Programming Languages and Techniques (CIS1200)

Lecture 17

Hidden State, Objects

Chapter 17

Announcements

- HW05 available soon, due *Tuesday*, March 18th (at 11.59pm)
  - Start early! This homework is difficult.
  - Tasks 0-1 can be done after class today
  - Tasks 2-4 can be done after class on Friday
  - Tasks 5-6 can be done after class on Monday

- Final Exam
  - Wednesday, May 7th, 9-11 am

Tail Recursion Recap

length (recursively)

```
(* Calculate the length of the queue recursively *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

As we've just seen, this implementation uses a lot of stack space if q is large.

Can we do better?

`length` (using iteration)

```
(* Calculate the length of the list using iteration *)
let length (q:'a queue) : int =
  let rec loop (no:'a qnode option) (len:int) : int =
      begin match no with
        | None -> len
        | Some n -> loop n.next (1+len)
      end
  in
    loop q.head 0
```

This implementation of `length` also uses a helper function, `loop`:

- This loop takes an extra argument, `len`, called the *accumulator*
- Unlike the previous solution, the computation happens "on the way down" as opposed to "on the way back up"
- Note that `loop` will always be called in an otherwise-empty workspace—the results of the call to `loop` never need to be used to compute another expression. In contrast, we had (1 + (`loop` …)) in the recursive version.

Crucial Observations

- Tail call optimization lets the stack take only a *fixed amount of space*.

- The recursive call to `loop` effectively updates the stack bindings in place.
  - We can think of these bindings as the *state* being modified by each iteration of the loop.

- These two properties are the essence of iteration.
  - They are the difference between general recursion and iteration

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

ANSWER: 3

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

0

The value 2 is returned

0%

The value 0 is returned

0%

StackOverflow

0%

Your program hangs

0%

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

ANSWER: 4

Infinite Loops

```
(* Accidentally go into an infinite loop… *)
let accidental_infinite_loop (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
```

- This program will go into an infinite loop.
- Unlike a non-tail-recursive program, which uses some space on each recursive call, there is no resource being exhausted, so the program will "silently diverge" and simply never produce an answer…

More iteration examples

to_list

print

chop

`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 current with
      | None -> …  (* iteration complete, produce result *)

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

Tail Recursion & Iteration

- A function call is in *tail position* if, when the call is evaluated in the workspace, there is no more work to be done before returning.
  - i.e., the result of the workspace is the result of the call

```
f x                  (* ← this is in tail position *)
```

```
if … then
    f x            (* ← this f is in tail position *)
else
   (f x) + 1       (* ← this f is not in tail position *)
```

```
begin match … with
 | … -> f x          (* ← this f is in tail position *)
 | … -> g (f x)      (* ← this f is not in tail position,*)
end                  (*    but g is in tail position *)
```

```
(g x) || (f x)  (* ← this f is in tail position, g isn't *)
```

```
cmd ; (f x)        (* ← this f is in tail position *)
```

Hidden State

Encapsulating State

An “incr” function

A function 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 directly modify count

Using Hidden State

Better: Make a function that creates a counter state plus 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 ()
```

17: What number is printed by this program?

