Programming Languages and Techniques (CIS120)

Lecture 18

February 24th, 2016

"Objects"

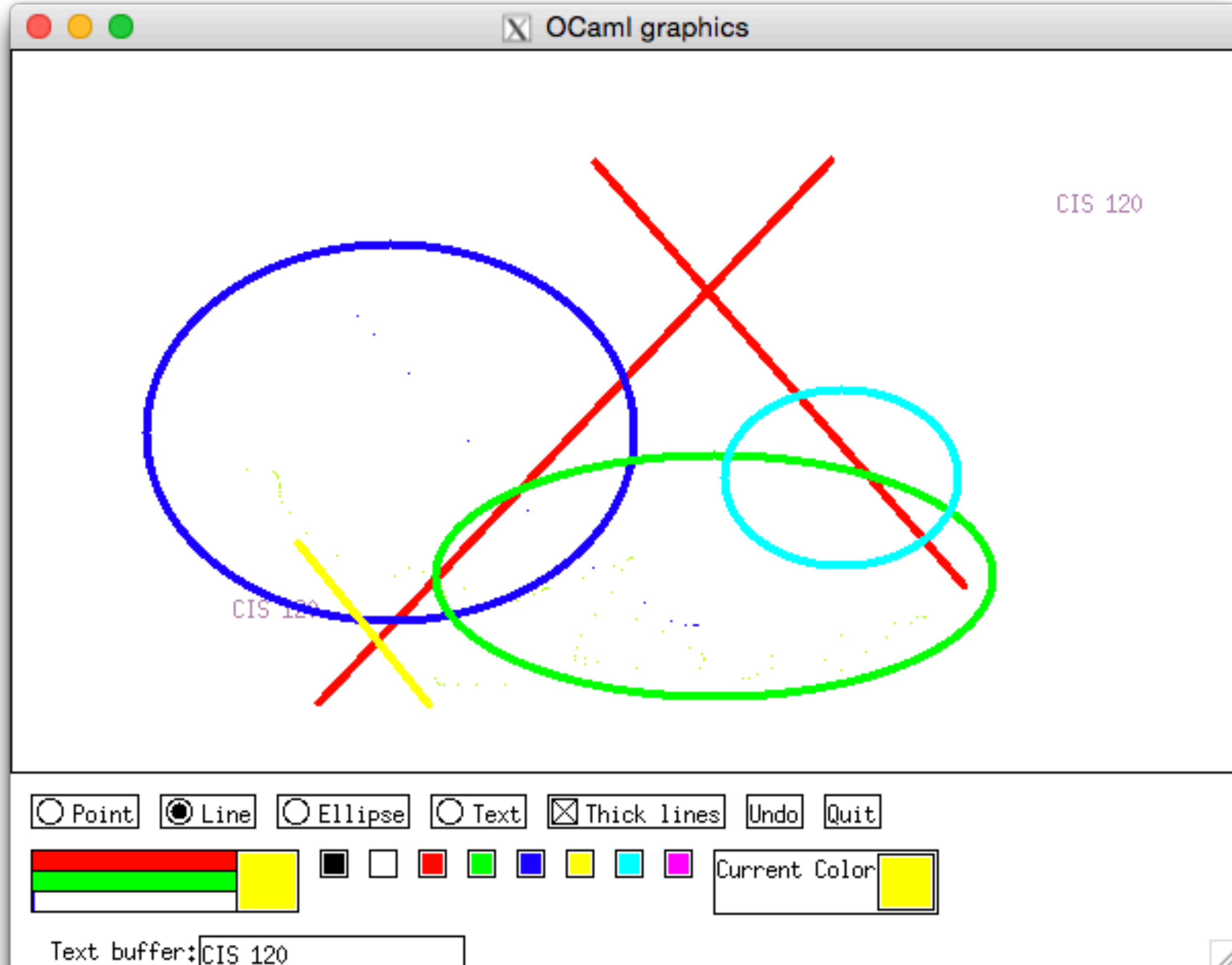
GUI project overview

Announcements

- Midterm exam
  - Solutions available on course website
  - View exams with Ms. Caliman (Levine 309)
  - If you would like a copy of your exam, send her an email (jackie@seas.upenn.edu) by Thursday at 9AM. She will have the copy available for you on Friday.

- HW5: GUI & Paint
  - Available on the web site
  - Due Thursday, March 3rd at midnight

Building a GUI and GUI Applications



Where we're going…

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

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

“Objects” and Hidden State

Encapsulating State

What number is printed by this program?

```
type state = { mutable count : int }
let f =
  let p = { count = 2 } in
  fun (y : int) -> p.count + y
let p = { count = 3 }
;; 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

Workspace

```
let mk_incr : unit -> unit ->
int = 

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 mk_incr : unit -> unit ->
int =   .

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

mk_incr

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

mk_incr

Heap

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

Running mk_incr

Workspace

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

Stack

mk_incr	

Heap

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

Running mk_incr

Workspace

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

Stack

mk_incr	

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

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

Stack

mk_incr	●

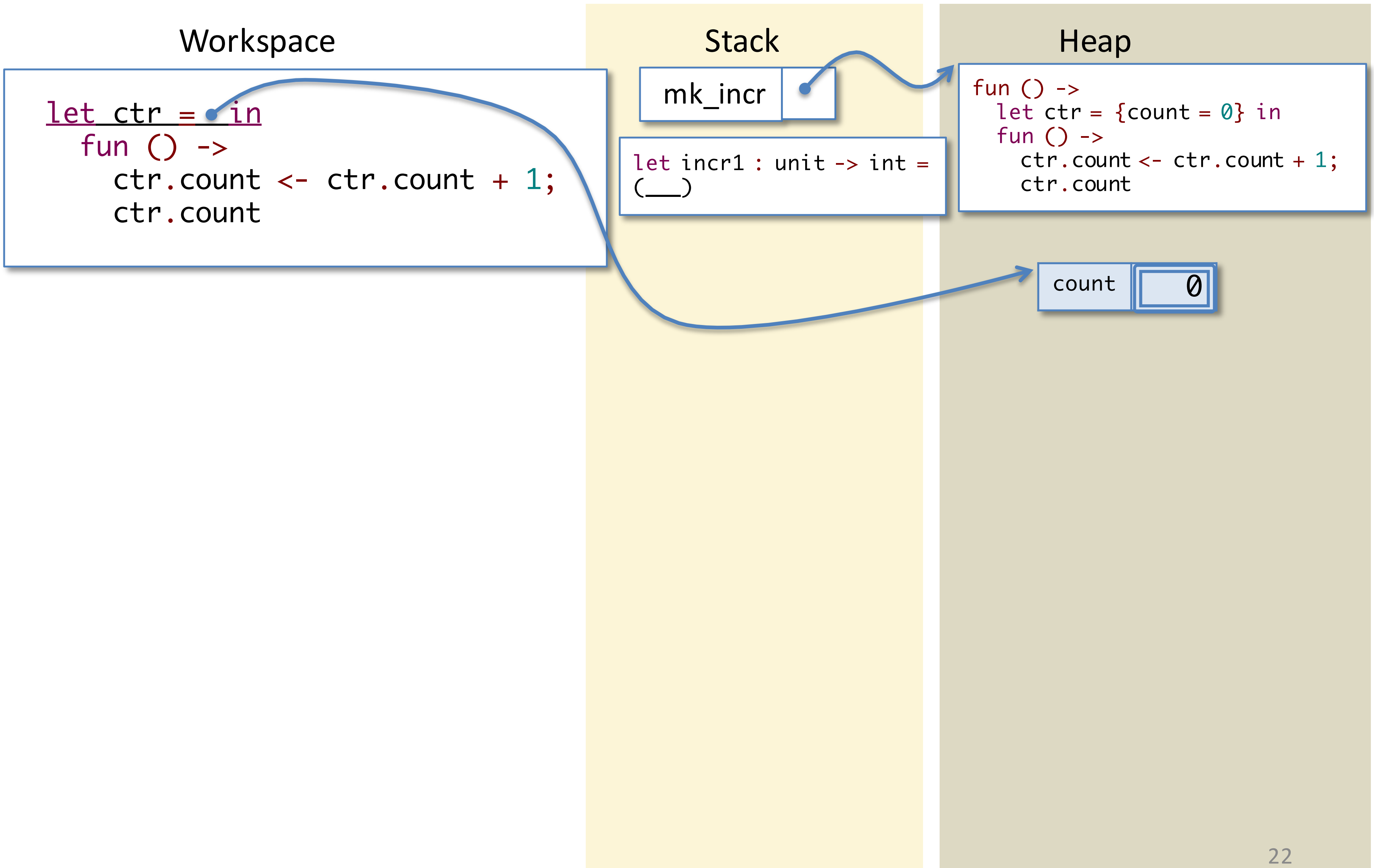
```
let incr1 : unit -> int =
(___)
```

Heap

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

count	0

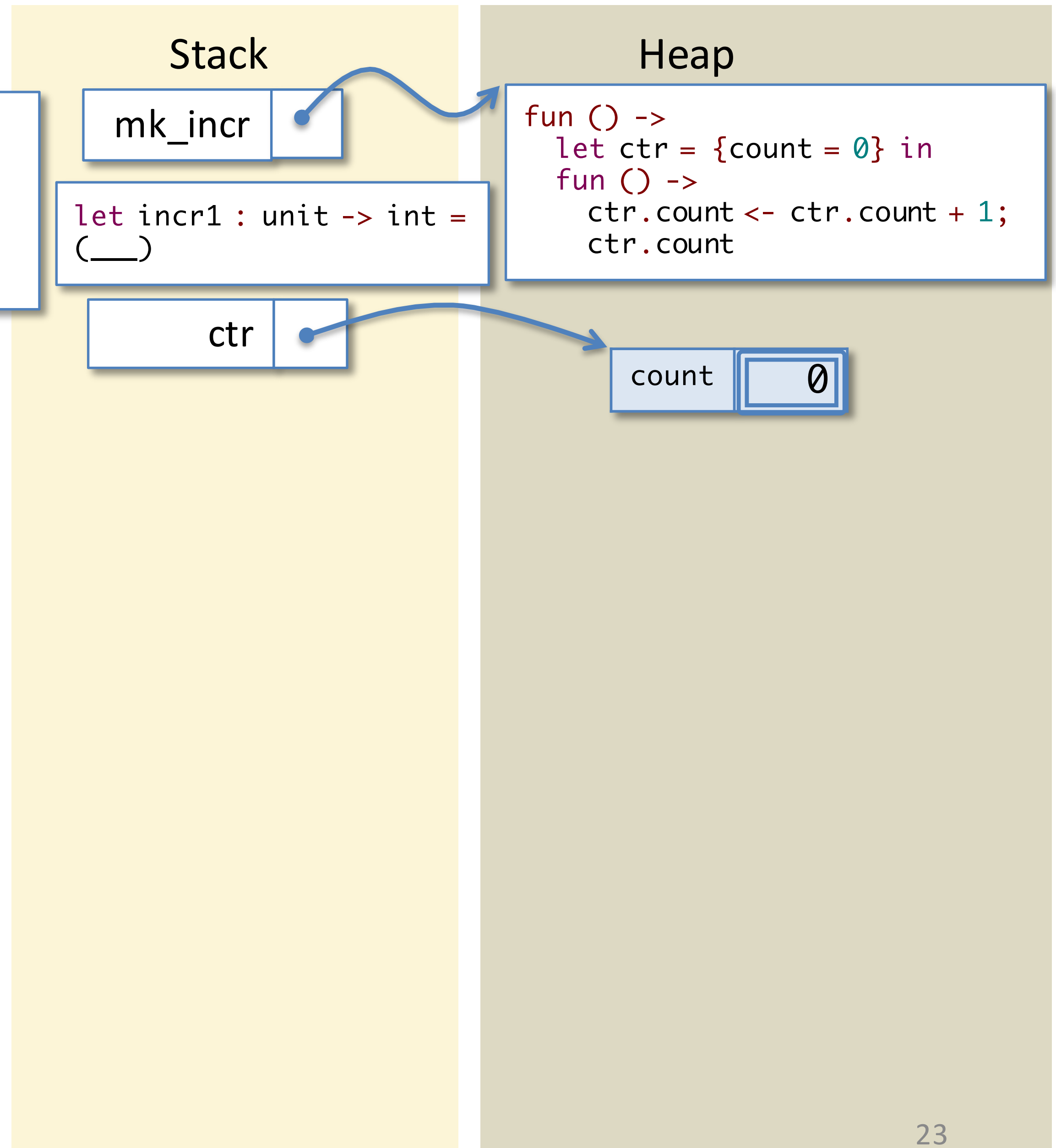
Running mk_incr



Running mk_incr

Workspace

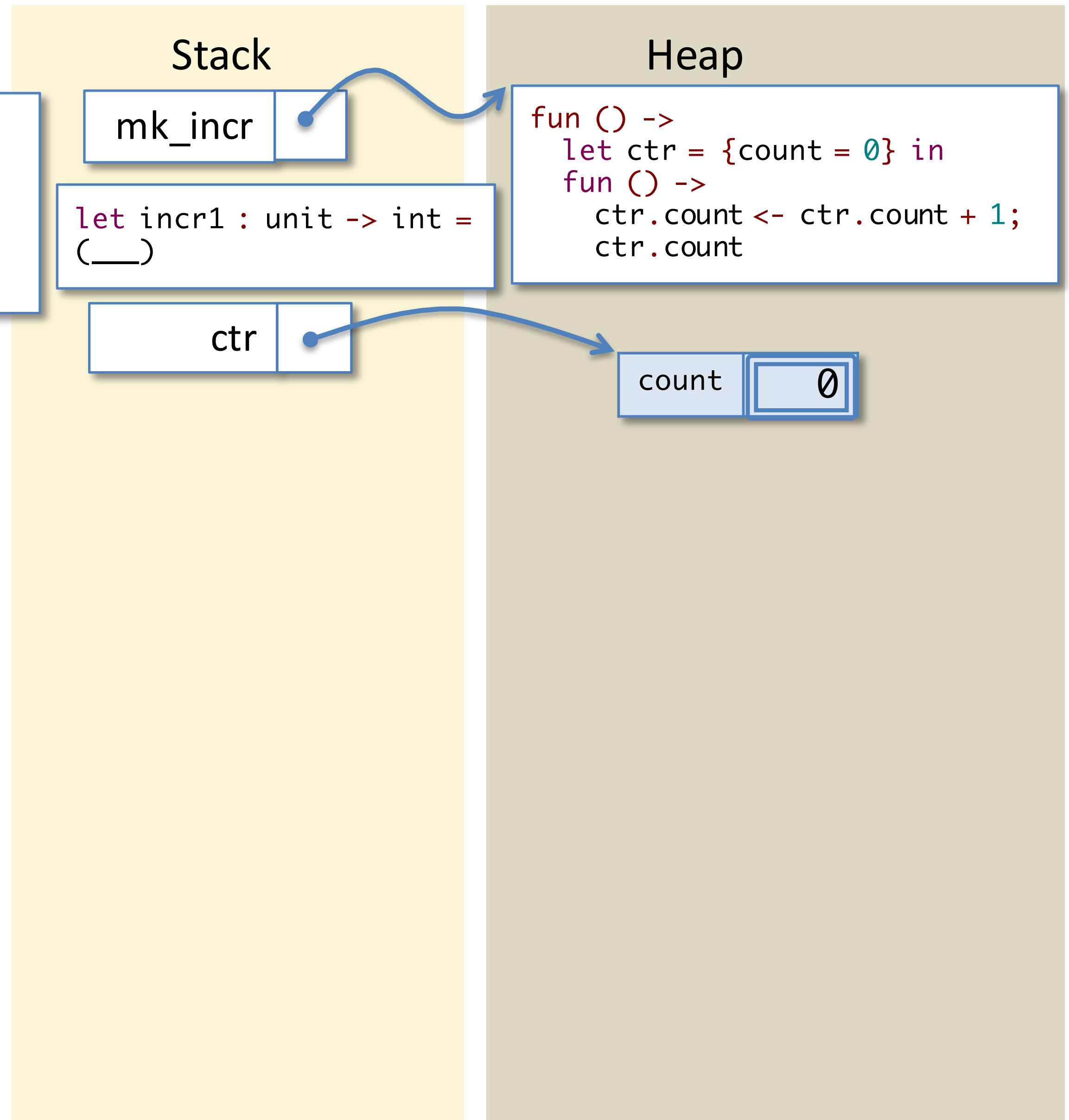
```
fun () ->
  ctr.count <- ctr.count + 1;
  ctr.count
```



