Programming Languages and Techniques (CIS120)

Lecture 17

Hidden State, Objects

Chapter 17

Announcements

- Homework 5: GUI Programming
  - Available soon
  - Due: Oct 22nd
- Fall Break:
  - No lab/recitation sections this week
  - No class on Friday

Hidden State

Encapsulating State

An “incr” function

- Functions with internal state

```
type counter_state = { mutable count:int }

let ctr = { count = 0 }

(* each call to incr will produce the next integer *)
let incr () : int =
  ctr.count <- ctr.count + 1;
  ctr.count
```

- Drawbacks:
  - *No modularity:* There is only one counter in the world. If we want another counter, we need to build another counter_state value (say, ctr2) and another incrementing function (incr2)
  - *No encapsulation*: Code anywhere in the rest of the program can modify count

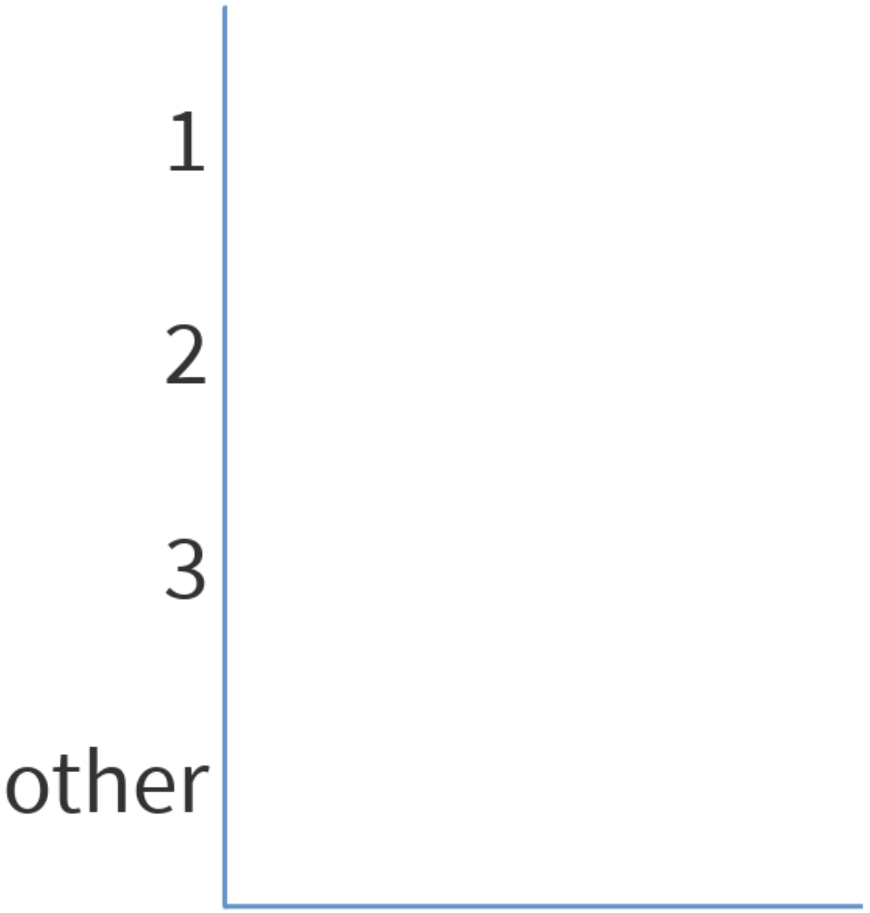
Using Hidden State

- Better: Make a function that creates a counter state and an incr function each time a counter is needed

```
(* More useful: a counter generator: *)
let mk_incr () : unit -> int =
  (* this ctr is private to the returned function *)
  let ctr = { count = 0 } in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

(* make one counter *)
let incr1 : unit -> int = mk_incr ()

(* make another counter *)
let incr2 : unit -> int = mk_incr ()
```

What number is printed by this program?

```
let mk_incr () : unit -> int =
  let ctr = { count = 0 } in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

let incr1 = mk_incr () (* make one counter *)
let incr2 = mk_incr () (* and another *)

let _ = incr1 () in print_int (incr2 ())
```



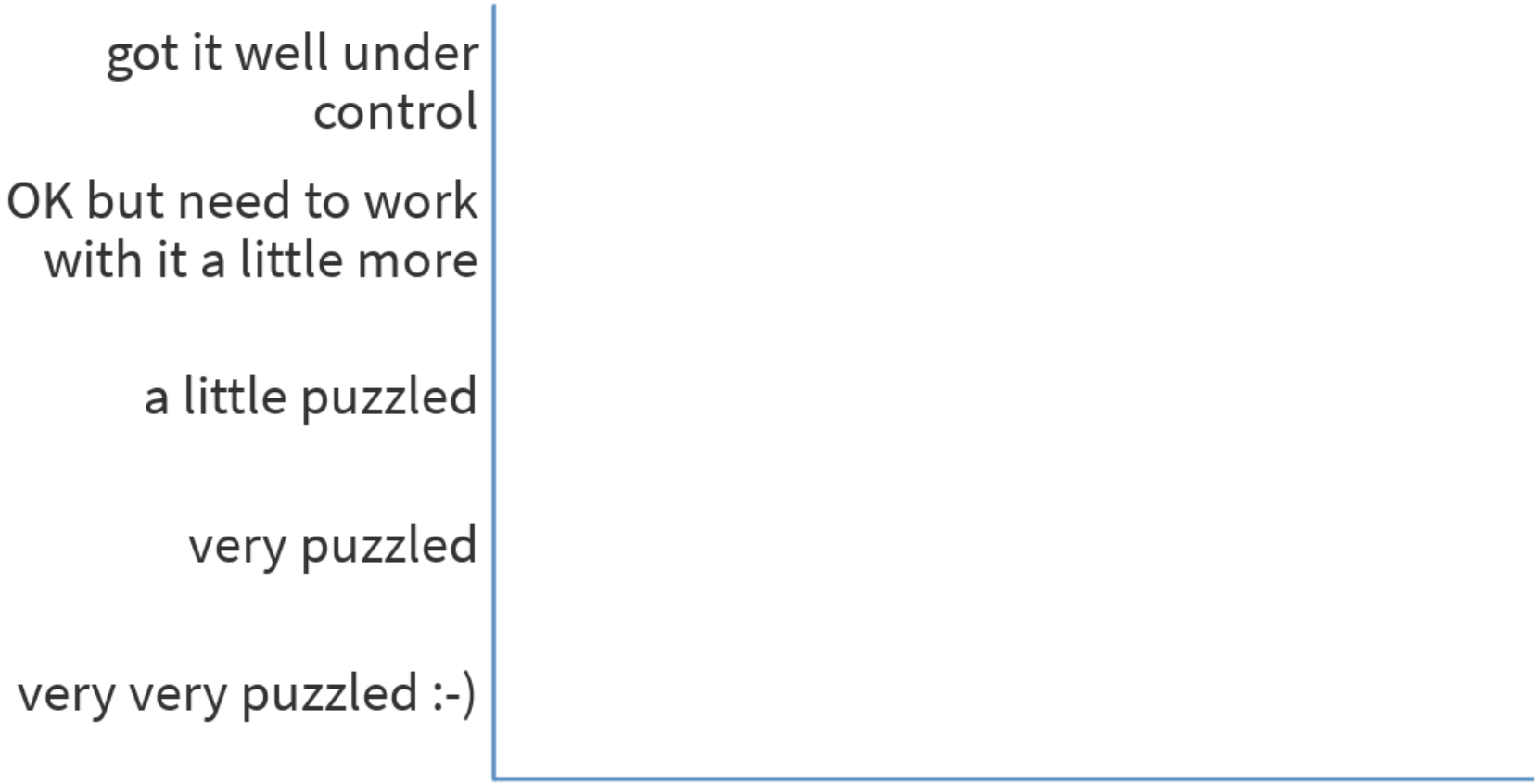
What number is printed by this program?

```
let mk_incr () : unit -> int =
  let ctr = { count = 0 } in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count

let incr1 = mk_incr () (* make one counter *)
let incr2 = mk_incr () (* and another *)

let _ = incr1 () in print_int (incr2 ())
```

1. 1
2. 2
3. 3
4. other

Answer: 1

What is your current level of comfort with the Abstract Stack Machine?



Running mk_incr

Workspace

```
let mk_incr () : unit -> int =
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count


let incr1 : unit -> int =
mk_incr ()
```

Stack

Heap

Running mk_incr

Workspace

```
let mk_incr : unit -> unit ->
int = fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count


let incr1 : unit -> int =
mk_incr ()
```

Stack

Heap

Running mk_incr

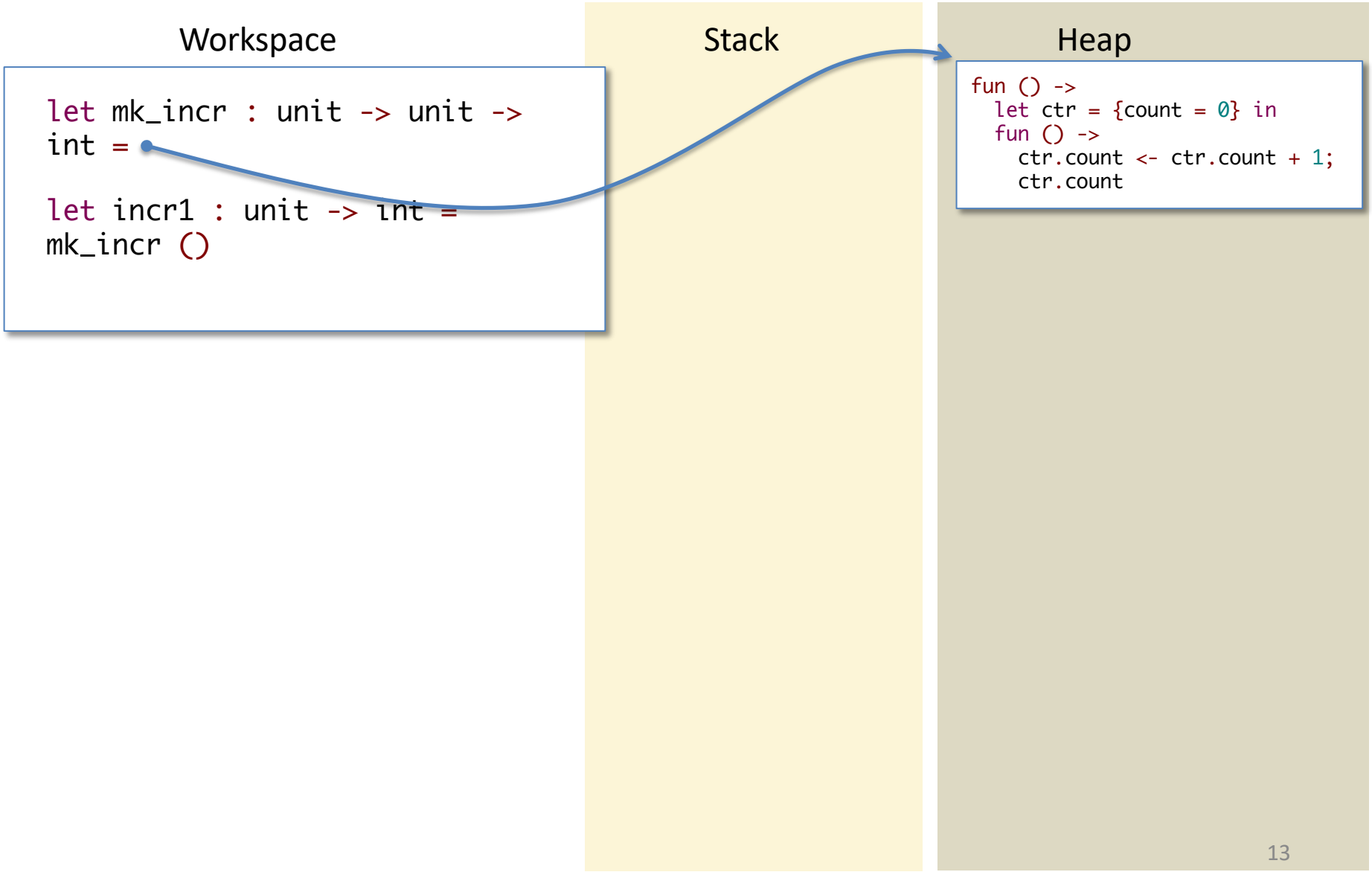
Workspace

```
let mk_incr : unit -> unit ->
int = fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count


let incr1 : unit -> int =
mk_incr ()
```

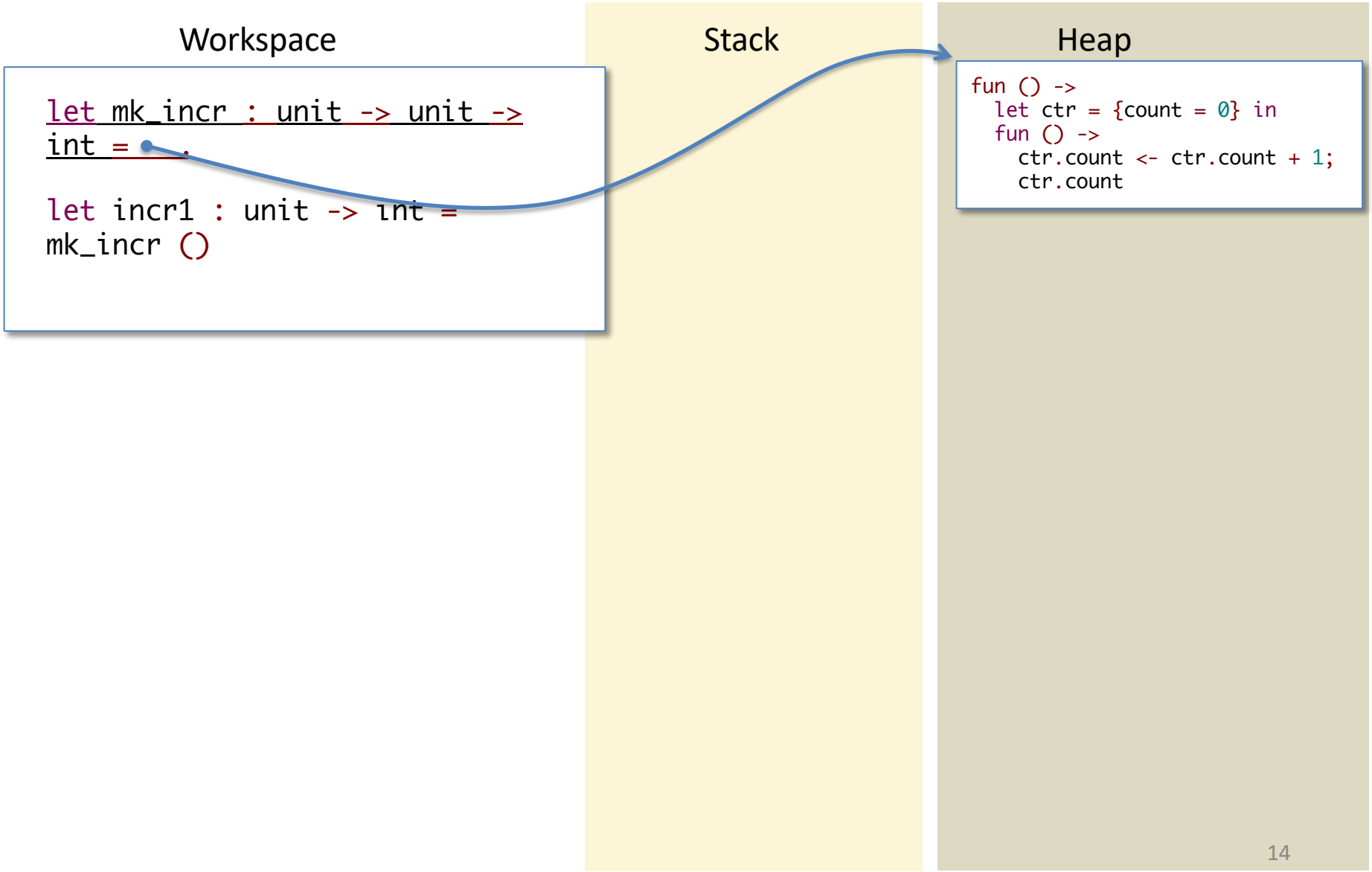
Stack

Heap

Running mk_incr



Running mk_incr

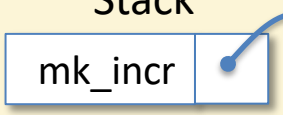


Running mk_incr

Workspace

```
let incr1 : unit -> int =
mk_incr ()
```

Stack



Heap

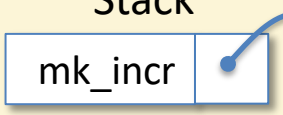
```
fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

Running mk_incr

Workspace

```
let incr1 : unit -> int =
mk_incr ()
```

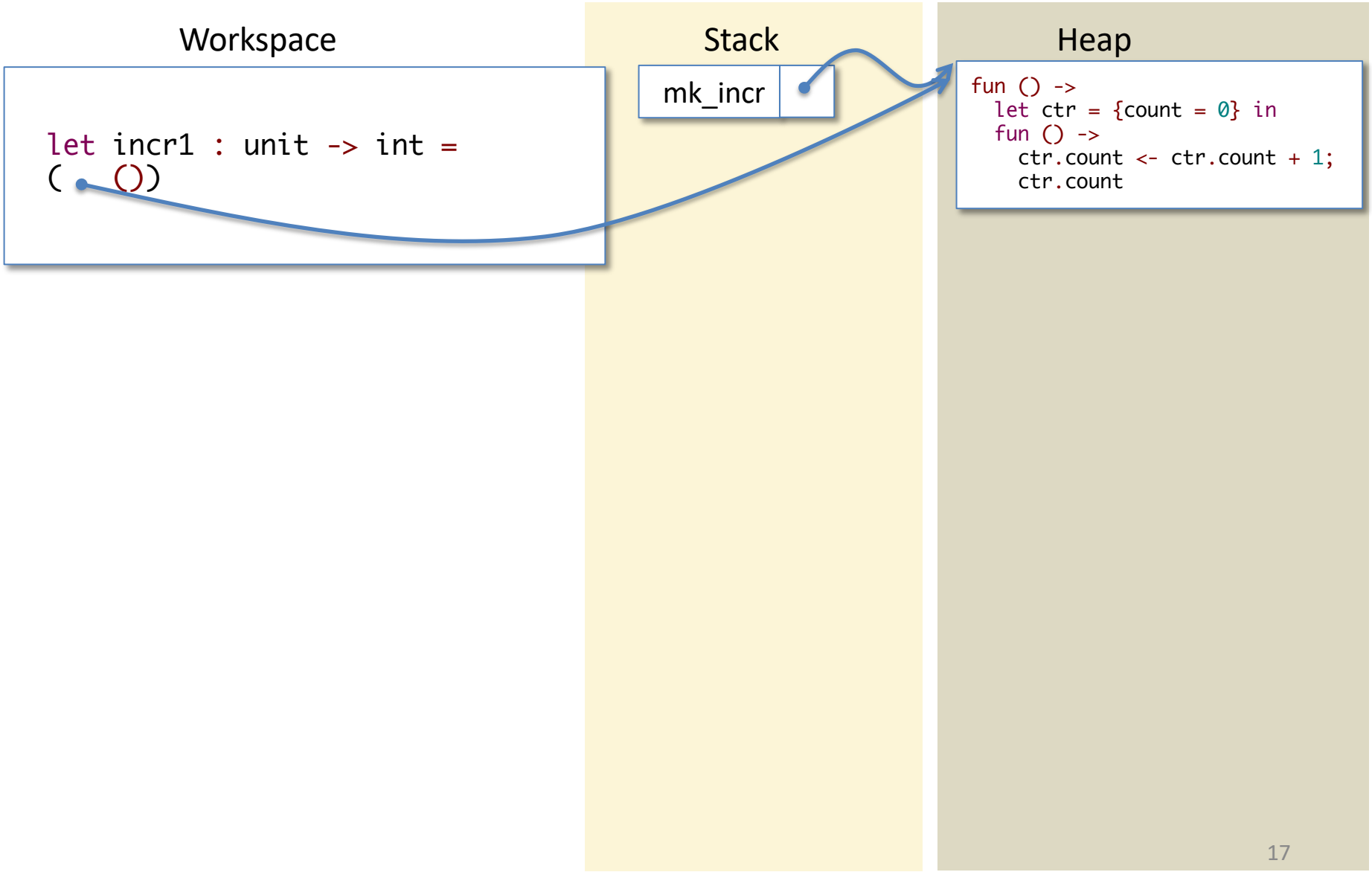
Stack



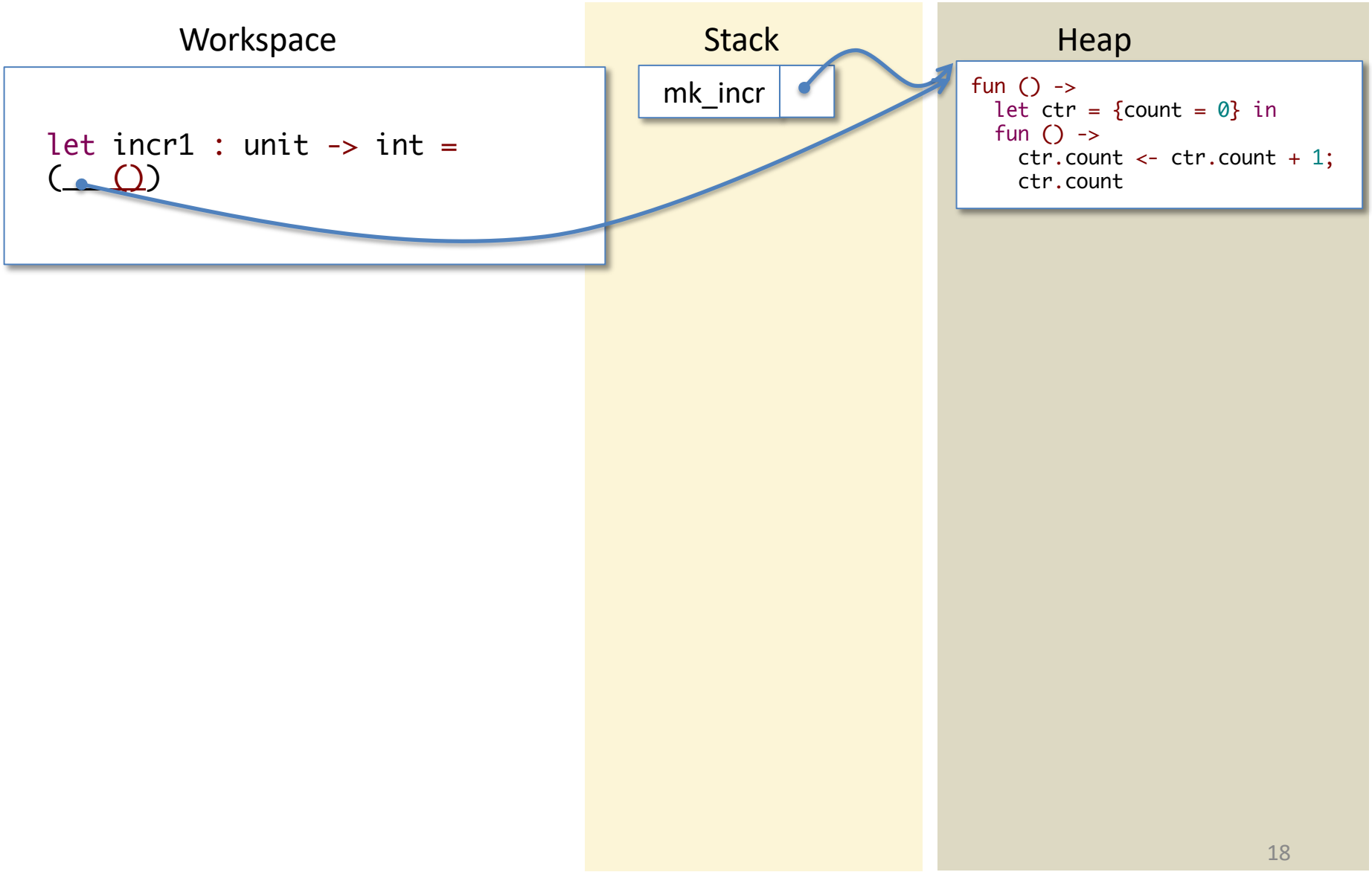
Heap

```
fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

Running mk_incr



Running mk_incr

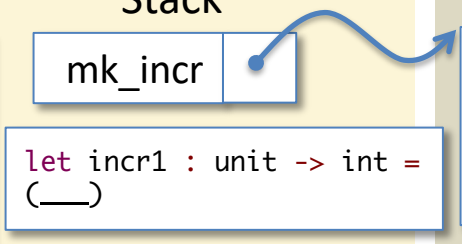


Running mk_incr

Workspace

```
let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

Stack



Heap

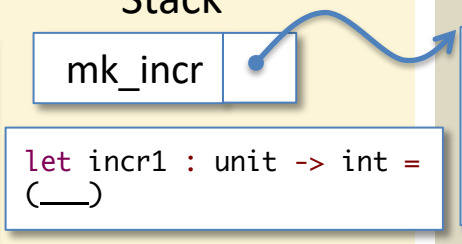
```
fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