0

1 0%

2 0%

3 0%

other 0%

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

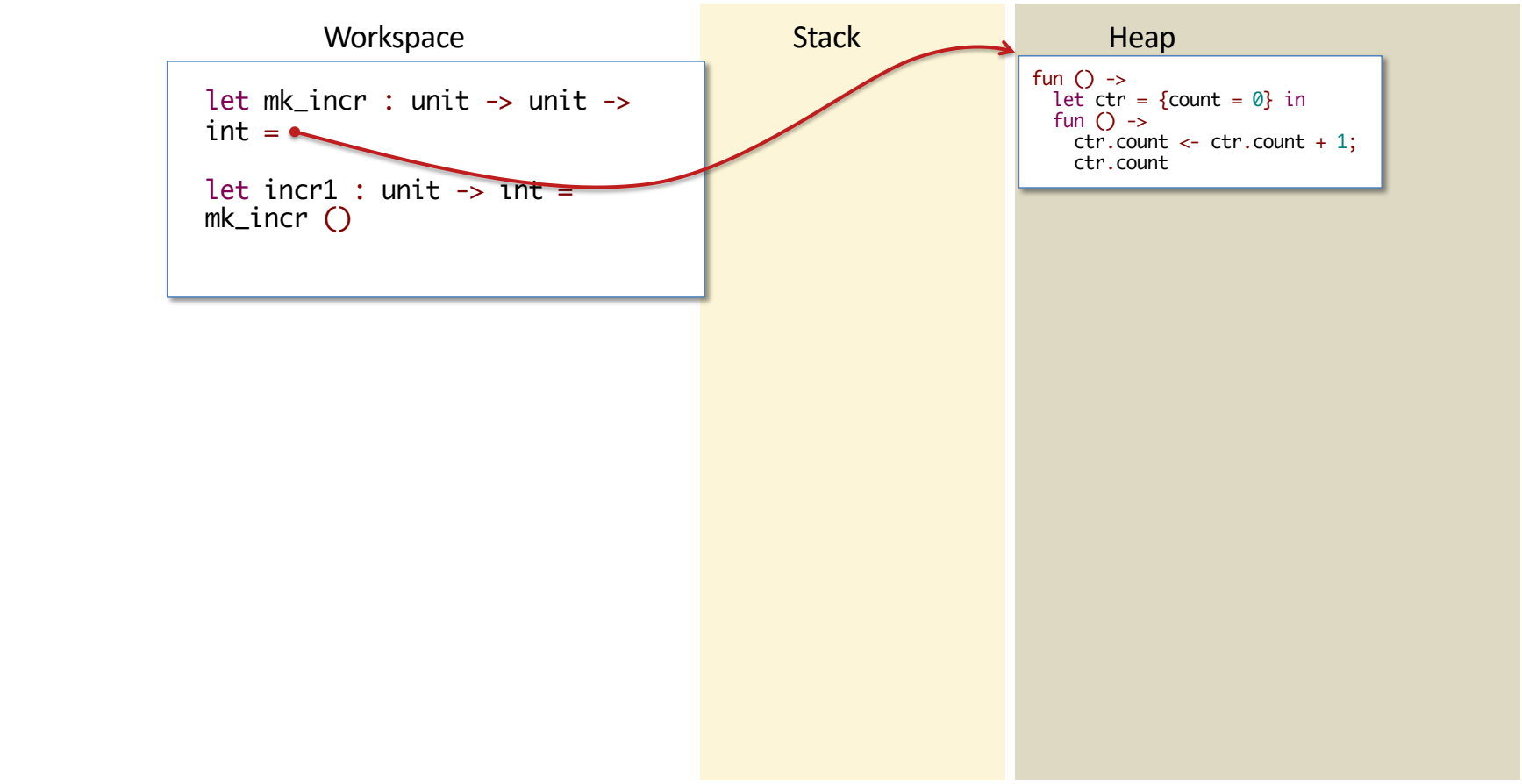
```
let mk_incr : unit -> unit ->
int = fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count


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

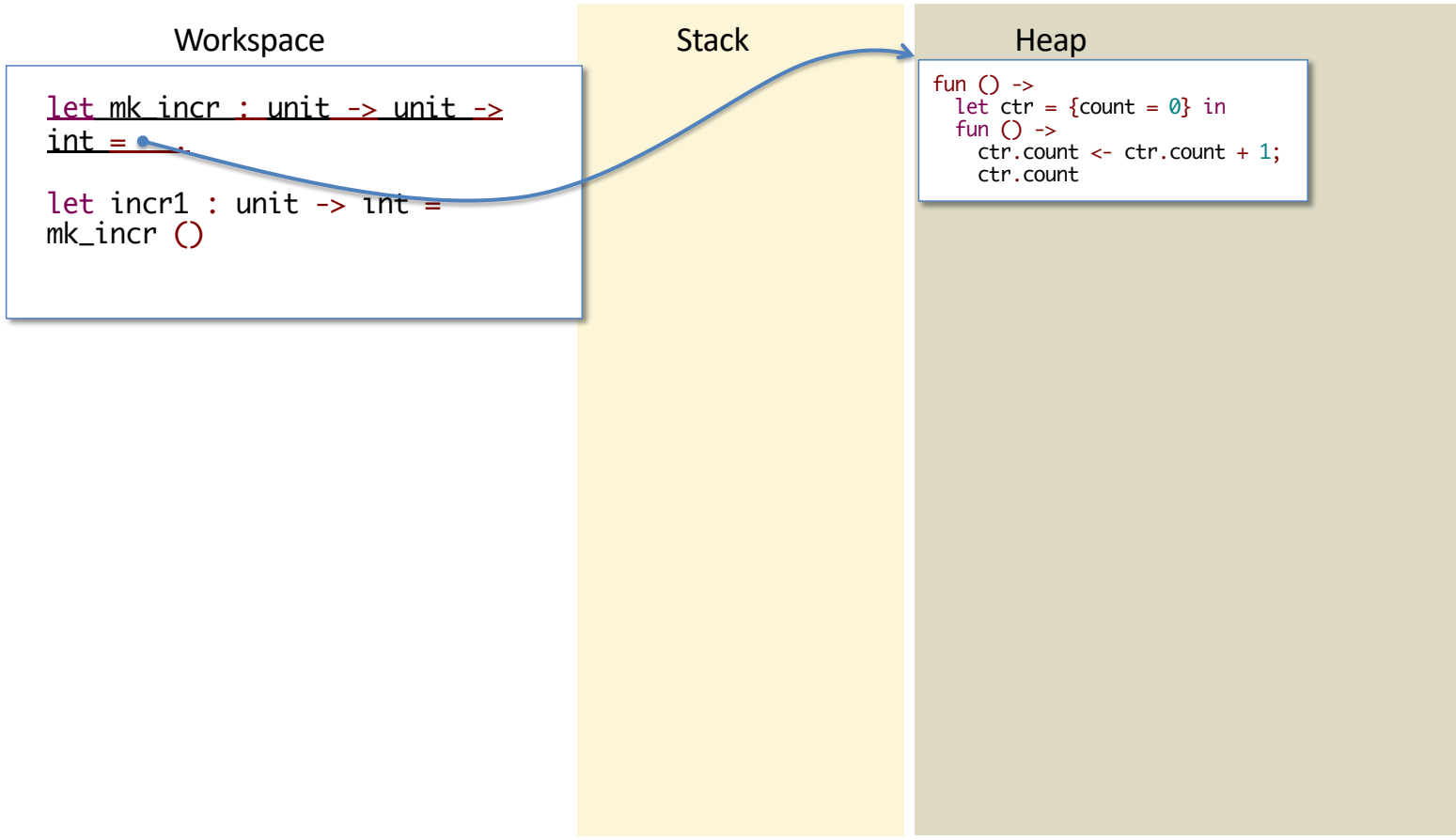
Stack

Heap

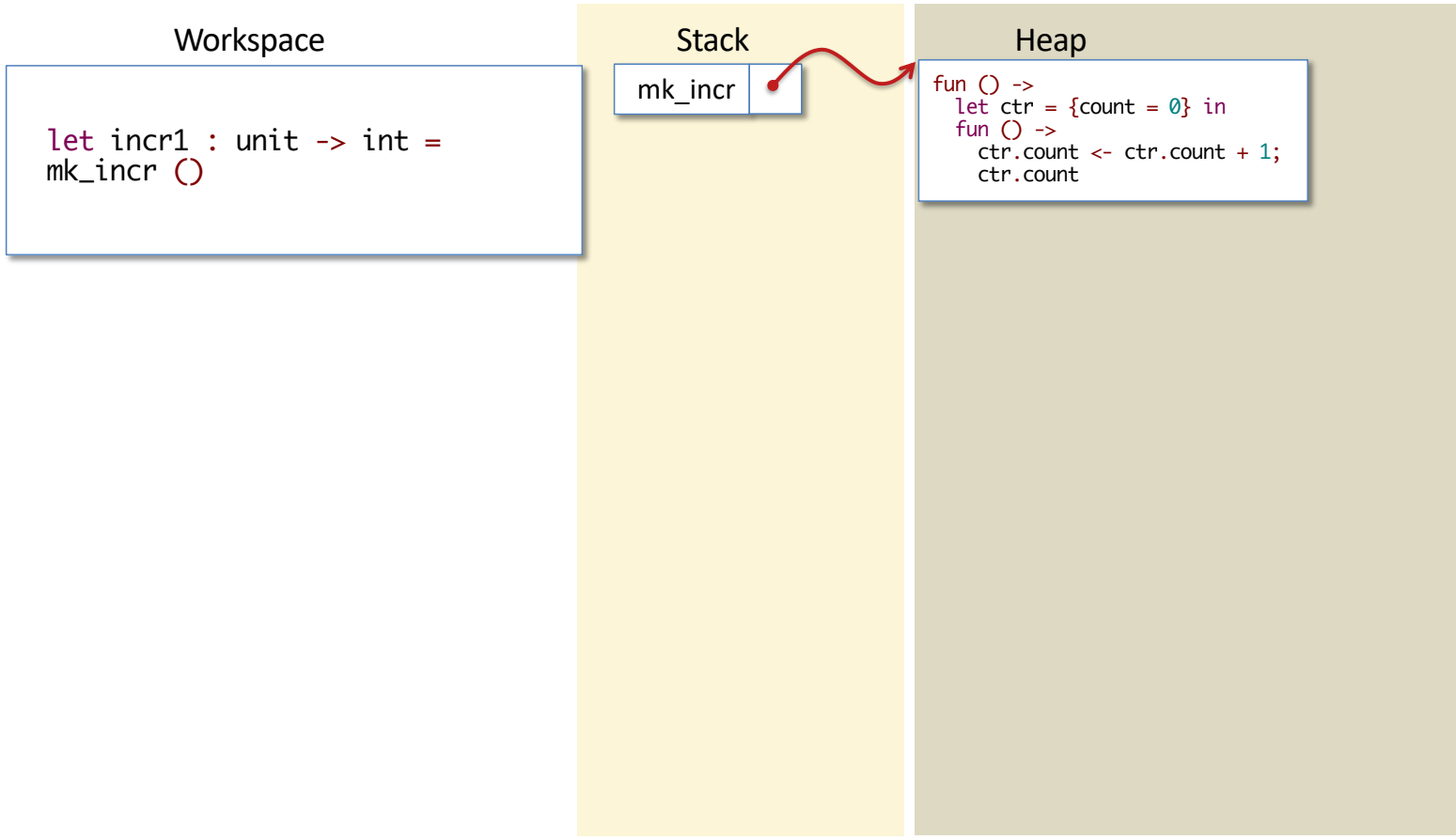
Running mk_incr



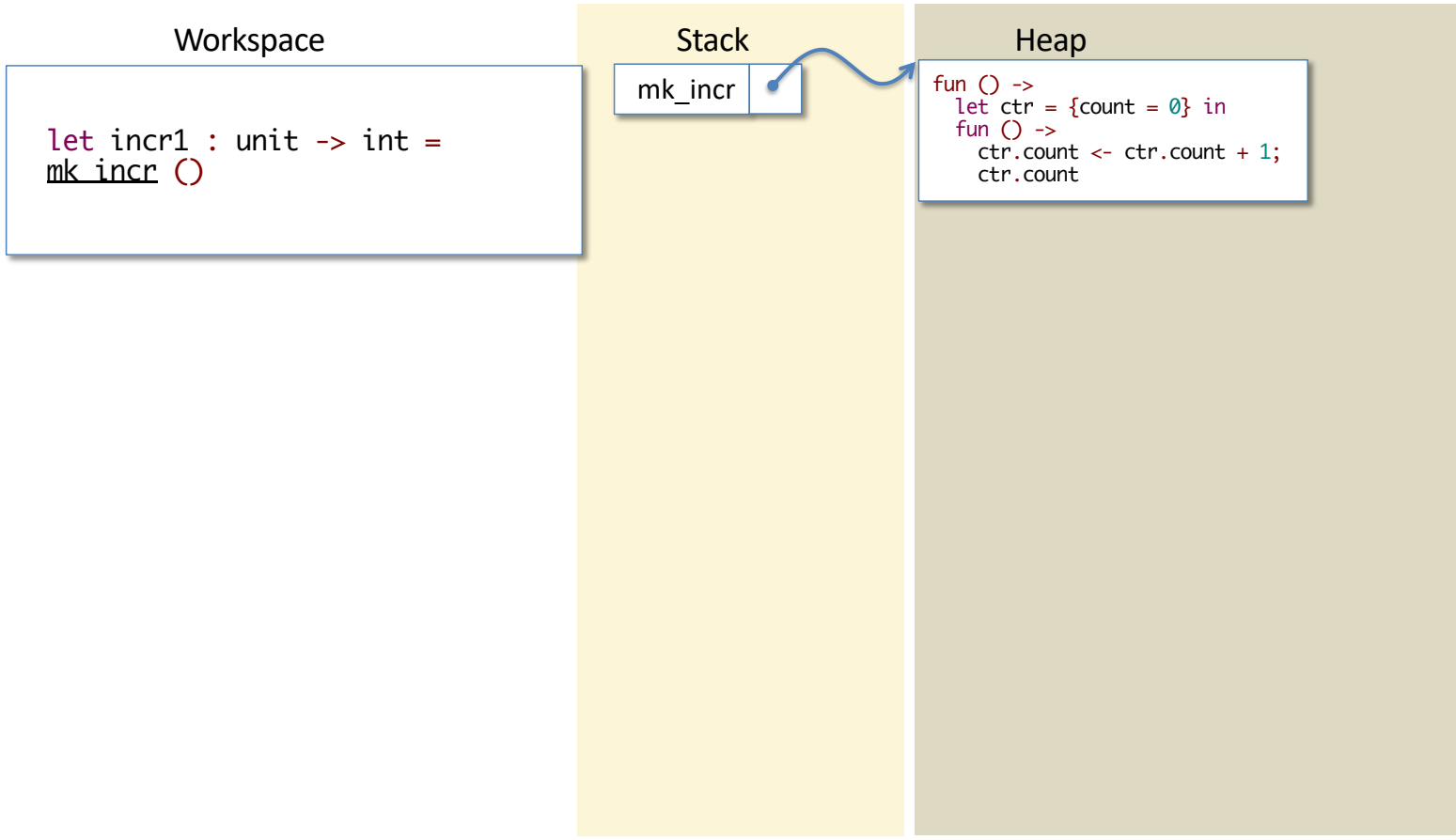
Running mk_incr



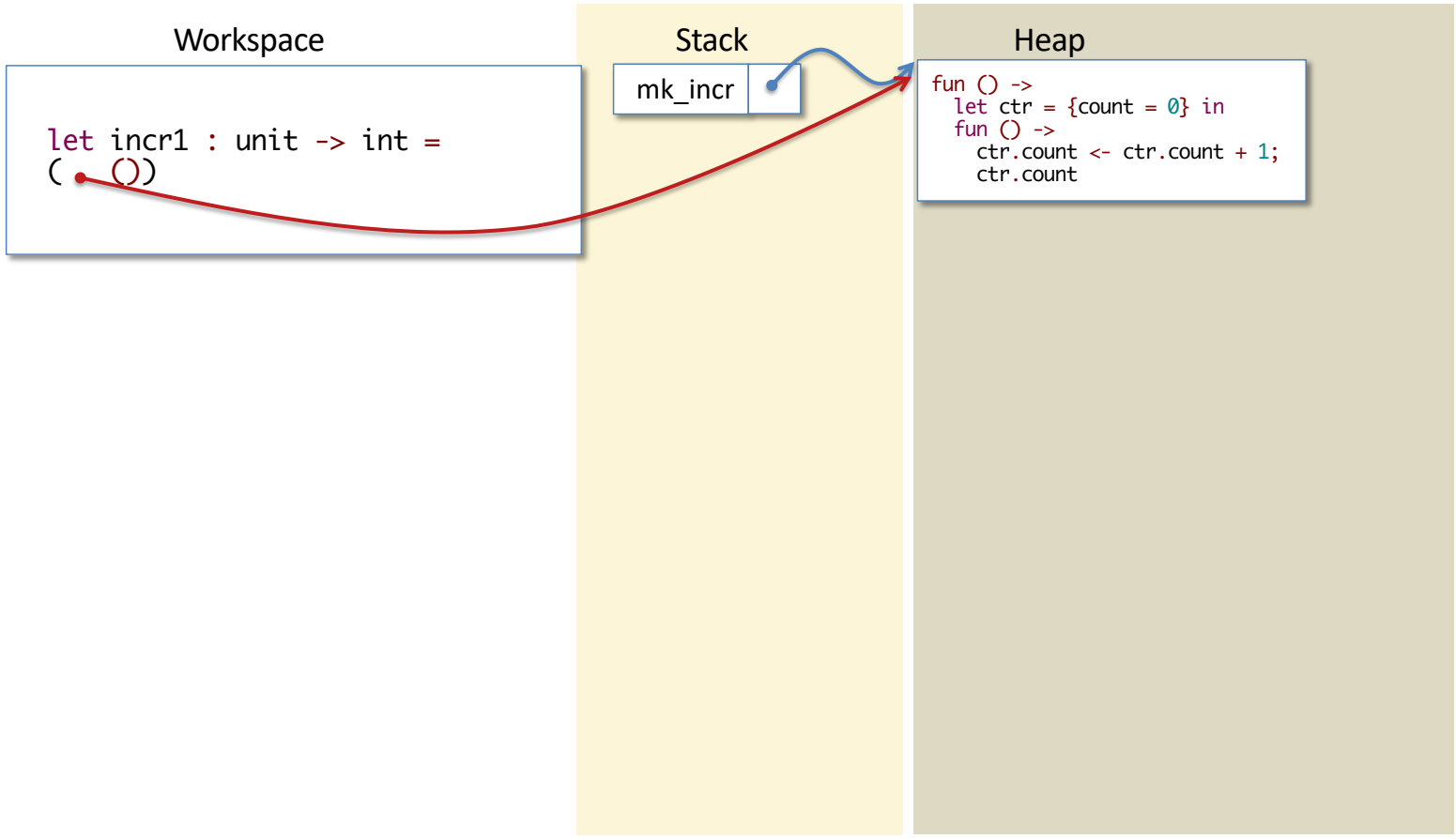
Running mk_incr



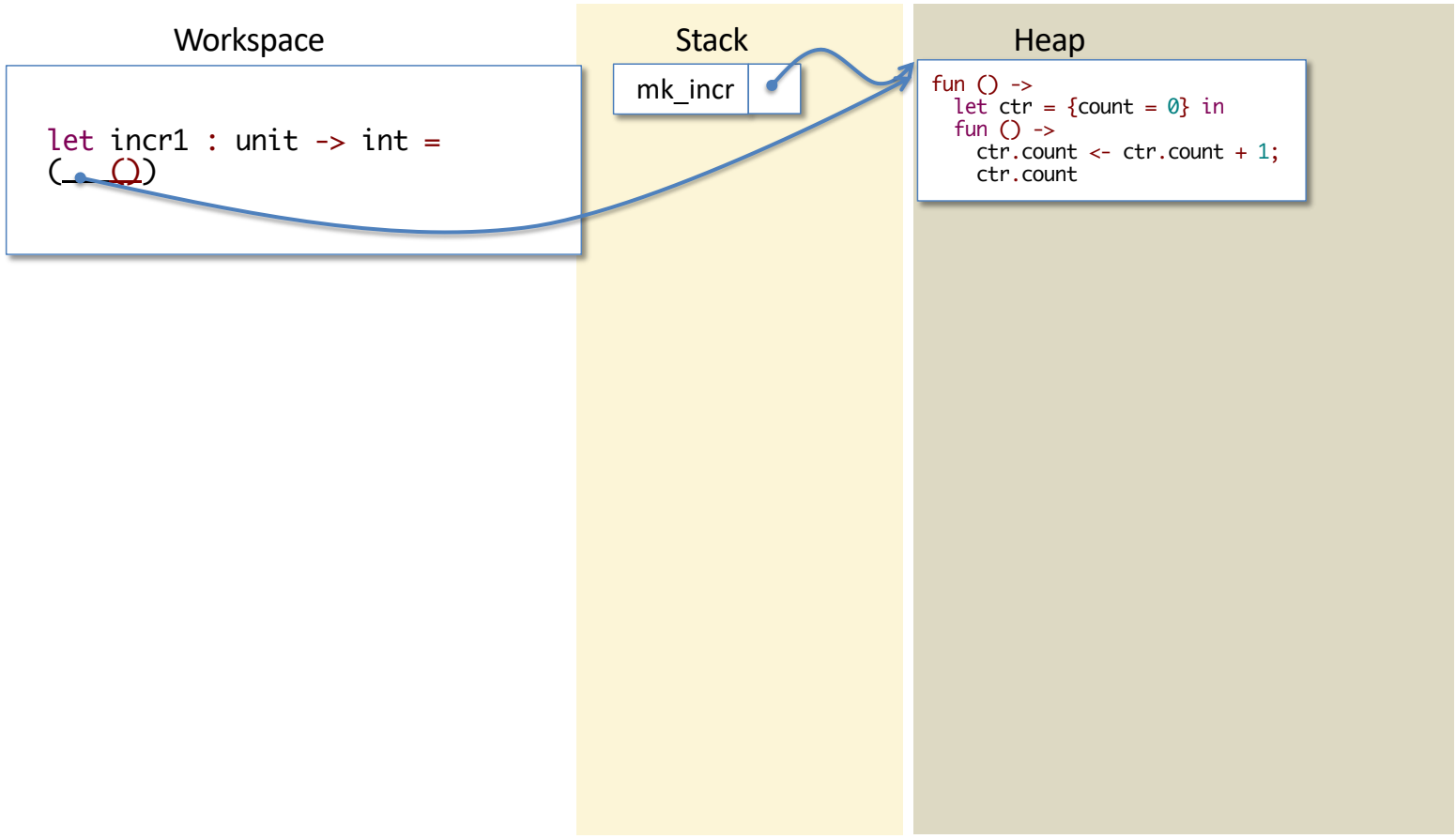
Running mk_incr



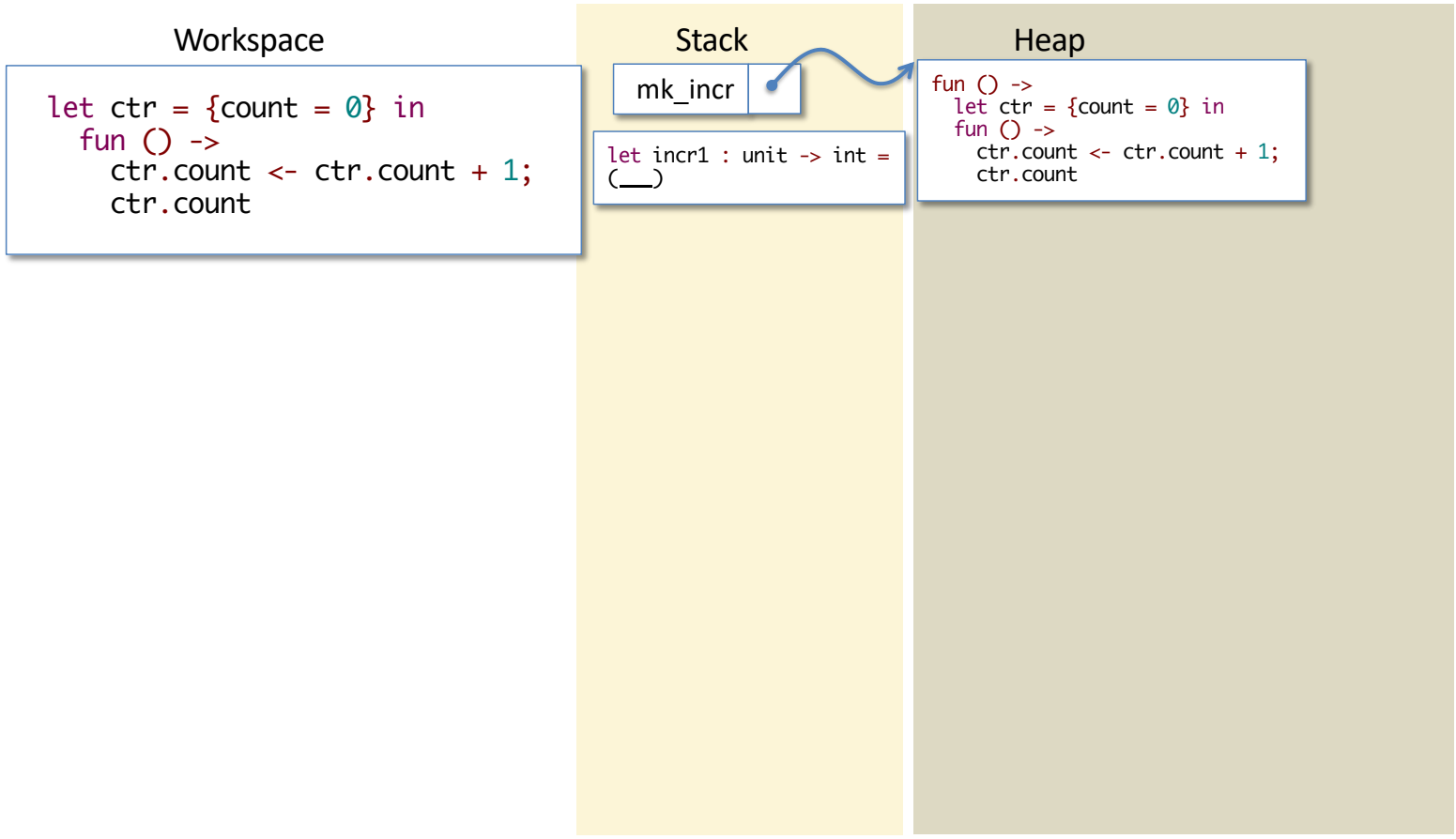
Running mk_incr



