Programming Languages and Techniques (CIS1200)

Lecture 18

GUI Library Design

Chapter 18

Announcements

- HW04 due tomorrow (at 11.59pm)

- HW05 available soon, due *Thursday*, October 24th (at 11.59pm)
  - Start early!
  - Tasks 0-1 can be done after class today
  - Tasks 2-4 can be done after class on Wednesday
  - Tasks 5-6 can be done after class on Friday

- Final Exam
  - Tuesday, December 17th, 12-2pm

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

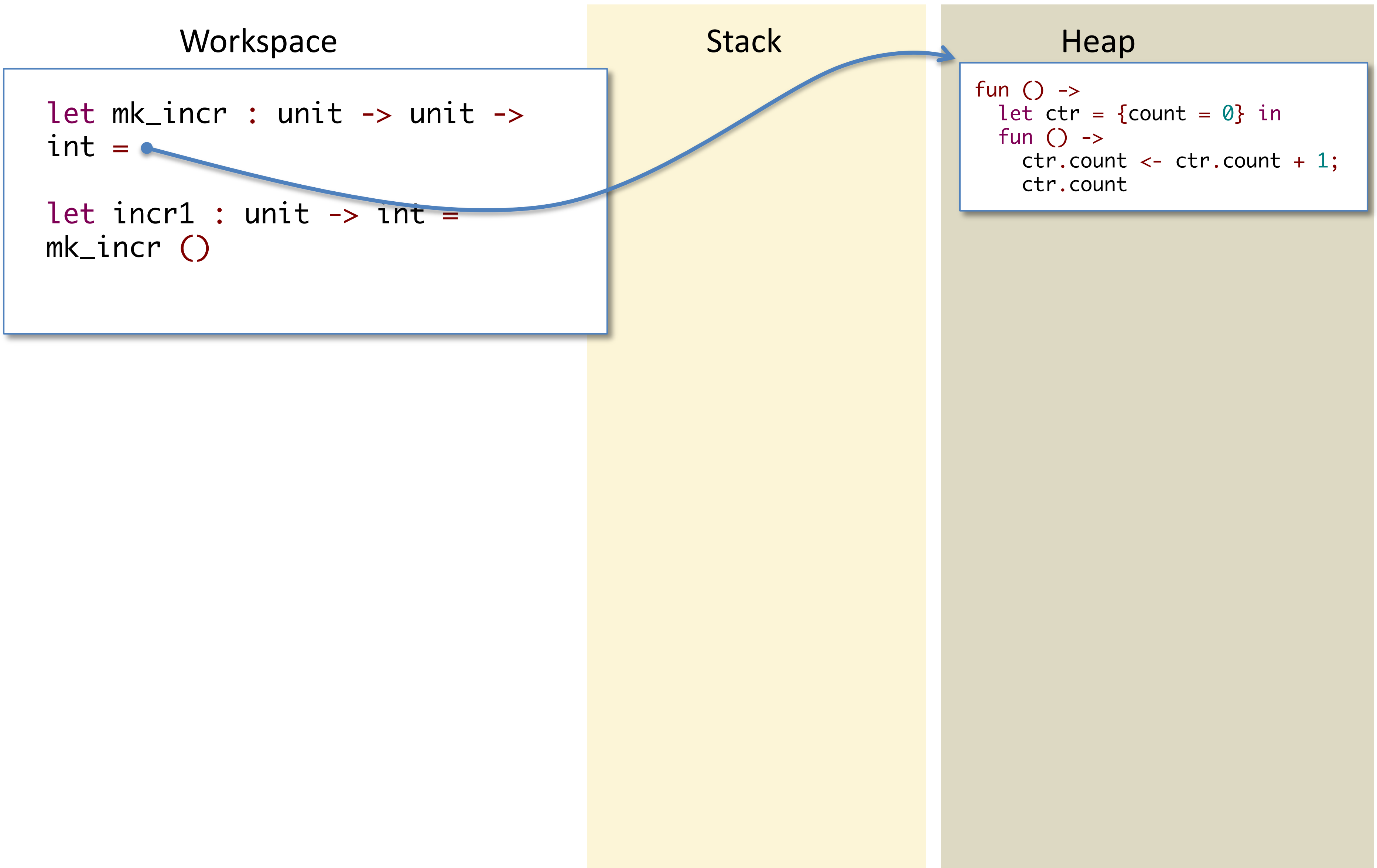
```
let mk_incr : unit -> unit ->
int = fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count


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

Stack

Heap

Running mk_incr



Running mk_incr

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 ()
```

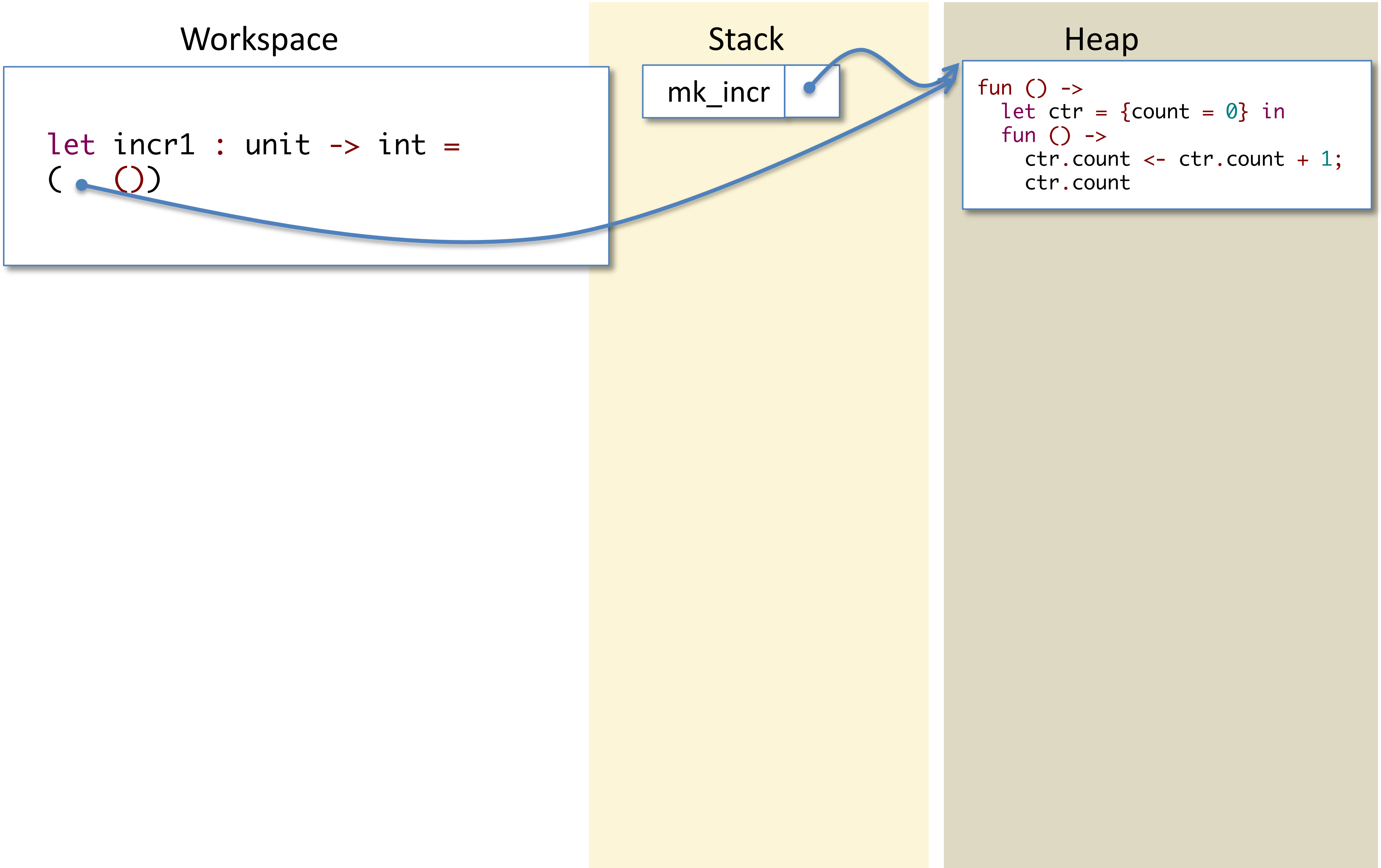
Stack

mk_incr

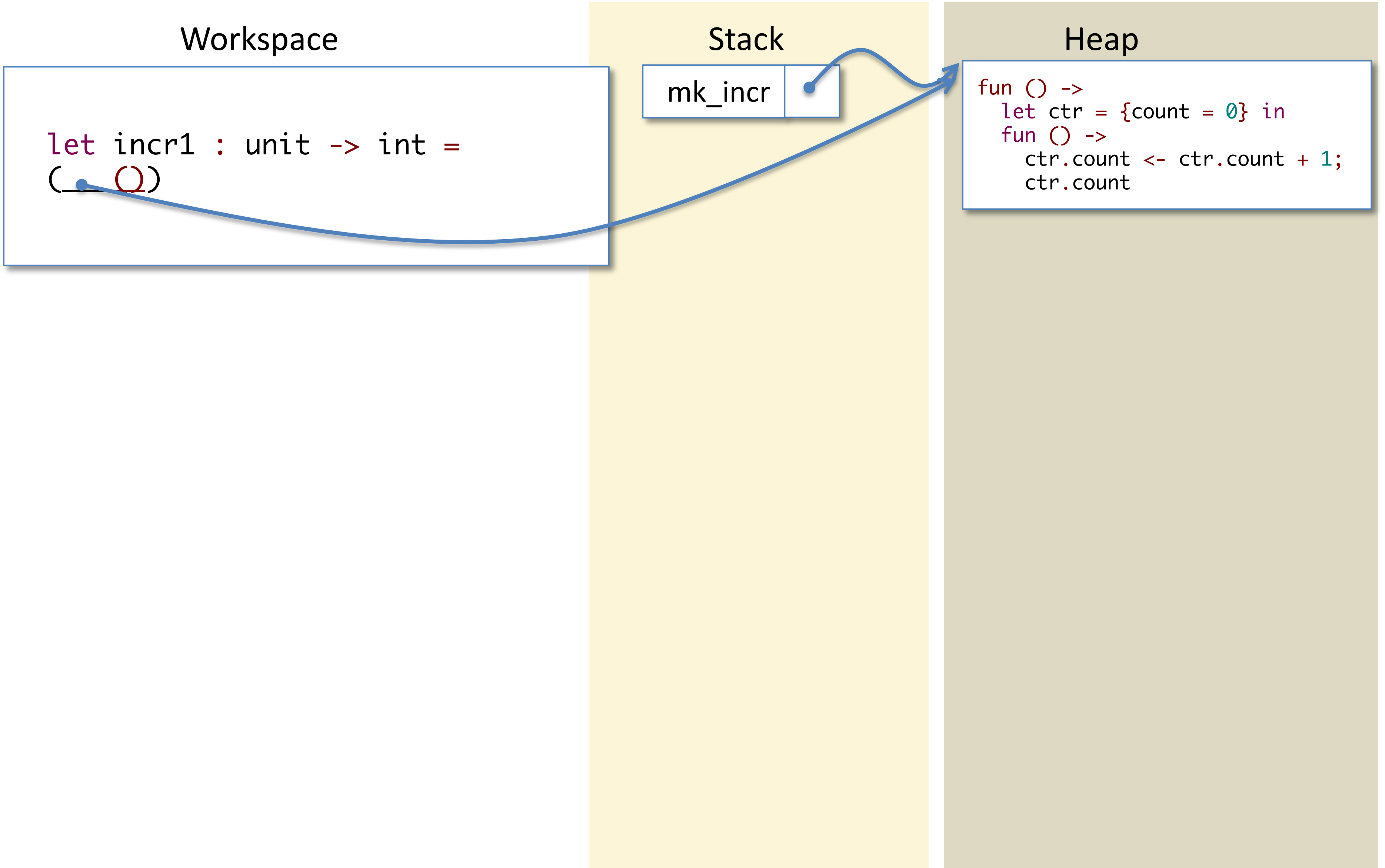
Heap

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

Running mk_incr



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 =
(___)
```

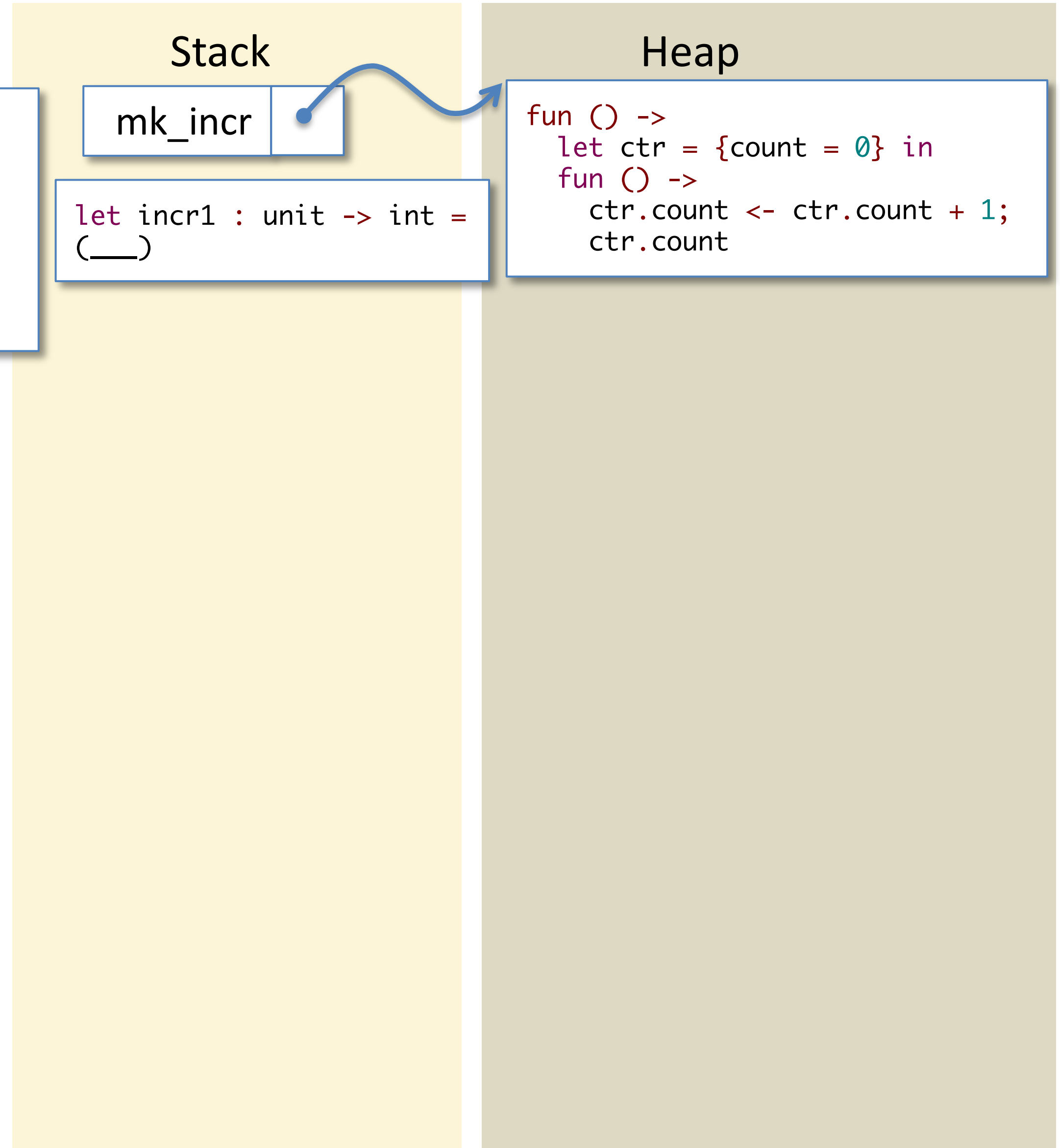
Heap

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

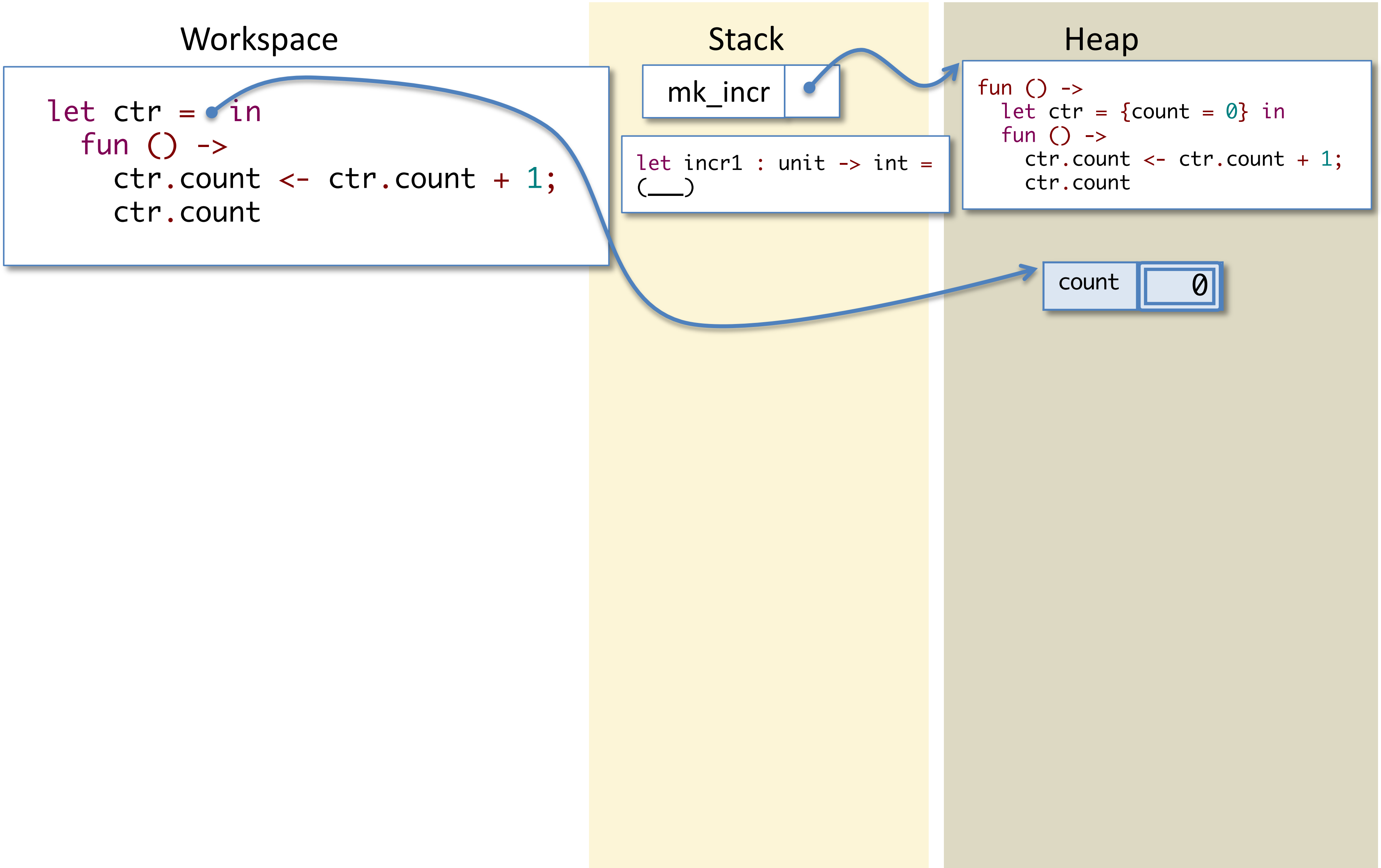
Running mk_incr

Workspace

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



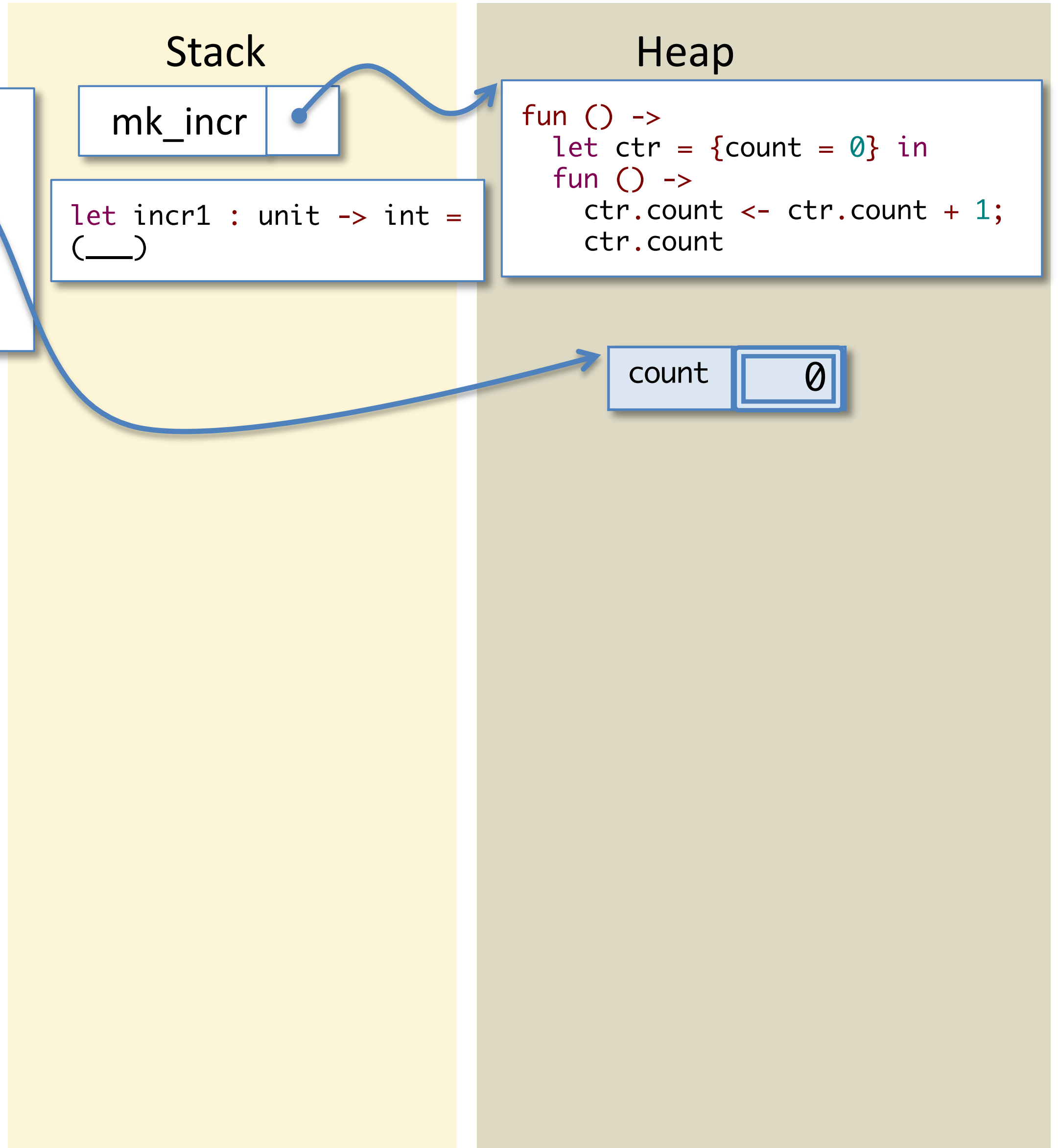
Running mk_incr



Running mk_incr

Workspace

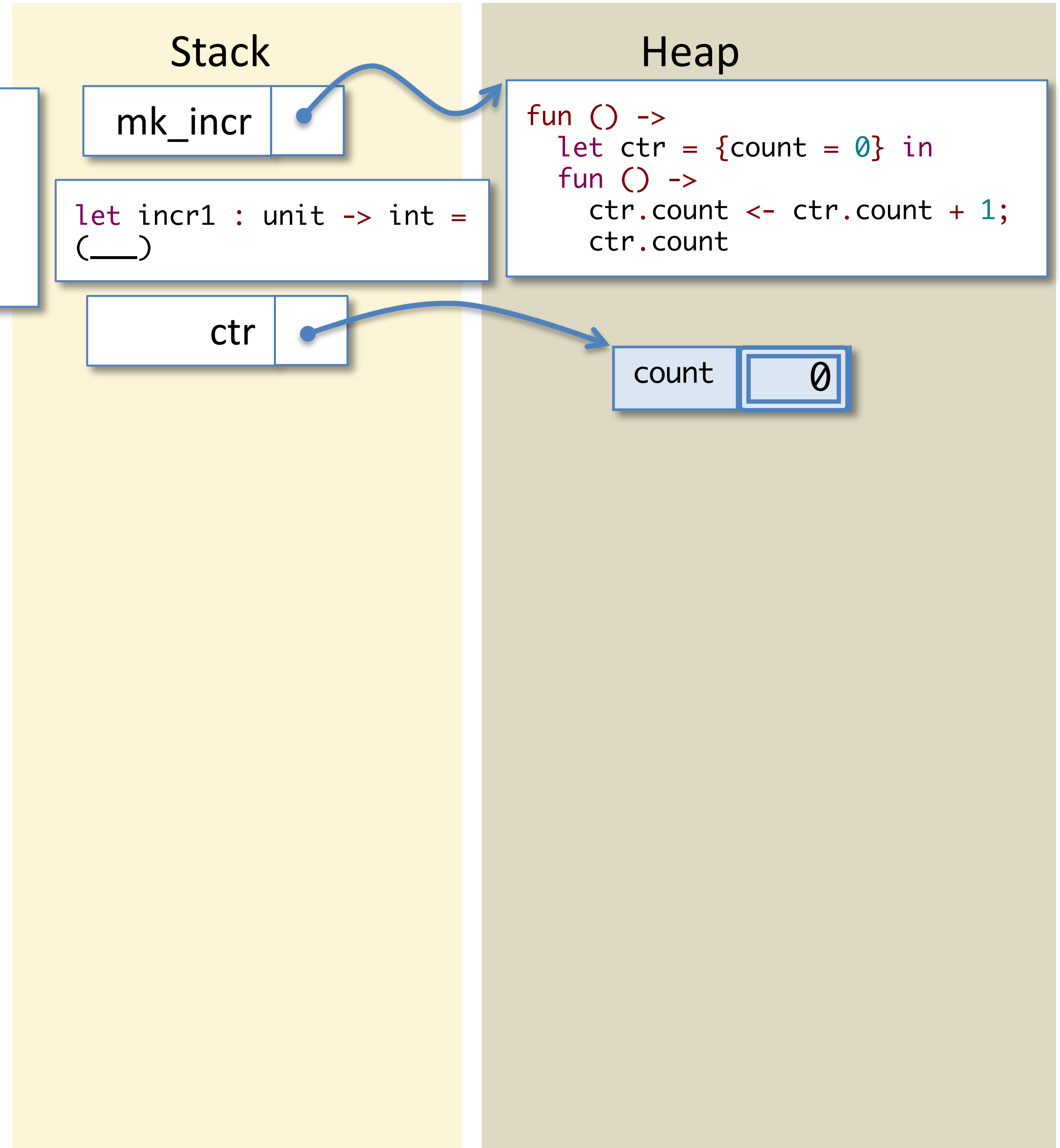
```
let ctr = ● in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```