Running mk_incr

Workspace

```
let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

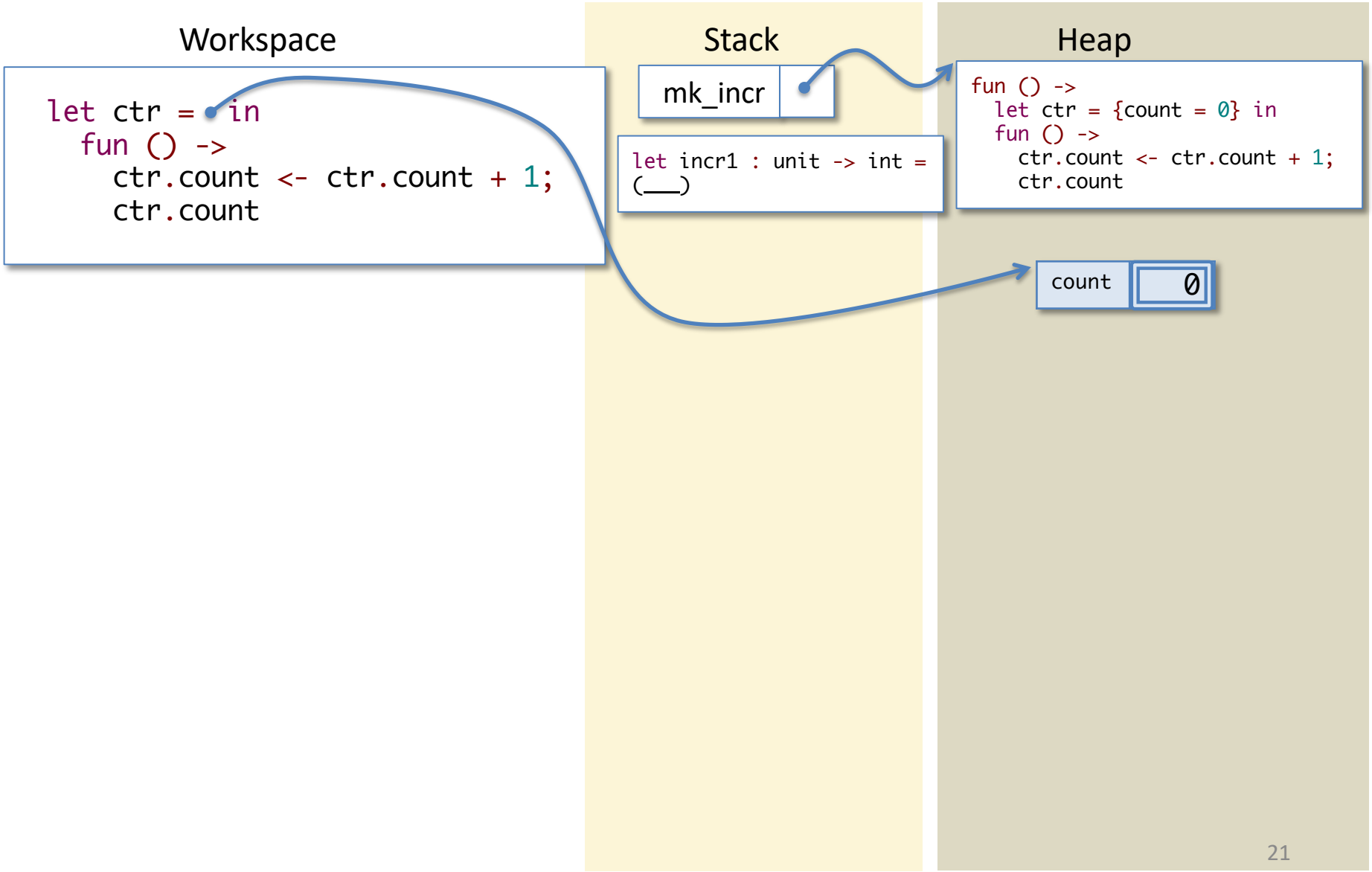
Stack



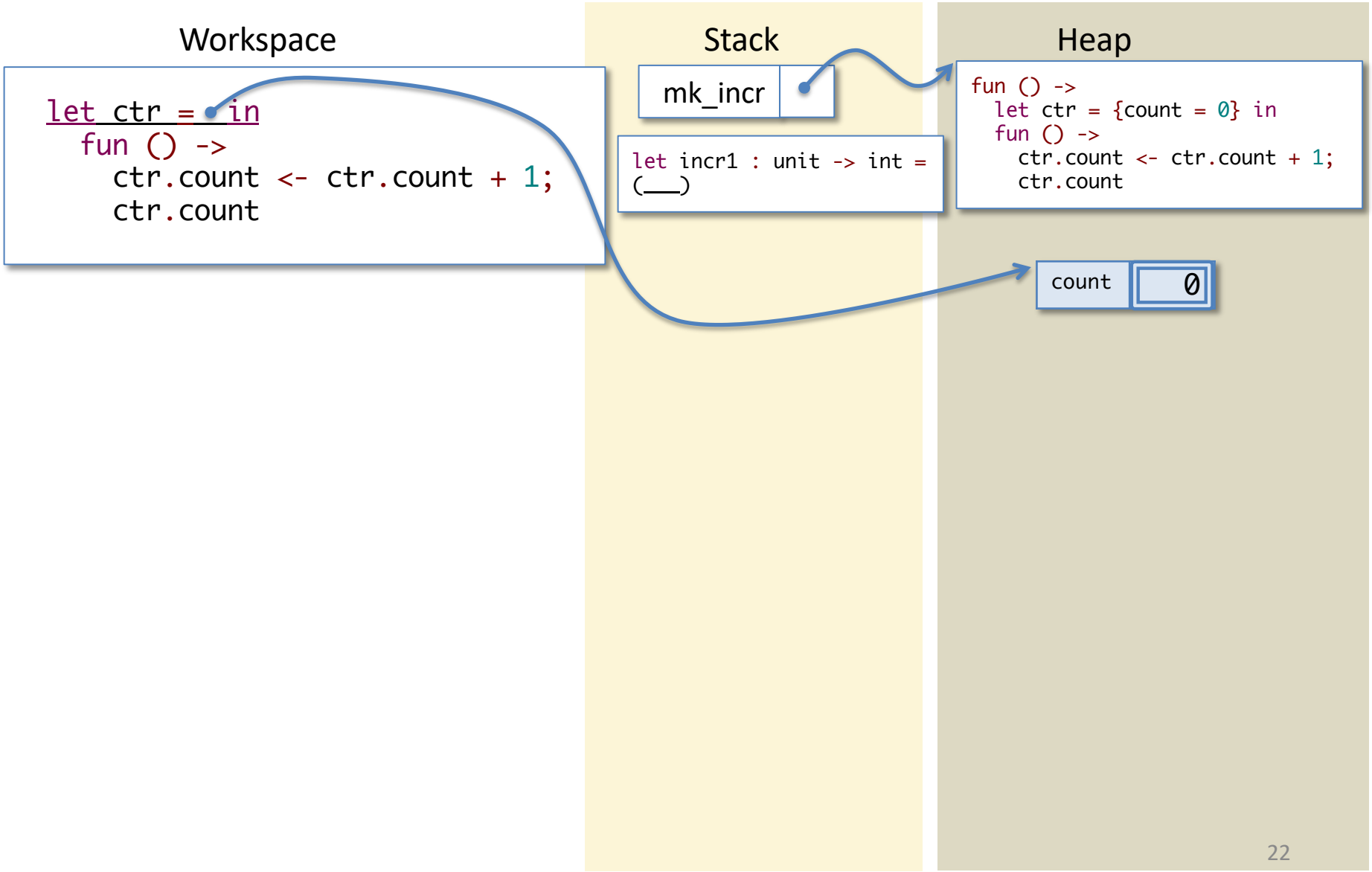
Heap

