (n - 1)
  )
```

THE STORE



First, we enter the call to `fact 2`.

```
fun fact 0 = !store
  | fact n =
    ( store := n * (!store);
      fact (n - 1)
    )
```

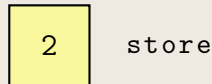
THE STORE



The first thing to happen is that we assign `store` to the contents of `store`, multiplied by 2, which is our current value of `n`.

```
fun fact 0 = !store
| fact n =
  ( store := n * (!store);
    fact (n - 1)
  )
```

THE STORE



The next thing we do is that we recurse on $n - 1$, which in this case is 1. We won't step through that call explicitly, but suffice to say that it will multiply the store by 1, keeping it the same, and then eventually return the contents of the store, which is 2.

```
fun fact 0 = !store
  | fact n =
    ( store := n * (!store);
      fact (n - 1)
    )
```

THE STORE



But what happens if we immediately then execute `fact 2`, again?

Then, we enter the function again...

```
fun fact 0 = !store
  | fact n =
    ( store := n * (!store);
      fact (n - 1)
    )
```

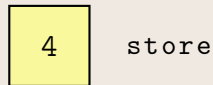
THE STORE



... and multiply what is in the `store` by `n`, which is 2

```
fun fact 0 = !store
| fact n =
  ( store := n * (!store);
    fact (n - 1)
  )
```

THE STORE



Then, we recurse finally on `fact 1`, which we know will keep the store the same and return the contents of the store, which is 4.

So `fact 2` \hookrightarrow 4.

Uh oh.

```
fun fact 0 = !store
  | fact n =
    ( store := n * (!store);
      fact (n - 1)
    )
```

THE STORE



Our problem was that our `ref` was too long lasting!

We used the same `ref`, which was *never reset*, at any point.

In fact, every call to `fact` uses that same `ref`, because we only called `ref` once, and at the top level, independent of any call to the function `fact`! This is sure to be a recipe for disaster!

A better way to do it would be not to share this memory between different function calls. We might prefer to have each call to `fact` spawn its own, private `ref` cell, for use in its calculations.

Let's rewrite our imperative `fact` with that idea in mind.

```
fun fact n =  
  let  
    val store = ref 1  
    fun fact' 0 = !store  
      | fact' n =  
          ( store := n * (!store);  
            fact' (n - 1)  
          )  
  in  
    fact' n  
  end
```



Now, our `fact` function works, because upon entering the function, a `ref` is allocated. Because it's gated by the function, it's guaranteed to be new on each invocation of the function, and thus it is safe to be used by the helper function `fact'`.

In the background, once the `fact` function finishes its run, the ref cell of `store` will be deallocated, and thus not waste any memory.

Def We say that functions like `fact` are **observationally pure**, in that they appear to an outside user to be pure, even though they use side effects such as mutability internally.

The key reason why observational purity is OK is because *it is impossible to tell from the outside* that the function is not pure! We also call such effects used in an observationally pure way a **benign effect**.

4 - Aliasing

Because of the fact that `ref` cells are values like any other, we can pass them around and bind them to new variables, like any other.

We have to be especially careful when we do something like this, however, so that we get the correct mental picture for what's happening!

For instance, take the following code:

```
val r1 : int ref      = ref 0
val r2 : int ref      = r1
val r3 : int ref ref  = ref r1
val r1 : int ref      = ref 1
val ()                = r2 := 3
```

What's going on here?

We start off with the empty store, as usual.

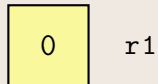
```
val r1 : int ref      = ref 0
val r2 : int ref      = r1
val r3 : int ref ref  = ref r1
val r1 : int ref      = ref 1
val ()                = r2 := 3
```

THE STORE

At a call to `ref`, we allocate a distinct `ref` cell for `r1`.

```
val r1 : int ref      = ref 0
val r2 : int ref      = r1
val r3 : int ref ref  = ref r1
val r1 : int ref      = ref 1
val ()                = r2 := 3
```

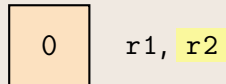
THE STORE



When we bind the value of `r1` to `r2`, this means that `r2` now refers to the *same box* as `r1`.