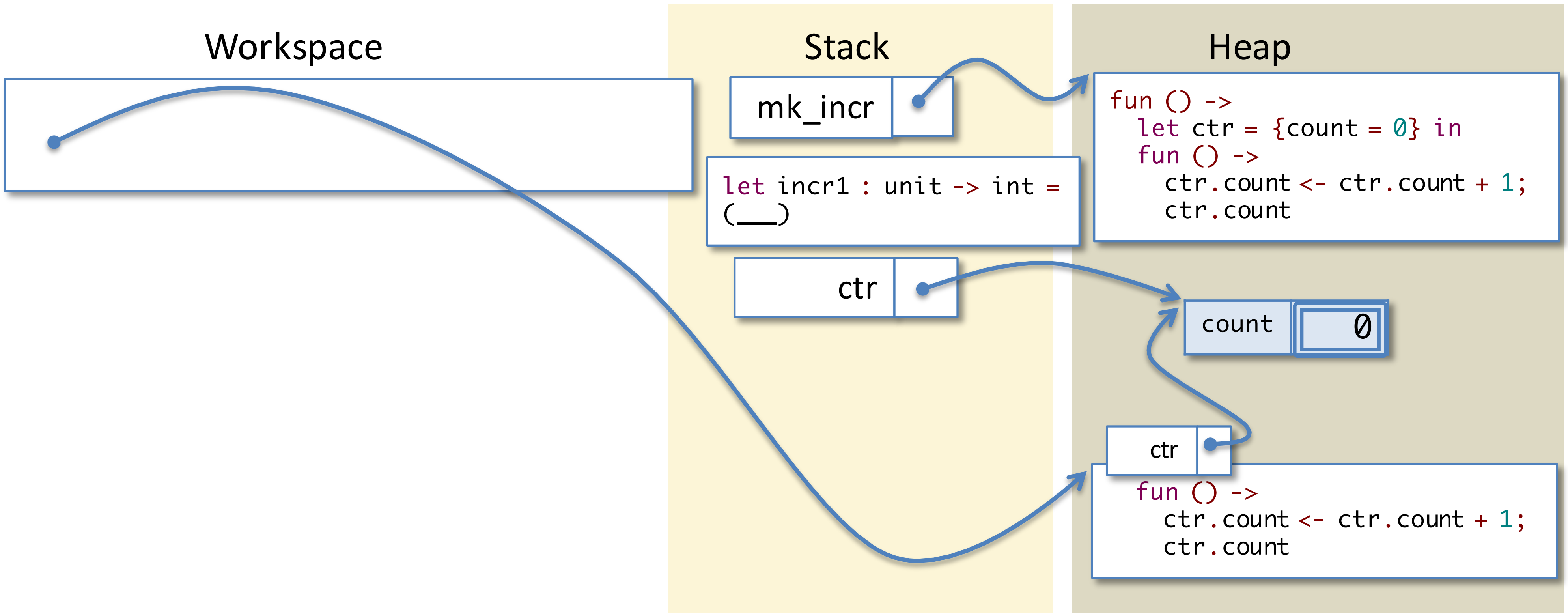
Running mk_incr

Workspace

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



Local Functions

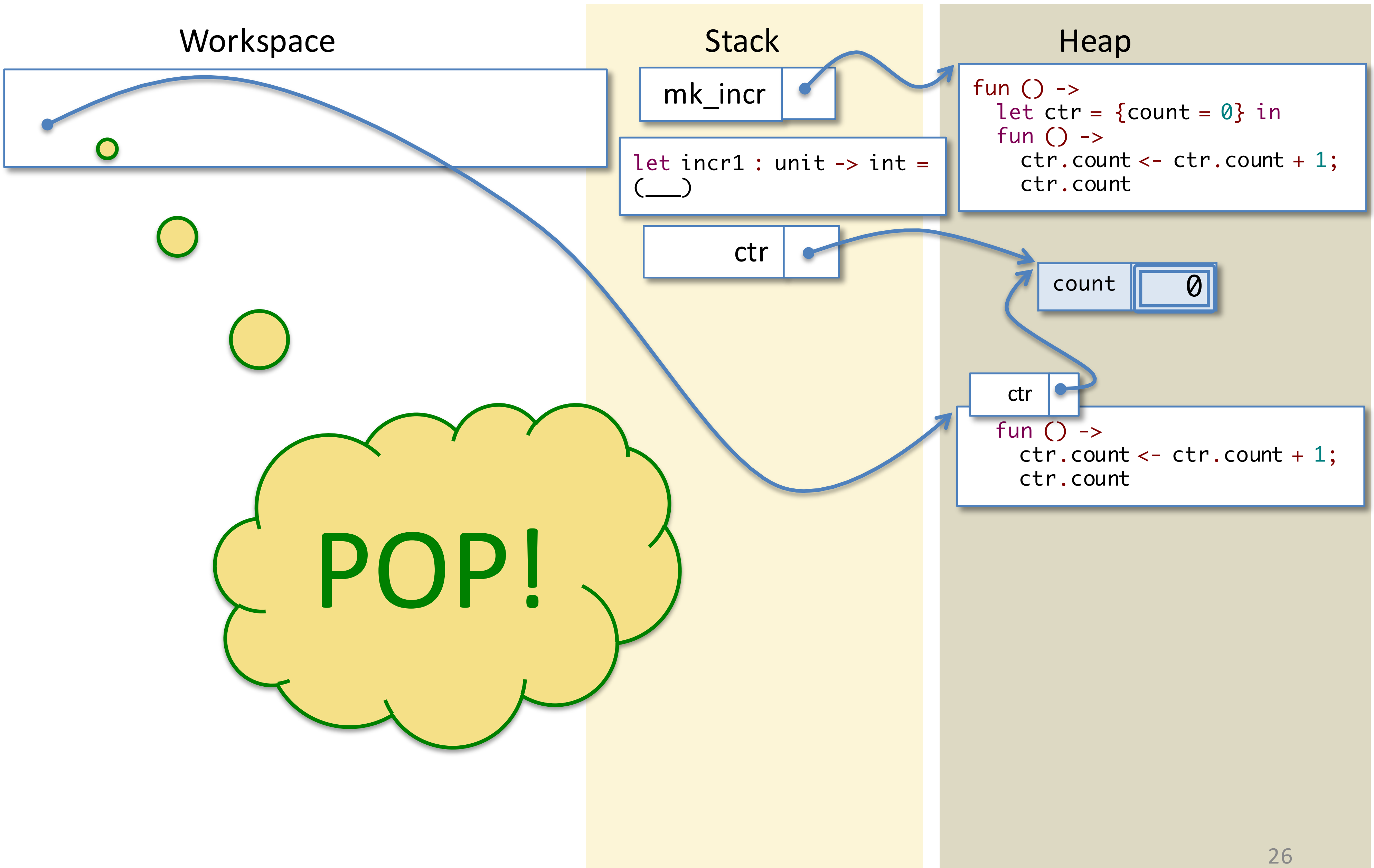


NOTE: We need one refinement of the ASM model to handle local functions. Why?

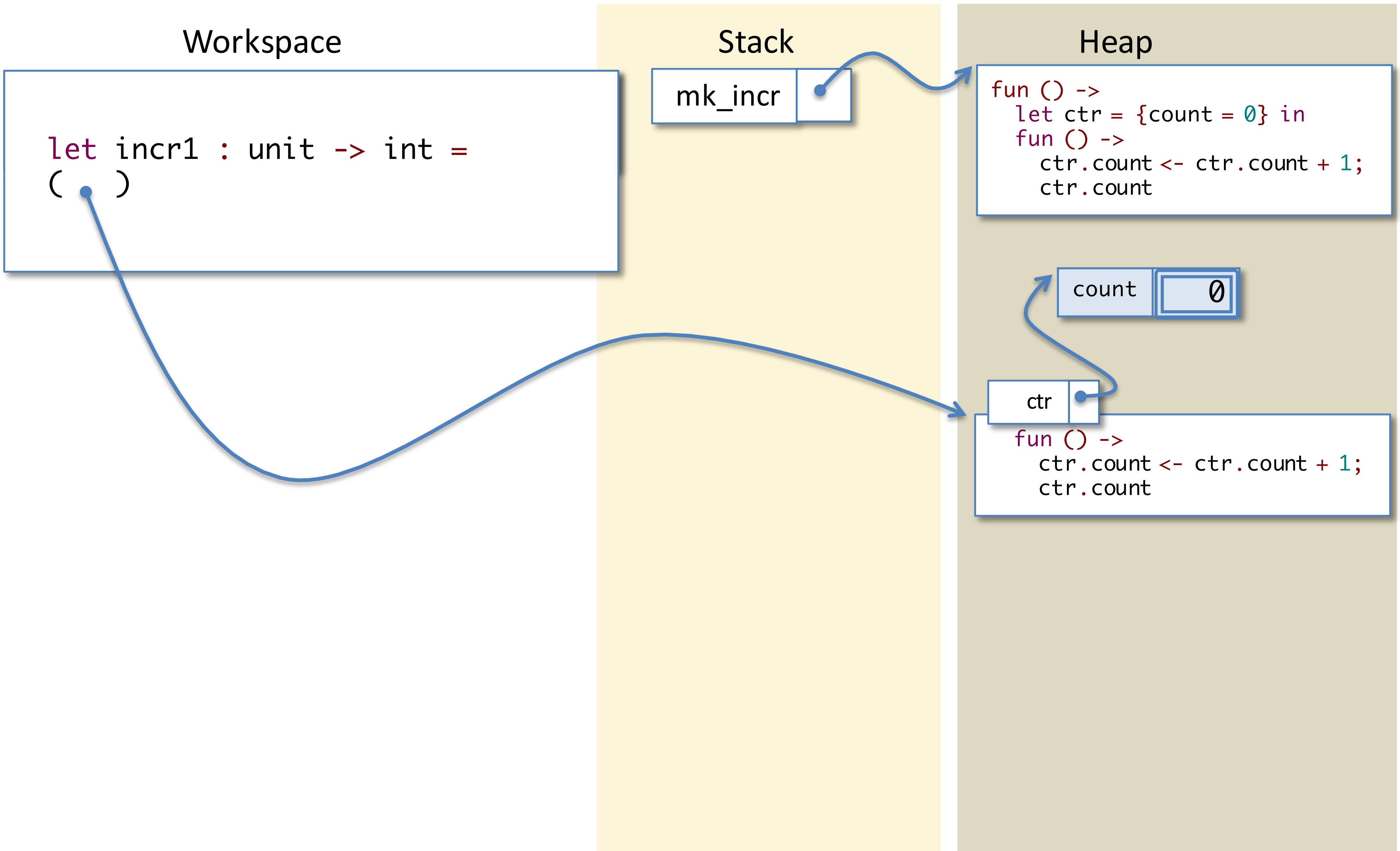
The function mentions "ctr", which is on the stack (but about to be popped off)…