Running mk_incr

Workspace

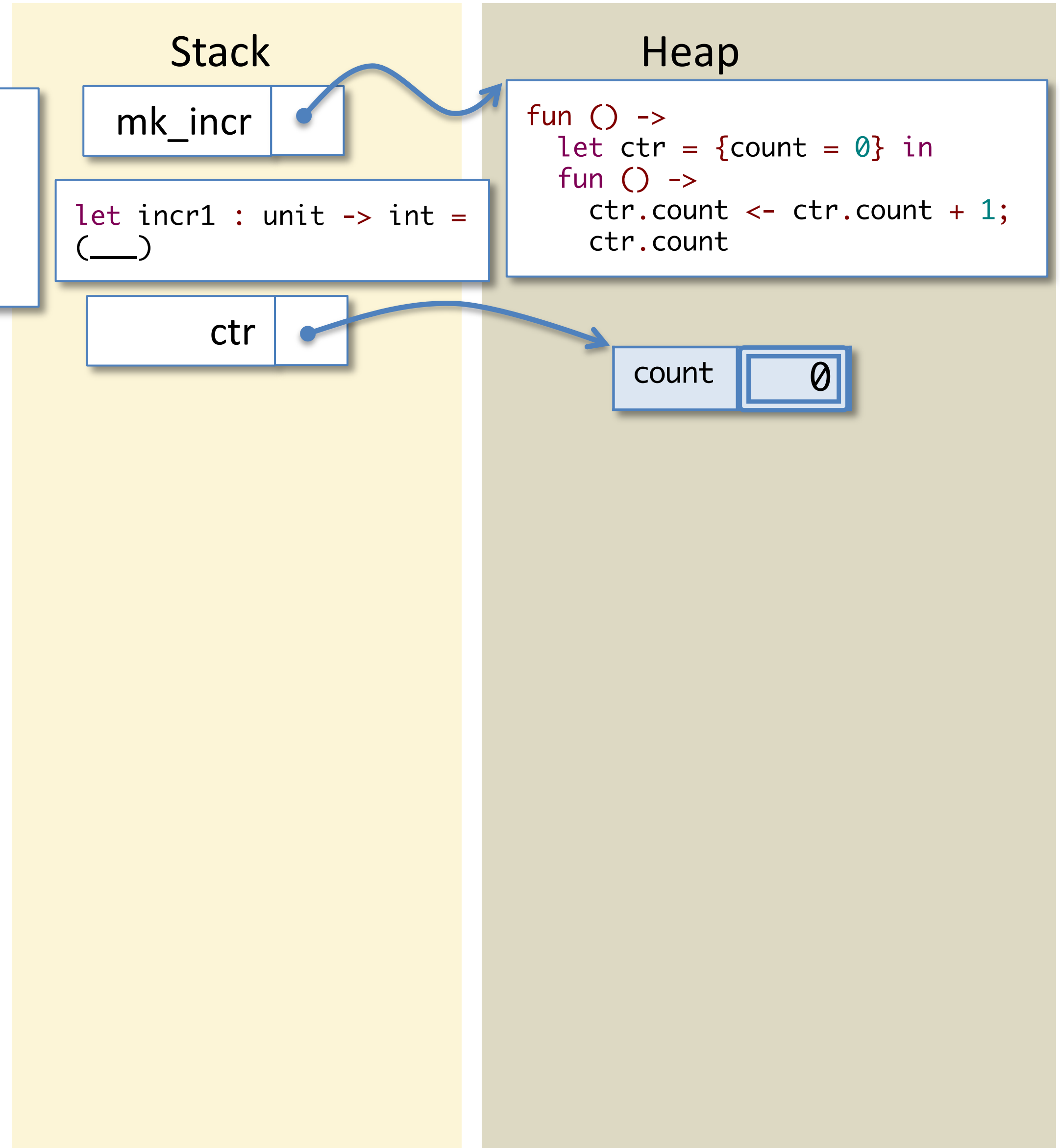
```
fun () ->
  ctr.count <- ctr.count + 1;
  ctr.count
```