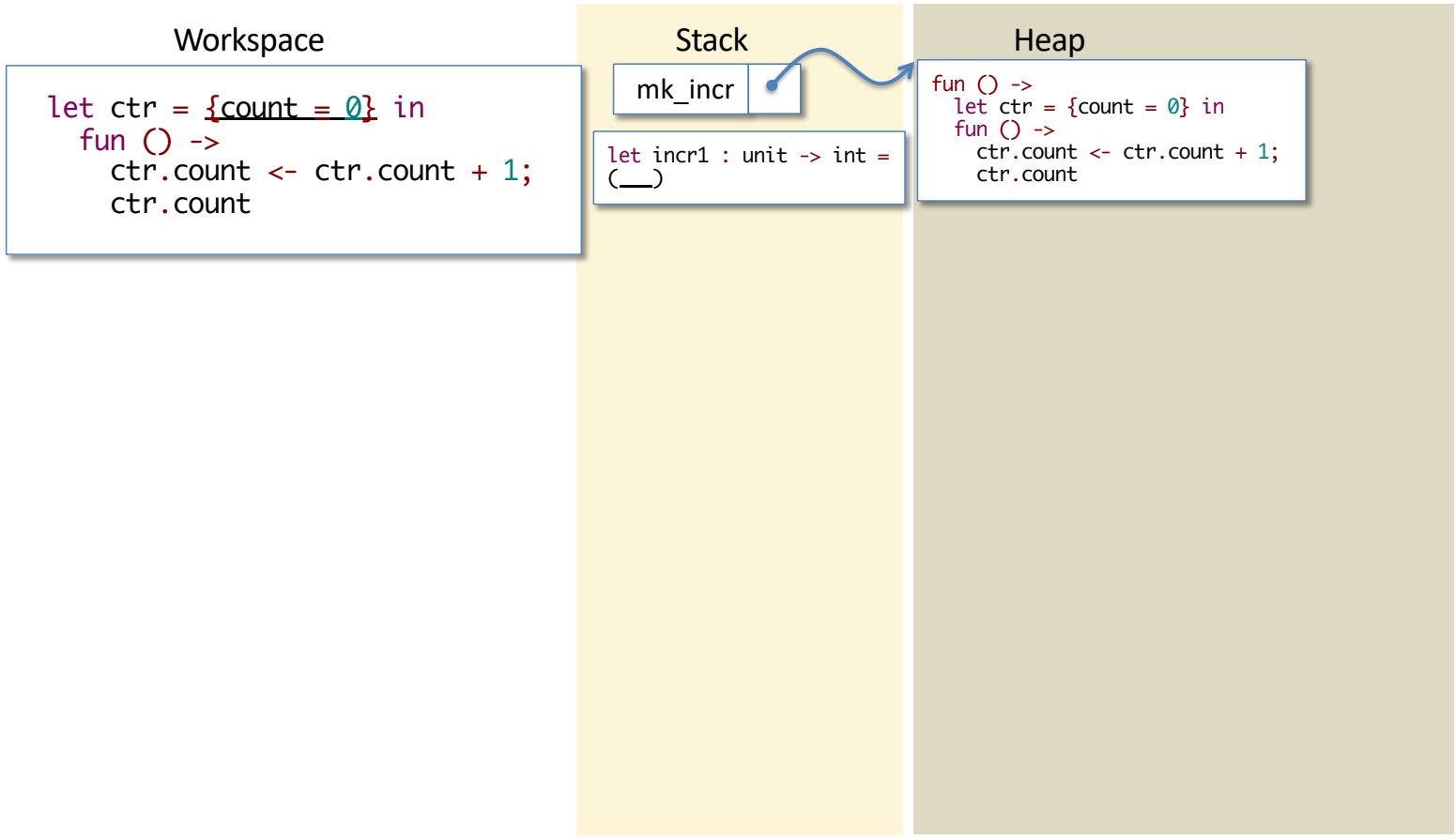
Running mk_incr



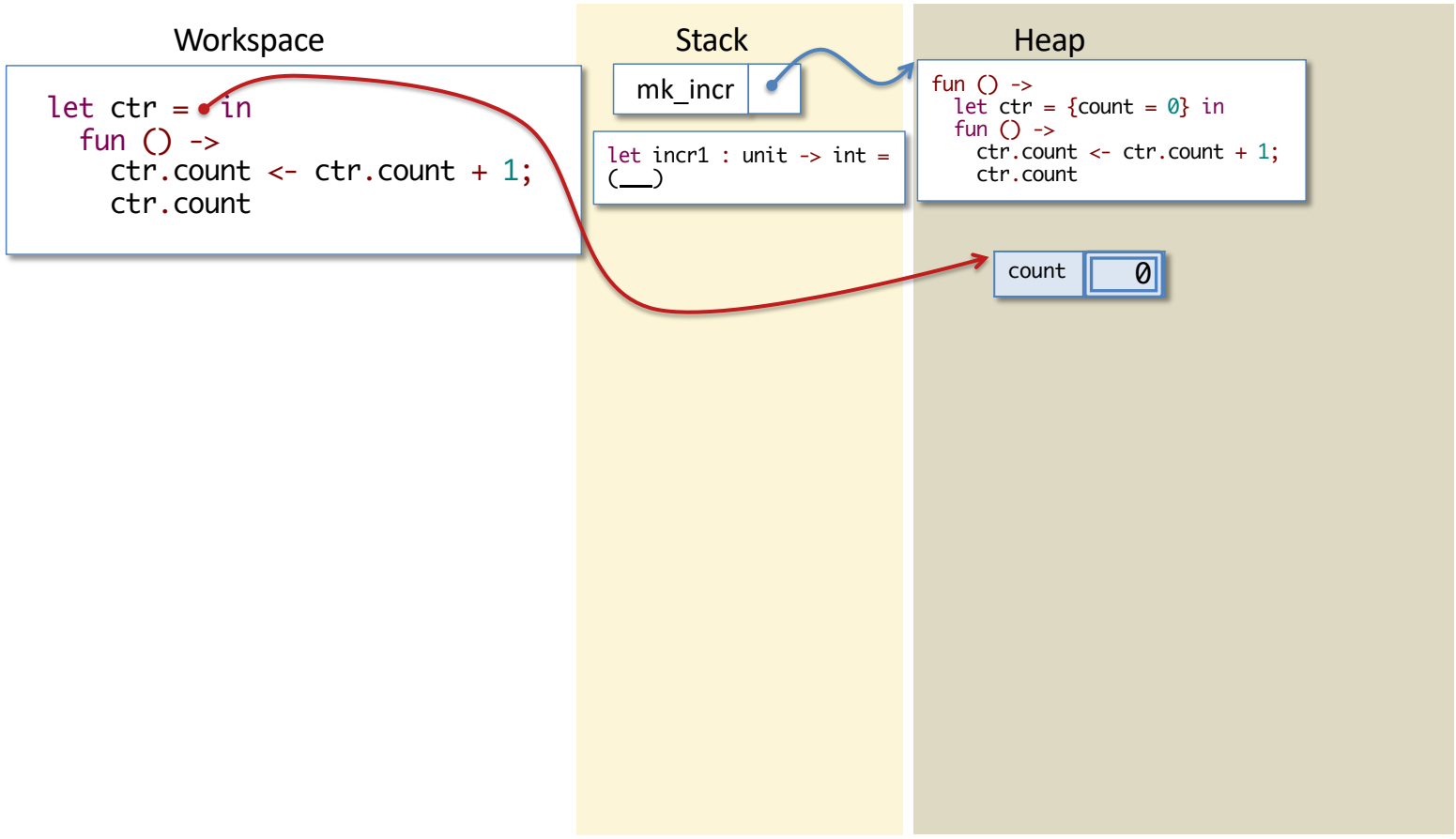
Running mk_incr



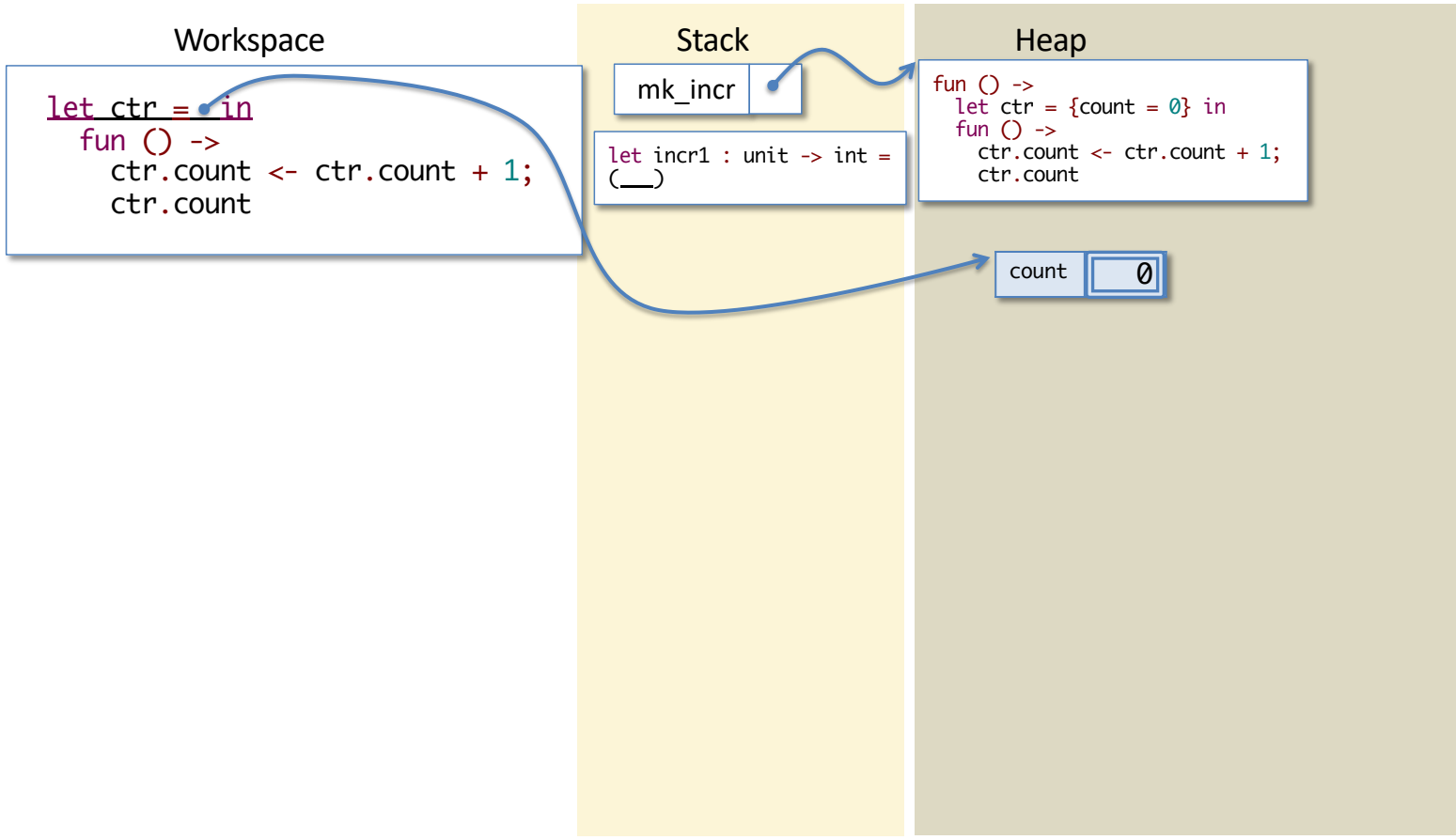
Running mk_incr



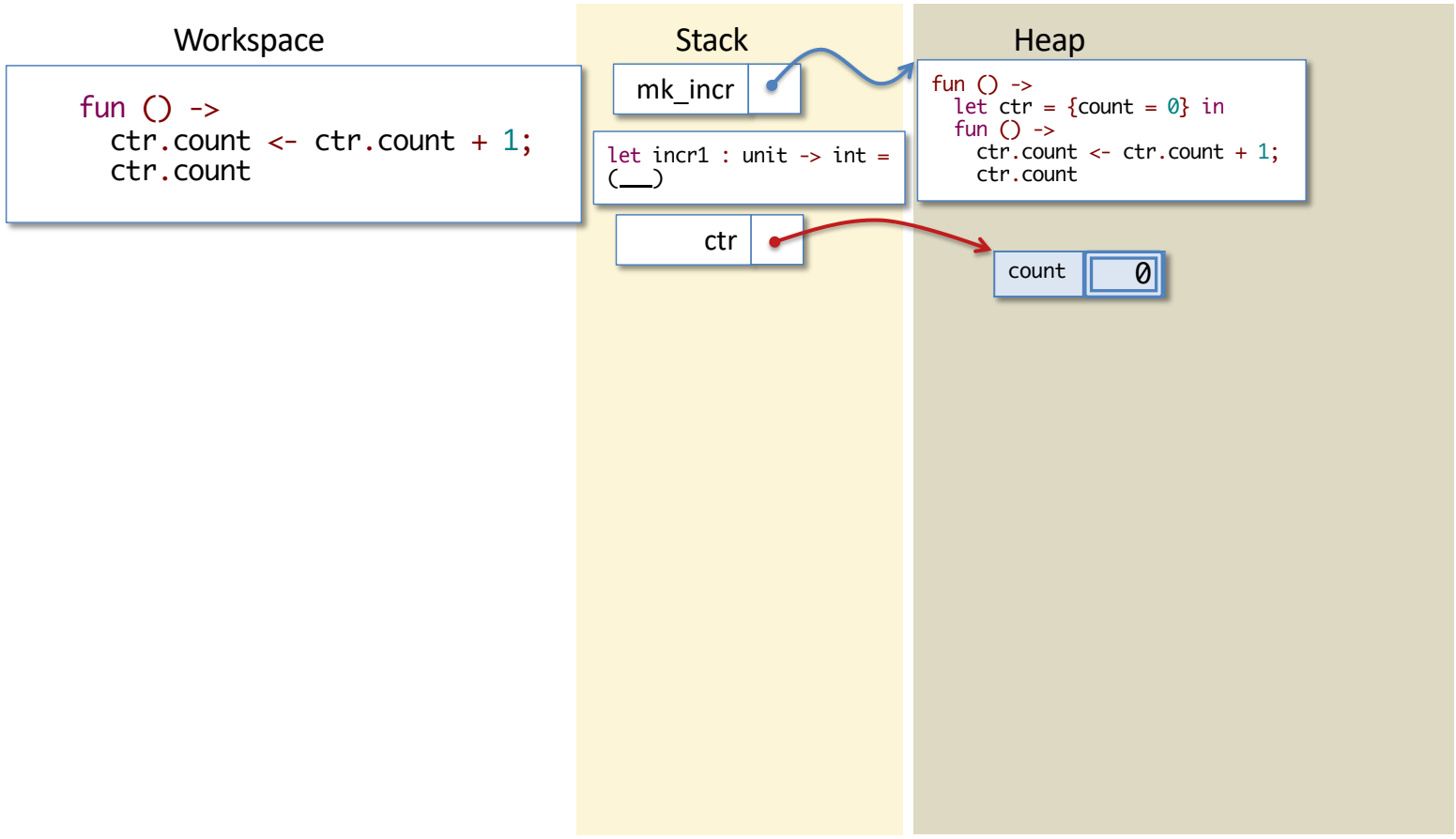
Running mk_incr



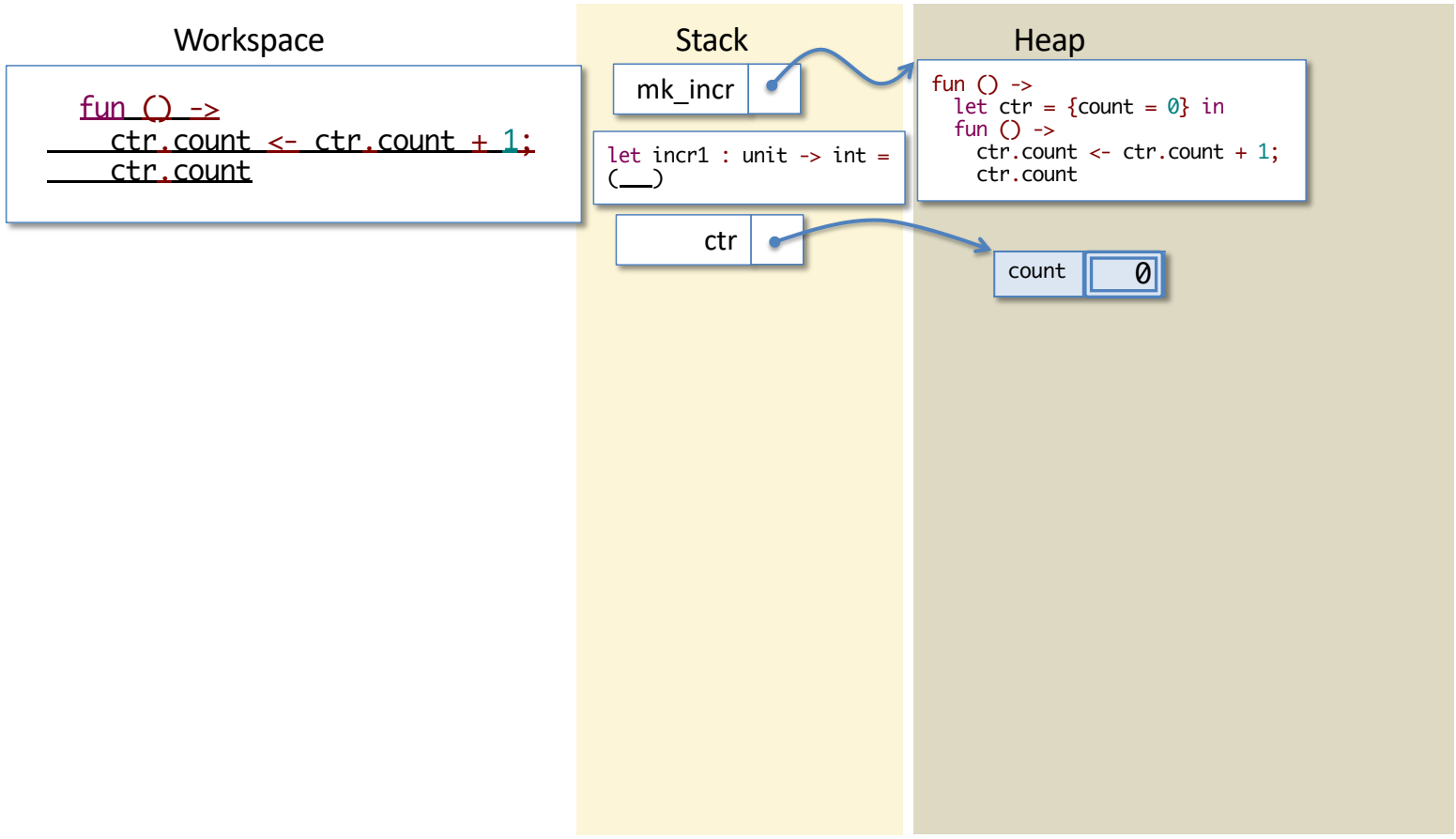
Running mk_incr



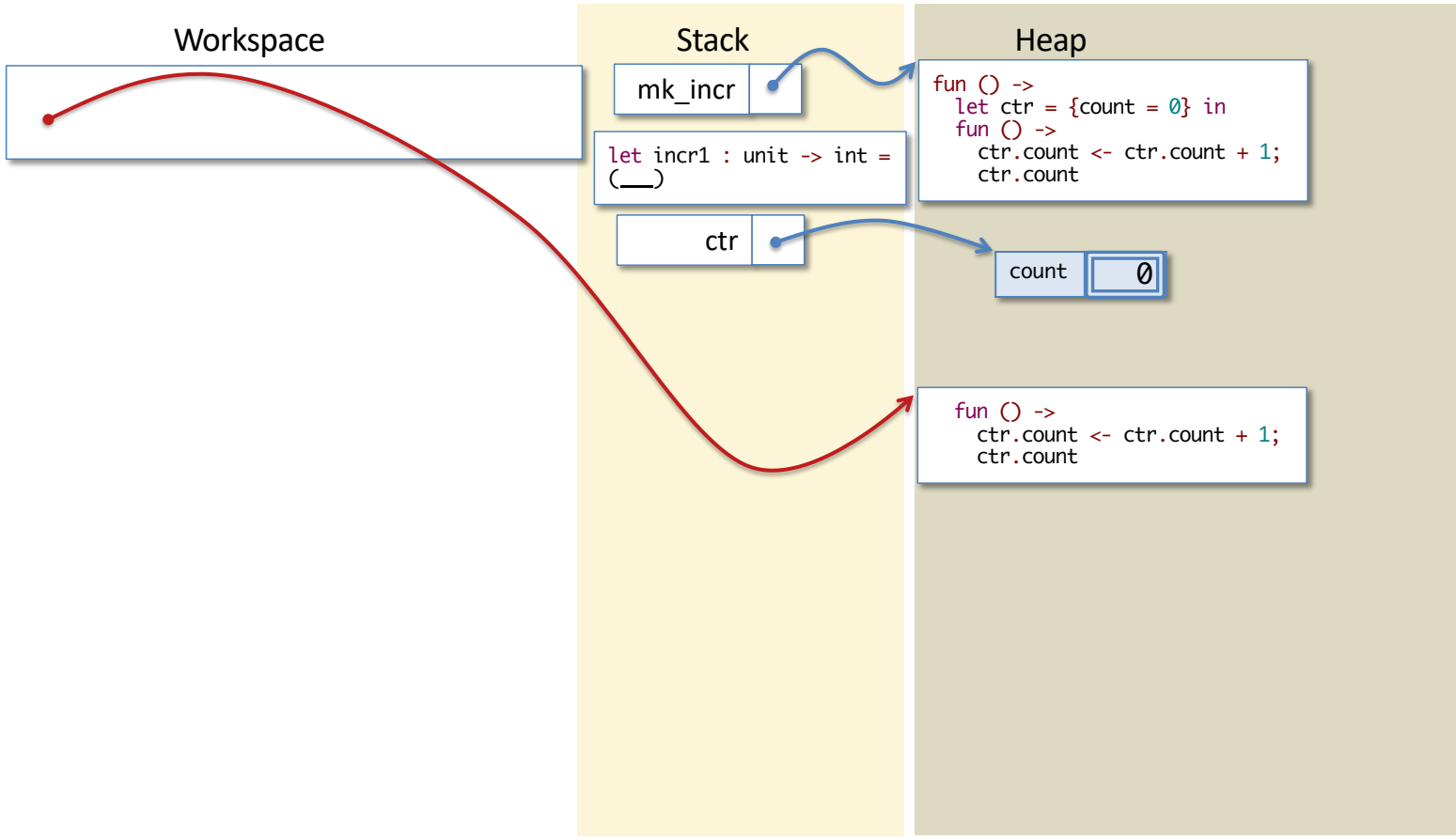
Running mk_incr



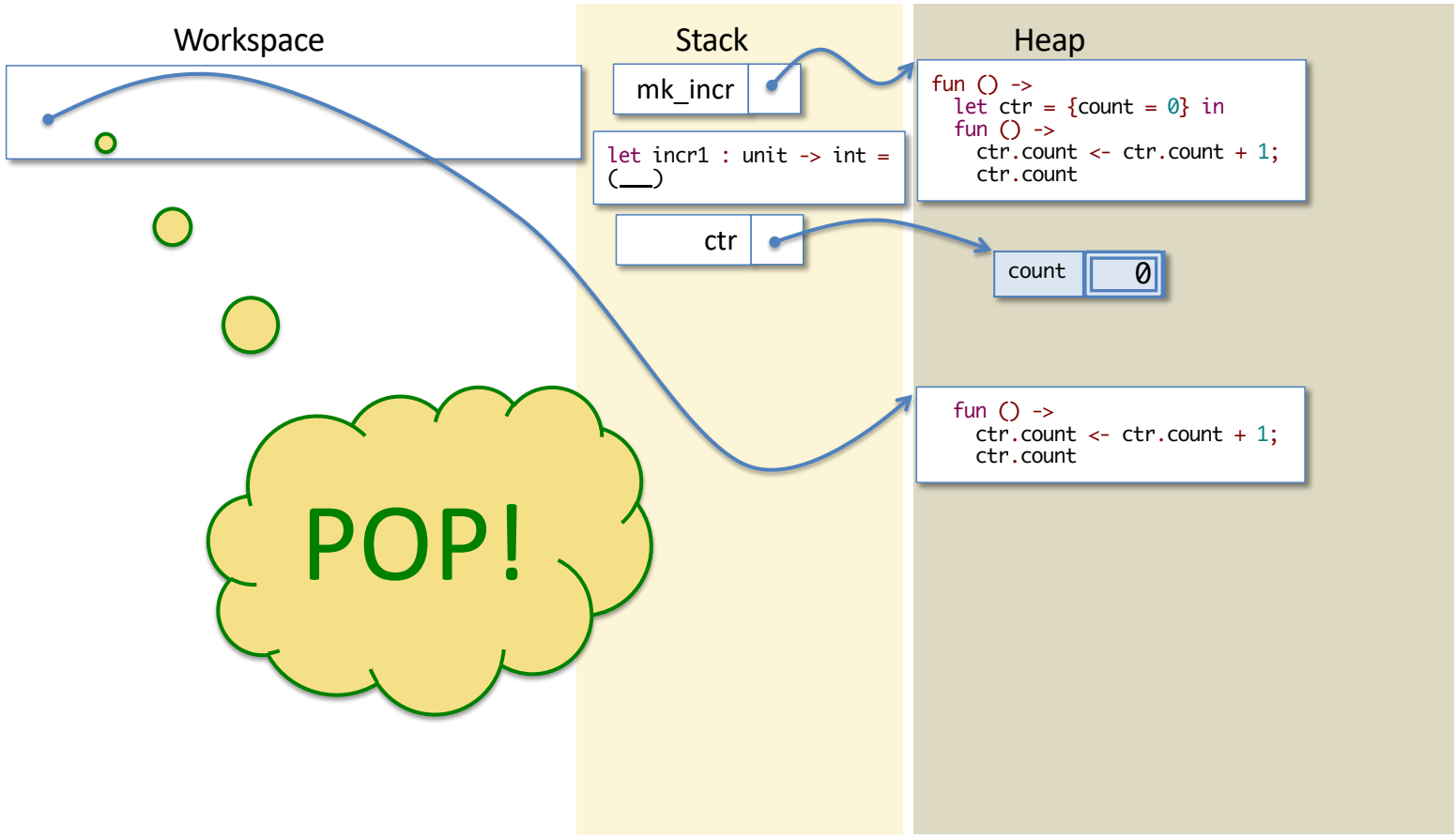
Running mk_incr



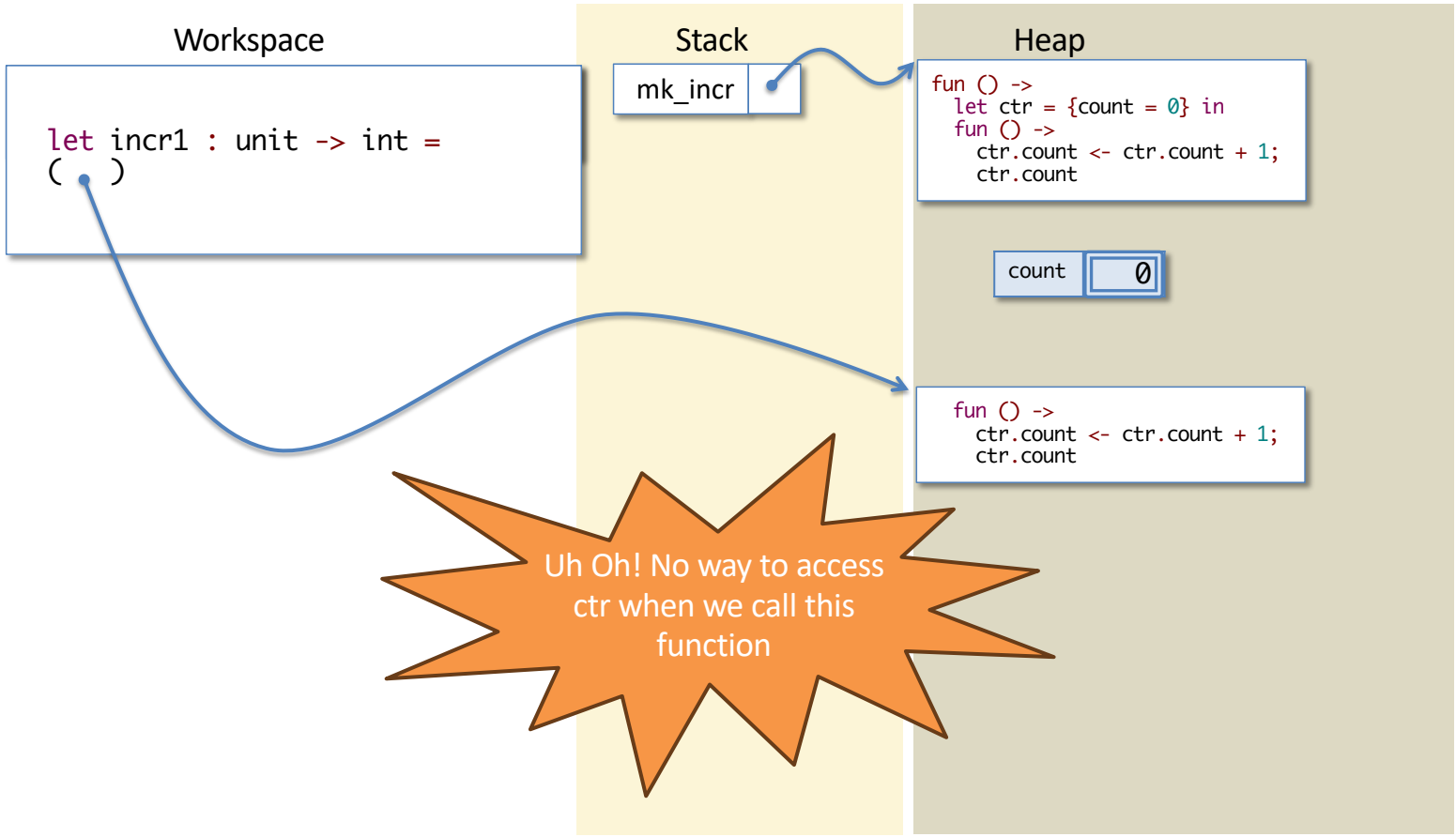
Local Functions (wrong)



Local Functions (wrong)

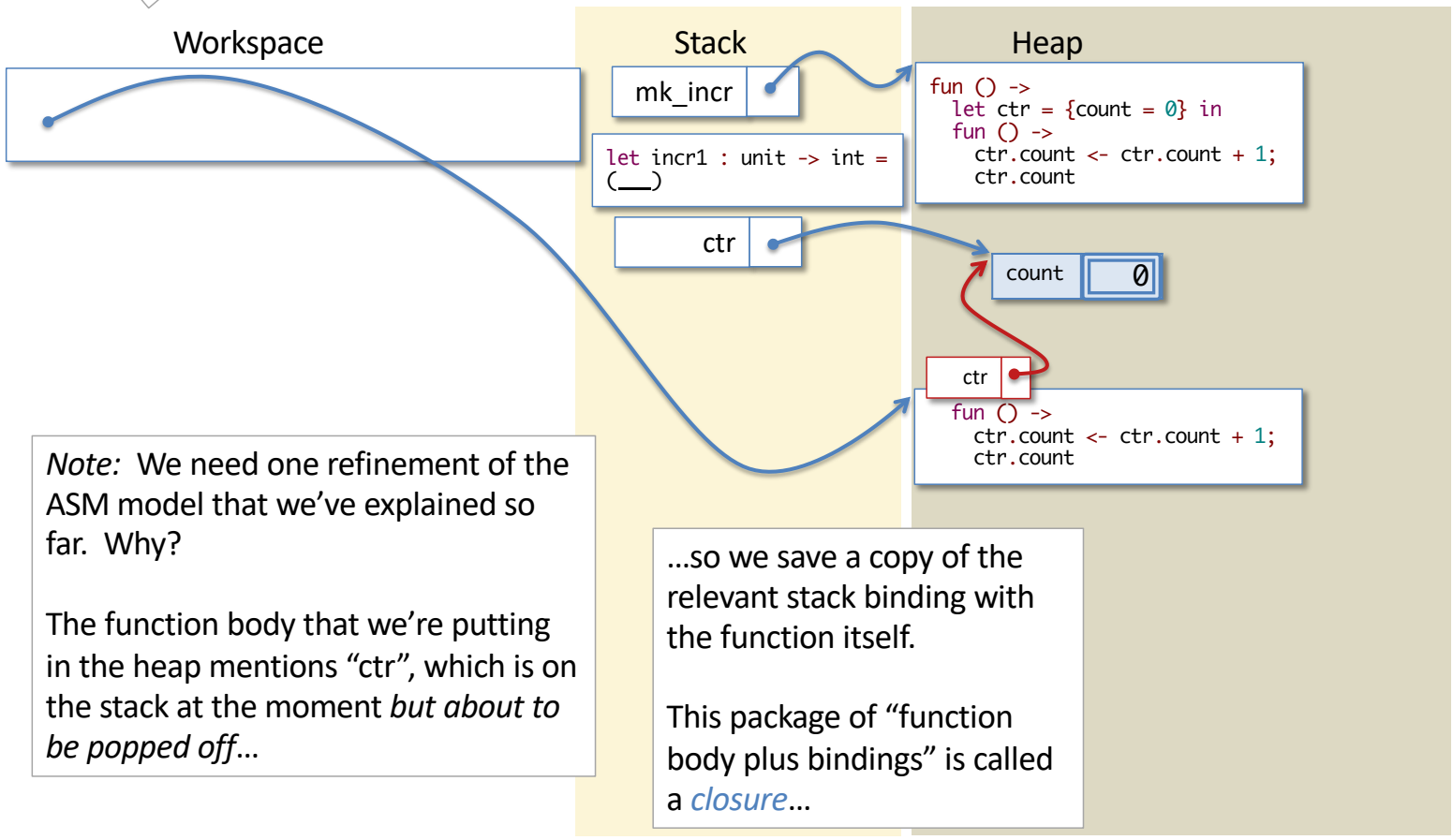


Local Functions (wrong)



Local Functions (right)

Key step!



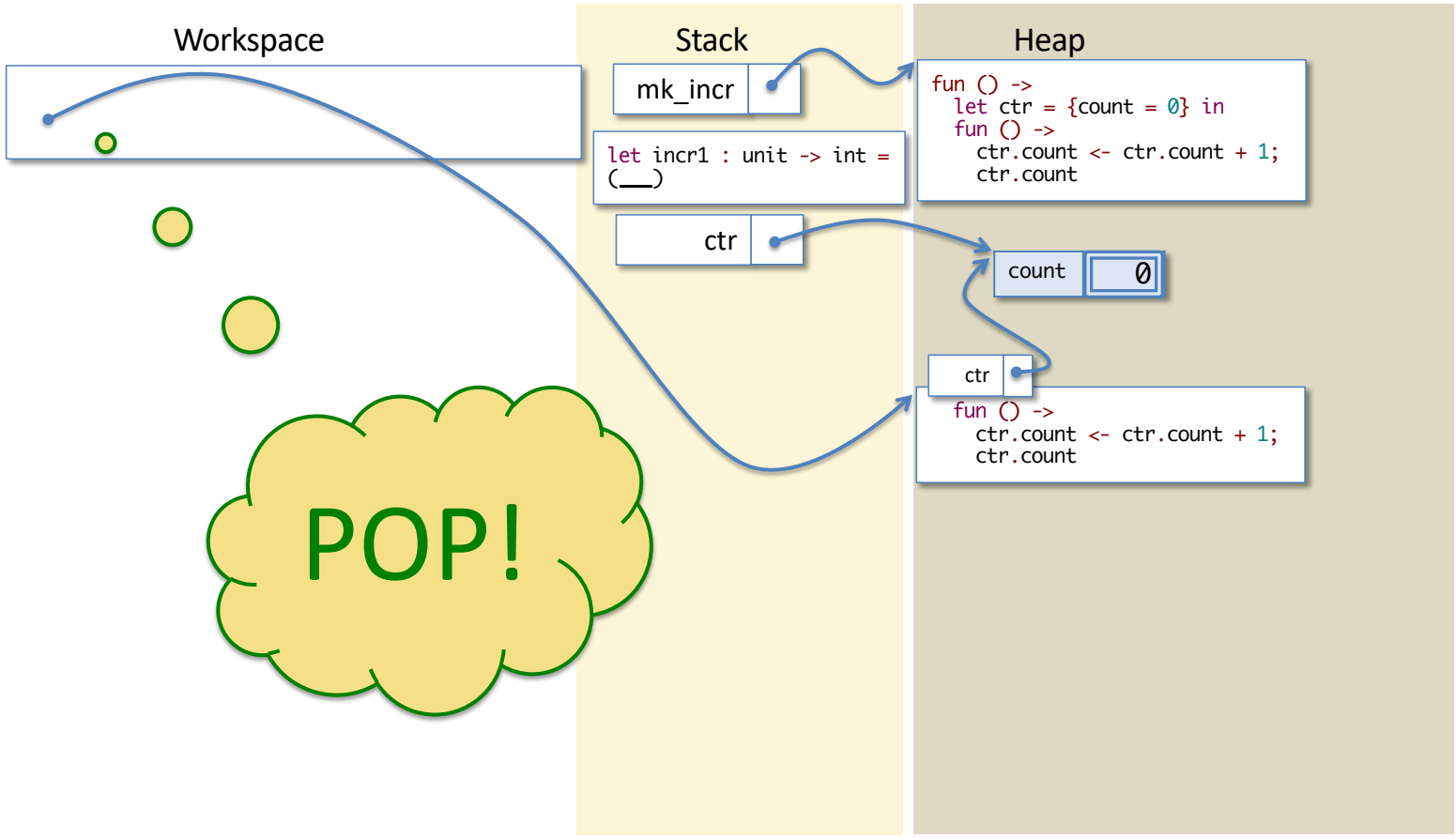