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

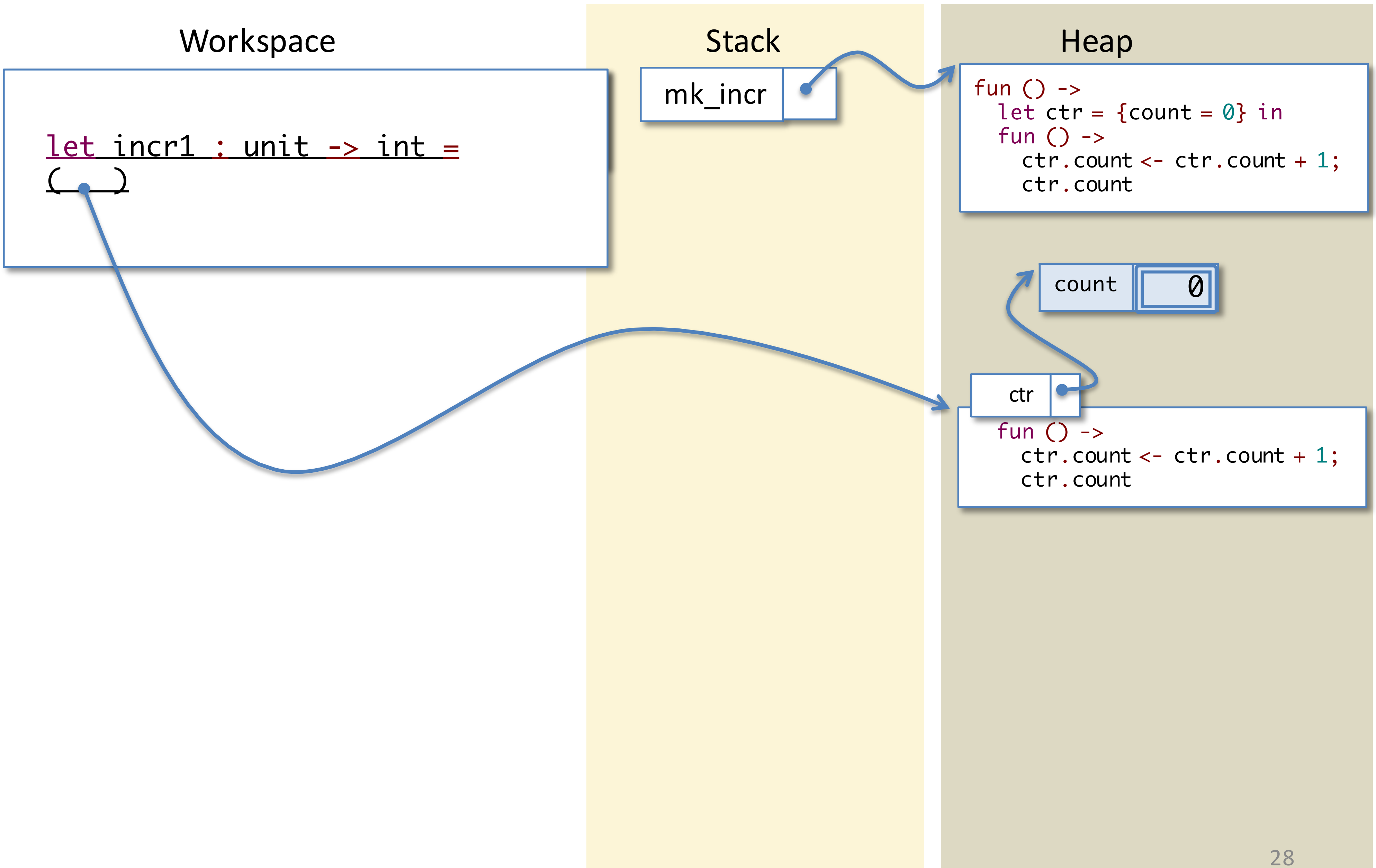
Local Functions



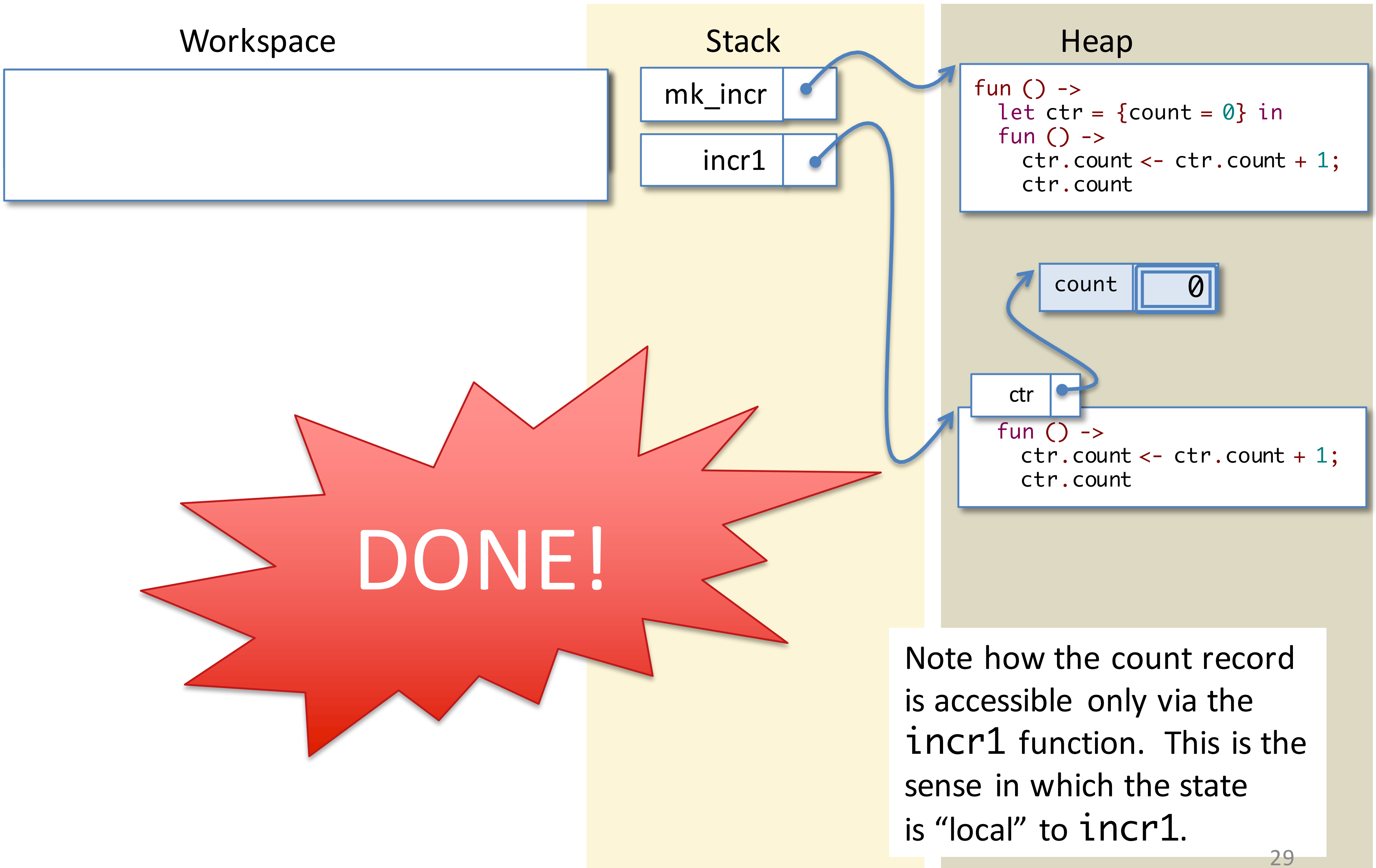
Local Functions



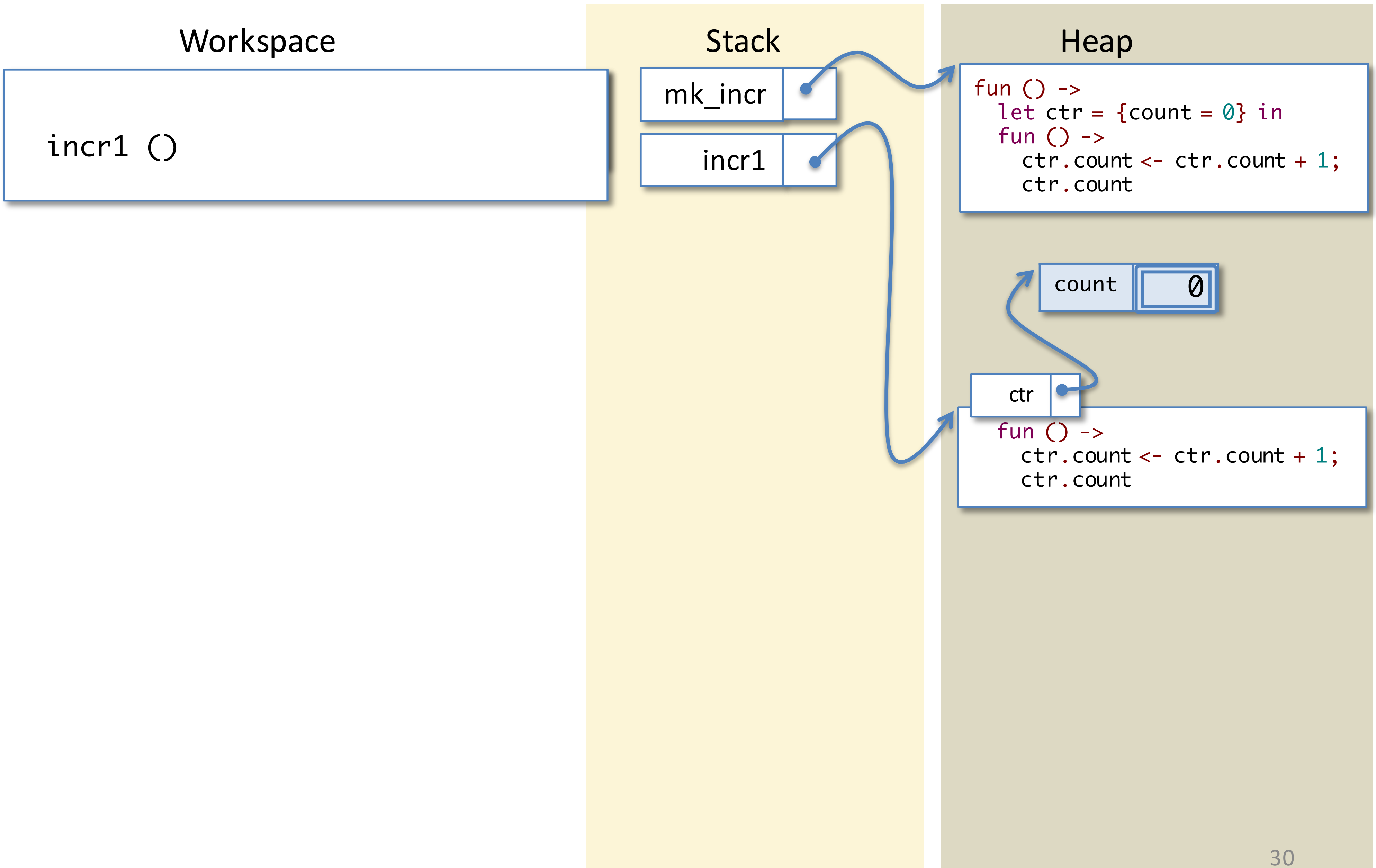
Local Functions



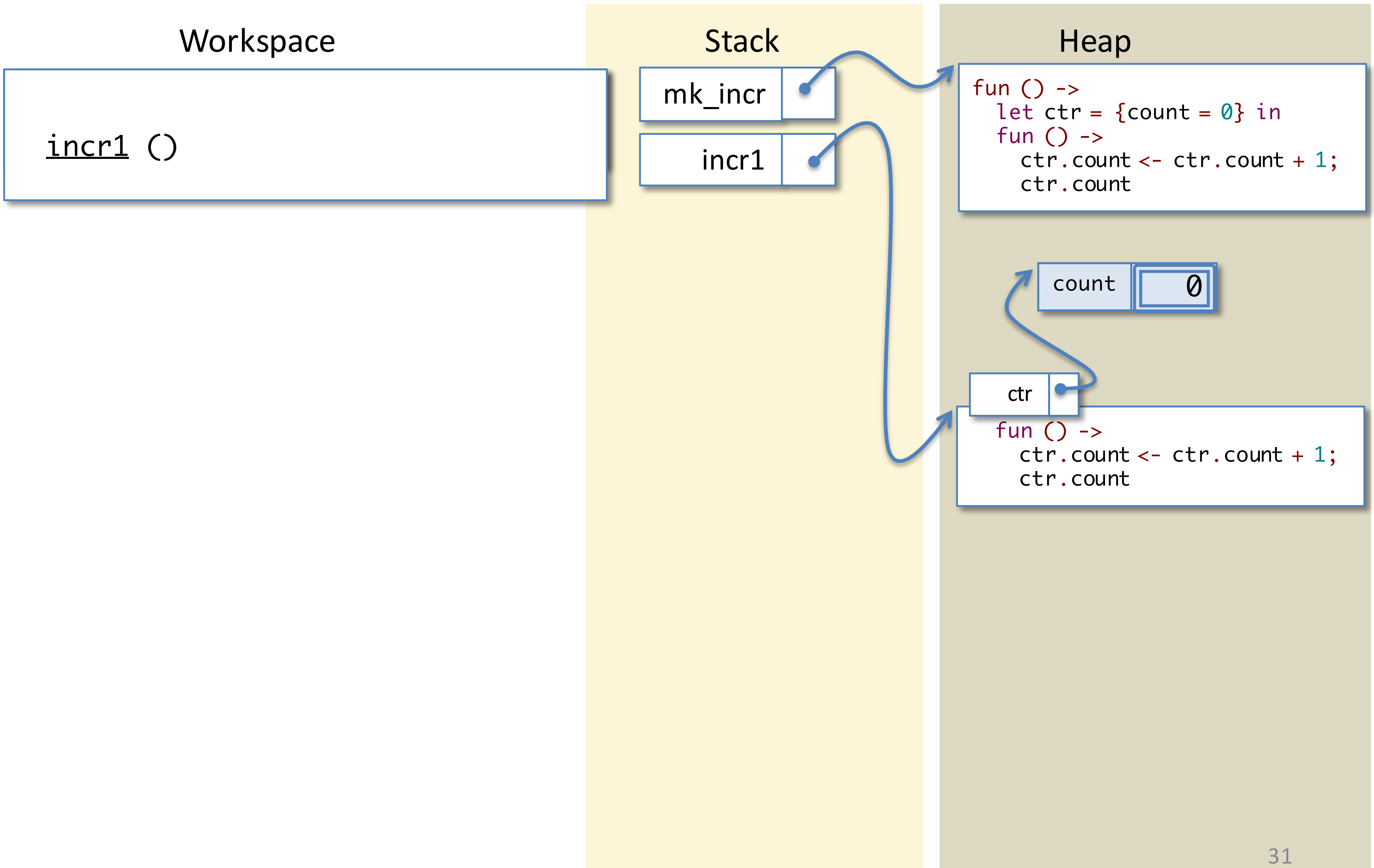
Local Functions



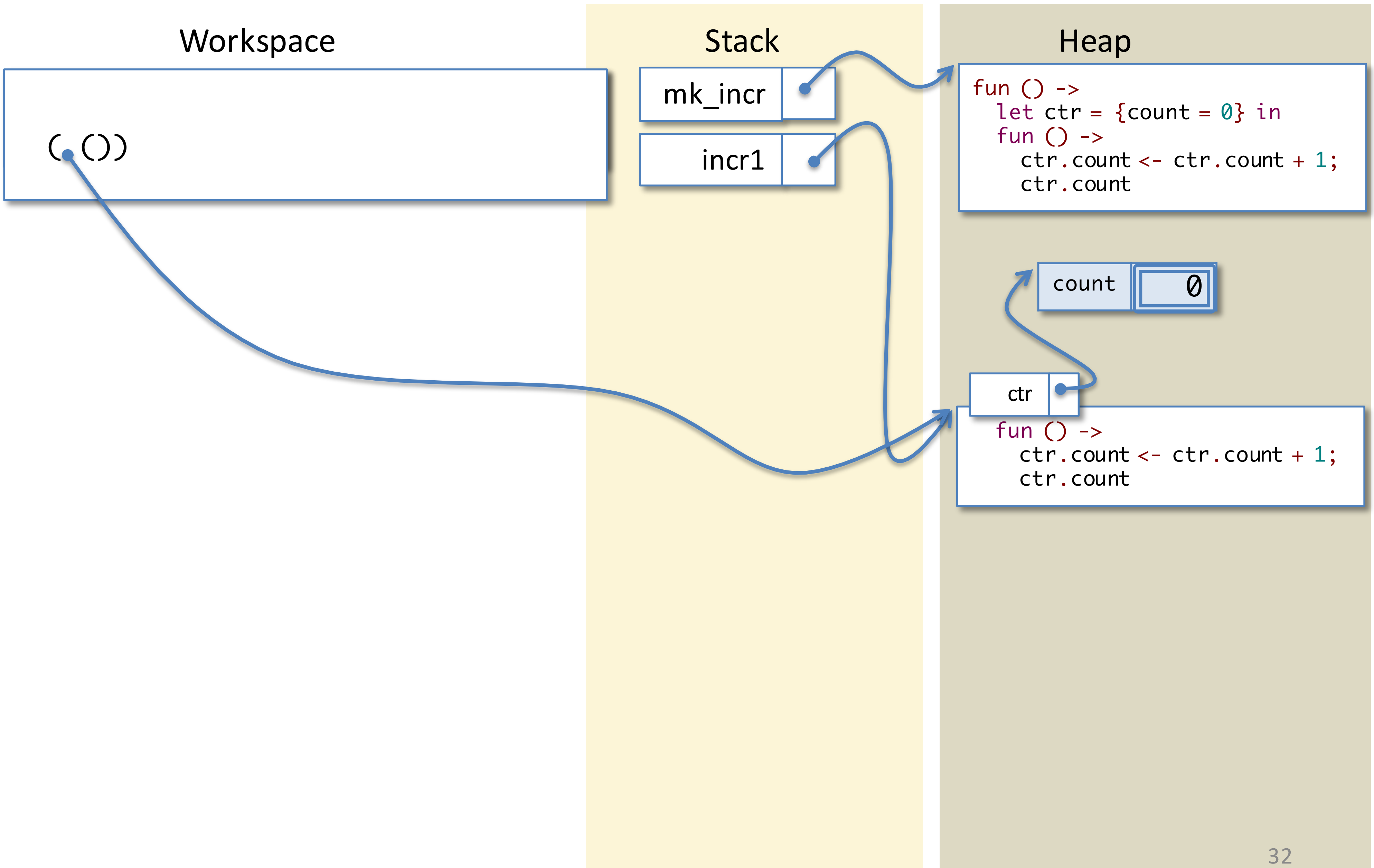
Now let's run "incr1 ()"



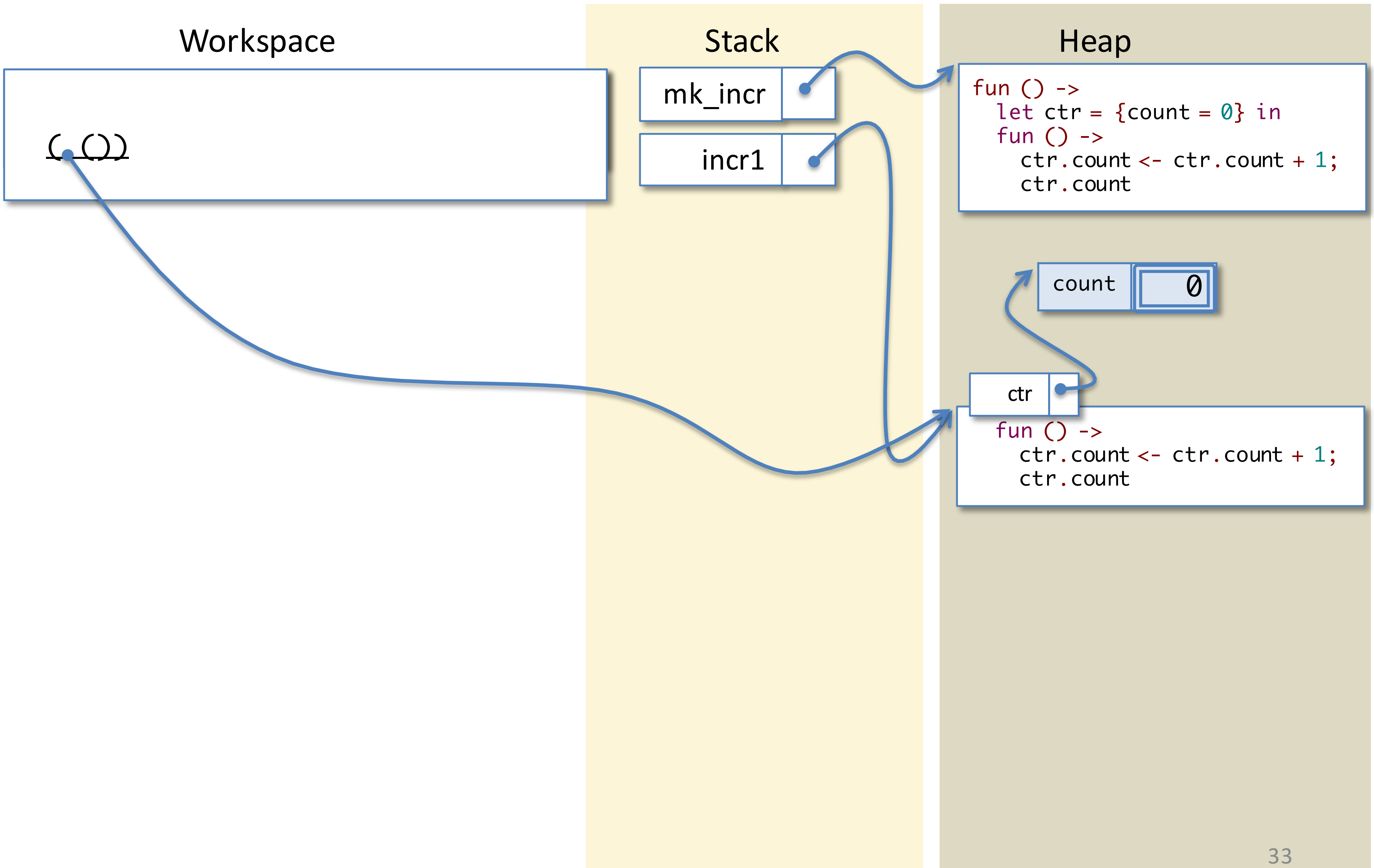
Now let’s run “incr1 ()”



Now let's run "incr1 ()"



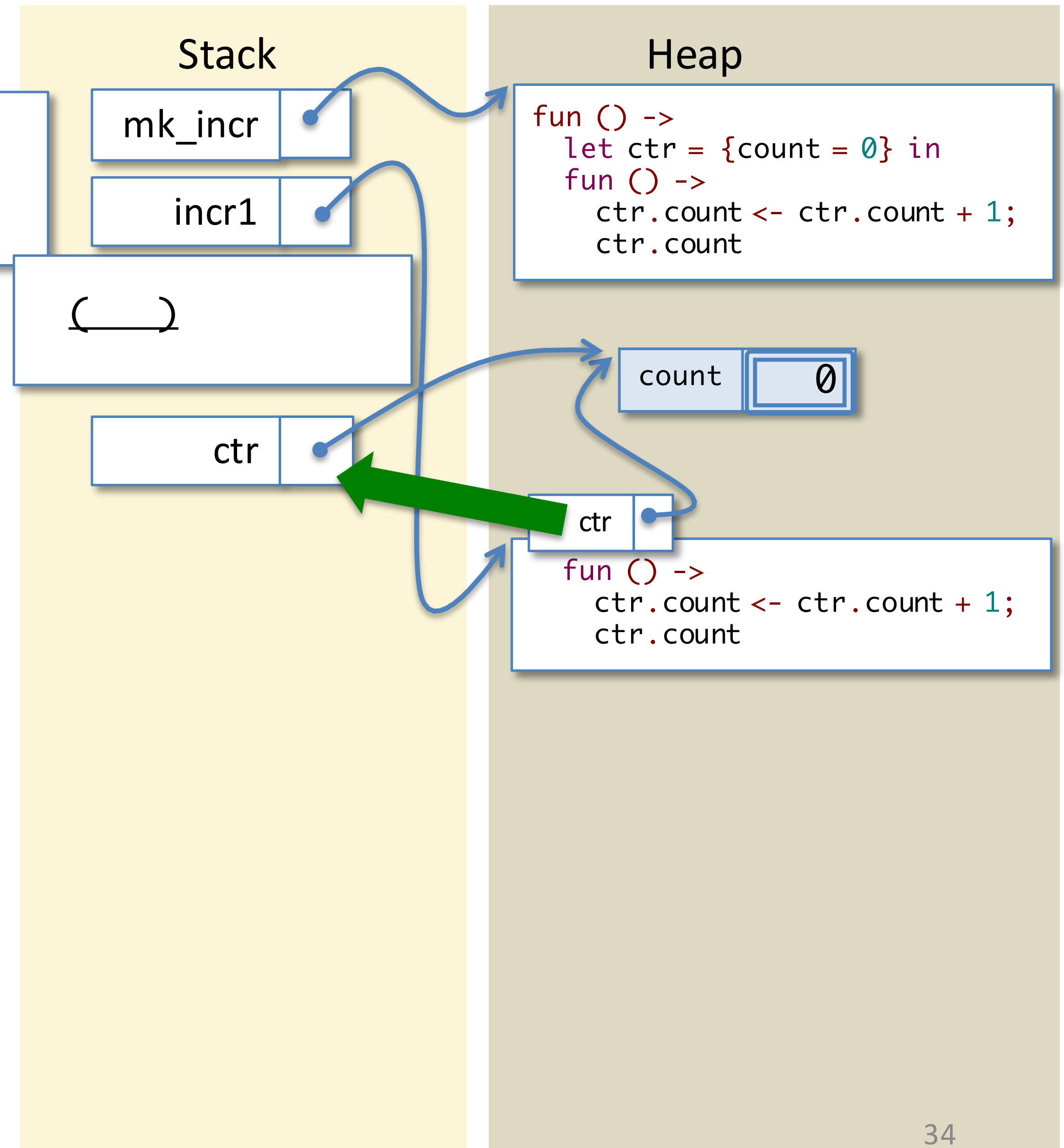
Now let's run "`incr1 ()`"



