Programming Languages and Techniques (CIS120)

Lecture 17

October 12th , 2015

More Queue Manipulation
"Objects"

Announcements

- Homework 4: Queues
  - Due: Tomorrow, October 13th at 11:59 pm

- Homework 5: GUI Library & Paint
  - Available Soon
  - Due: Thursday, October 22nd

What happens when you run this function on a (valid) queue containing 2 elements?

```
let f (q:'a queue) : int =
    let rec loop (qn:'a qnode option) : int =
      begin match qn with
        | None -> 0
        | Some n -> 1 + loop qn
      end
    in loop q.head
```

1. The value 2 is returned
2. The value 0 is returned
3. StackOverflow
4. Your program hangs

What happens when you run this function on a (valid) queue containing 2 elements?

```
let f (q:'a queue) : int =
    let rec loop (qn:'a qnode option) (len:int) : int =
      begin match qn with
        | None -> len
        | Some n -> loop qn (len + 1)
      end
    in loop q.head 0
```

1. The value 2 is returned
2. The value 0 is returned
3. StackOverflow
4. Your program hangs

More queue iteration examples

to_list

print

get_tail

to_list (using iteration)

```
(* Retrieve the list of values stored in the queue,
   ordered from head to tail. *)
let to_list (q: 'a queue) : 'a list =
  let rec loop (no: 'a qnode option) (l:'a list) : 'a list =
      begin match no with
      | None -> List.rev l
      | Some n -> loop n.next (n.v::l)
      end
  in loop q.head []
```

- Here, the state maintained across each iteration of the loop is the queue "index pointer" no and the (reversed) list of elements traversed.
- The "exit case" post processes the list by reversing it.

print (using iteration)

```
let print (q:'a queue) (string_of_element:'a -> string) : unit =
  let rec loop (no: 'a qnode option) : unit =
    begin match no with
      | None -> ()
      | Some n -> print_endline (string_of_element n.v);
                  loop n.next
    end
  in
    print_endline "--- queue contents ---";
    loop q.head;
    print_endline "--- end of queue -----"
```

- Here, the only state needed is the queue “index pointer”.

Singly-linked Queue Processing

- General structure (schematically) :

```
(* Process a singly-linked queue. *)
let queue_operation (q: 'a queue) : 'b =
  let rec loop (current: 'a qnode option) (s:'a state) : 'b =
      begin match no with
      | None -> …  (* iteration complete, produce result *)

      | Some n ->  … (* do something with n,
                        create new loop state *)
                   loop current.next new_s

      end
  in loop q.head init
```

- What is useful to put in the state?
  - Accumulated information about the queue (e.g. length so far)
  - Link to previous node (so that it could be updated, for example)

valid

Either:
(1) head and tail are both None (i.e. the queue is empty)
or
(2) head is Some n1, tail is Some n2 and
- n2 is reachable from n1 by following 'next' pointers
- n2.next is None

```
(* Determine whether the q satisfies the q invariant *)
let valid (q: 'a queue) : bool =
      begin match (q.head,q.tail) with
      | (None,None) ->  true
      | (Some n1,Some n2) ->
          begin match get_tail n1 with
            | Some n -> n2 == n   (* tail is the last node *)
            | None -> false
          end
      | (_,_) ->  false
      end
```

get_tail (using iteration)

```
(* get the tail (if any) from a queue *)
let rec get_tail (hd: 'a qnode) : 'a qnode option =
    let rec loop (qn: 'a qnode) (seen: 'a qnode list)
        : 'a qnode option =
      begin match qn.next with
        | None -> Some qn
        | Some n ->
            if contains_alias n seen then None
            else loop n (qn::seen)
      end
    in loop hd []
```

- This function works even if the queue has cycles.
  - It returns Some qn if qn is a "tail" reachable from hd (qn may be hd)
  - It returns None if there is a cycle
- The state is an index pointer and a list of all the nodes seen.
  - contains_alias is a helper function that checks to see whether n has an alias in the list

General Guidelines

- Processing *must* maintain the queue invariants
- Update the head and tail references (if necessary)
- If changing the link structure:
  - Sometimes useful to keep reference to the previous node (allows removal of the current node)
- Drawing pictures of the queue heap structure is helpful

- If iterating over the whole queue (e.g. to find an element)
  - It is usually *not useful* to use helpers like “is_empty” or “contains” because you will have to account for those cases during the traversal anyway!

“Objects” and Hidden State

Encapsulating State

What number is printed by this program?

```
let f =
  let x = 2 in
  fun (y : int) -> x + y
let x = 4
;; print_int (f 1)
```

1. 1
2. 2
3. 3
4. 4
5. 5
6. other

How did you answer this question?

1. Substitution model
2. Abstract Stack Machine
3. I just knew the answer
4. I didn't know, so I guessed

Answer: 3

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 abstraction:* There is only one counter in the world. If we want another, we need another counter_state value and another *incr* function.
  - *No encapsulation*: Any other code can modify count, too.

Using Hidden State

- 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 ())
```

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

Answer: 1

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

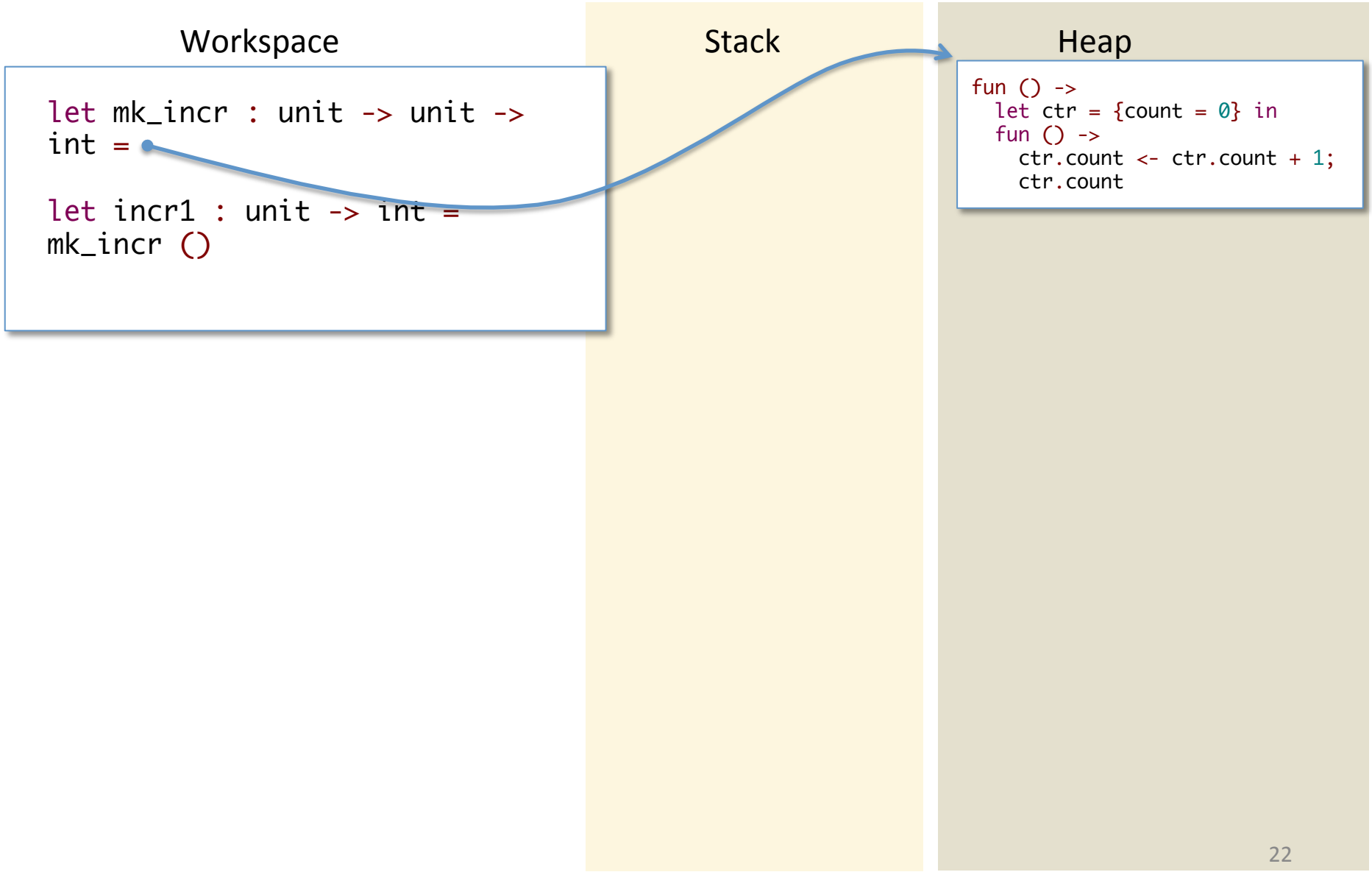
```
let mk_incr : unit -> unit ->
int = fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count


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

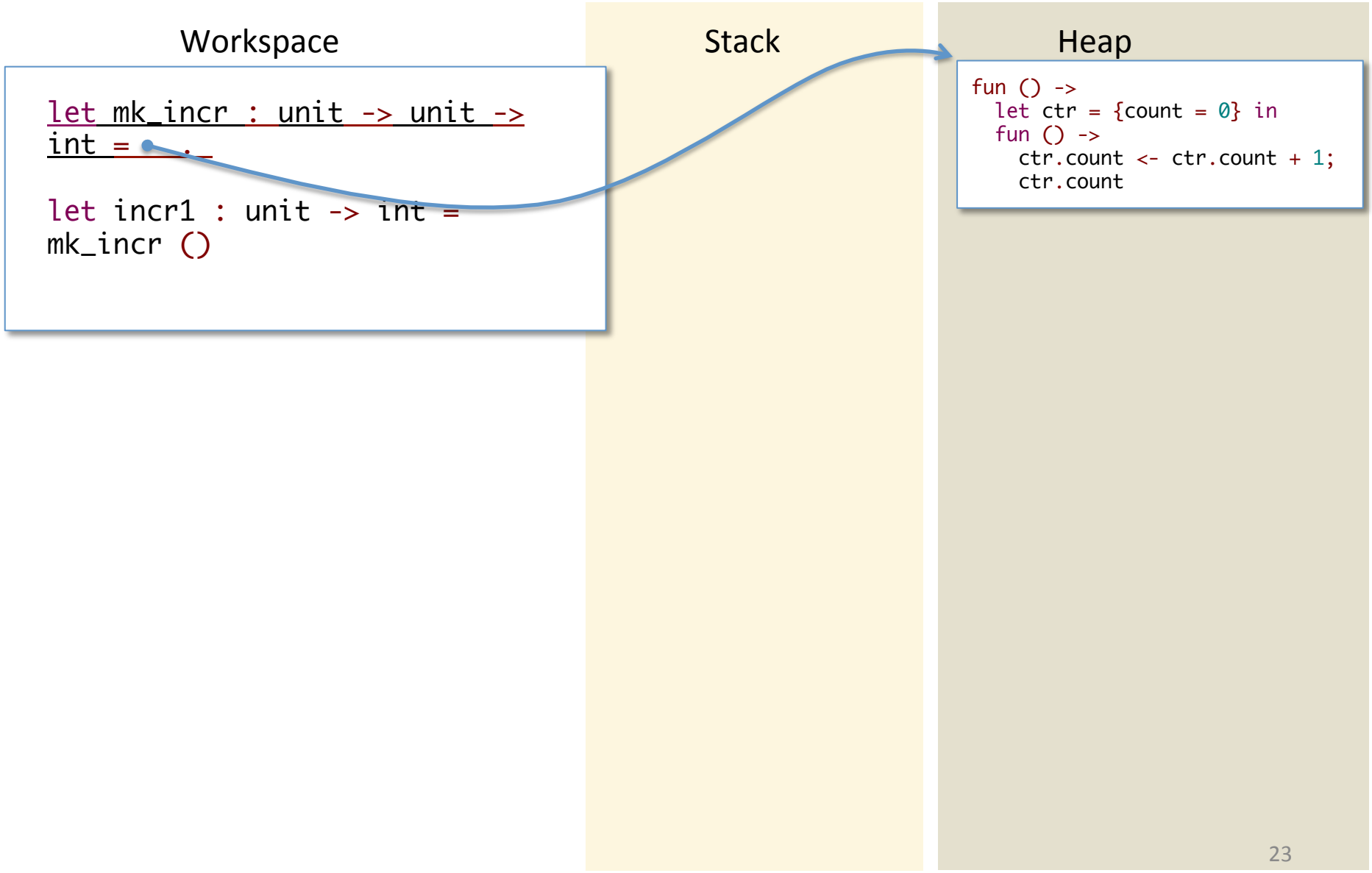
Stack

Heap

Running mk_incr



Running mk_incr

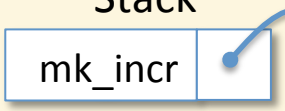


Running mk_incr

Workspace

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

Stack



Heap

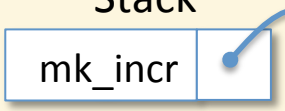
```
fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

Running mk_incr

Workspace

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

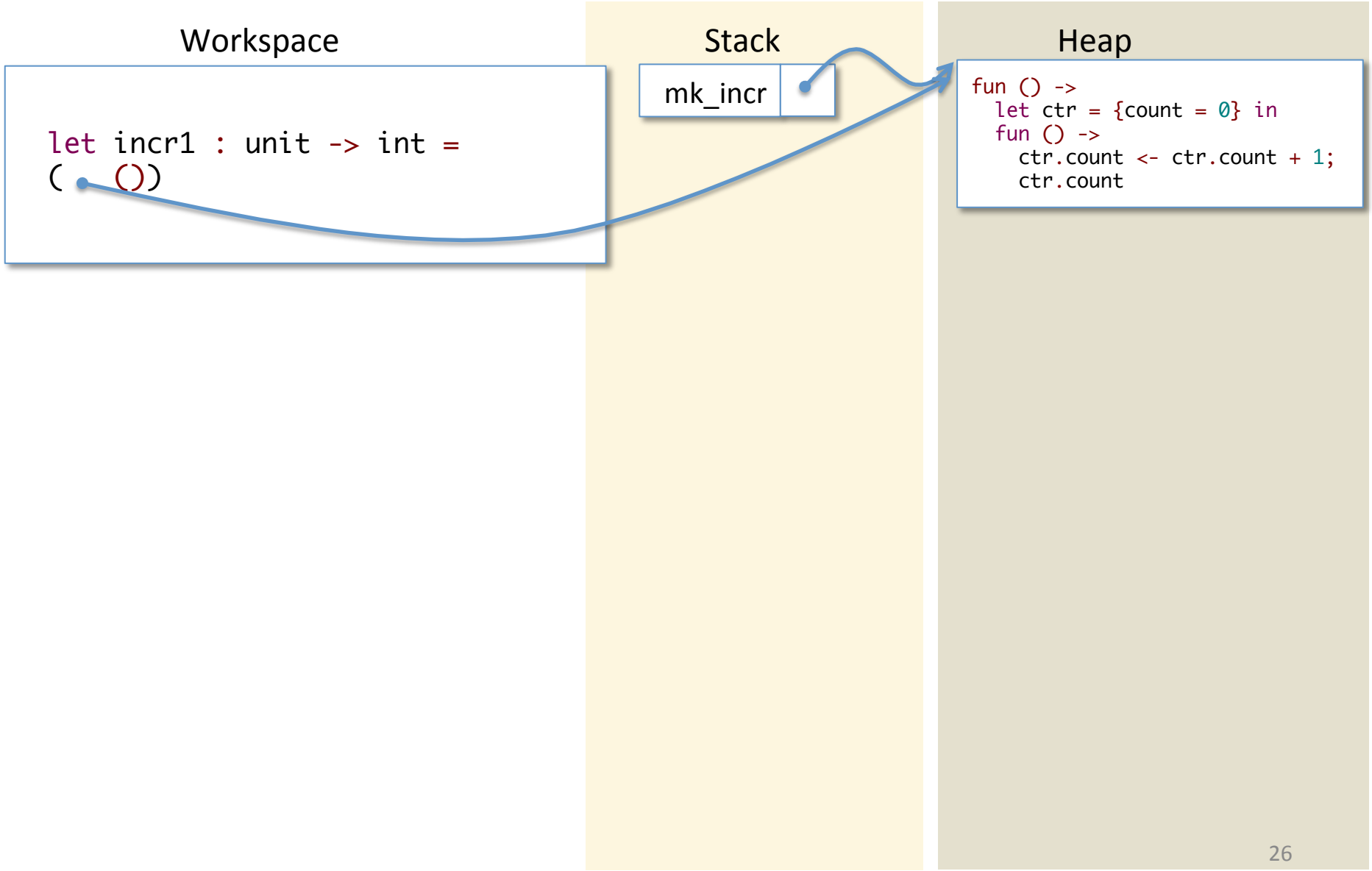
Stack



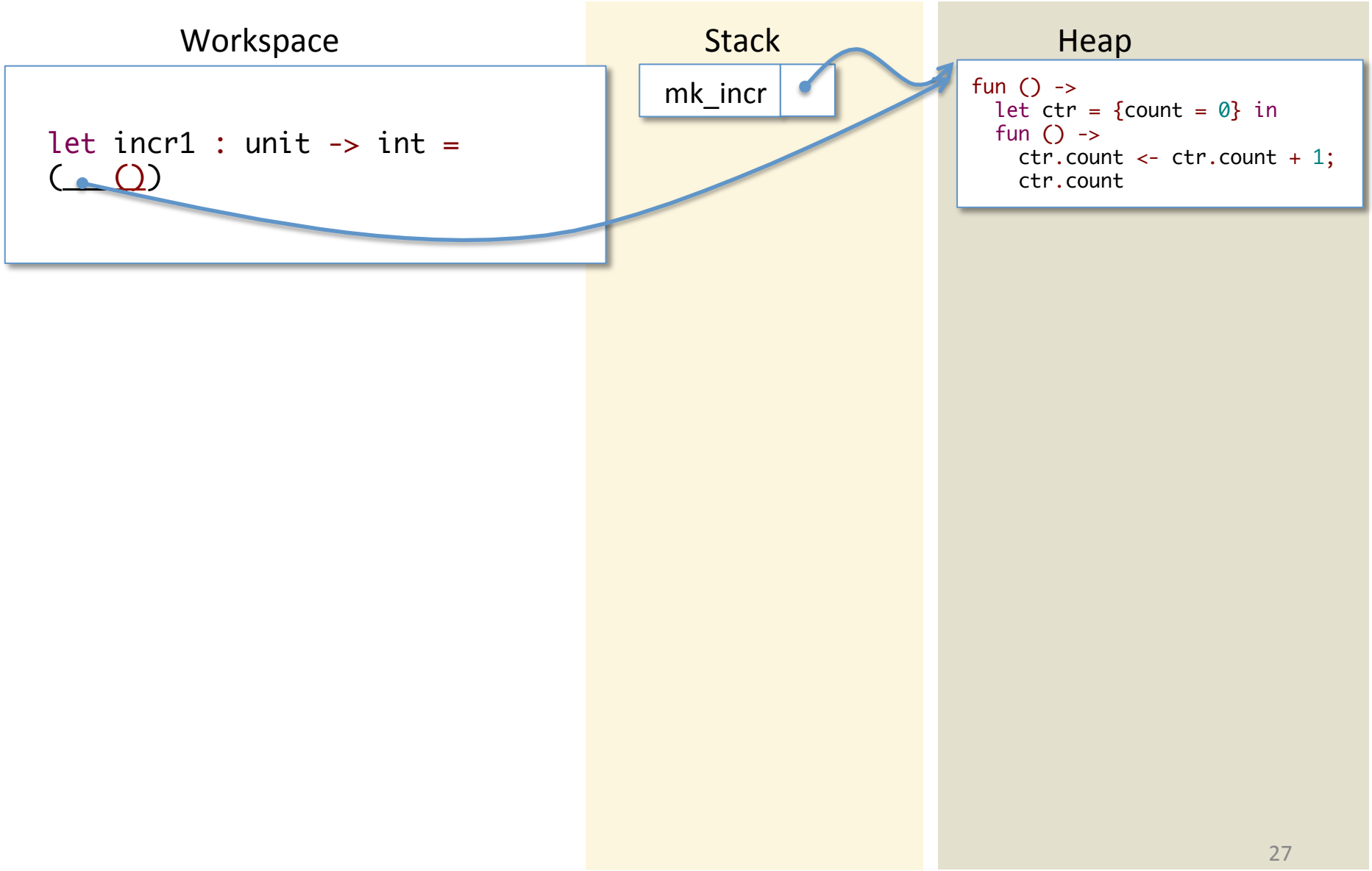
Heap