```
val r1 : int ref      = ref 0
val r2 : int ref      = r1
val r3 : int ref ref  = ref r1
val r1 : int ref      = ref 1
val ()                = r2 := 3
```

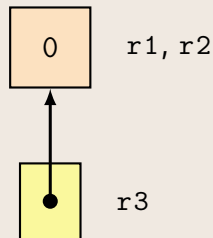
THE STORE



We then allocate a box which itself *points* to the box of `r1`, which we denote via a box with an arrow directed at the box of `r1`.

```
val r1 : int ref      = ref 0
val r2 : int ref      = r1
val r3 : int ref ref  = ref r1
val r1 : int ref      = ref 1
val ()                = r2 := 3
```

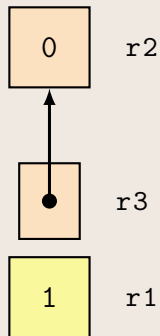
THE STORE



We then re-bind `r1` to a different `ref` cell. This affects neither `r2`, which has the same value, nor what `r3` is pointing at!

```
val r1 : int ref      = ref 0
val r2 : int ref      = r1
val r3 : int ref ref  = ref r1
val r1 : int ref      = ref 1
val ()                = r2 := 3
```

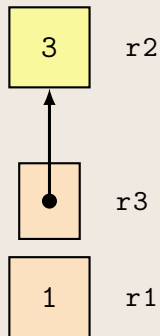
THE STORE



We can then update the contents of `r2`'s box to 3, which is independent of `r1`'s new box, but affects `r3` indirectly.

```
val r1 : int ref      = ref 0
val r2 : int ref      = r1
val r3 : int ref ref  = ref r1
val r1 : int ref      = ref 1
val ()                = r2 := 3
```

THE STORE



Such is the pointer-chasing logic common to imperative programming.

The diagram is made easier to understand if you cognize the fact that, since there are three calls to `ref`, there must be three ref cells allocated over the course of the trace.

In addition, it is impossible to change the contents of a box without a call to `:=!` So be on the lookout for cheap ways to verify your thinking, when thinking about `refs` and pointers.

Remember, there's nothing special about boxes. They can be bound to other variables at will.

Note One might say, *boxes are values*.⁶

⁶And pointers are boxes. Let transitivity take it from here.

As we saw in the previous example, we can have `refs` which point to other `refs`.

In conjunction with recursive types, this lets us define arbitrary powerful imperative structures. For instance, we could define a type for imperative linked lists:

```
(* I used the term "llist" last lecture, unfortunately *)  
datatype 'a mut_list = Nil | Cons of 'a * 'a mut_list ref
```

Notably, such a type can have cycles, which is not something you can have with ordinary recursive datatypes! For instance:

```
val r : int mut_list ref = ref Nil  
val l = Cons (1, r)  
val () = r := l
```

Let's explore exactly what's going on here. We start off with the empty store.

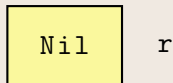
```
val r : int mut_list ref = ref Nil
val l = Cons (1, r)
val () = r := l
```

THE STORE

Then, we allocate a single reference cell containing the empty `mut_list`.

```
val r : int mut_list ref = ref Nil
val l = Cons (1, r)
val () = r := l
```

THE STORE

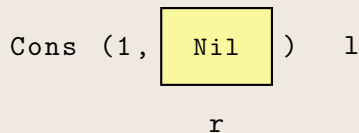


We then bind the value of `Cons (1, r)` to `l`.

```
val r : int mut_list ref = ref Nil
val l = Cons (1, r)
val () = r := l
```

THE STORE

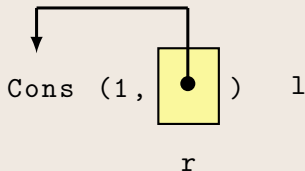
Cons (1, Nil) l
 r

A diagram illustrating the state of the store. It shows a `Cons` cell containing the value `1` and a pointer to `Nil`. The pointer is labeled `r`. The entire `Cons` cell is enclosed in a yellow box. To the right of the `Cons` cell is the variable `l`, which is bound to the `Cons` cell.

Here's where the "contains" analogy becomes imperfect. But basically, we cause `r` to now point to the `Cons` node, which is perfectly legal.