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

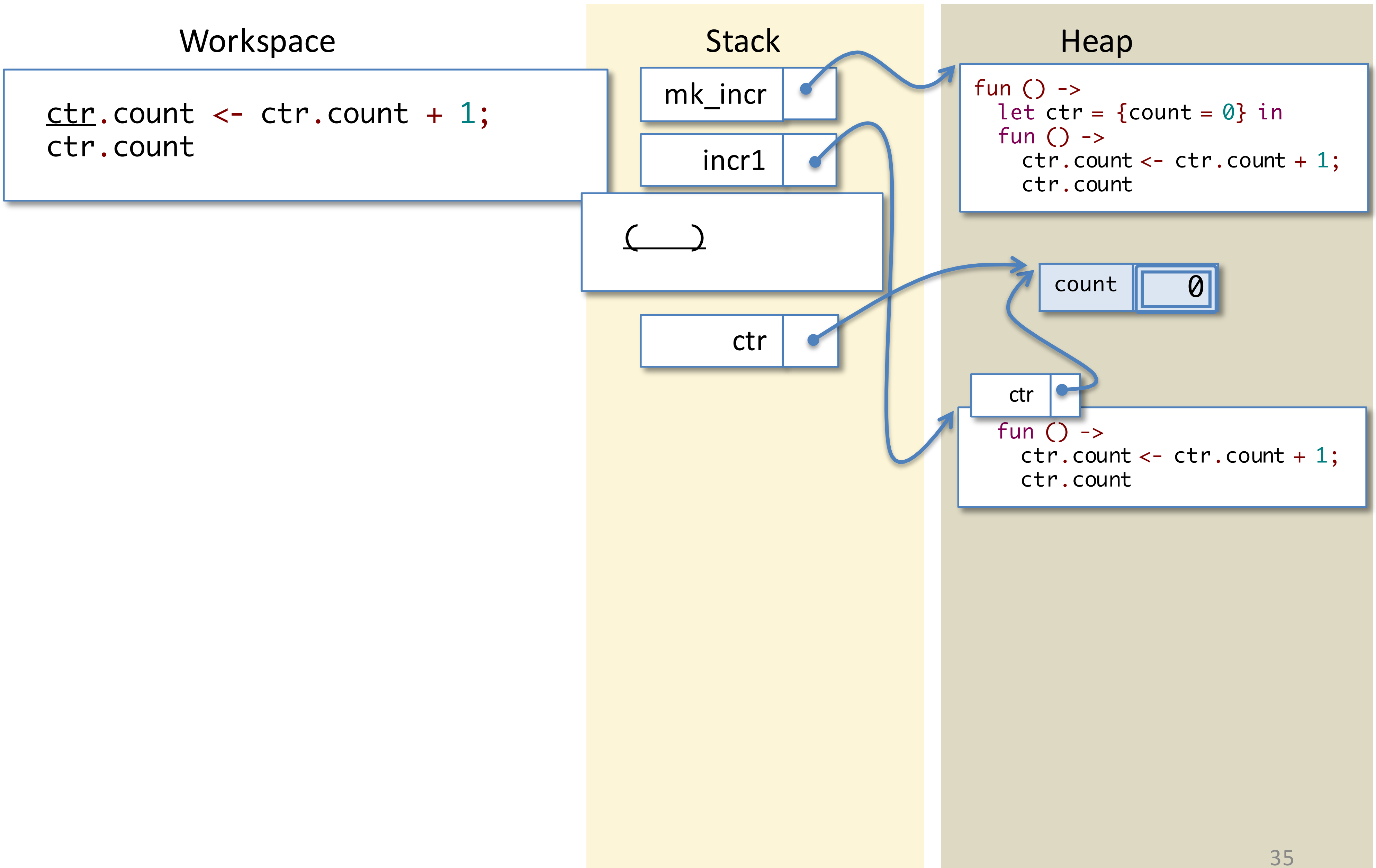
Workspace

```
ctr.count <- ctr.count + 1;
ctr.count
```

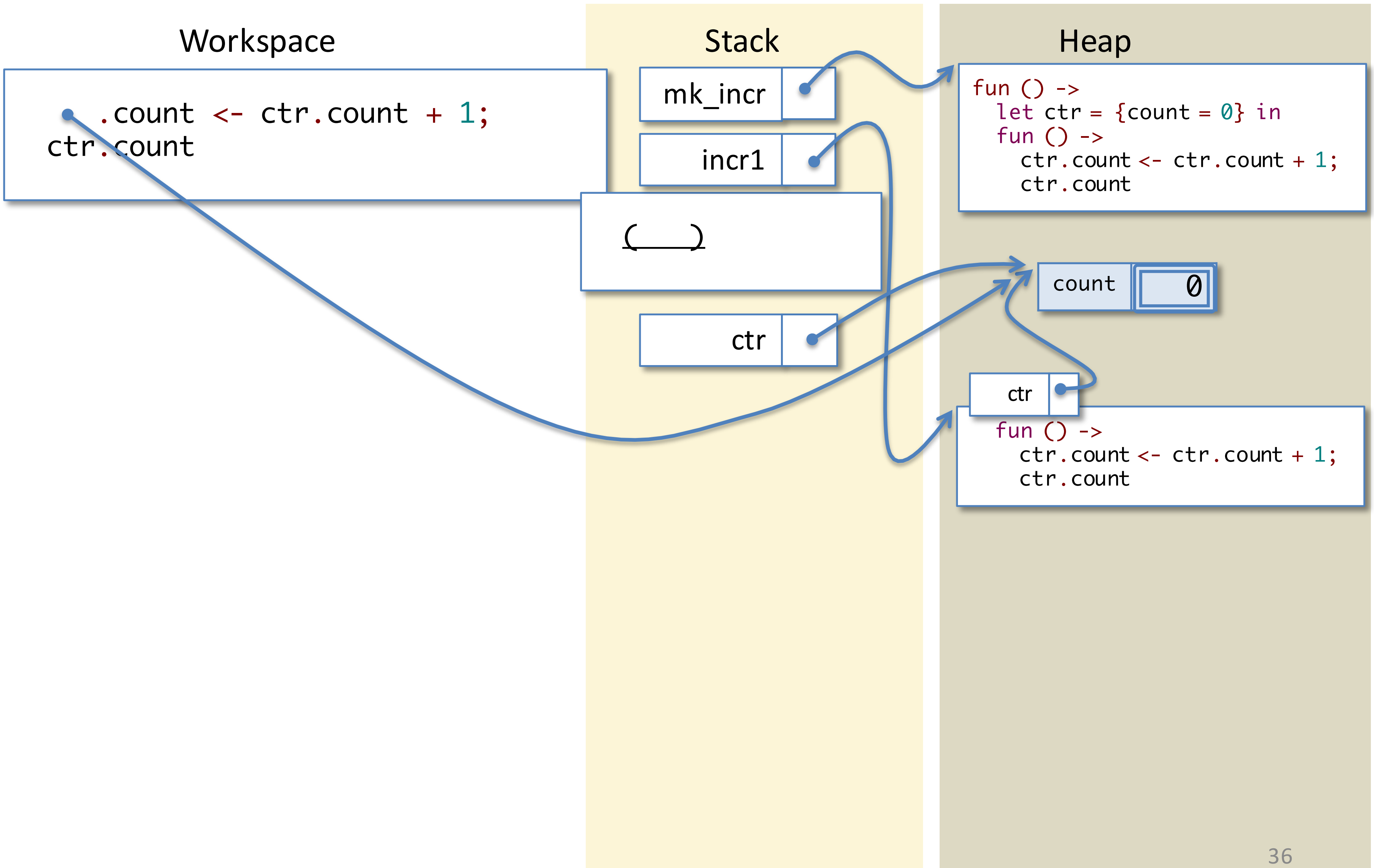


NOTE: Since the function had some saved stack bindings, we add them to the stack at the same time that we put the code in the workspace.

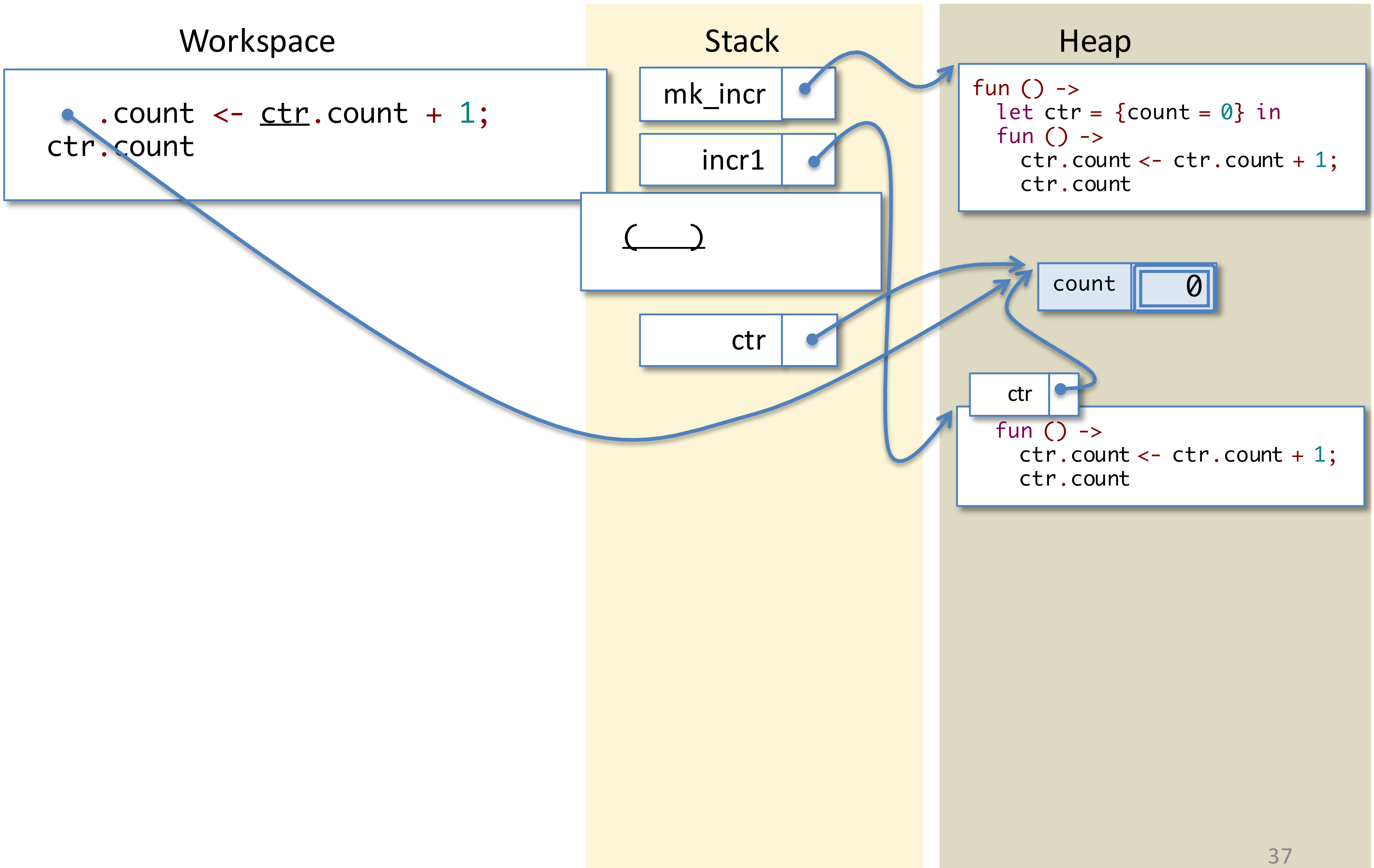
Now let's run "`incr1 ()`"



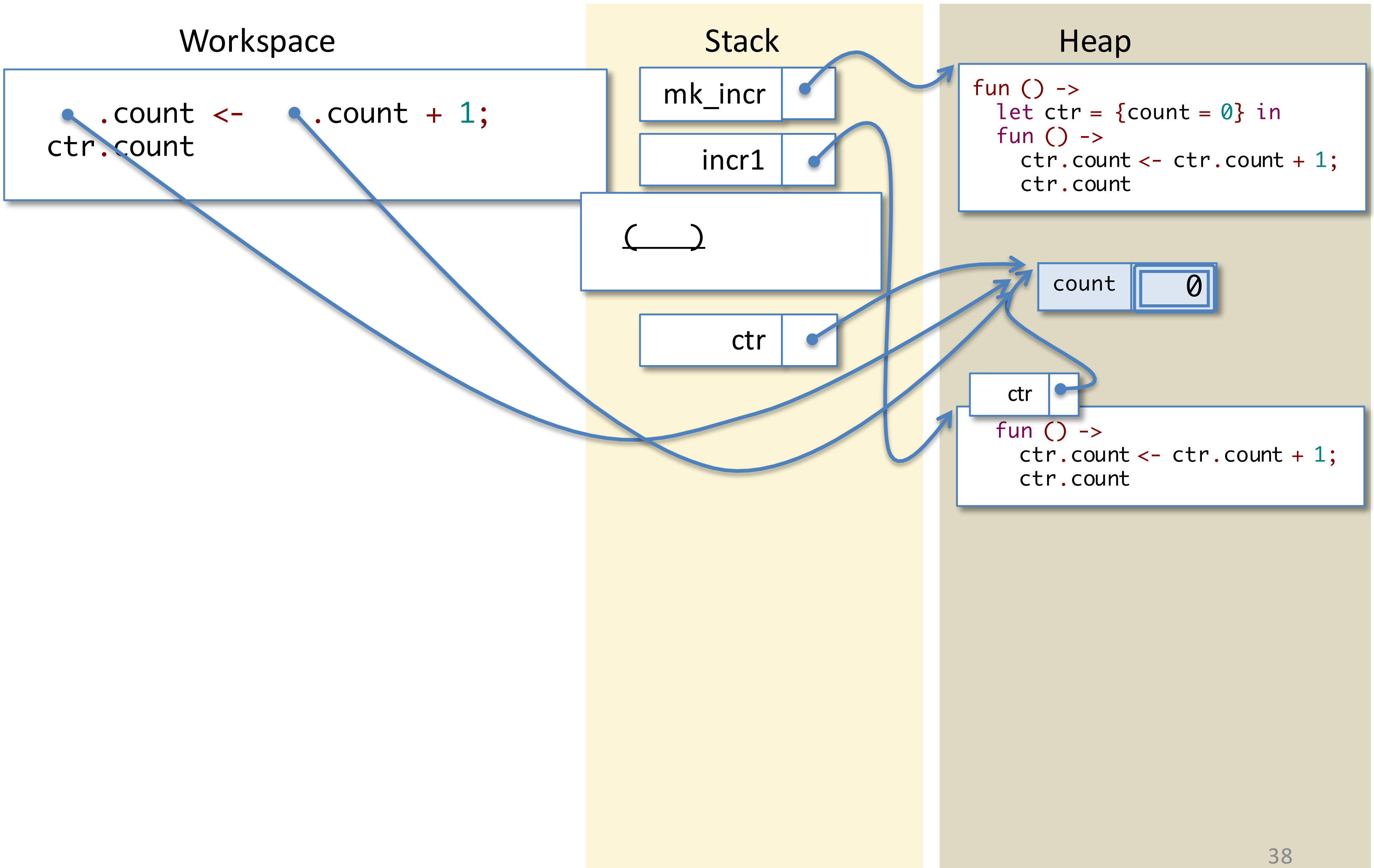
Now let's run "`incr1 ()`"



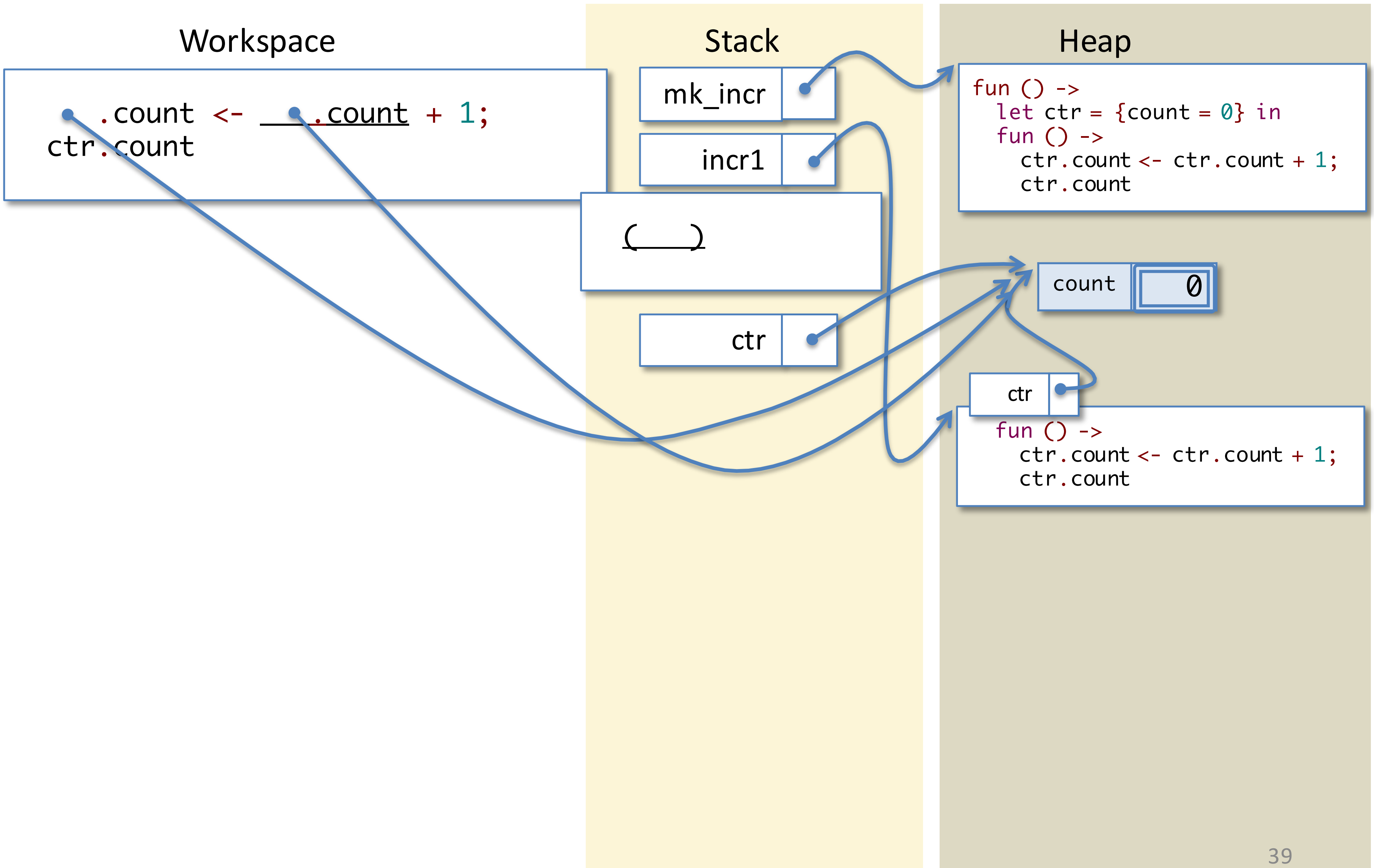
Now let's run "incr1 ()"