Note: We need one refinement of the ASM model that we've explained so far. Why?

The function body that we're putting in the heap mentions "ctr", which is on the stack at the moment *but about to be popped off*…

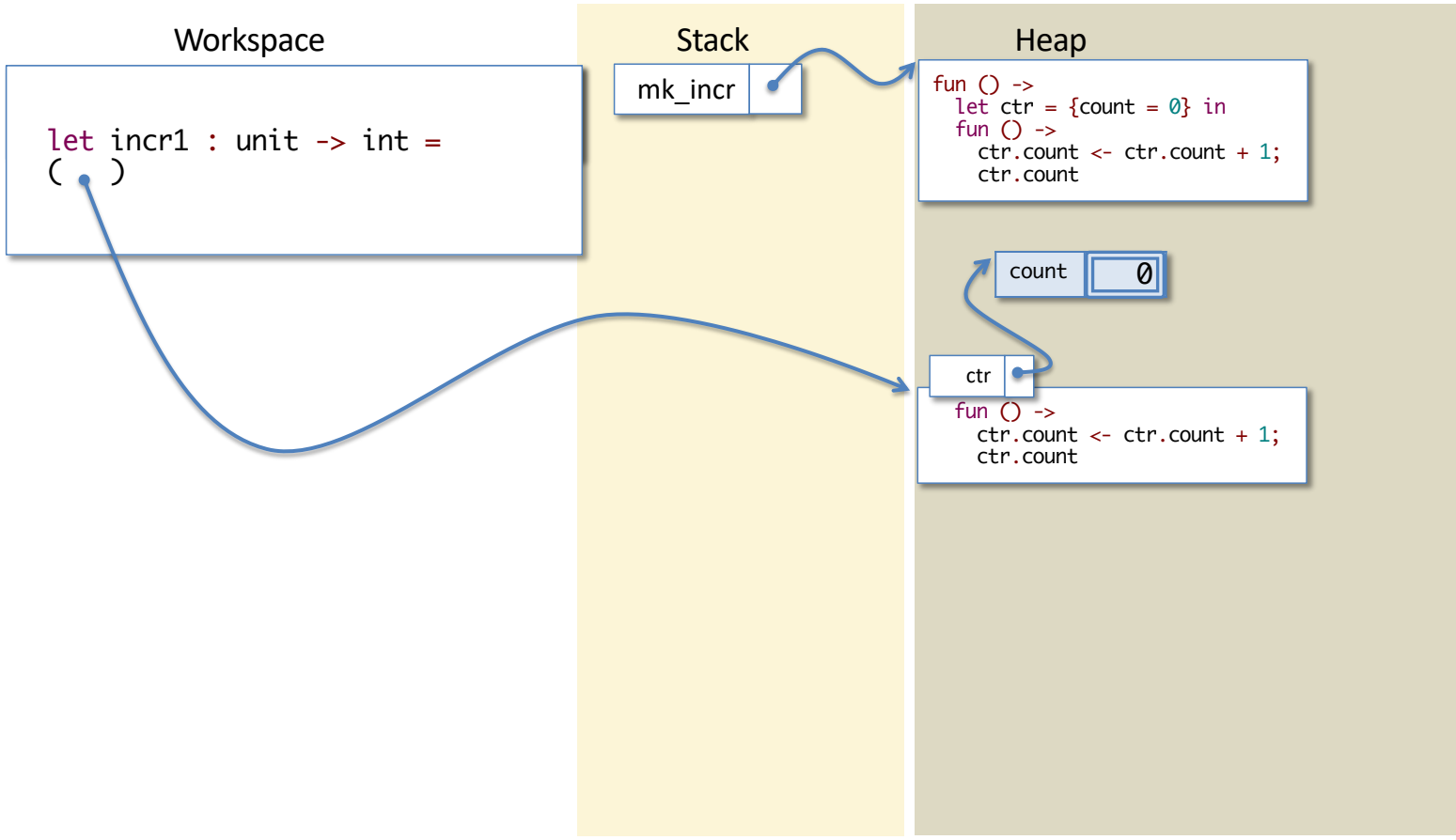
…so we save a copy of the relevant stack binding with the function itself.

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

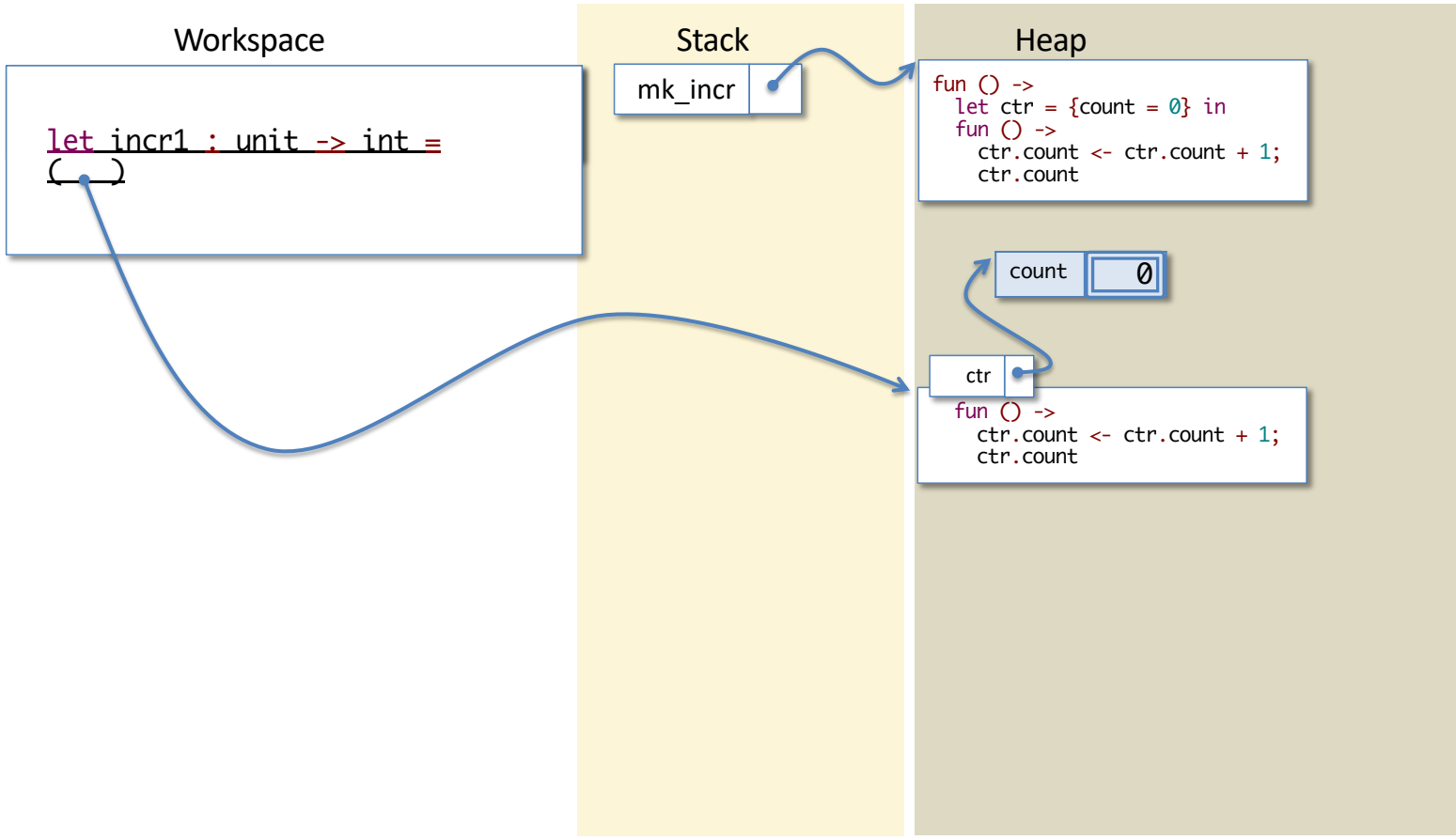
Local Functions



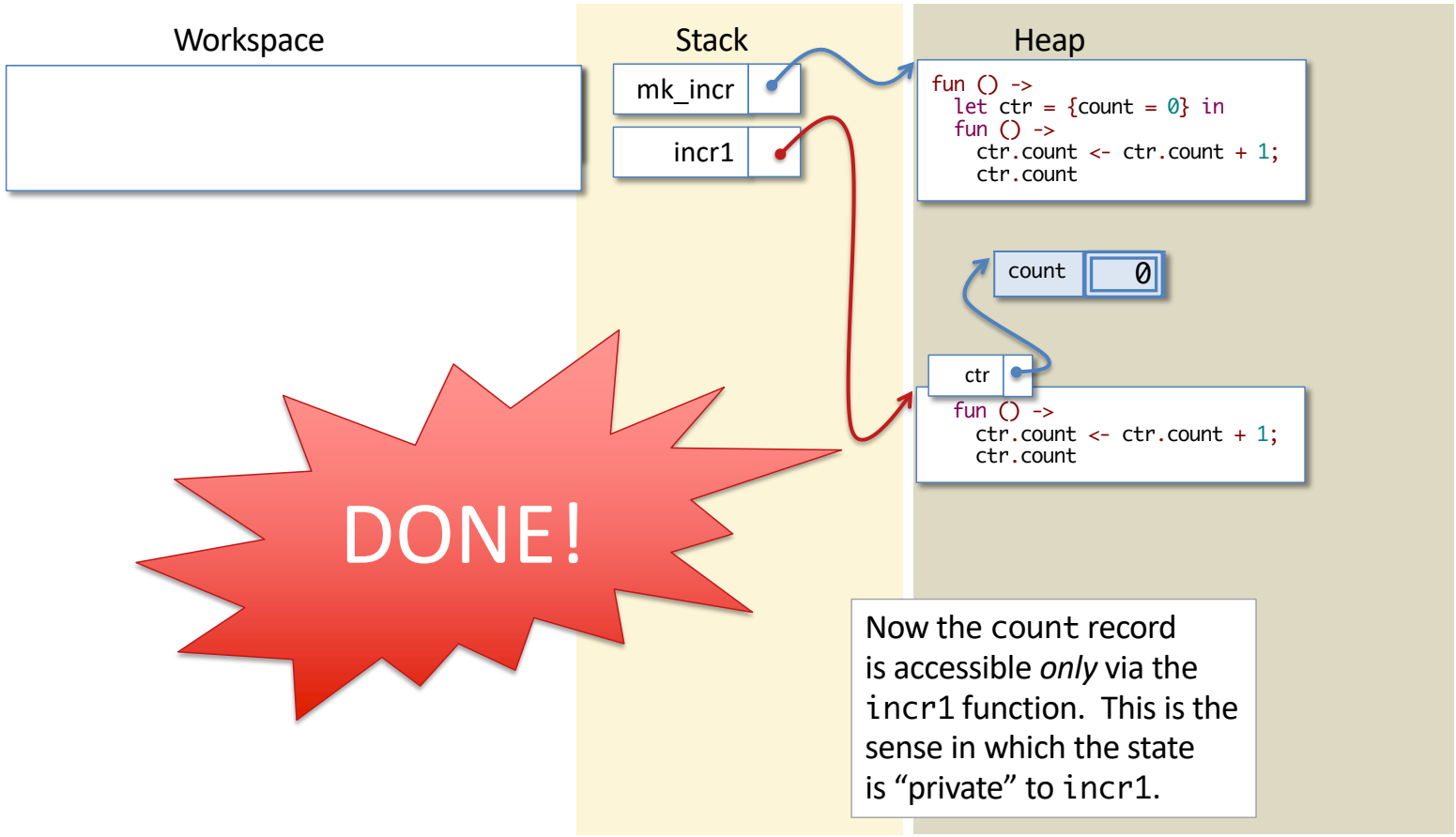
Local Functions



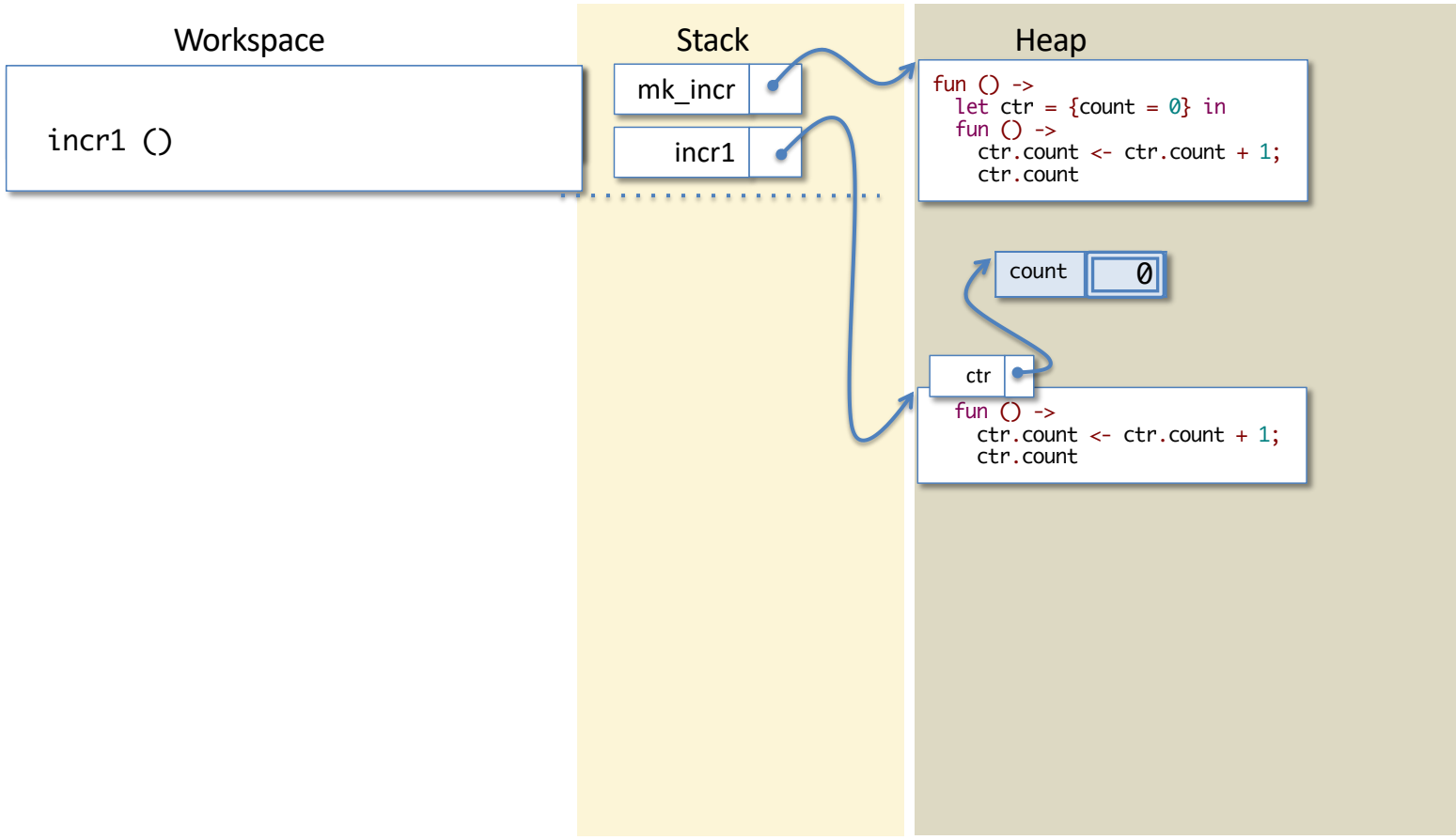
Local Functions



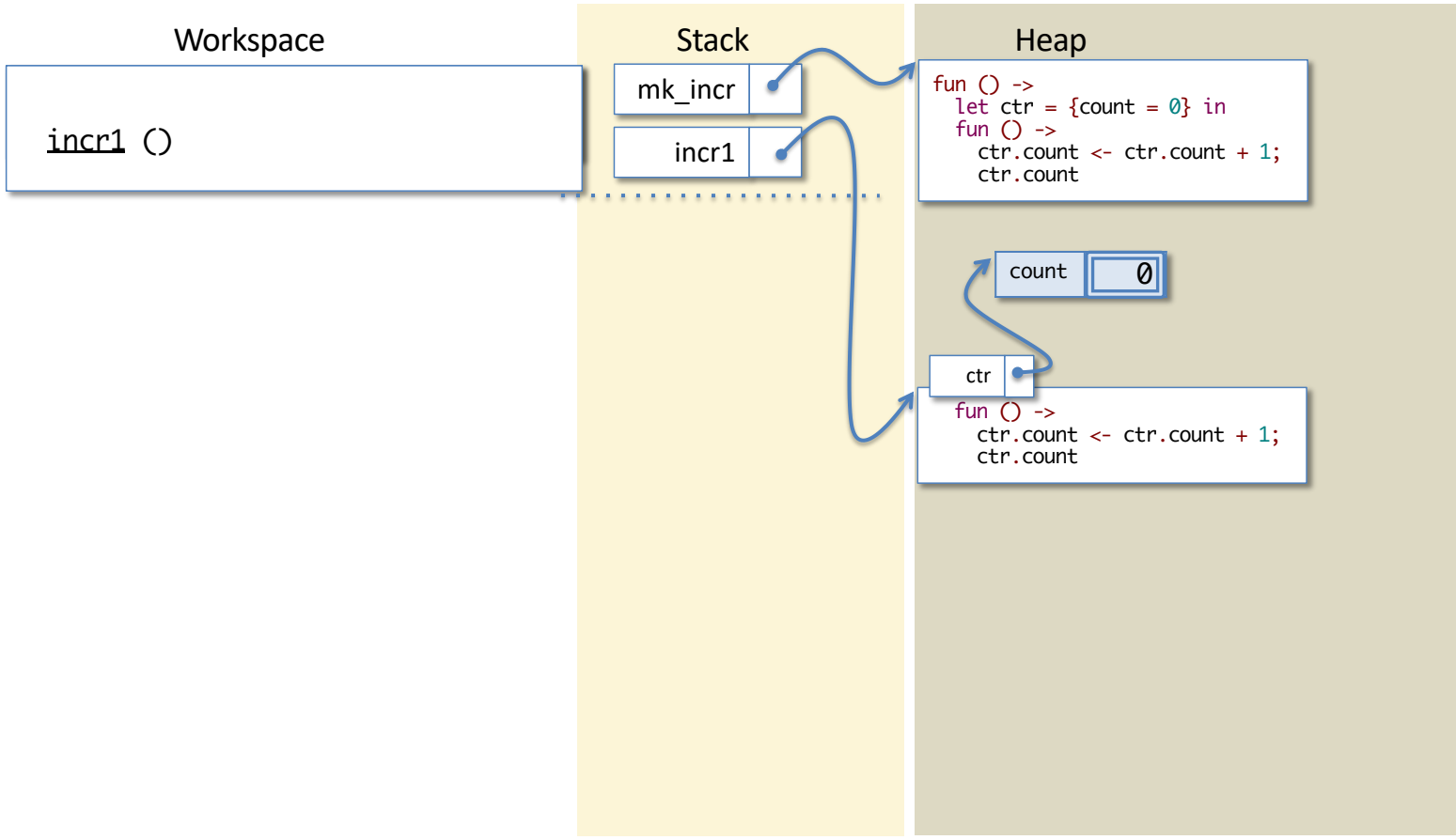
Local Functions