```
fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

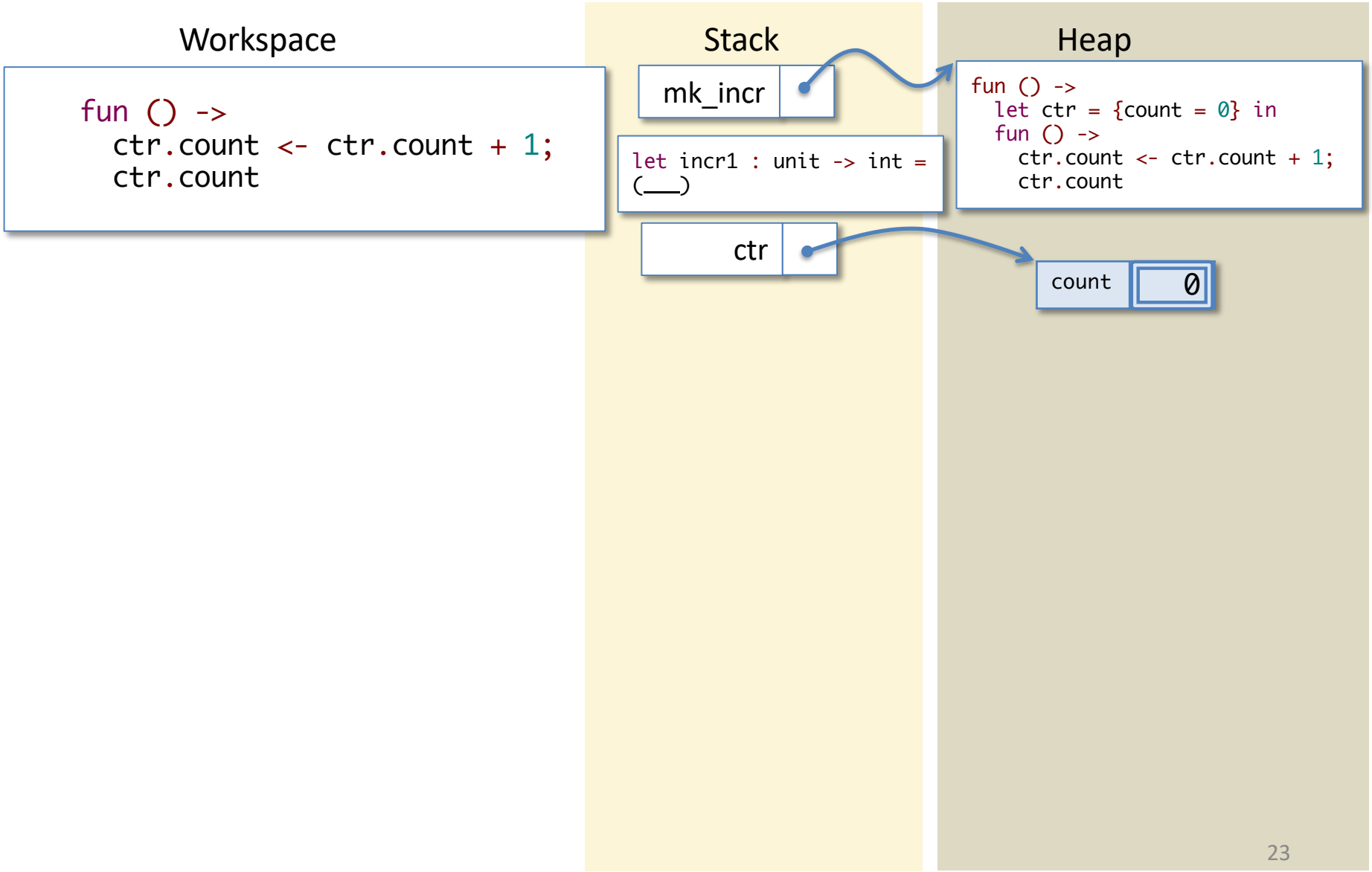
Running mk_incr



Running mk_incr



Running mk_incr



Running mk_incr

Workspace

```
fun () ->
  ctr.count <- ctr.count + 1;
  ctr.count
```

Stack

mk_incr

```
let incr1 : unit -> int =
(___)
```

ctr

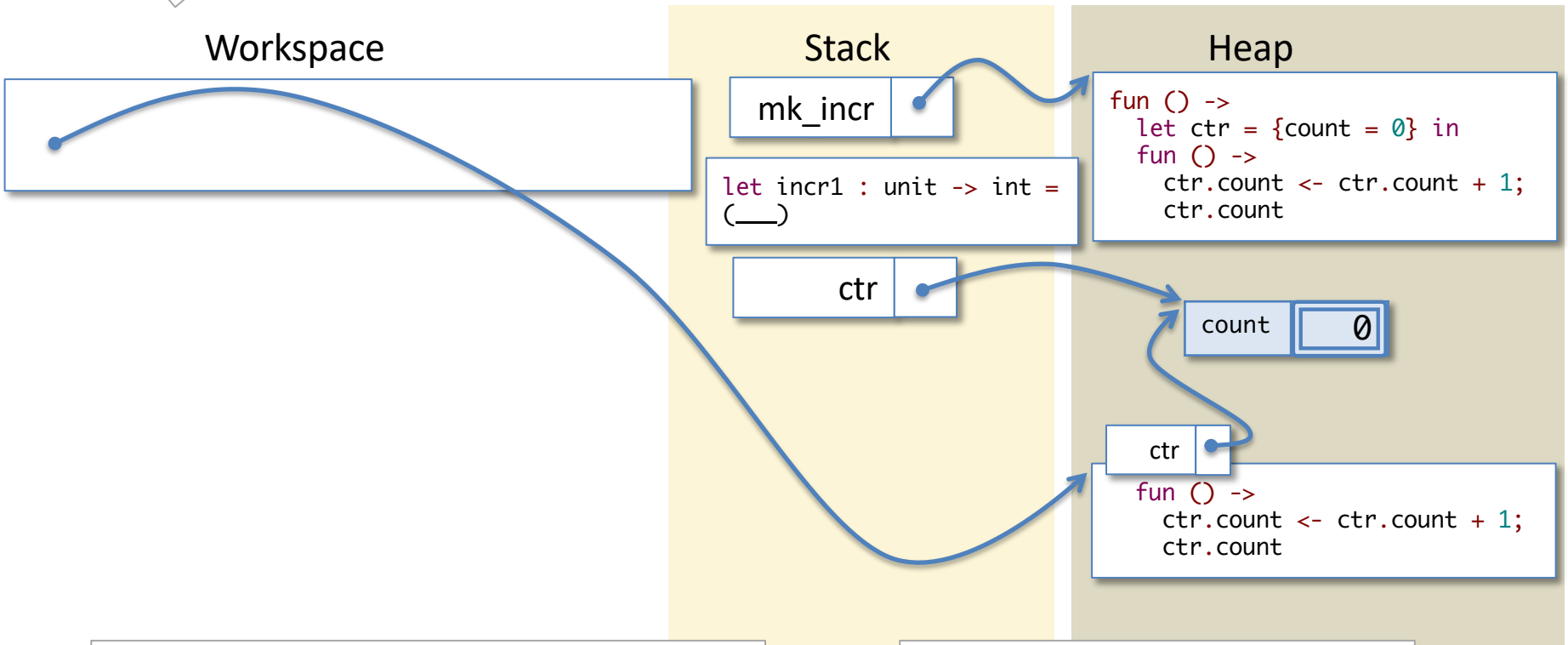
Heap

```
fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

count 0

Local Functions

Key step!



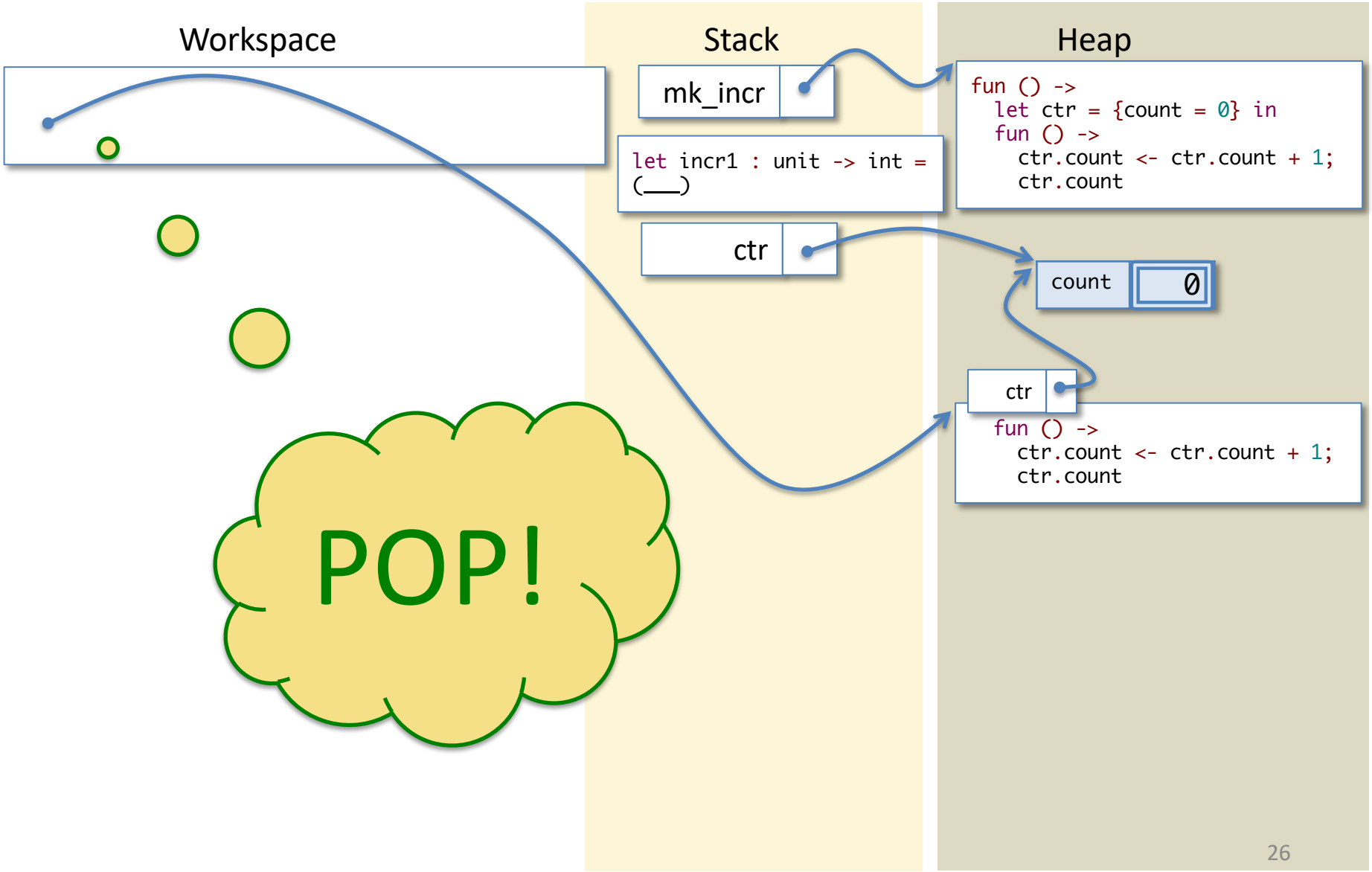
Note: We need one refinement of the ASM model that we've explained so far. Why?

The function body mentions "ctr", which is on the stack at the moment *but about to be popped off*…

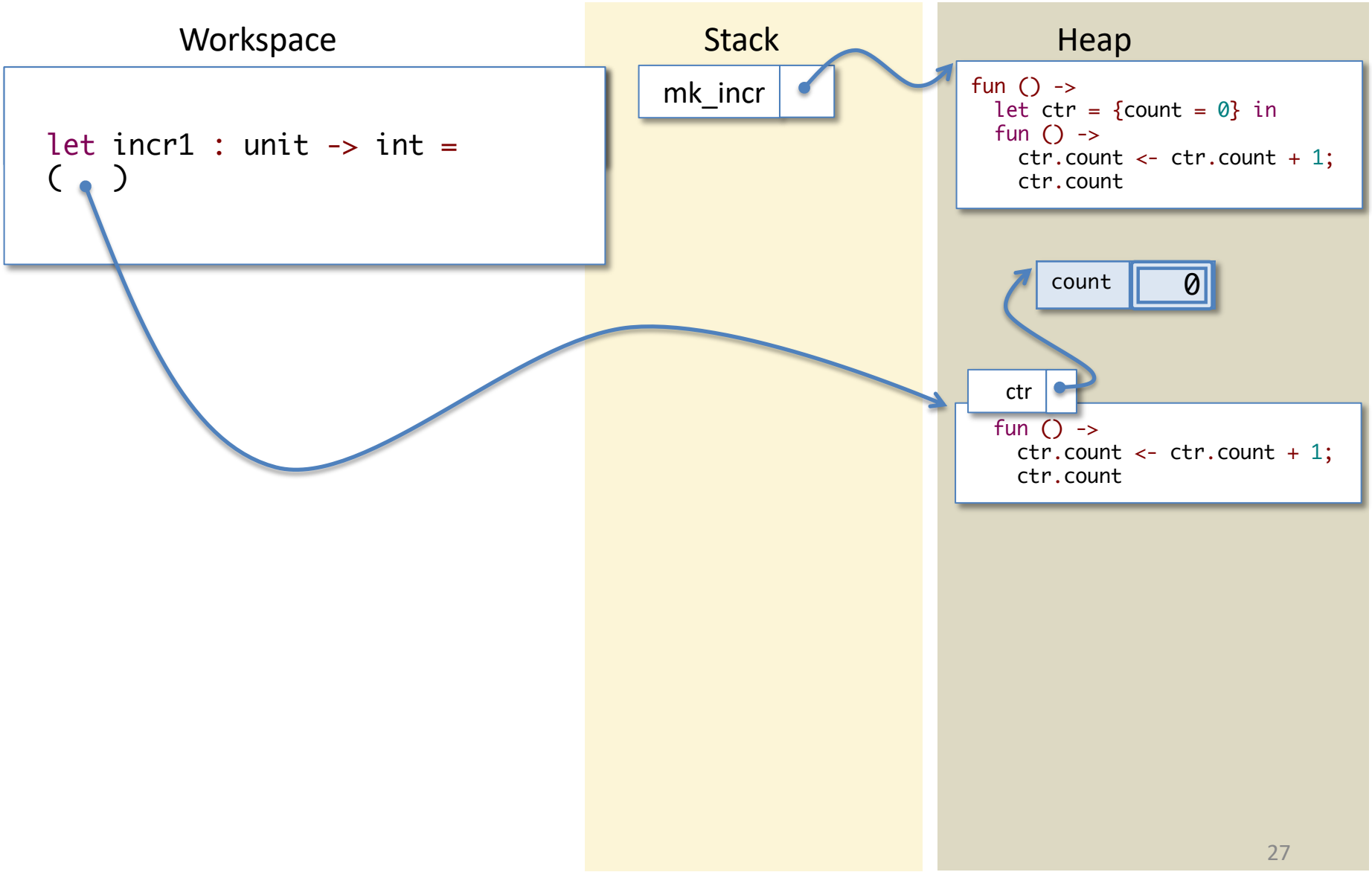
…so we save a copy of the needed stack bindings with the function itself.

This package of "function body plus needed bindings" is called a *closure*…