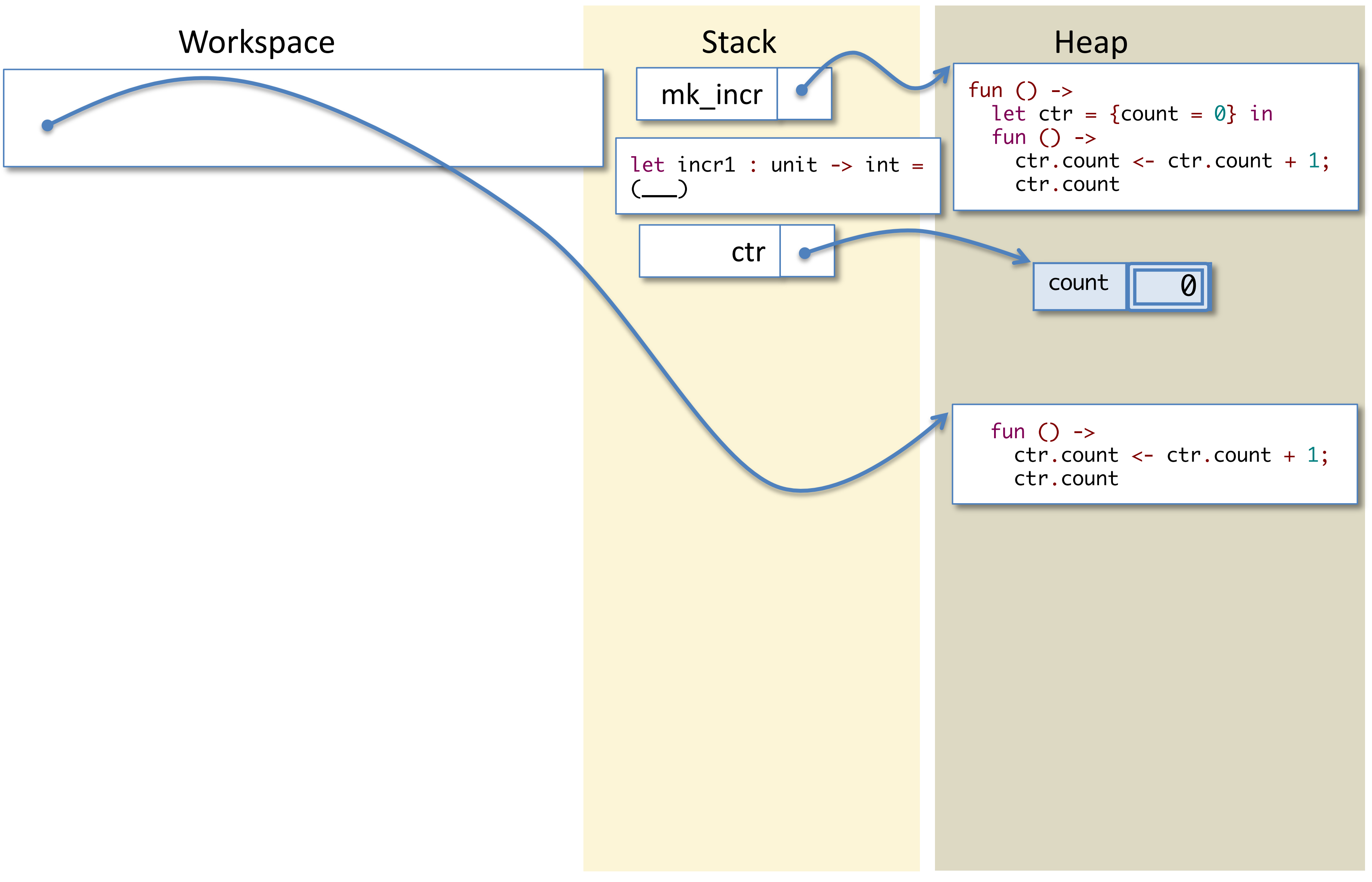
Running mk_incr

Workspace

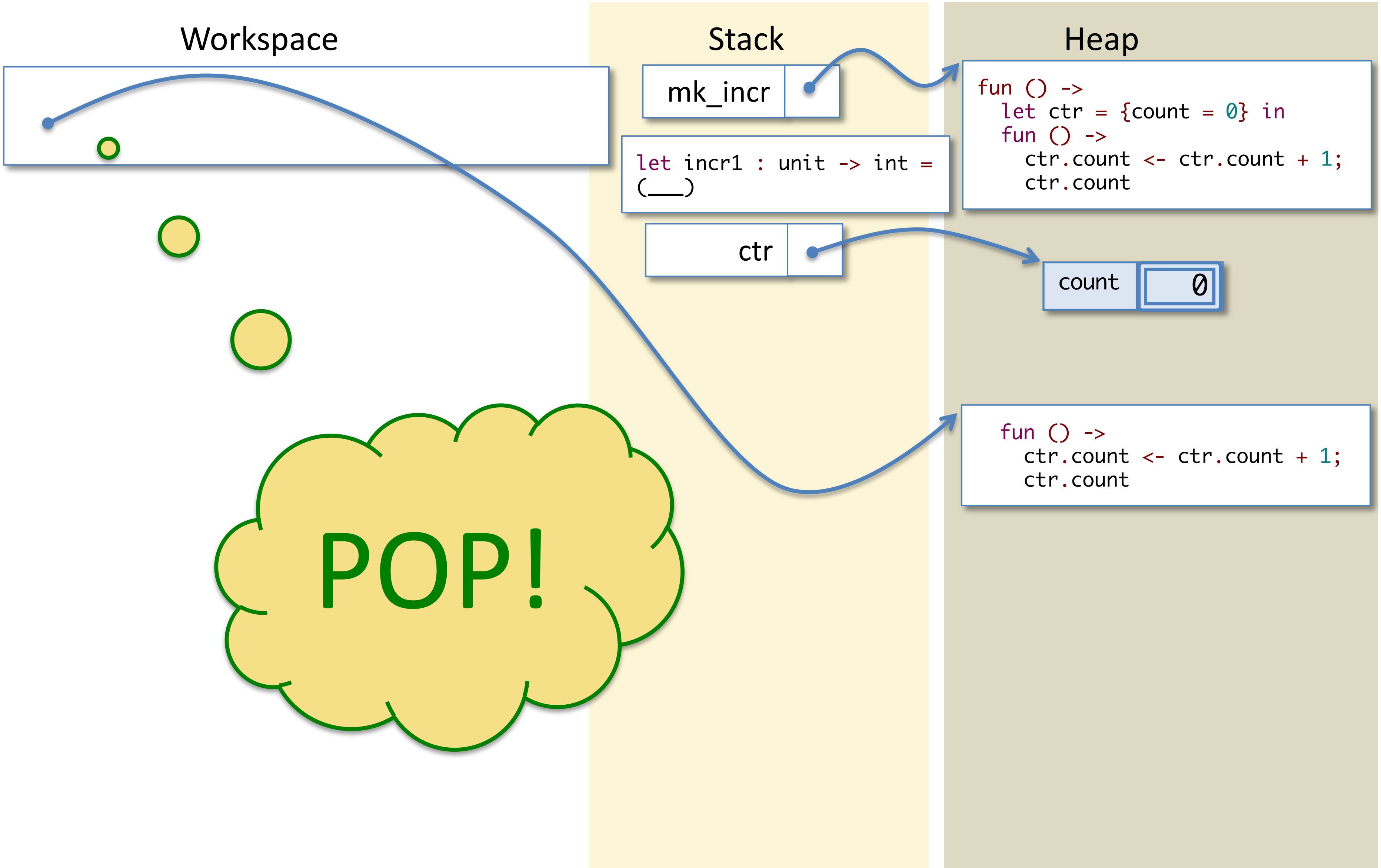
```
fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```



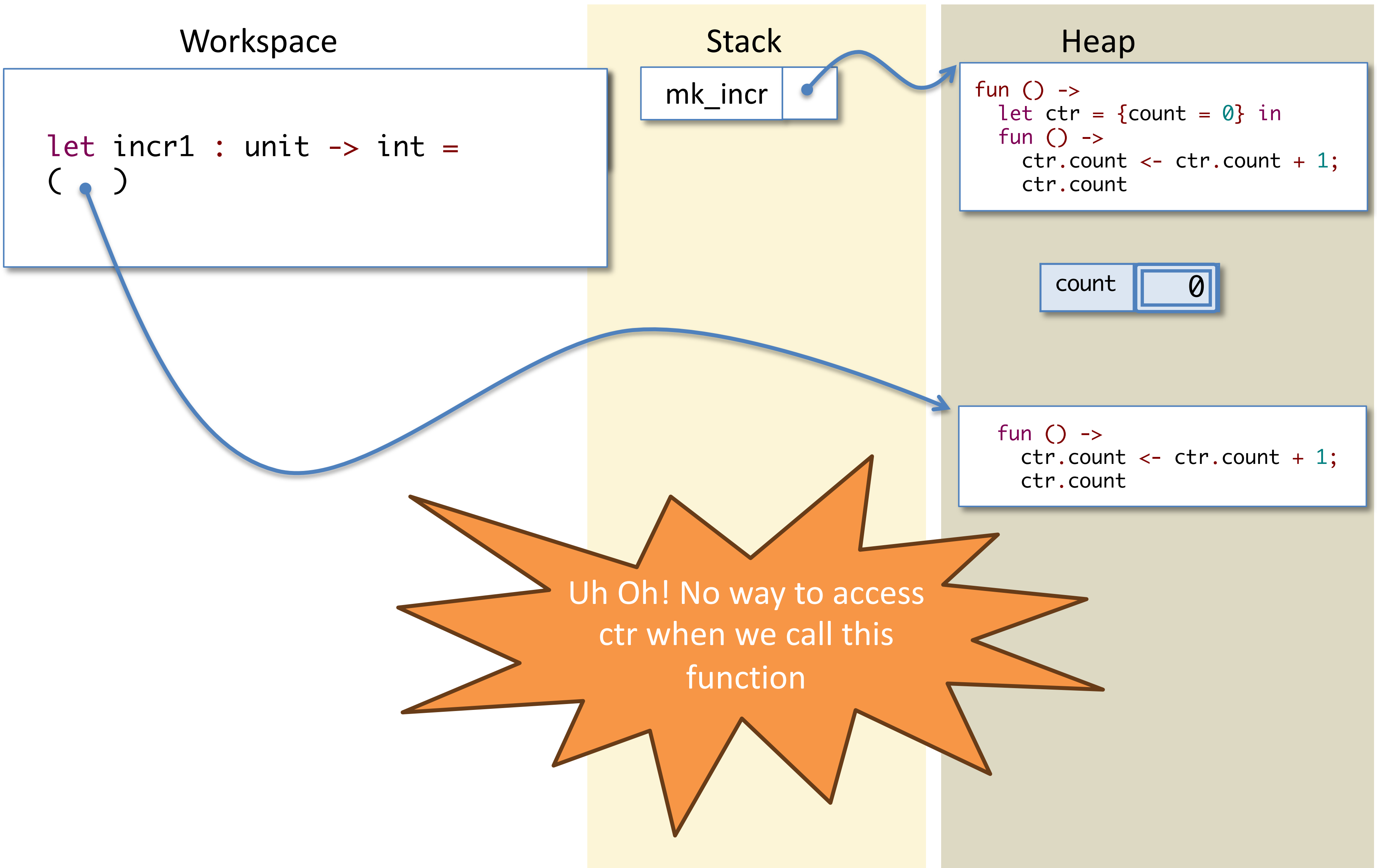
Local Functions (wrong)



Local Functions (wrong)

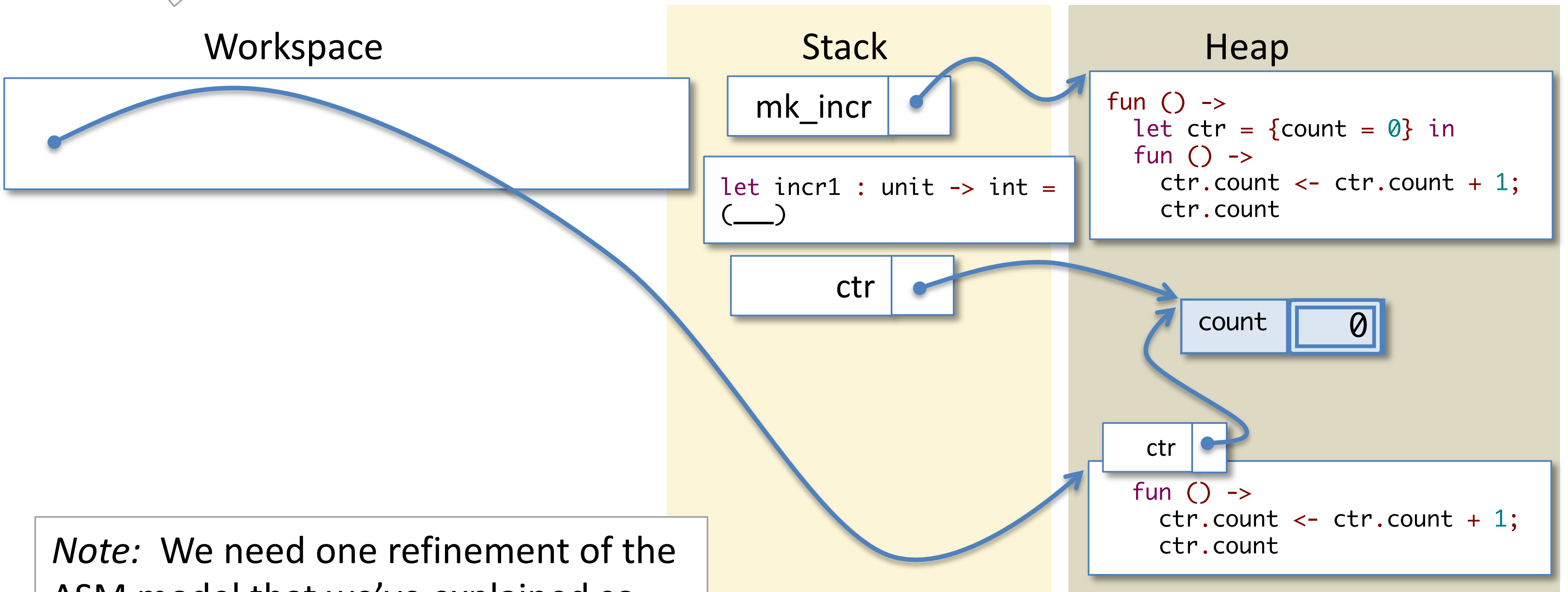


Local Functions (wrong)



Local Functions (right)

Key step!



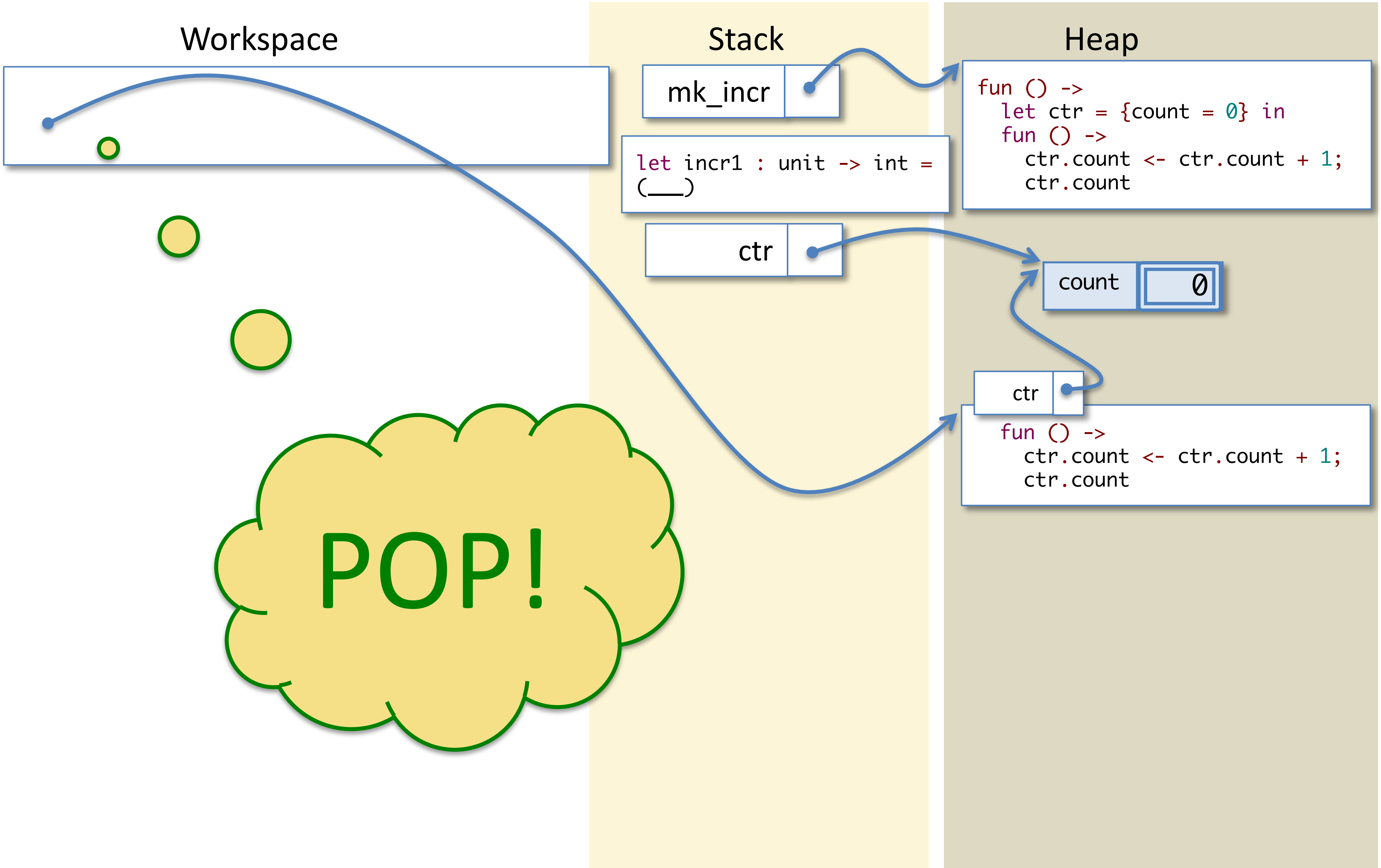
Note: We need one refinement of the ASM model that we've explained so far. Why?

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*…

