Ref

t ref (t : any ML type)

$\boxed{v}$ → a cell e.g. $\boxed{7}$ a value of type int ref containing the value 7 of type int.
ref 7

ref e If e → v, then ref e → $\boxed{v}$

!e If e → $\boxed{v}$, then !e → v

$e_1 := e_2$ If $e_1 \to \boxed{w}$ and if $e_2 \to v$, then replace w with v in the cell and return ()

```
e.g. val c = ref 7    }  [4] / c
     val () = c := 4  }
     val v = !c       }  4 / v
```

type rules
- ref e : t ref if e : t
- !e : t if e : t ref
- $e_1 := e_2$: unit if e_1 : t ref and e_2 : t

ref is a constructor, 'a → 'a ref
can pattern matching

aliasing

```
val c = ref 10
val w = !c        c & d are aliases
val d = c
val () = d := 42
val v = !c   → 10/w  42/v
```

equality

```
val c = ref 10
val e = ref 10    => !c = !e
val d = c            c = d → true
                     c = e → false
```

equal iff bound to the same cell!

sequential expression

$(e_1; e_2; \cdots; e_n) : t_n$ if $\exists\, t_i$ s.t. $e_i : t_i,\ i = 1, \ldots, n$

evaluate left → right

If e_i loops forever / raises exc, overall expression loops forever / raises exc

e.g. (print (...); ref 10) → ref 10

Store: the set of accessible reference cells & their contents

$\{e; s\} \Rightarrow \{e'; s'\}$ with e, e', s, s' stores

$e \cong e'$ independent of store: $\{e; s\} \Rightarrow \{v; s'\}$ and $\{e'; s\} \Rightarrow \{v; s'\}$ with v a value $\forall$ initial stores s

Race condition

```
fun deposit a n = a := !a + n    : int ref -> int -> int ref
fun withdraw a n = a := !a - n
val chk = ref 100
val _ = (deposit chk 50; withdraw chk 80)
!chk ~> 70
val _ = (deposit chk 50, withdraw chk 80)
no definitive !chk
```

	Persistent	Ephemeral
Sequential	Functional programming is good tool	FP is fine, reasoning hard
Parallel		concurrency ??

```
(* reach : graph -> int * int -> bool *)
fun reach (g: graph) (x, y) =
    let
        fun dfs n = (n = y) orelse (List.exists dfs (g n))
```

check if y is reachable from any of n's neighbors

```
    in
        dfs x
    end
```

☆ possible infinite loop

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

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

```
signature RANDOM =
sig
  type gen (* abstract *)
  val init : int -> gen
  val random : gen ->
end
```

```
structure R :> RANDOM =
struct
    type gen = real ref
    val a = 16807.0
    val m = 2147483667.0
    fun next r = a*r - m*real (floor (a*r/m))
    val init = ref o real
    fun random g b = (g := next (!g); floor ((!g/m) * (real b)))
end
```

Stream Memoization

```
fun delay d =
   let
      val cell = ref d
      fun memoFn () =
          let
             val r = d()
          in
             (cell := (fn () => r); r)
          end handle E => (cell := (fn () => raise E); raise E)
      val _ = cell := memoFn
   in
      Stream (fn () => !cell ())
   end
```