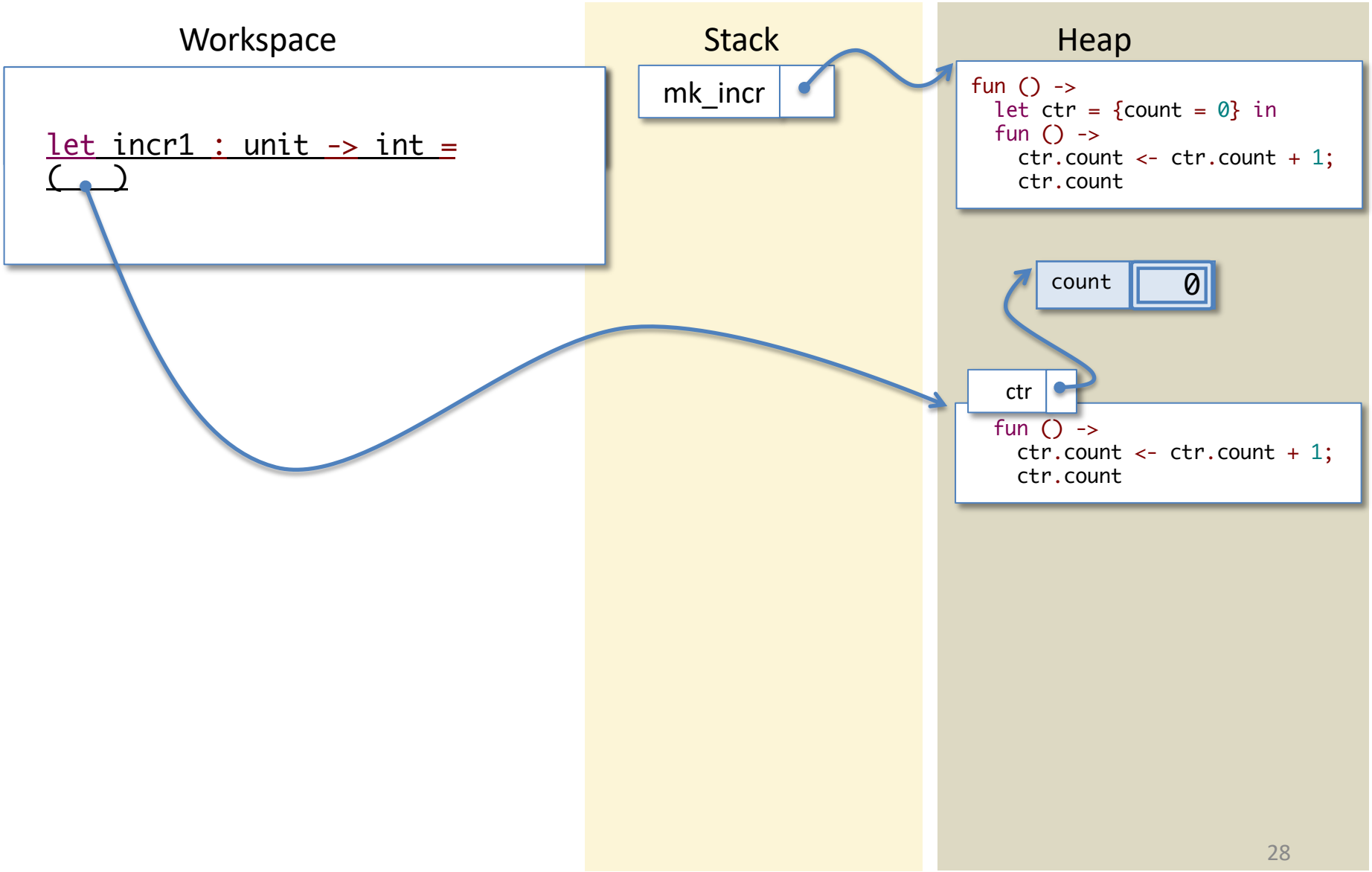
Local Functions



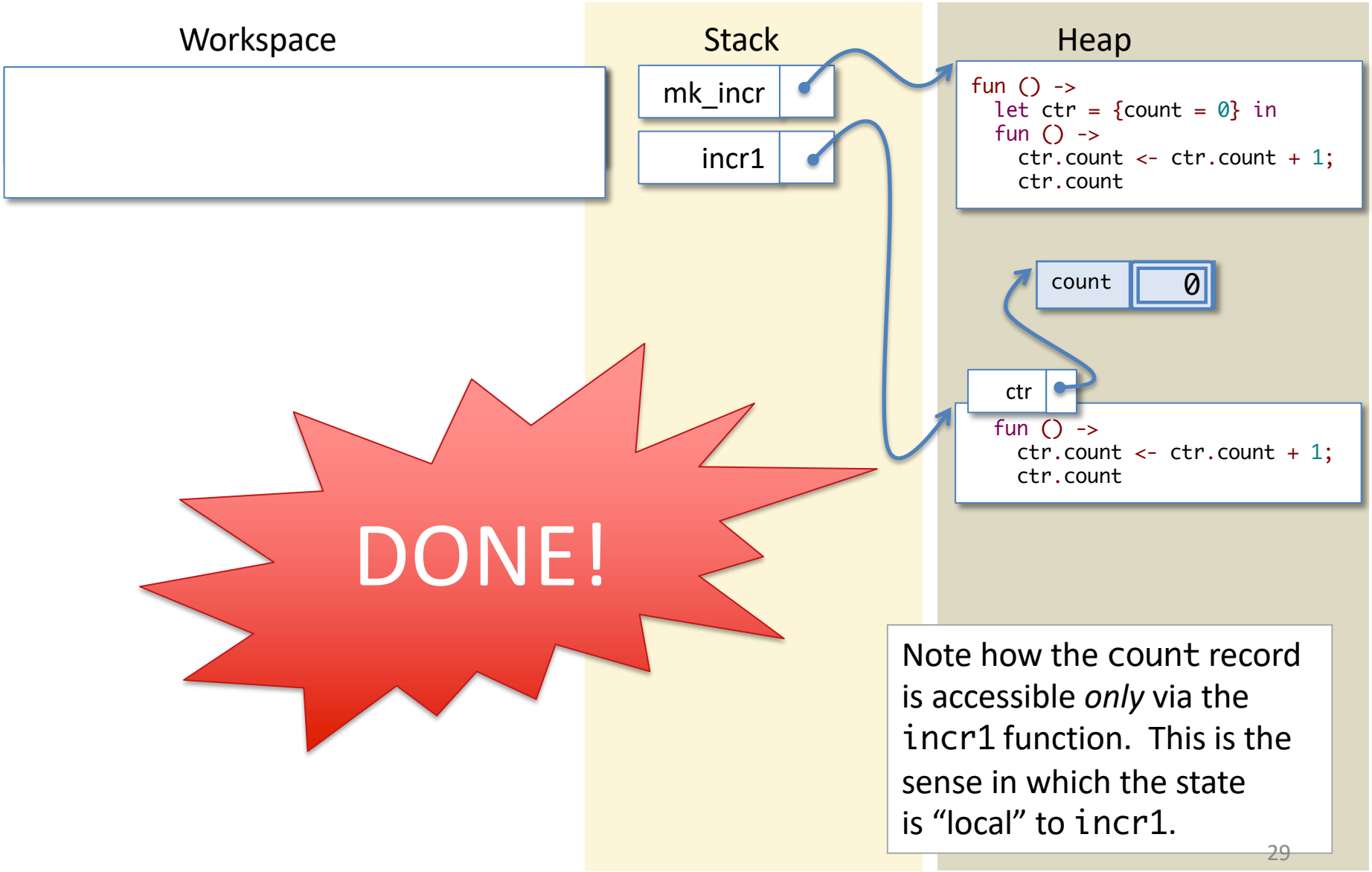
Local Functions



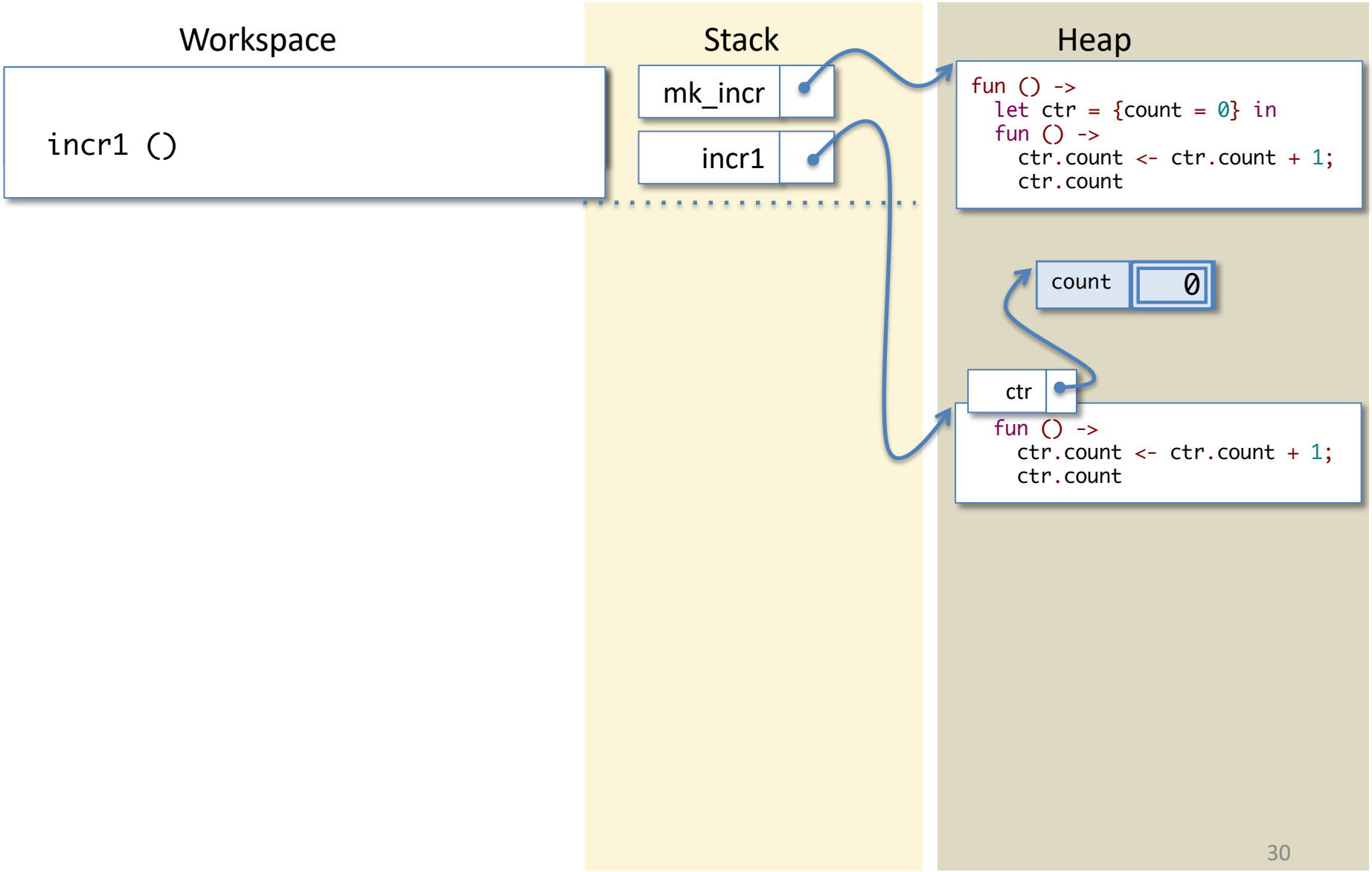
Local Functions



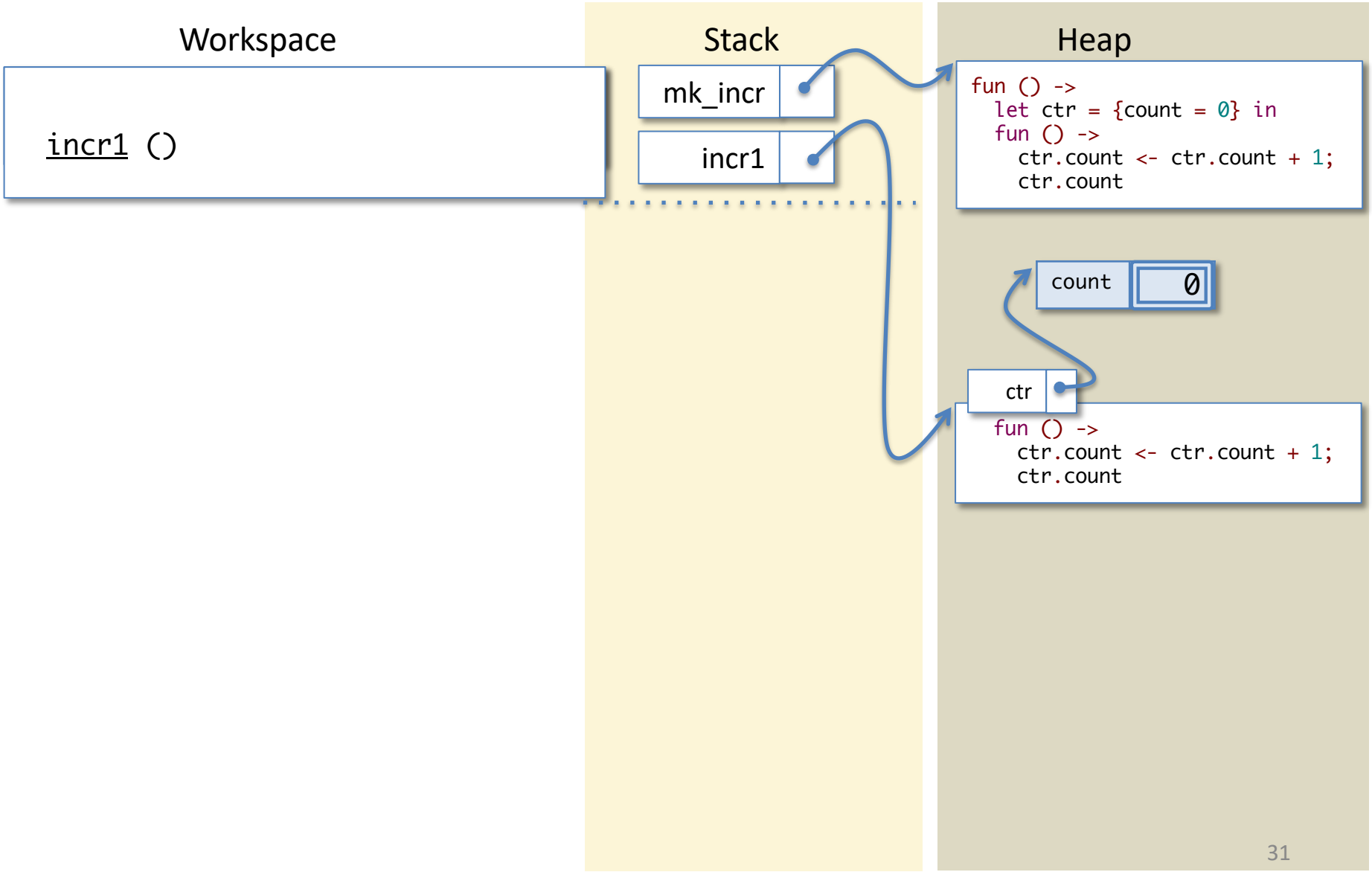
Local Functions



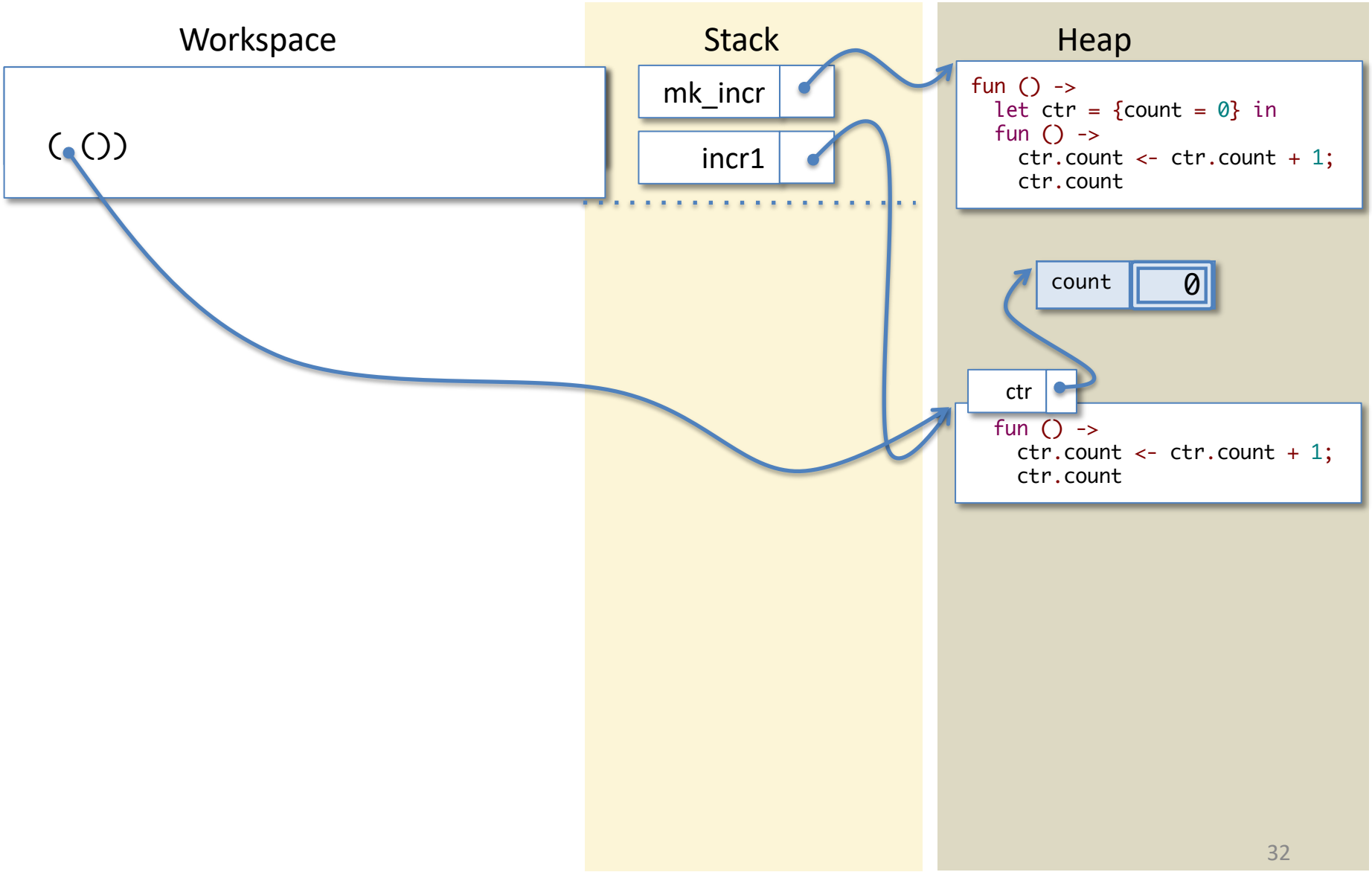
Now let's run "`incr1 ()`"



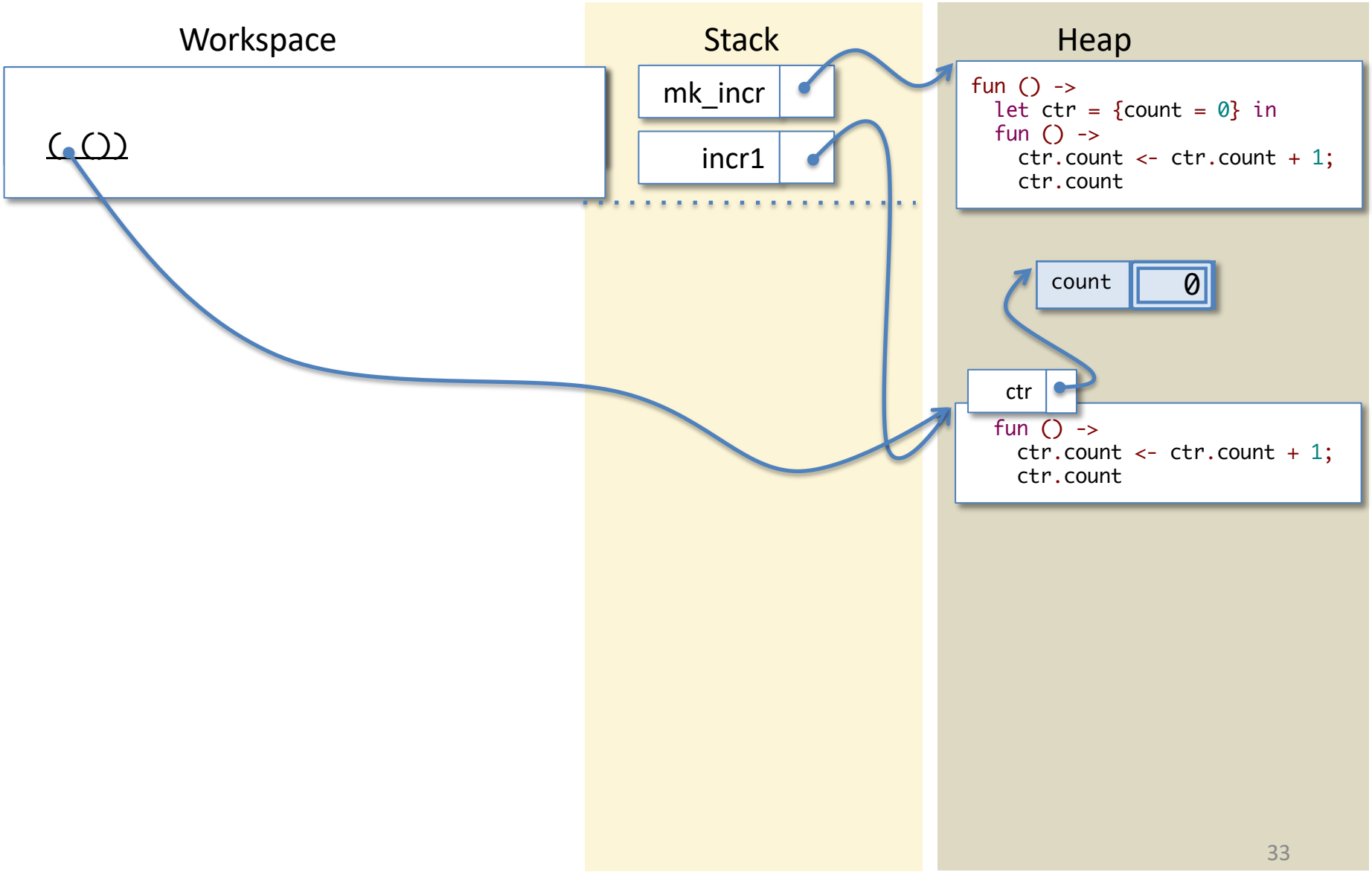
Now let's run "`incr1 ()`"



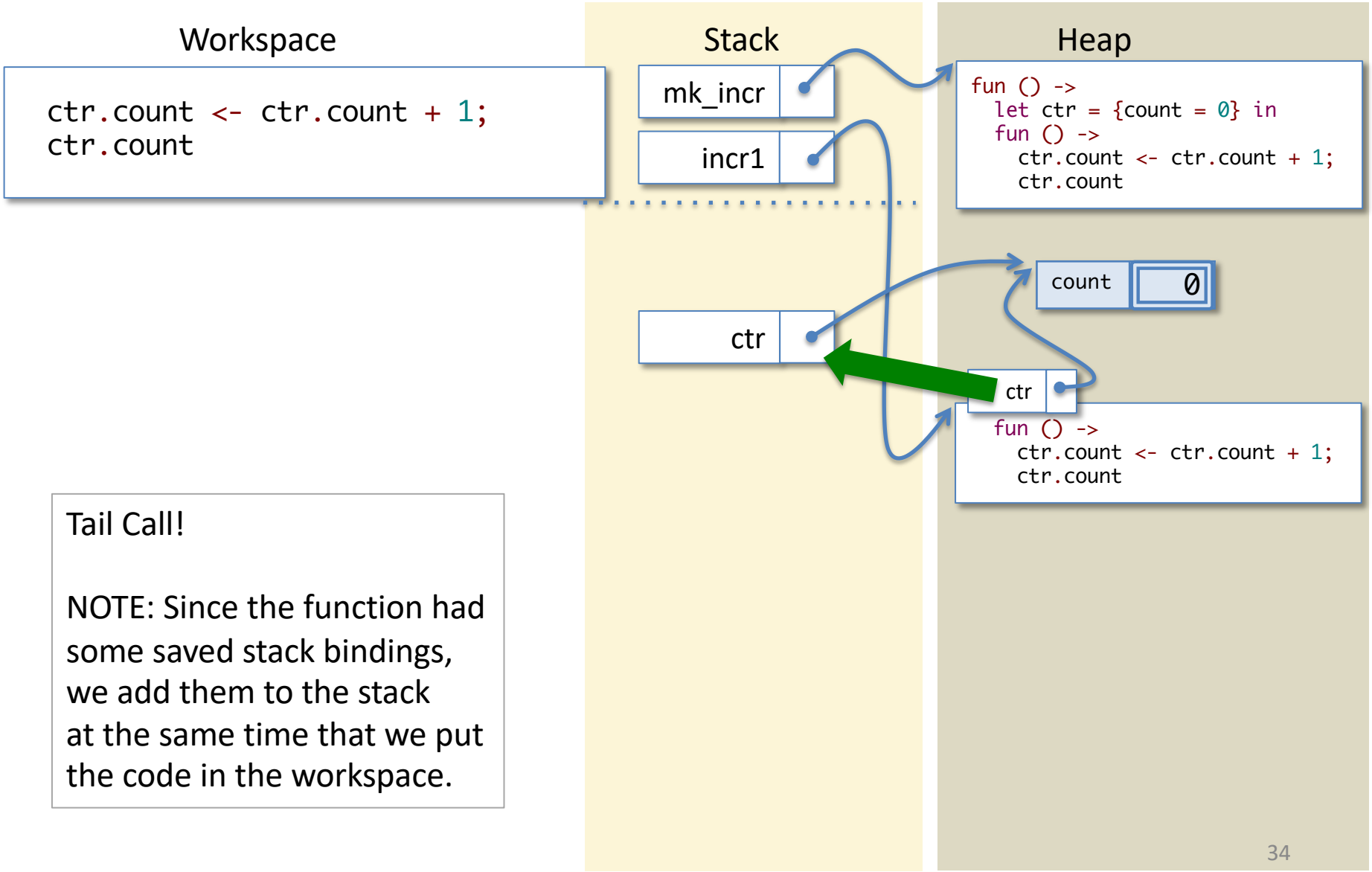
Now let's run "`incr1 ()`"



Now let's run "incr1 ()"



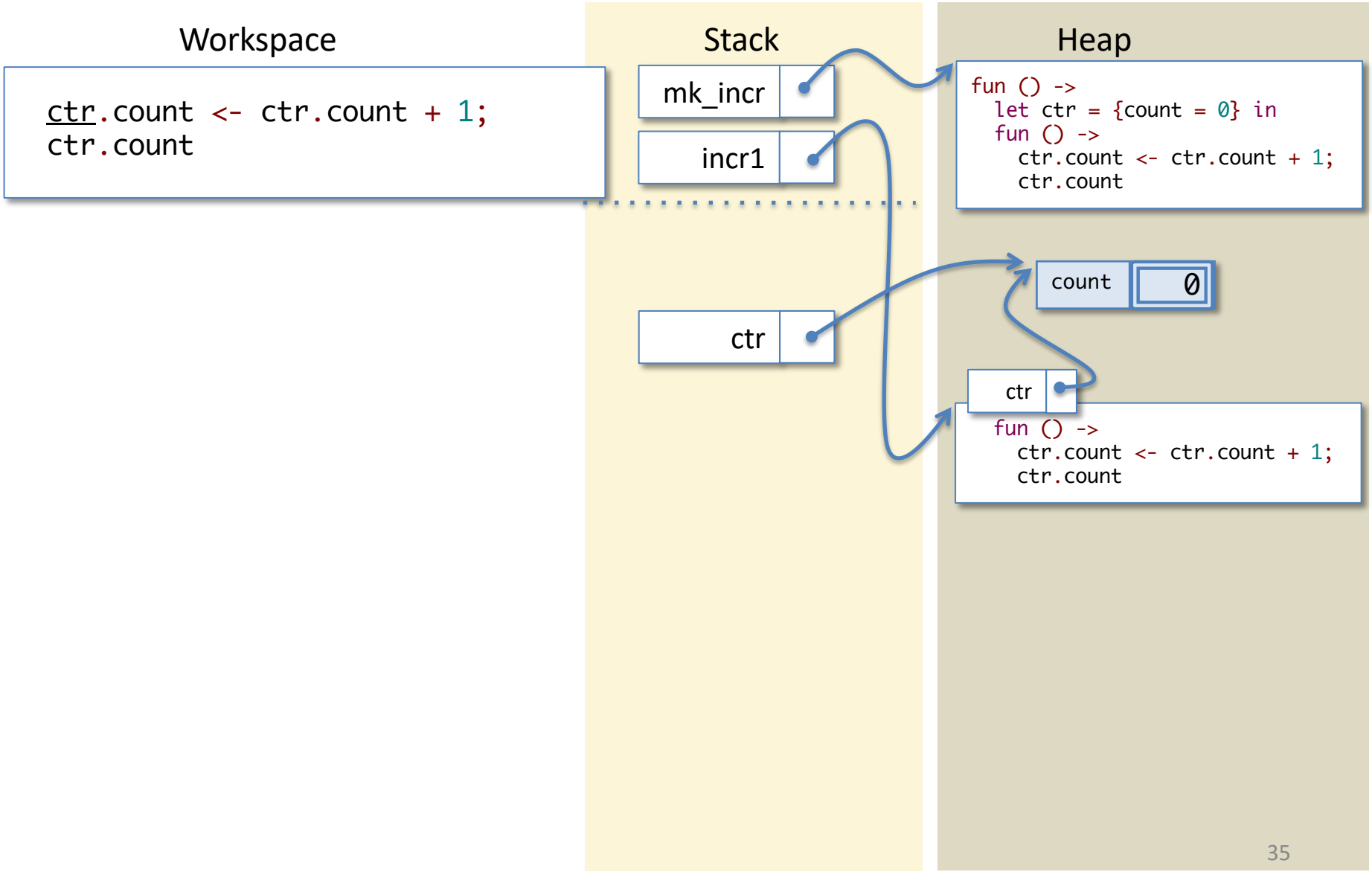
Now let's run "incr1 ()"



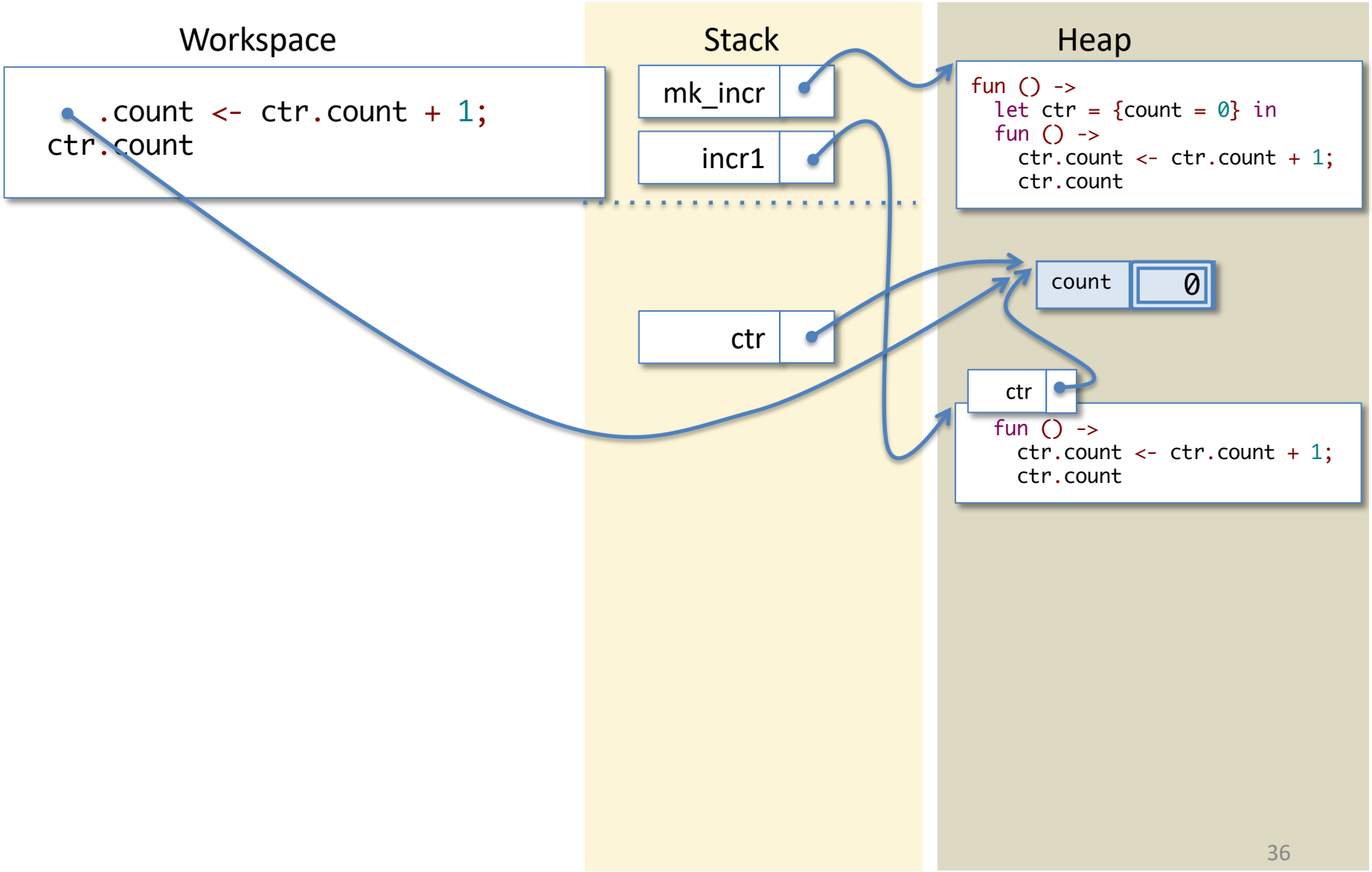
Tail Call!

NOTE: Since the function had some saved stack bindings, we add them to the stack at the same time that we put the code in the workspace.

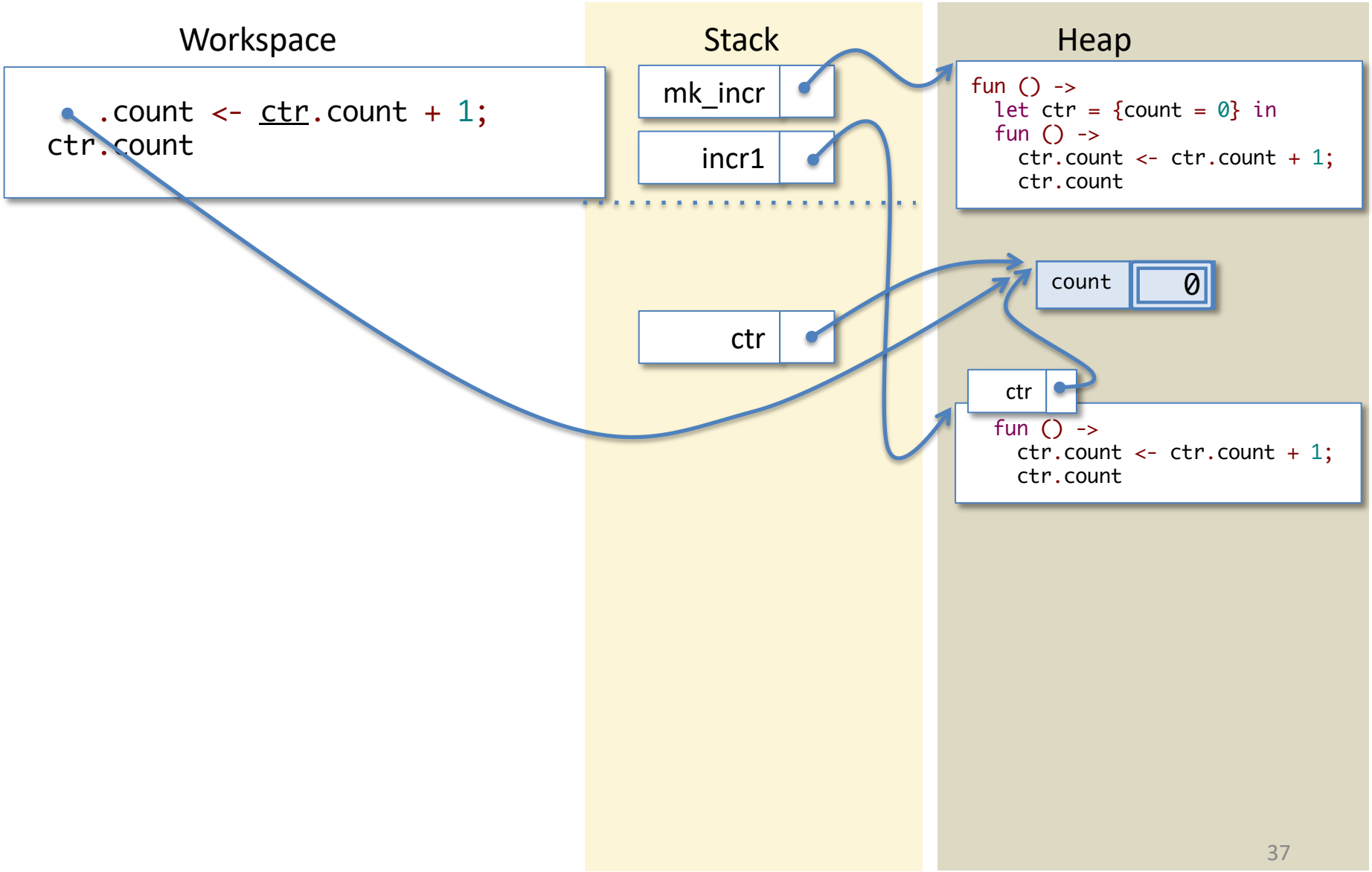
Now let's run "incr1 ()"



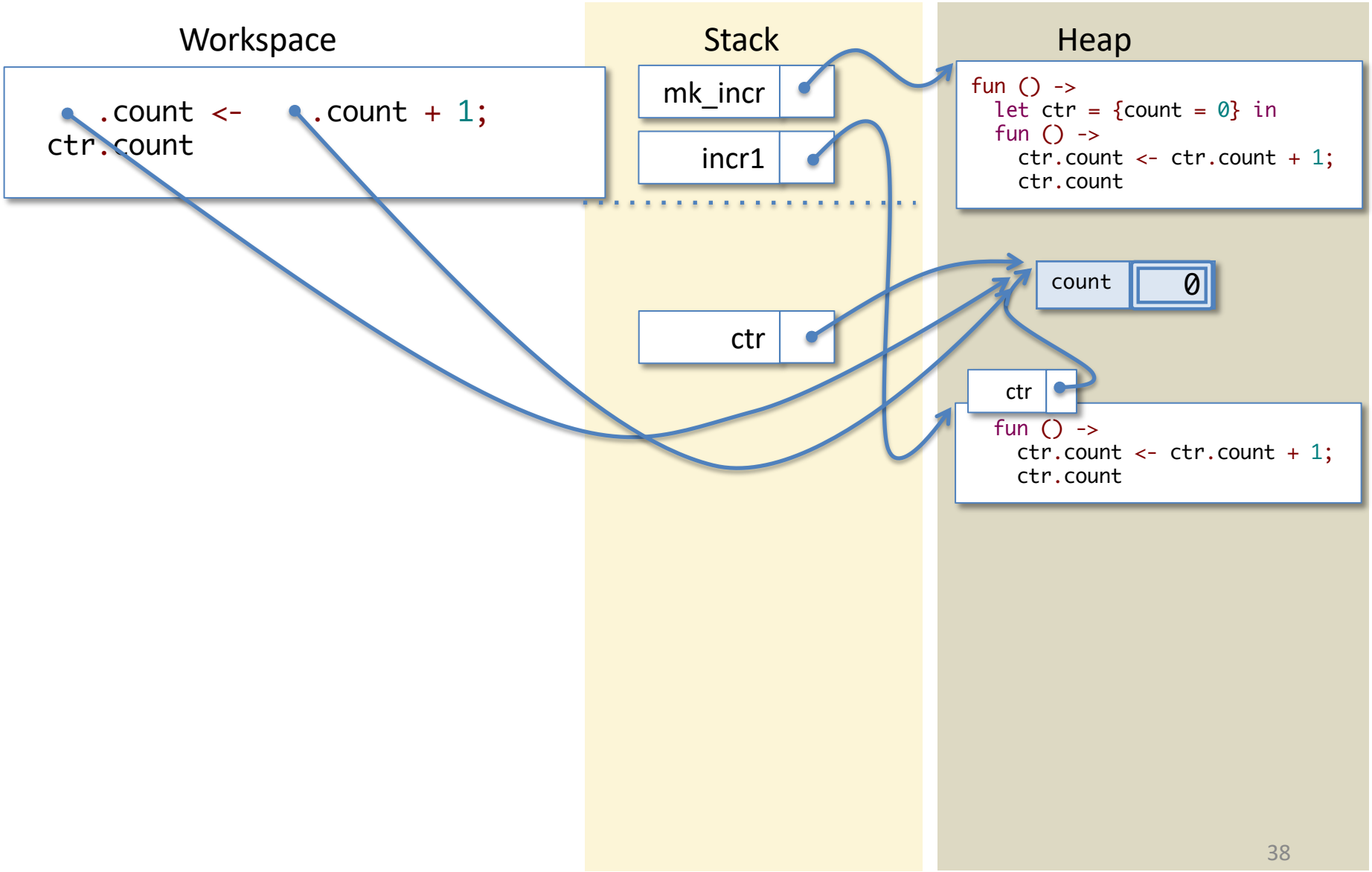