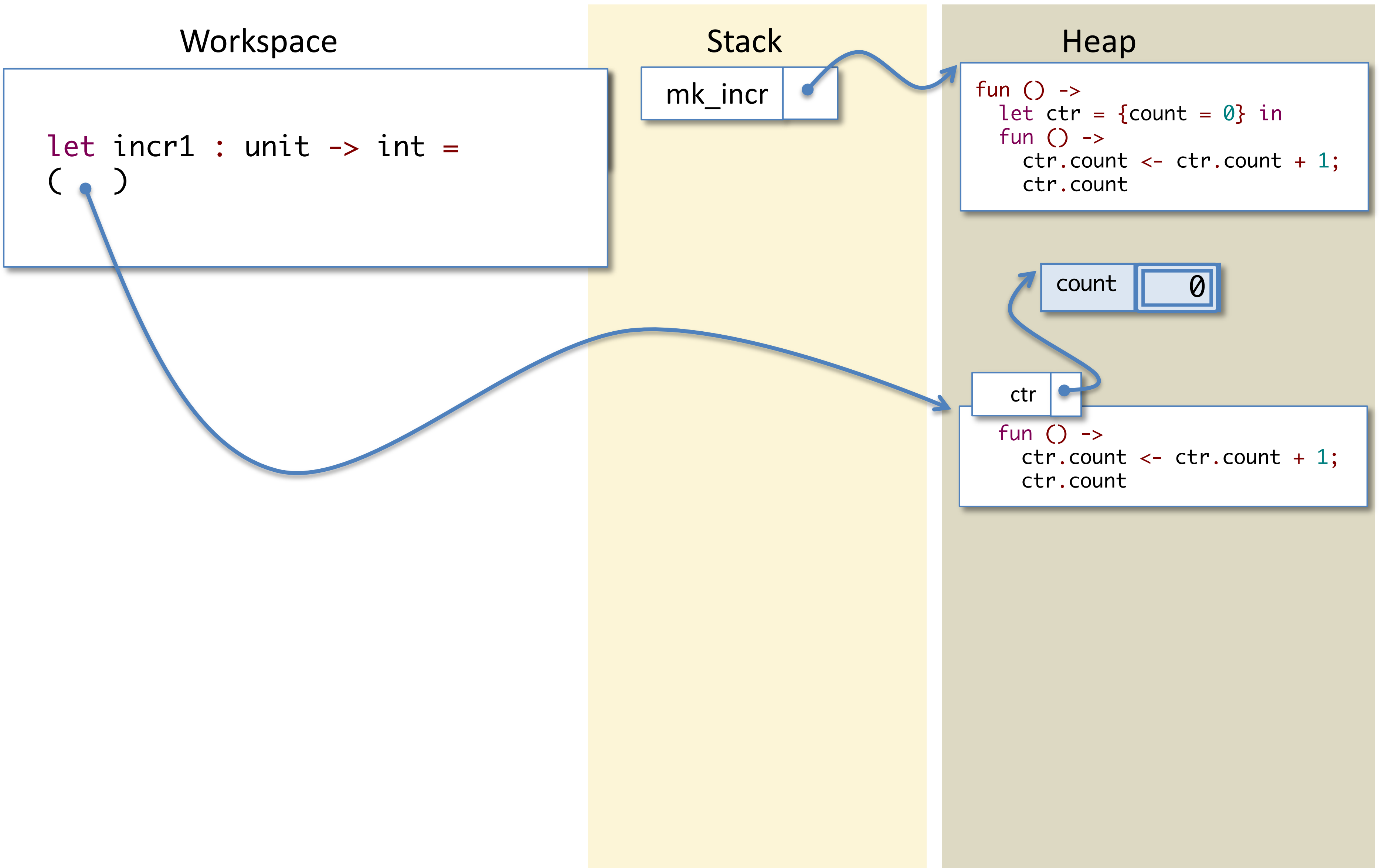
…so we save a copy of the relevant stack binding with the function itself.

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

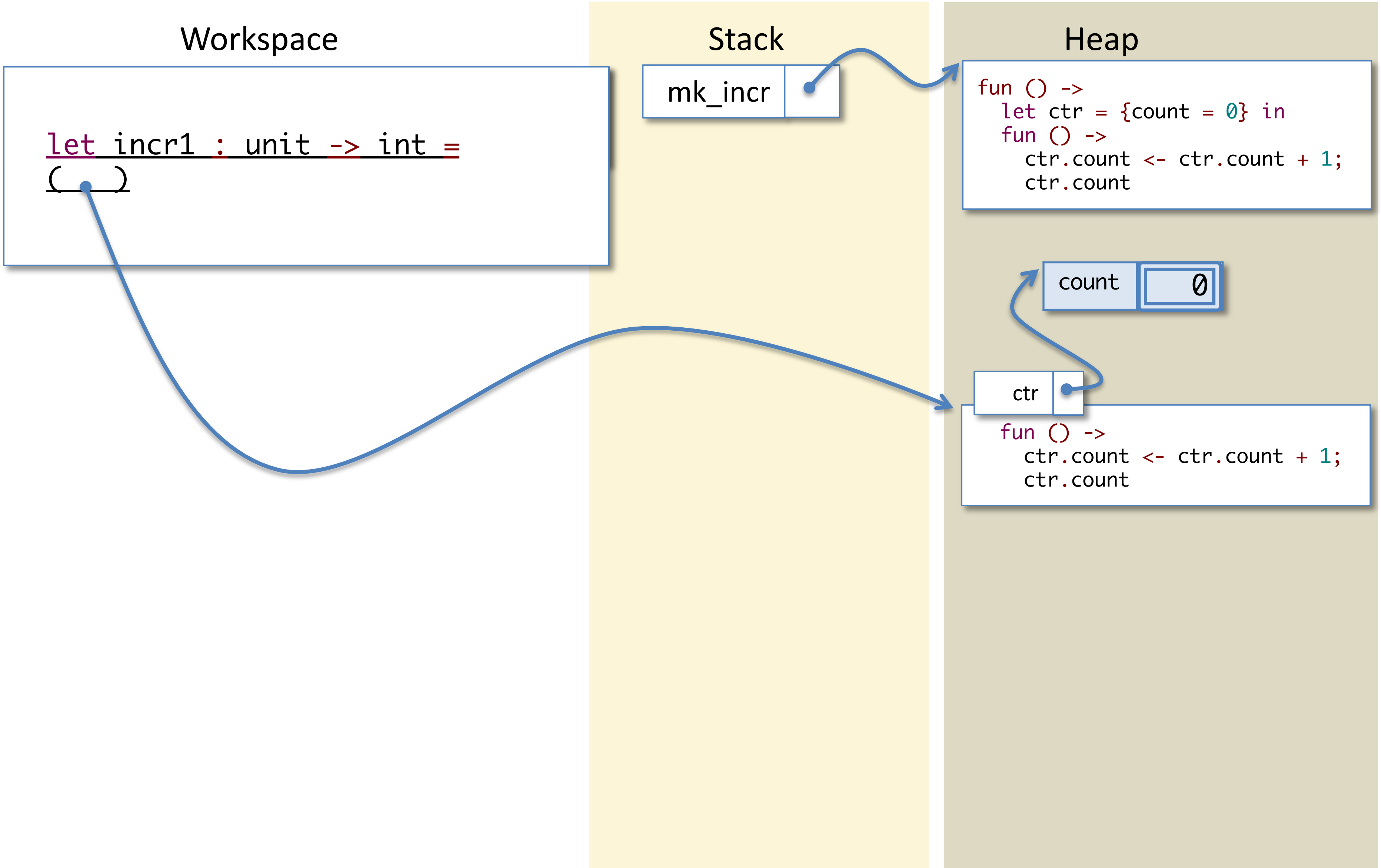
Local Functions



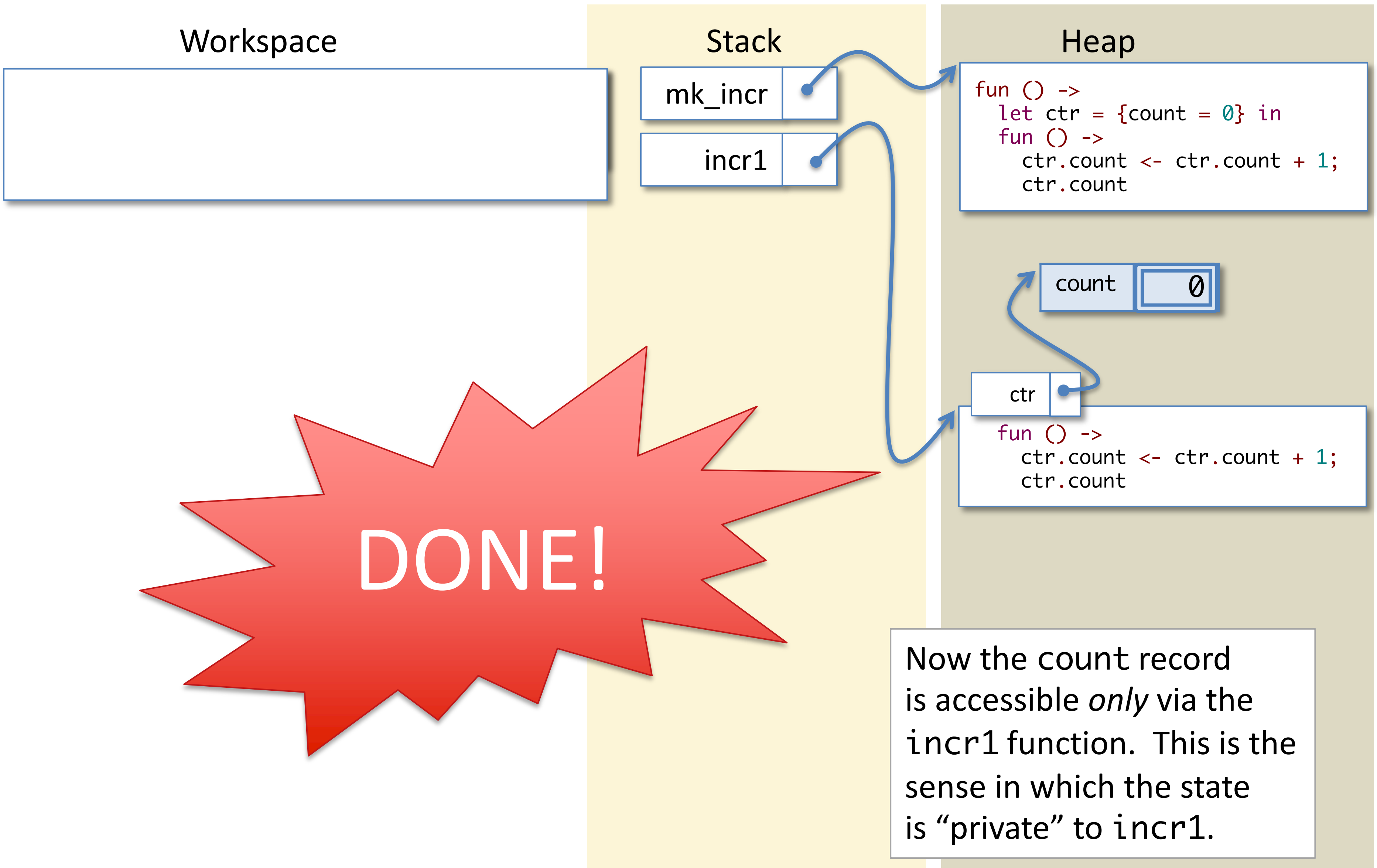
Local Functions



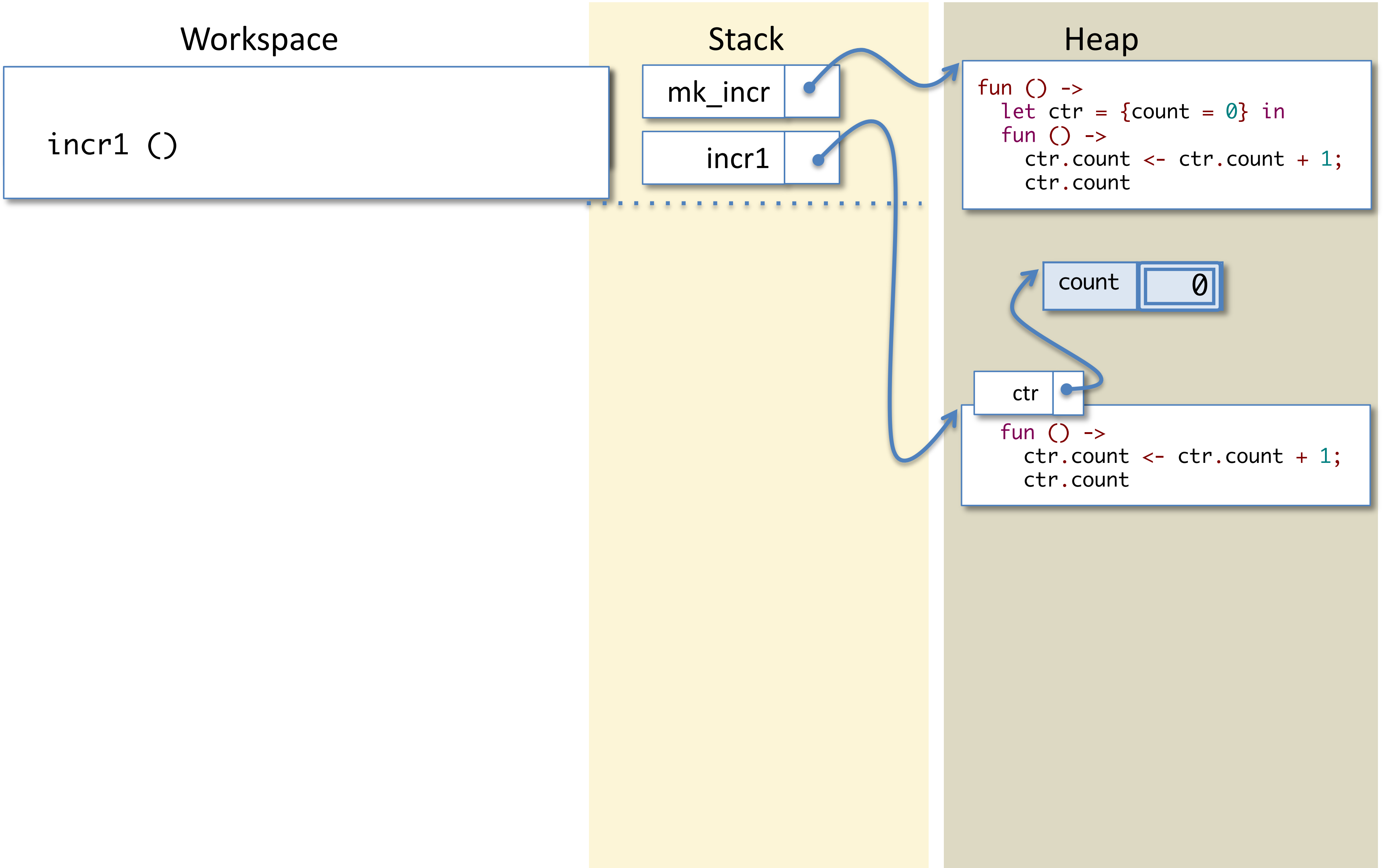
Local Functions



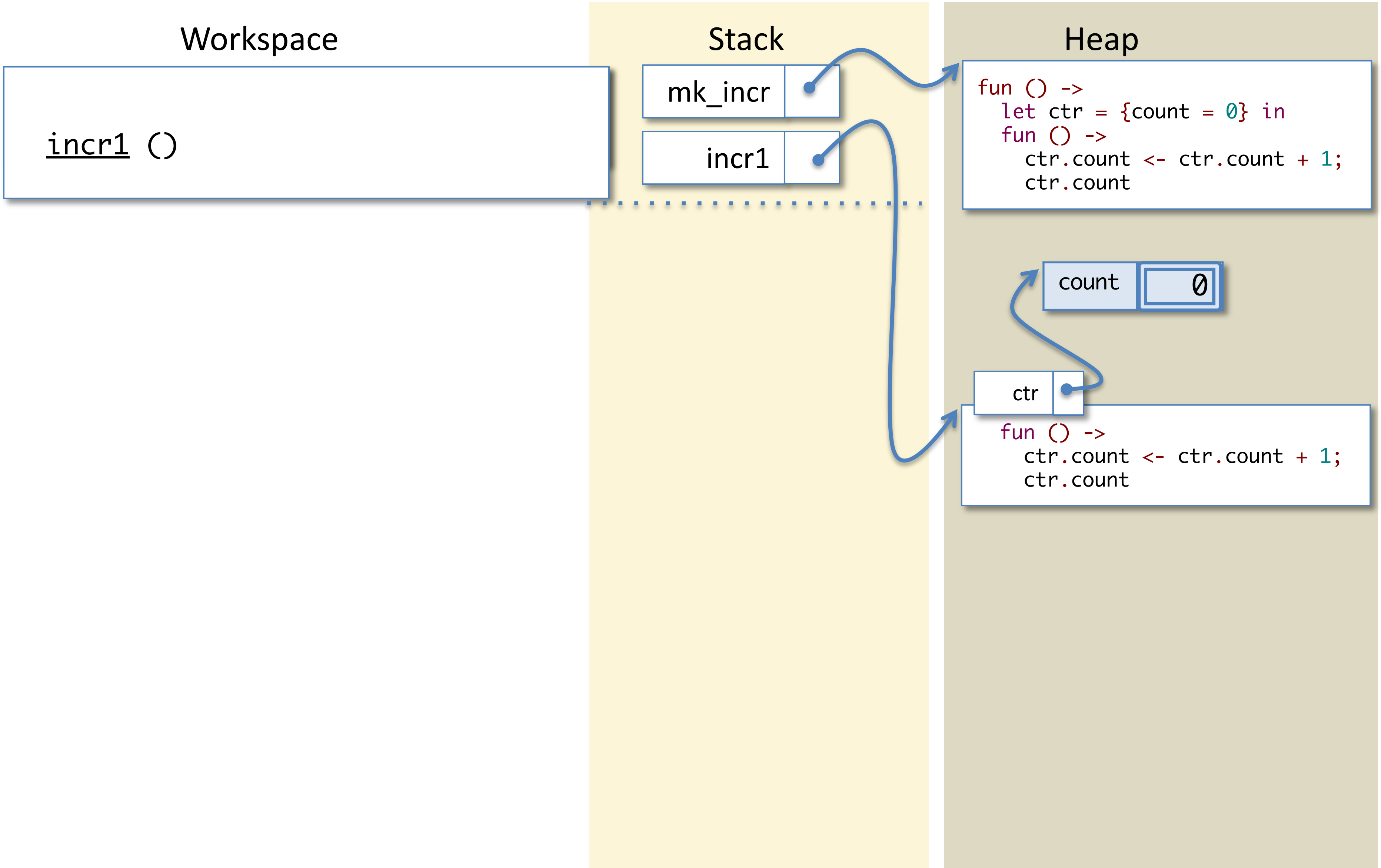
Local Functions



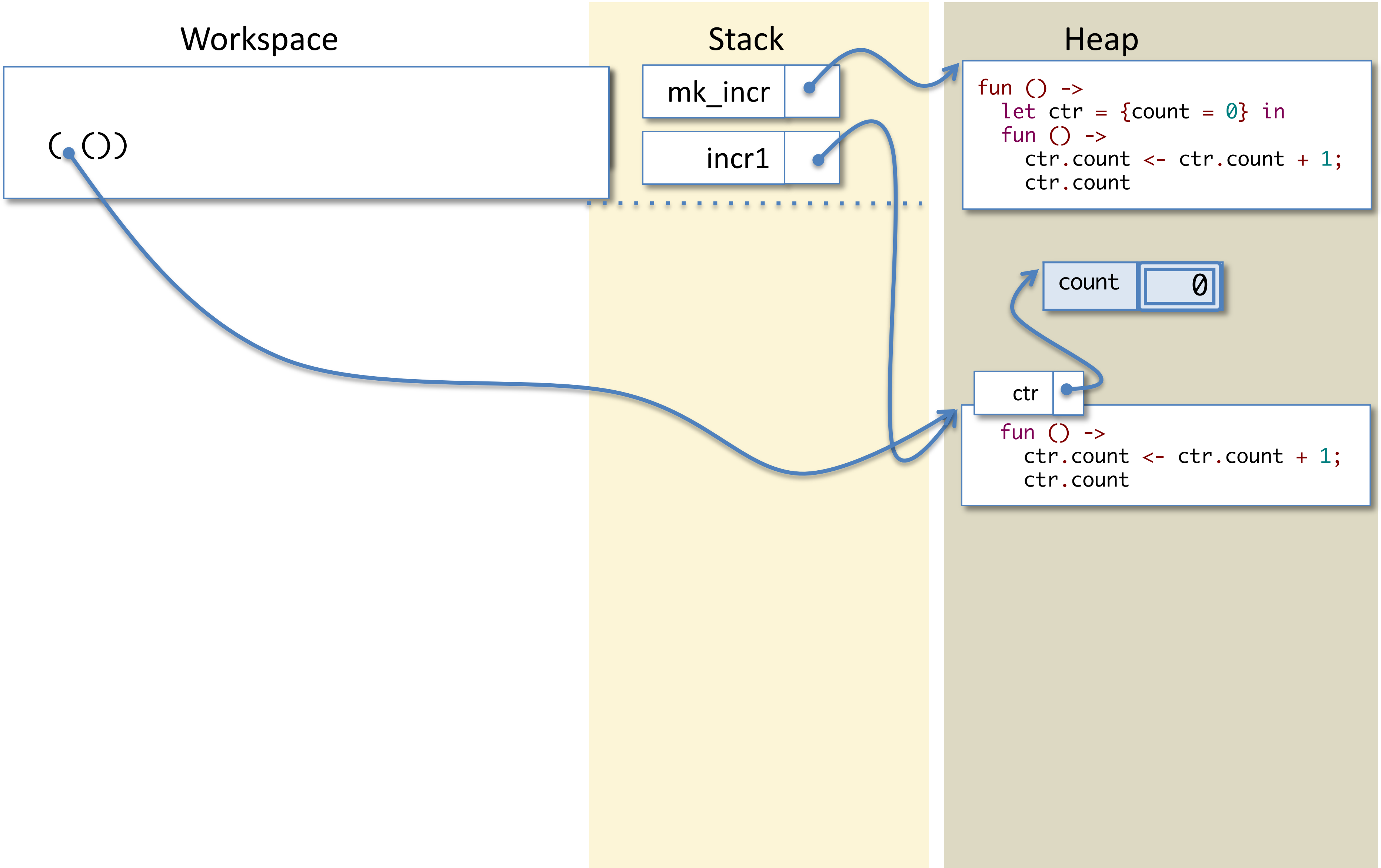
Now let's run "incr1 ()"



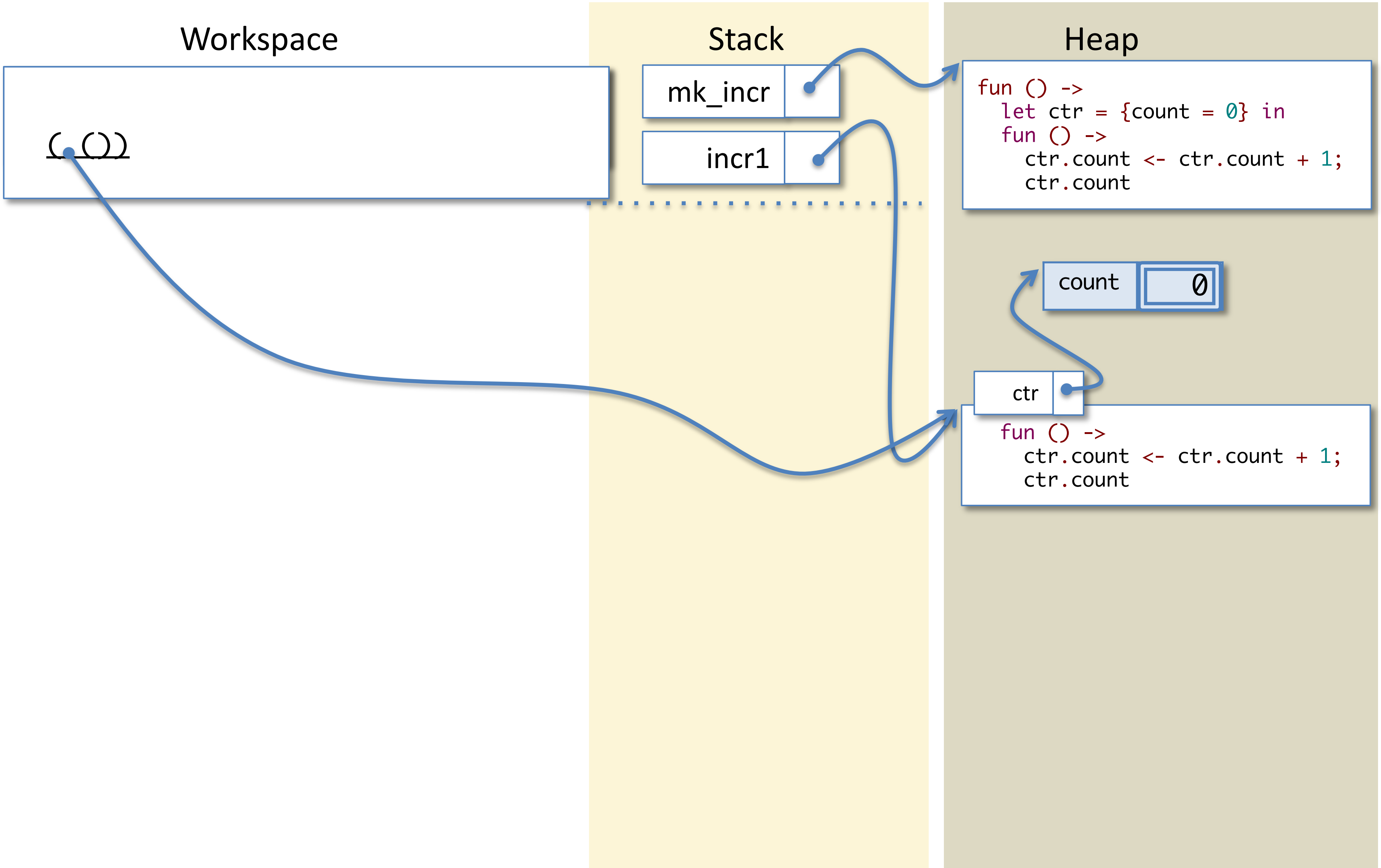