```
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

mk_incr

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

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

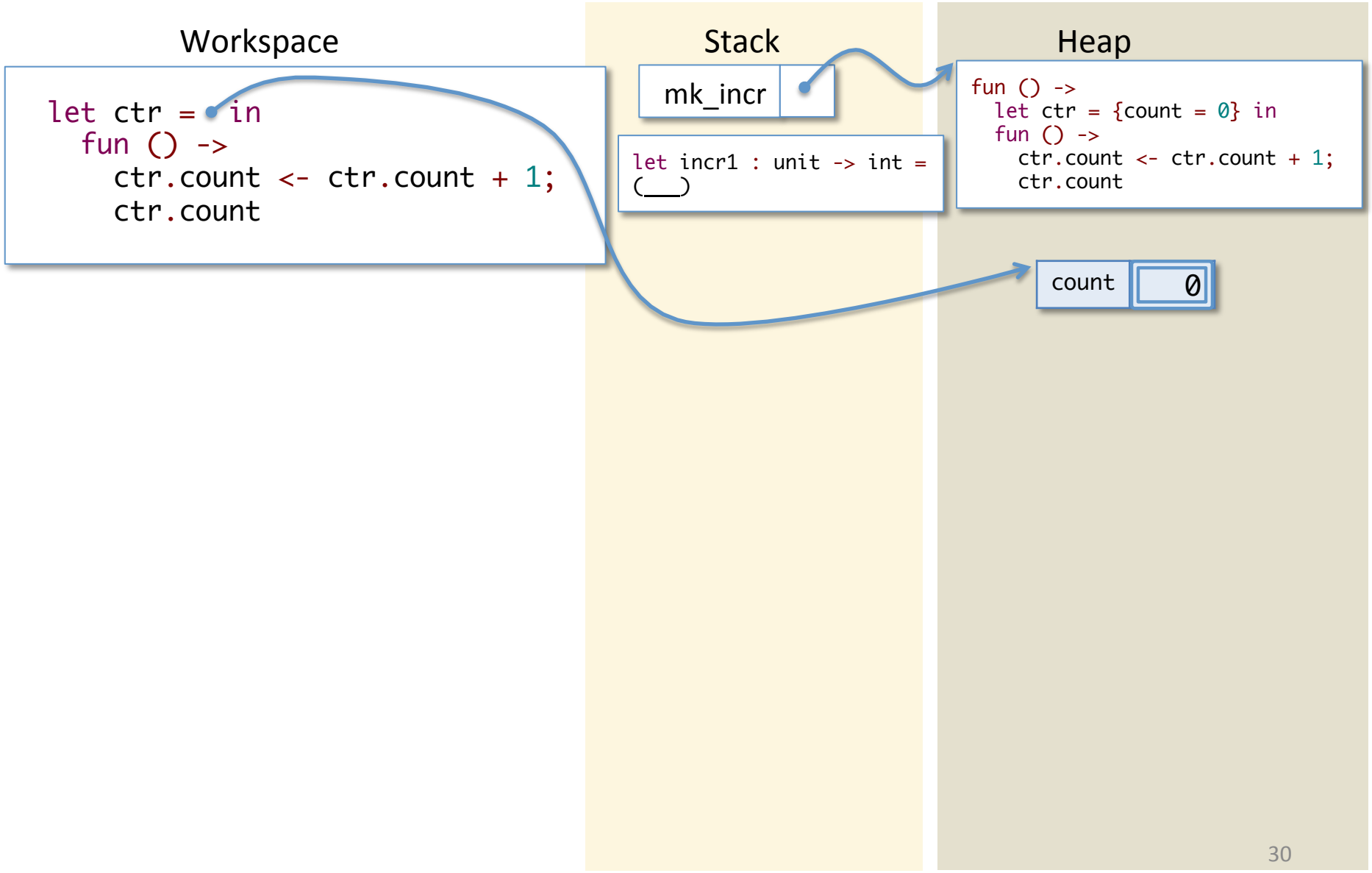
mk_incr

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

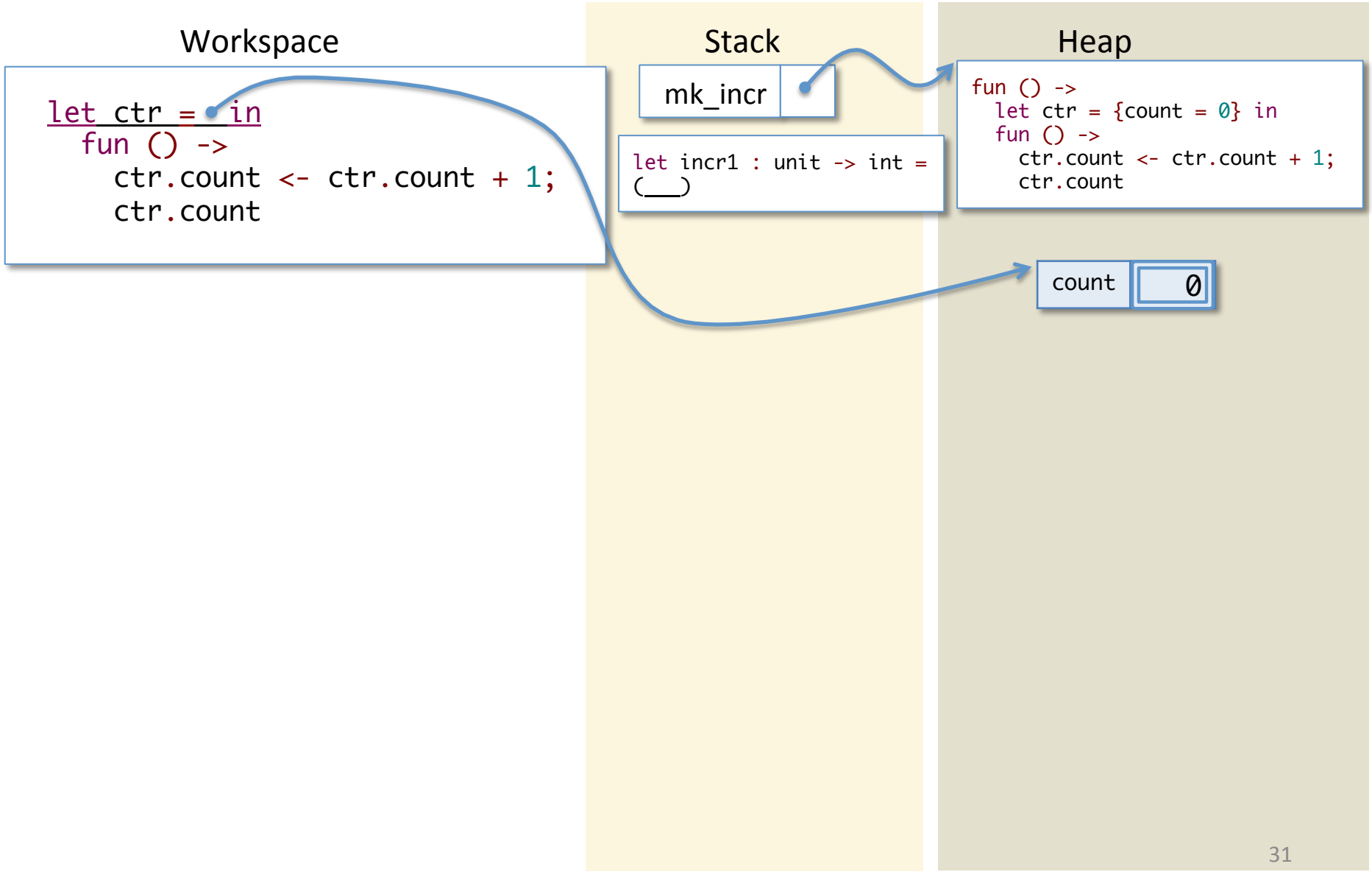
Heap

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

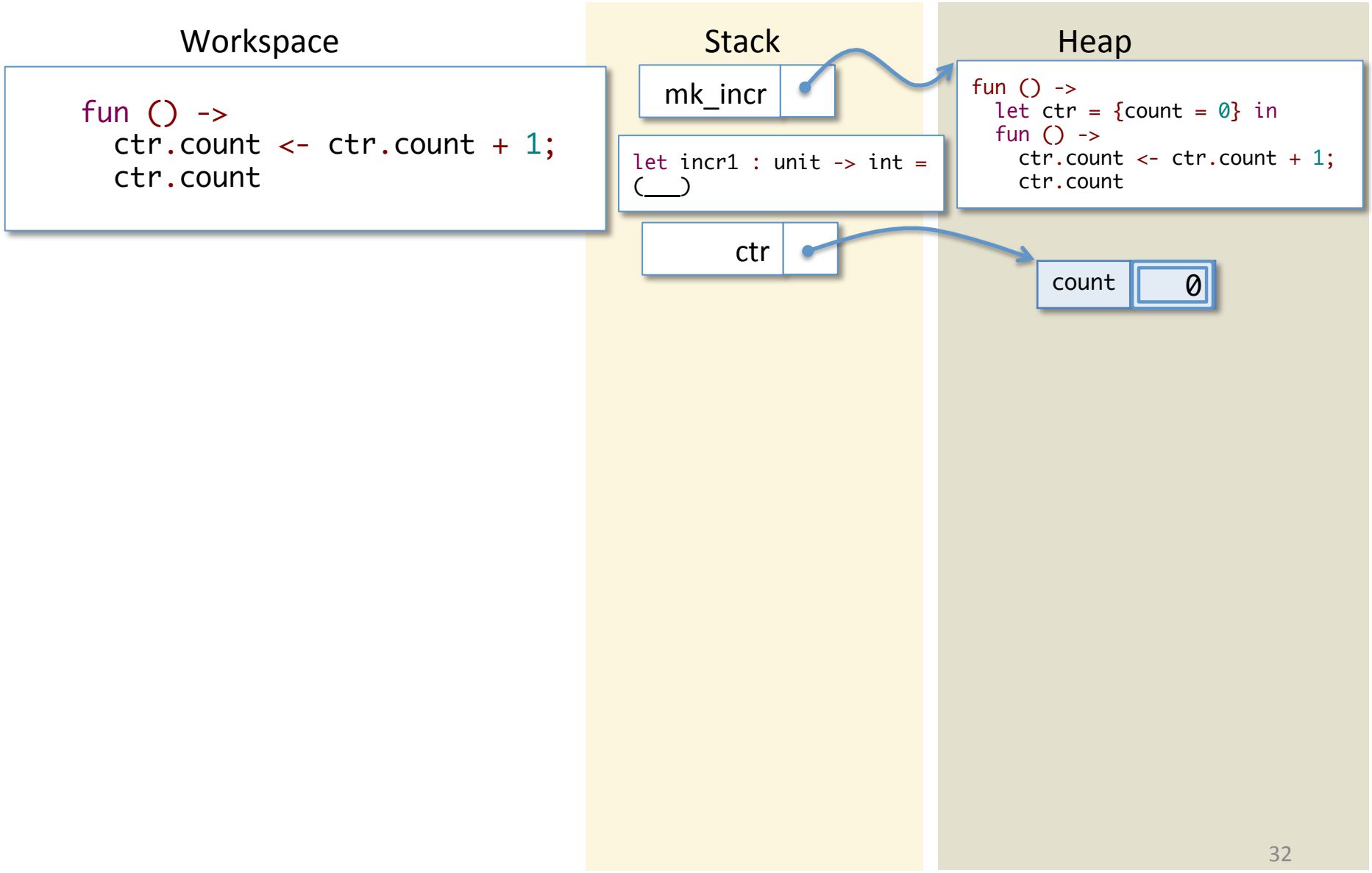
Running mk_incr



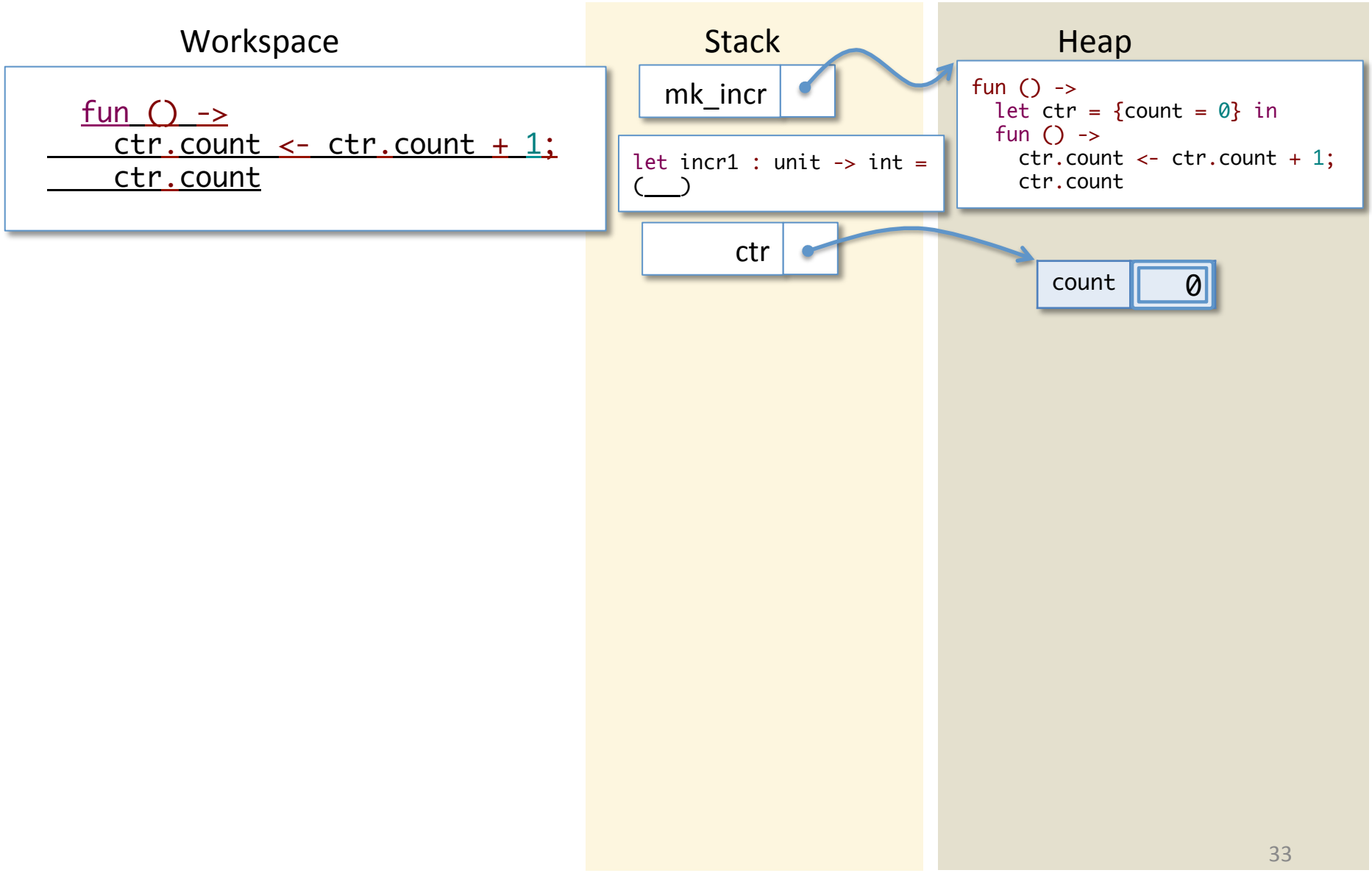
Running mk_incr



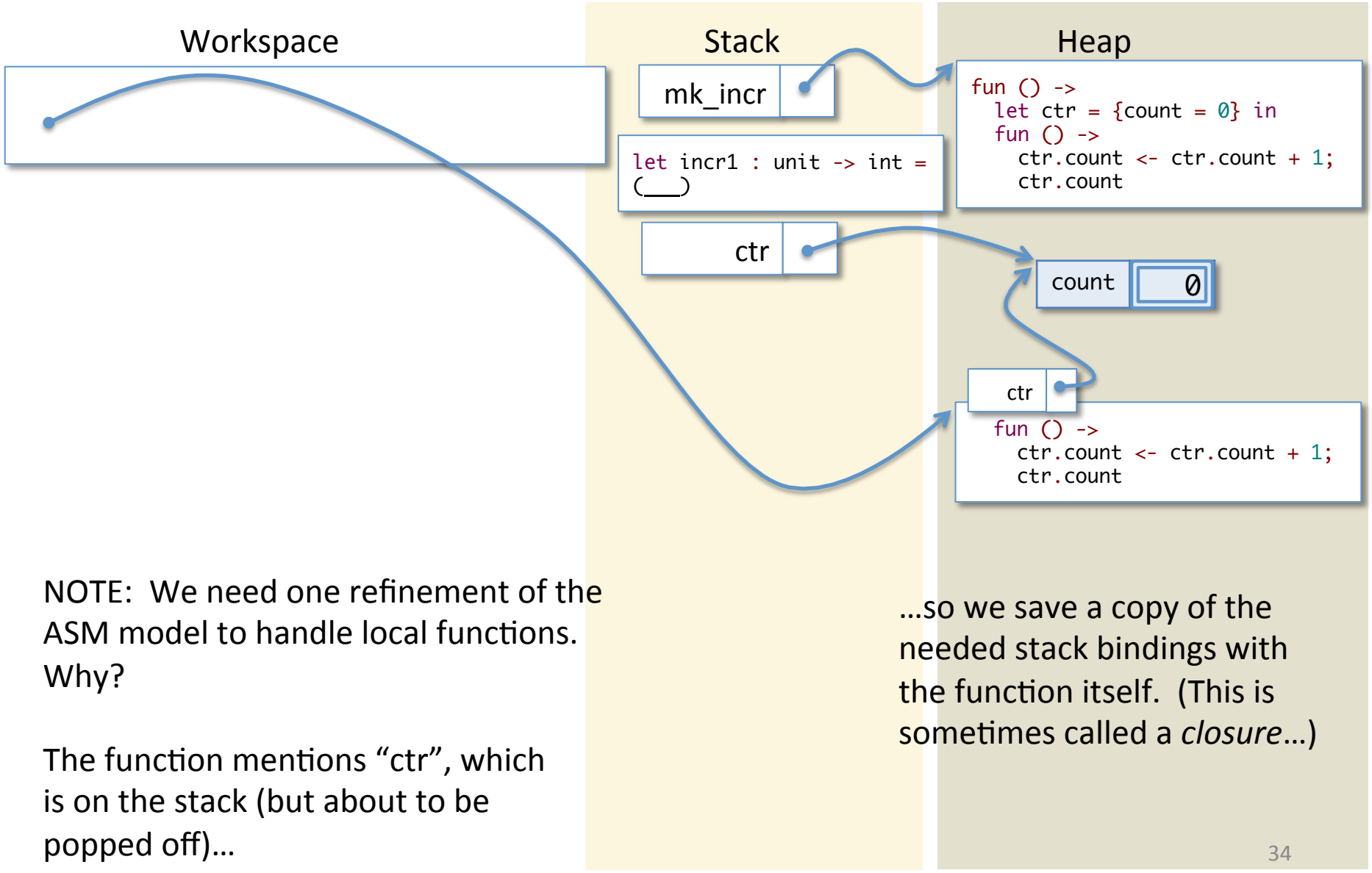
Running mk_incr



Running mk_incr



Local Functions

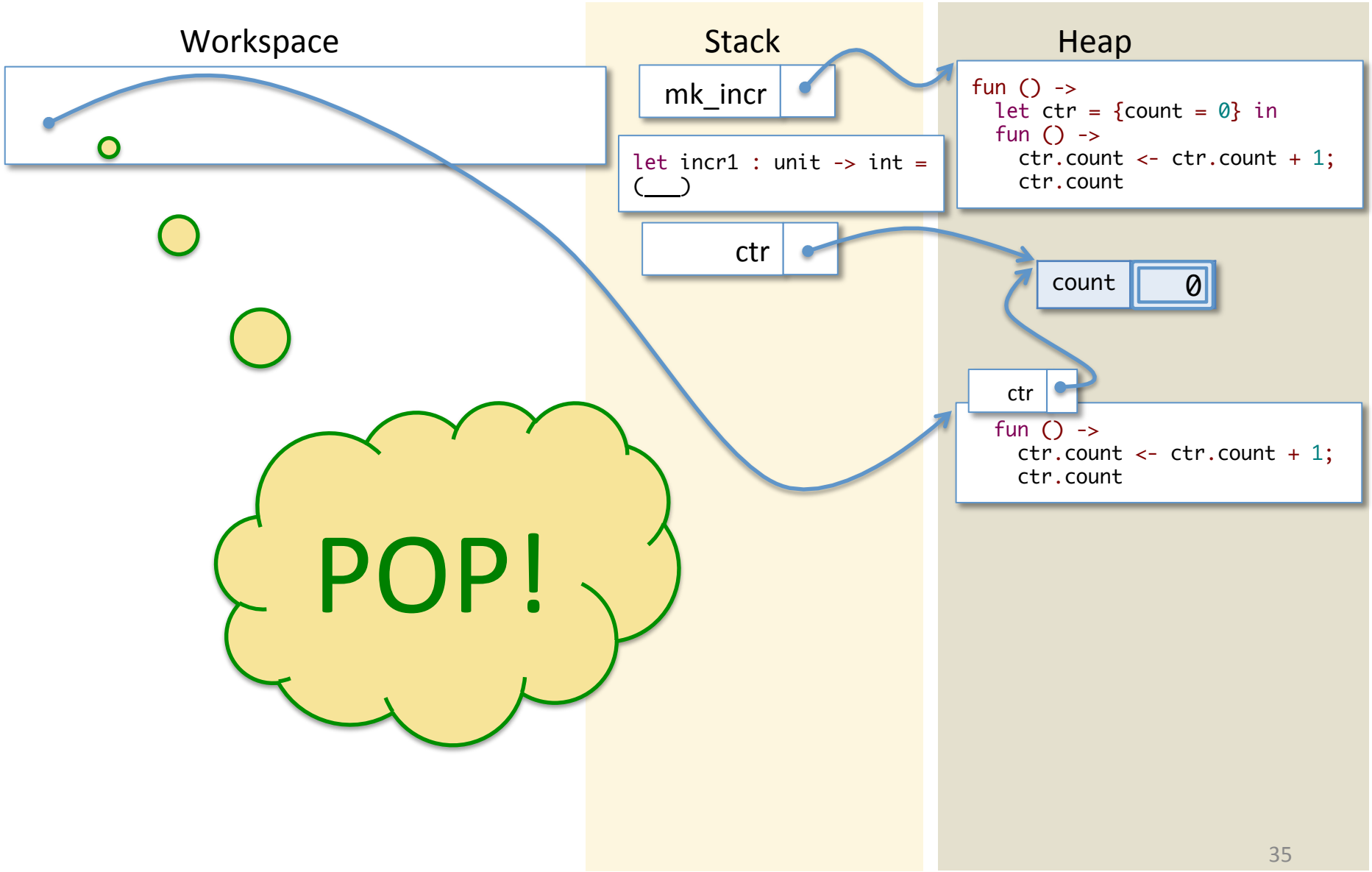


NOTE: We need one refinement of the ASM model to handle local functions. Why?

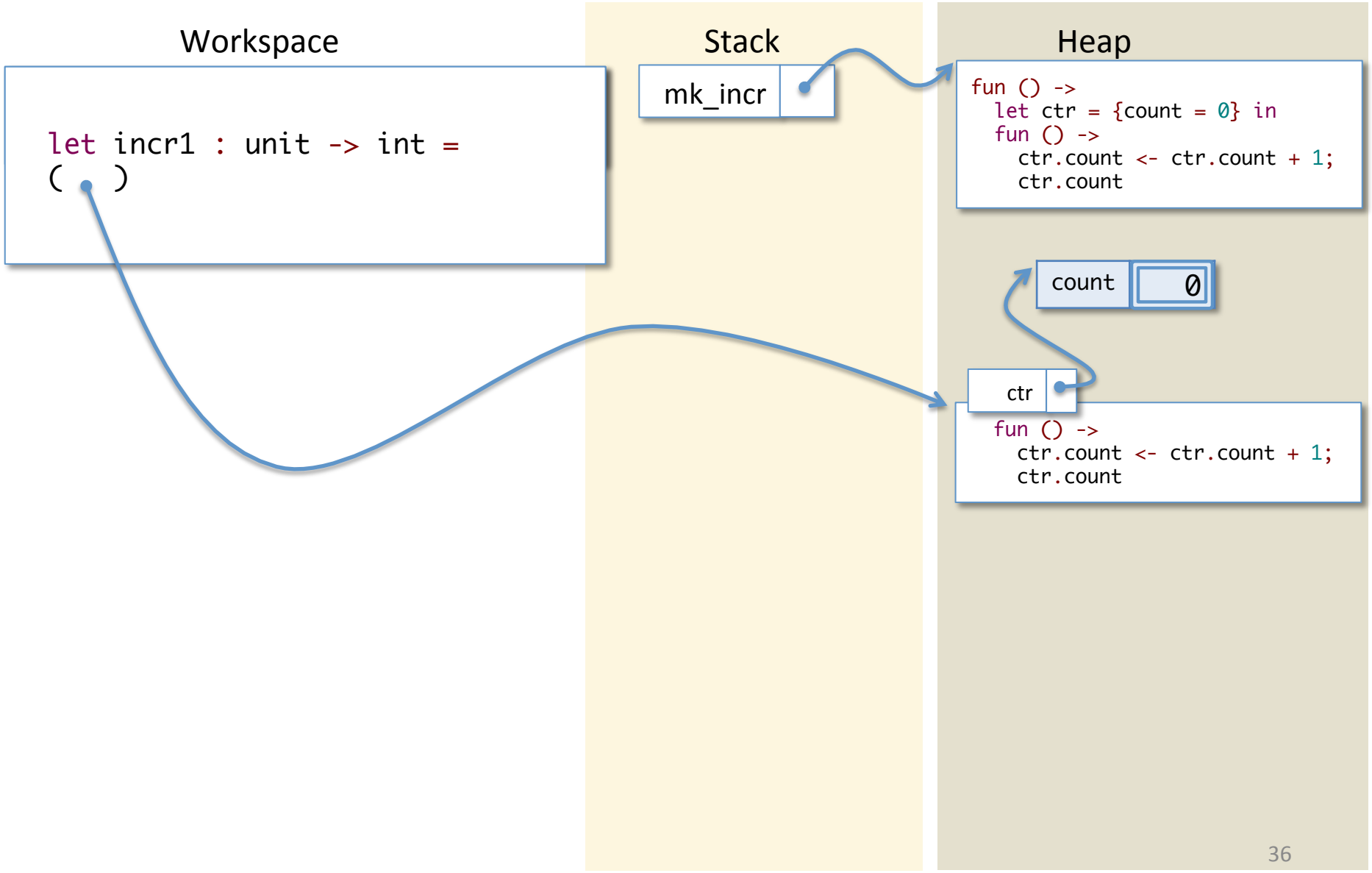
The function mentions "ctr", which is on the stack (but about to be popped off)…

…so we save a copy of the needed stack bindings with the function itself. (This is sometimes called a *closure*…)

