Mutation

COS 326

Andrew W. Appel

Princeton University

Mutation?


Breaking News
LIVE Mutant Alert! New cow-goat
TVN
EXCLUSIVE
ICANHASCHEEZBURGER.COM

Reasoning about Mutable State is Hard

mutable set

```
insert i s1;
f x;
member i s1
```

immutable set

```
let s1 = insert i s0 in
f x;
member i s1
```

Is member i s1 == true? …

- When s1 is mutable, one must look at f to determine if it modifies s1.
- Worse, one must often solve the *aliasing problem*.
- Worse, in a concurrent setting, one must look at *every other function* that *any other thread may be executing* to see if it modifies s1.

Thus far…

We have considered the (almost) purely functional subset of Ocaml.

- We've had a few side effects: printing & raising exceptions.

Two reasons for this emphasis:

- *Reasoning about functional code is easier*.
  - Both formal reasoning
    - equationally, using the substitution model
    - and informal reasoning
  - Data structures are *persistent*.
    - They don't change – we build new ones and let the garbage collector reclaim the unused old ones.
  - *Hence, any invariant you prove true stays true*.
    - e.g., 3 is a member of set S.
- *To convince you that you don't need side effects for many things where you previously thought you did*.
  - Programming with *basic immutable data like ints, pairs, lists is easy*.
    - once it type checks, it is often right or just about right
    - do not fear recursion!
  - You can implement *expressive*, *highly reuseable functional* data structures like polymorphic 2-3 trees or dictionaries or stacks or queues or sets or expressions or programming languages with reasonable space and time.

But alas…

Purely functional code is pointless.

- The whole reason we write code is to have some effect on the world.
- For example, the OCaml top-level loop prints out your result.
  - Without that printing (a side effect), how would you know that your functions computed the right thing?

Some algorithms or data structures need mutable state.

- Hash-tables have (essentially) constant-time access and update.
  - The best functional dictionaries have either:
    - logarithmic access & logarithmic update
    - constant access & linear update
    - constant update & linear access
  - Don't forget that we give up something for this:
    - we can't go back and look at previous versions of the dictionary. We *can* do that in a functional setting.
- Robinson's unification algorithm
  - A critical part of the OCaml type-inference engine.
  - Also used in other kinds of program analyses.
- Depth-first search, more ...

However, ~~purely~~ mostly functional code is amazingly productive

Tune ev'ry heart and ev'ry voice…

John Alan Robinson
1928 –
PhD Princeton 1956 (philosophy)
Professor (emeritus), Syracuse U.

- Robinson's unification algorithm (1965)
  - A critical part of the OCaml type-inference engine.
  - Also used in other kinds of program analyses, and many other applications

The value of a classics degree

Inventor (1960s) of algorithms now fundamental to computational logical reasoning (about software, hardware, and other things…)

John Alan Robinson
1928 –
PhD Princeton 1956 (philosophy)

"Robinson was born in Yorkshire, England in 1930 and left for the United States in 1952 with a classics degree from Cambridge University. He studied philosophy at the University of Oregon before moving to Princeton University where he received his PhD in philosophy in 1956. He then worked at Du Pont as an operations research analyst, where he learned programming and taught himself mathematics. He moved to Rice University in 1961, spending his summers as a visiting researcher at the Argonne National Laboratory's Applied Mathematics Division. He moved to Syracuse University as Distinguished Professor of Logic and Computer Science in 1967 and became professor emeritus in 1993."
--Wikipedia

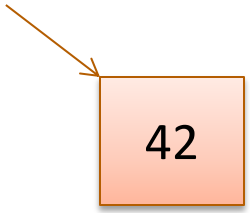
OCAML MUTABLE REFERENCES

References

- New type: `t ref`
  - Think of it as a pointer to a *box* that holds a t value.
  - The contents of the box can be read or written.

References

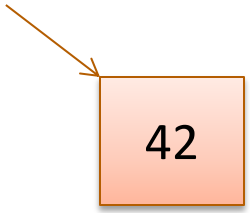
- New type: `t ref`
  - Think of it as a pointer to a *box* that holds a t value.
  - The contents of the box can be read or written.
- To create a fresh box: `ref 42`
  - allocates a new box, initializes its contents to 42, and returns a pointer:



  - `ref 42 : int ref`

References

- New type: `t ref`
  - Think of it as a pointer to a *box* that holds a t value.
  - The contents of the box can be read or written.
- To create a fresh box: `ref 42`
  - allocates a new box, initializes its contents to 42, and returns a pointer:



  - `ref 42 : int ref`
- To read the contents: `!r`
  - if `r` points to a box containing 42, then return 42.
  - if `r : t ref` then `!r : t`
- To write the contents: `r := 5`
  - updates the box that r points to so that it contains 5.
  - if `r : t ref` then `r := 5 : unit`

Example

```
let c = ref 0 ;;

let x = !c ;;    (* x will be 0 *)

c := 42 ;;

let y = !c ;;    (* y will be 42.
                    x will still be 0! *)
```

Another Example

```
let c = ref 0 ;;

let next() =
  let v = !c in
  (c := v+1 ; v)
```