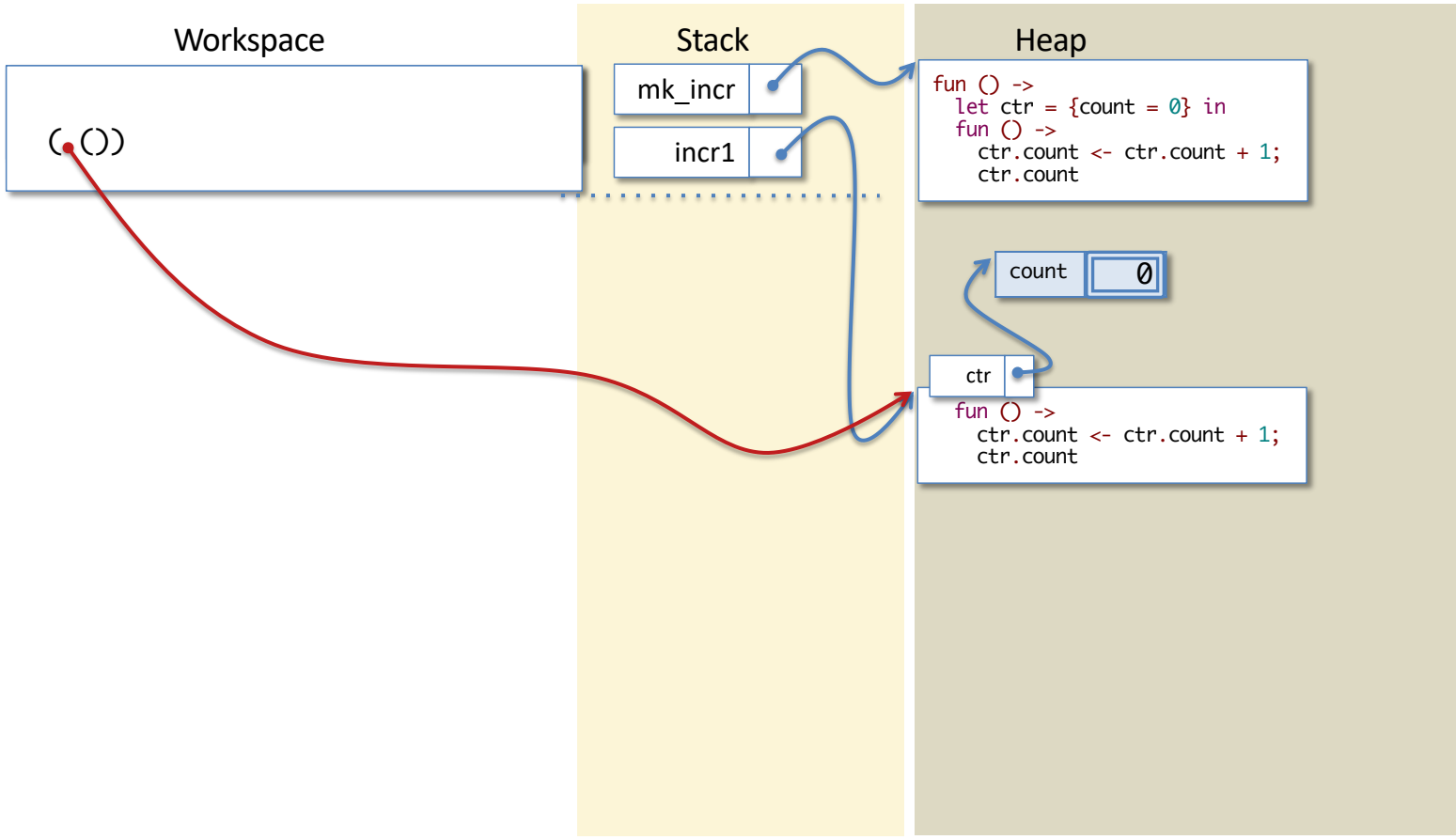
Now let's run "`incr1 ()`"



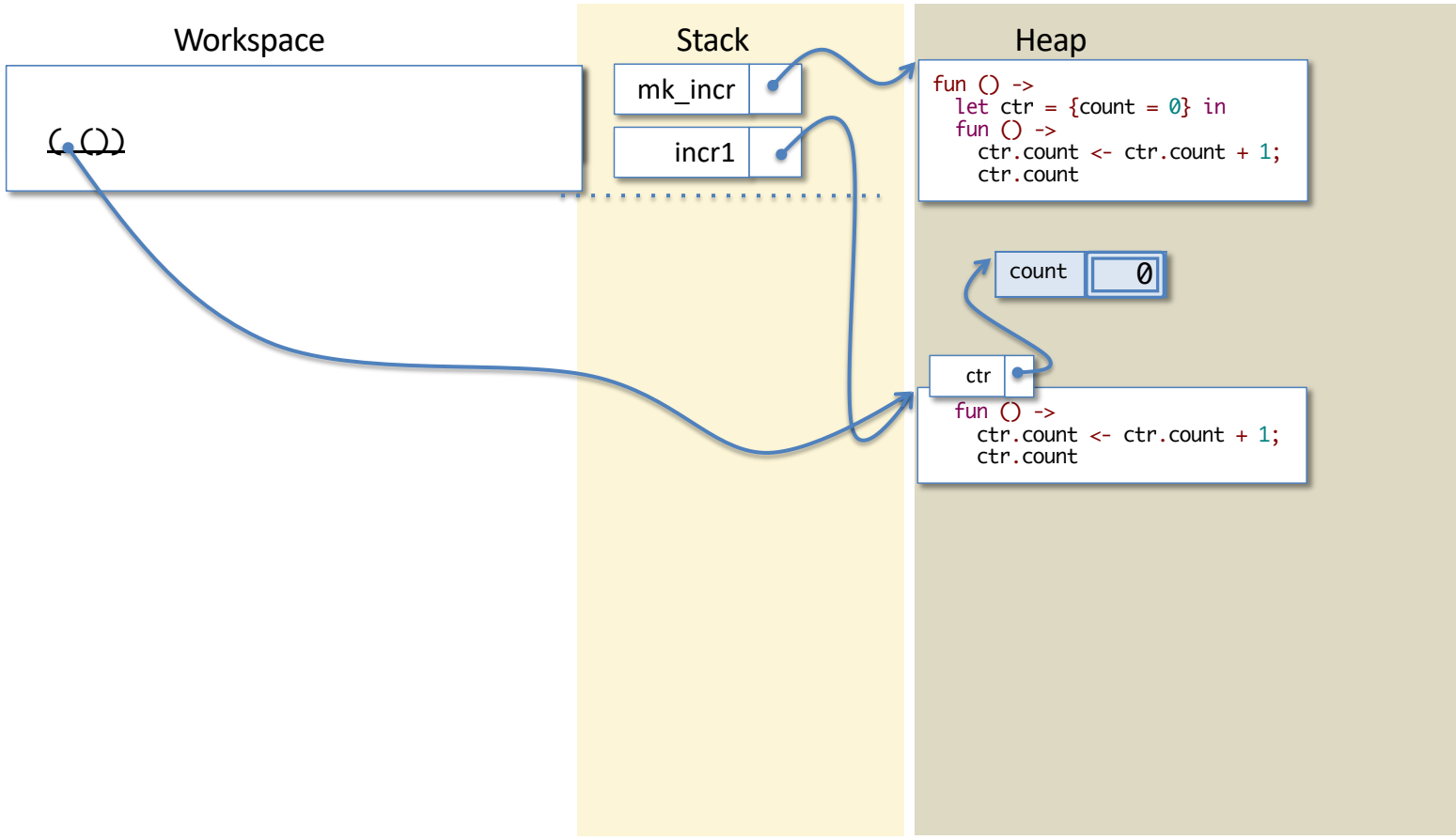
Now let's run "`incr1 ()`"



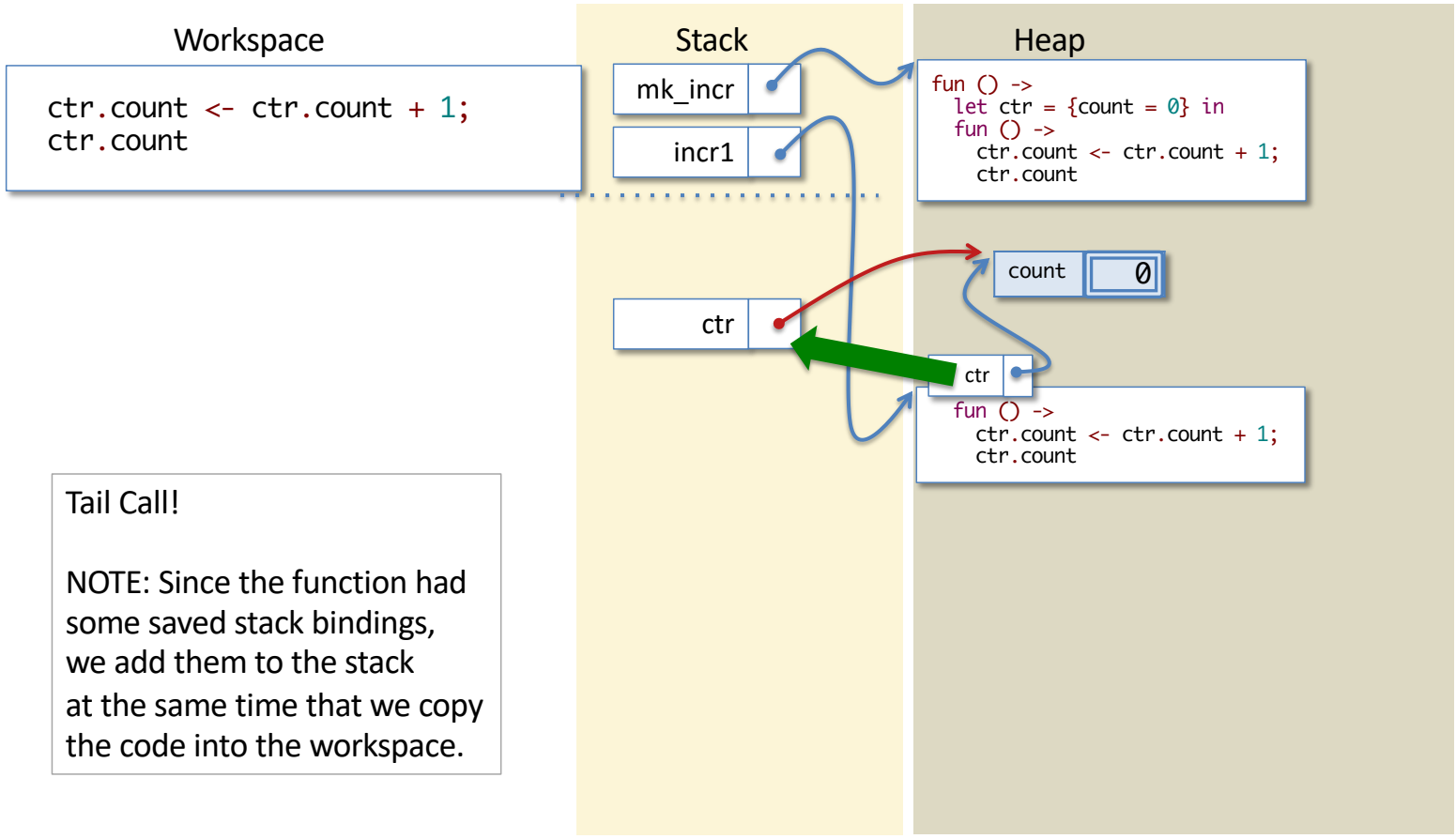
Now let's run "`incr1 ()`"



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



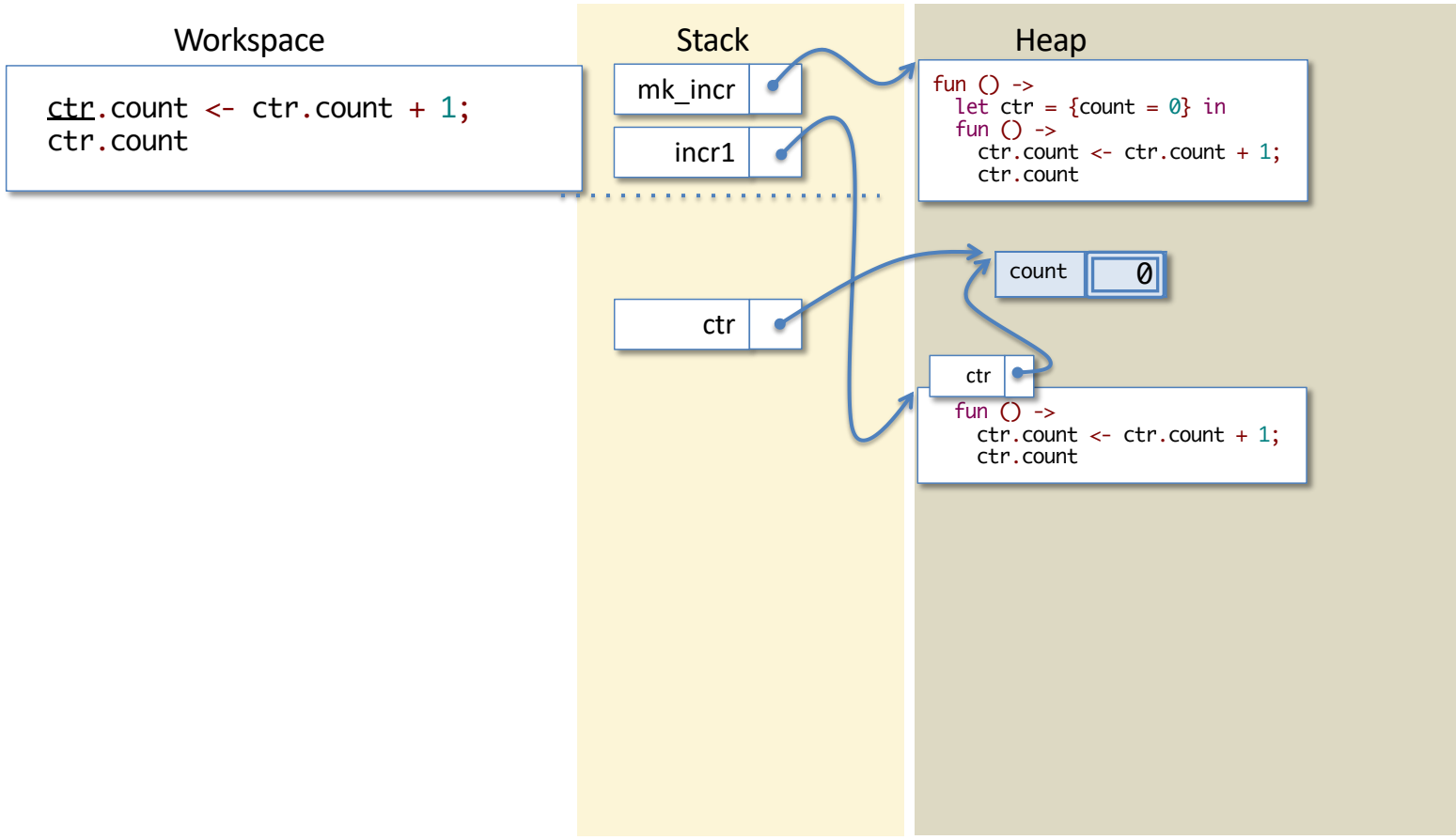
Now let's run "incr1 ()"



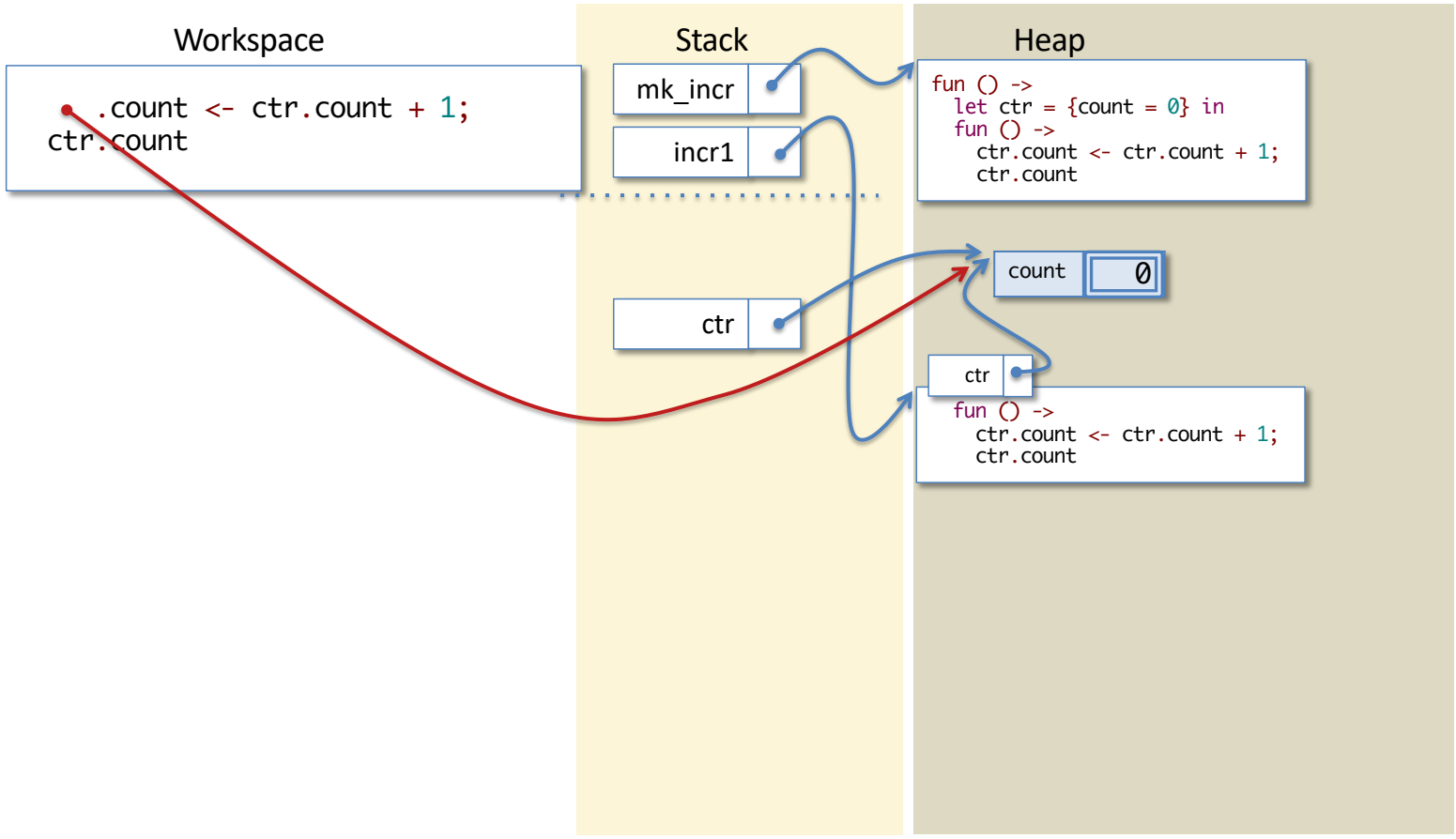
Tail Call!

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

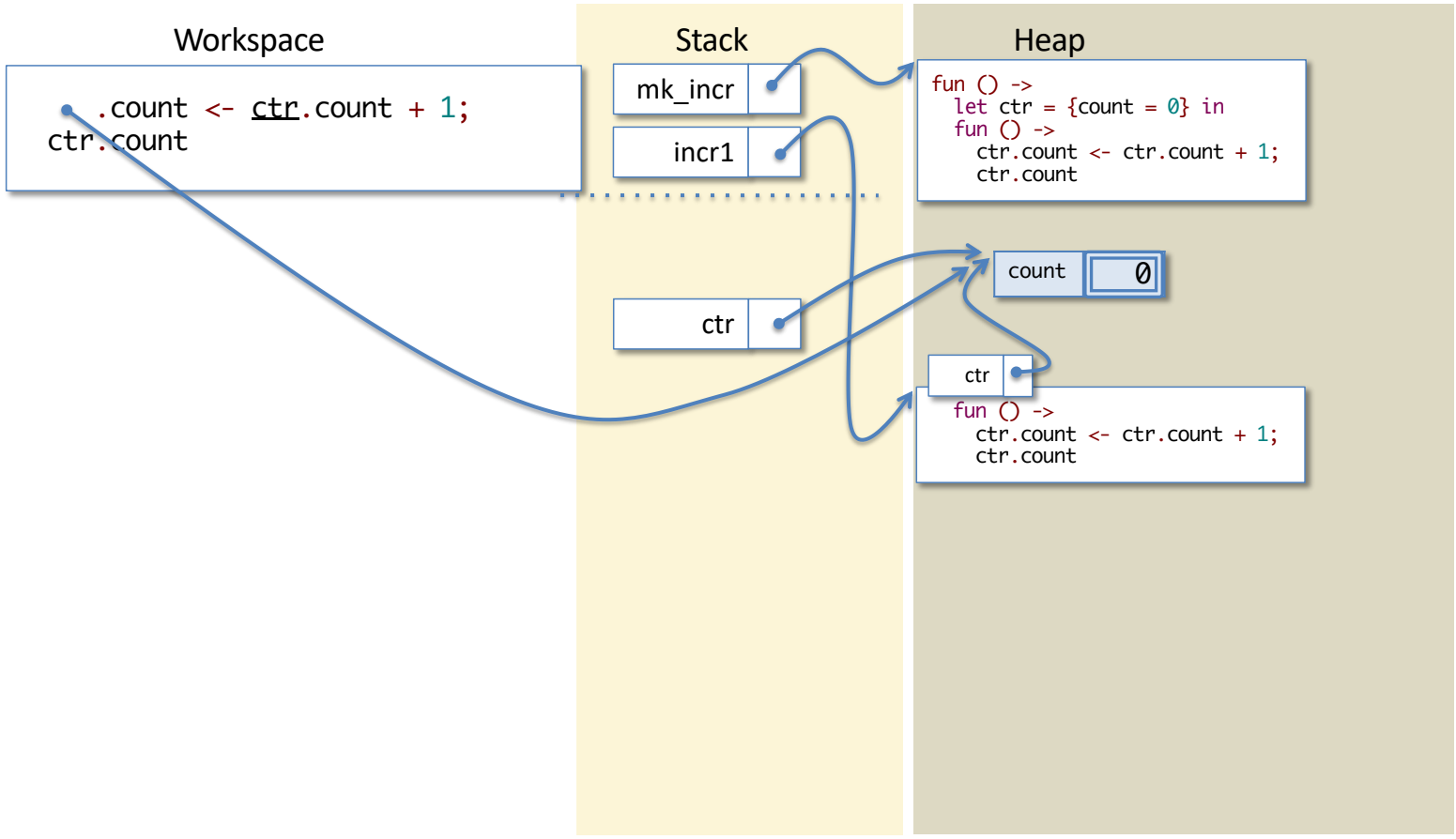
Now let's run "`incr1 ()`"



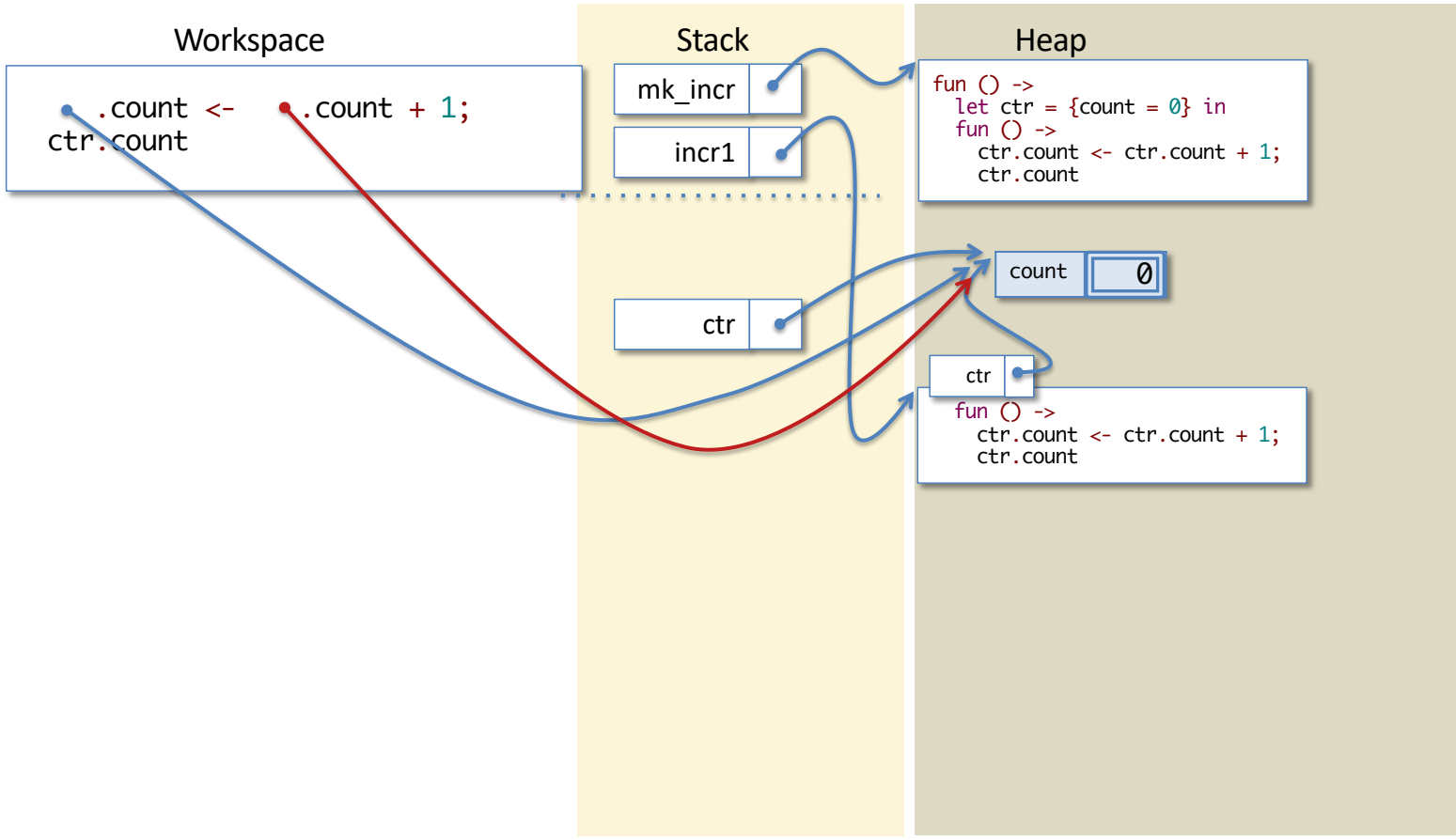
Now let's run "`incr1 ()`"



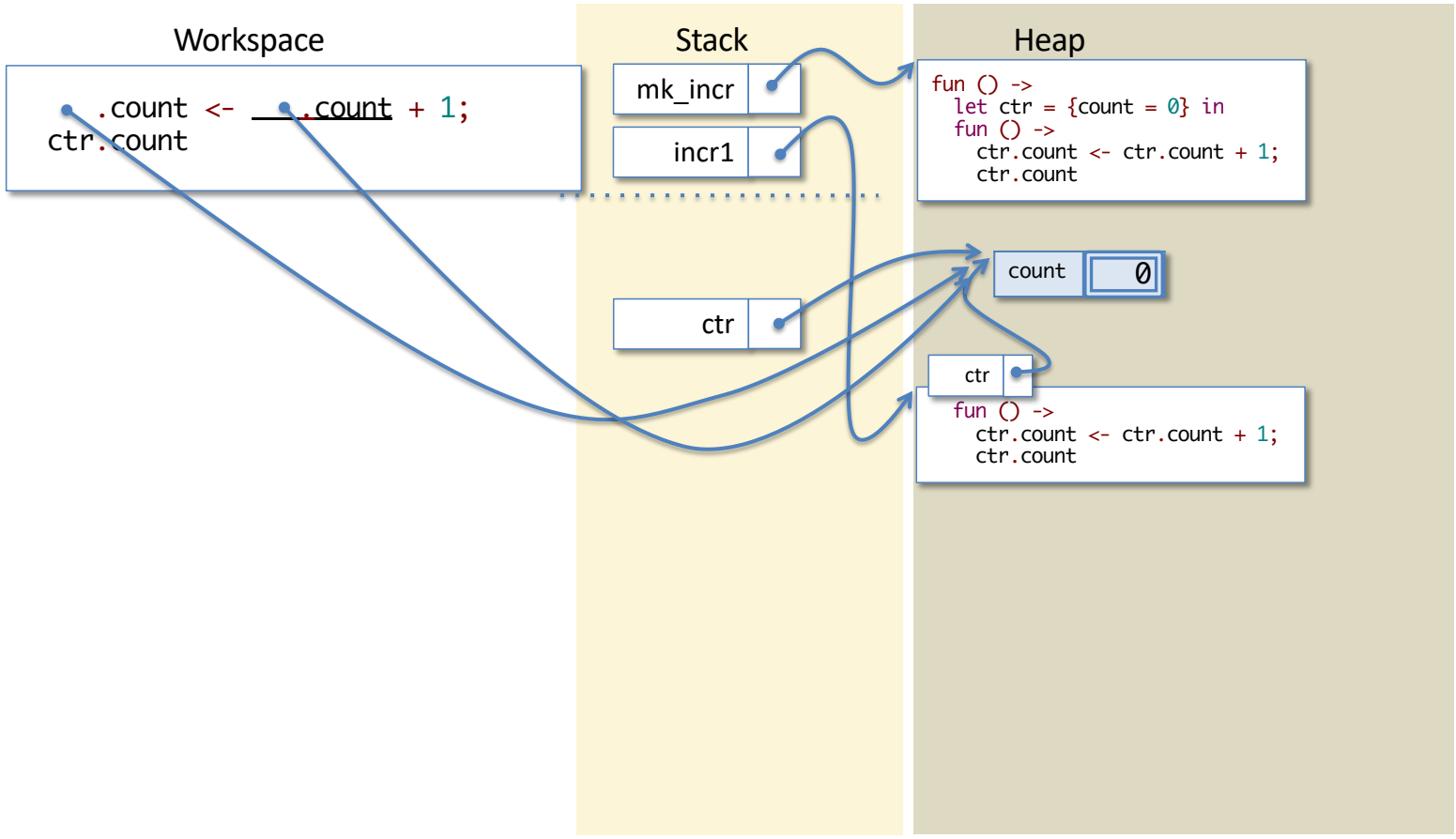