Now let's run "incr1 ()"



Now let's run "incr1 ()"



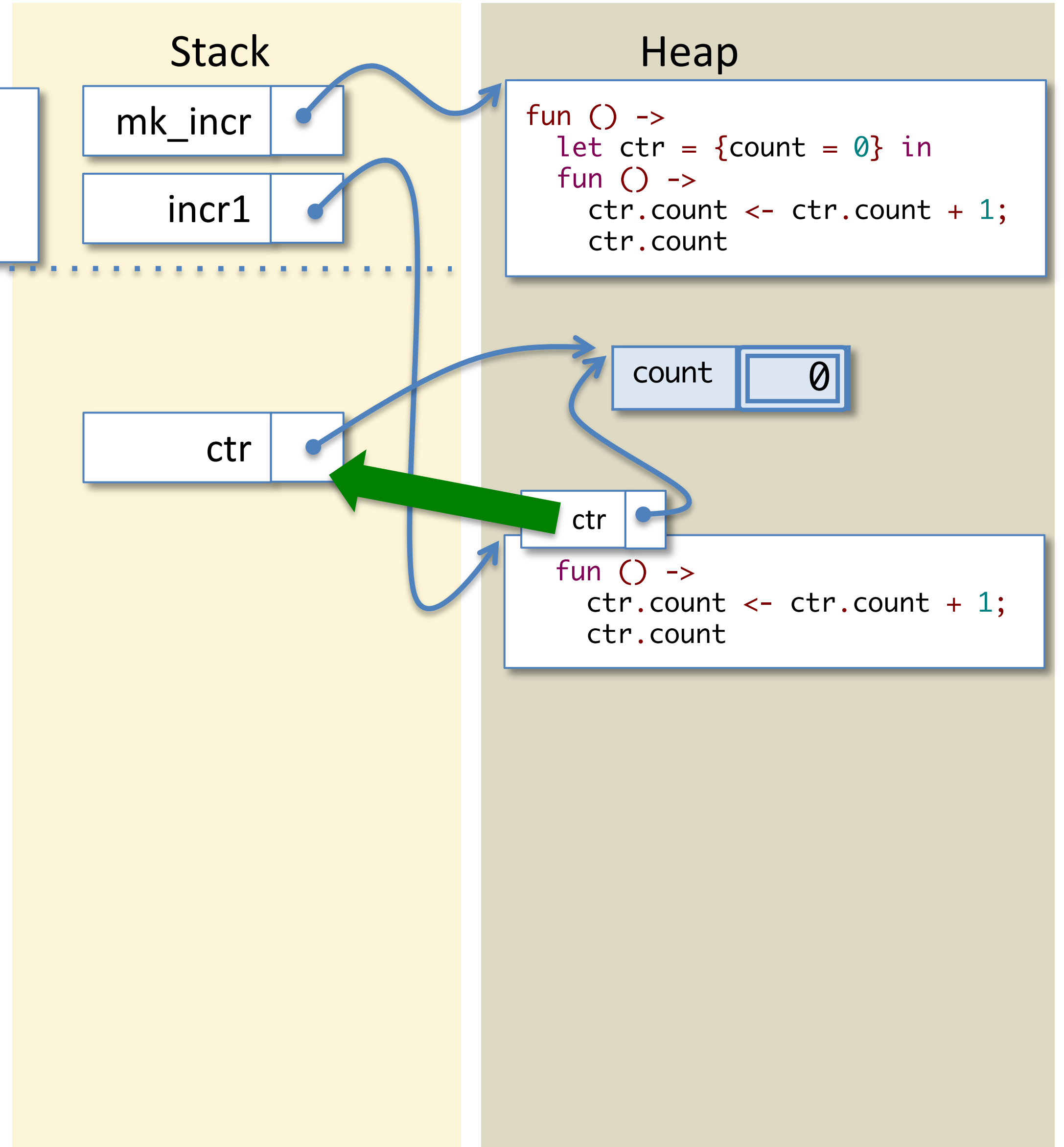
Now let's run "incr1 ()"



Now let's run "incr1 ()"

Workspace

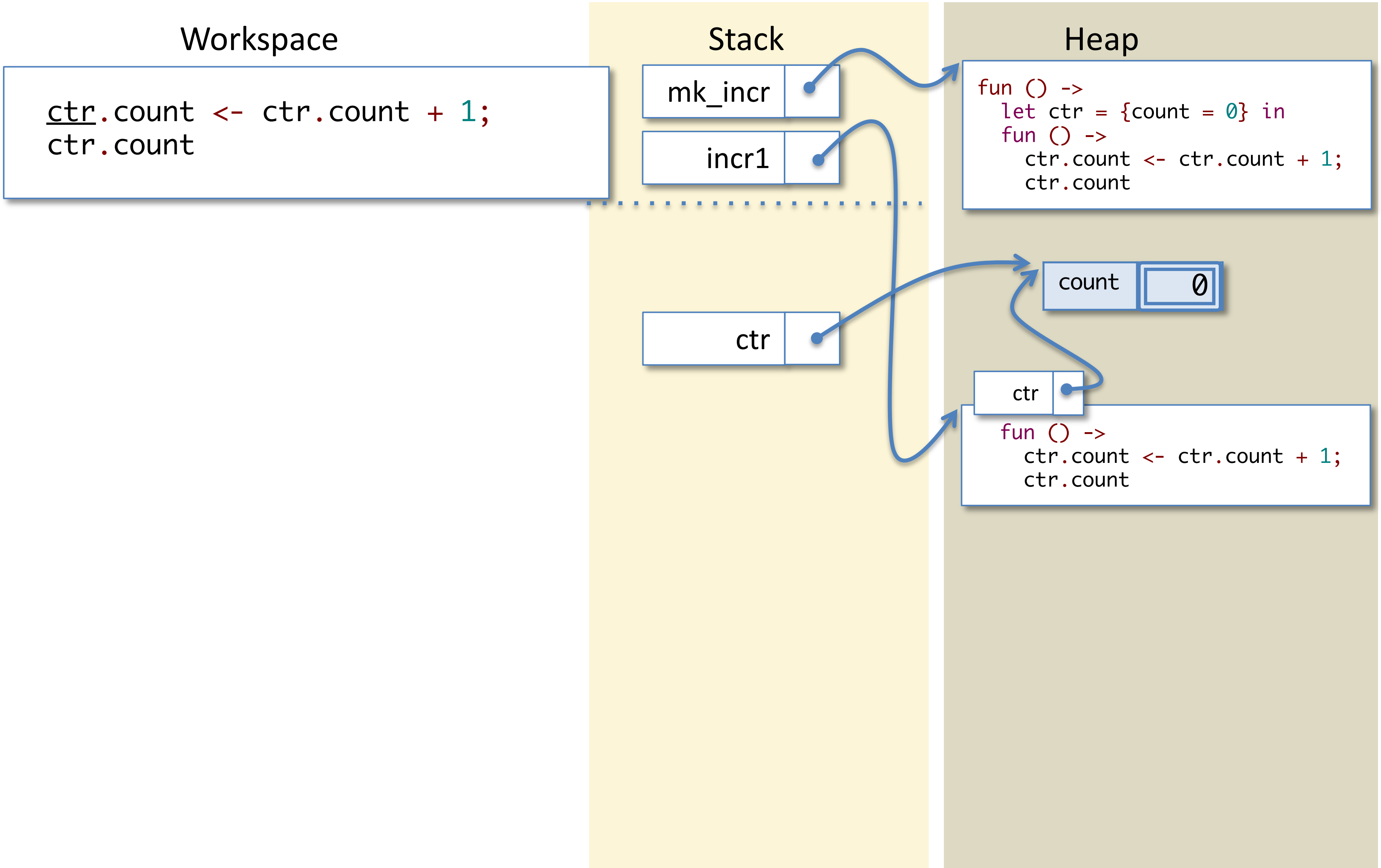
```
ctr.count <- ctr.count + 1;
ctr.count
```



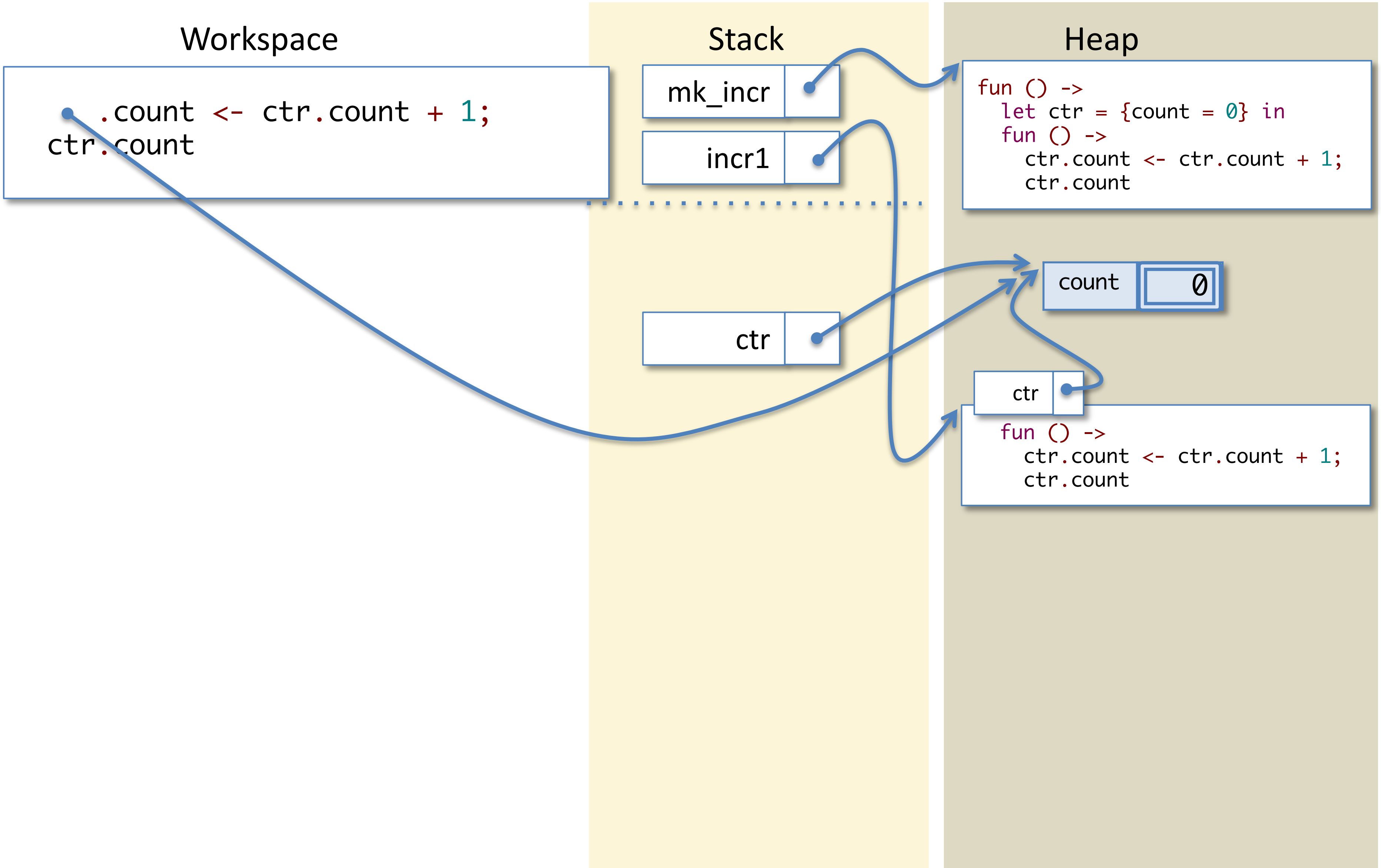
Tail Call!

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.

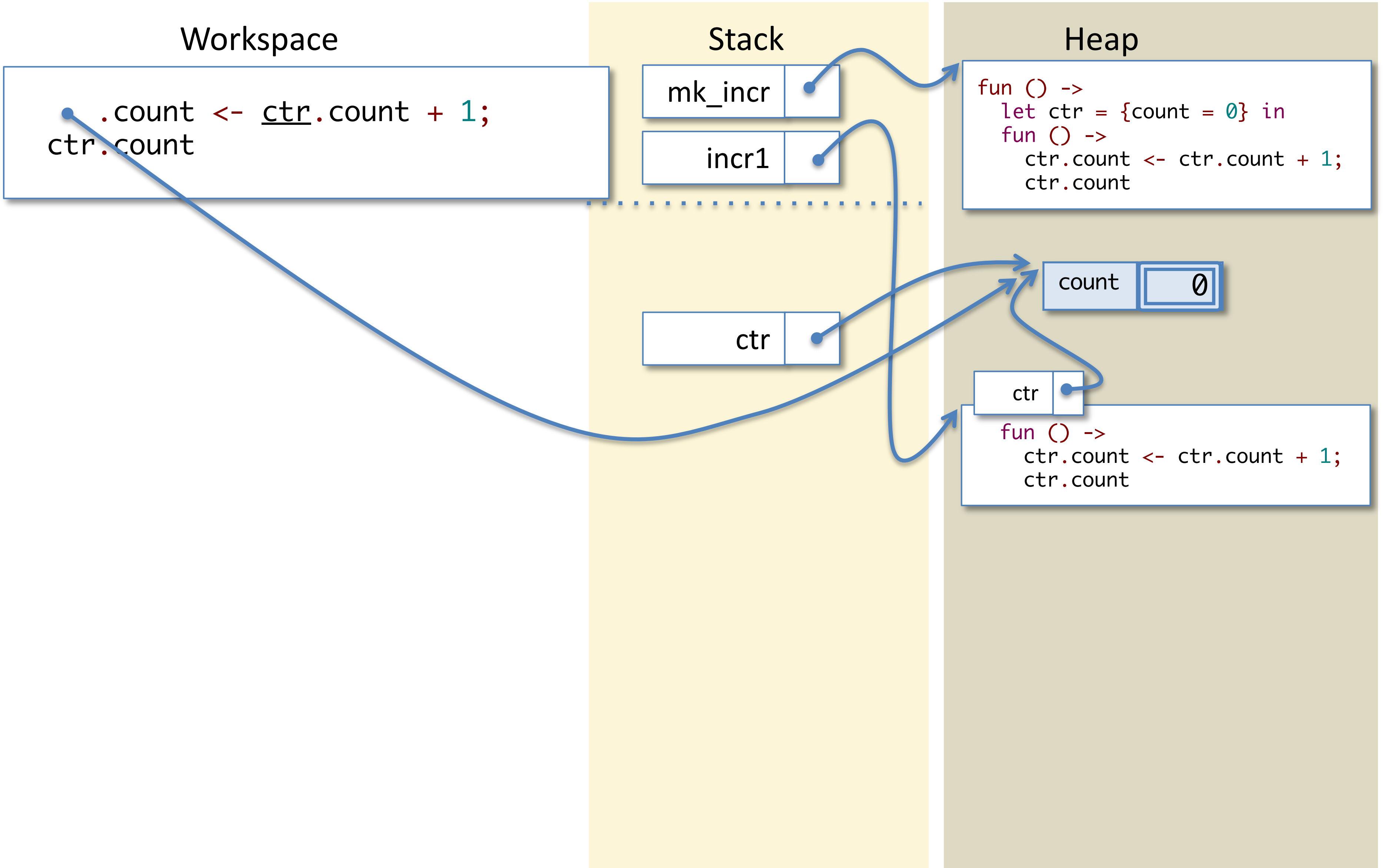
Now let's run "incr1 ()"



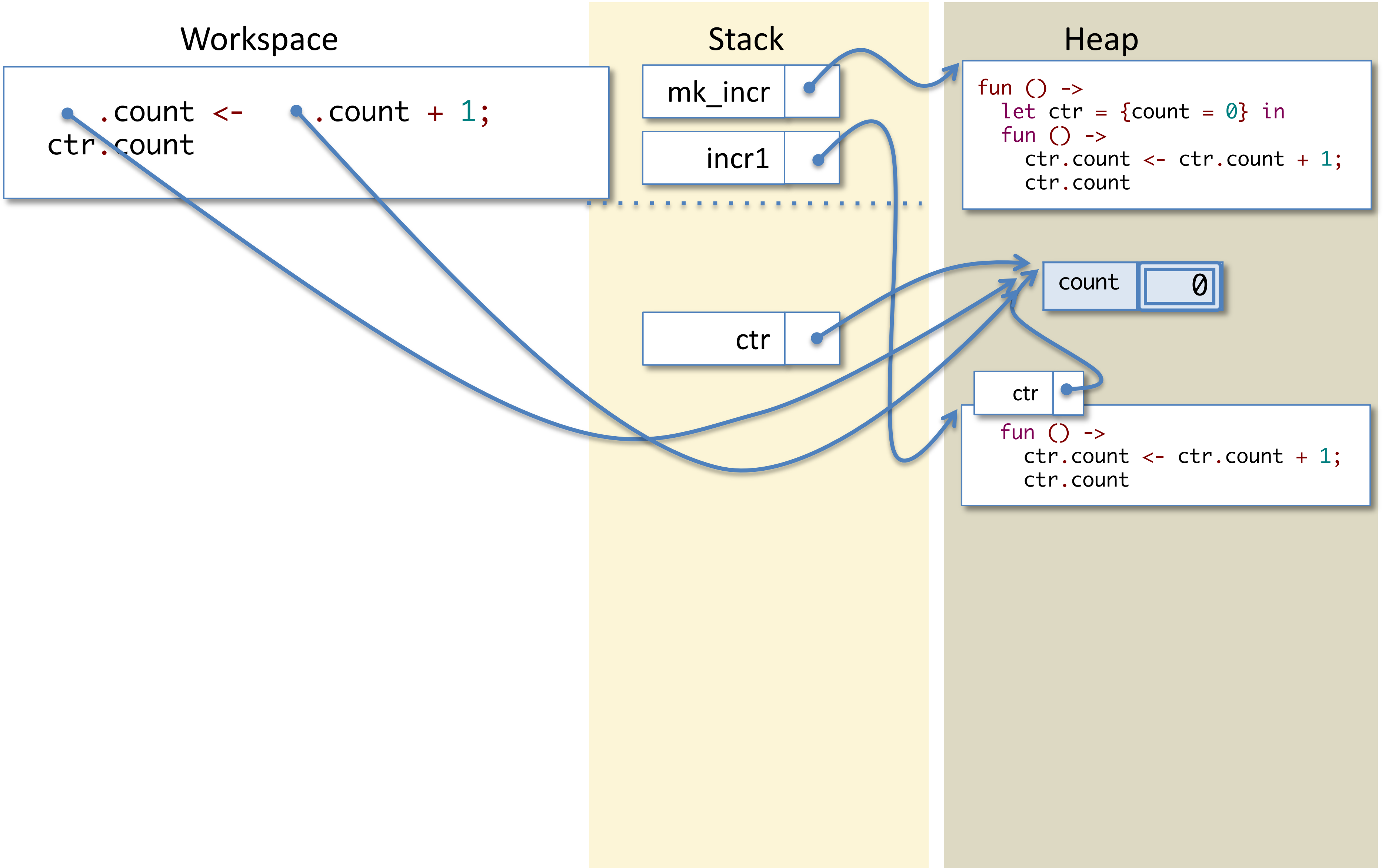
Now let’s run “incr1 ()”