Now let's run "incr1 ()"



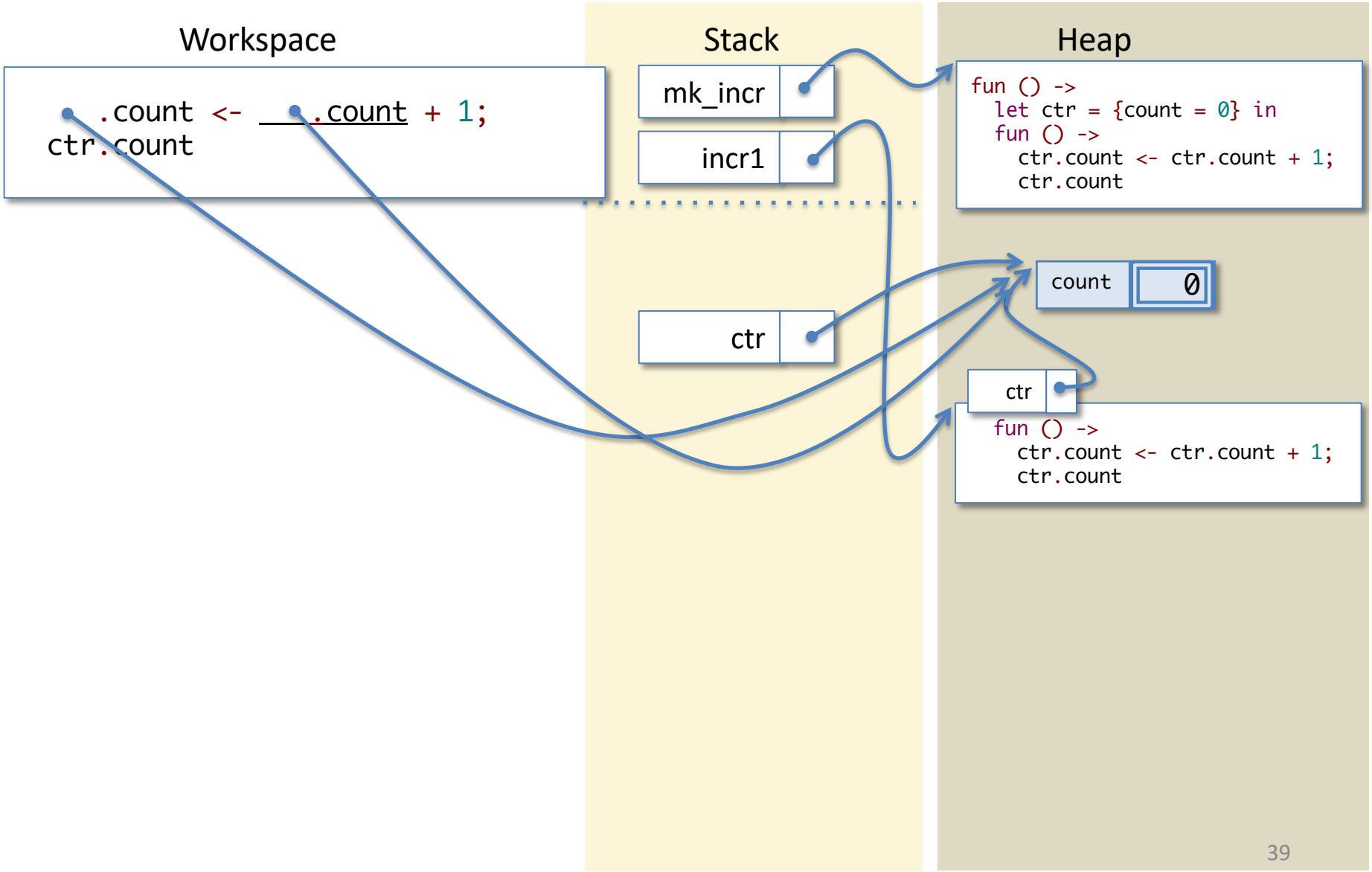
Now let's run "`incr1 ()`"



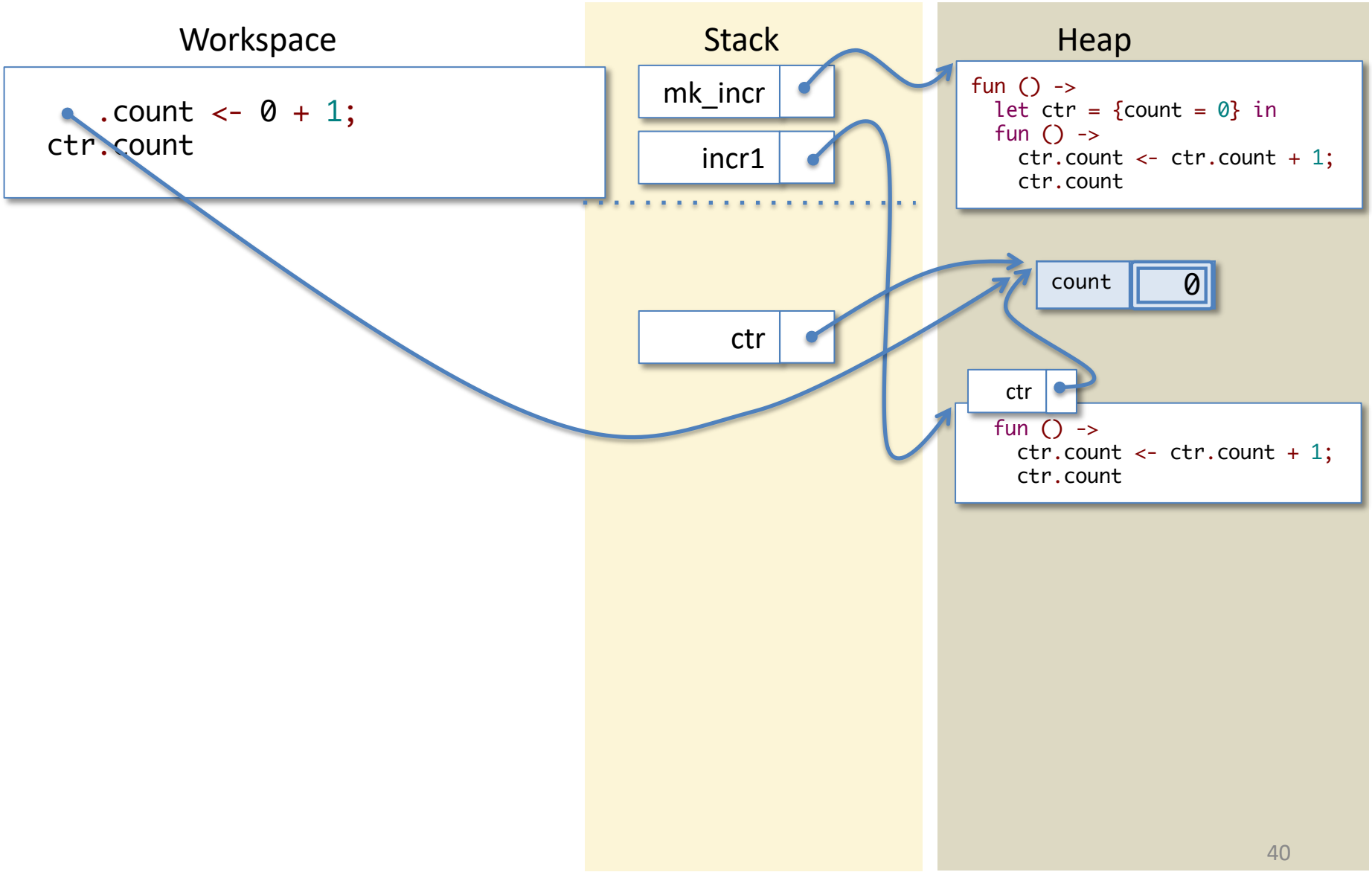
Now let's run "`incr1 ()`"



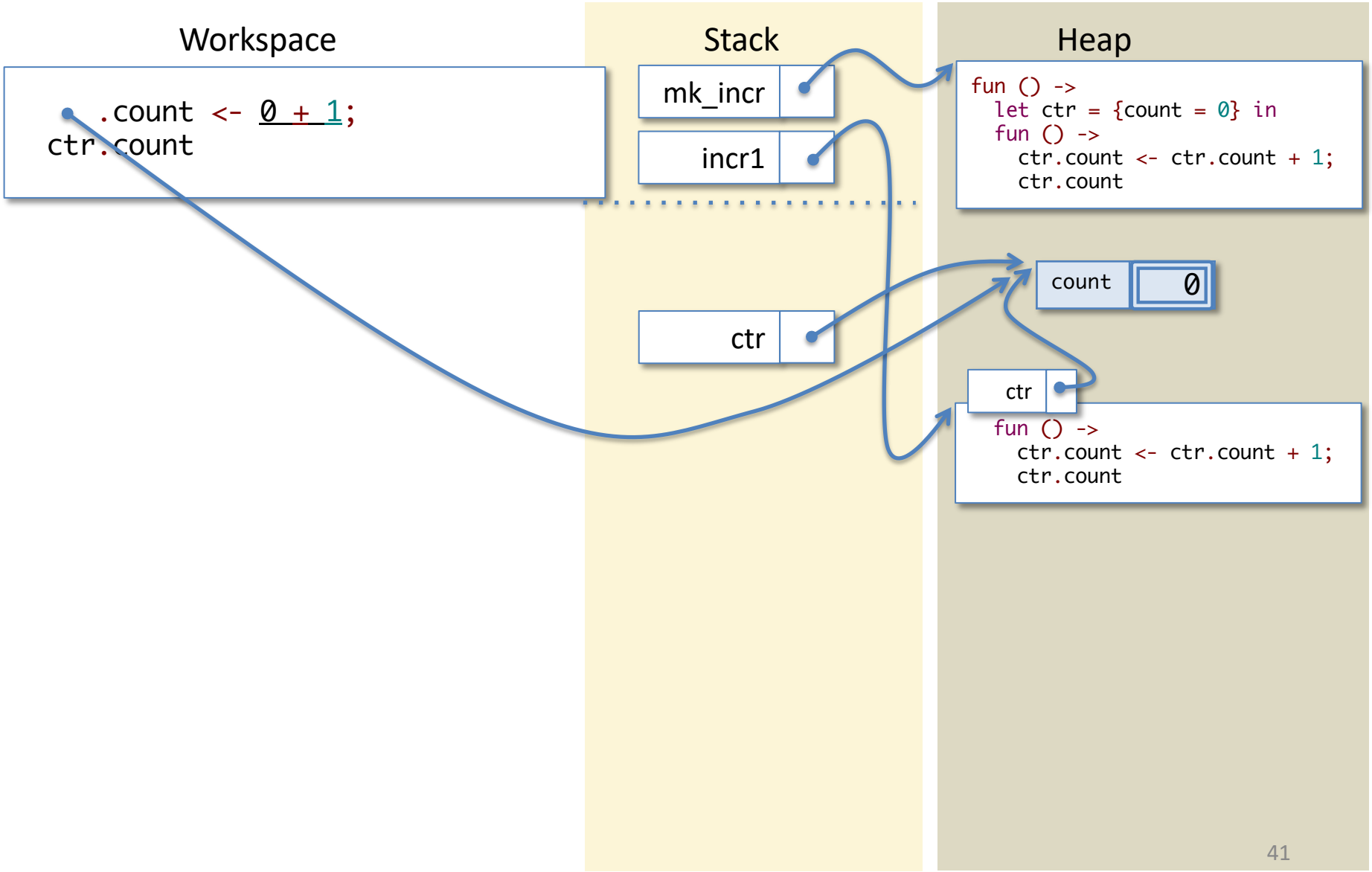
Now let's run "incr1 ()"



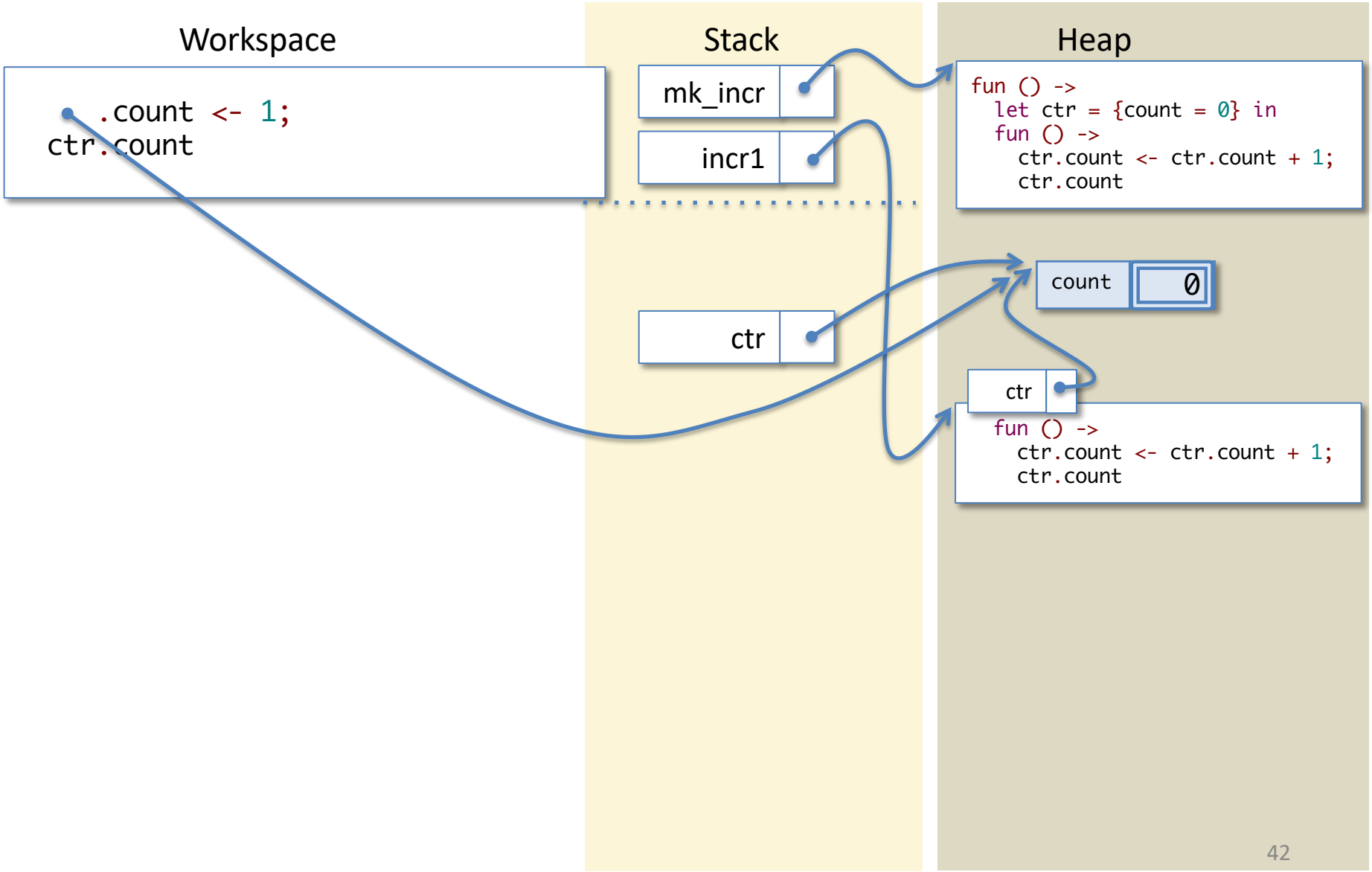
Now let's run "incr1 ()"



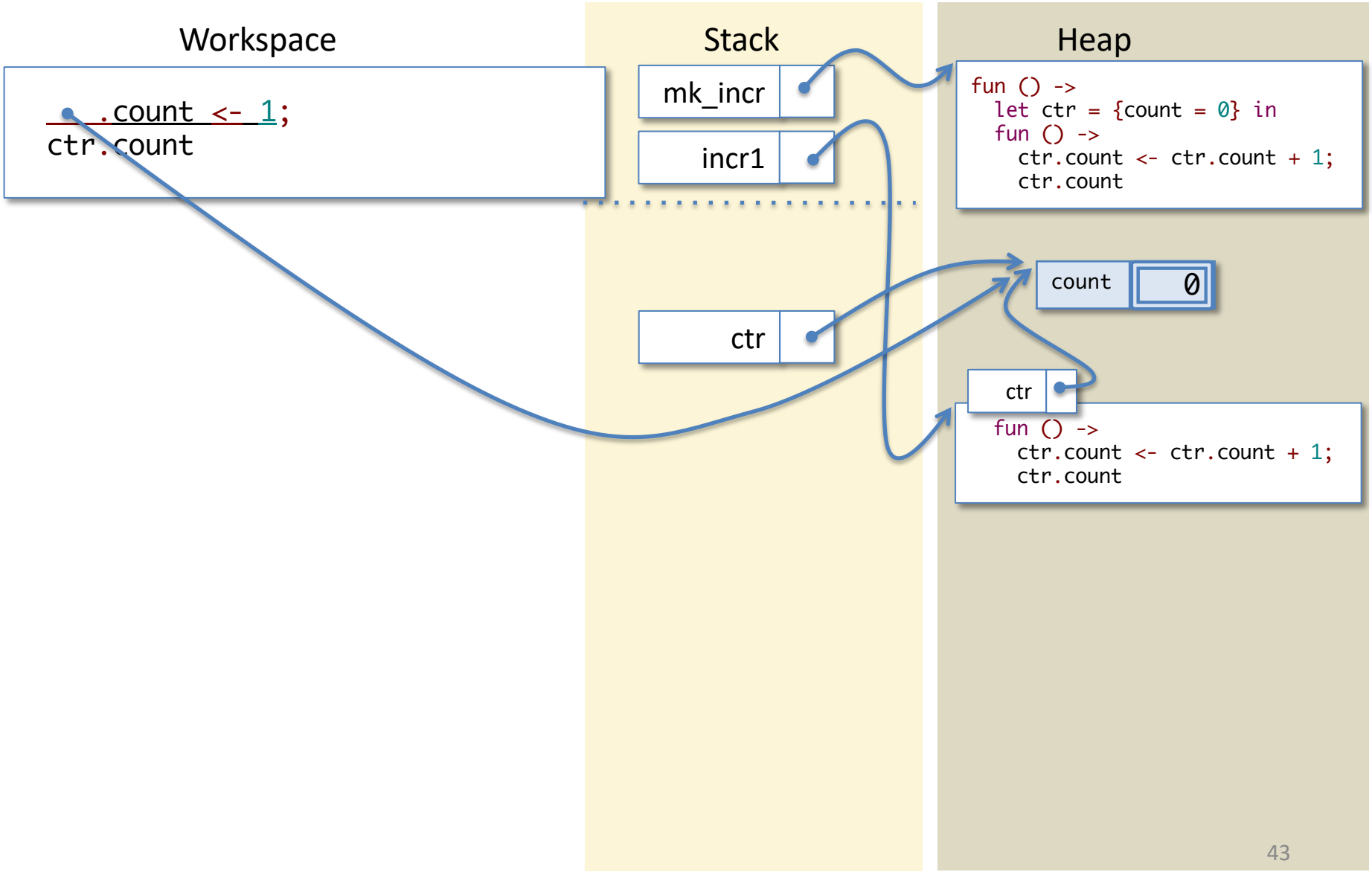
Now let's run "incr1 ()"



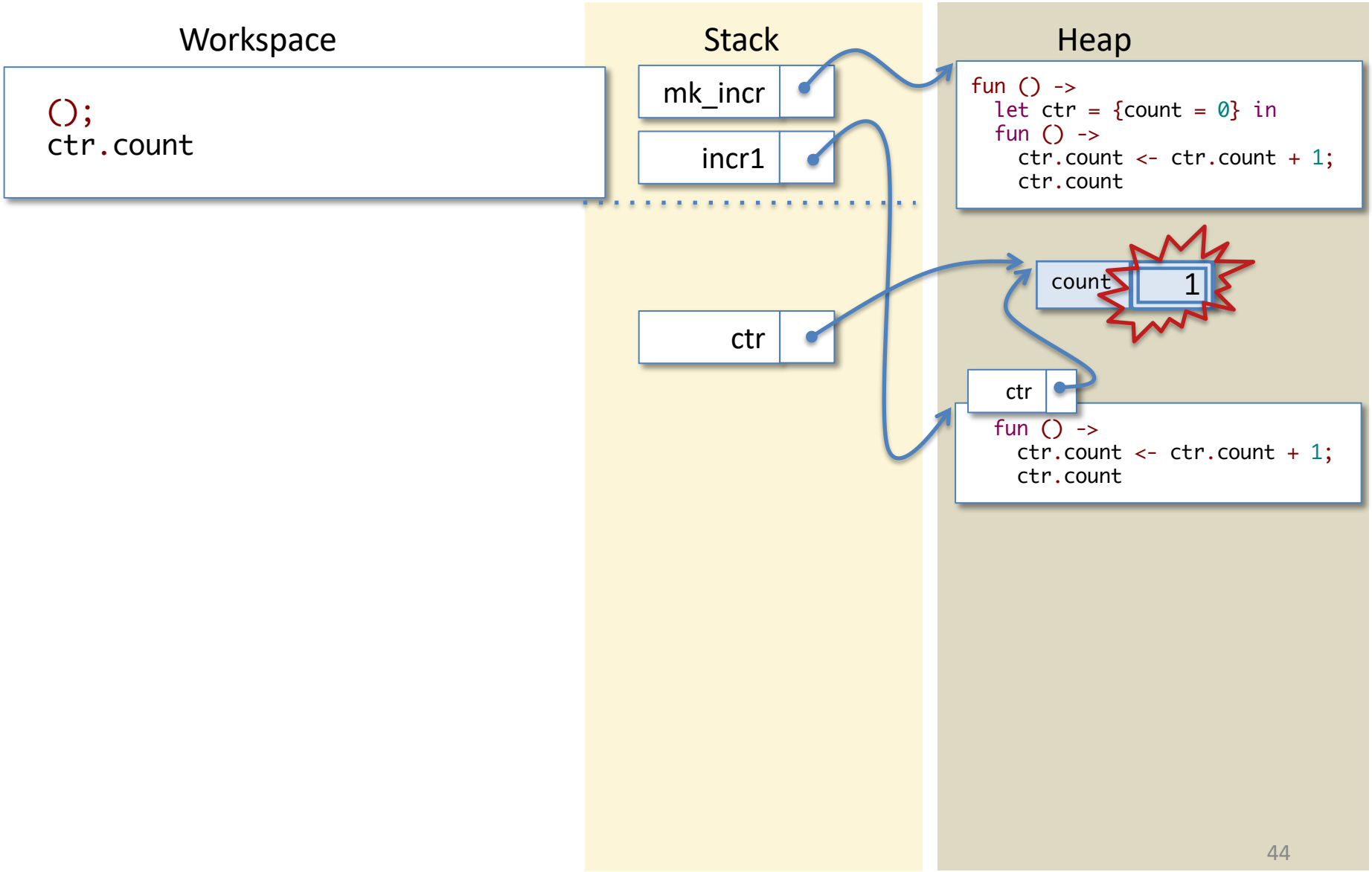
Now let's run "`incr1 ()`"



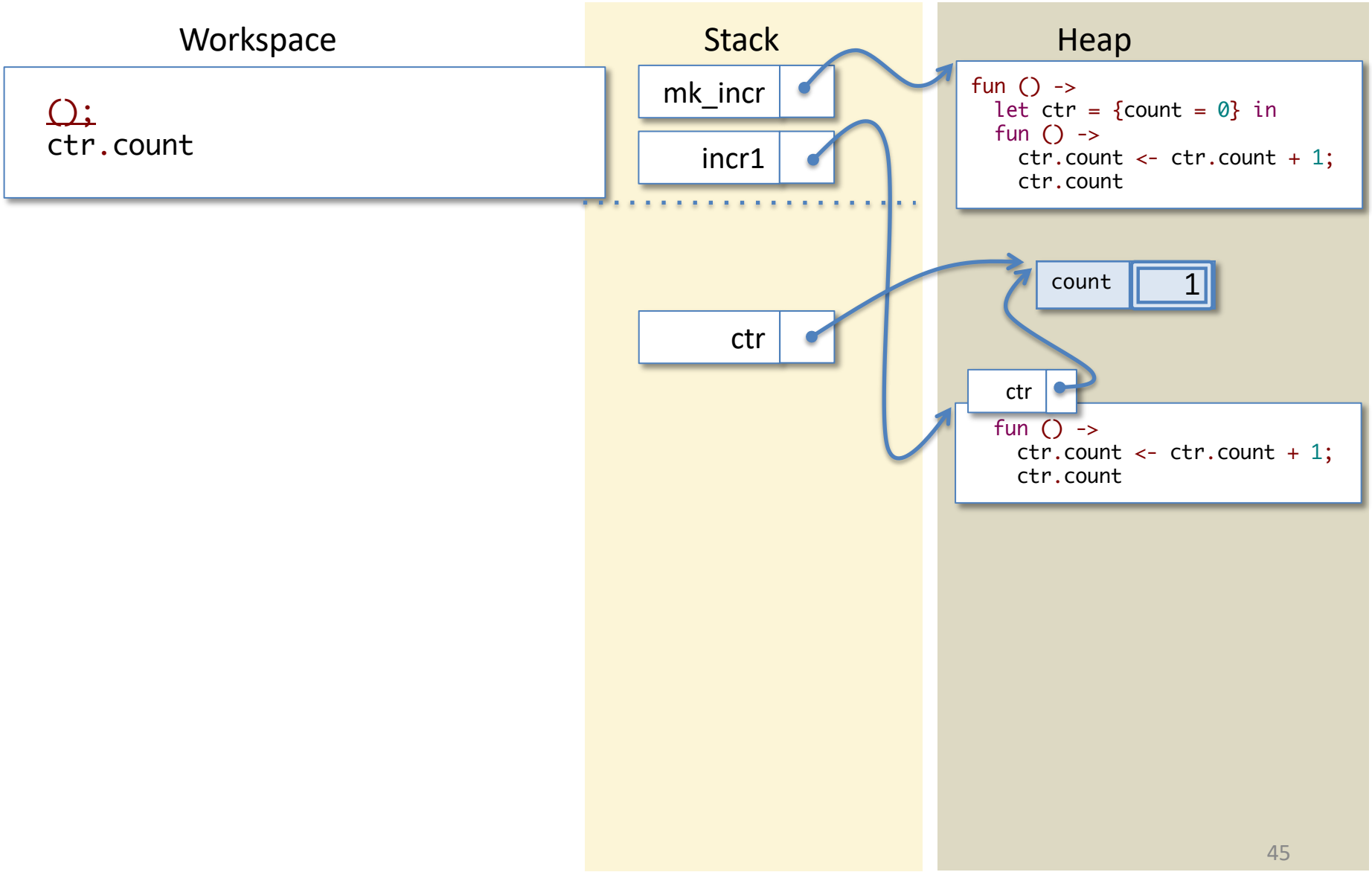
Now let's run "`incr1 ()`"



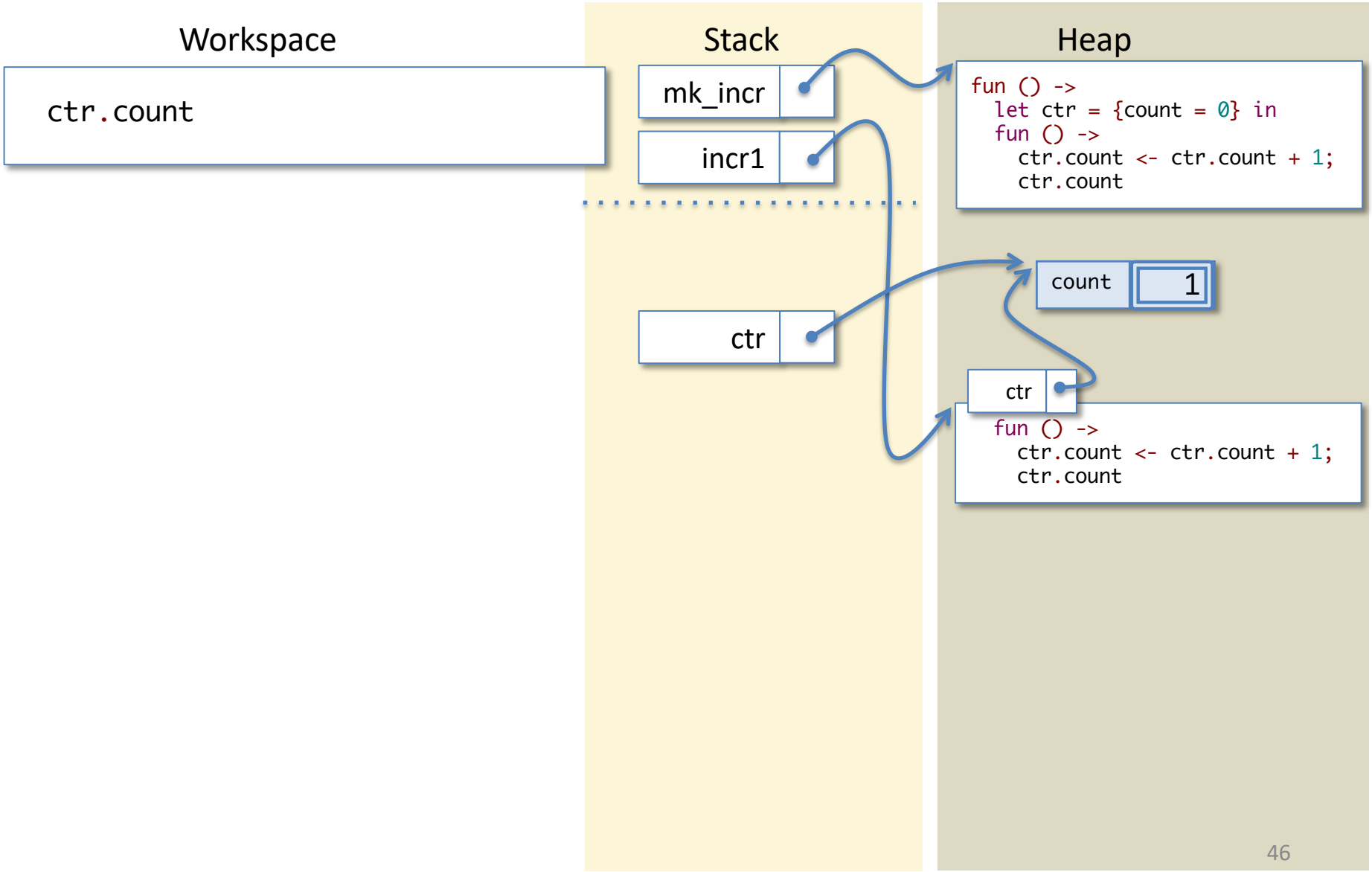
Now let's run "incr1 ()"



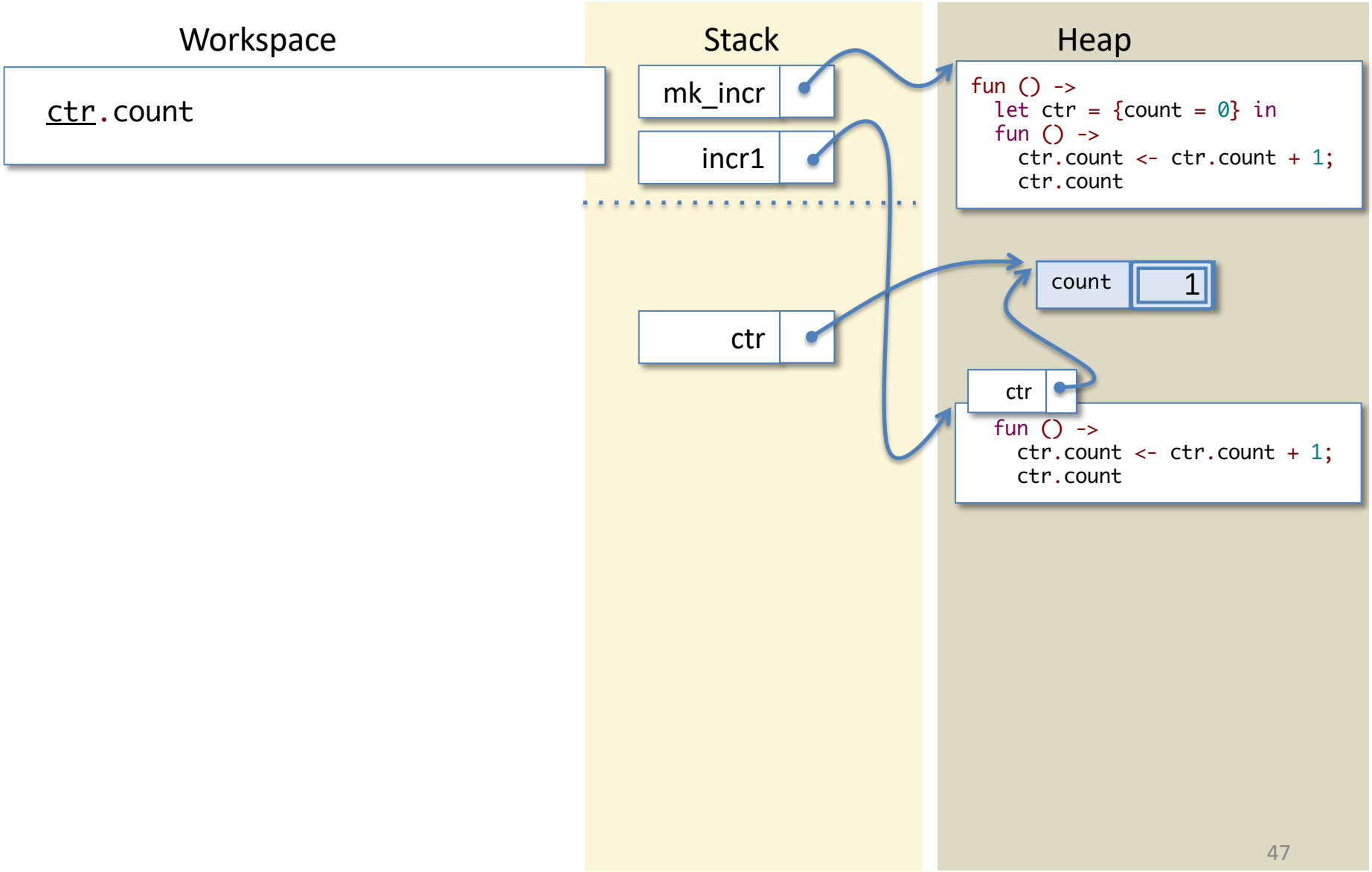
Now let's run "incr1 ()"



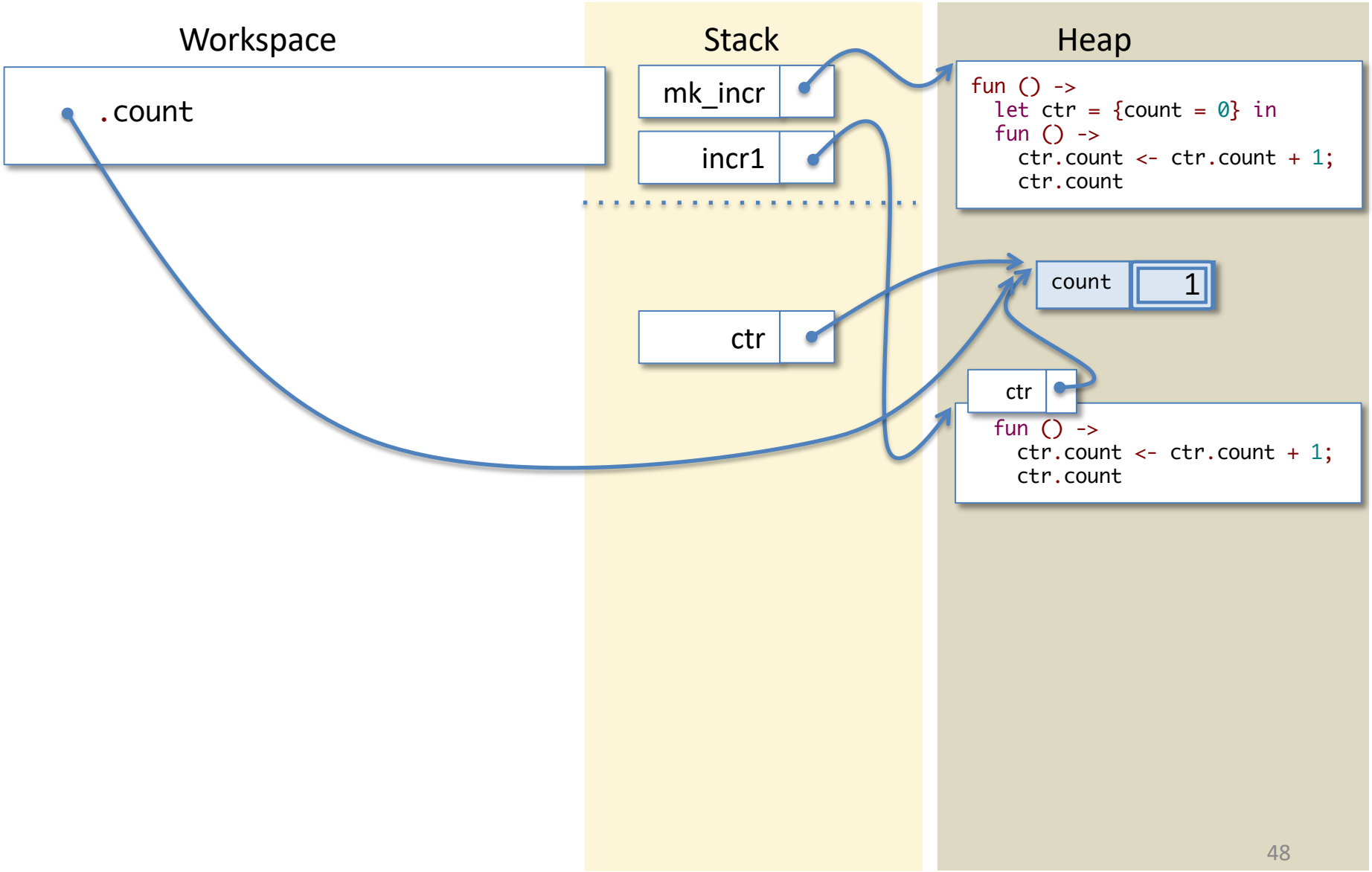
Now let's run "`incr1 ()`"



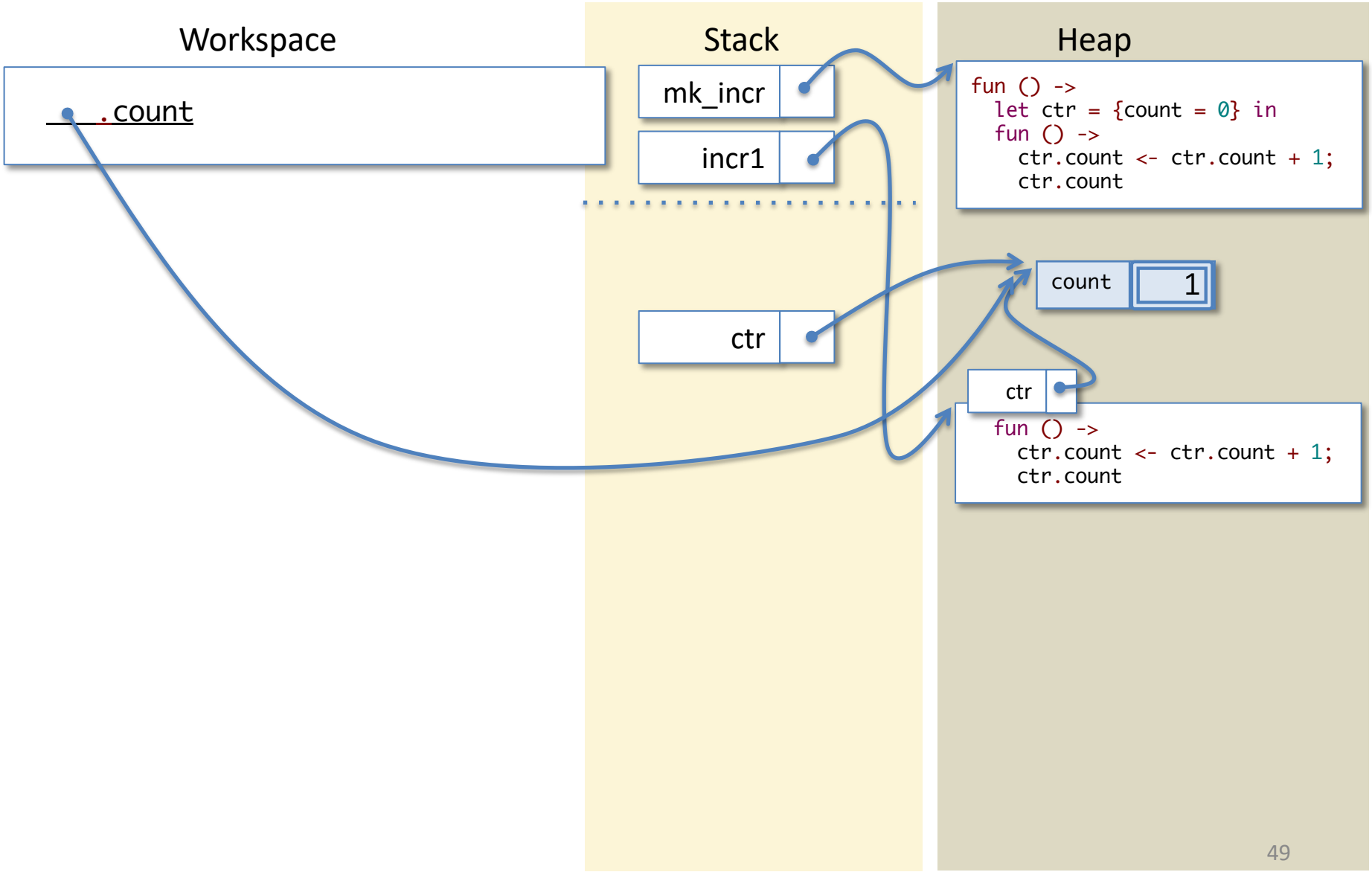
Now let's run "`incr1 ()`"



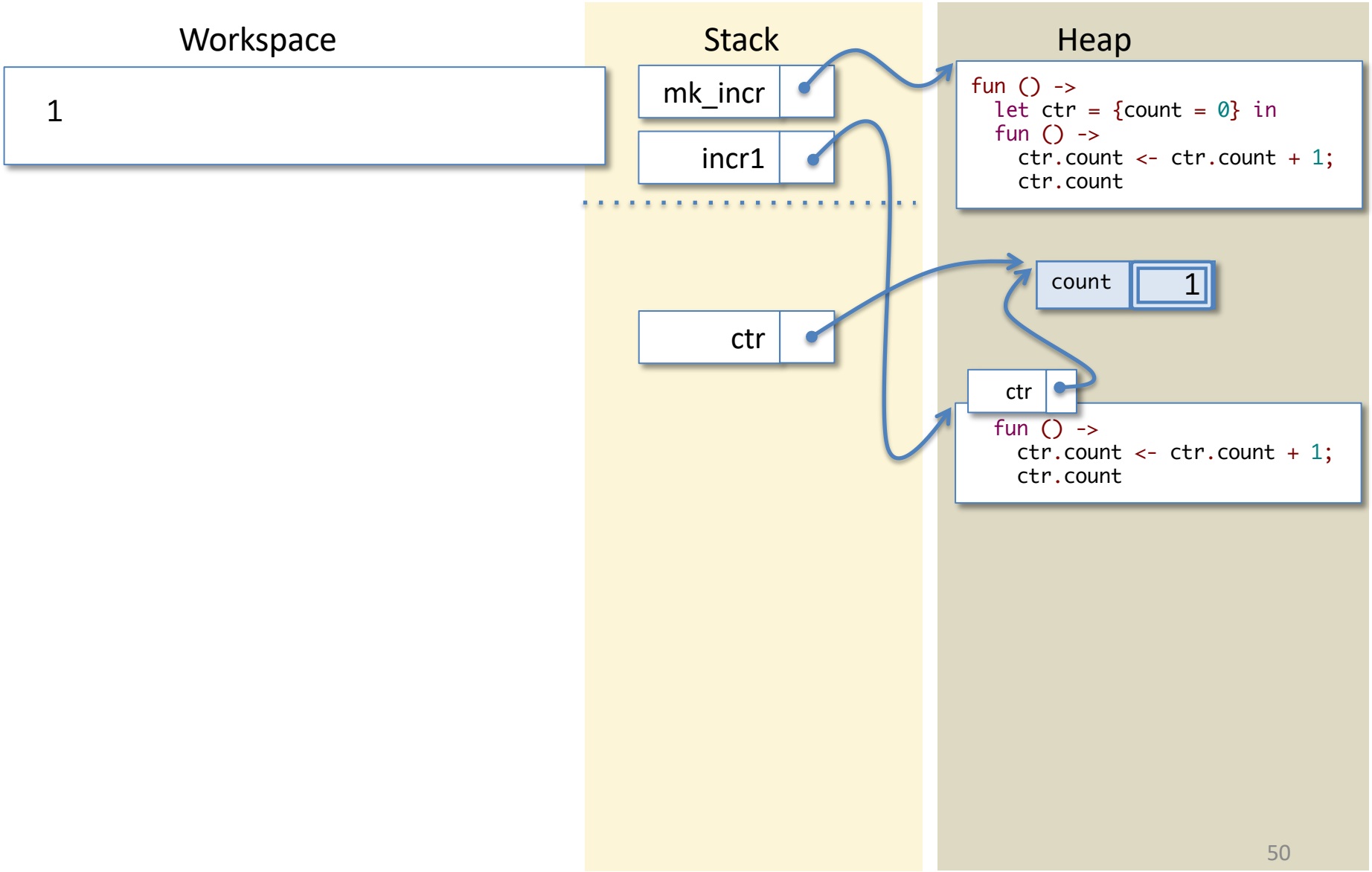
Now let's run "`incr1 ()`"



