λ

CS 251 Spring 2020
Principles of Programming Languages
Ben Wood

WELLESLEY

Restricted Mutable State

ML has (restricted) mutation

- Mutable data structures are okay/useful in some situations
  - When "update to state of world" is appropriate model
  - But want most language constructs truly immutable

- ML does this with an explicit separate construct: **references**

- **Do not use references on your homework.**

Reference Cells

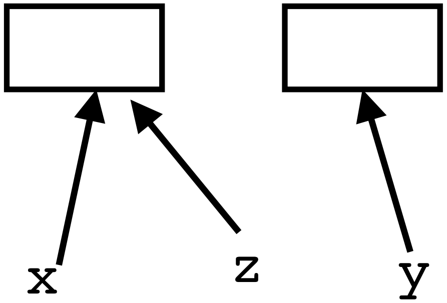
New types: `'a ref`

New expressions:

- Creation: `ref e`
  - **Evaluation:** create a ref cell holding result of evaluating `e`
  - Type-checking: if `e : t`, then `ref e : t ref`
- Update contents: `e1 := e2`
  - **Evaluation:** evaluate `e1` to a ref cell, `e2` to a value; update ref cell to hold value as its contents.
  - Type-checking:
    if `e1 : t ref` and `e2 : t`, then `e1 := e2 : unit`
- Get contents: `!e`
  - **Evaluation:** evaluate `e` to a ref cell; result is its contents.
  - Type-checking: if `e : t ref`, then `!e : t`

References example

```
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!y) + (!z) (* 85 *)
(* x + 1 does not type-check *)
```



- A **variable** bound to a ref cell is still **immutable:** permanently bound to the same ref cell
  - There may be ***aliases*** to the reference, which matter a lot
- References are **first-class** values
  - Like a one-field mutable object. `:=` and `!` don't specify field
- *Contents* of the reference may change via `:=`

optional

Callback idiom

Library takes function to apply later, when an *event* occurs.

Library interface:

```
   val onKeyEvent : (int -> unit) -> unit
```

Other examples:

- When a key is pressed, mouse moves, data arrives
- When the program enters some state (e.g., turns in a game)

A library may accept multiple callbacks

- Different callbacks need different private data with different types
- Callback function's type does not include the types of bindings in its environment!

optional

Library implementation

Mutable state not absolutely necessary,
but is reasonably appropriate.

Create new ref cell
with initial contents []

```
val cbs : (int -> unit) list ref = ref []
```

Get contents of ref cell.

```
fun onKeyEvent f =  cbs := f :: (!cbs)
```

Set contents of ref cell.

```
fun onEvent i =
    let
      fun loop fs =
          case fs of
            []     => ()
          | f::fs’ => (f i; loop fs’)
    in
      loop (!cbs)
    end
```

Sequencing expression ;
Evaluate left side and throw away result,
then evaluate right side and use result.

optional

Clients

Closure's environment captures any necessary context, possibly including mutable state for "remembering" history.

```
val timesPressed = ref 0
val _ = onKeyEvent (fn _ =>
            timesPressed := (!timesPressed) + 1)
fun printIfPressed i =
   onKeyEvent (fn j =>
       if i=j
       then print ("pressed " ^ Int.toString i)
       else ())
fun makeCounterCallback k =
   let count = ref 0 in
     onKeyEvent (fn i => if i=k
                         then count := !count + 1
                         else ());
     count
   end
```