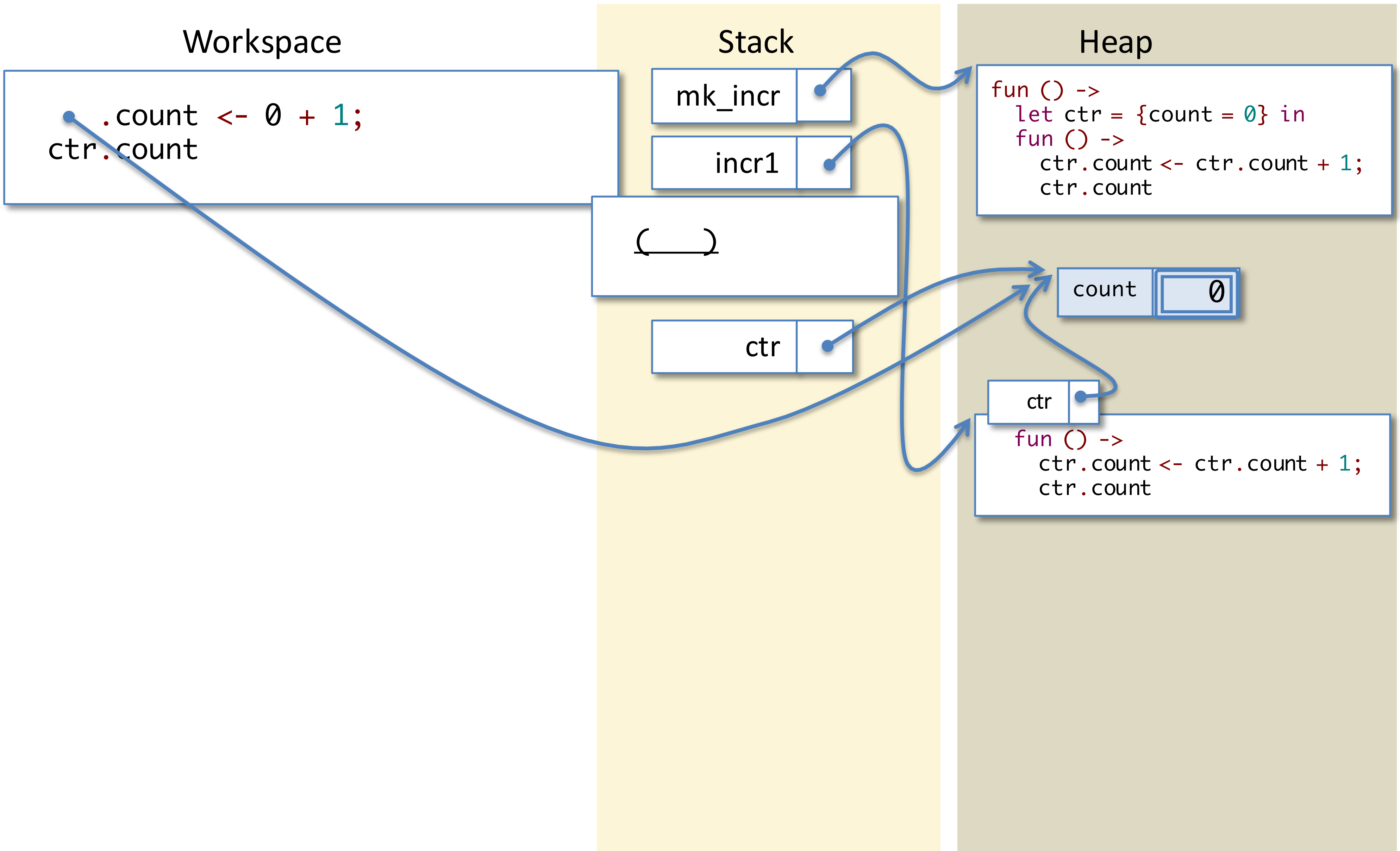
Now let's run "incr1 ()"



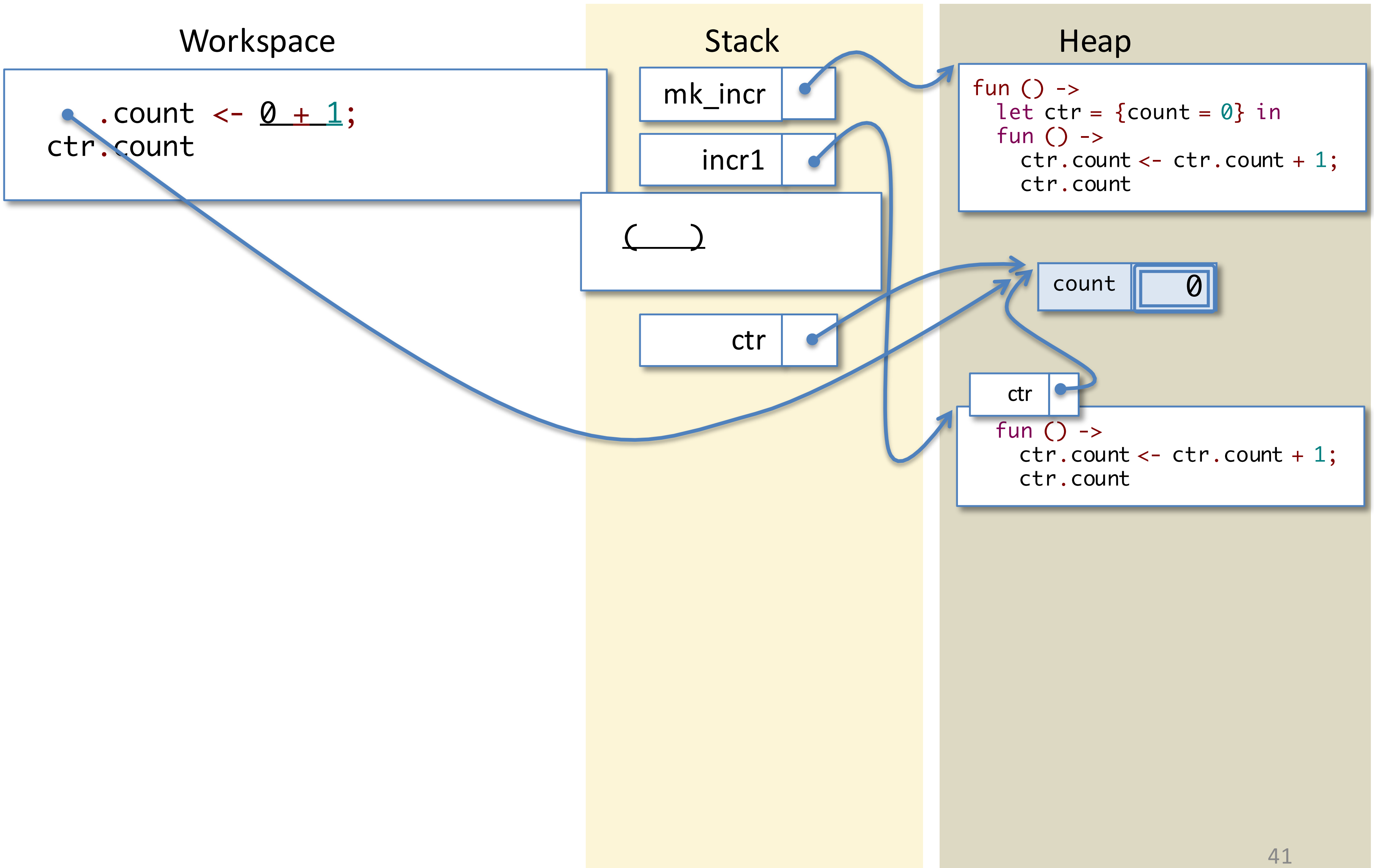
Now let's run "`incr1 ()`"



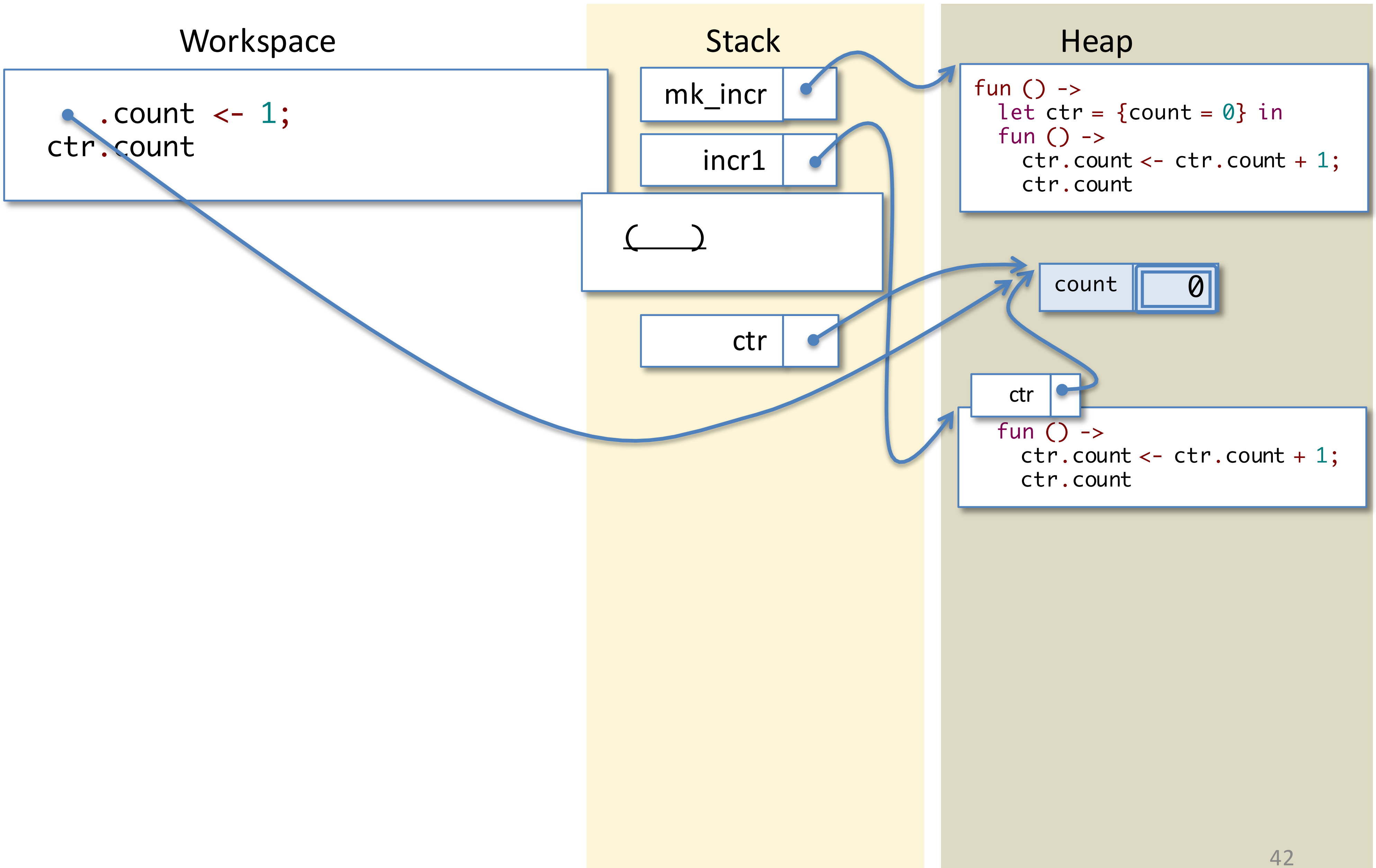
Now let's run "incr1 ()"



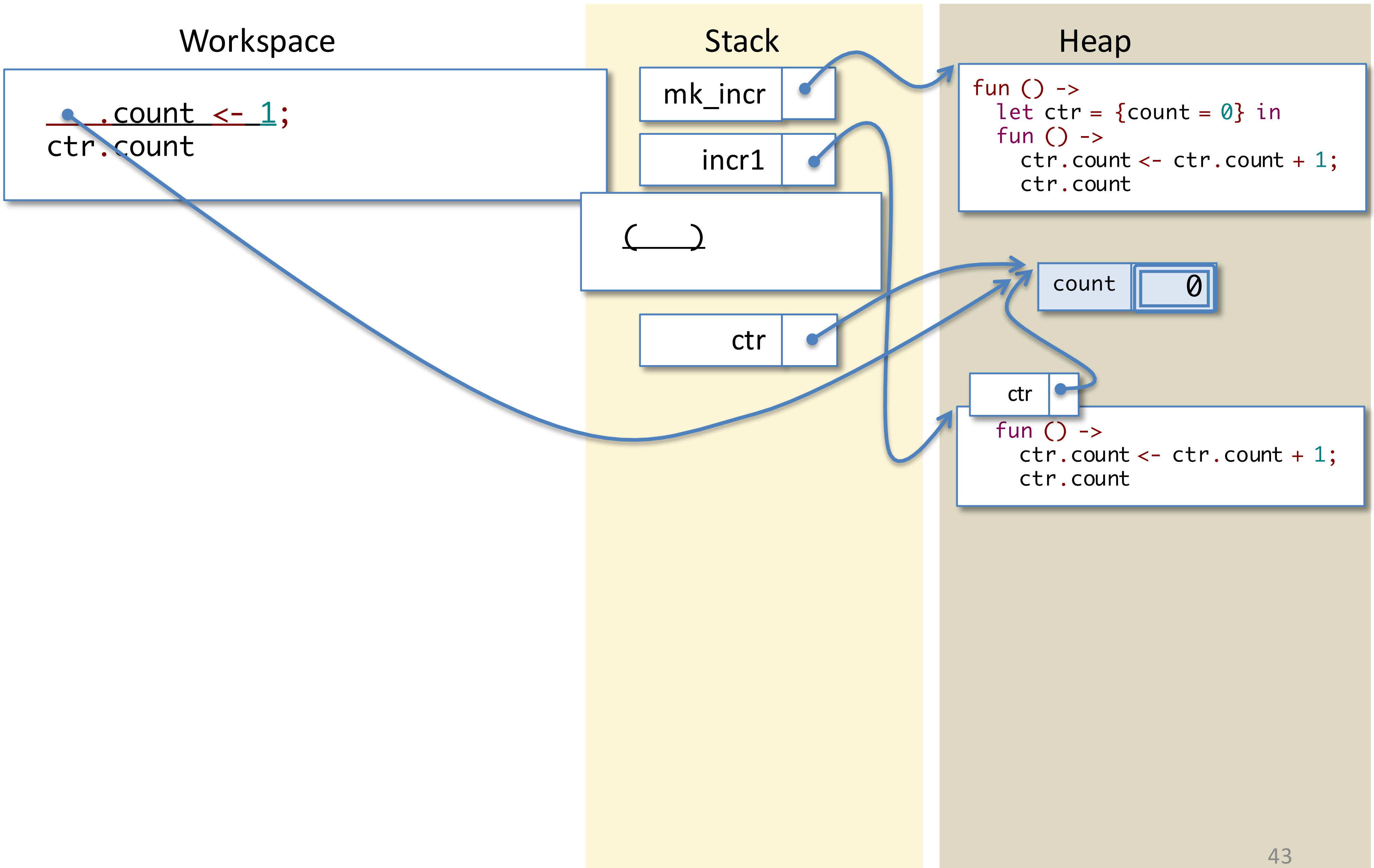
Now let's run "`incr1 ()`"