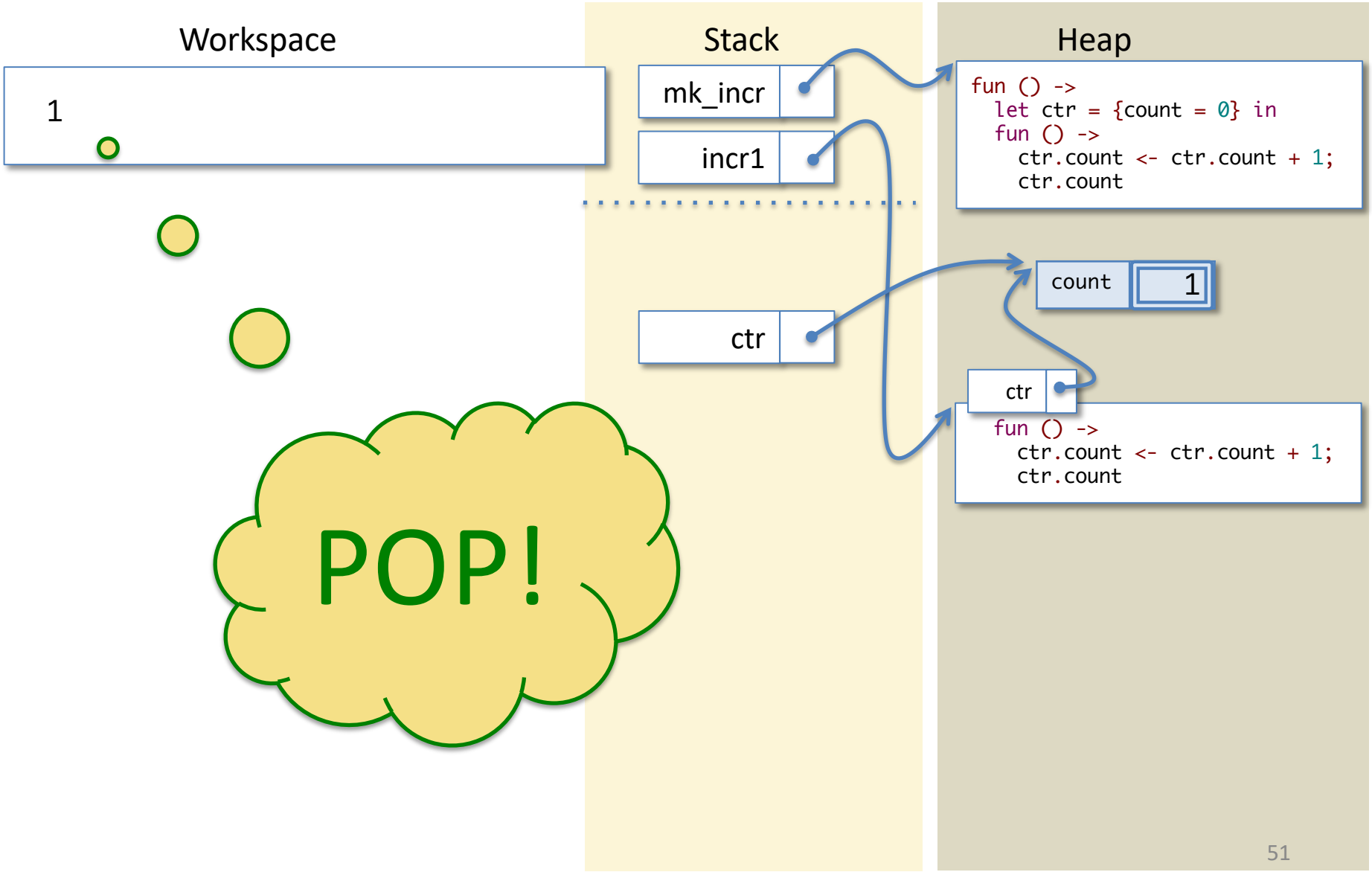
Now let's run "`incr1 ()`"



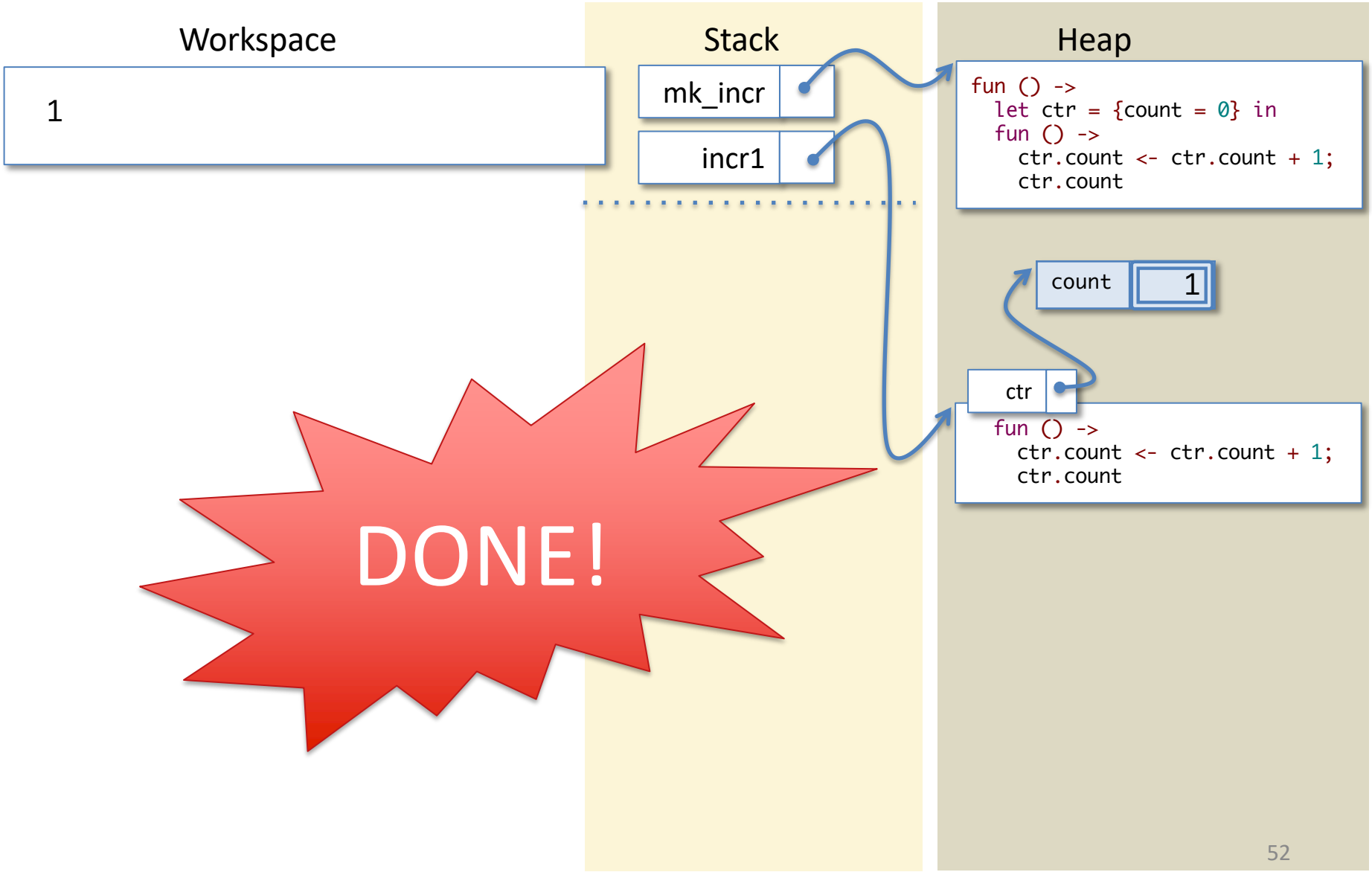
Now let's run "`incr1 ()`"



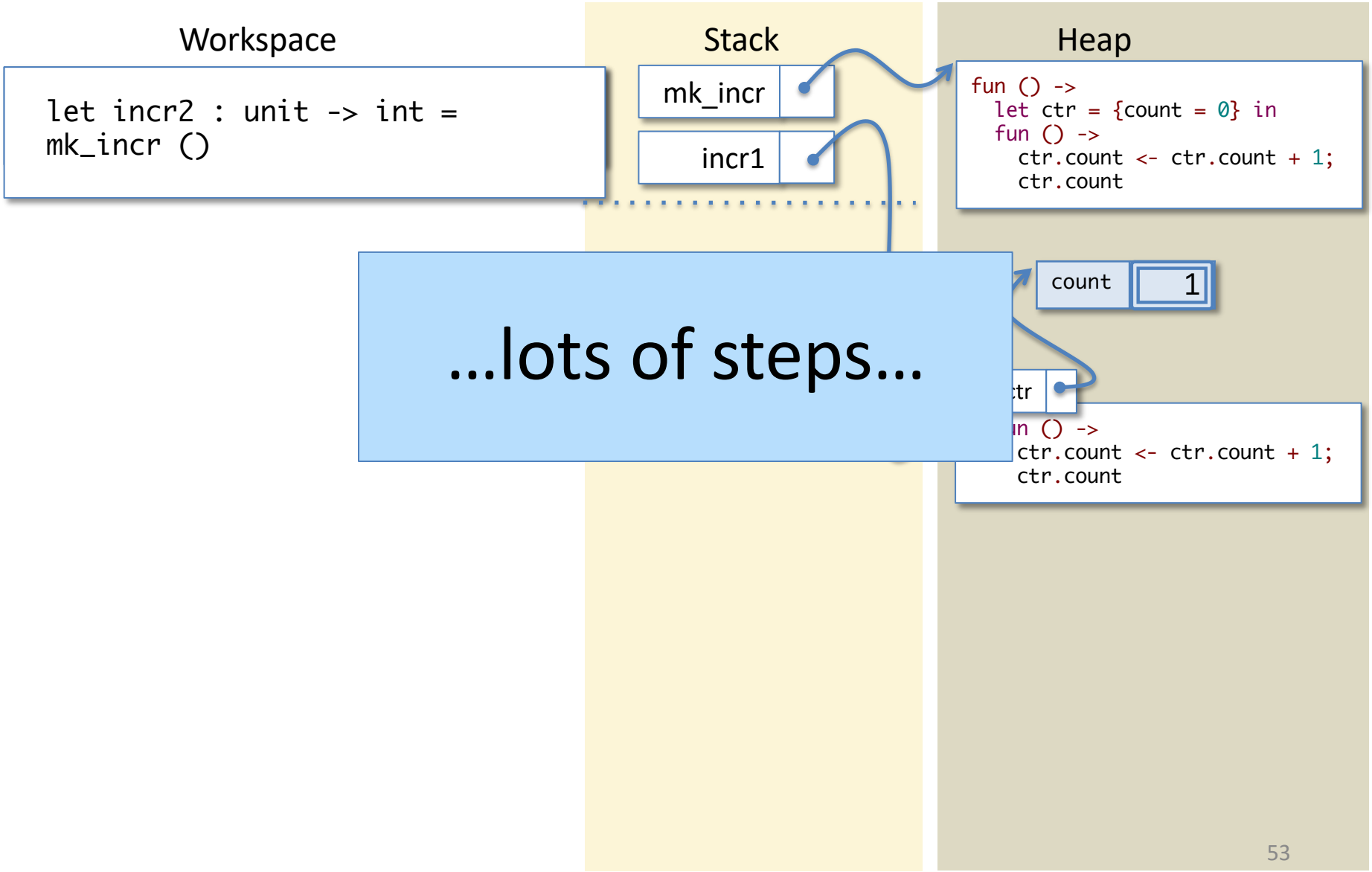
Now let's run "`incr1 ()`"



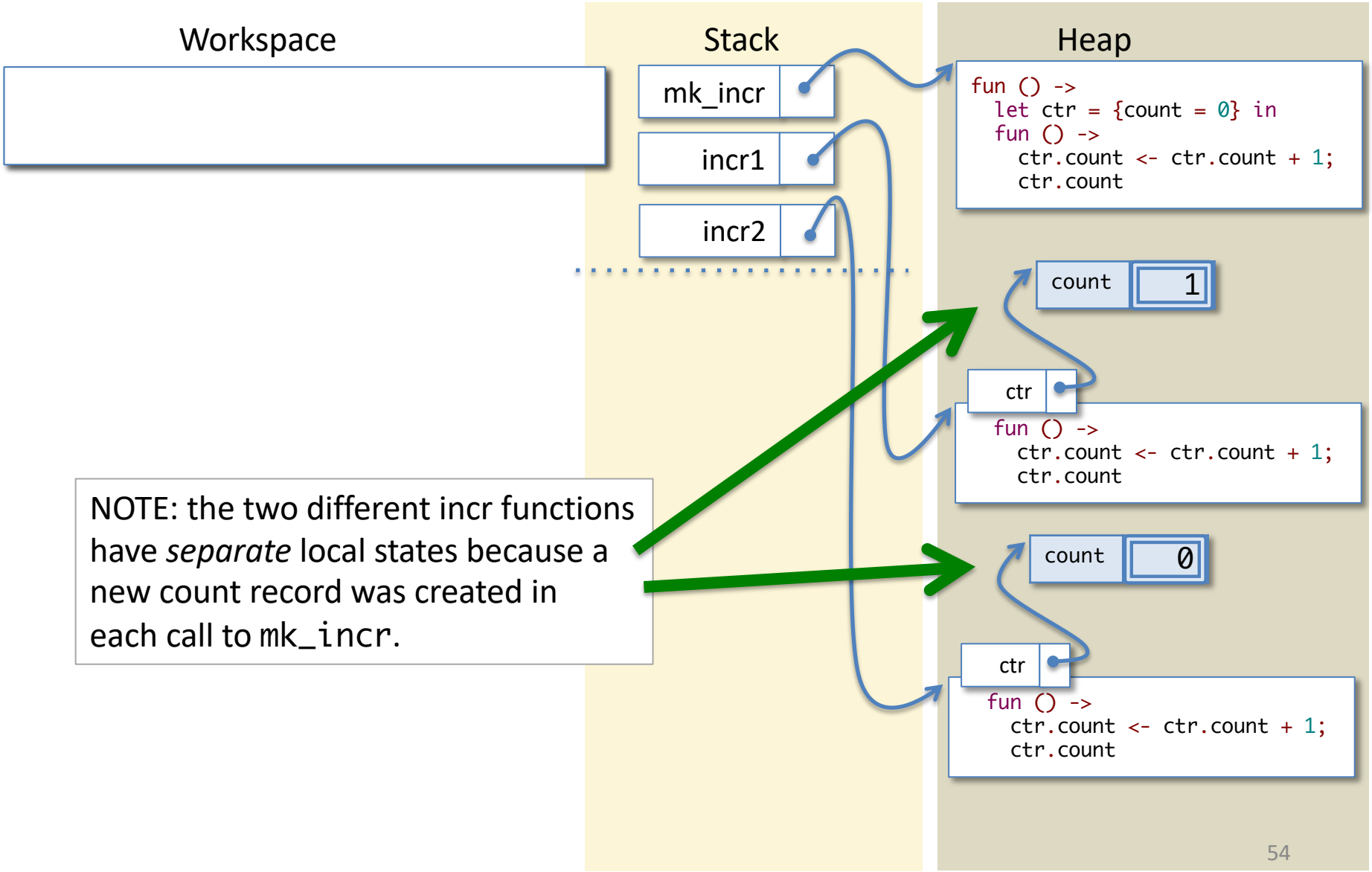
Now let's run "`incr1 ()`"



Now Let's run `mk_incr` again



After creating `incr2`…



Objects

One step further…

- `mk_incr` illustrates how to create different instance of local state so that we can make as many counters as we need
- What if we wanted to bundle together *several* operations that share the same local state?
  - e.g. incr and decr operations that work on the same counter

Key Concept: *Object*

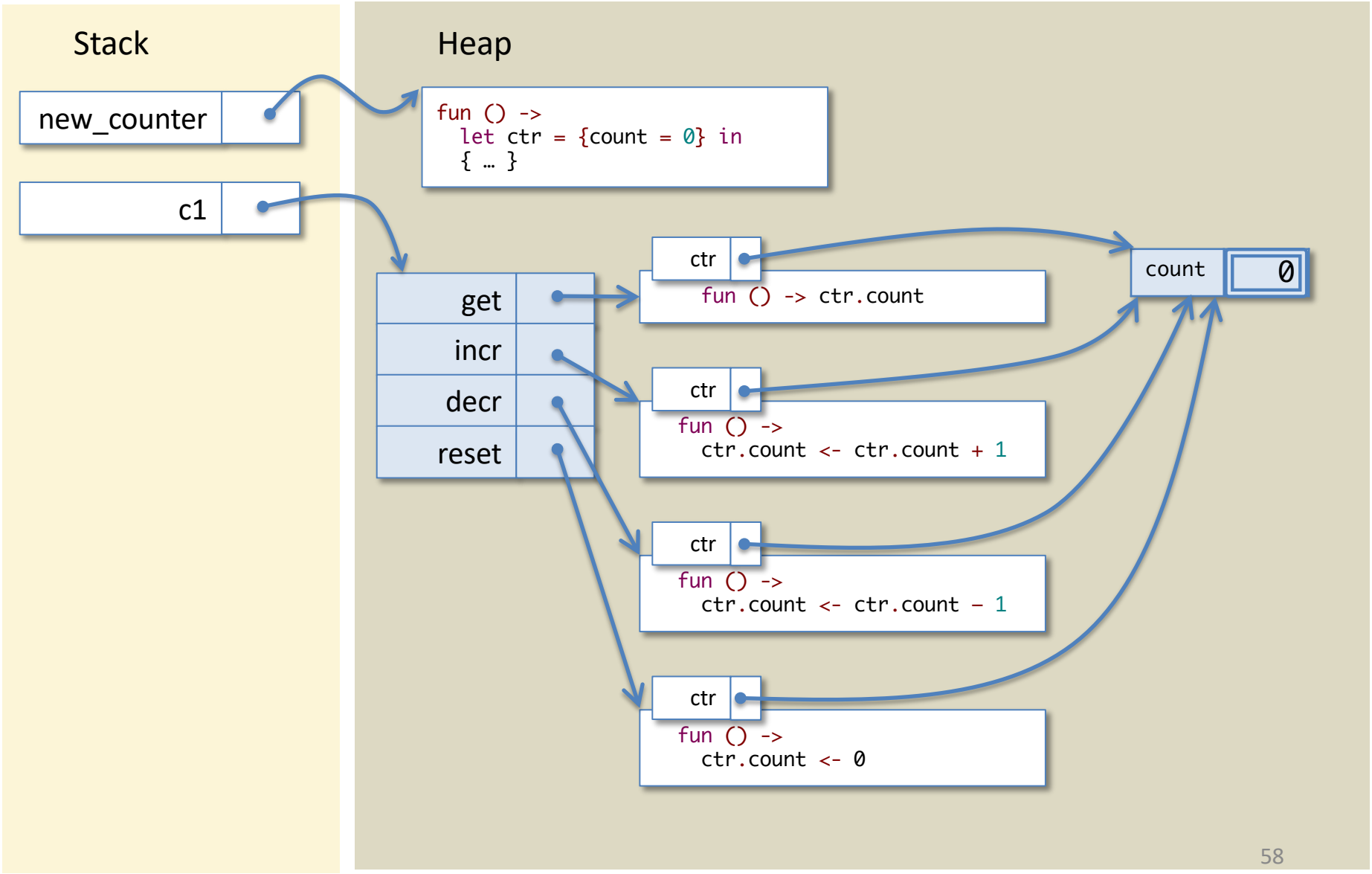
An object consists of:

- some encapsulated mutable state (*fields*)
- a set of operations that manipulate that state (*methods*)

A Counter *Object*