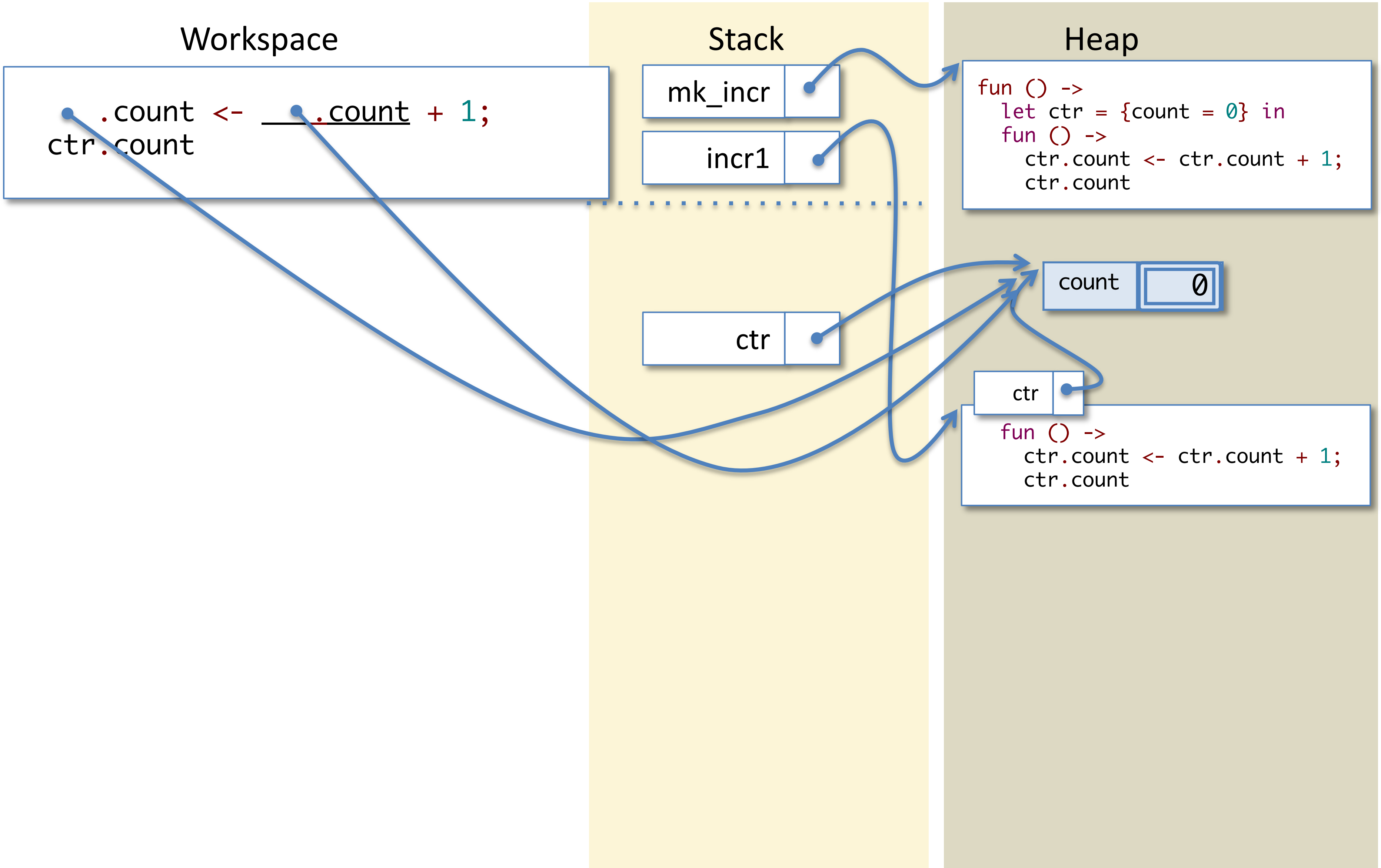
Now let's run "incr1 ()"



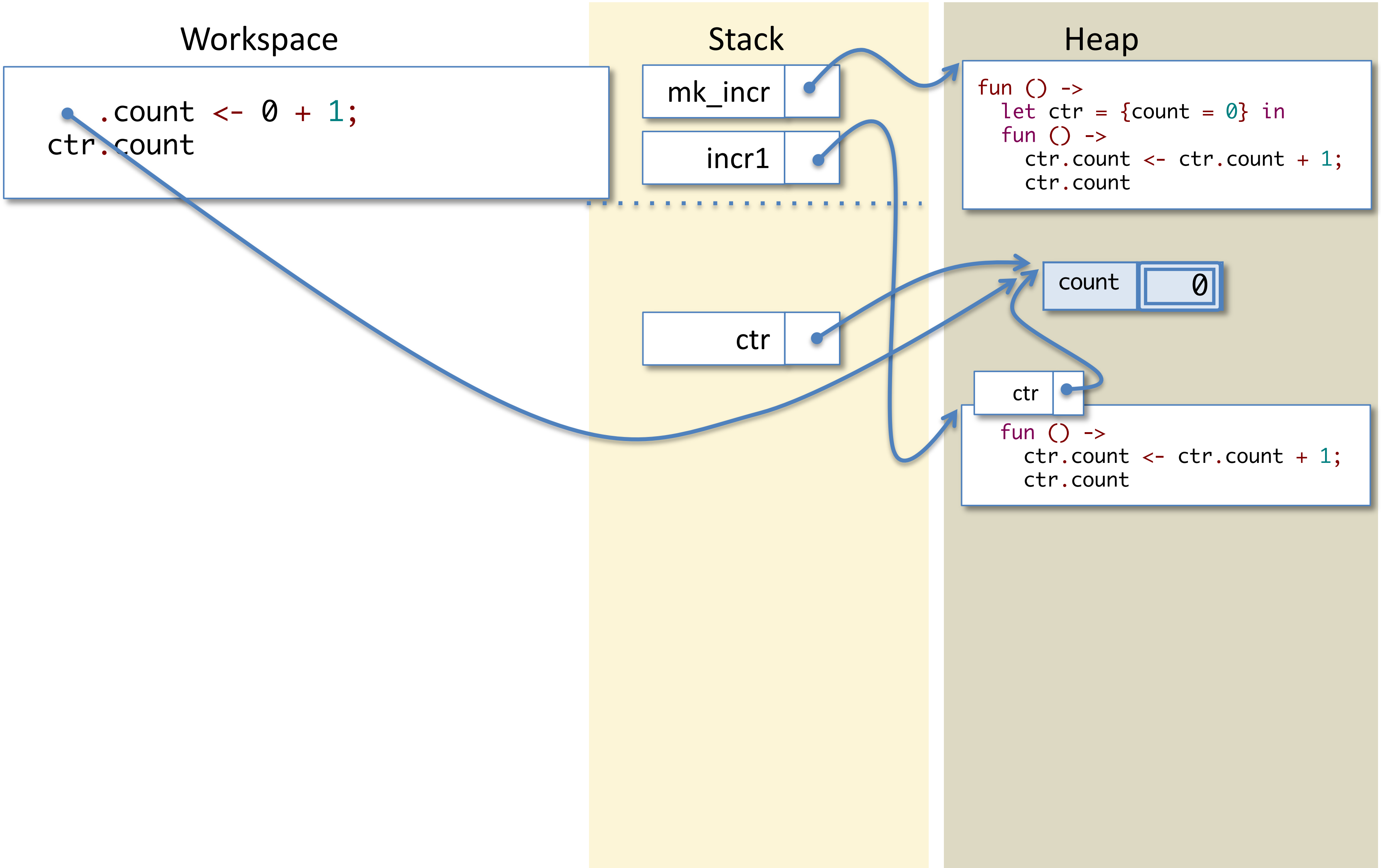
Now let’s run “incr1 ()”



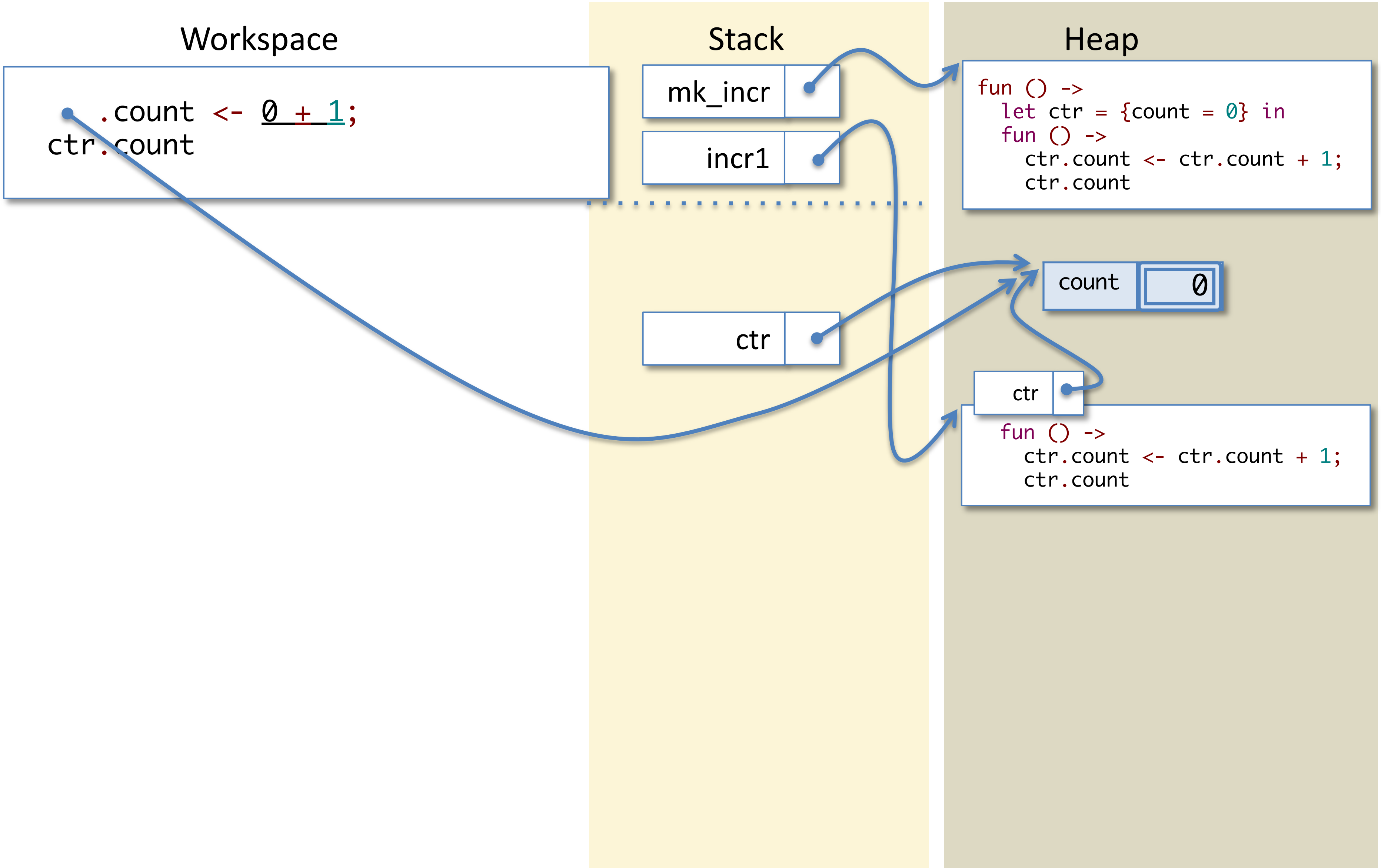
Now let's run "incr1 ()"



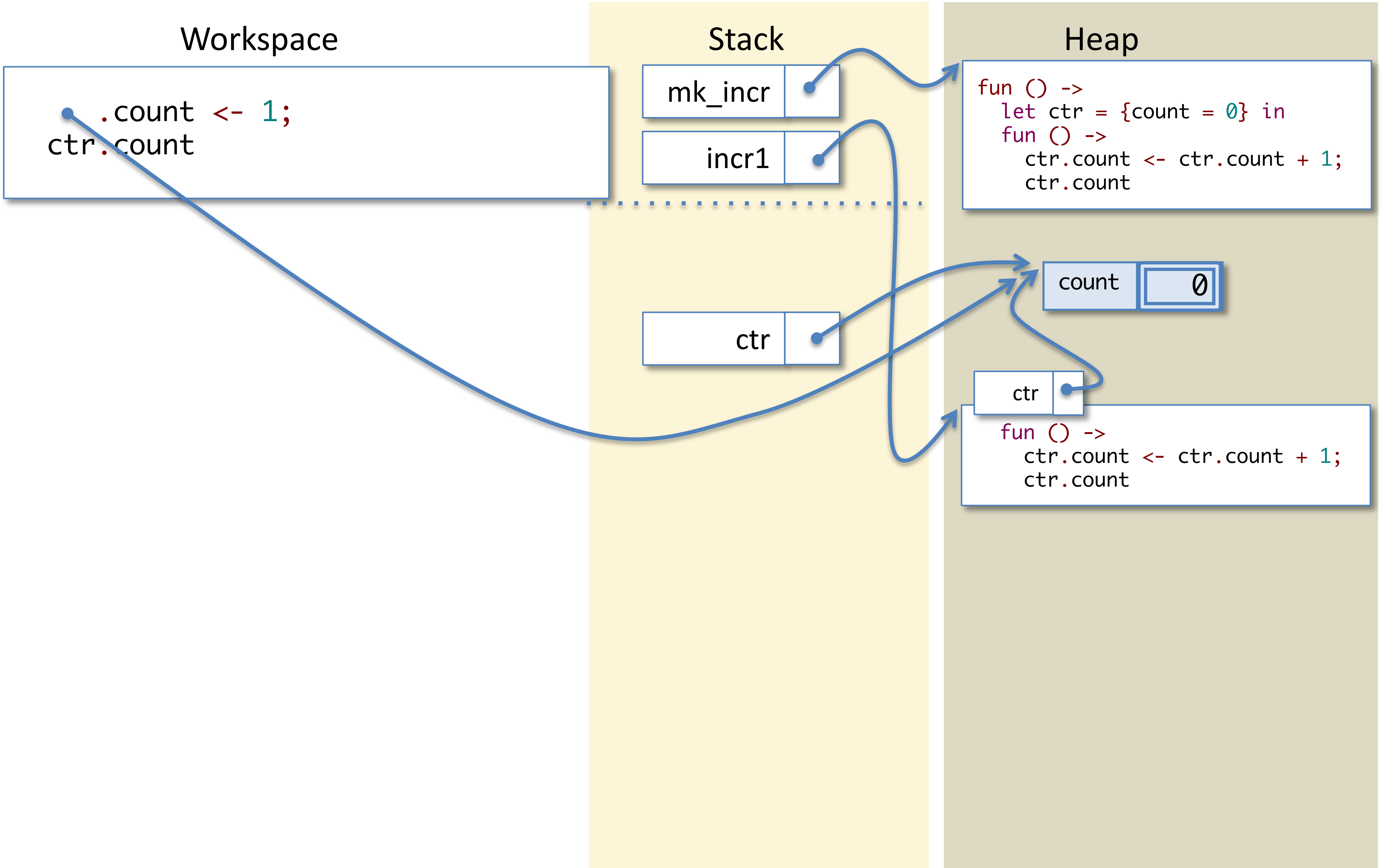
Now let's run "incr1 ()"



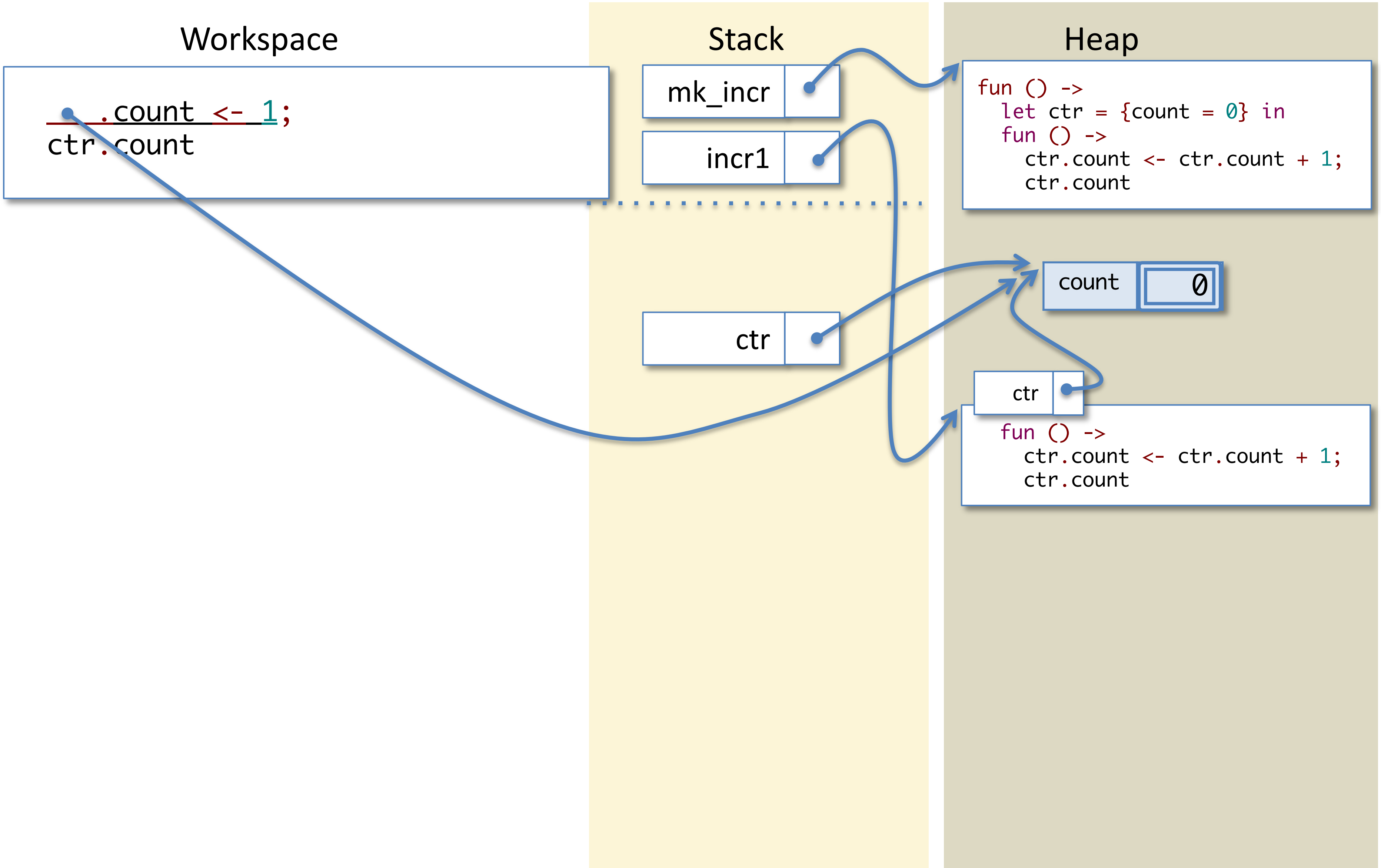
Now let's run "incr1 ()"



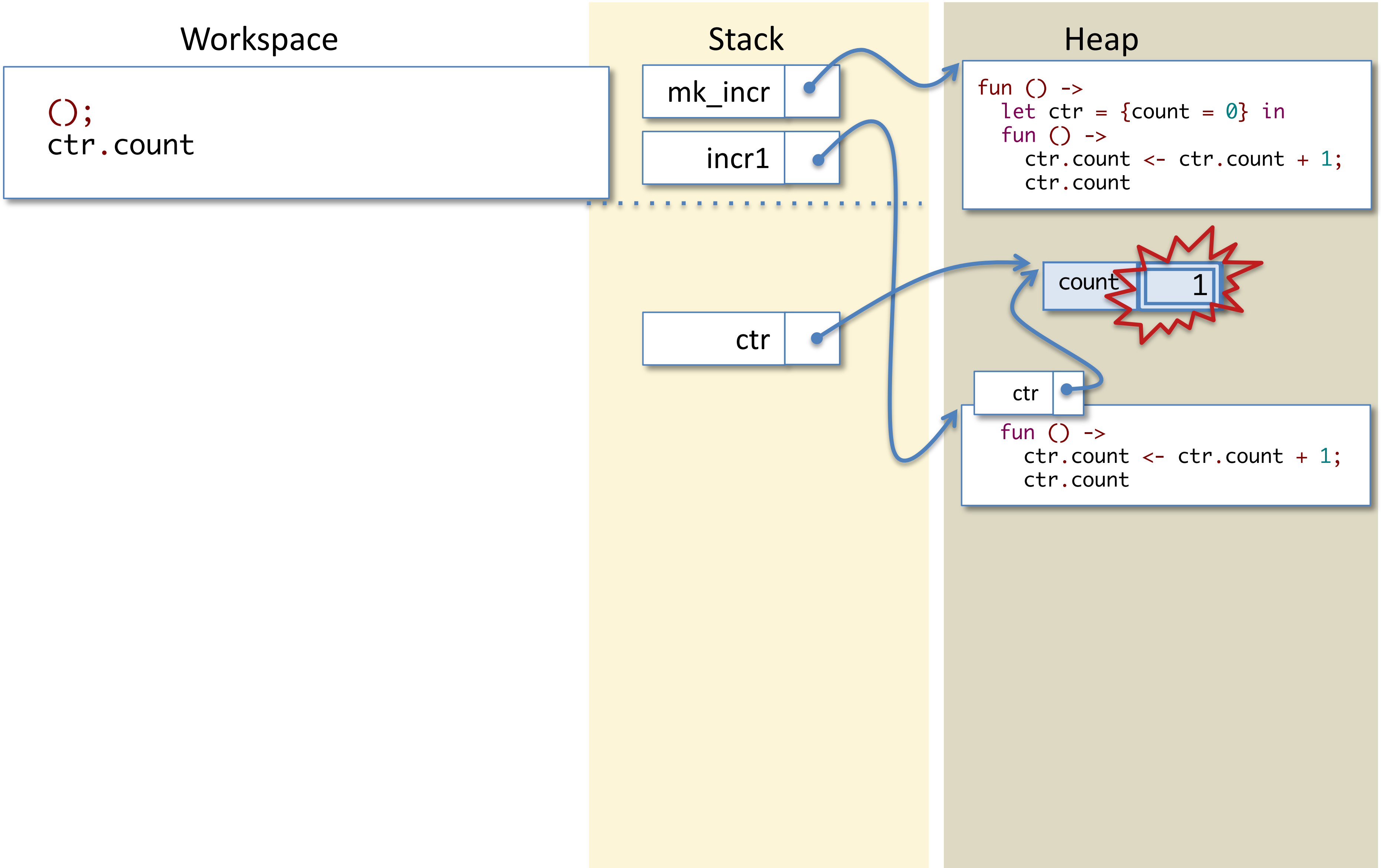
Now let's run "incr1 ()"



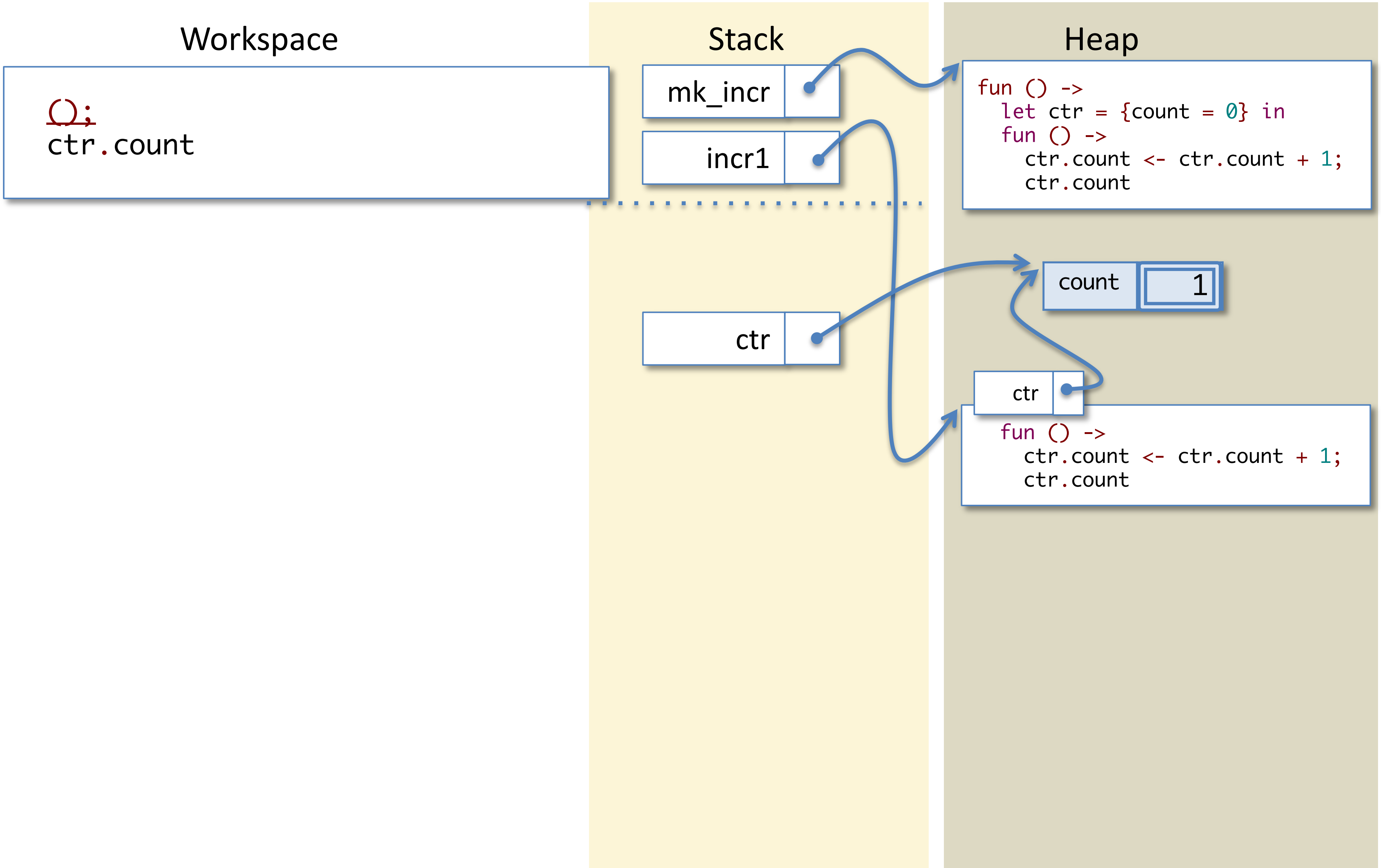
Now let's run "incr1 ()"



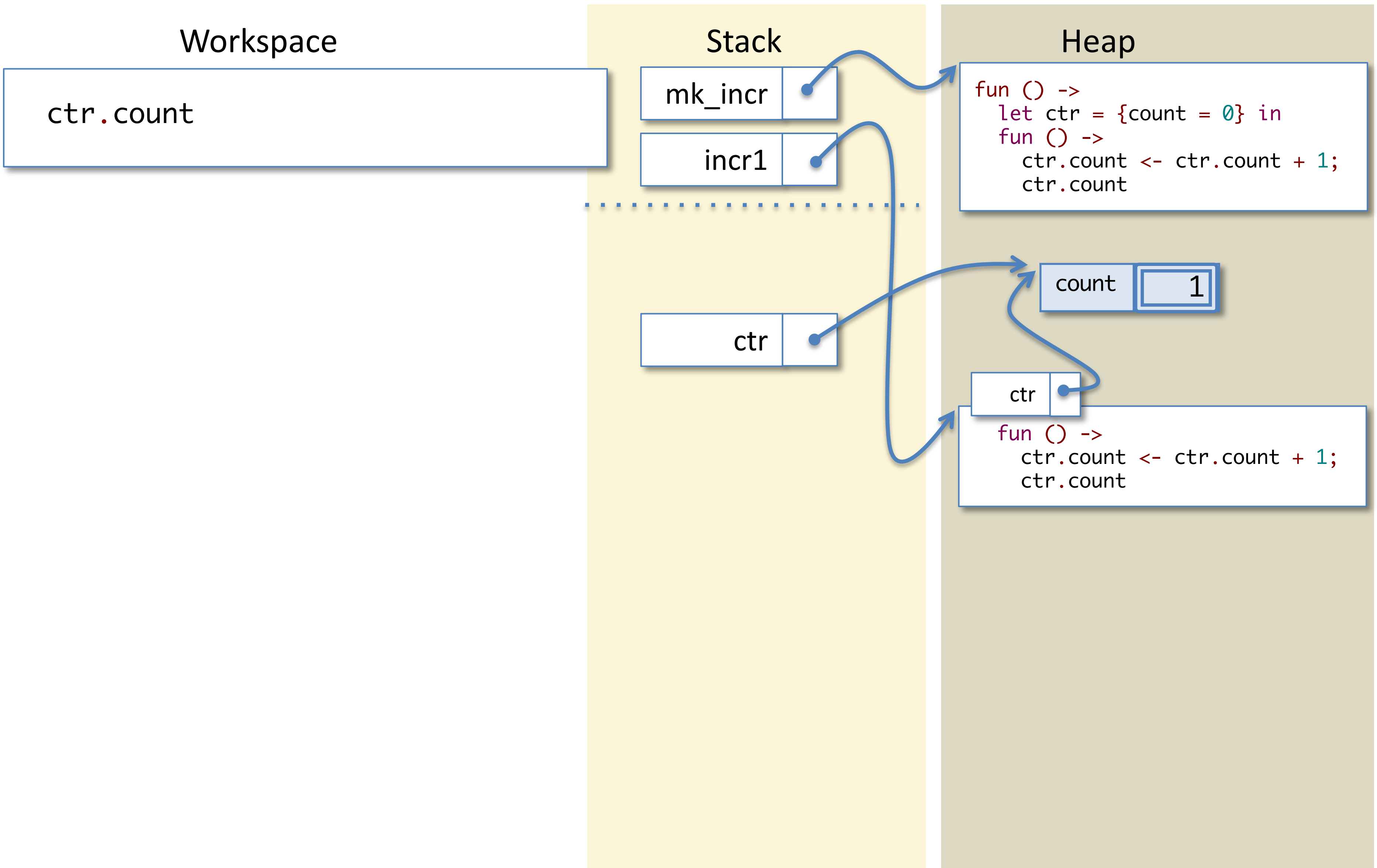
Now let's run "incr1 ()"