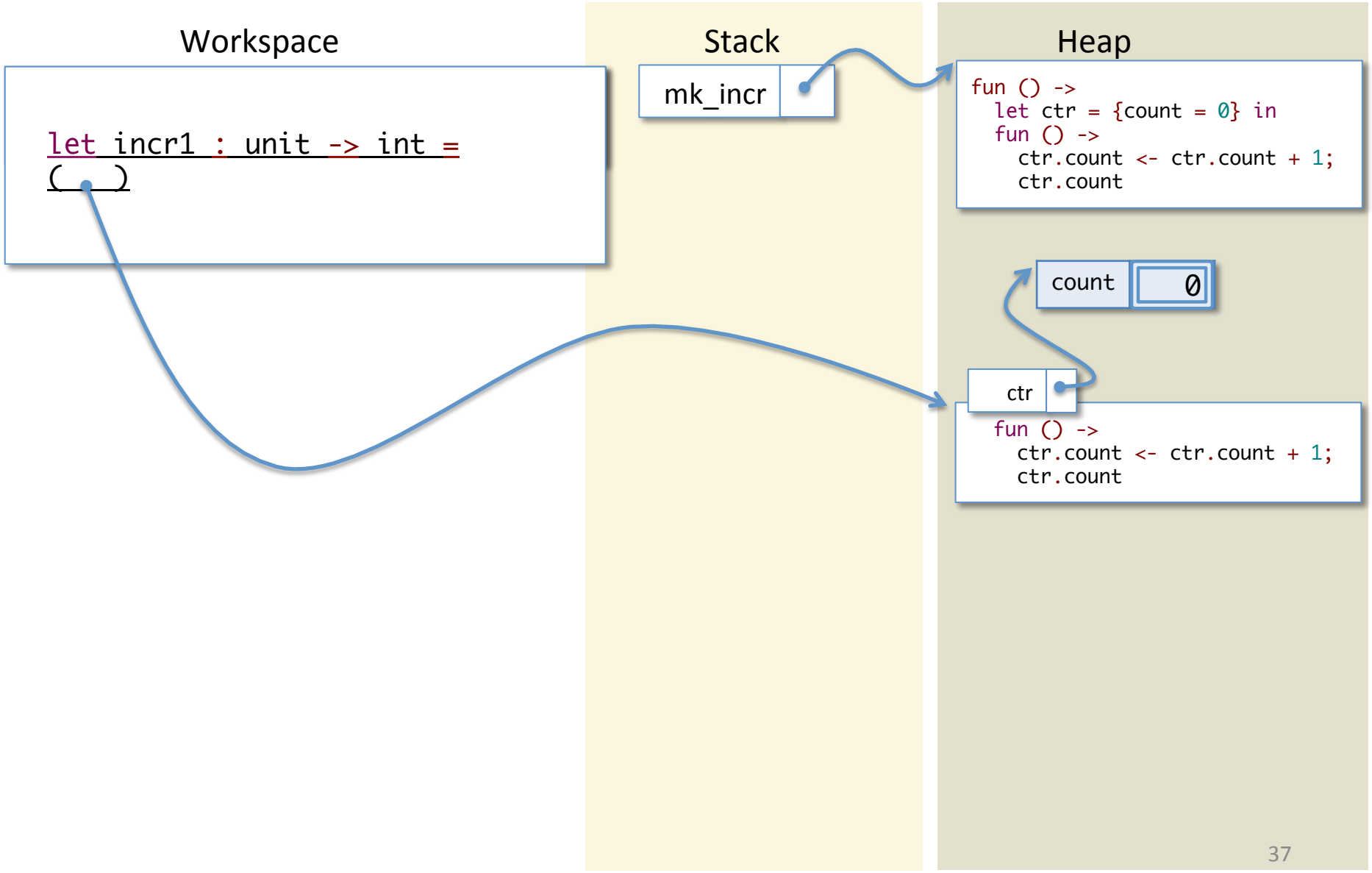
Local Functions



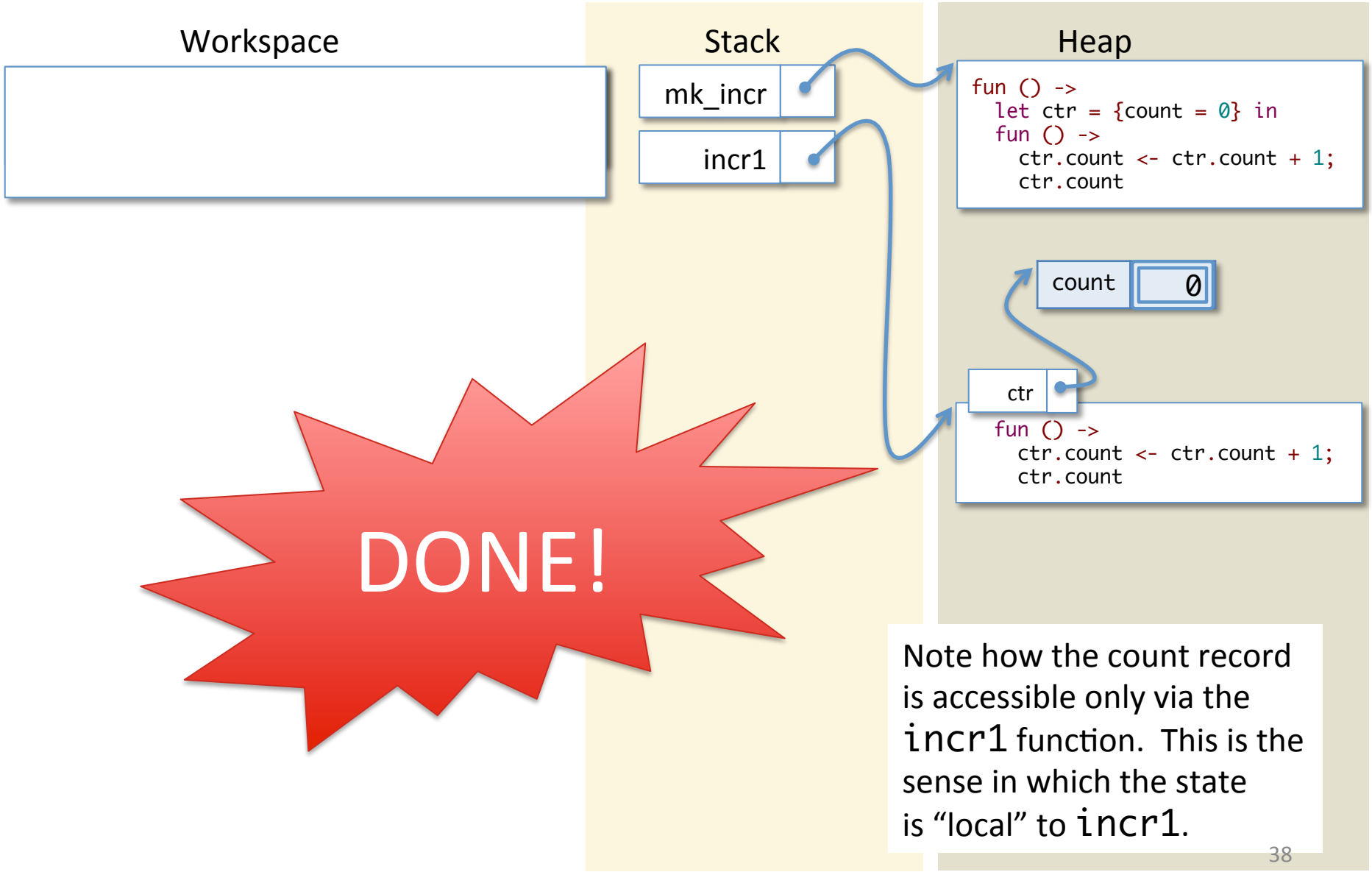
Local Functions



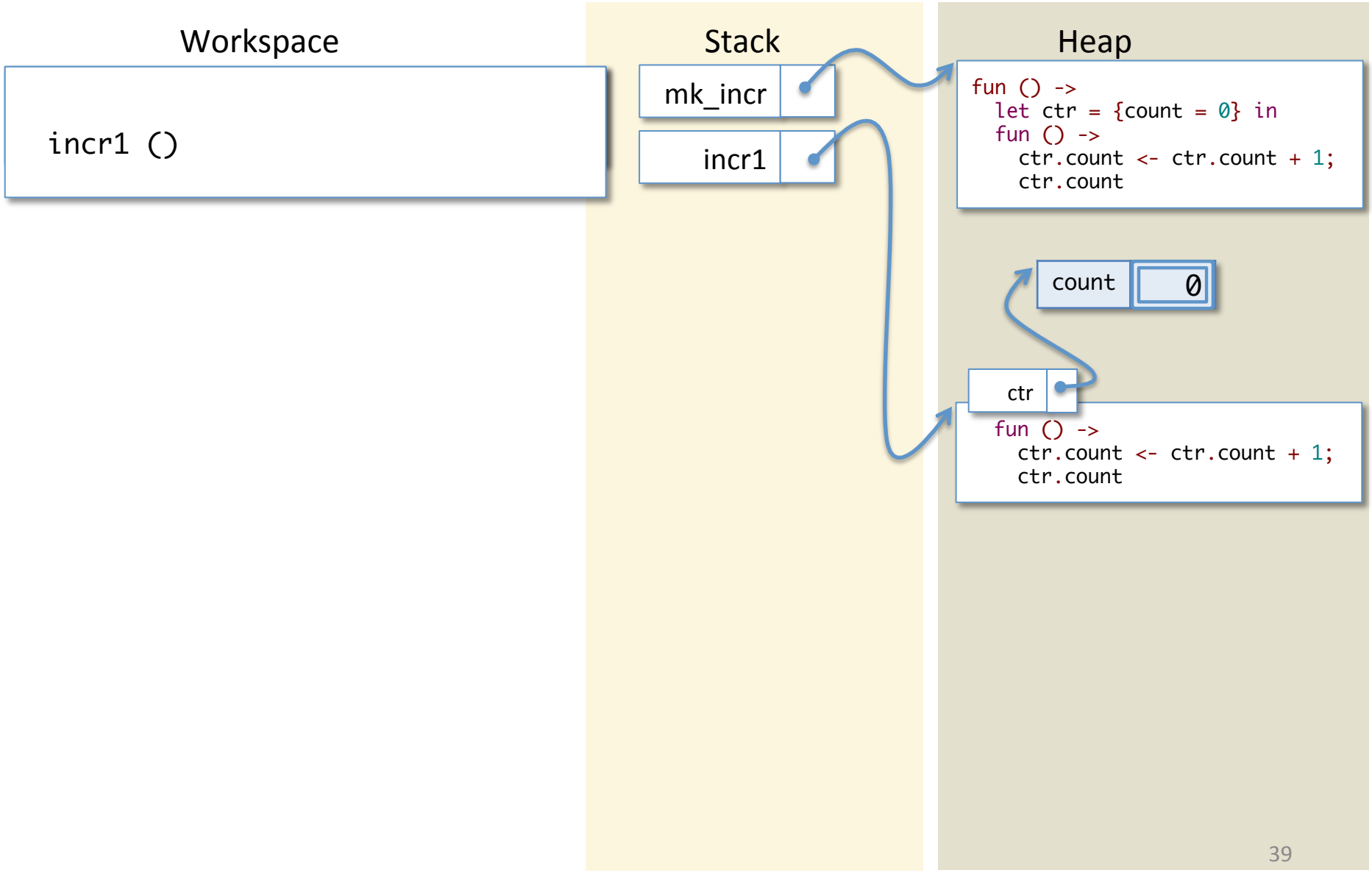
Local Functions



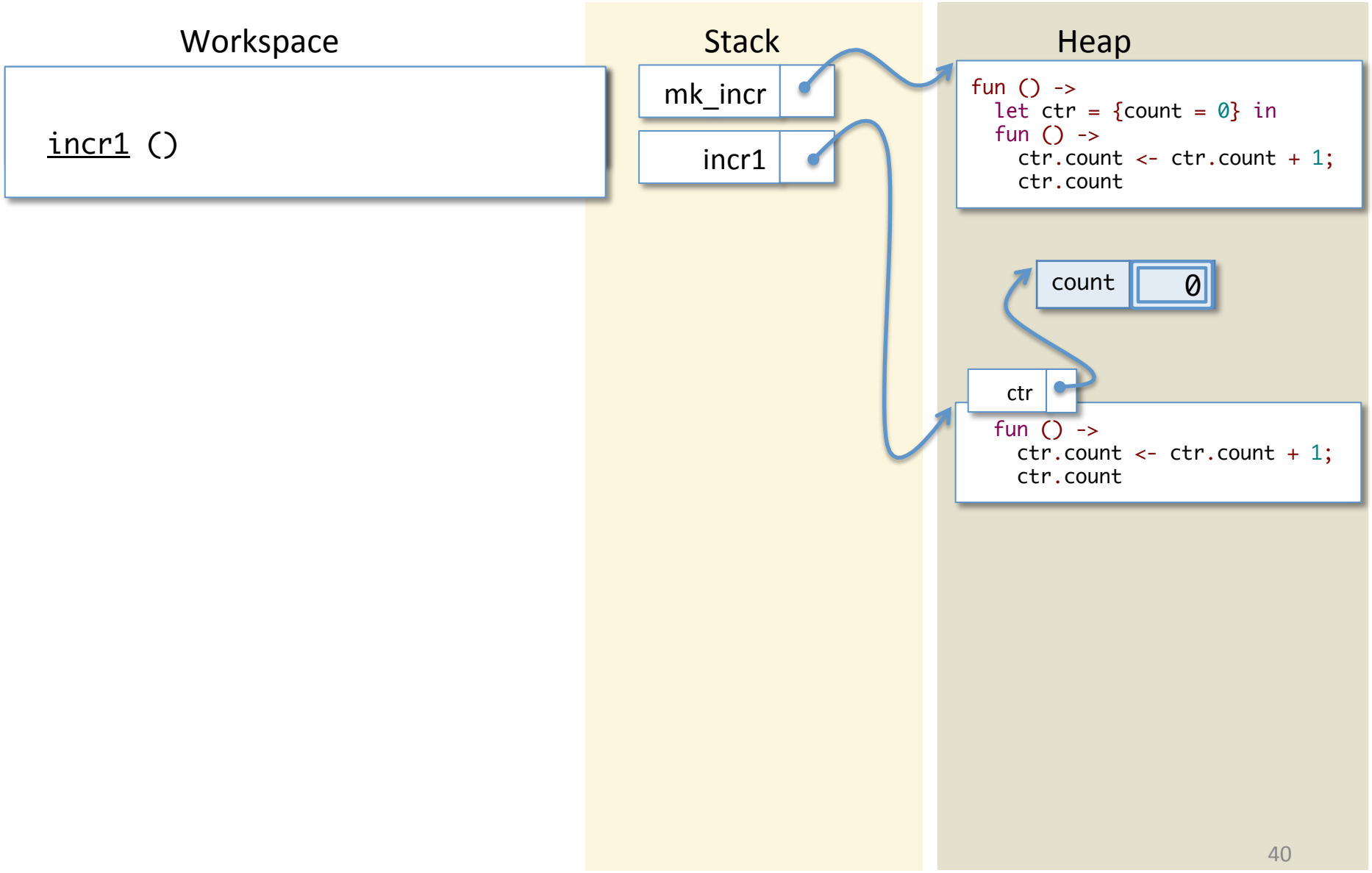
Local Functions



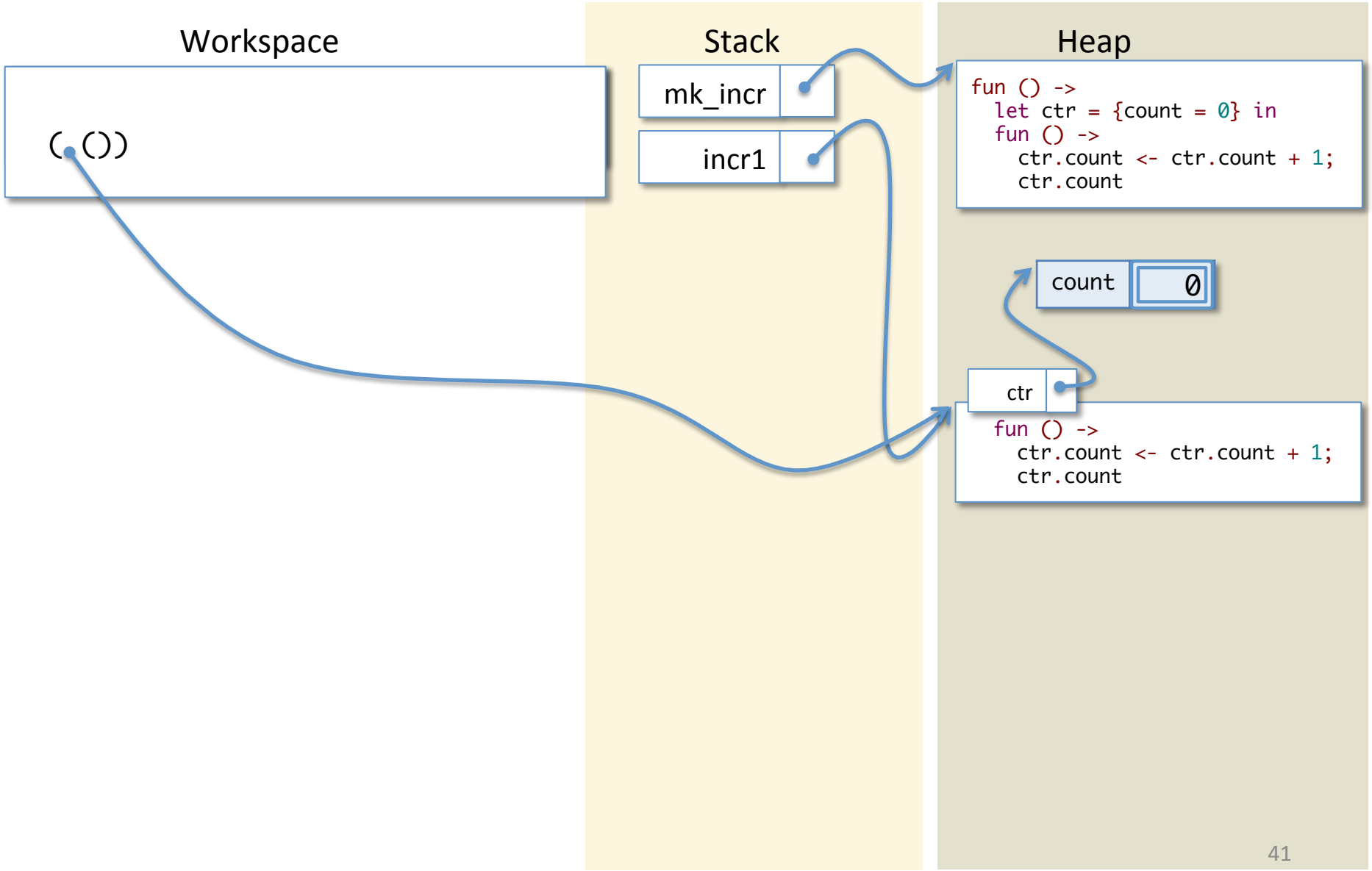