```
(* The type of counter objects *)
type counter = {
    get   : unit -> int;
    incr  : unit -> unit;
    decr  : unit -> unit;
    reset : unit -> unit;
}

(* Create a fresh counter object with hidden state: *)
let new_counter () : counter =
  let ctr = {count = 0} in
  {
   get   = (fun () -> ctr.count) ;
   incr  = (fun () -> ctr.count <- ctr.count + 1) ;
   decr  = (fun () -> ctr.count <- ctr.count - 1) ;
   reset = (fun () -> ctr.count <- 0) ;
  }
```

`let c1 = new_counter ()`



Using Counter Objects

```
(* a helper function to create a nice string for printing *)
let ctr_string (s:string) (i:int) =
    s ^ ".ctr = " ^ (string_of_int i) ^ "\n"

let c1 = new_counter ()
let c2 = new_counter ()

;; print_string (ctr_string "c1" (c1.get ()))
;; c1.incr ()
;; c1.incr ()
;; print_string (ctr_string "c1" (c1.get ()))
;; c1.decr ()
;; print_string (ctr_string "c1" (c1.get ()))
;; c2.incr ()
;; print_string (ctr_string "c2" (c2.get ()))
;; c2.decr ()
;; print_string (ctr_string "c2" (c2.get ()))
```

Typechecking Revisited

How does OCaml* typecheck your code?

*Historical aside: the algorithm we are about to see is known as the Damas-Hindley-Milner type inference algorithm. Turing Award winner Robin Milner was, among other things, the inventor of "ML" (for "meta language"), from which OCaml gets its "ml".

OCaml Typechecking Errors

```
115   let length (q: 'a queue) : int =
116     let rec loop (qn : 'a qnode option) (acc: int) : int
117   This expression has type 'a list
          but an expression was expected of type int
118           | None -> []
119           | Some n -> loop n.next (1 + acc)
120
121     in
122     lo
```

```
type ('k,'v) map = ('k * 'v) list

(* A finite map that contains no entries. *)
let empty () = []

let rec mem
  begin ma
  | [] ->
  | (k,v):
    if key
    (key = k) || (mem key rest)
  end

;; run_test "mem test" (fun () ->
  mem "b" [("a",3); ("b",4)]
)
```

```
Signature mismatch:
       ...
       Values do not match:
         val empty : unit -> 'a list
       is not included in
         val empty : ('k, 'v) map
       File "finiteMap.ml", line 13, characters 2-27: Expected
declaration
       File "finiteMap.ml", line 60, characters 6-11: Actual declaration
```

Typechecking

How do you determine the type of an expression?

1. Recursively determine the types of *all* of the sub-expressions
   - Some expressions have "obvious" types:

     `3 : int` `"foo" : string` `true : bool`
   - Identifiers have the types assigned where they are bound
     - `let` and function arguments have type annotations
     - Or, take the types from the module signature

2. Expressions that *construct* structured values have compound types built from the types of sub-expressions:

```
(3, "foo")                        : int * string
(fun (x:int) -> x + 1)            : int -> int
Node(Empty, (3, "foo"), Empty) : (int * string) tree
```

Typechecking II

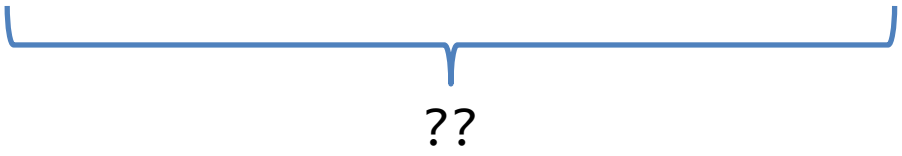
3. The type of a function-application expression is obtained as the result from the function type:
   - Given an expression of function type f : T_1 -> T_2
   - and an argument expression e : T_1 (of the input type)
   - (f e) : T_2

```
((fun (x:int) (y:bool) -> y)  3)  : ??
```

Typechecking II

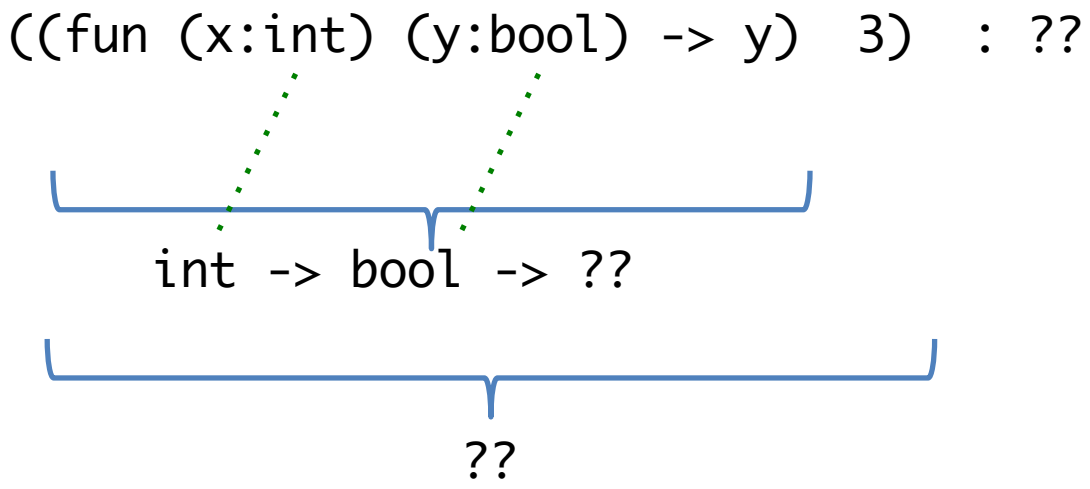
3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function f : T_1 -> T_2
   - and an argument e : T_1 of the input type
   - (f e) : T_2

```
((fun (x:int) (y:bool) -> y)  3)  : ??
```



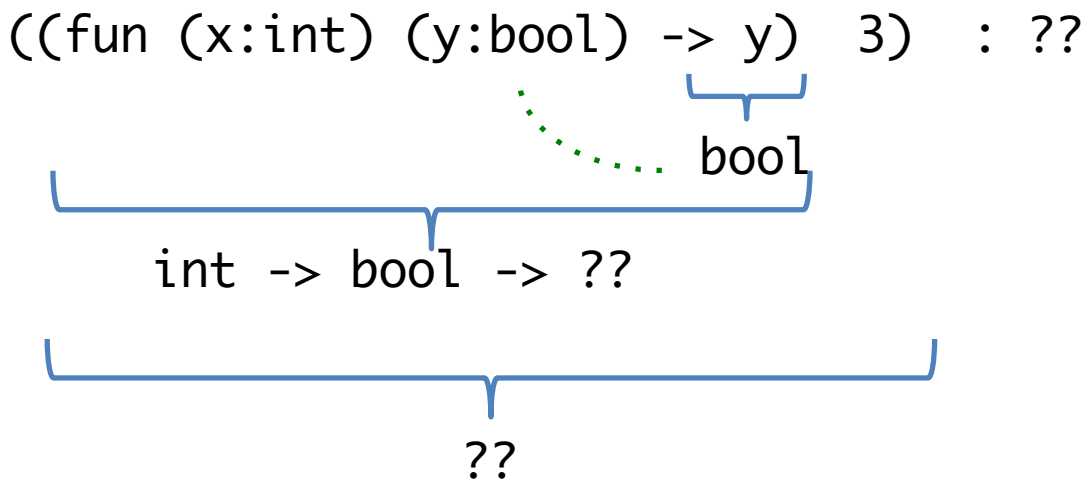
Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function f : T_1 -> T_2
   - and an argument e : T_1 of the input type
   - (f e) : T_2



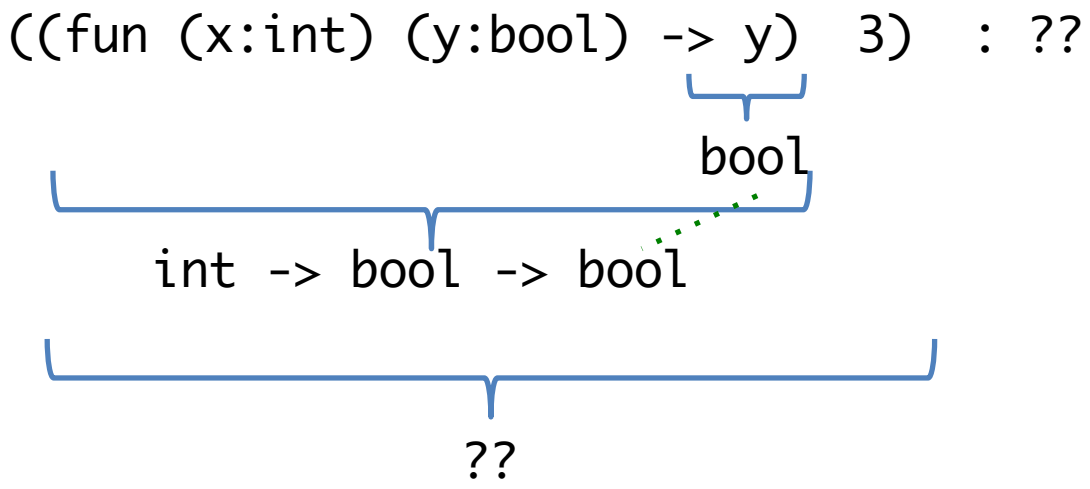
Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function f : T_1 -> T_2
   - and an argument e : T_1 of the input type
   - (f e) : T_2



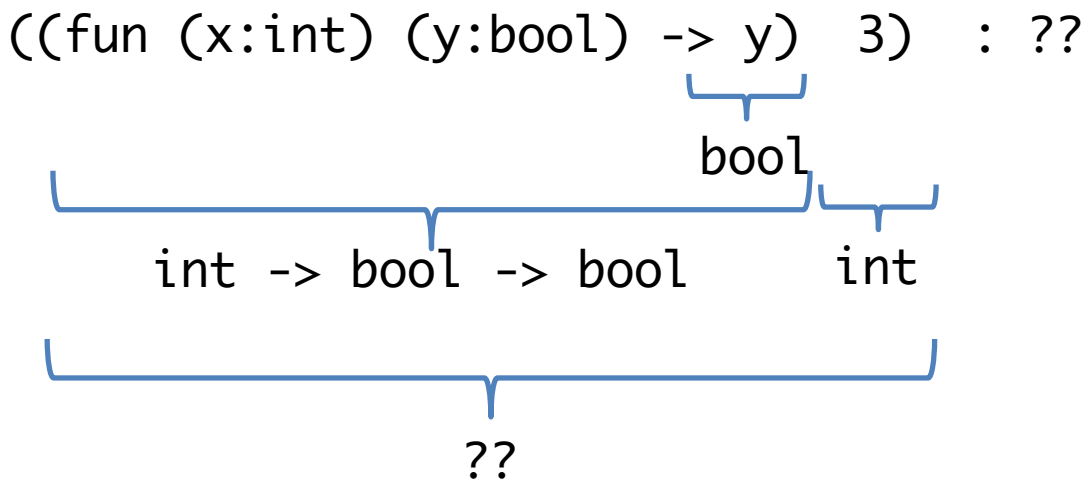
Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function f : T_1 -> T_2
   - and an argument e : T_1 of the input type
   - (f e) : T_2



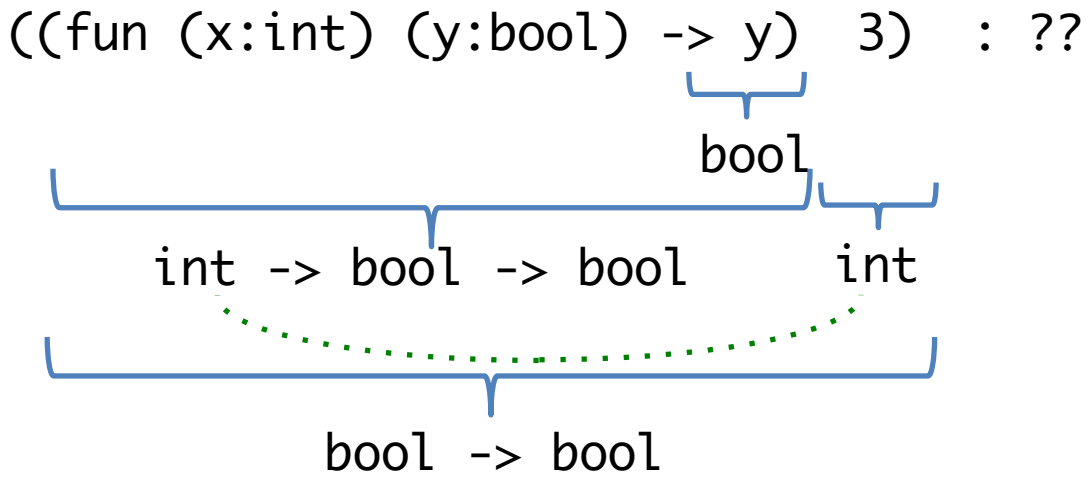
Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function f : T_1 -> T_2
   - and an argument e : T_1 of the input type
   - (f e) : T_2



Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function f : T_1 -> T_2
   - and an argument e : T_1 of the input type
   - (f e) : T_2



Here:
T_1 = int
T_2 = bool -> bool

Typechecking III

- For generic expressions:
  - *Unify* the types based on use:
  - Given a function f : T_1 -> T_2
  - and an argument e : U_1 of the input type

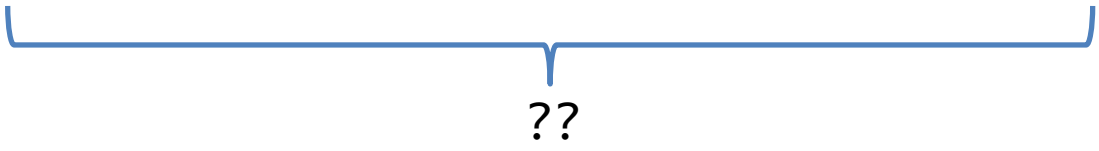
    - *"match up"* T_1 and U_1 to obtain information about type parameters in T_1 and U_1 based on the their usage

  - Obtain an *instantiation*: e.g. `'a = int list`
  - *Propagate* that information to all occurrences of `'a`

Example Typechecking Problem

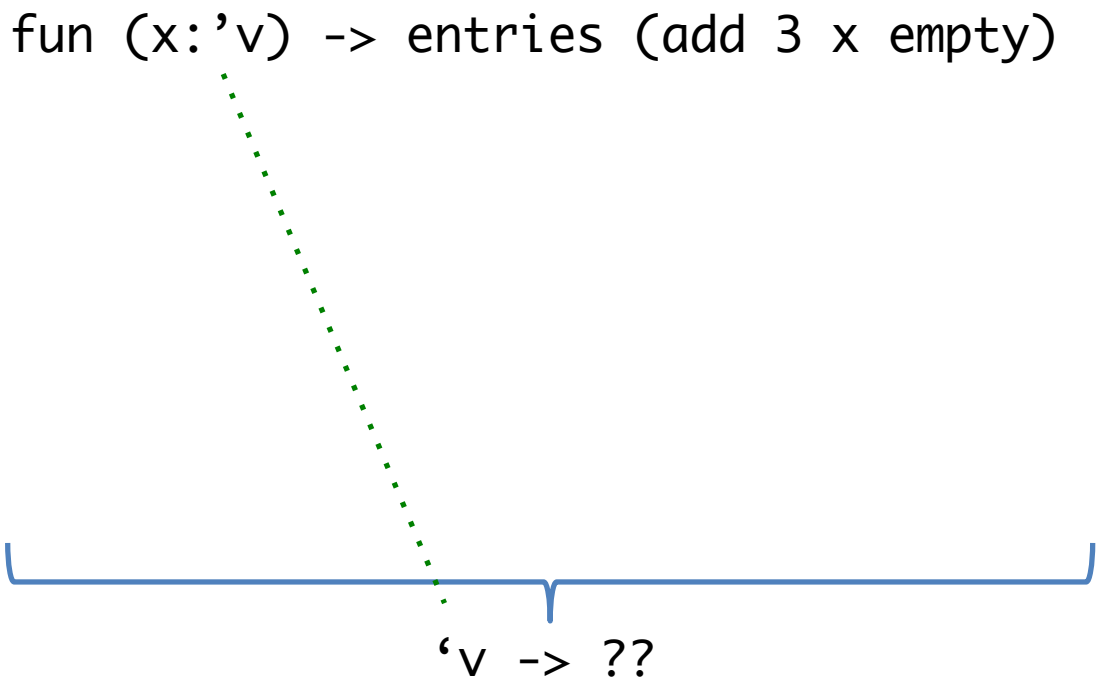
```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```

```
fun (x:’v) -> entries (add 3 x empty)
```



Example Typechecking Problem

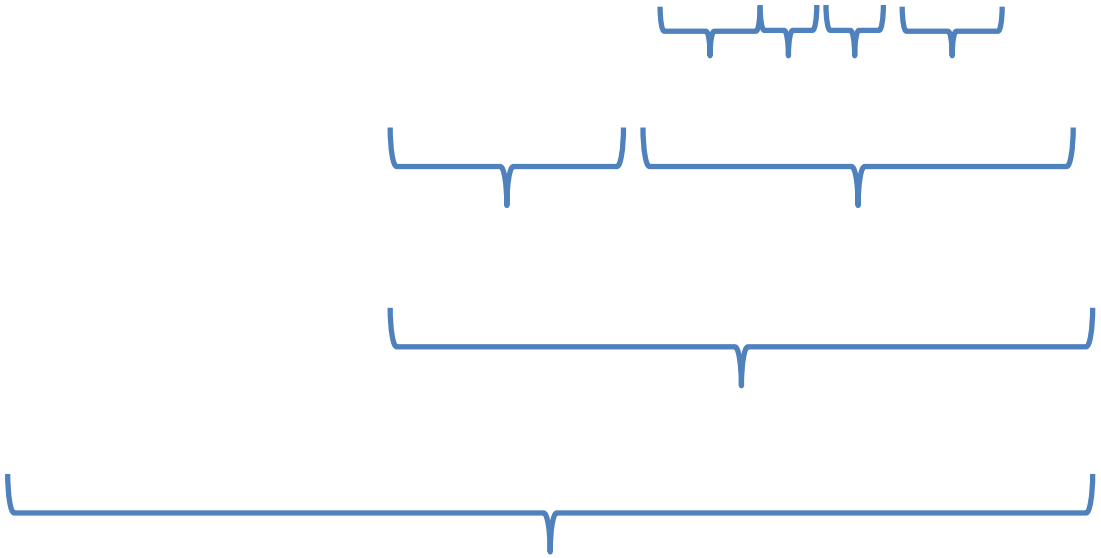
```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```



Example Typechecking Problem

```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```

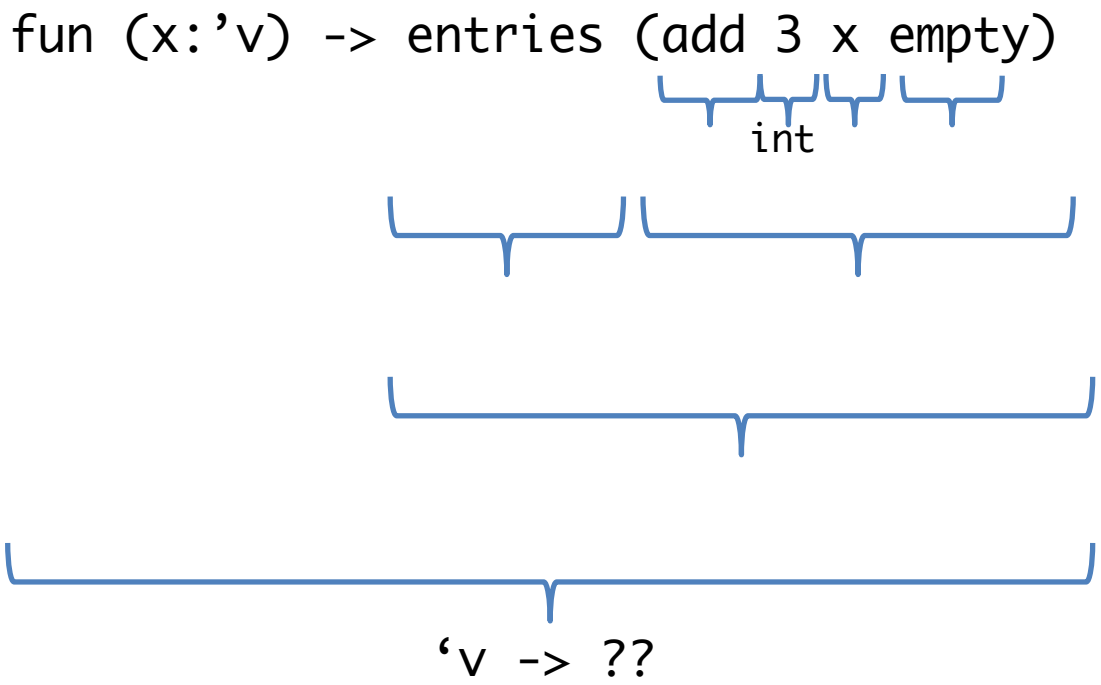
```
fun (x:’v) -> entries (add 3 x empty)
```



‘v -> ??