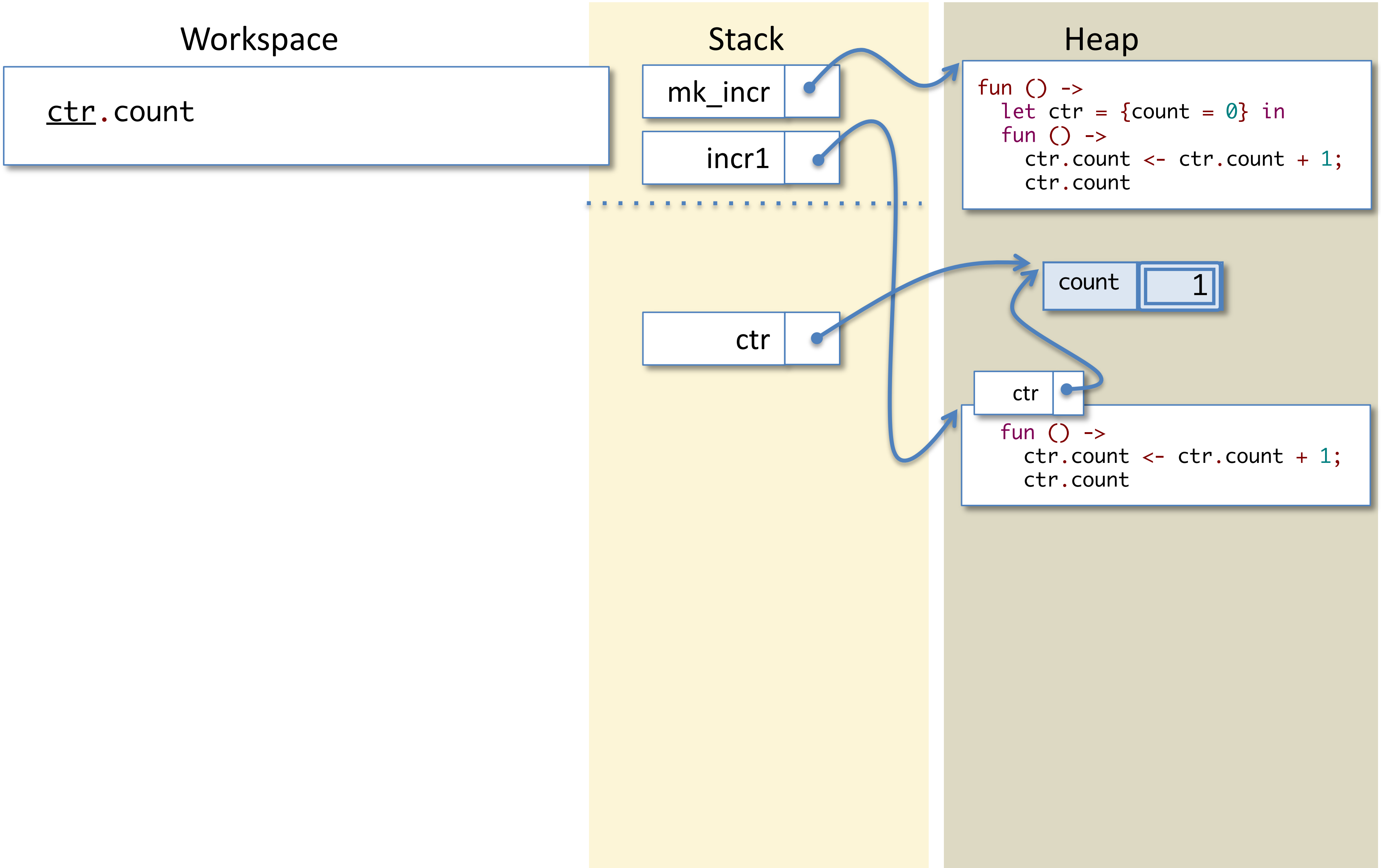
Now let's run "incr1 ()"



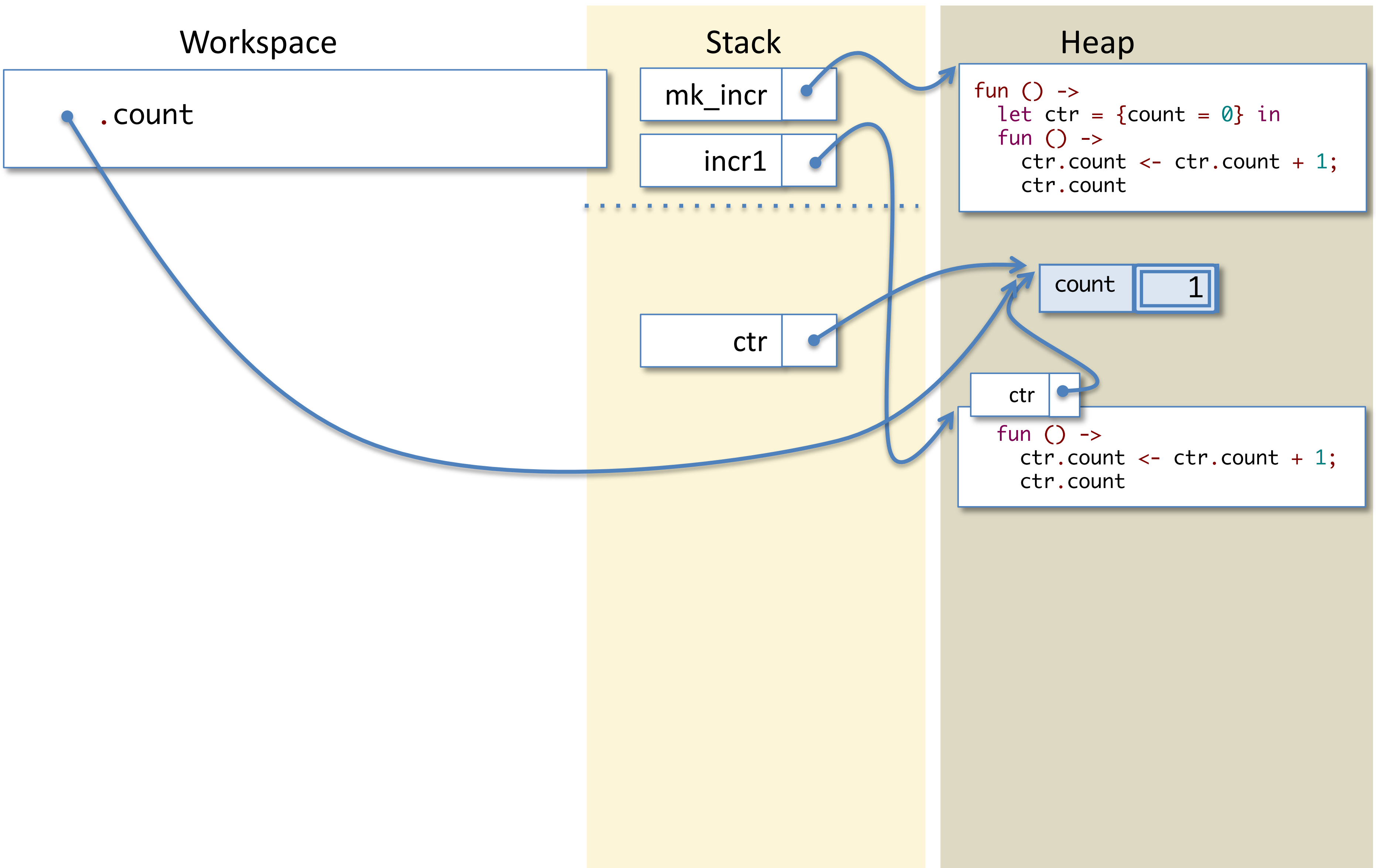
Now let's run "incr1 ()"



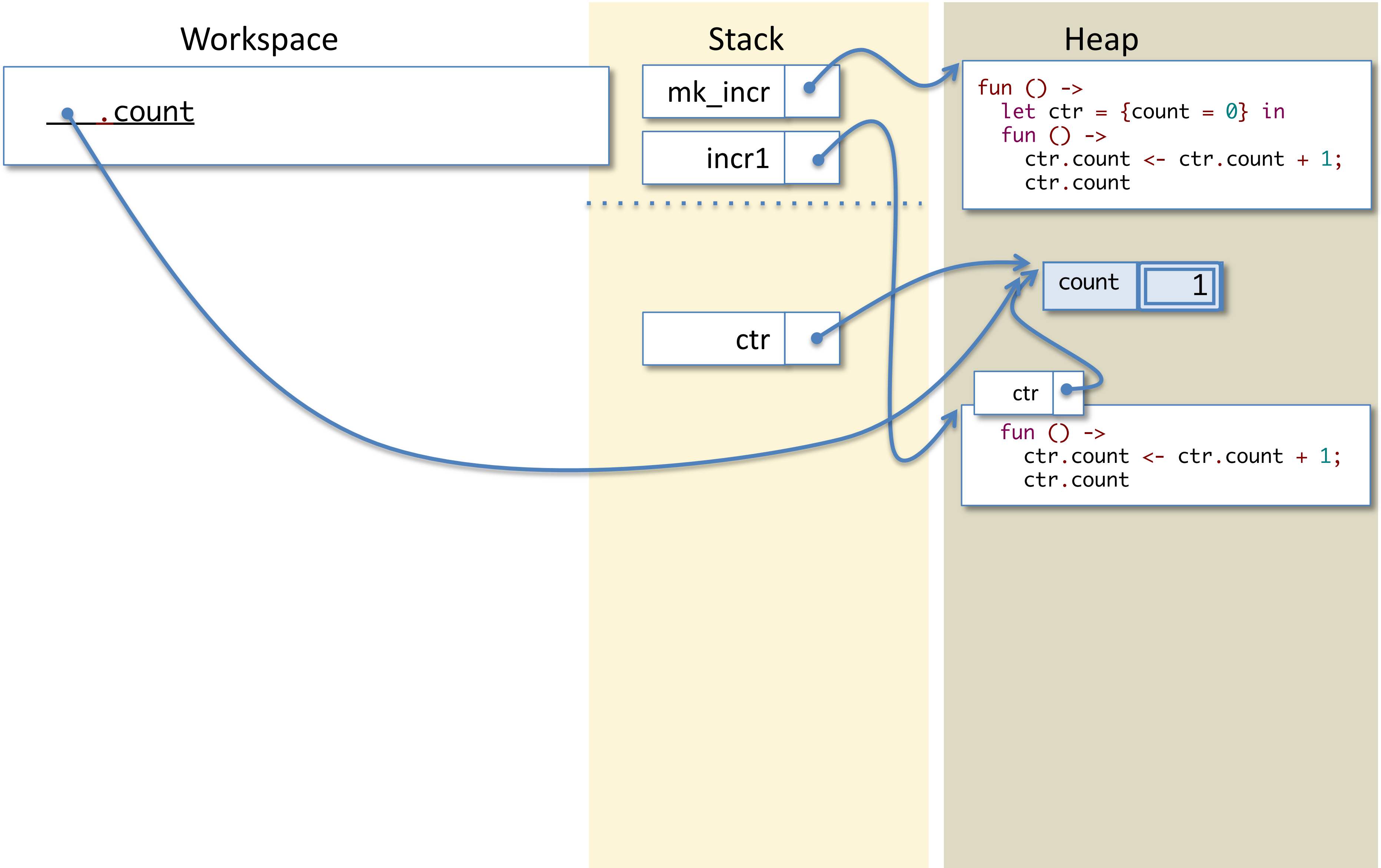
Now let's run "incr1 ()"



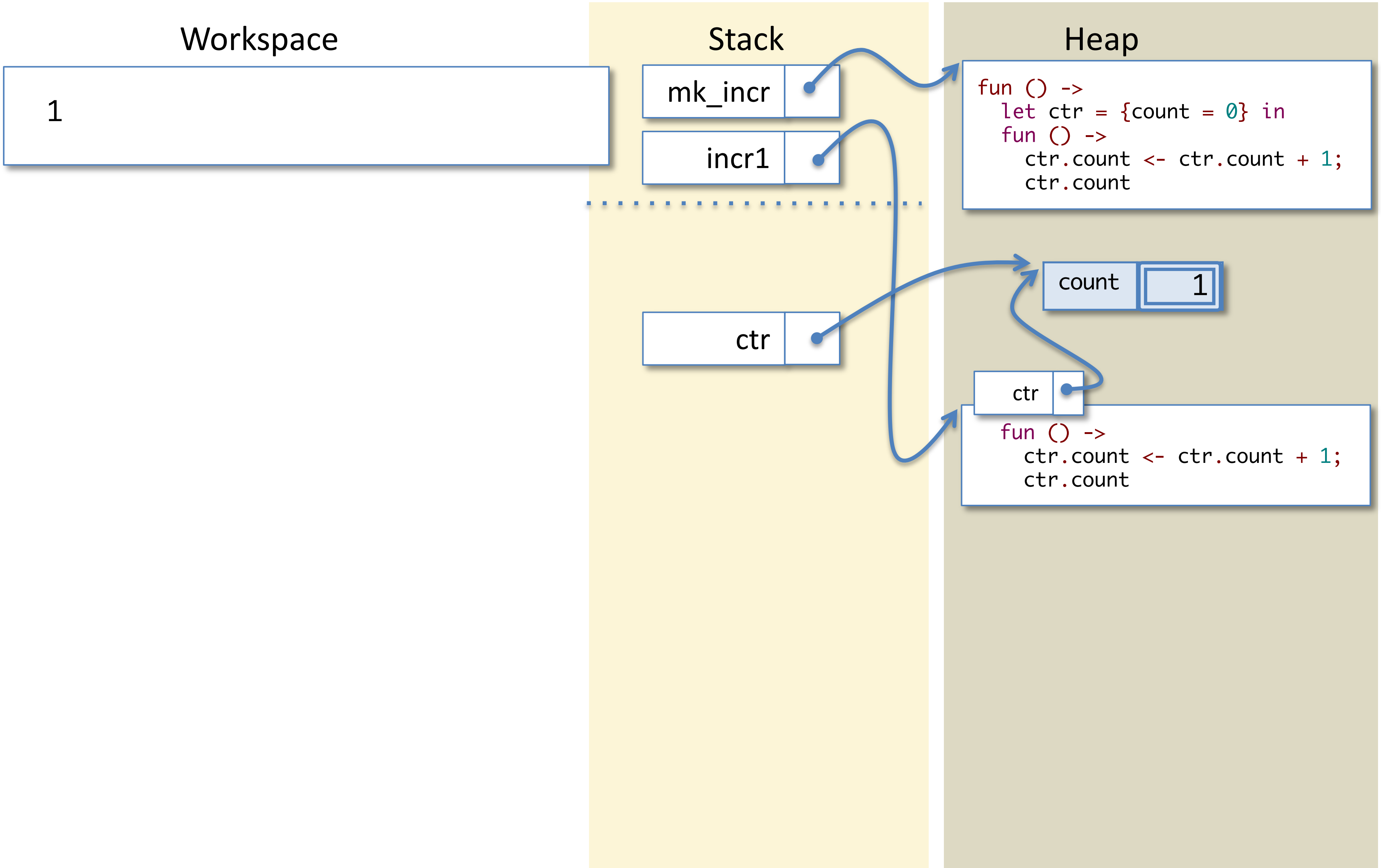
Now let's run "incr1 ()"



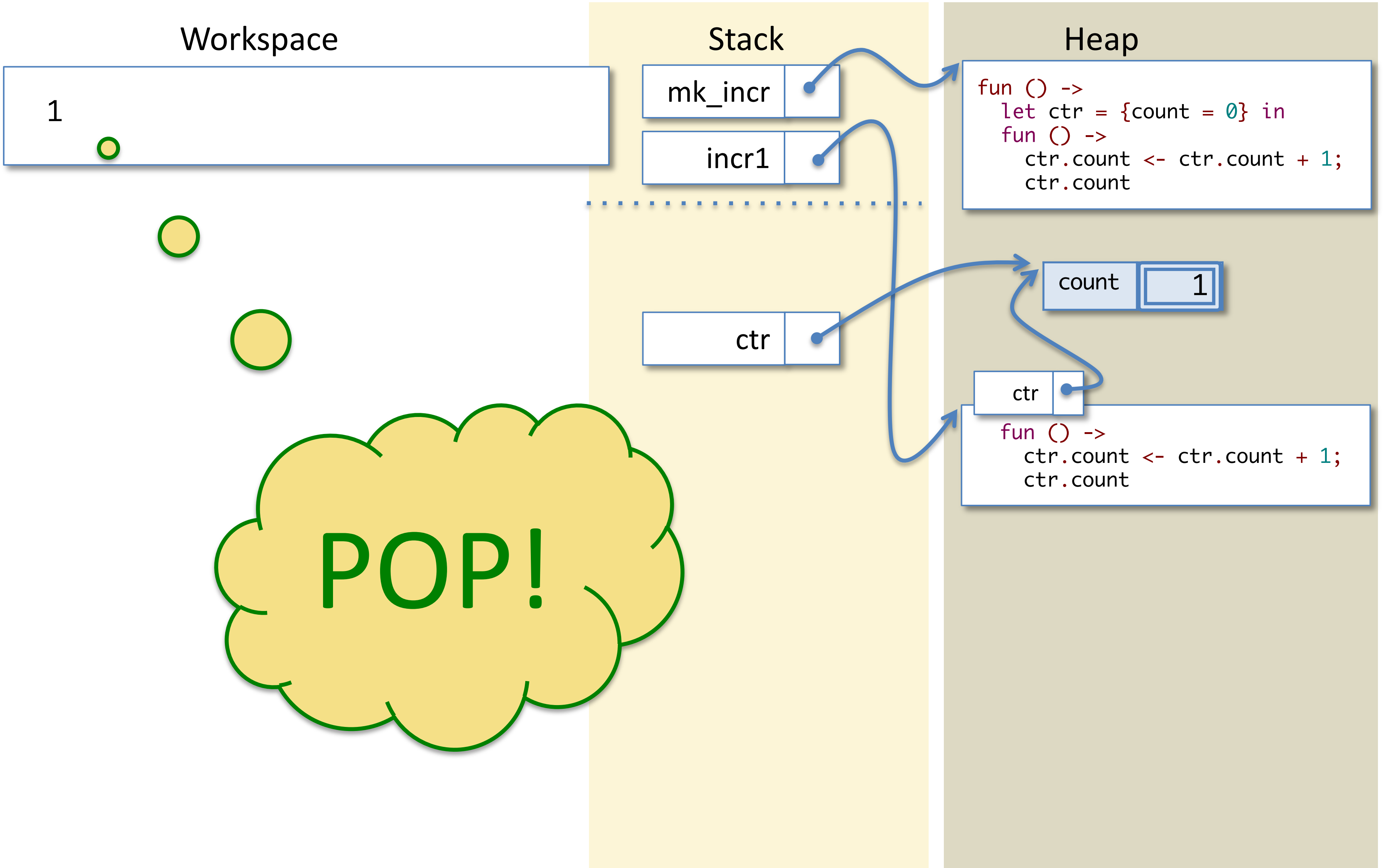
Now let's run "incr1 ()"



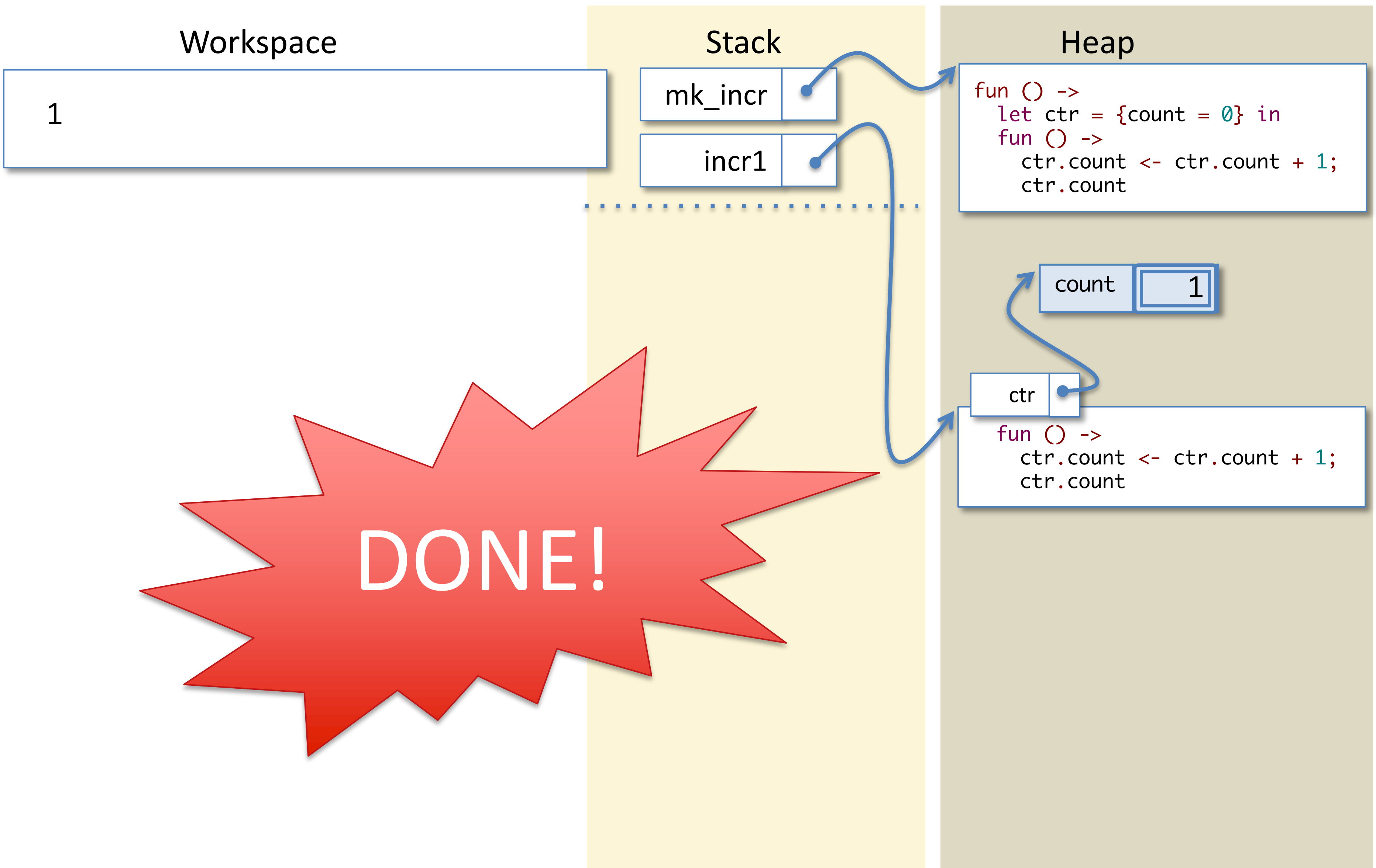
Now let's run "`incr1 ()`"



Now let's run "incr1 ()"



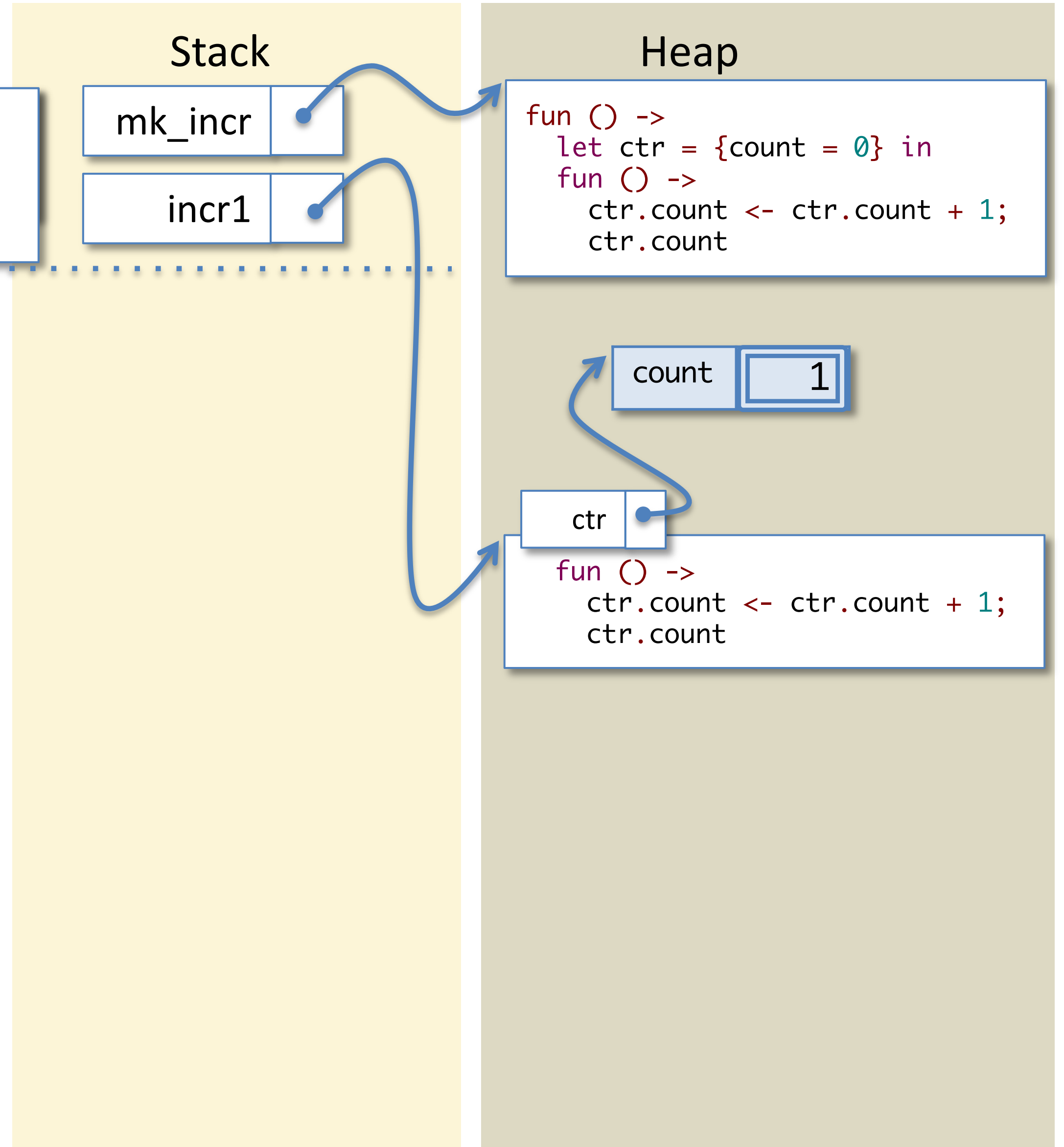
Now let's run "`incr1 ()`"