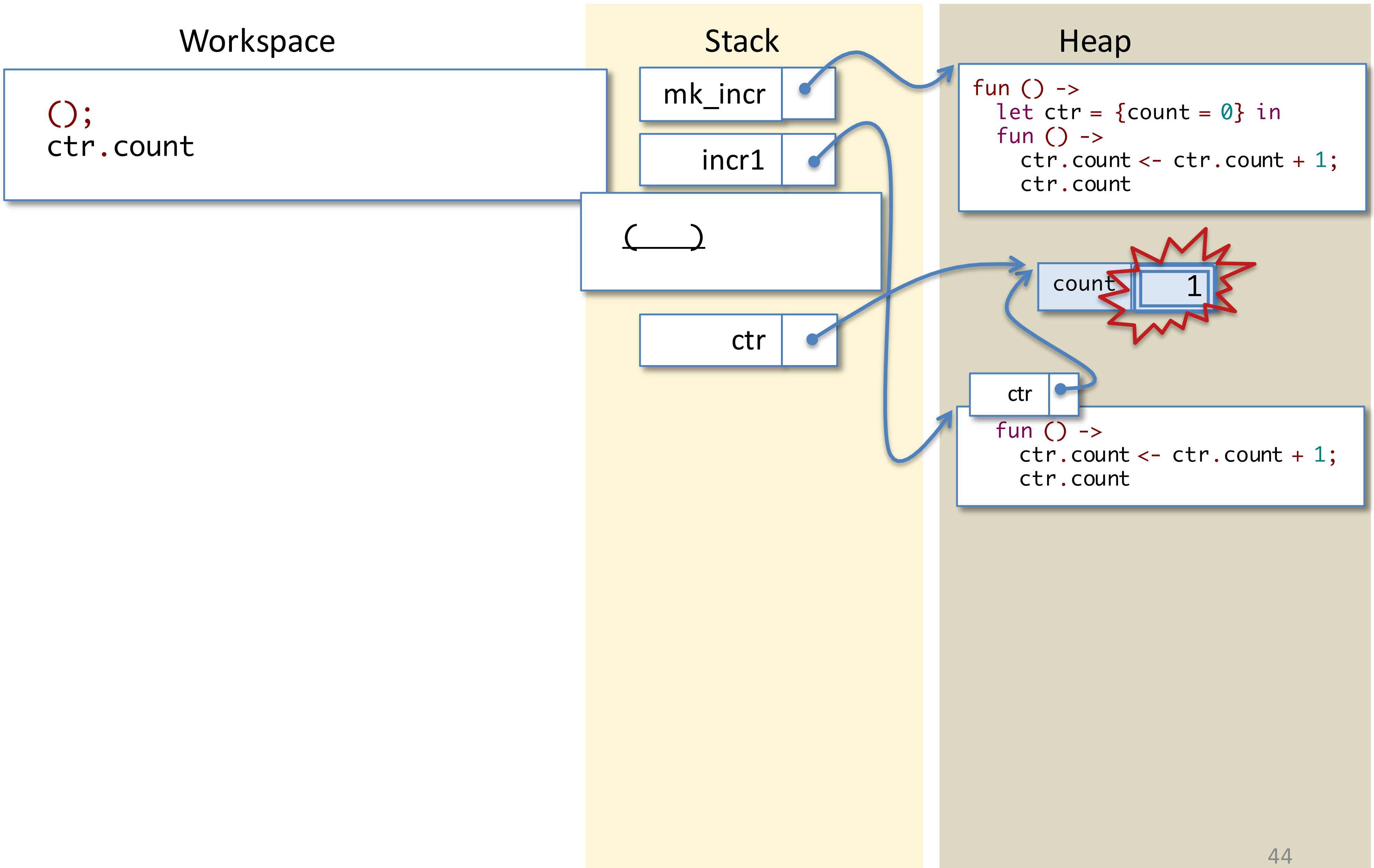
Now let’s run “`incr1 ()`”



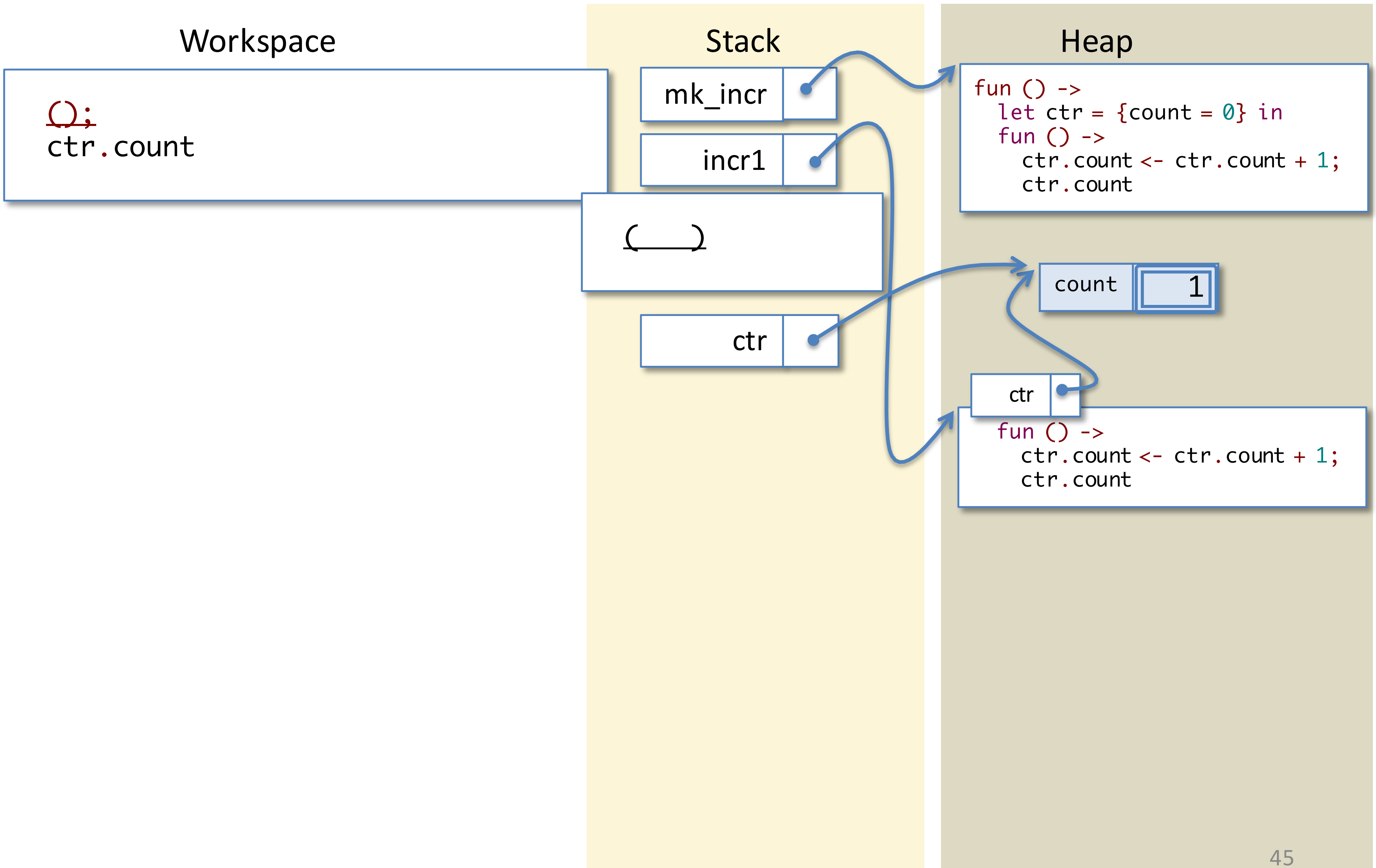
Now let's run "incr1 ()"



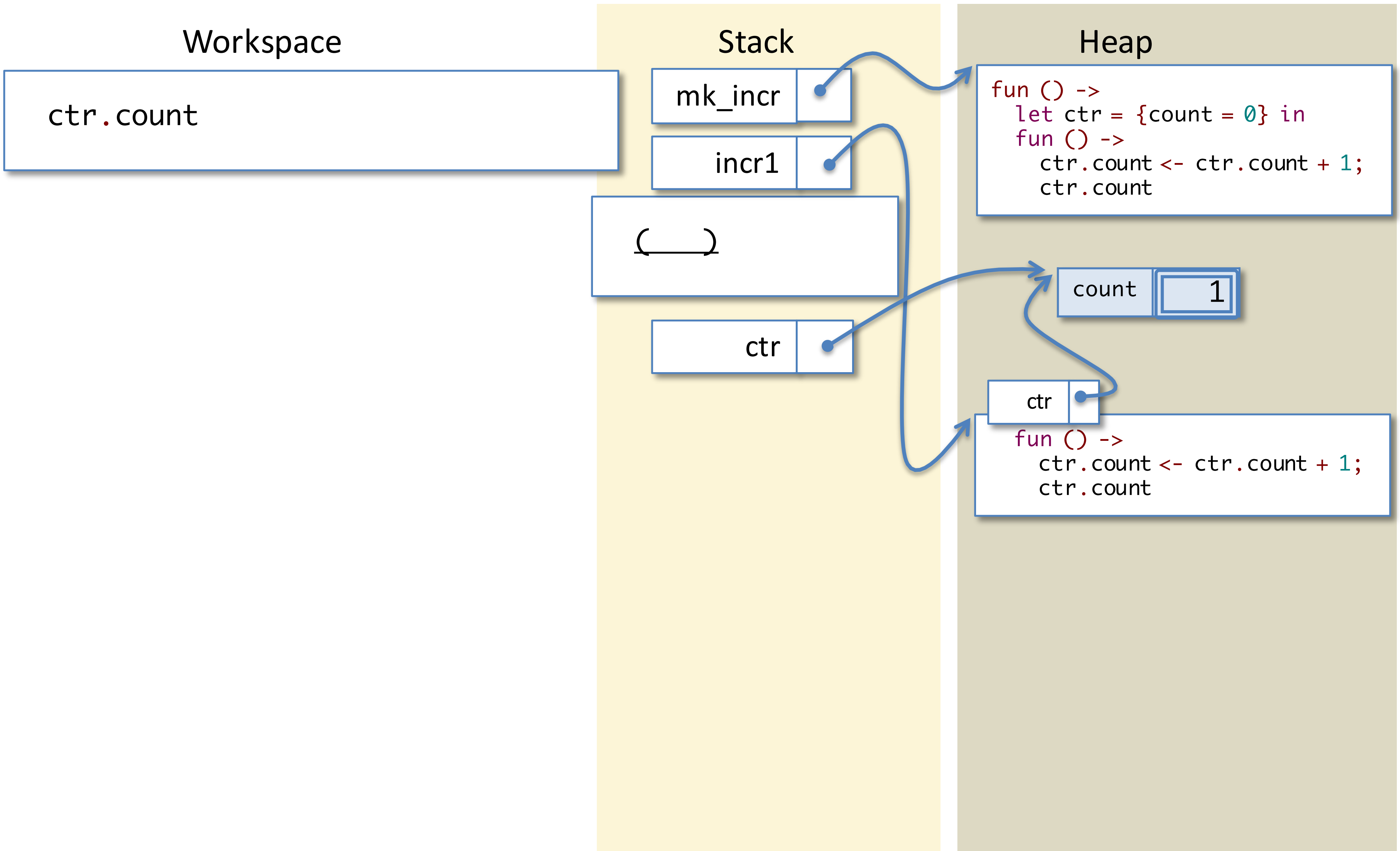
Now let's run "incr1 ()"



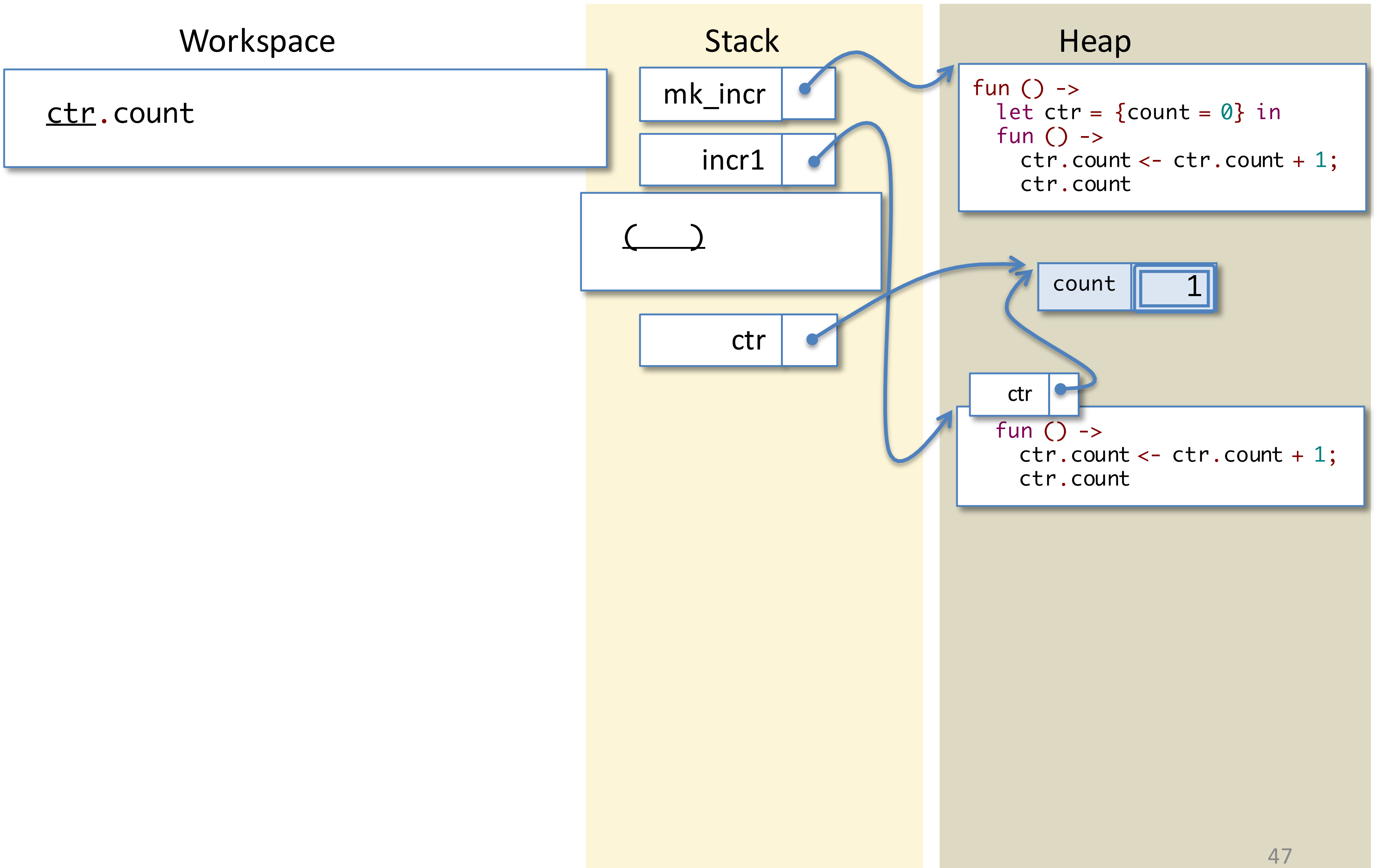
Now let’s run “incr1 ()”



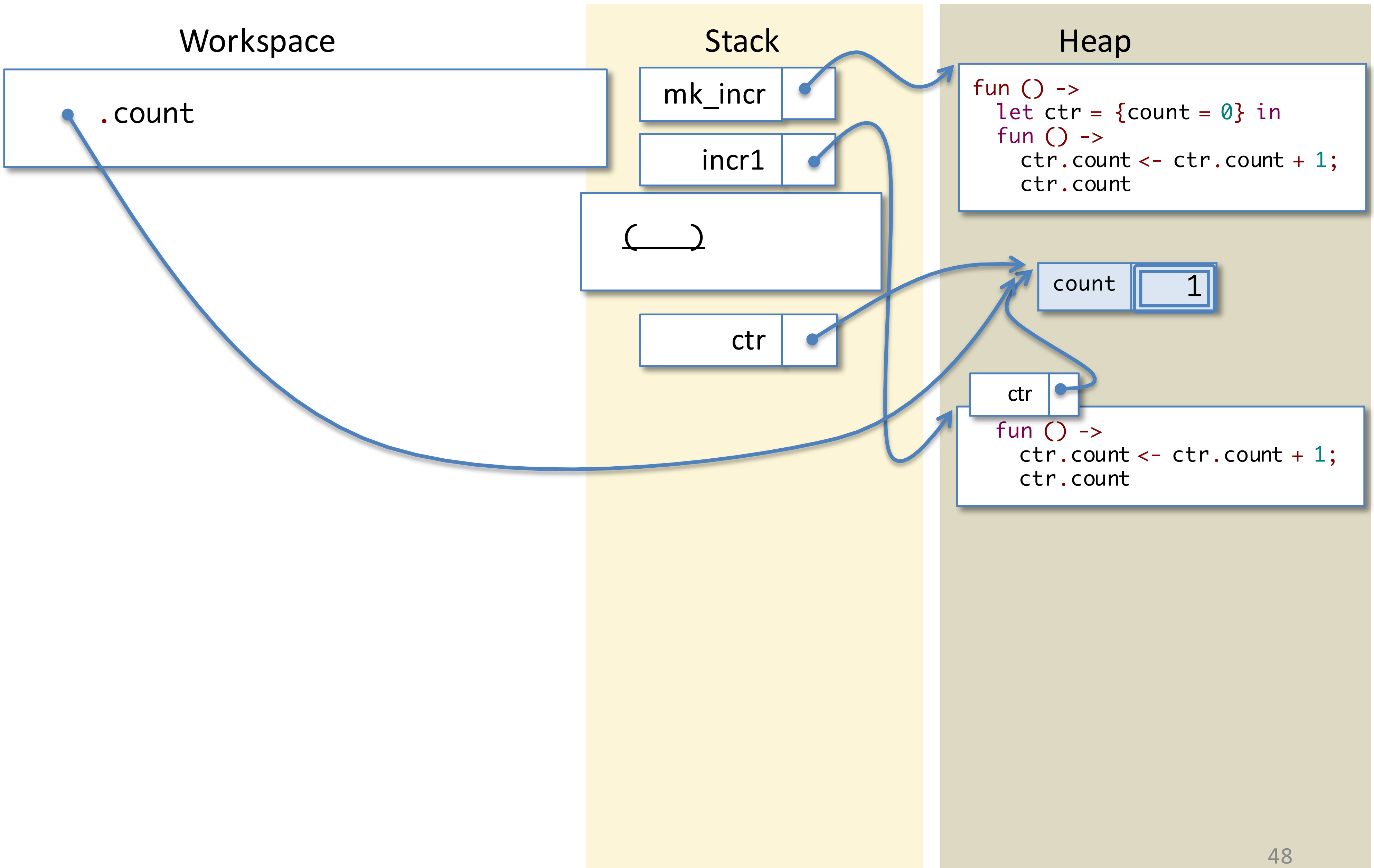
Now let's run "incr1 ()"



