$\boxed{\text{Lec 21}}$ Imperative Programming

> 122 is crap ... sadly it's sometimes useful

Reference type !

type: t ref for any type t

value: a memory cell $\boxed{v}$ ← thing in box

ref 7 $\hookrightarrow \boxed{7}$ ← allocate some cell and put 7 in there
but thing in box remains changable

Rule on ref e
1. evaluate e
2. if e $\hookrightarrow$ v then ref e $\hookrightarrow \boxed{v}$

ref e : t ref if e:t

Rule on !e
1. evaluate e
2. if e $\hookrightarrow \boxed{v}$ then !e $\hookrightarrow$ v

!e : t if e : t ref

$e_1 := e_2$
1. eval e_1
2. eval e_2
3. if $e_1 \hookrightarrow \boxed{v}$, $e_2 \hookrightarrow v'$, then overwrite content of cell with v' ← "mutation" ← side effect
4. return ()

$e_1 := e_2$: unit if e_1: t ref , e_2 : t

val c = ref 12 $\boxed{12}$ / c
val () = c := 4 $\boxed{4}$ / c
val x = !c 4 / x

* No longer functional programming

Note these are not same :

val r = ref 1
val r' = ref 1

] ref to different boxes

Multiple vars can bind to same cell

val c = ref 10 $\boxed{10} \leftarrow$ / c
val d = c ↰ /d

] aliasing

The operations

ref : 'a → 'a $\underline{\text{ref}}$ $\longrightarrow$ almost constructor :
! : 'a ref → 'a - pattern matching allowed
:= : 'a ref * 'a → unit - but application to value isn't value

```
fun containsZero (ref 0) = true
  |     - - -        _   = false
```

Value restriction : Only value can be polymorphic. Non value must have type.

val x = $\underline{\text{ref nil}}$
└ not value, so doesn't work

val x : $\underline{\text{int list ref}}$ = ref nil
└ annotating type will work

# Sequential expressions

⌐ keep result here

$(e_1 ; e_2 ; \cdots ; e_n) \hookrightarrow v_n$ if $e_i \hookrightarrow v_i$

└ not care about value. only for side effect. Evaluated left to right

$(e_1 ; e_2 ; \ldots ; e_n) : t_n$ if $e_i : t_i$

Ex.

```
let val c = ref 10
in (c := 11 ; !c)
end
      ↪ 11
```

Getting rid of allocated memory

→ Garbage collector

# Extensional Equivalence

* Still under research! Gets complicated

if $e, e' : t$, we say $e \cong e'$ if

$(e, s) \Rightarrow (v, s')$ and

$(e', s) \Rightarrow (v', s')$ and

$v \cong v'$ for every store viz. memory s.

These are sufficient, but too strong. Take 312 for more

# Race condition — Bank

```
fun deposit a n = a := !a + n
fun withdraw a n = a := !a - n
```

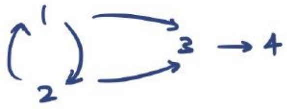
```
val acc = ref 100
val _   = ( deposit acc 50 ; withdraw acc 70 )
val x   = ! acc
    ↳ 80      Nothing special .
```

```
val acc = ref 100
val _   = ( deposit acc 50 , withdraw acc 70 )  ← in parallel
val x   = ! acc
    ↳ non-determistic !  Could be 150, 30 , 80 , ... or junk
```

* Persistent : no mutation
* Ephemeral : may have mutation

	Persistent	Ephemeral
Sequential	Functional Programming	Harder but possible
Parallel	☺	Concurrency ☹

# Benign uses of imperative feature : imperative feature for abstraction



```
type graph = int → int list
fun g 1 = [1,2]
  | g 2 = [1,3]
  | g 3 = [4]
  | g 4 = []
```

```
(* DFS *) ← Doesn't work... cycles
fun reach g (x,y) =
let fun dfs n = n = y
    orelse List.exists dfs (g n)
in
    dfs x
end
```

Solutions :

- → Keep track of visited places as list ← complicated
- → Use references

```
fun reach g (x, y) =   ← benign : whoever using can't tell it uses references
  let
    val visited = ref []
    fun dfs n = n=y orelse
        (not ( member n (! visited ))
        andalso ( visited := n::!visited ; List.exists dfs (g n)))
  in
    dfs n
  end
```

# Lazy references ?

Issue : stream access not cached

Sol : memoisation

```
fun delay d =
  let
    val answer = ref NONE
    fun f () =
      ( case !answer of
          SOME x => x     |
          NONE => (
            let val x = d () in
              ( answer := SOME x ; x )
            end )
      )
  in
    stream f
  end
```