Now let's run "`incr1 ()`"



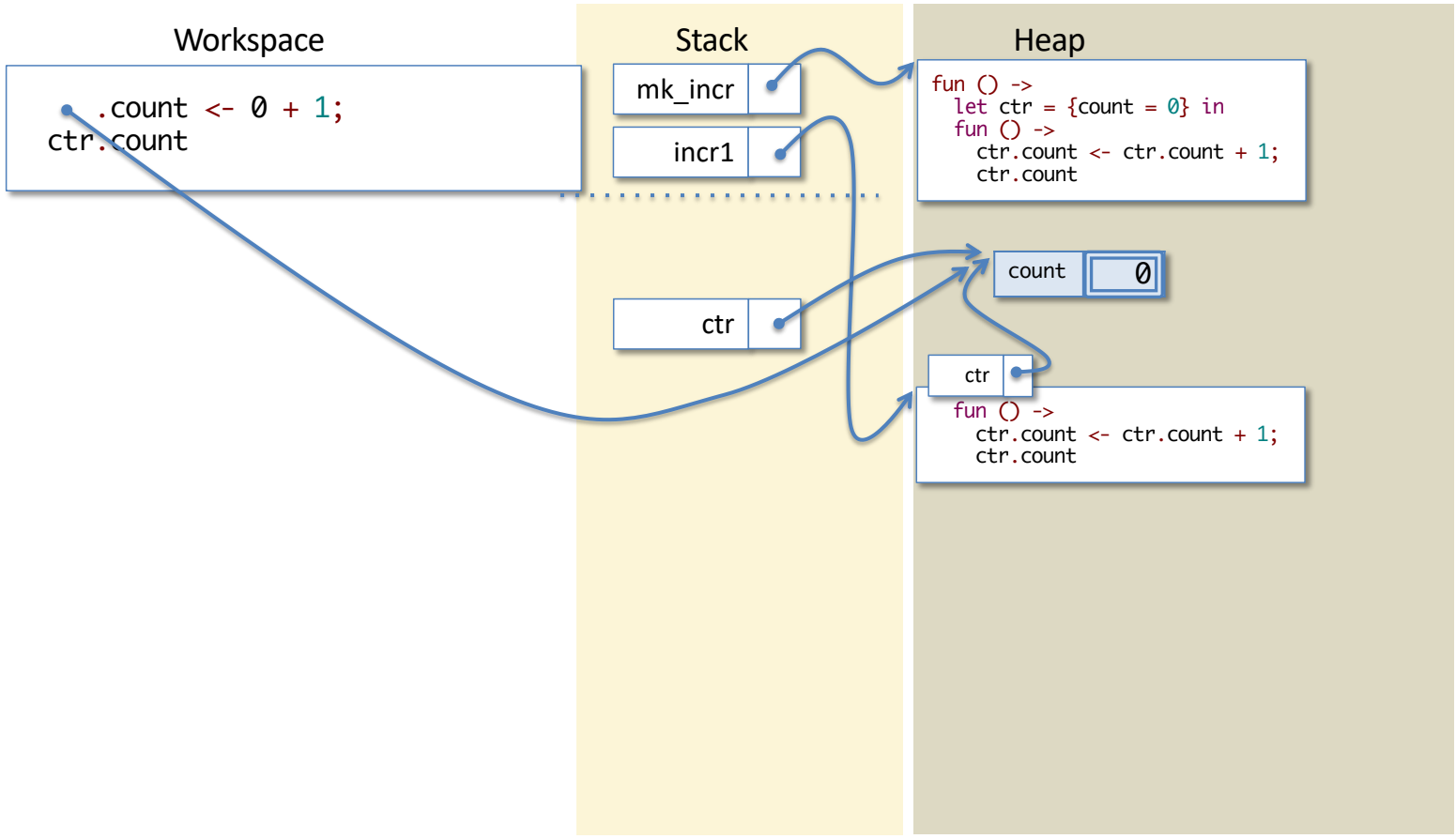
Now let's run "`incr1 ()`"



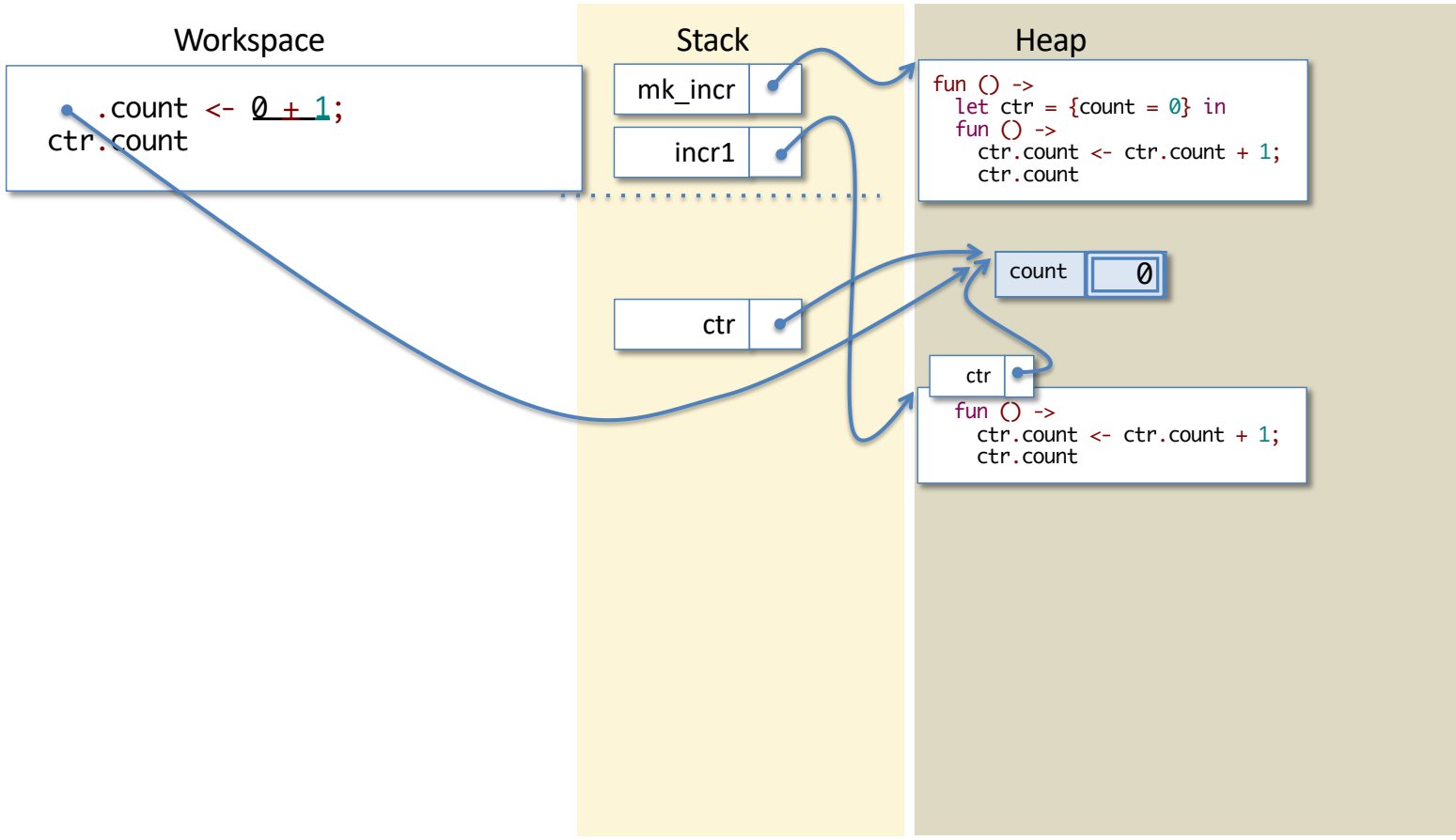
Now let's run "`incr1 ()`"



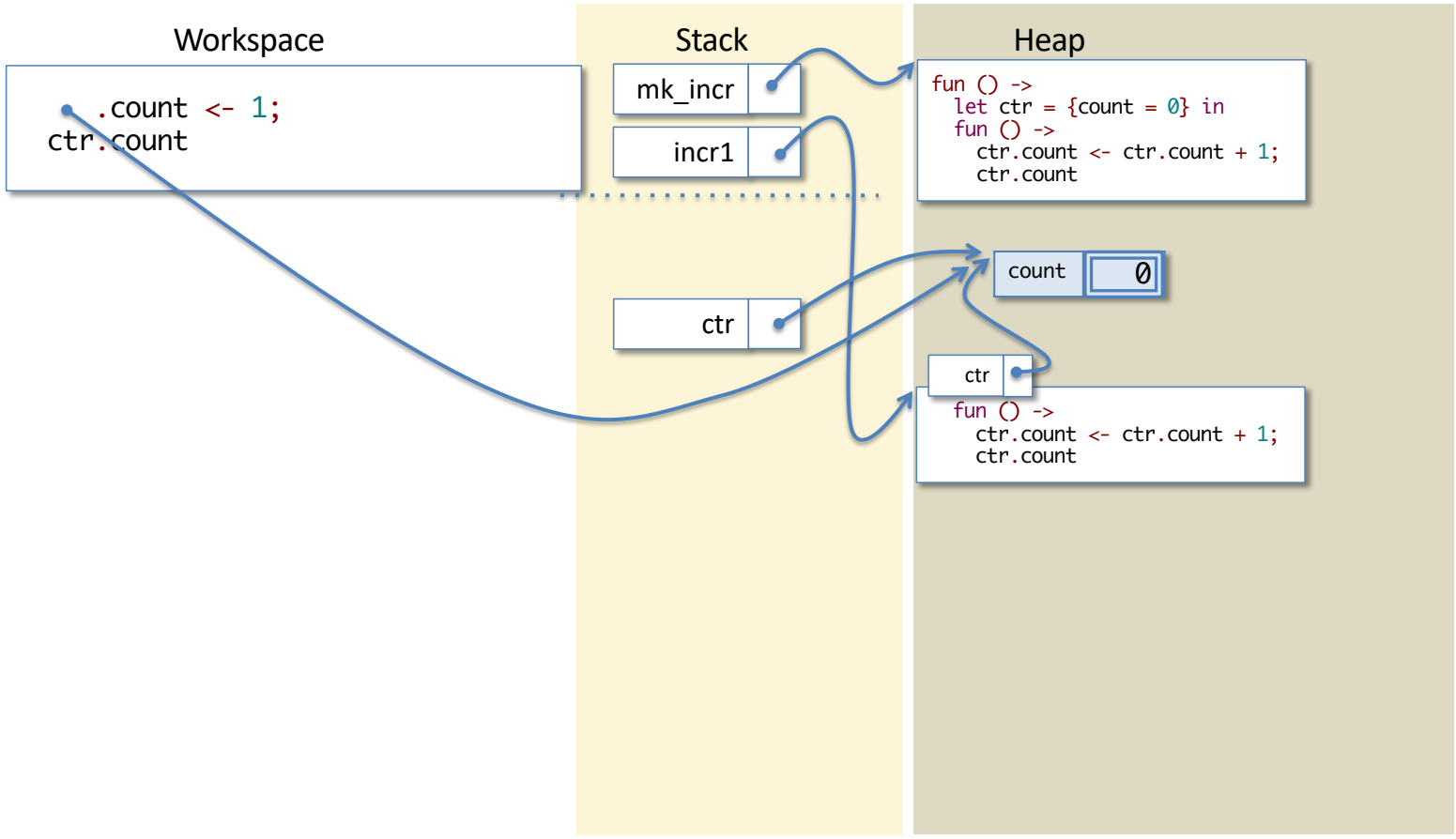
Now let's run "`incr1 ()`"



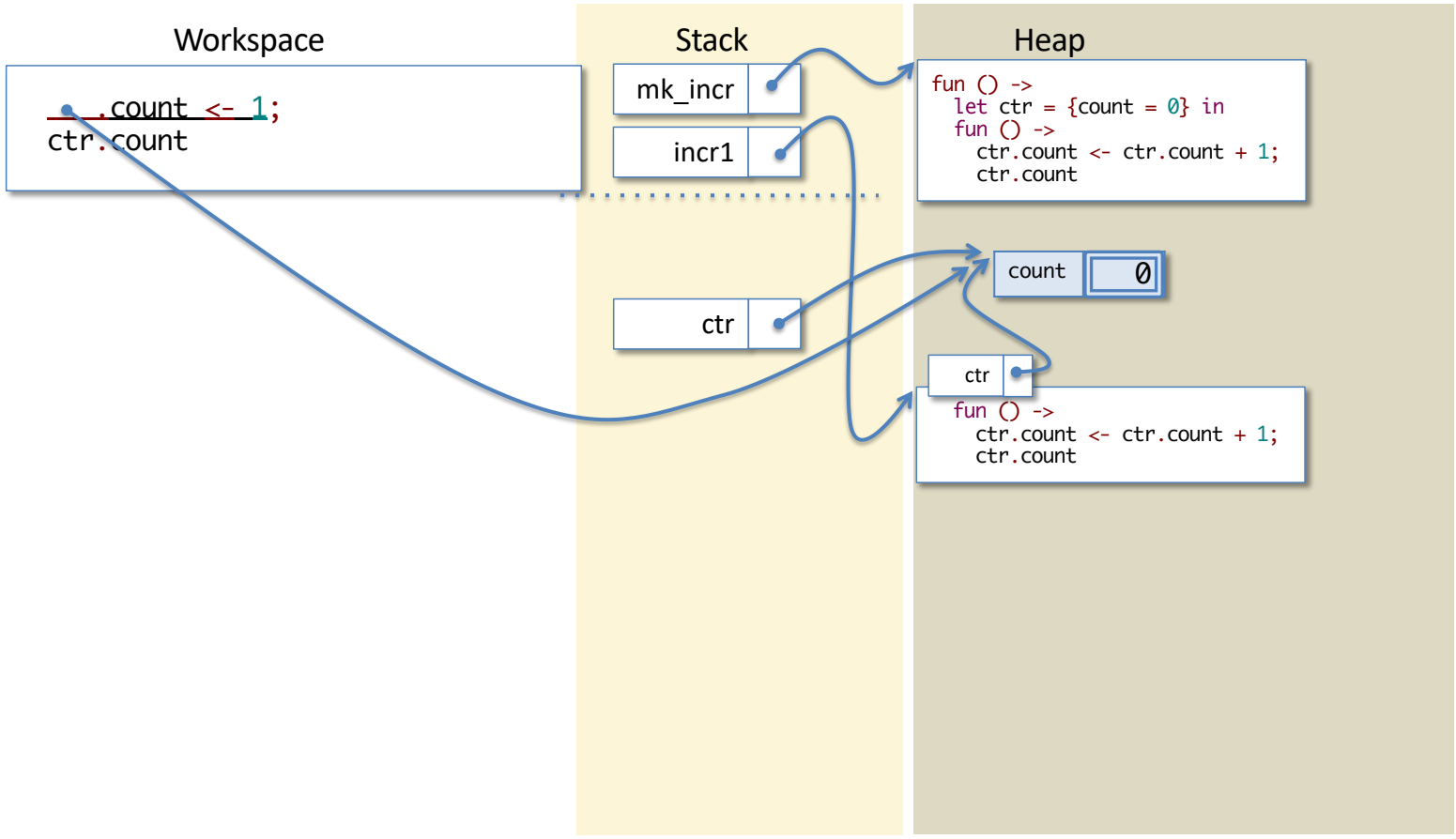
Now let's run "`incr1 ()`"



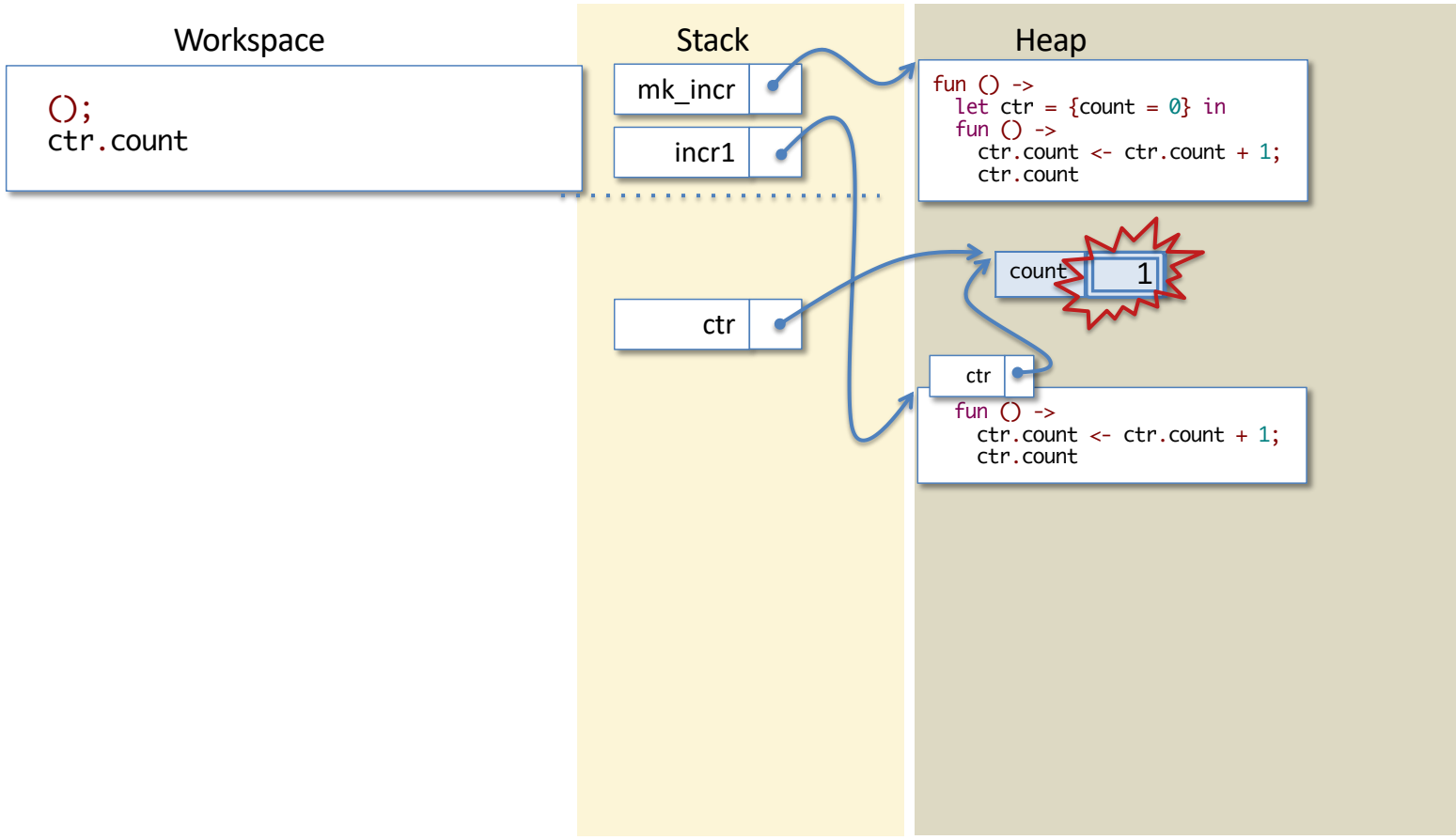
Now let's run "`incr1 ()`"



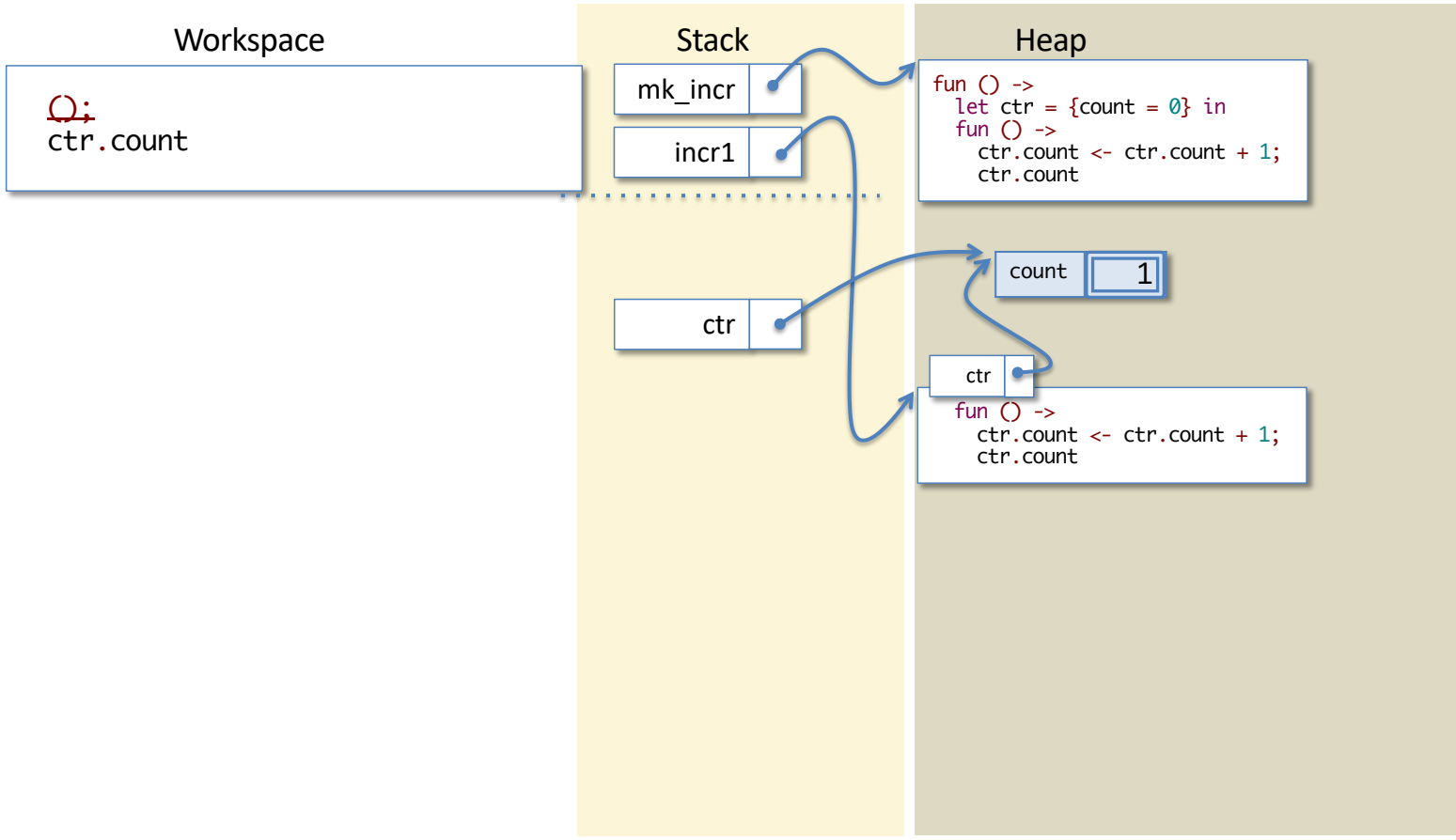
Now let's run "incr1 ()"



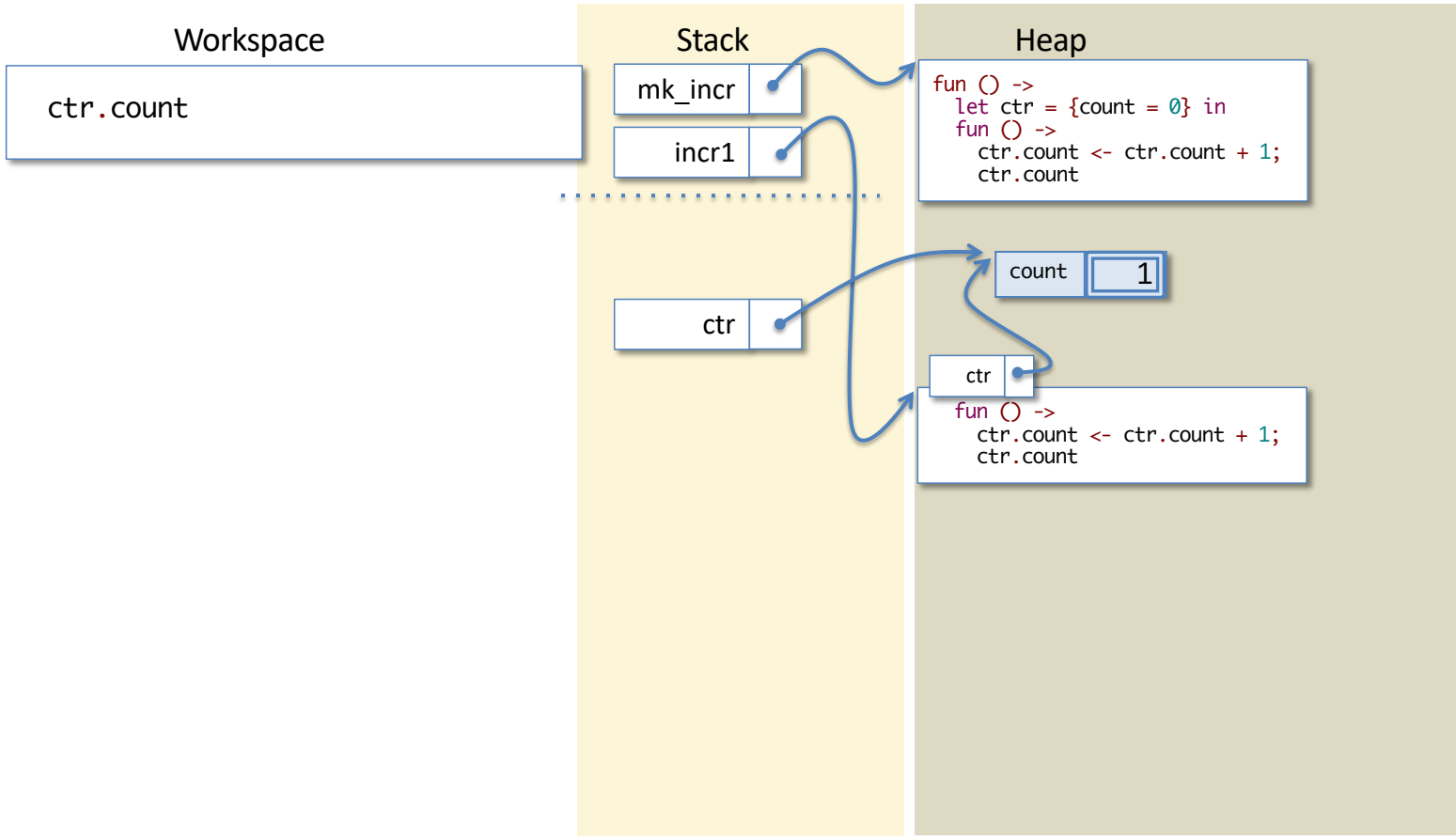
Now let's run "`incr1 ()`"



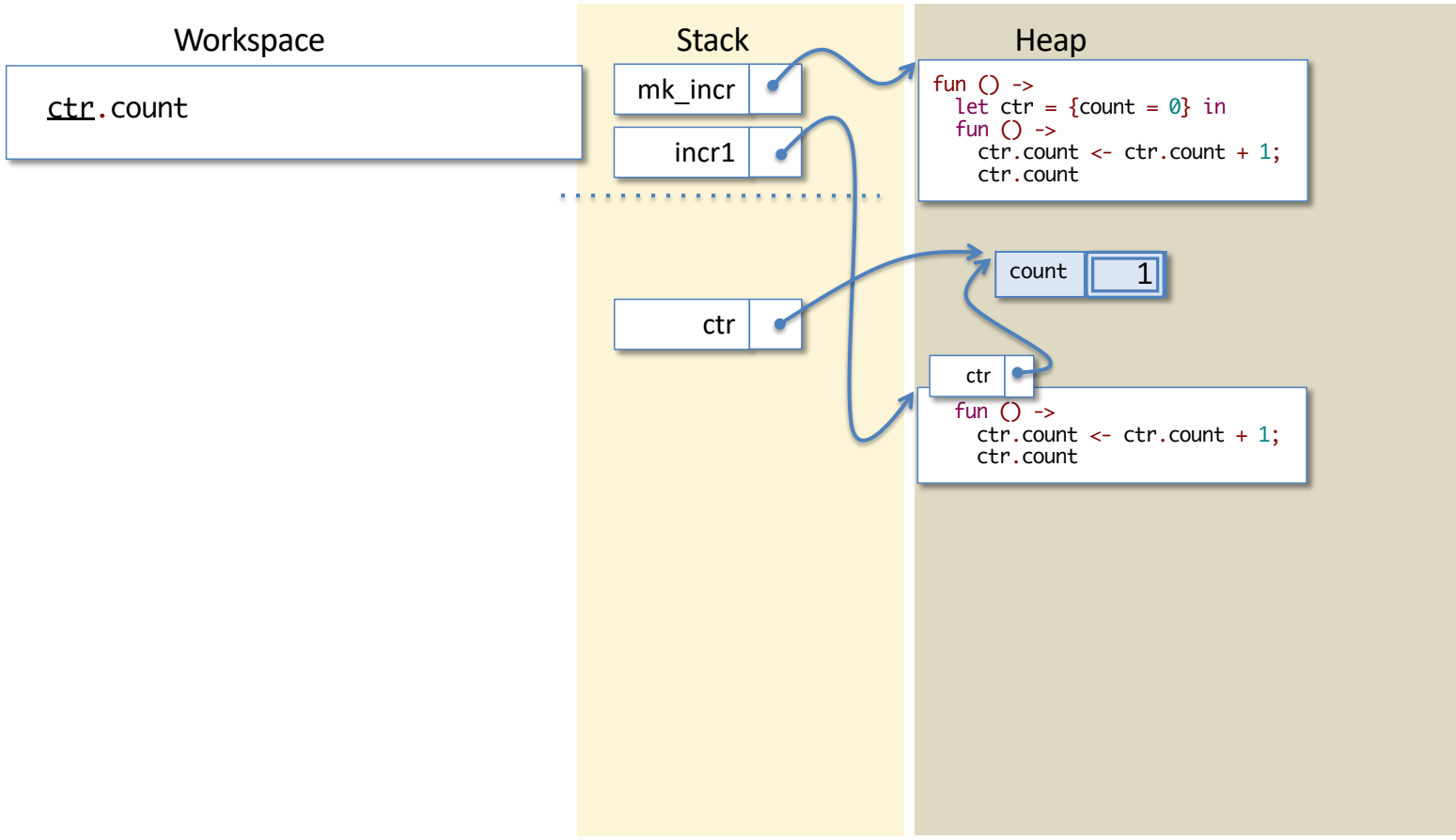
Now let's run "`incr1 ()`"



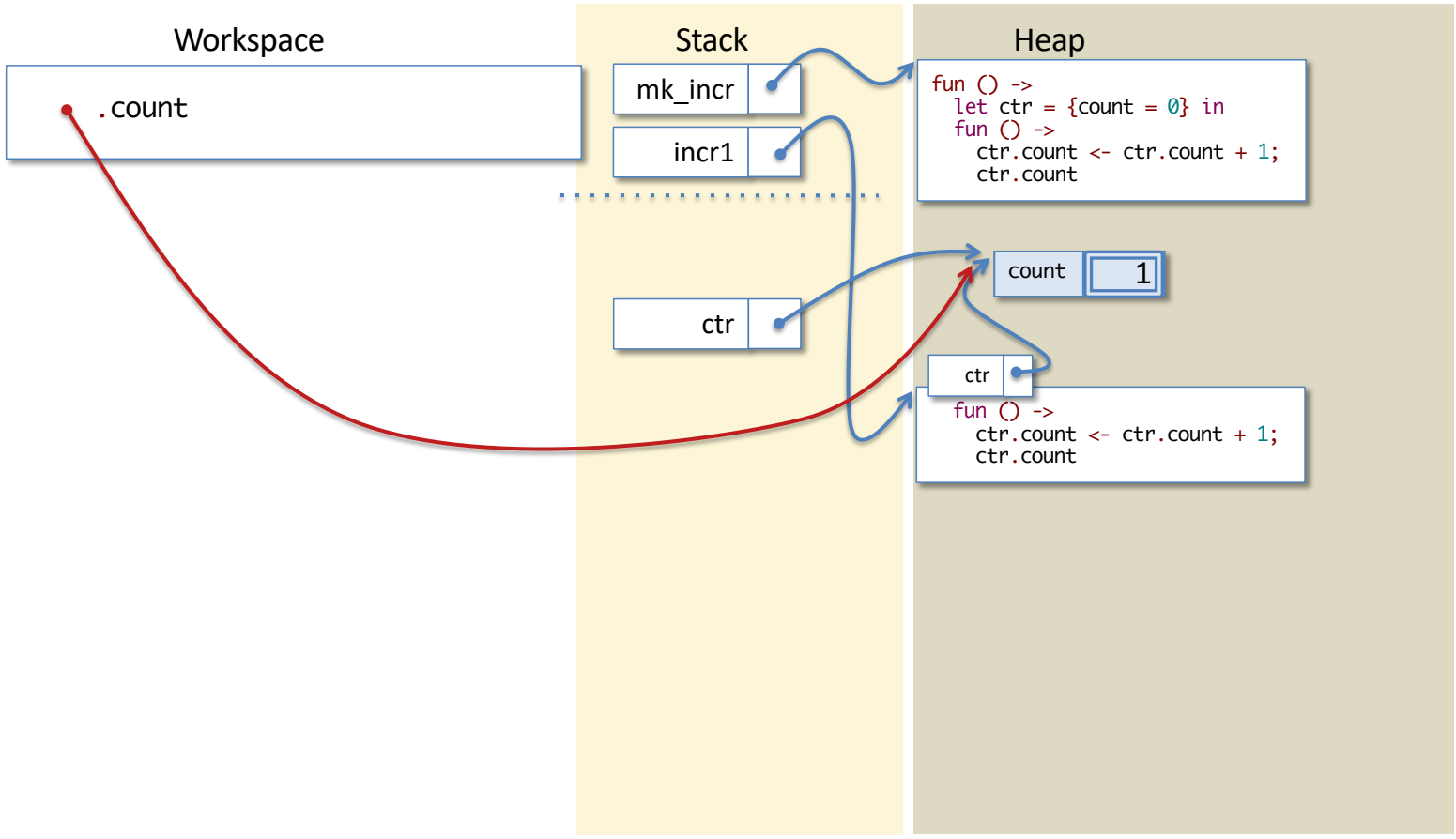