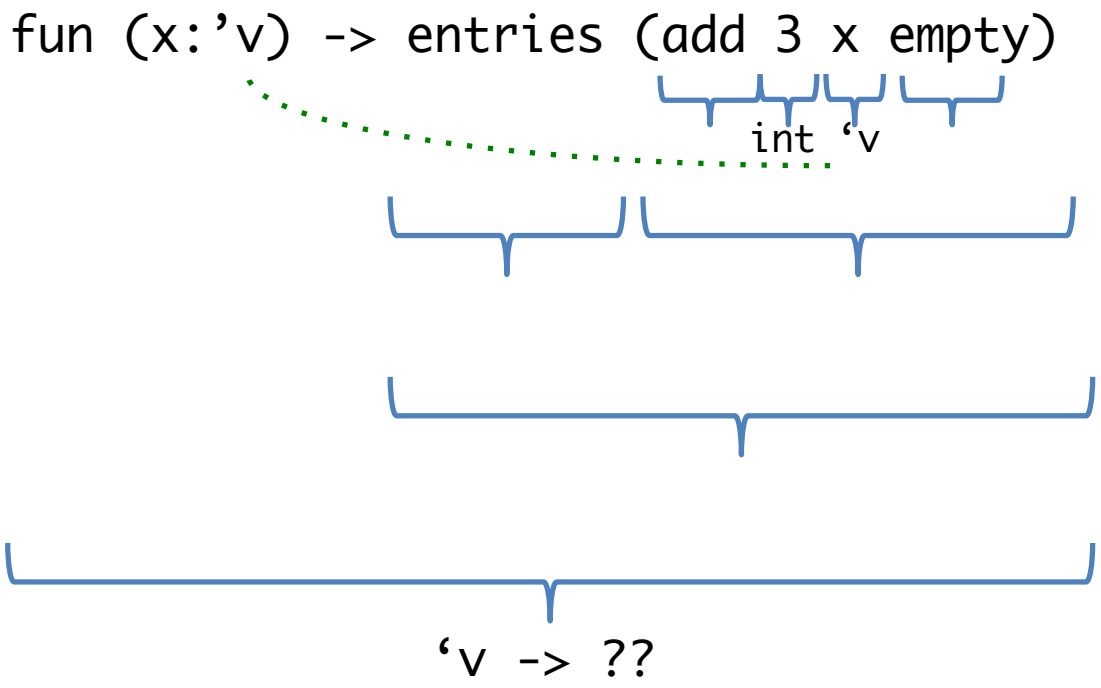
Example Typechecking Problem

```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```



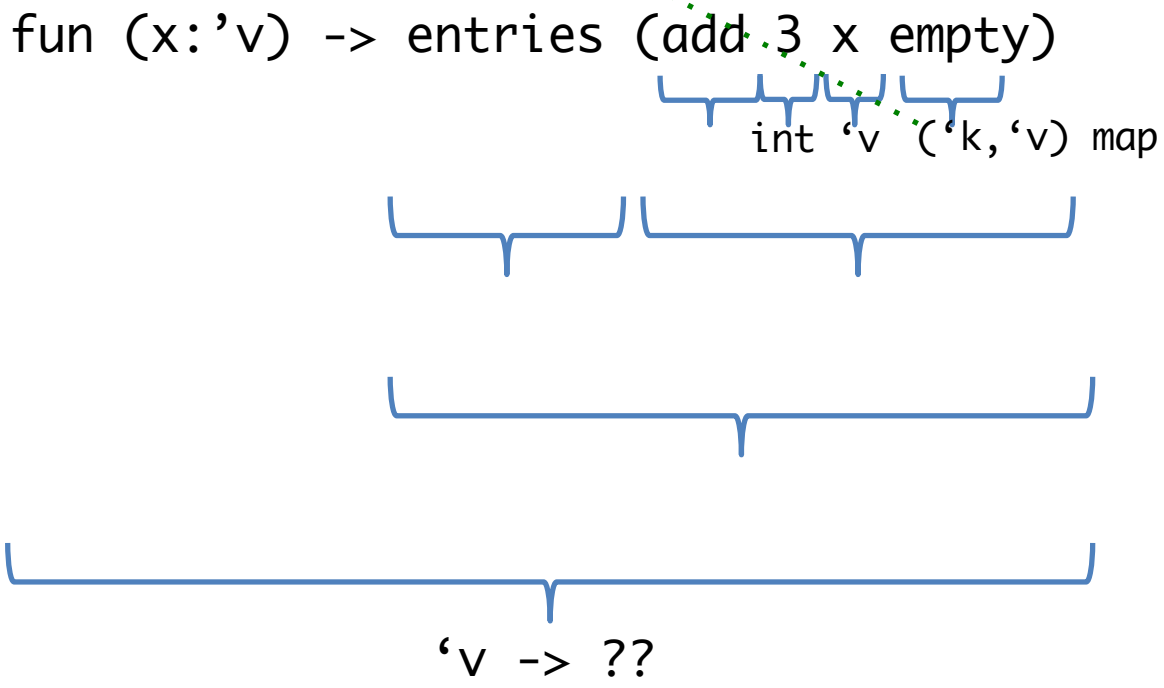
Example Typechecking Problem

```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```



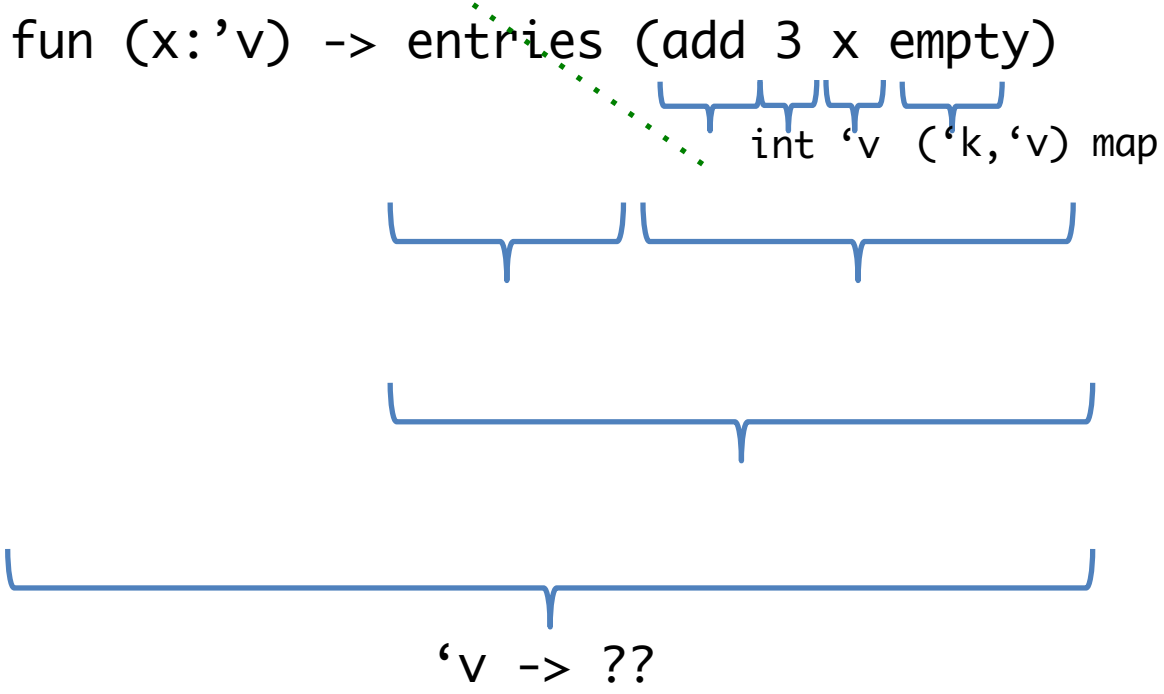
Example Typechecking Problem

```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```



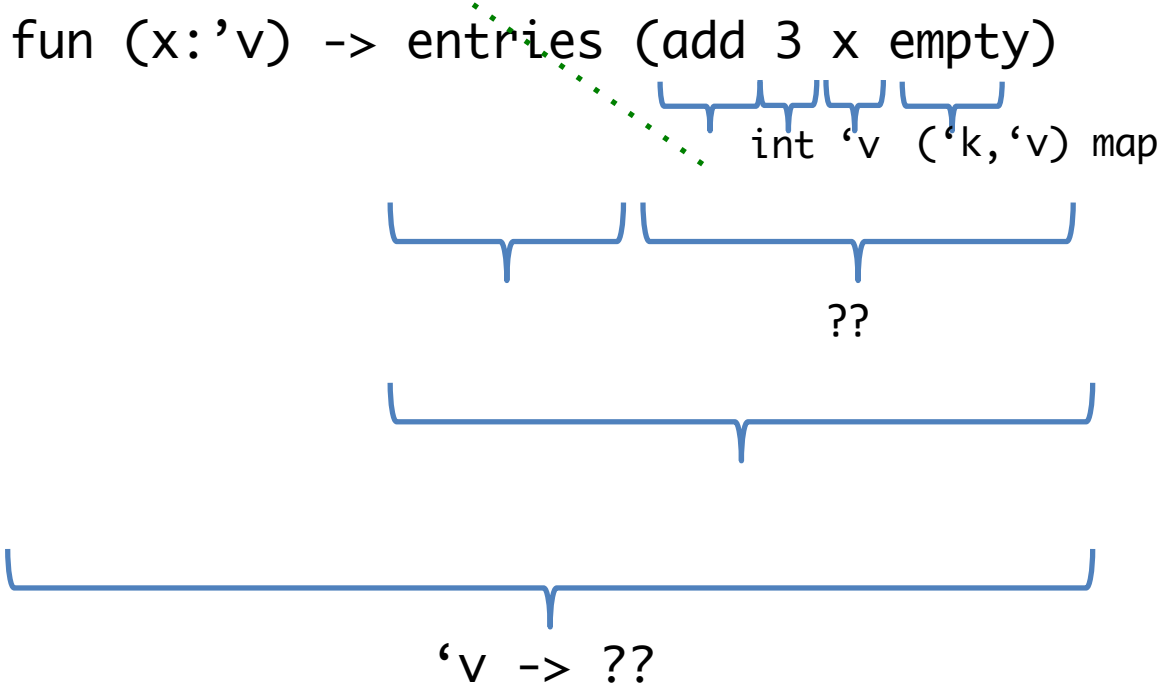
Example Typechecking Problem

```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```



Example Typechecking Problem

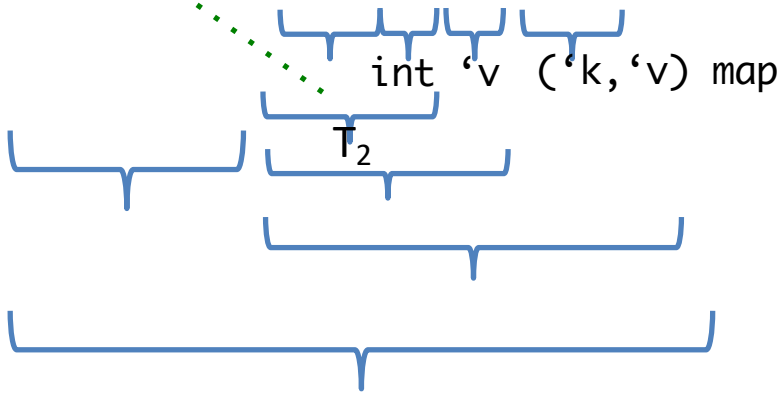
```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```



Example Typechecking Problem

```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```

```
fun (x:’v) -> entries (add 3 x empty)
```



Application:
T_1 = ‘k
T_2 = ‘v -> (‘k,’v) map -> (‘k,’v) map

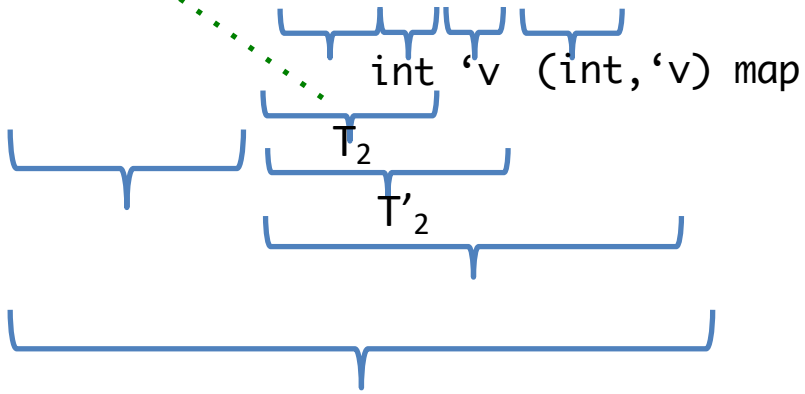
Instantiate: ‘k = int

‘v -> ??

Example Typechecking Problem

```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```

```
fun (x:’v) -> entries (add 3 x empty)
```



Another Application:

T'_1 = ‘v

T'_2 = (int,’v) map -> (int,’v) map

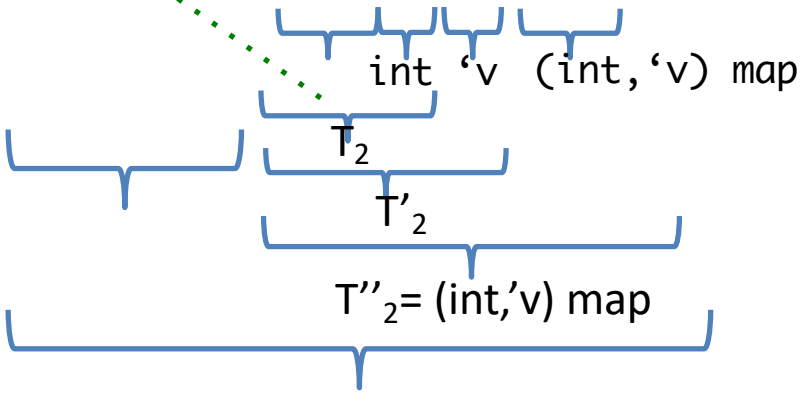
Instantiate: ‘v = ‘v

‘v -> ??

Example Typechecking Problem

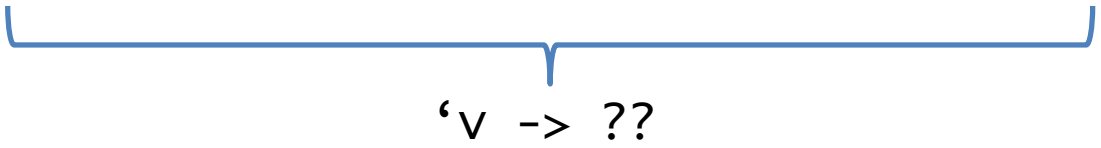
```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```

```
fun (x:’v) -> entries (add 3 x empty)
```



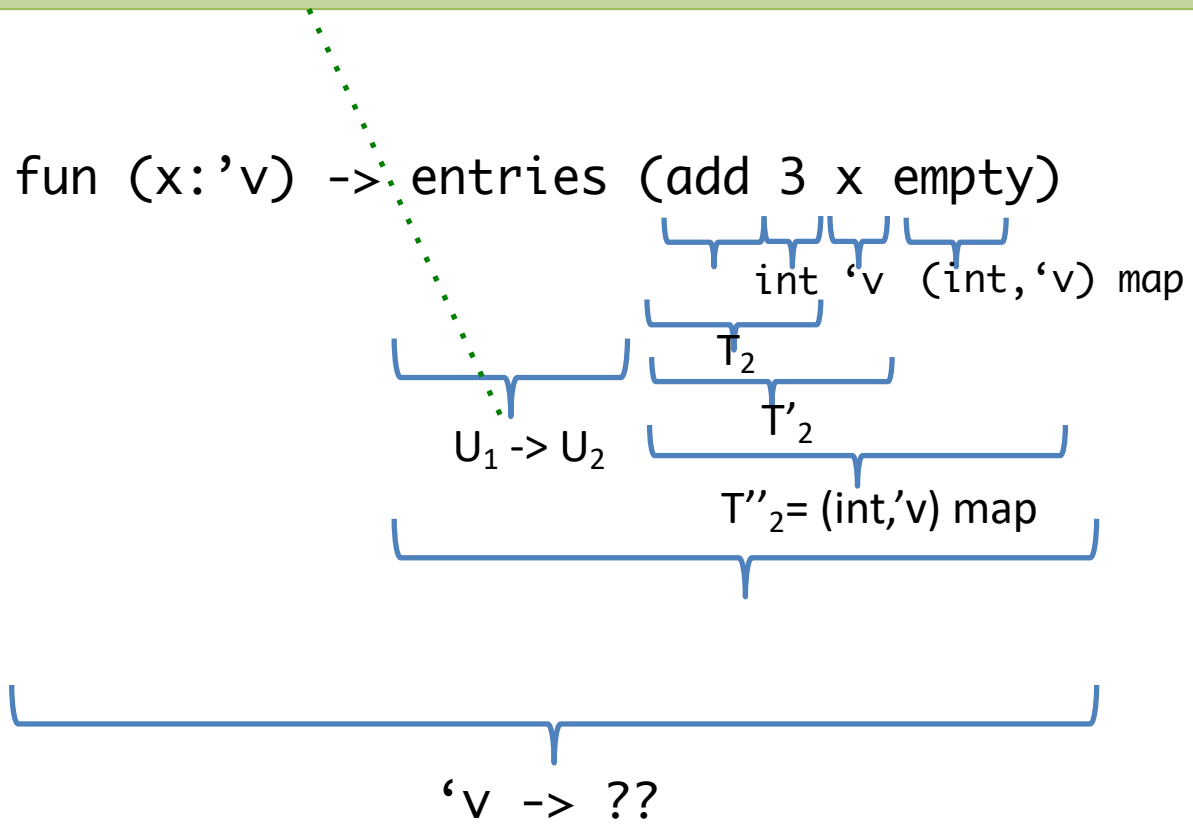
A third Application:
T''_1 = (int,’v) map
T''_2 = (int,’v) map

Argument and argument type already agree



Example Typechecking Problem

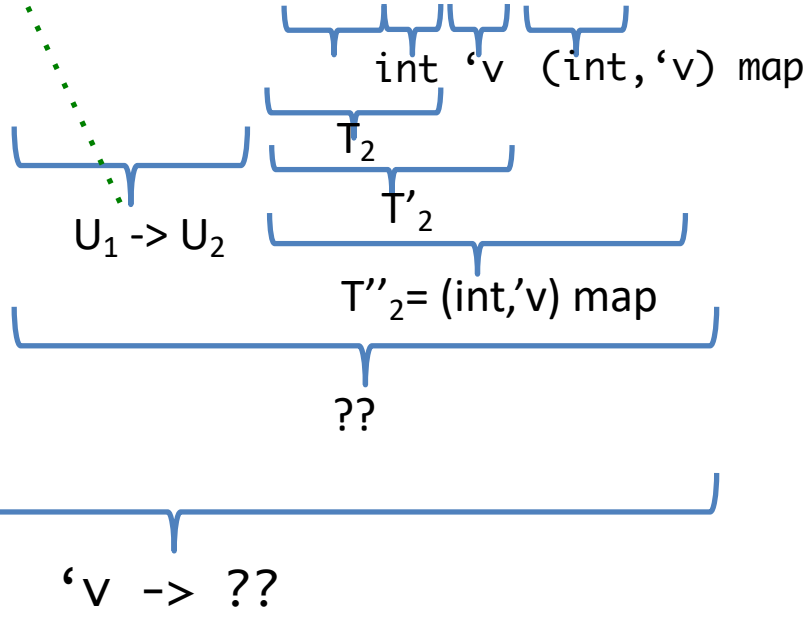
```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```



Example Typechecking Problem

```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```

```
fun (x:’v) -> entries (add 3 x empty)
```



Another Application:
U_1 = (‘k,’v) map
U_2 = (‘k * ‘v) list

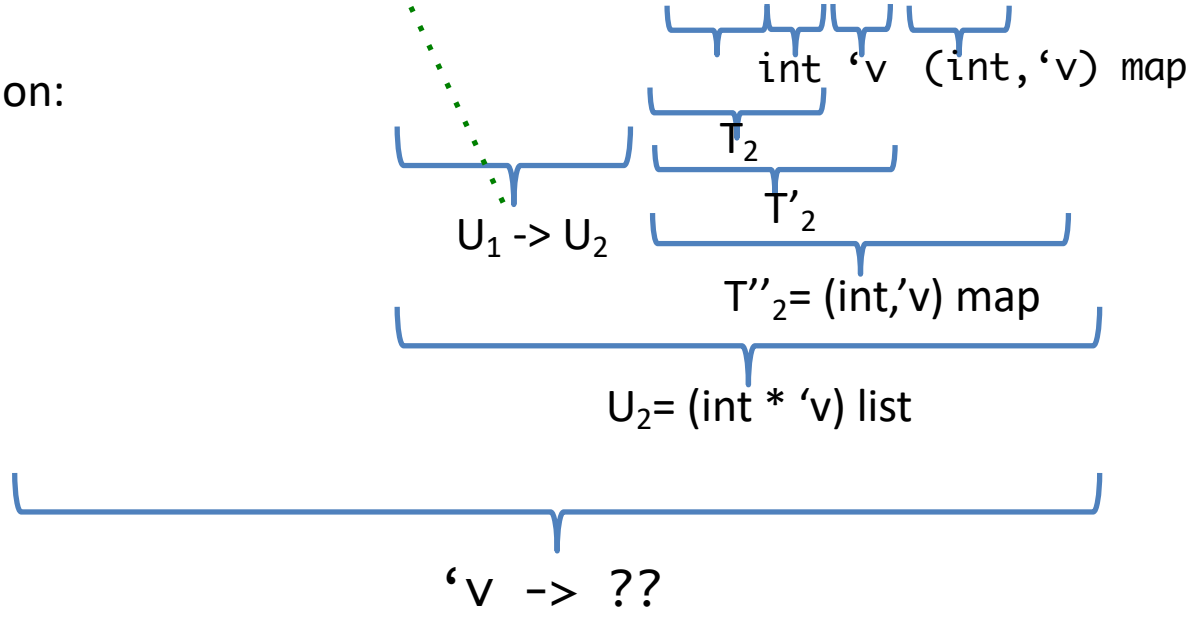
Unify U_1 with T''_2
(‘k,’v) map ~~ (int, ‘v) map

Instantiate ‘k = int

Example Typechecking Problem

```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```

```
fun (x:’v) -> entries (add 3 x empty)
```

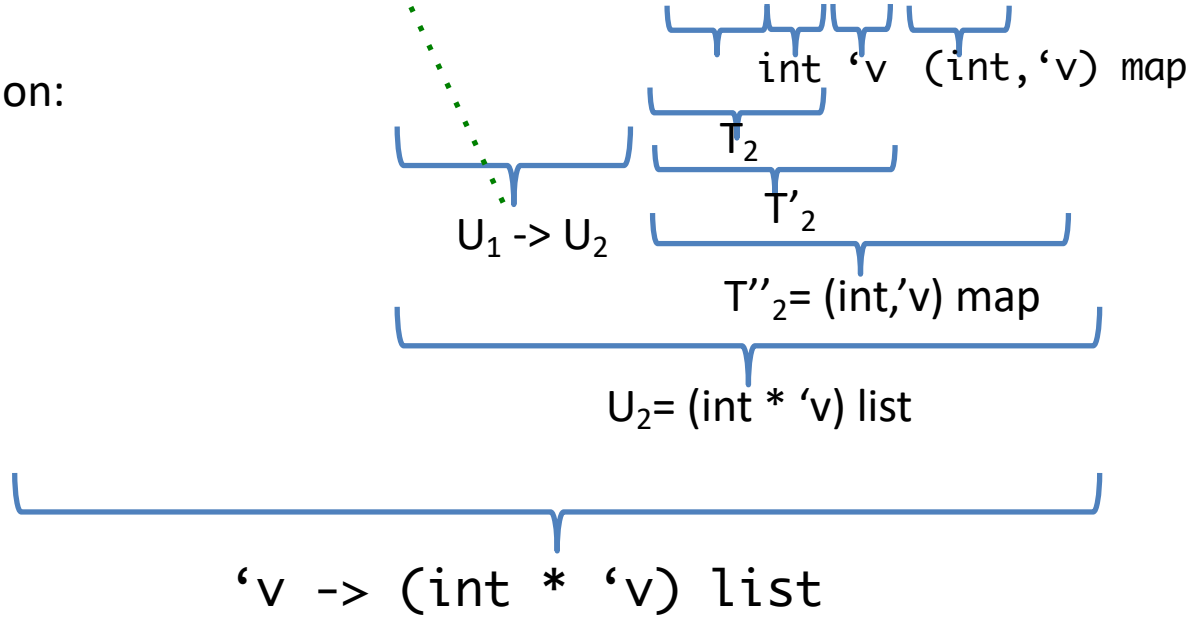


Another Application:
U_1 = (int,’v) map
U_2 = (int * ‘v) list

Example Typechecking Problem

```
empty   : (‘k,’v) map
add     : ‘k -> ‘v -> (‘k,’v) map -> (‘k,’v) map
entries : (‘k, ‘v) map -> (‘k * ‘v) list
```

```
fun (x:’v) -> entries (add 3 x empty)
```



Another Application:
U_1 = (int,’v) map
U_2 = (int * ‘v) list

Ill-typed Expressions?

- An expression is ill-typed if, during this type checking process, inconsistent constraints are encountered:

```
empty   : ('k,'v) map
add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

```
add 3 true (add "foo" false empty)
```

Error: found `int` but expected `string`