Now let's run "`incr1 ()`"



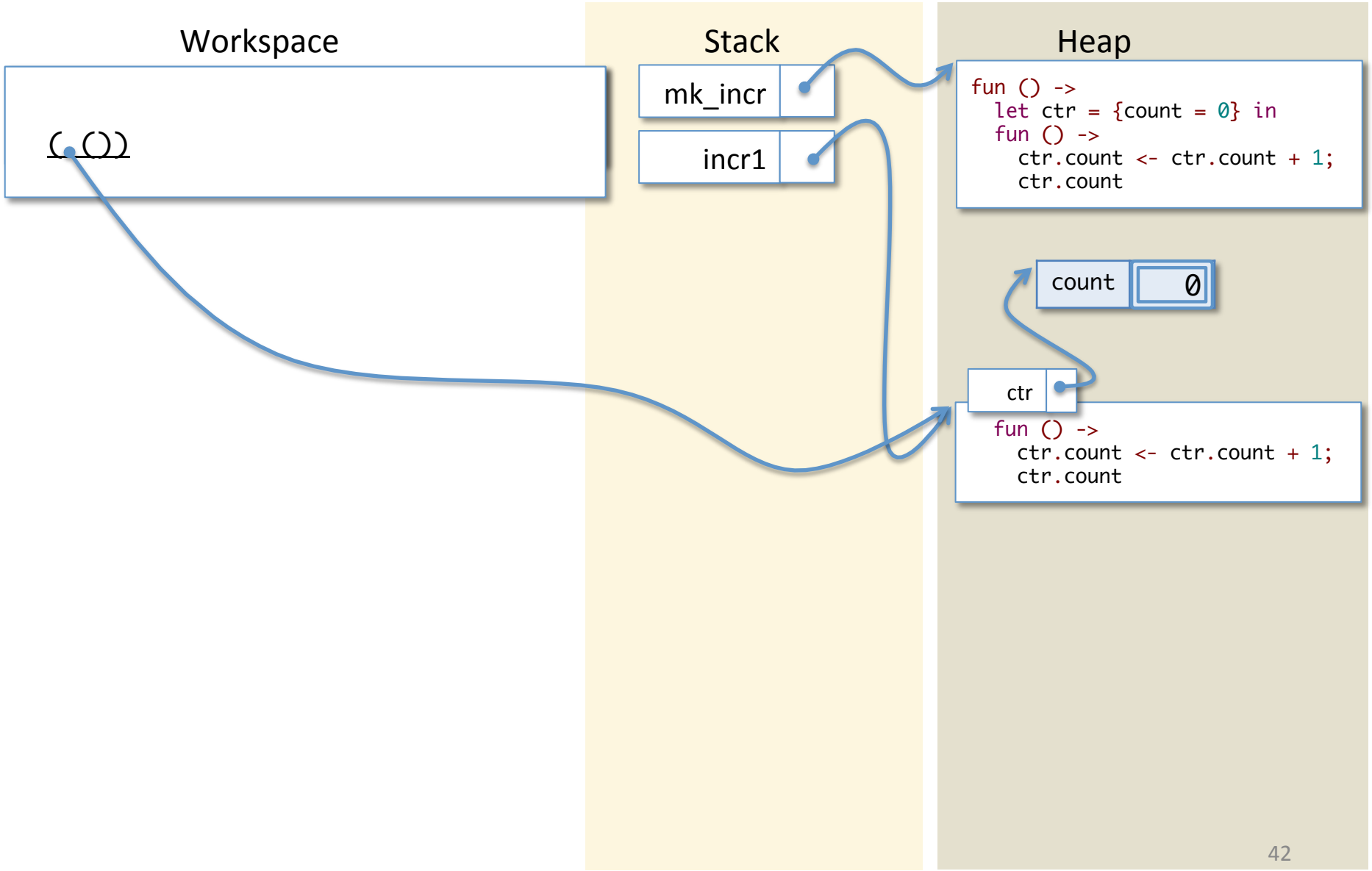
Now let's run "`incr1 ()`"



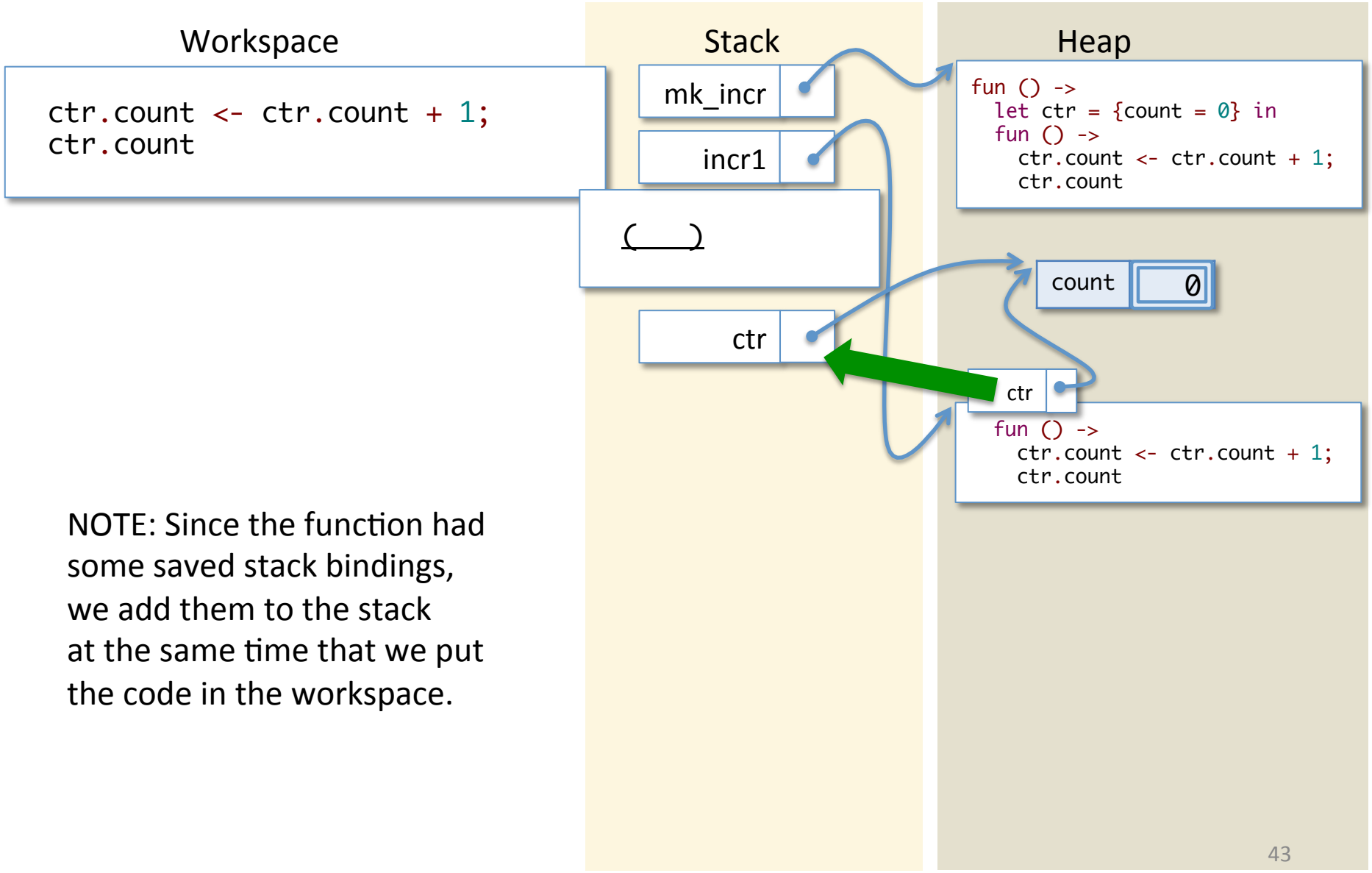
Now let's run "`incr1 ()`"



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

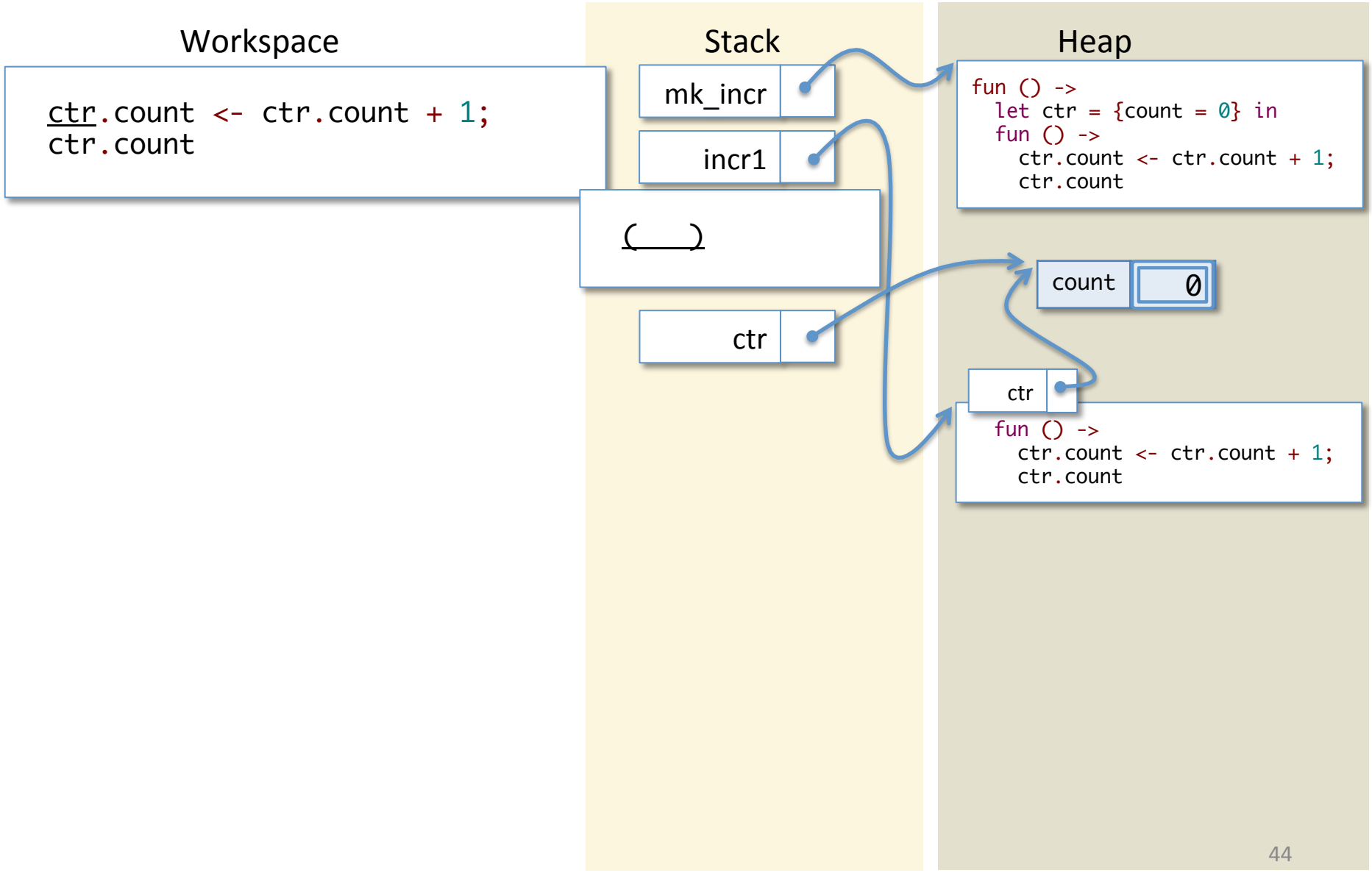


Now let's run "`incr1 ()`"