Another Example

```
let c = ref 0 ;;

let next() =
  let v = !c in
  (c := v+1 ; v)
```

If `e1 : unit`
and `e2 : t` then
`(e1 ; e2) : t`

You can also write it like this:

```
let c = ref 0 ;;

let next() : int =
  let (v : int)  = !c in
  let (_ : unit) = c := v+1 in
  v
```

```
(e1 ; e2) == (let _ = e1 in e2)
```

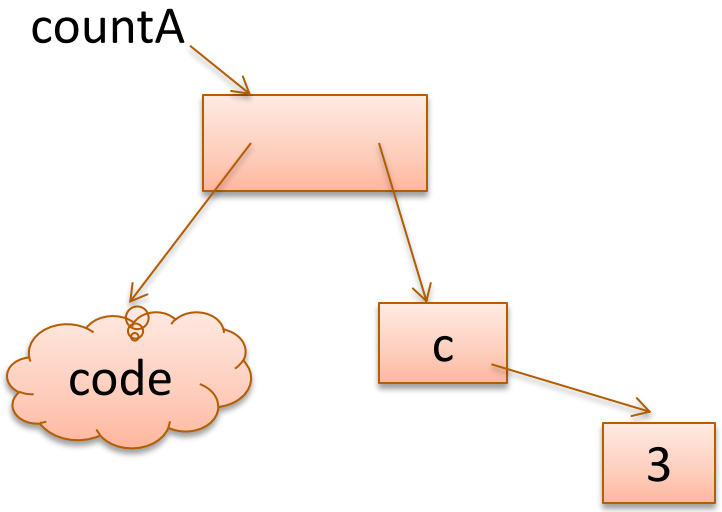
(syntactic sugar)

Another Idiom

Global Mutable Reference

```
let c = ref 0 ;;

let next () : int =
  let v = !c in
  (c := v+1 ; v)
;;
```



Mutable Reference Captured in Closure

```
let counter () =
   let c = ref 0 in
   fun () ->
     let v = !c in
     (c := v+1 ; v)
;;

let countA = counter() in
let countB = counter() in
countA() ;; (* 1 *)
countA() ;; (* 2 *)
countB() ;; (* 1 *)
countB() ;; (* 2 *)
countA() ;; (* 3 *)
```

Imperative loops

```
(* sum of 0 .. n *)

let sum (n:int) =
  let         s = ref 0 in
  let current = ref n in
  while !current > 0 do
    s := !s + !current;
    current := !current - 1
  done;
  !s
;;
```

```
(* print n .. 0 *)
let count_down (n:int) =
  for i = n downto 0 do
    print_int i;
    print_newline()
  done;
;;

(* print 0 .. n *)
let count_up (n:int) =
  for i = 0 to n do
    print_int i;
    print_newline()
  done;
;;
```

Imperative loops?

```
(* print n .. 0 *)

let count_down (n:int) =
  for i = n downto 0 do
    print_int i;
    print_newline()
  done
;;
```

```
(* for i=n downto 0 do f i *)

let rec for_down
          (n : int)
          (f : int -> unit)
             : unit =
  if n >= 0 then
   (f n; for_down (n-1) f)
  else
   ()
;;

let count_down (n:int) =
  for_down n (fun i ->
    print_int i;
    print_newline()
  )
;;
```

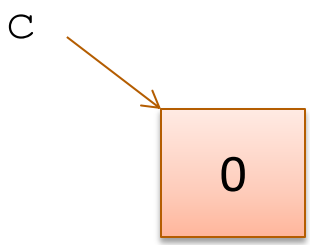
Aliasing

```
let c = ref 0 ;;

let x = c ;;

x := 42 ;;

!c ;;
```



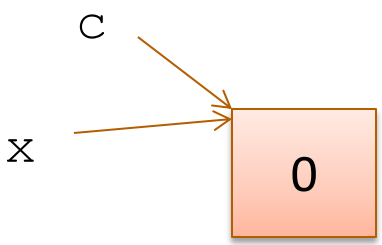
Aliasing

```
let c = ref 0 ;;

let x = c ;;

x := 42 ;;

!c ;;
```



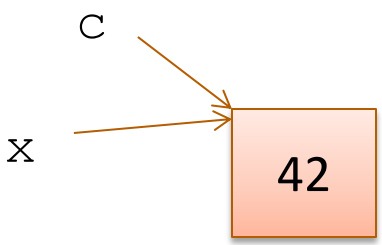
Aliasing

```
let c = ref 0 ;;

let x = c ;;

x := 42 ;;

!c ;;
```



Aliasing

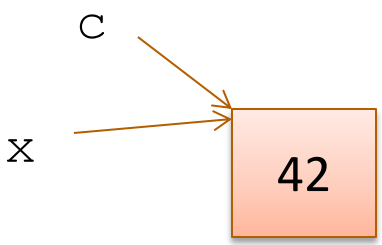
warning! we can't say !c == 0

```
let c = ref 0 ;;

let x = c ;;

x := 42 ;;

!c ;;
```



result: 42

END