Now let's run "`incr1 ()`"



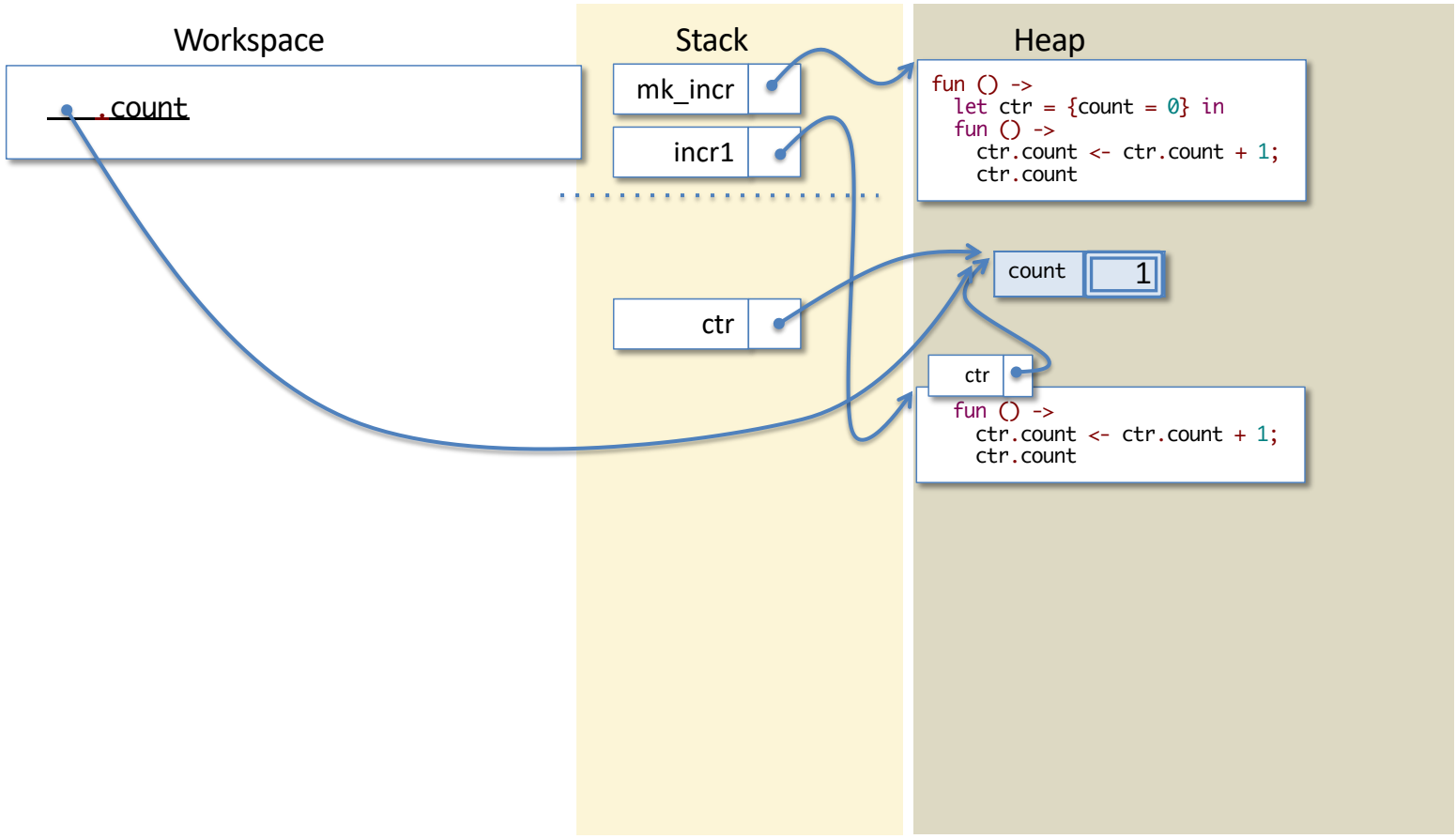
Now let's run "`incr1 ()`"



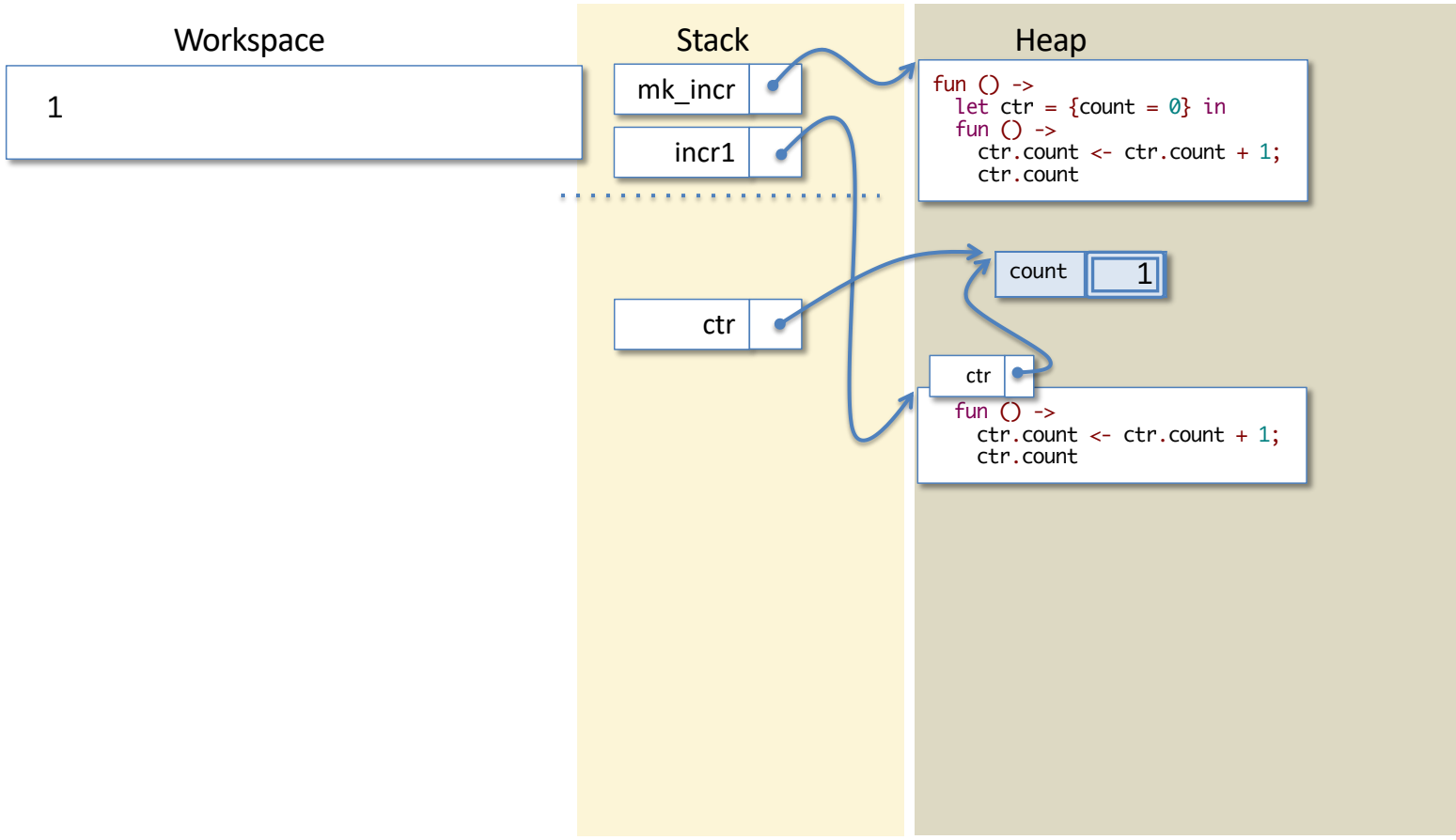
Now let's run "`incr1 ()`"



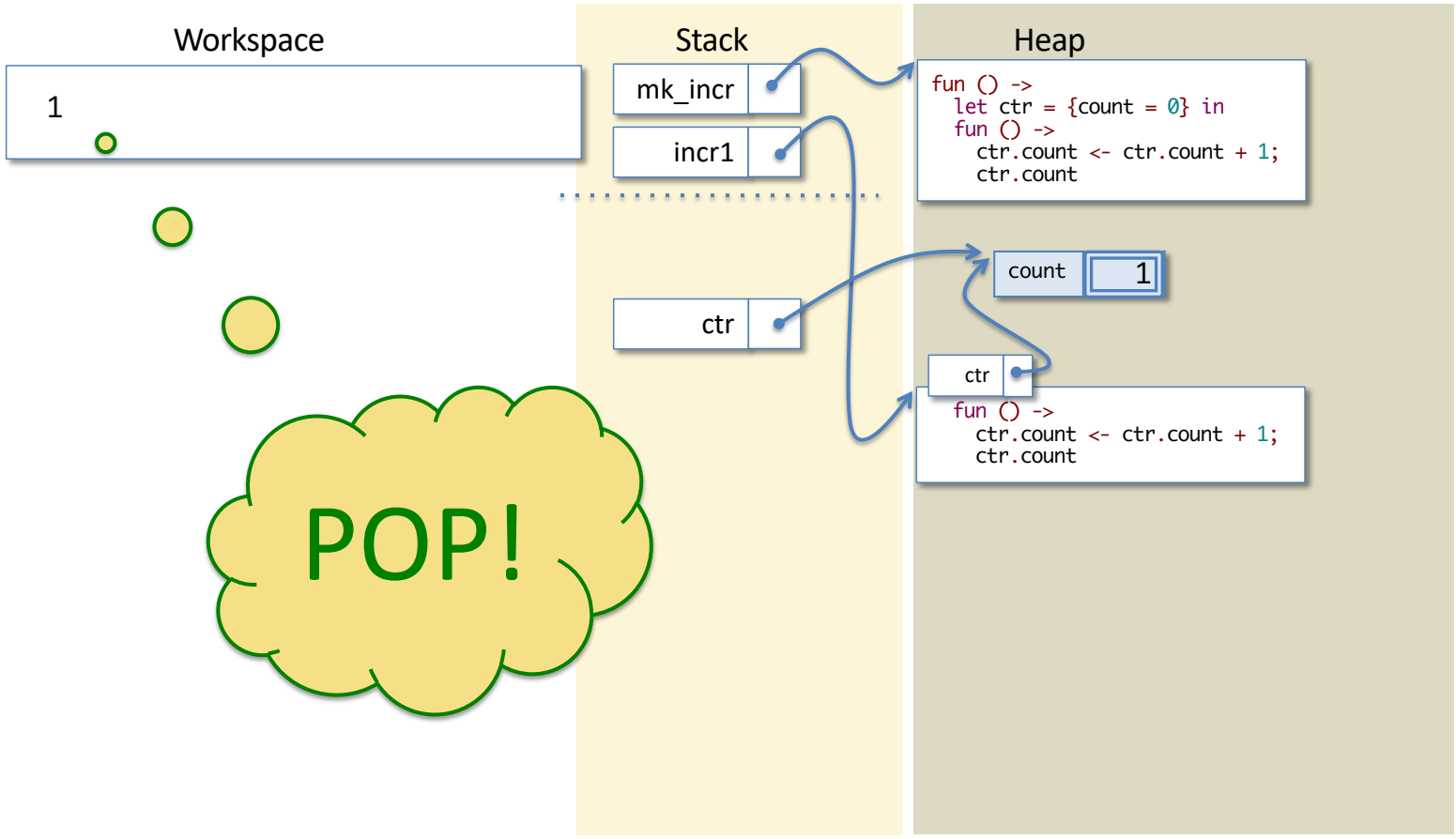
Now let's run "`incr1 ()`"



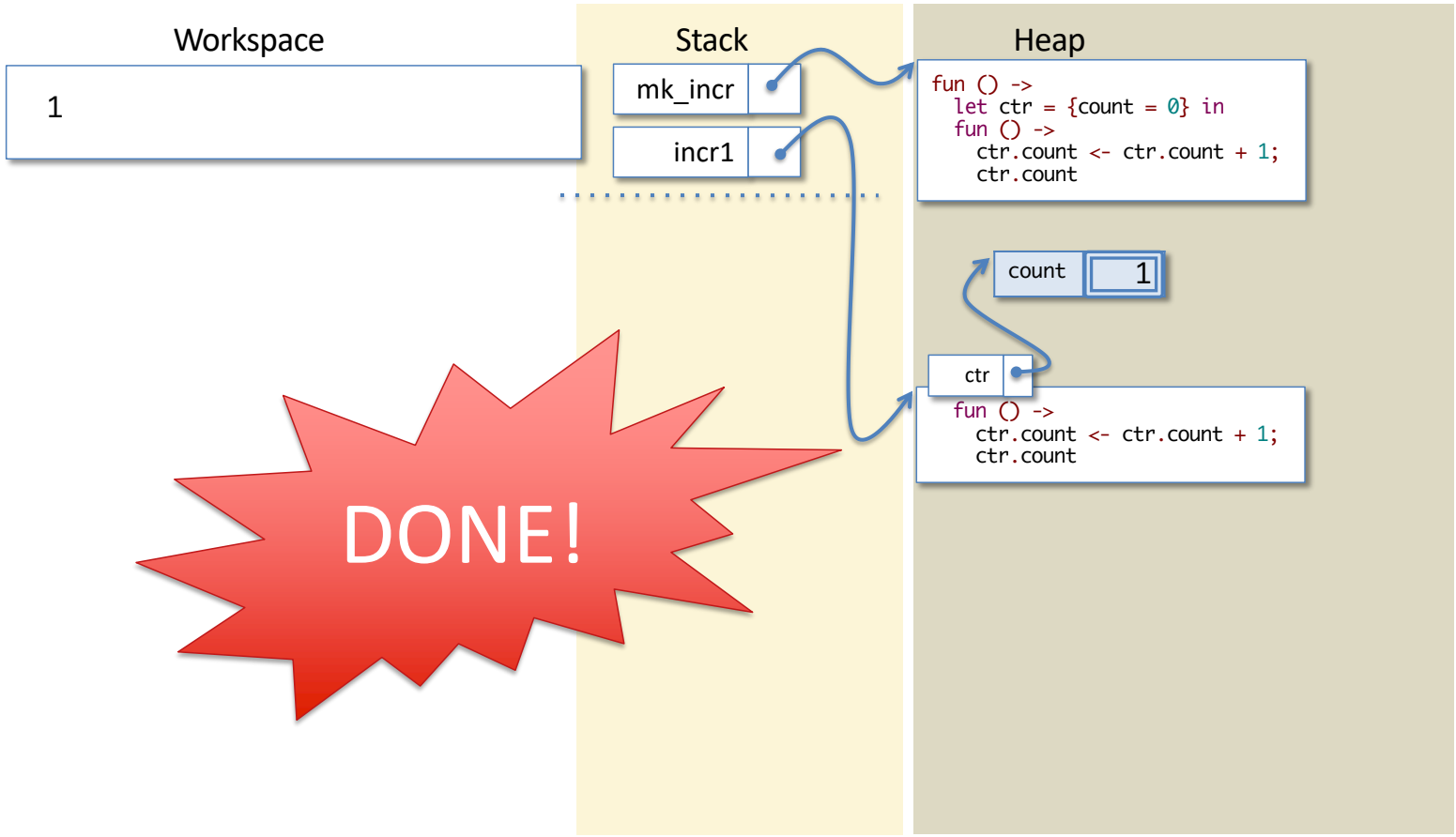
Now let's run "`incr1 ()`"



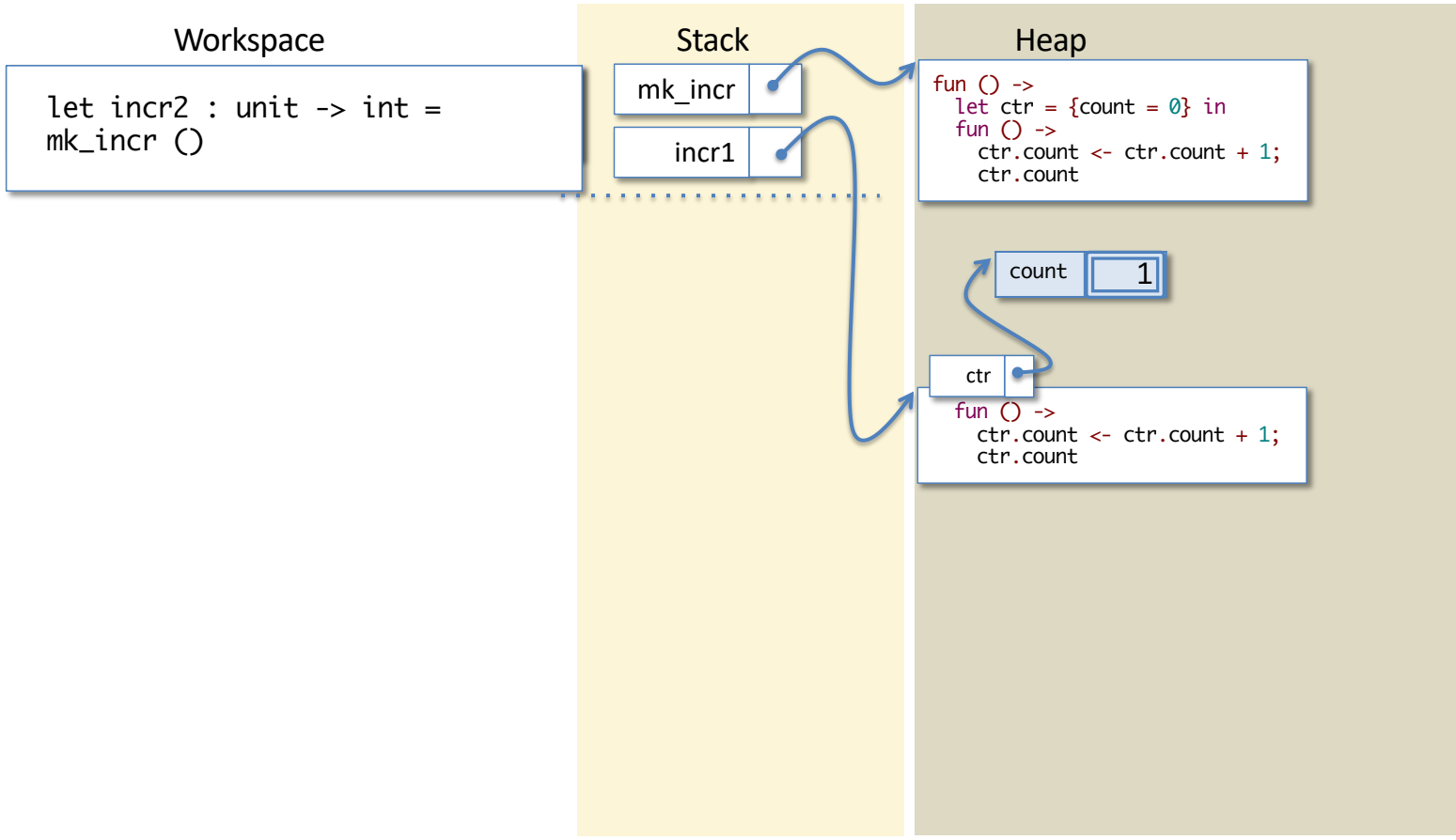
Now let's run "`incr1 ()`"



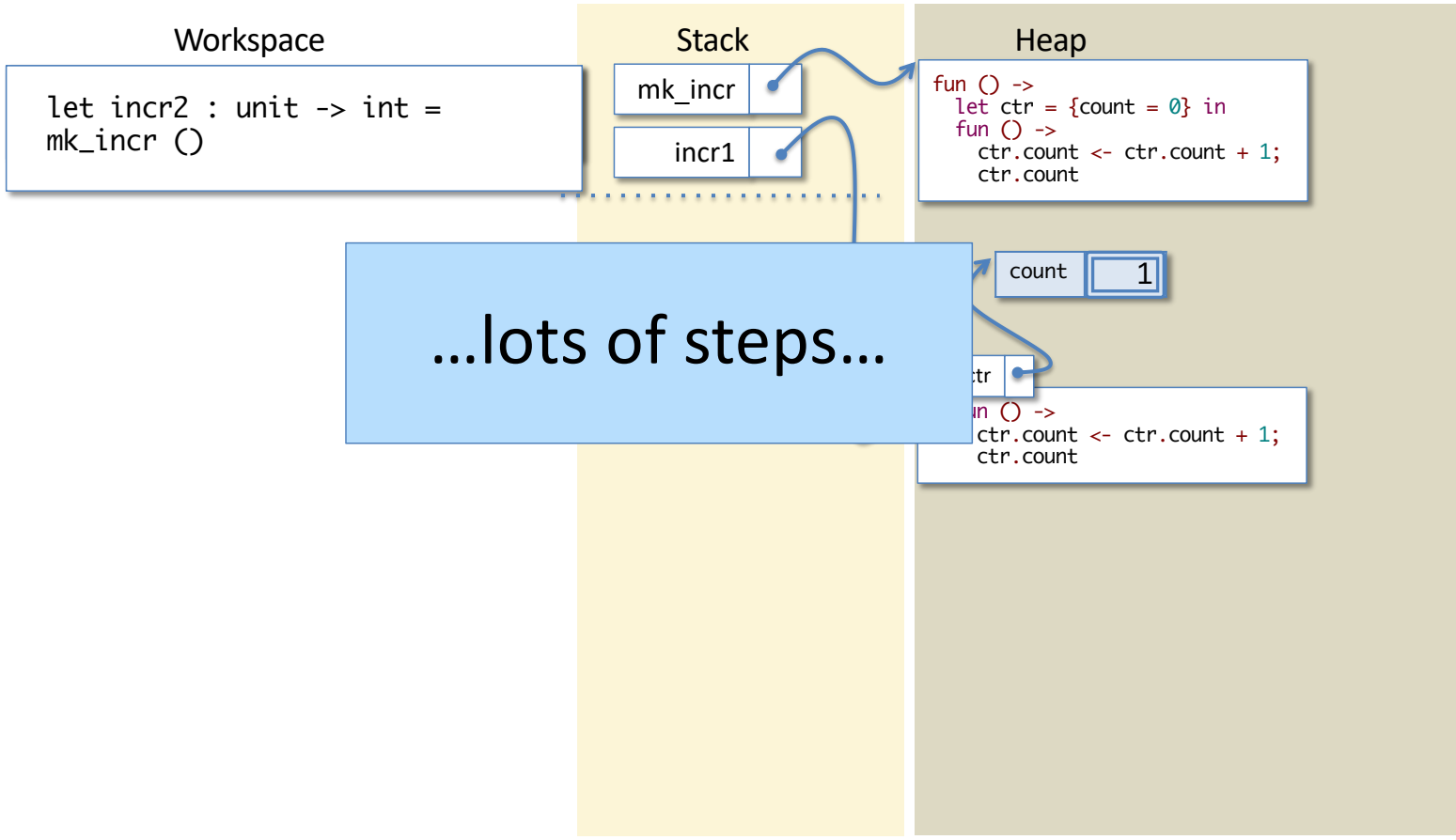
Now let's run "`incr1 ()`"



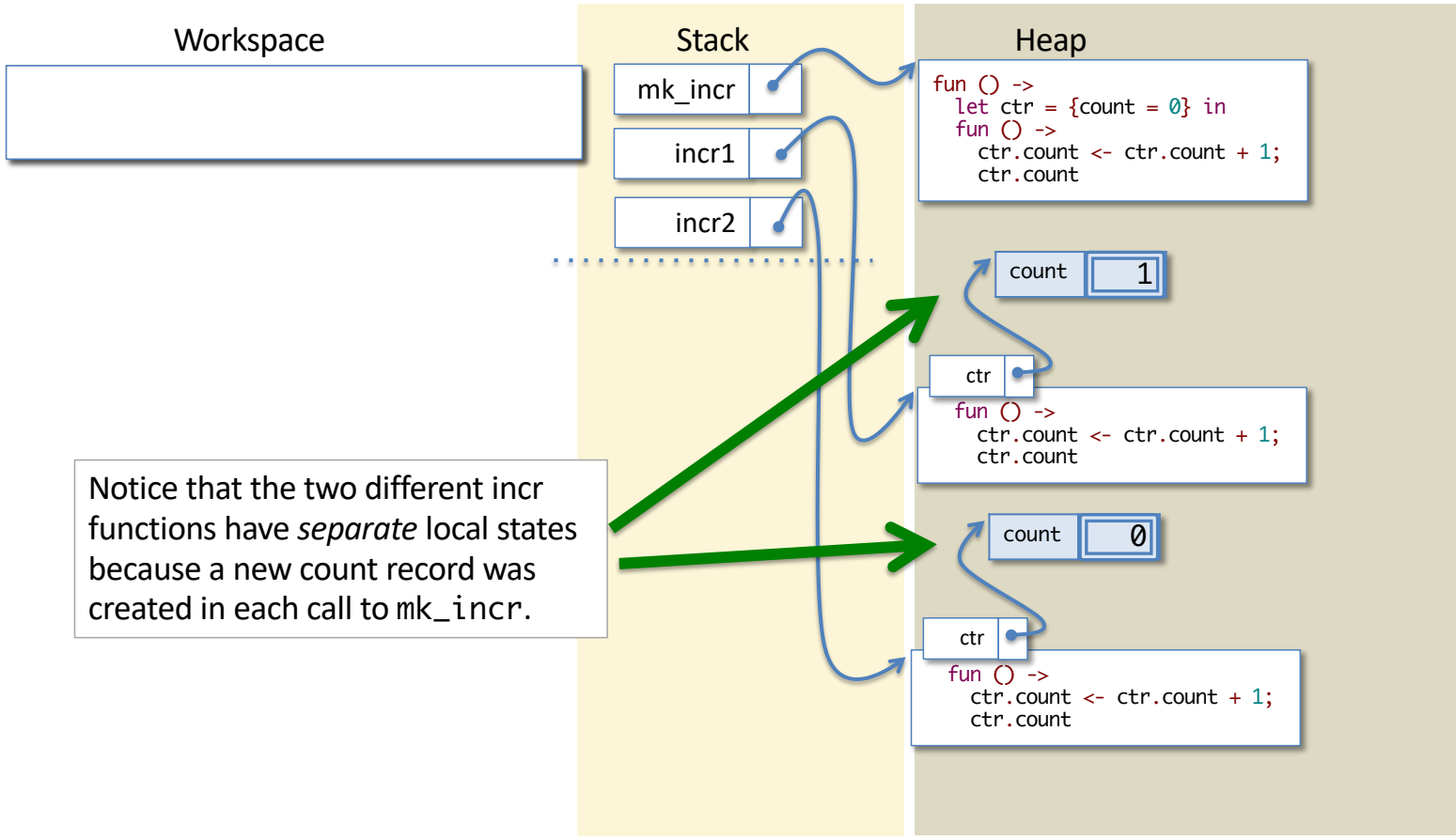
Now Let's run `mk_incr` again



Now Let's run `mk_incr` again



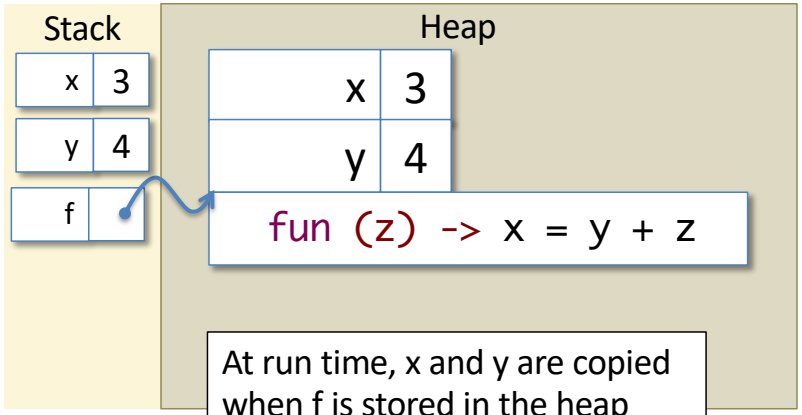
After creating `incr2`…



Key Idea: Closures