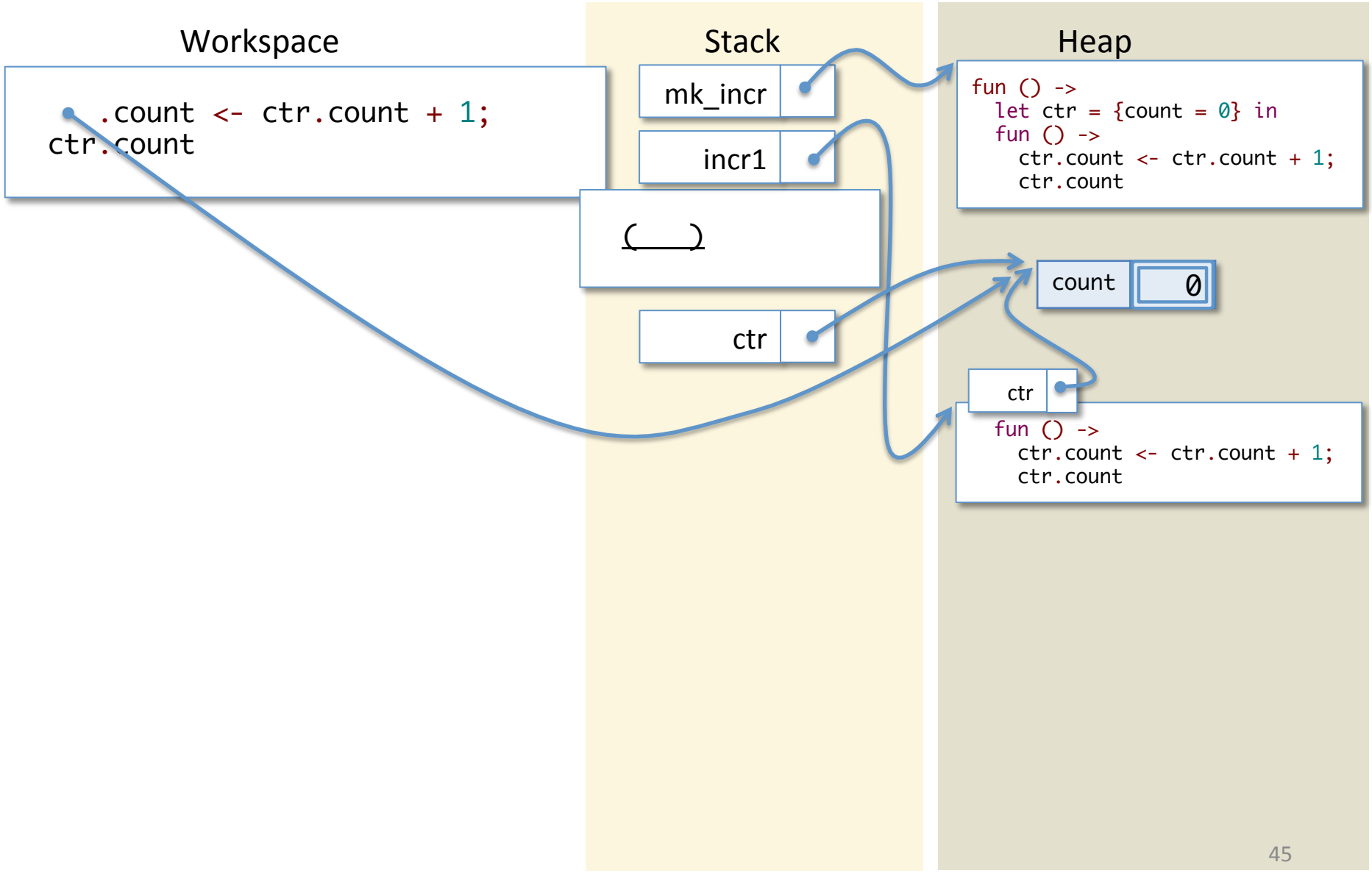


NOTE: Since the function had some saved stack bindings, we add them to the stack at the same time that we put the code in the workspace.

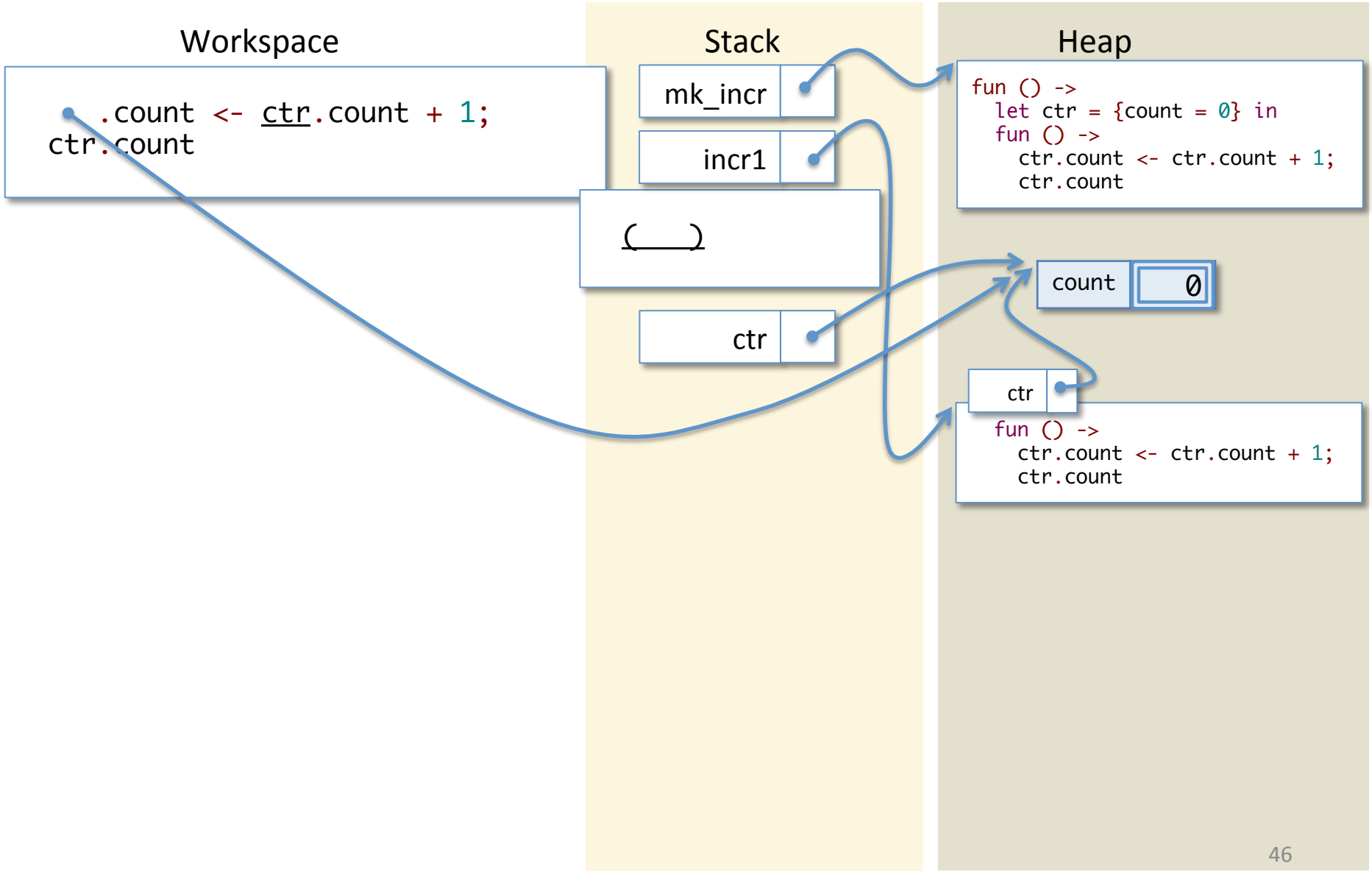
Now let's run "`incr1 ()`"



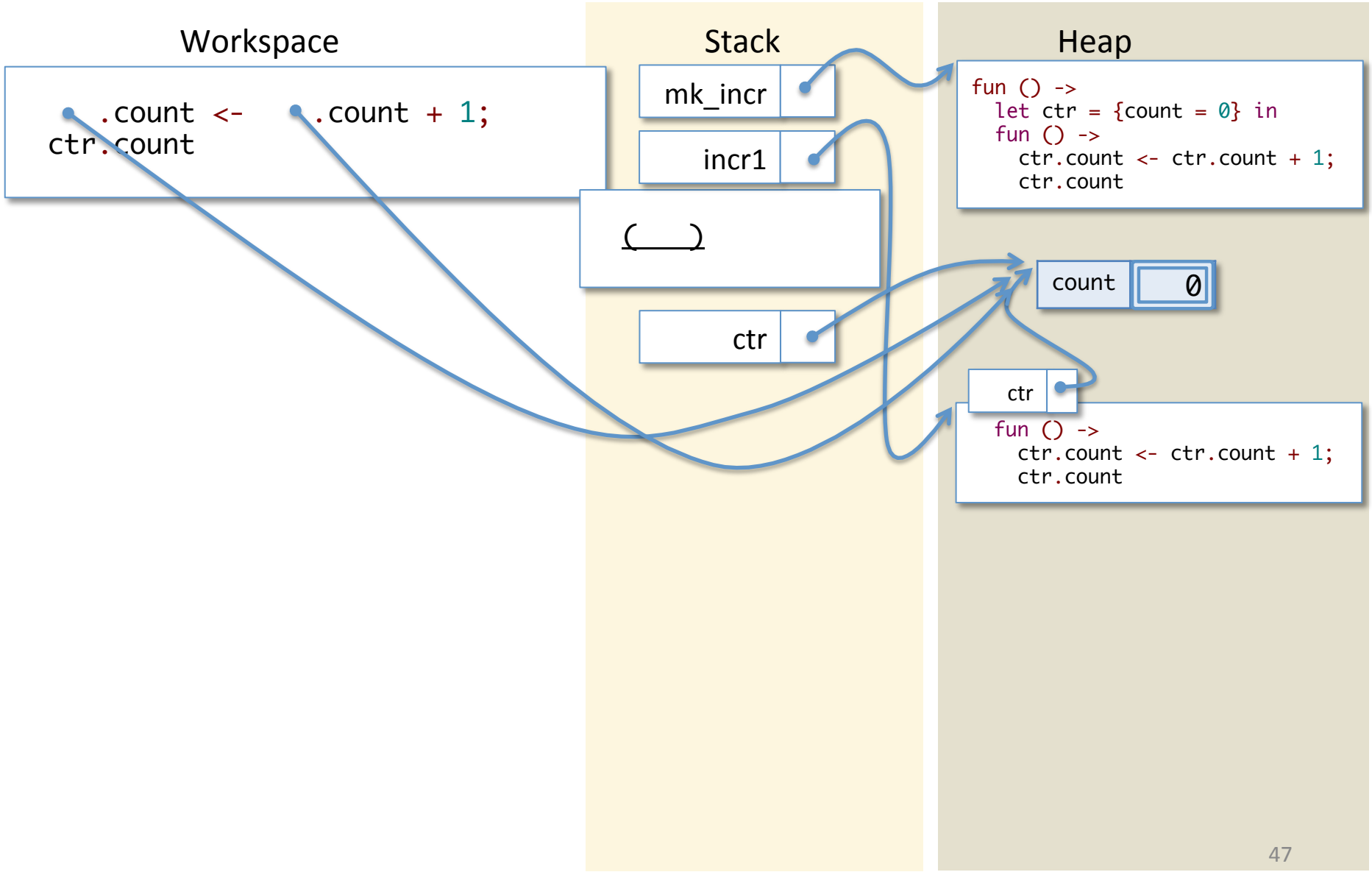
Now let's run "`incr1 ()`"



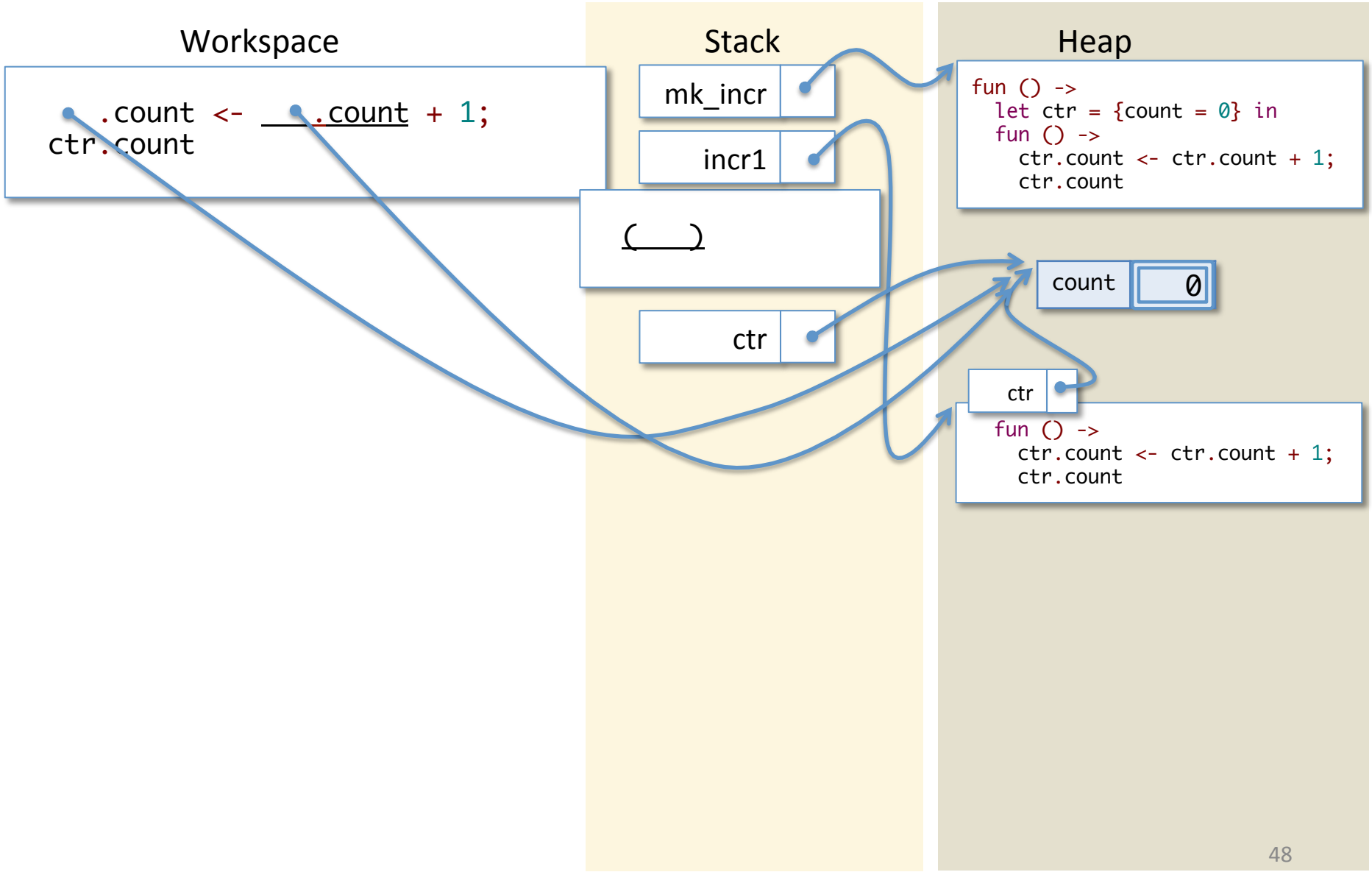
Now let's run "`incr1 ()`"



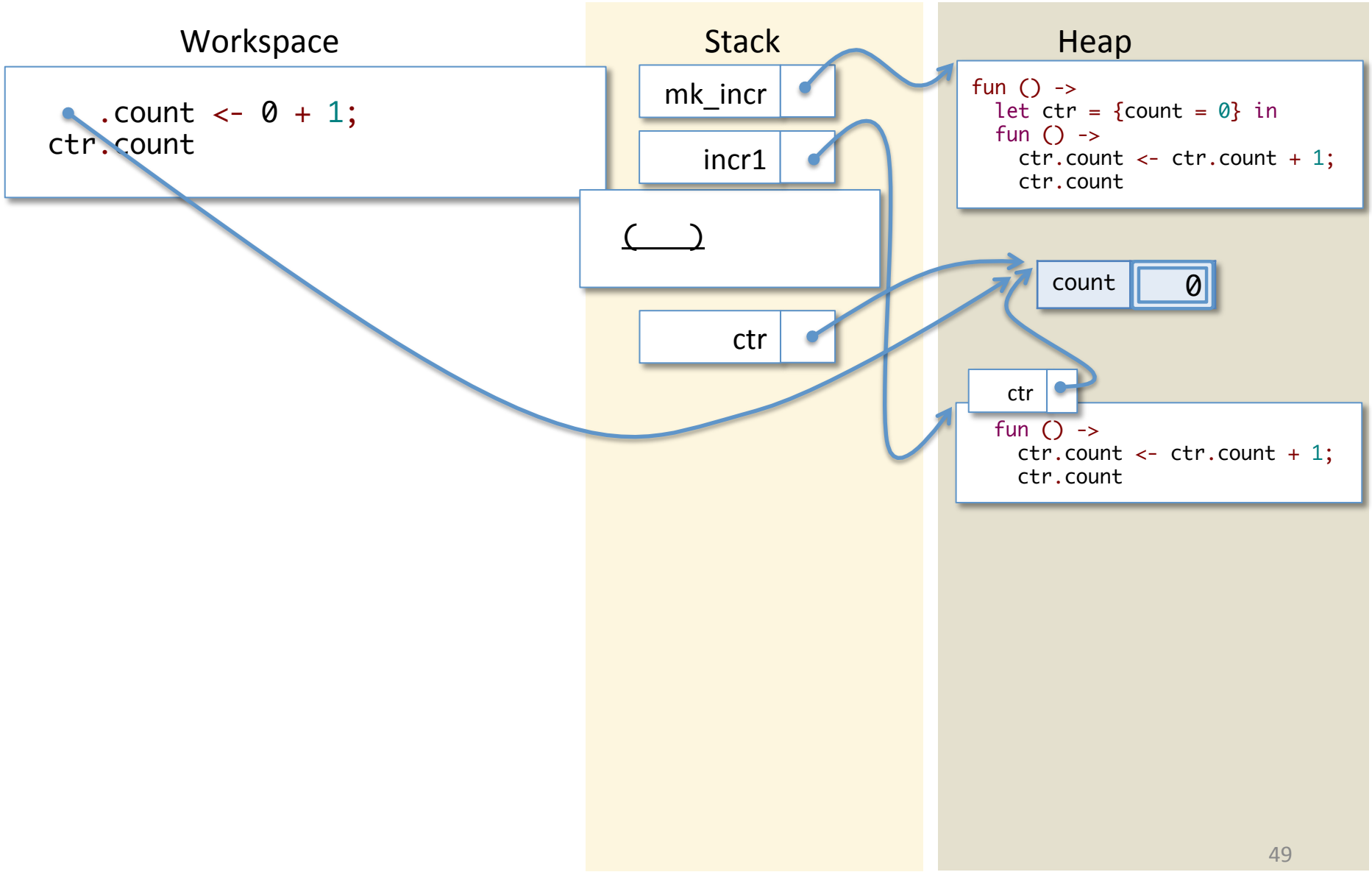
Now let's run "`incr1 ()`"