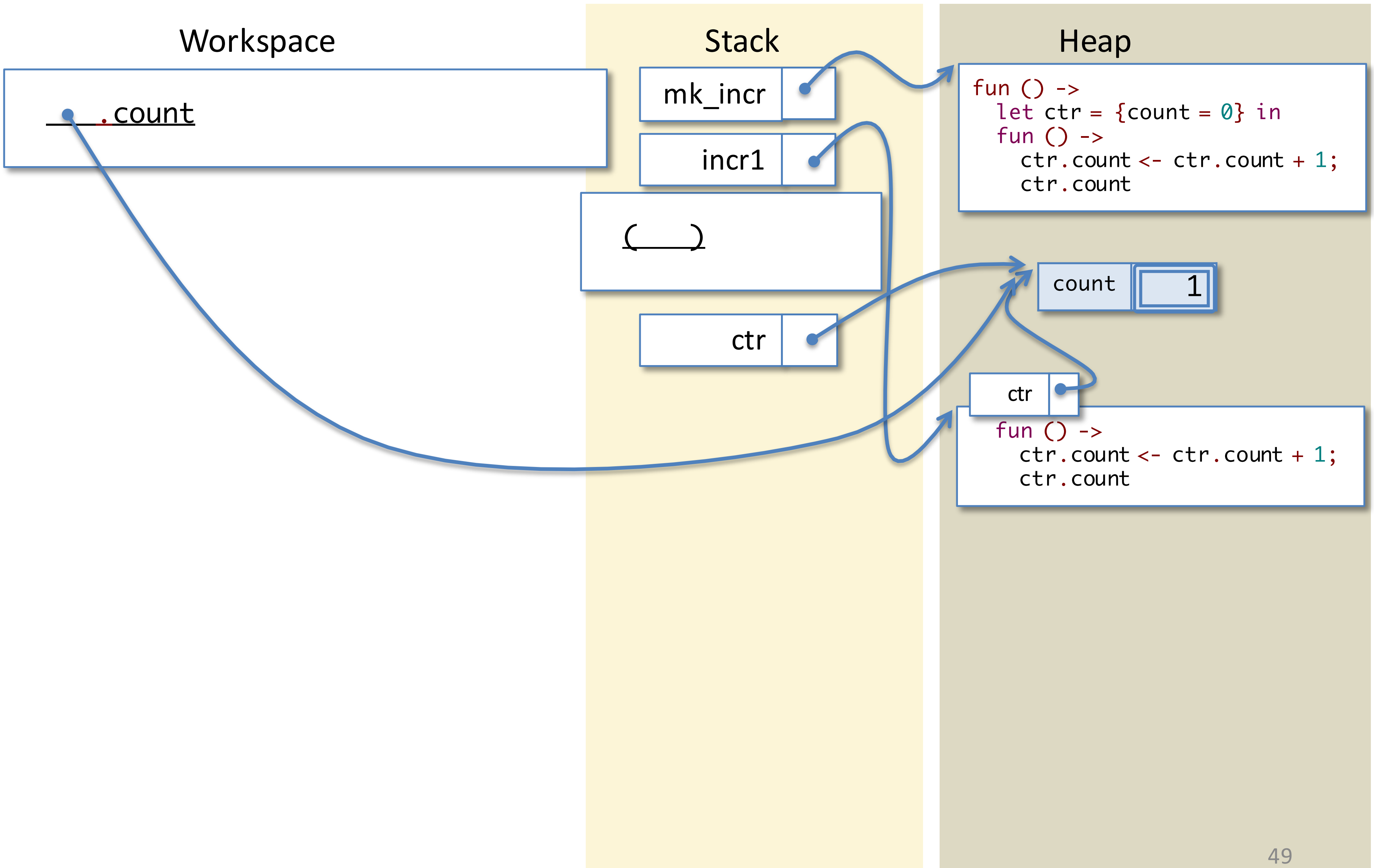
Now let’s run “incr1 ()”



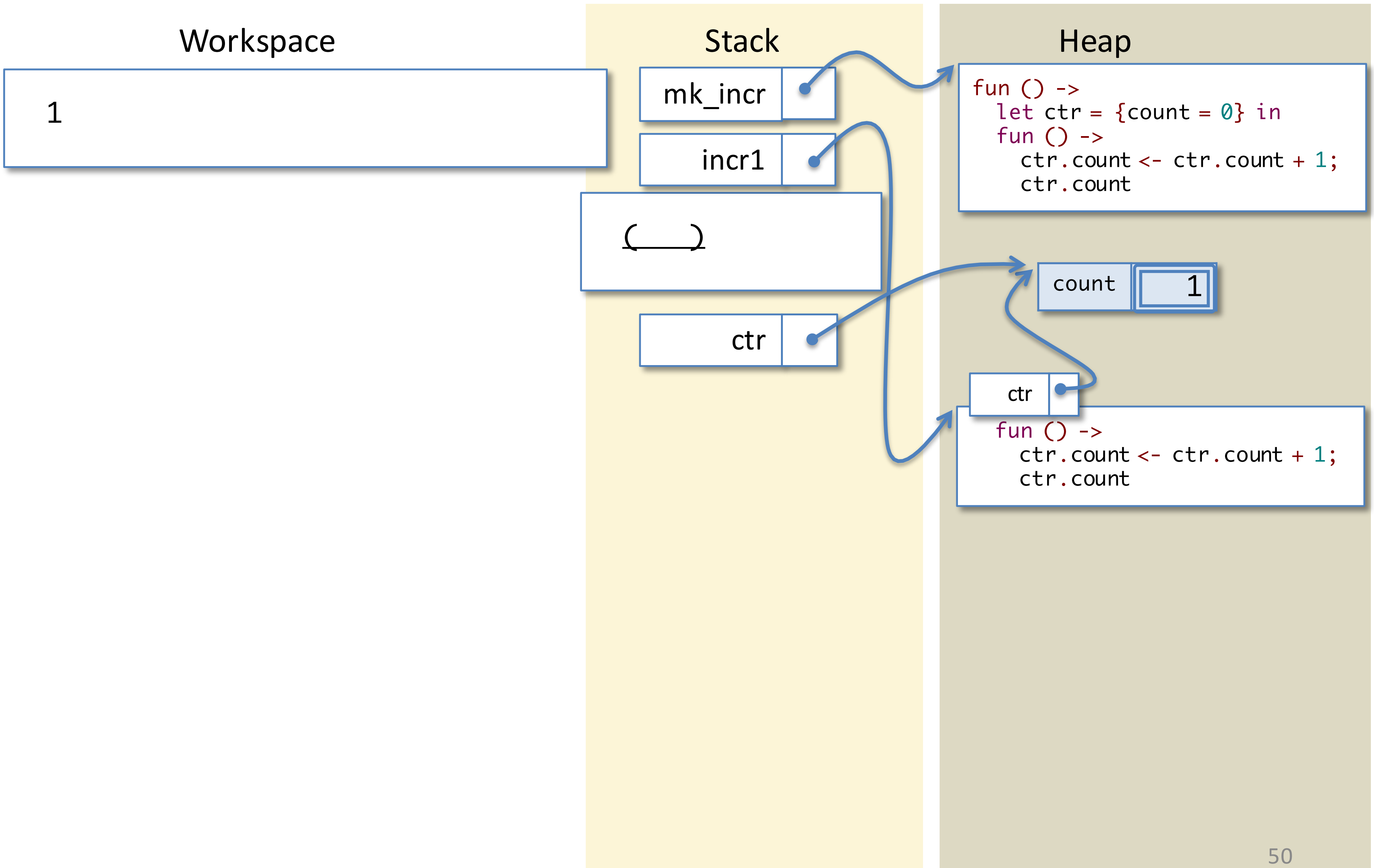
Now let's run "incr1 ()"



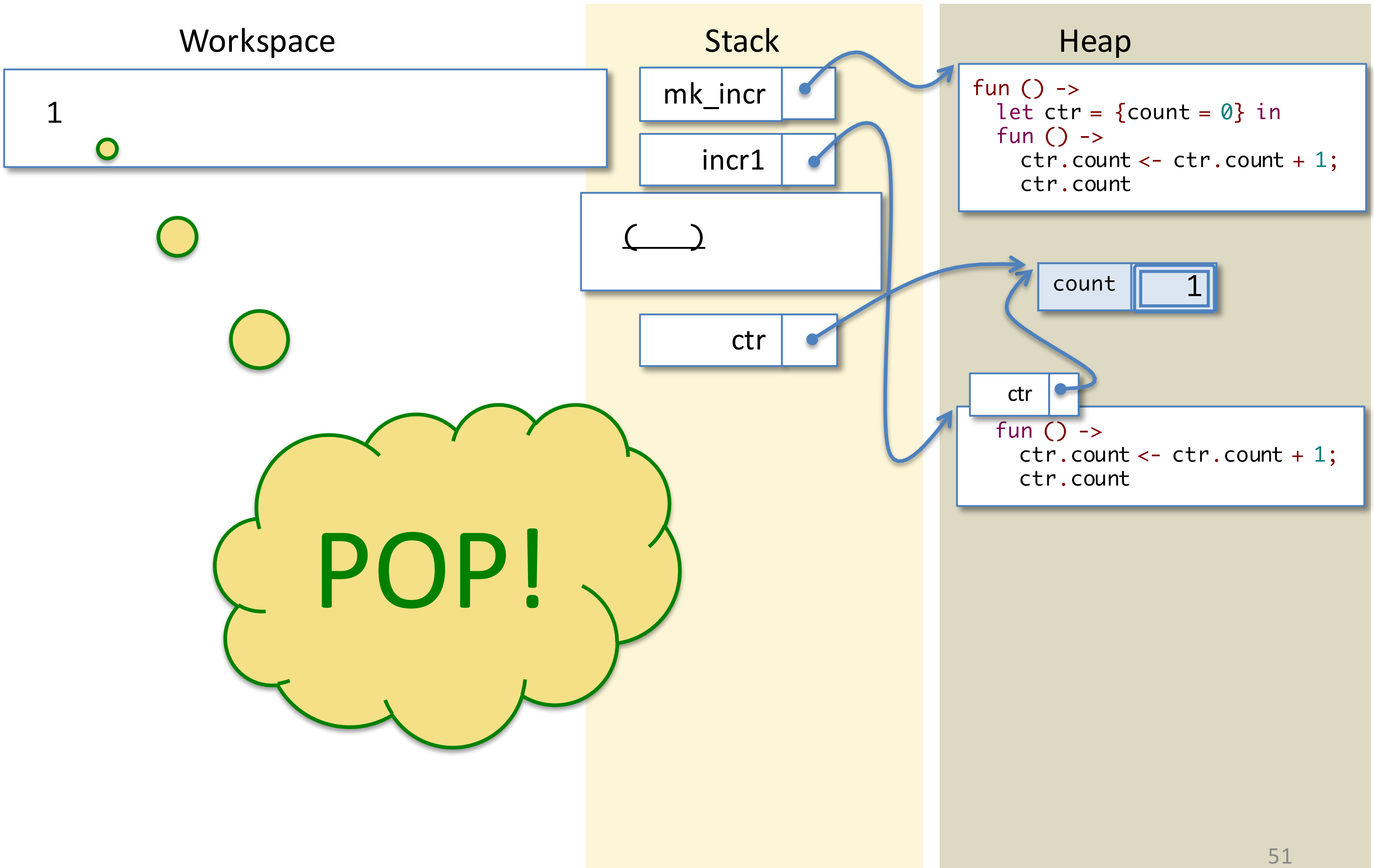
Now let's run "`incr1 ()`"



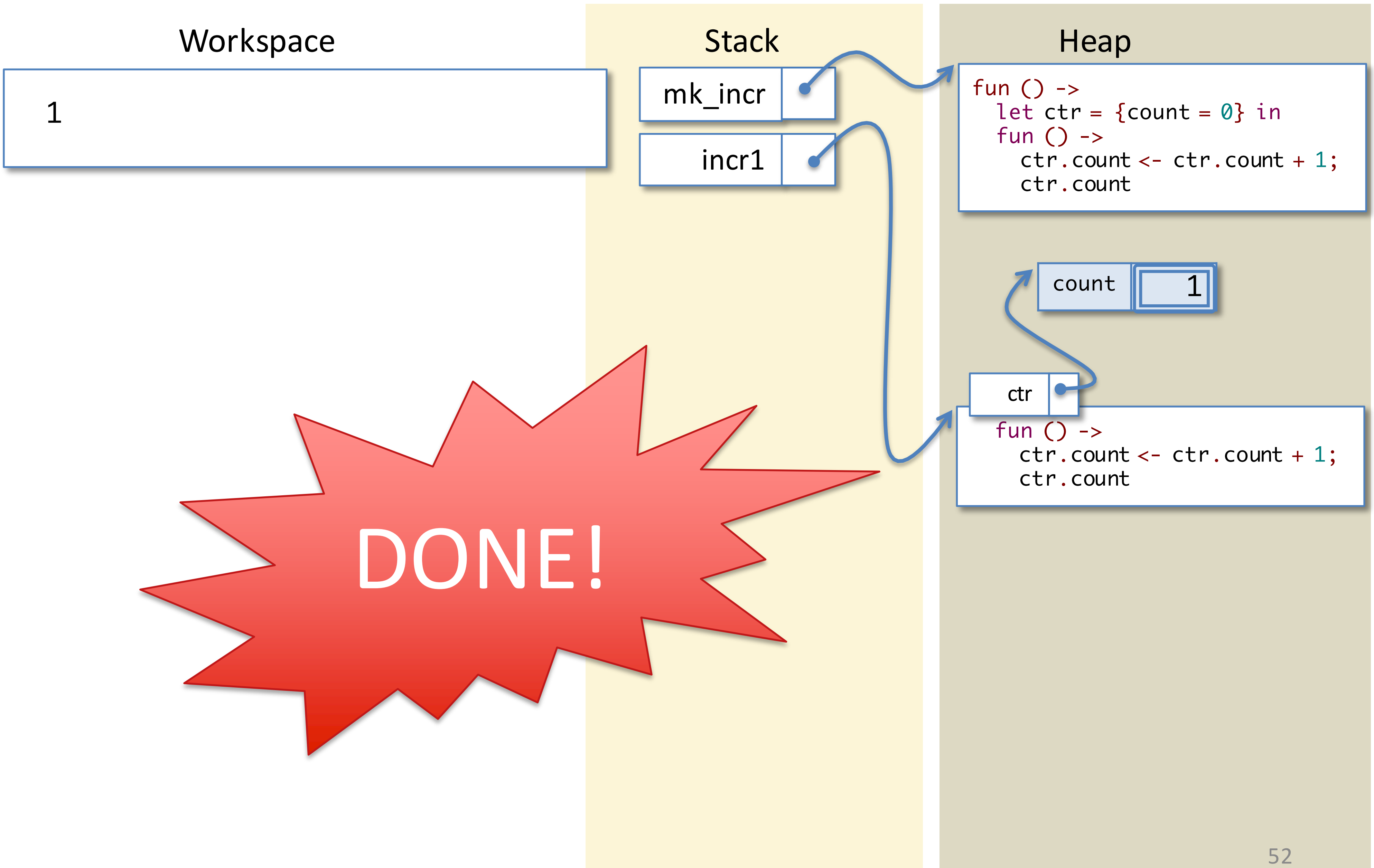
Now let's run "`incr1 ()`"



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



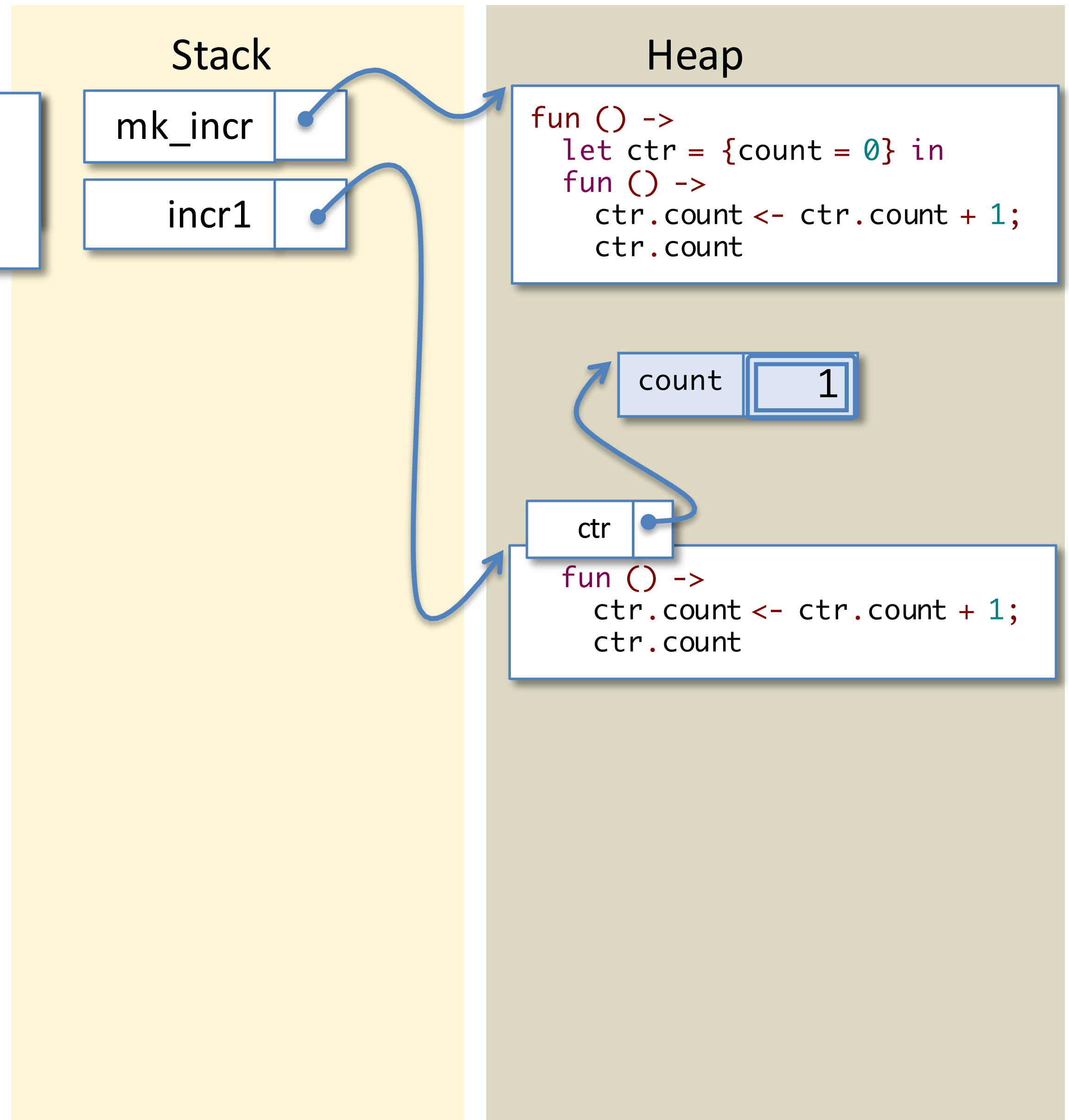
Now let's run "incr1 ()"