```
val r : int mut_list ref = ref Nil
val l = Cons (1, r)
val () = r := l
```

THE STORE



5 - Applications of Mutability (Bonus)

Mutability is generally problematic, but it doesn't need to be avoided at all costs.

One thing to realize about mutability is that it inherently causes problems with parallelism. One nice property we talked about on the first day was the ability to reason about components of functional programs independently – this is because those components must always evaluate the same, in the absence of effects.

Mutability *can* be safe in a deterministic environment, but it becomes much more difficult to control when running a parallel program.

	Sequential	Parallel
Immutable	safe	safe
Mutable	harder, but possible	no man's land

There are some common techniques that can be used in a functional setting with `ref` cells, however. These are generally more innocent, and have less to do with persistent usage of mutable state, as more occasional usages of mutability as a labor-saving device.

Remember, purity is a trap. We don't serve immutability – immutability and mutability serve us.

We can use `ref` cells to generate fresh integers (or *temps*), that are guaranteed to be unique across a single program's lifetime.

Use Cases Unique identifiers for each node in a graph, each person in a database, each variable in a program, each transaction in a system.

We do this by simply using an `int ref` which we increment every time we get a new ID.

```
val id_store = ref 0

fun fresh () =
  ( id_store := 1 + (!id_store);
    !id_store
  )
```

These IDs are known to be `ints`, though, so it's possible we might accidentally do arithmetic on them and get a different ID when we didn't mean to.

We can use this technique in conjunction with opaque ascription to prevent that!

```
structure UNIQUE_ID =  
  sig  
    (* an abstract type of fresh identifiers *)  
    type t  
  
    val fresh : unit -> t  
  
    (* need an equality function, or the type is useless *)  
    val eq : t * t -> bool  
  end
```

Then, we just opaquely ascribe to the above signature to be able to make sure users can only construct values of type `UniqueId.t` through the structure.

```
structure UniqueId :> UNIQUE_ID =
  struct
    (* internally an 'int', but users don't know that! *)
    type t = int

    val id_store = ref 0

    fun fresh () =
      ( id_store := 1 + (!id_store);
        !id_store
      )

    val eq = (op=)
  end
```

Suppose that you want to be able to dynamically load a given function, but you don't know what it is, necessarily, and you also can't write the function right now.

There's lots of reasons why you might want to be able to use a function, but you can't write it in the file you need it in. This could be due to compilation dependencies, types you need being defined elsewhere, or just general program logic living somewhere else.

You can make your dependencies run *backwards* by providing a `ref` cell that will *eventually* contain the function you want, and then **backpatching** it at a later point, filling in the cell before you use it.

Def A **hook** is a `ref` of type `t option ref`, where `t` is usually a function type. It starts off as `NONE`, and then at a later point is filled in to be `SOME v`, downstream.

```
val function_i_need_but_dont_have_hook :  
  (int -> int) ref = ref NONE
```

Then, several files downstream, we might write:

```
fun f x = (* ... *)  
  
val _ = function_i_need_but_dont_have_hook := SOME f
```

This fills in the hook, usually before we even get the chance to use it, so we cover our use cases.

This is also a useful way to make a piece of software's behavior easily changeable via an outside consumer, by having users have the option to set the hook.

Another classic technique is to have a `ref` which mediates some setting which governs the program's entire runtime.

This is better than explicitly passing that value down through every place where it needs to be used, which would cause an immense amount of bloat in the program!

```
fun initialize_api is_verbose () =  
  let  
    val () = if is_verbose then  
              print "Initializing API...\n"  
            else ()  
  in  
    (* ... *)  
  end
```

Here, we kill the explicit argument `is_verbose`, which would presumably need to be passed through approximately fifty billion functions, and instead reference a `bool ref` called `is_verbose`, which is set someplace else.

```
fun initialize_api () =  
  let  
    val () = if !is_verbose then  
              print "Initializing API...\n"  
            else ()  
  in  
    (* ... *)  
  end
```

This is safer to do, because we generally only will set this property once per invocation of a program.

Thank you!