```
let f : int -> bool =
  let x : int = 3 in
  let y : int = 4 in
  (fun z -> x = z + y)
```

In the code, x and y are defined *in a local scope*



- **A *closure* is a function with local *bindings* (i.e., part of the stack), stored together on the heap**
  - Closures are the dynamic (run time) implementation of static scope
  - When functions are allocated on the heap, we **copy** part of the stack
  - When the functions are called, the copy goes back on the stack
- **Only immutable variables can be stored in closures**
  - All variables in OCaml are immutable (even if they point to mutable data structures in the heap)

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
  - this state is *encapsulated* because it is only accessible by the closure
- What if we wanted to bundle together *multiple* operations that share the *same* local state?
  - e.g. incr and decr operations that work on the *same* counter state

Key Concept: *Object*

An object consists of:

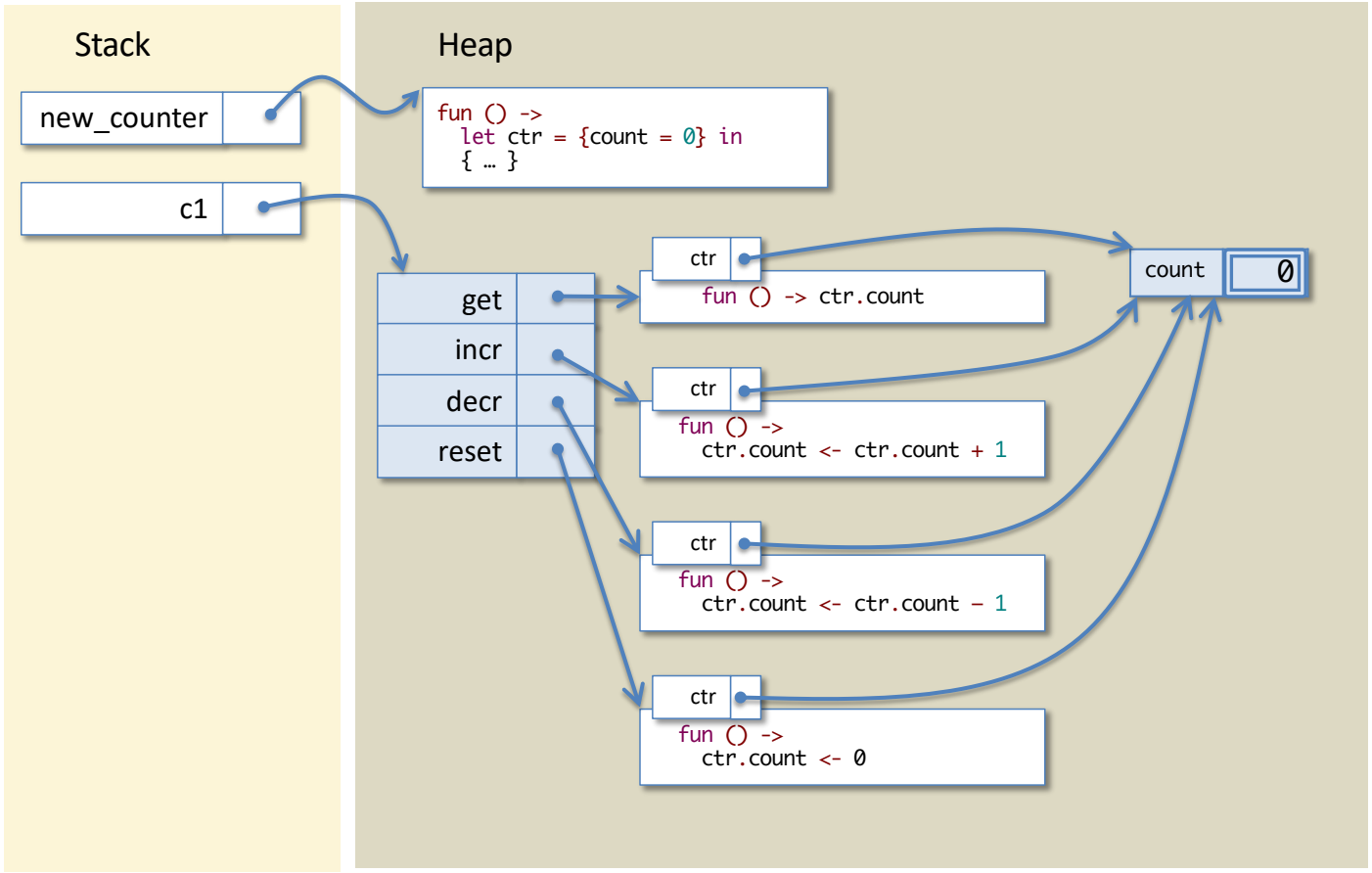
- encapsulated mutable state (*fields*)
- operations that manipulate that state (*methods*)

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 ()))
```

Objects and GUIs

Where we're going…

- HW 5: Build a GUI library and client application *from scratch* in OCaml

- Goals:
  - Practice with *first-class functions* and *hidden state (Ch 17)*
  - Bridge to object-oriented programming in Java
  - Illustrate the *event-driven programming* model
  - Give a feel for how GUI libraries (like Java's Swing) are put together
  - Apply everything we've seen so far to do some pretty serious programming

17: Have you ever used a GUI library (such as Java's Swing) to construct a user interface?

0

Yes

0%

No

0%

Building a GUI library & application