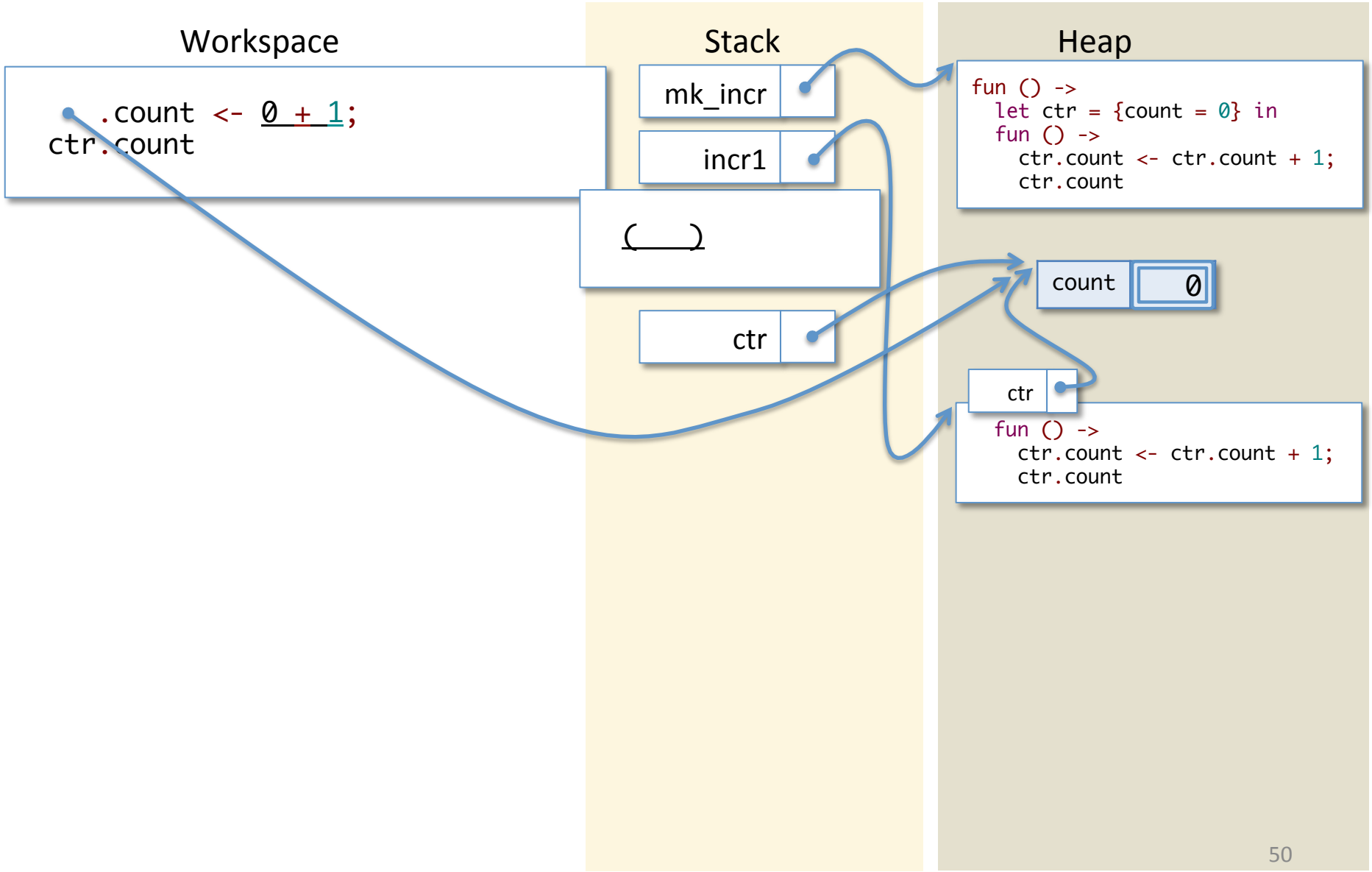
Now let's run "incr1 ()"



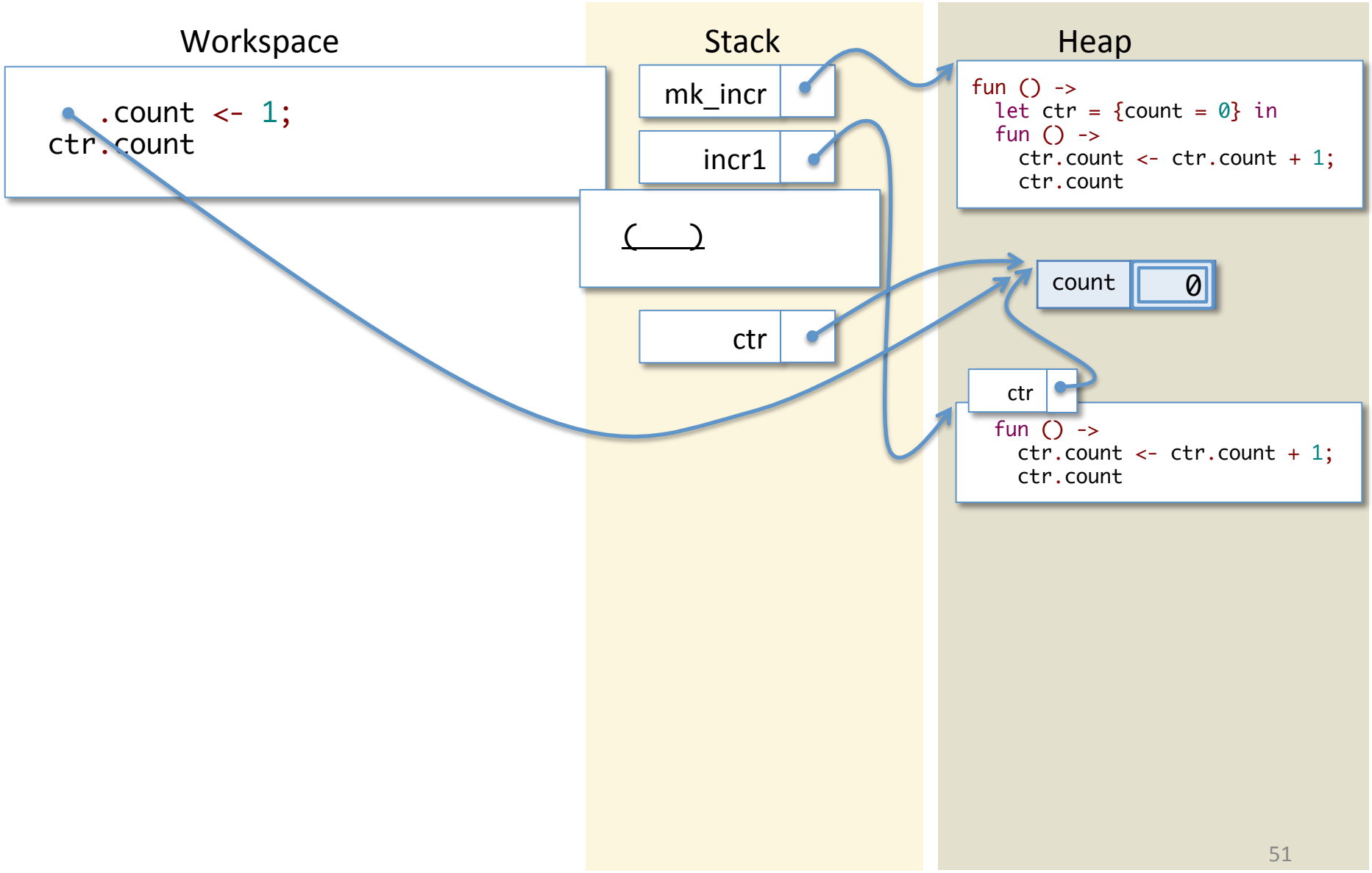
Now let's run "`incr1 ()`"



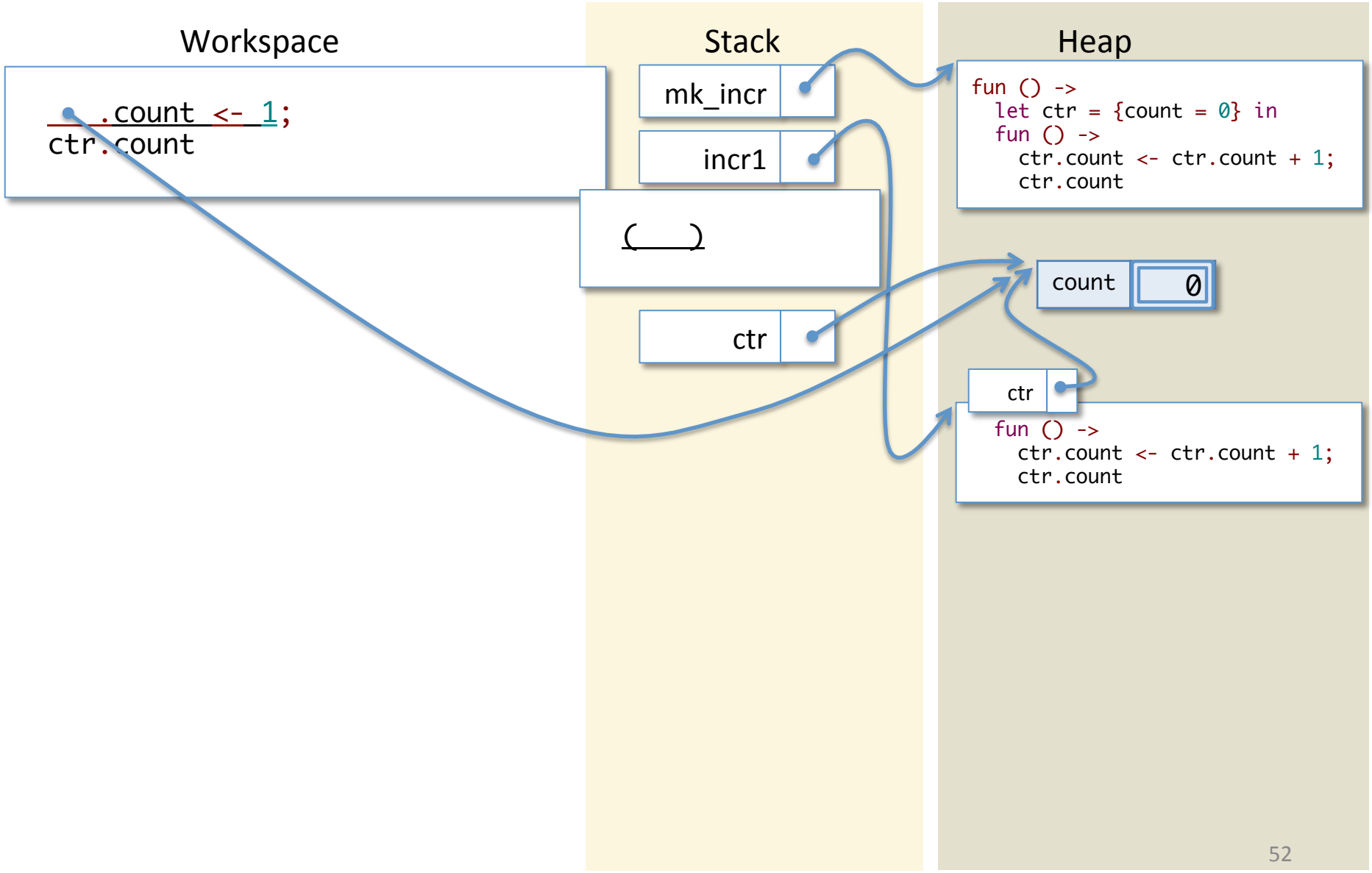
Now let's run "incr1 ()"



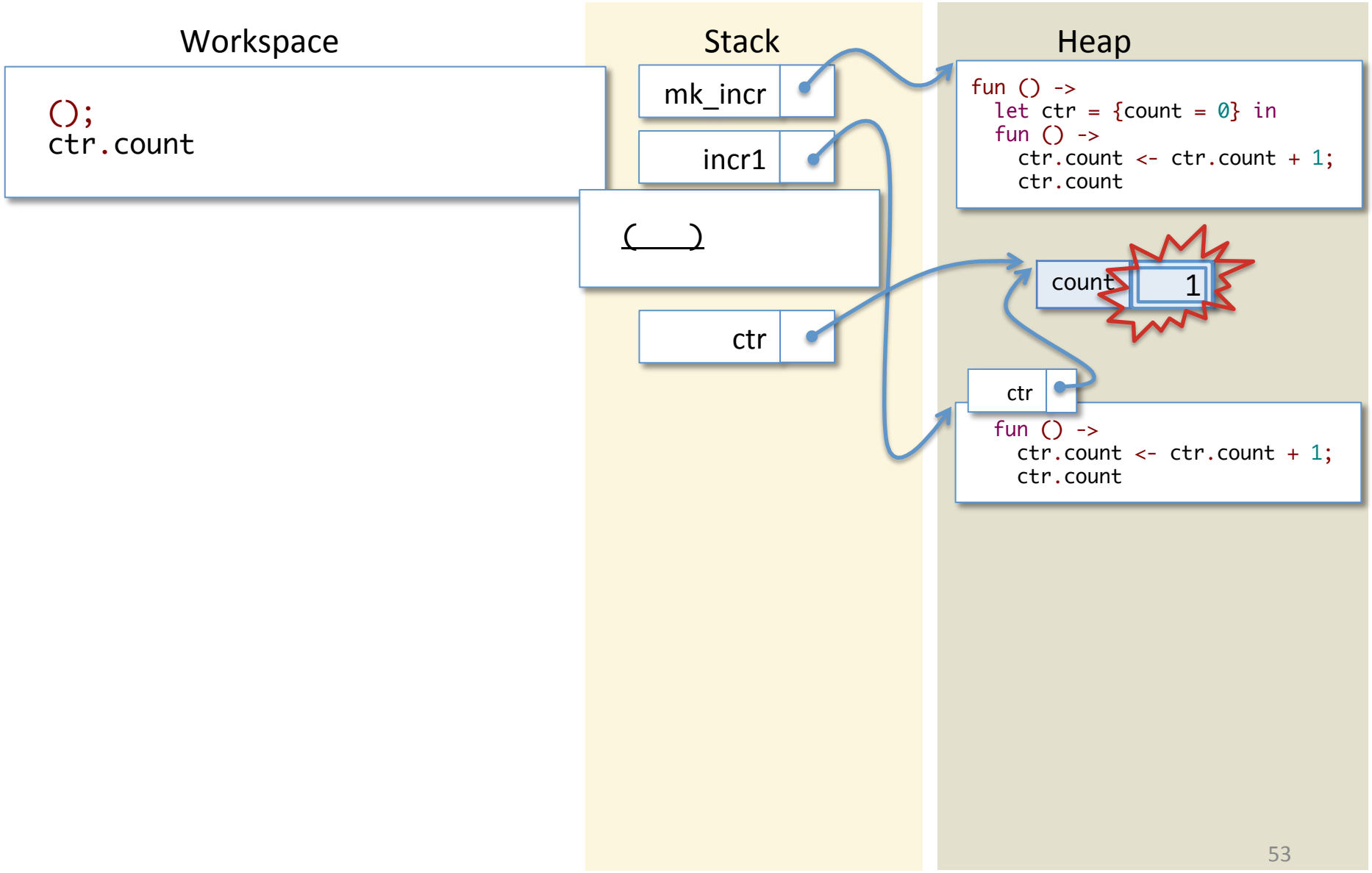
Now let's run "`incr1 ()`"



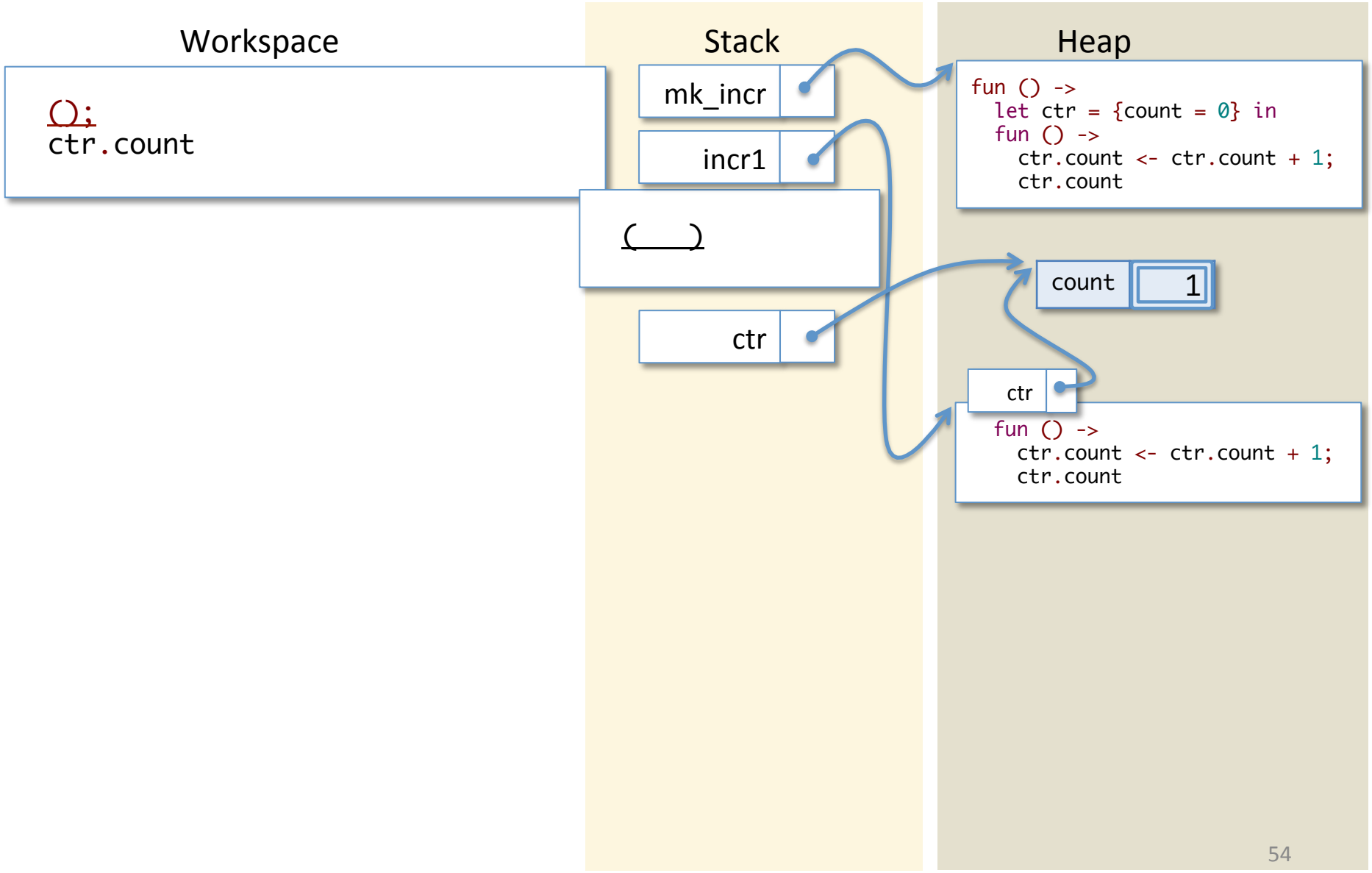
Now let's run "`incr1 ()`"