Now Let’s run `mk_incr` again

Workspace

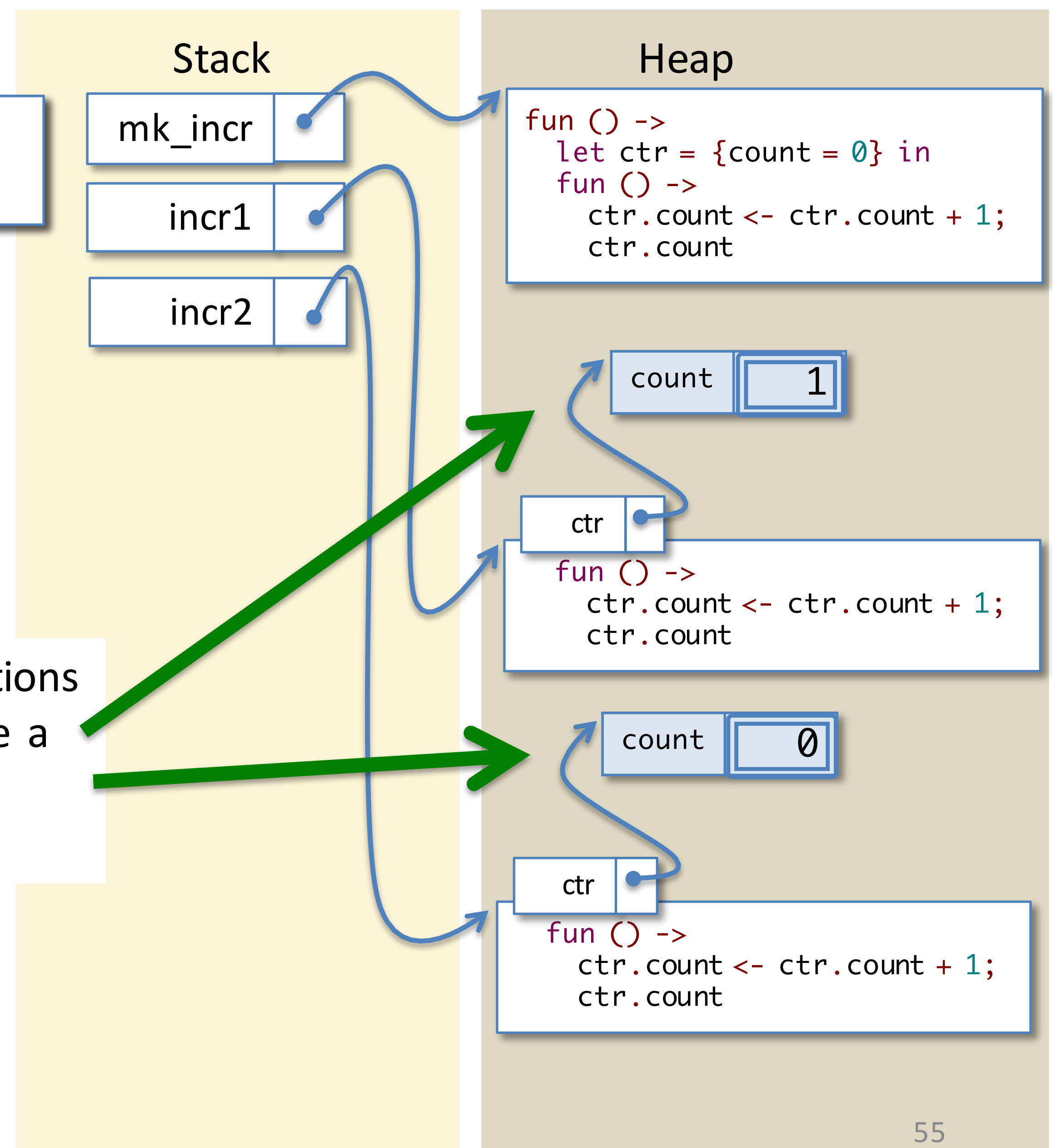
```
let incr2 : unit -> int =
mk_incr ()
```



…time passes…

After creating incr2

Workspace



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

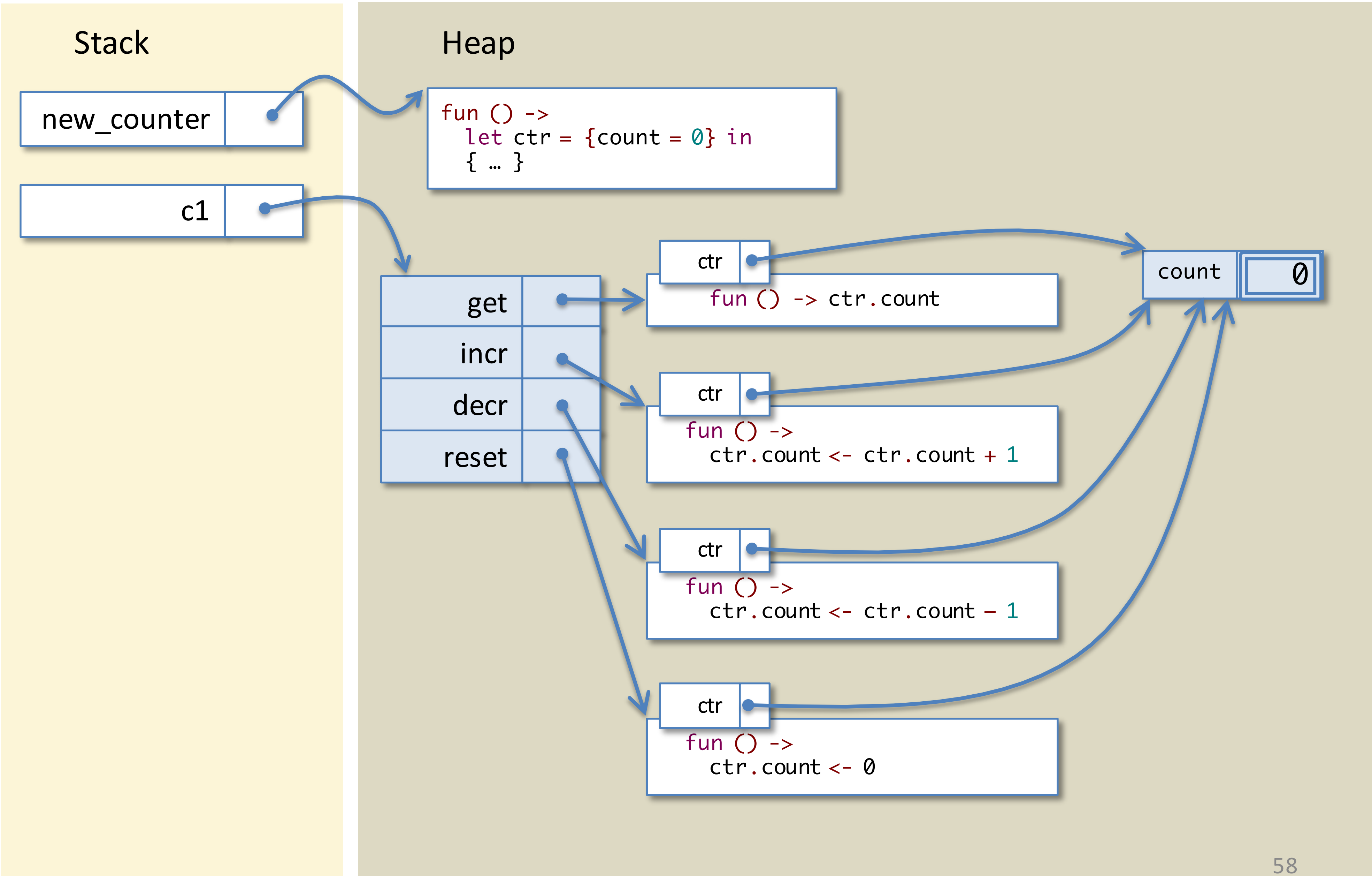
- mk_incr shows us how to create different instance of local state so that we can have several different counters.
- What if we want to bundle together *several* operations that share the same local state?
  - e.g. incr and decr operations that work on the same counter

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 = 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 ()))
```

GUI Design

putting objects to work

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

1. Yes
2. No

Step #1: Understand the Problem

- We don’t want to build just one graphical application: we want to make sure that our code is *reusable.*
- What are the concepts involved in GUI libraries and how do they relate to each other?
- How can we separate the various concerns on the project?

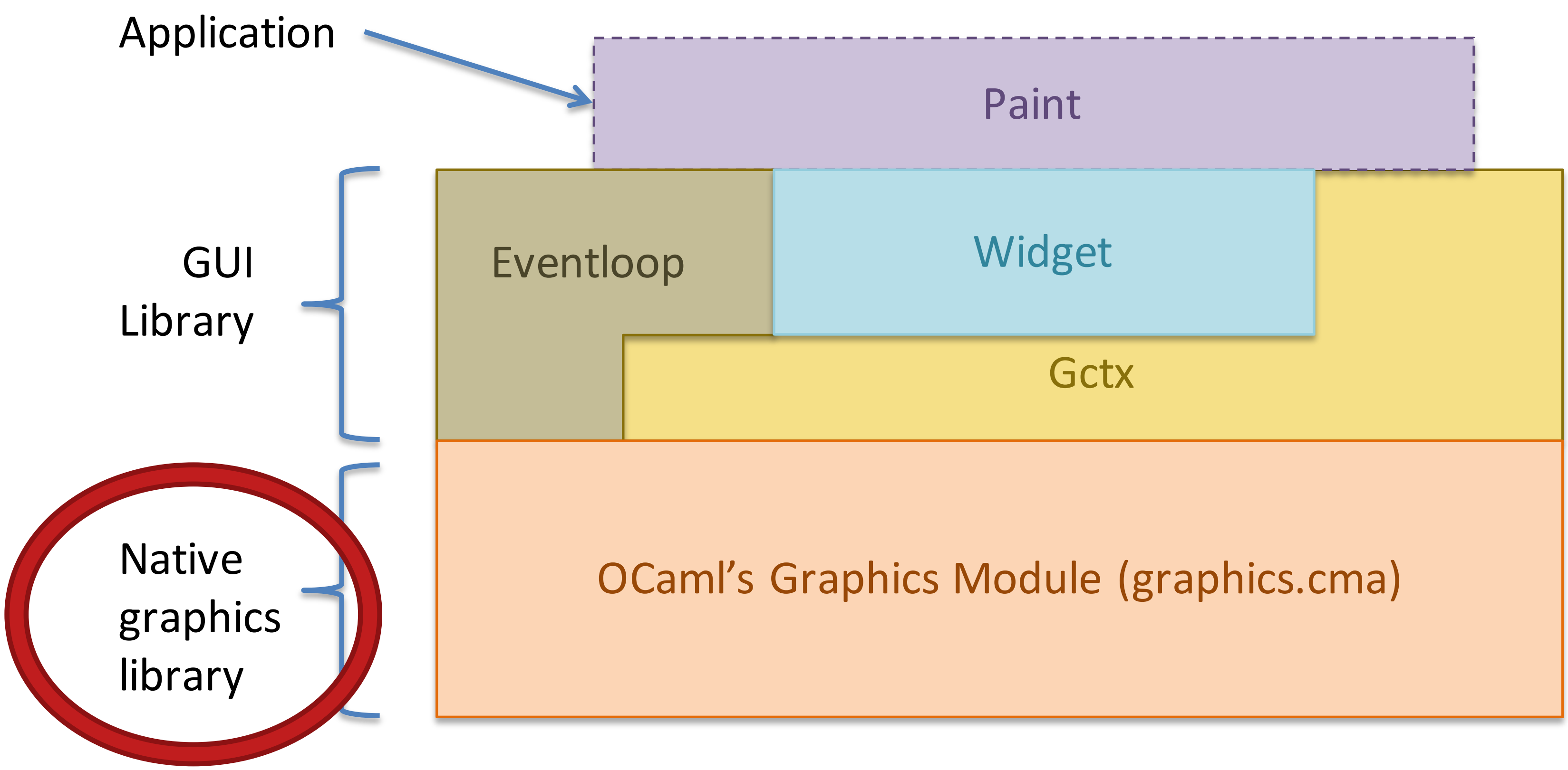
Designing a GUI library – Starting point

- OCaml's Graphics library provides very *simple* primitives for:
  - Creating a window
  - Drawing various shapes: points, lines, text, rectangles, circles, etc.
  - Getting the mouse position, whether the mouse button is pressed, what key is pressed, etc.
  - See: http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html

- How do we go from that to a functioning, reusable GUI library?

Step 2, Interfaces: Project Architecture*

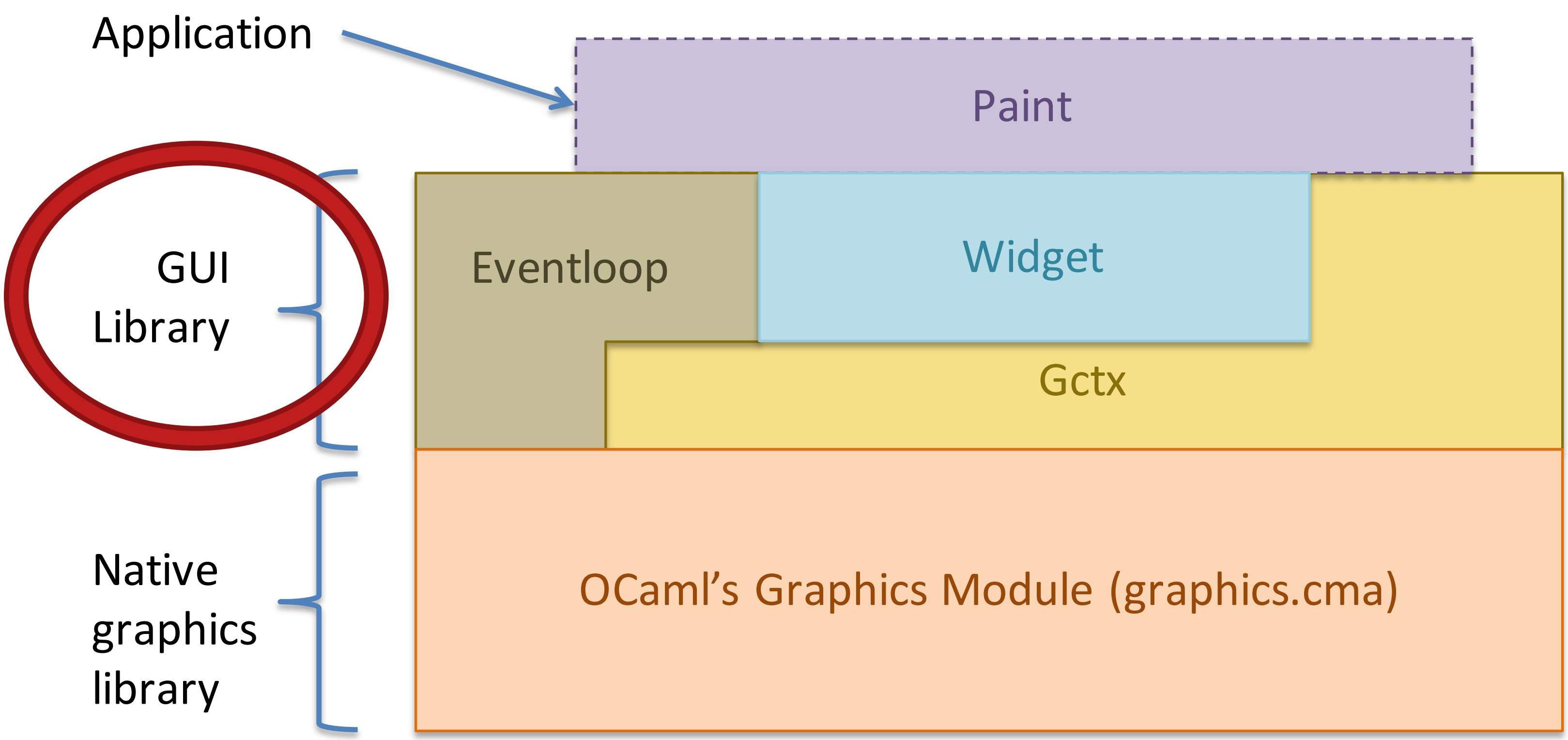
*Note: Subsequent program snippets are color-coded according to this diagram.



Goal of the GUI library: provide a consistent layer of abstraction *between* the application (Paint) and the Graphics module.

Step 2, Interfaces: Project Architecture*

*Note: Subsequent program snippets are color-coded according to this diagram.



Goal of the GUI library: provide a consistent layer of abstraction *between* the application (Paint) and the Graphics module.

GUI terminology – Widget*

- Basic element of GUIs : buttons, checkboxes, windows, textboxes, canvases, scrollbars, labels
- All have a position on the screen and know how to display themselves
- May be composed of other widgets (for layout)

- Widgets are often modeled by *objects*
  - They often have hidden state (string on the button, whether the checkbox is checked)
  - They need functions that can modify that state

*Each GUI library uses its own naming convention for what we call “Widget”. Java’s Swing calls them “Components”; iOS UIKit calls them “UIViews”; WINAPI, GTK+, X11’s widgets, etc.…

GUI terminology - Eventloop

- Main loop of any GUI application

```
let run (w:widget) : unit =
  Graphics.open_graph "";                  (* open a new window *)
  Graphics.auto_synchronize false;

  let rec loop () : unit =
    Graphics.clear_graph ();

    repaint w;

    Graphics.synchronize ();               (* force window update *)

    wait for user input (mouse movement, key press)
    inform w about the input so widgets can react to it;

    loop ()                                (* tail recursion! *)
  in
    loop ()
```

- Takes "top-level" widget w as argument. That widget *contains* all others in the application.