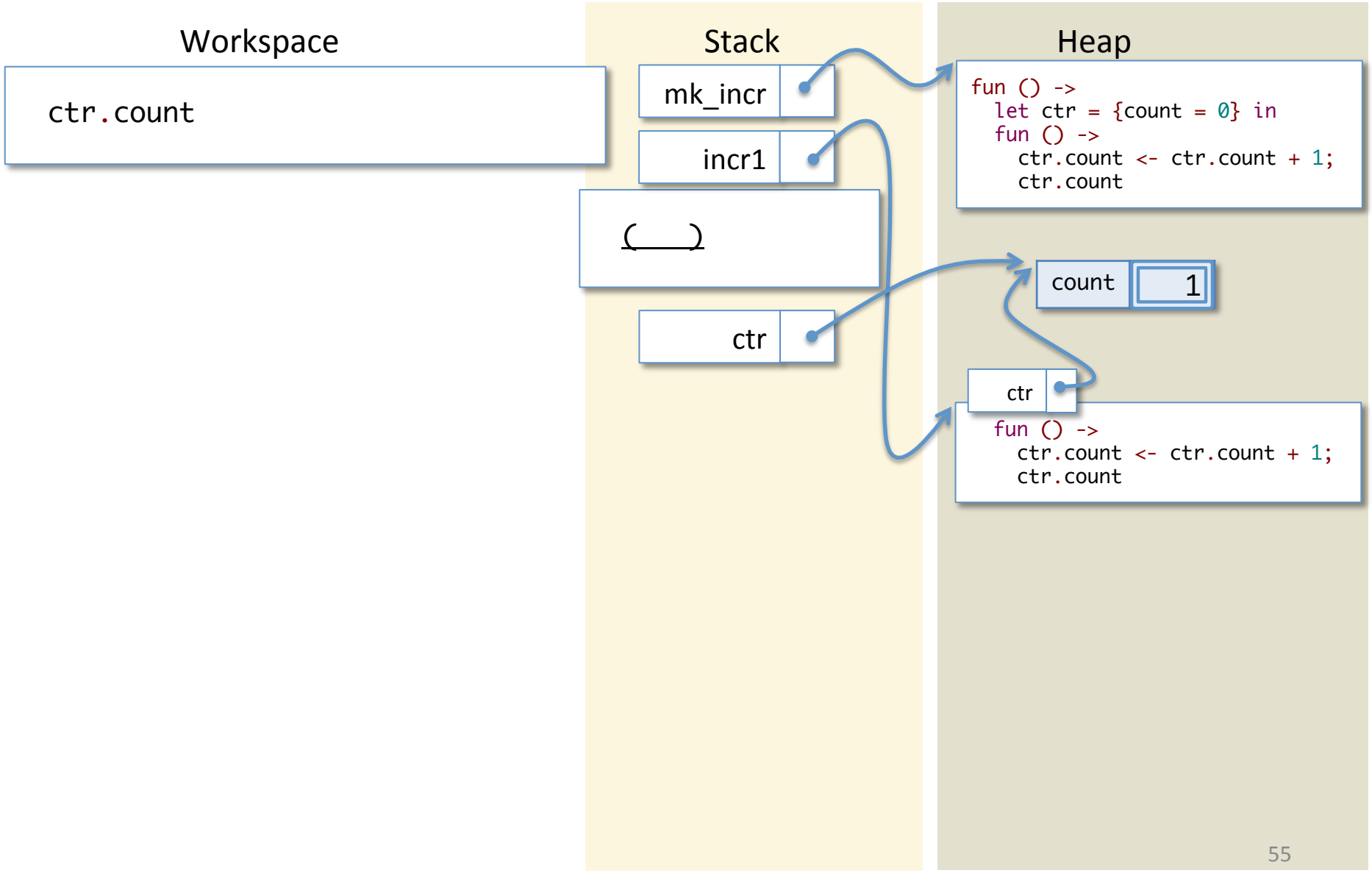
Now let's run "`incr1 ()`"



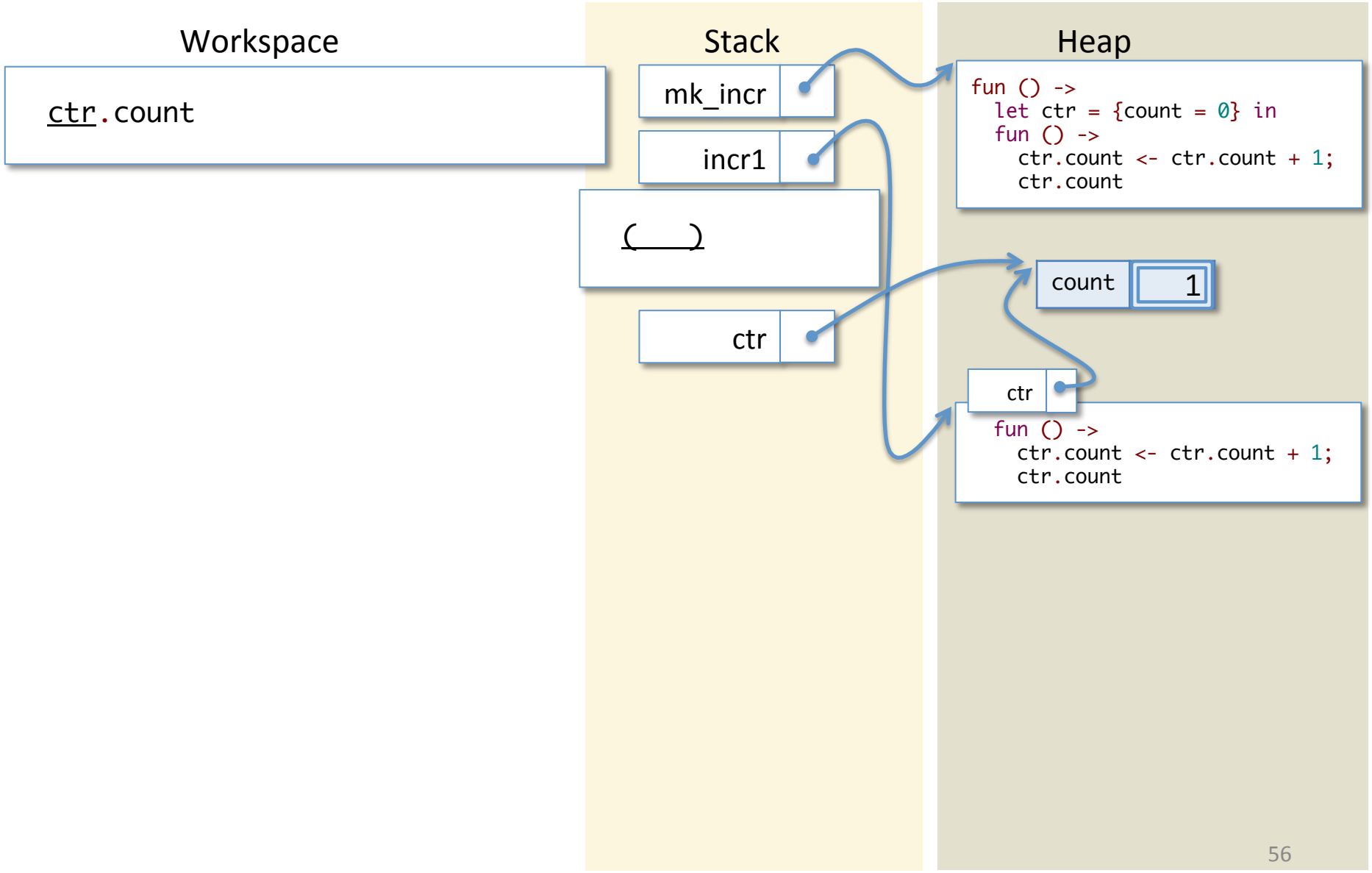
Now let's run "`incr1 ()`"



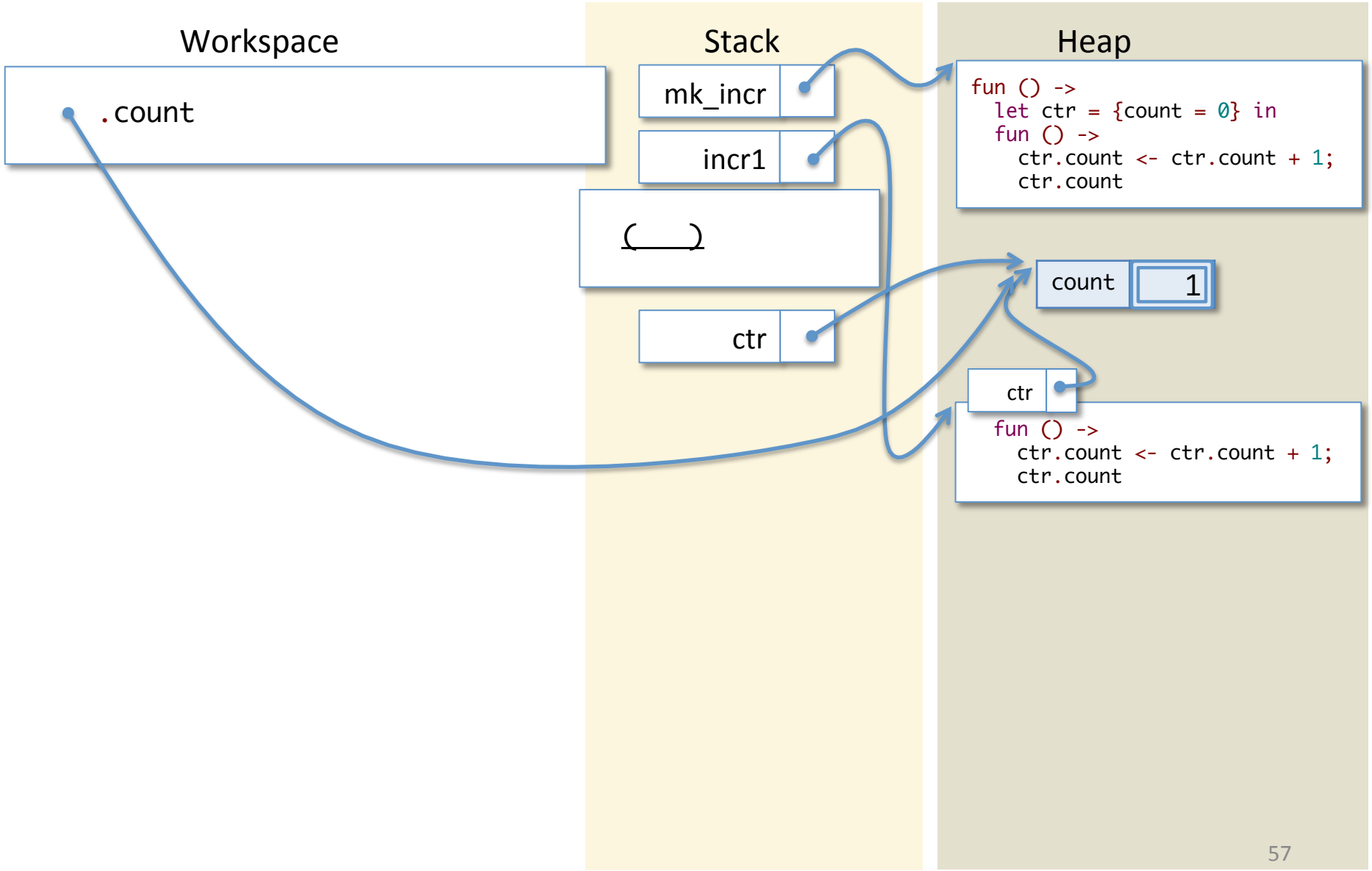
Now let's run "`incr1 ()`"



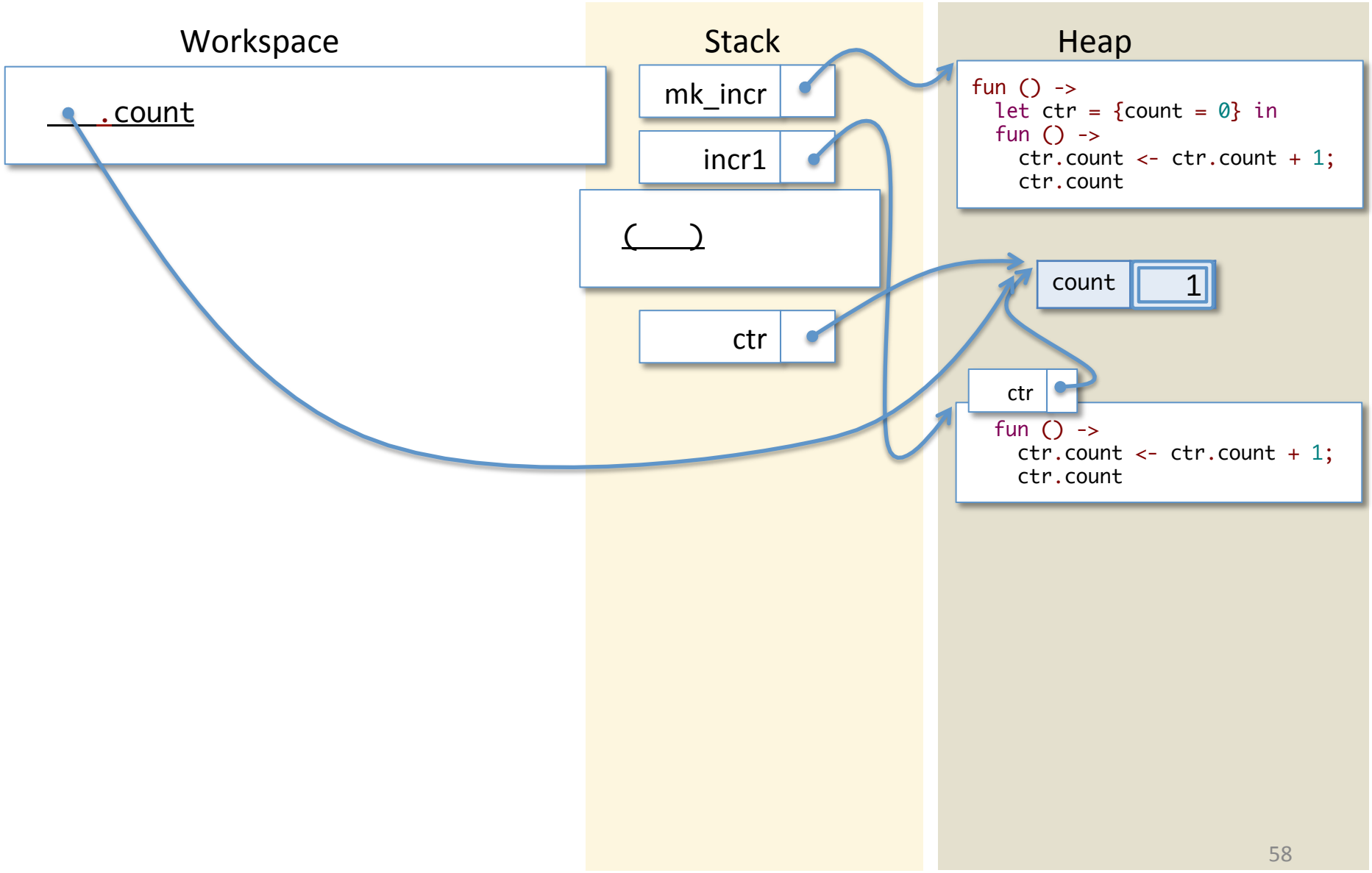
Now let's run "`incr1 ()`"



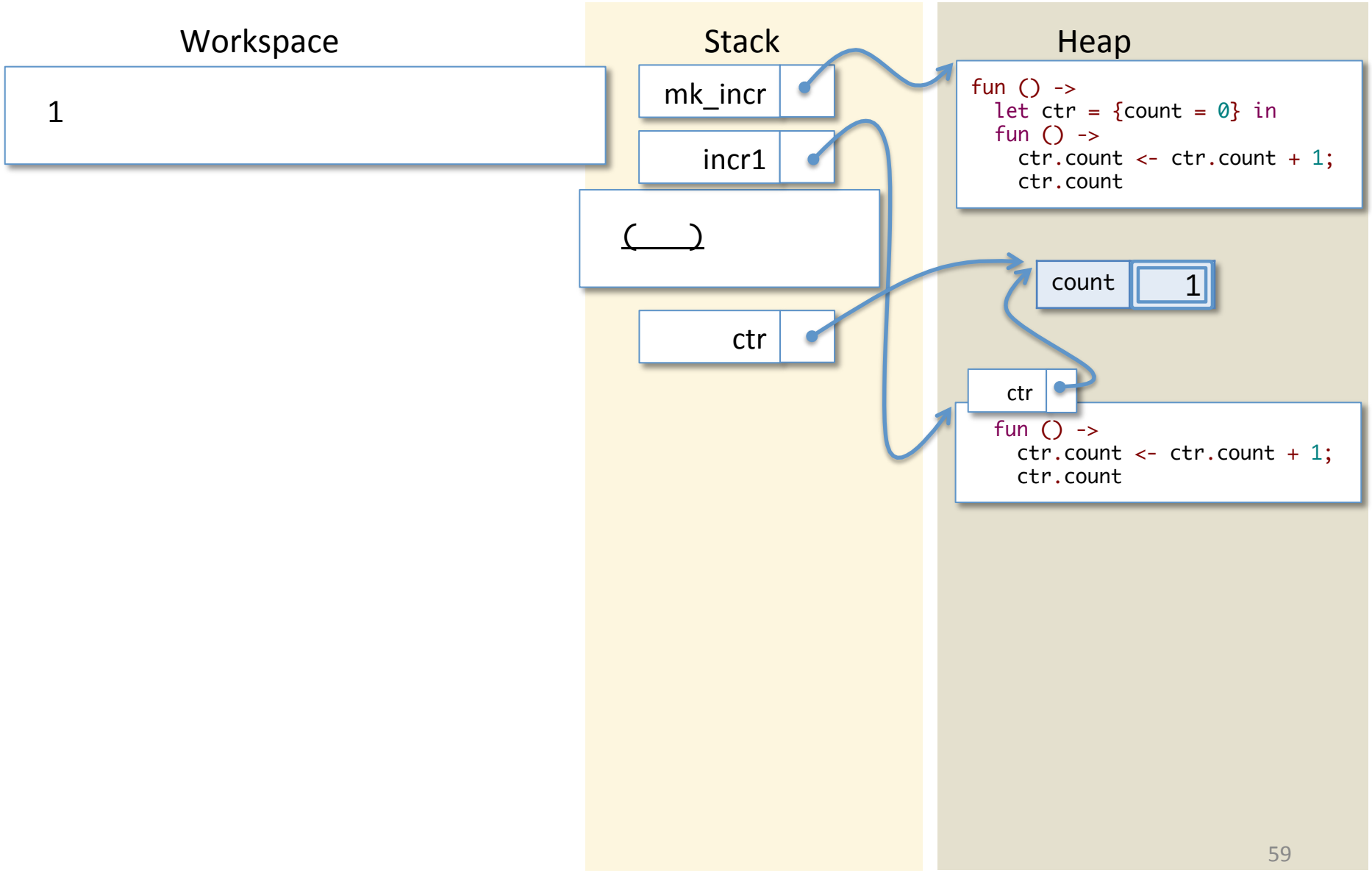
Now let's run "`incr1 ()`"



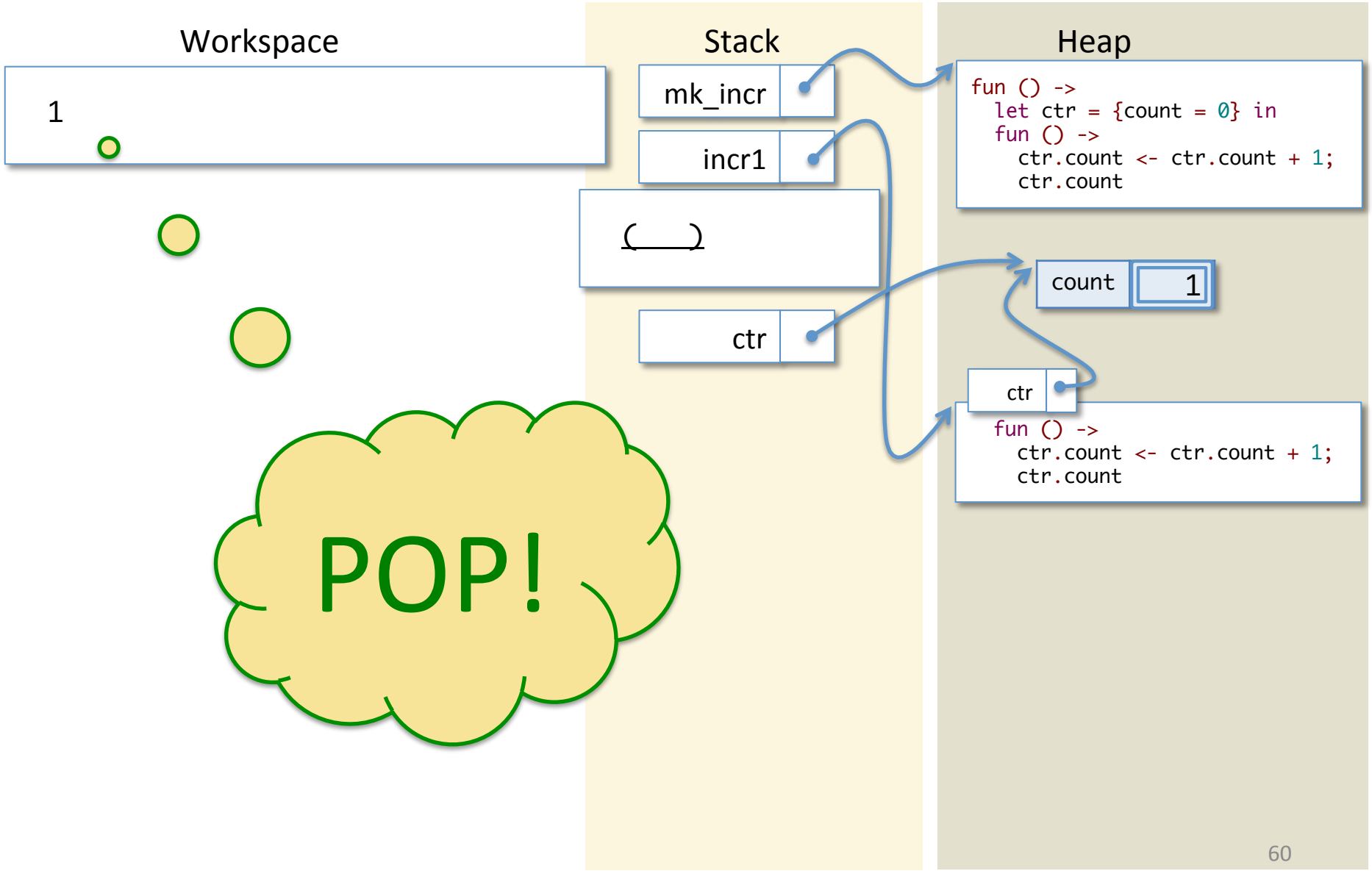
Now let's run "`incr1 ()`"



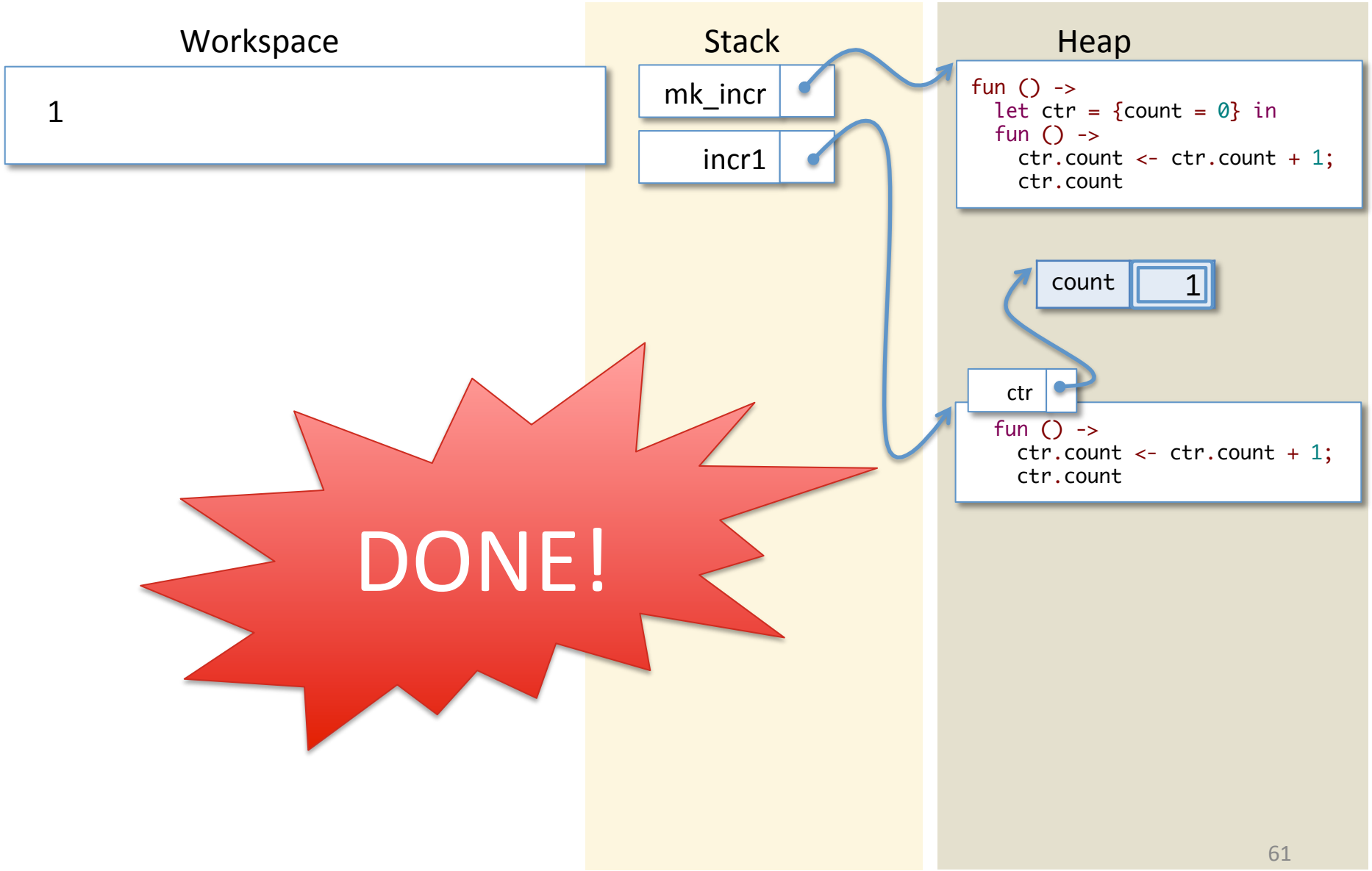
Now let's run "`incr1 ()`"



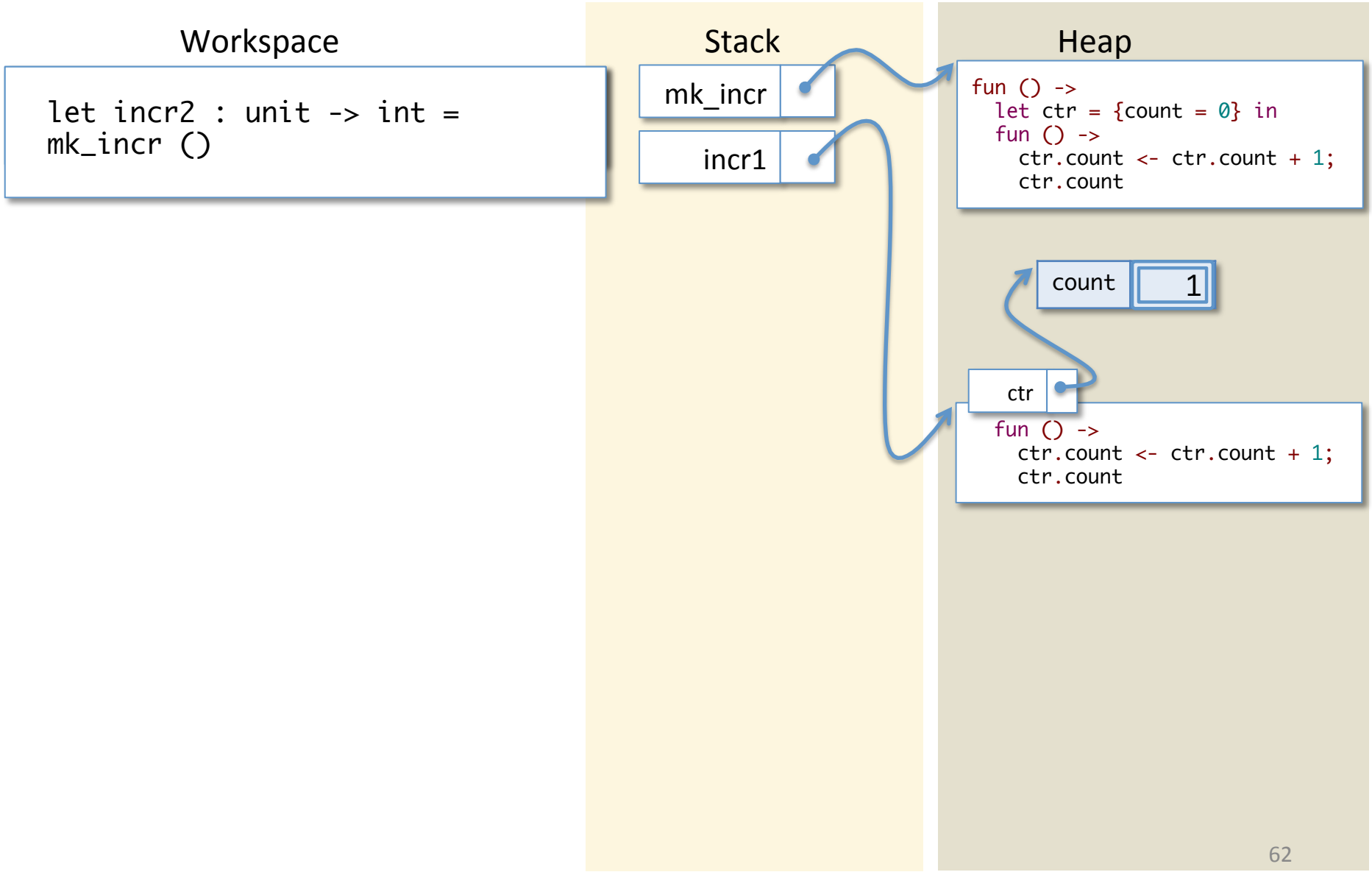
Now let's run "`incr1 ()`"



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

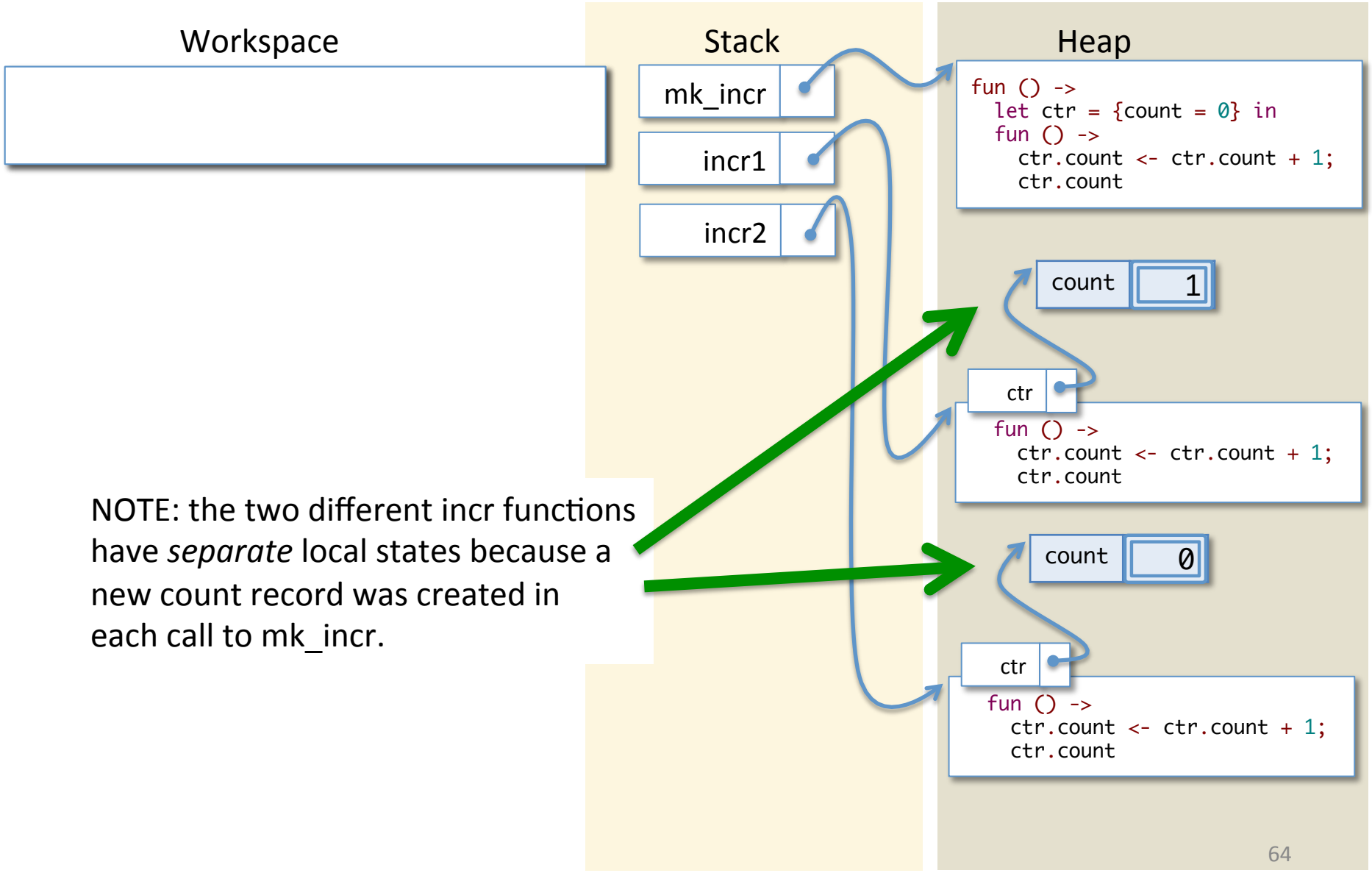


Now Let's run `mk_incr` again



…time passes…

After creating incr2



NOTE: the two different incr functions have *separate* local states because a new count record was created in each call to mk_incr.

One step further

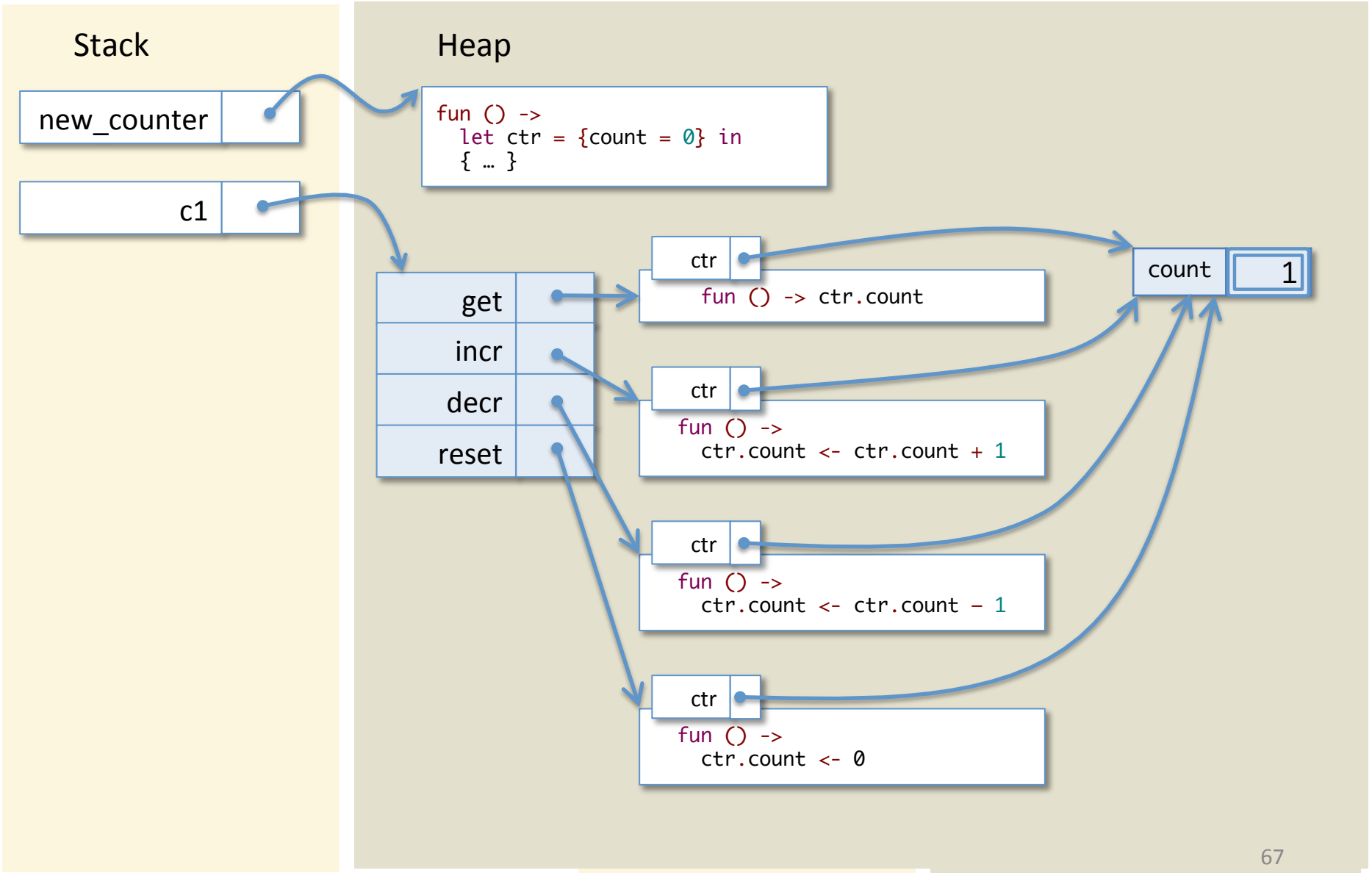
- mk_incr shows us how to create different instance of local state so that we can have several different counters.
- What if we want to bundle together *several* operations that share the same local state?
  - e.g. incr and decr operations that work on the same counter

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 = mk_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 ()))
```