Now Let's run `mk_incr` again

Workspace

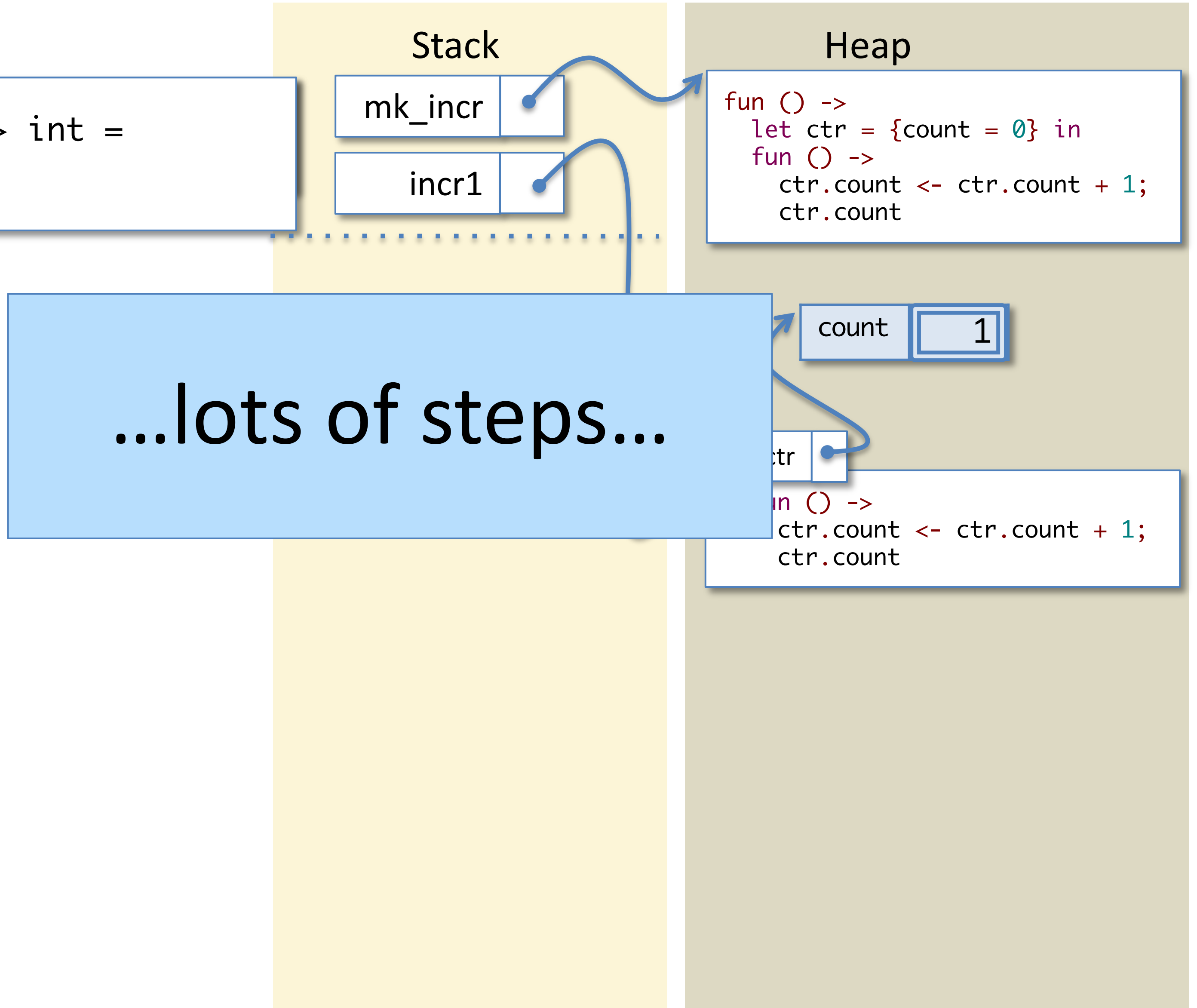
```
let incr2 : unit -> int =
mk_incr ()
```



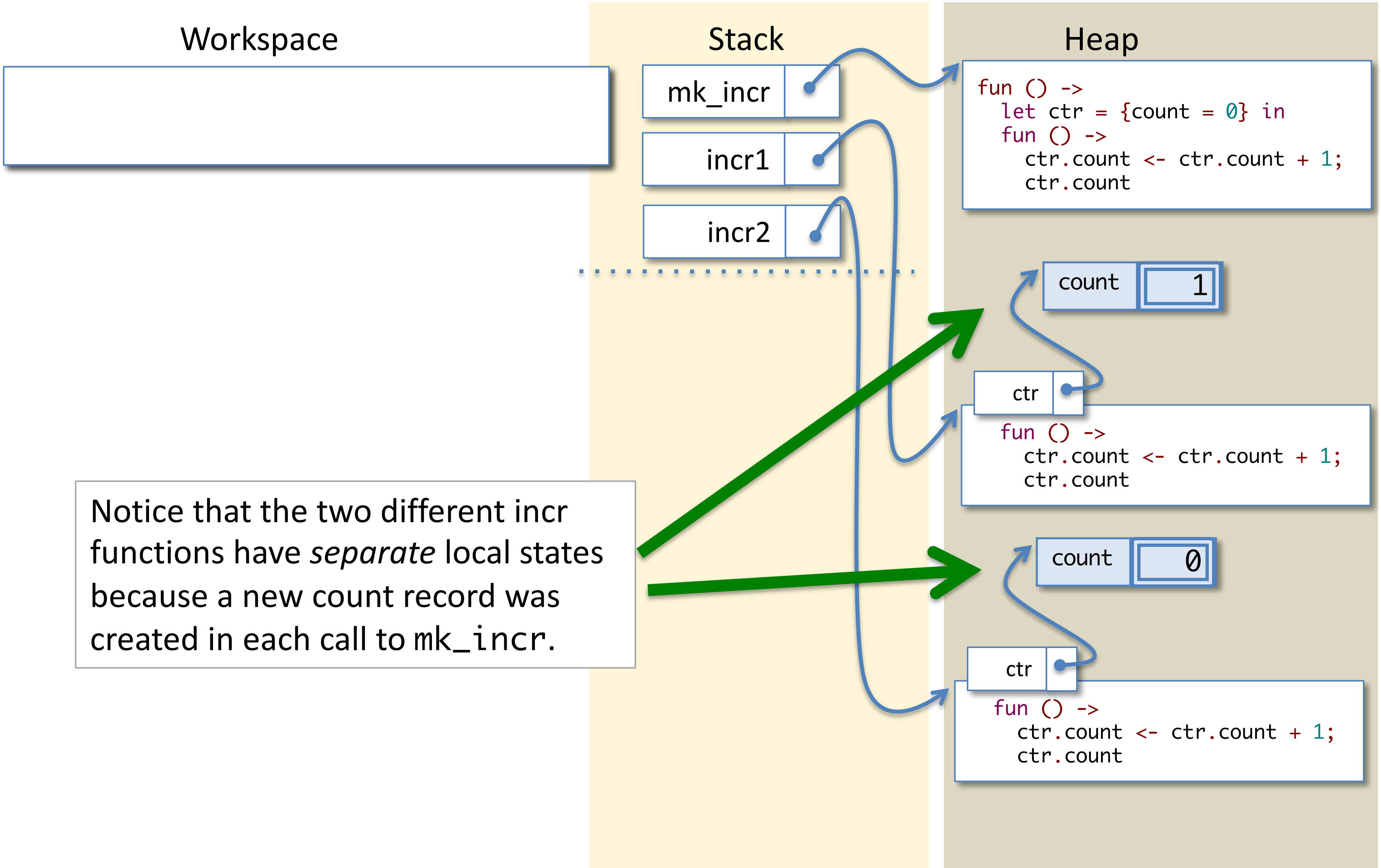
Now Let's run mk_incr again

Workspace

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



After creating `incr2`...

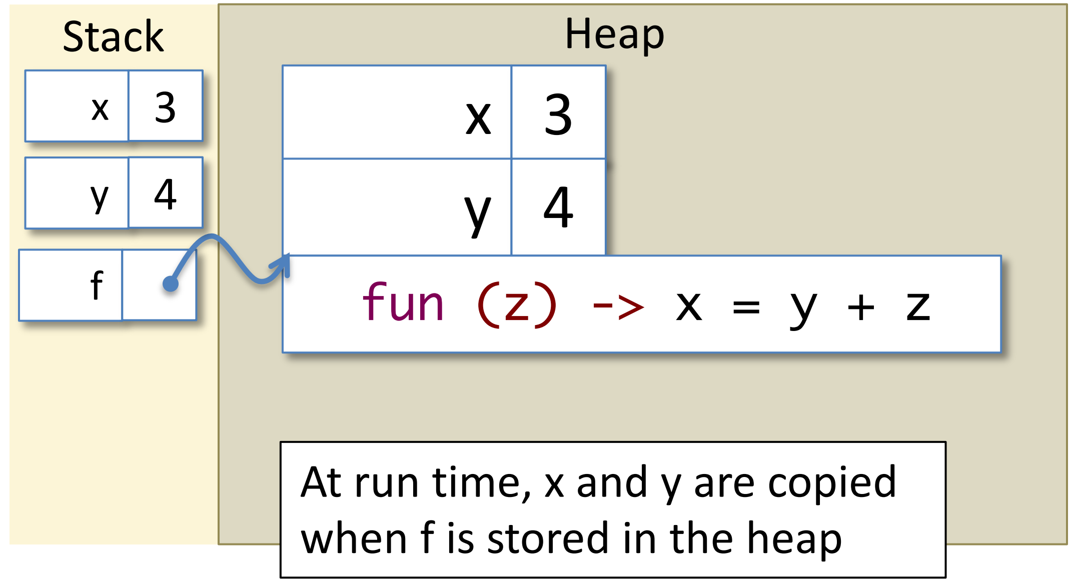


Notice that the two different incr functions have *separate* local states because a new count record was created in each call to `mk_incr`.

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:

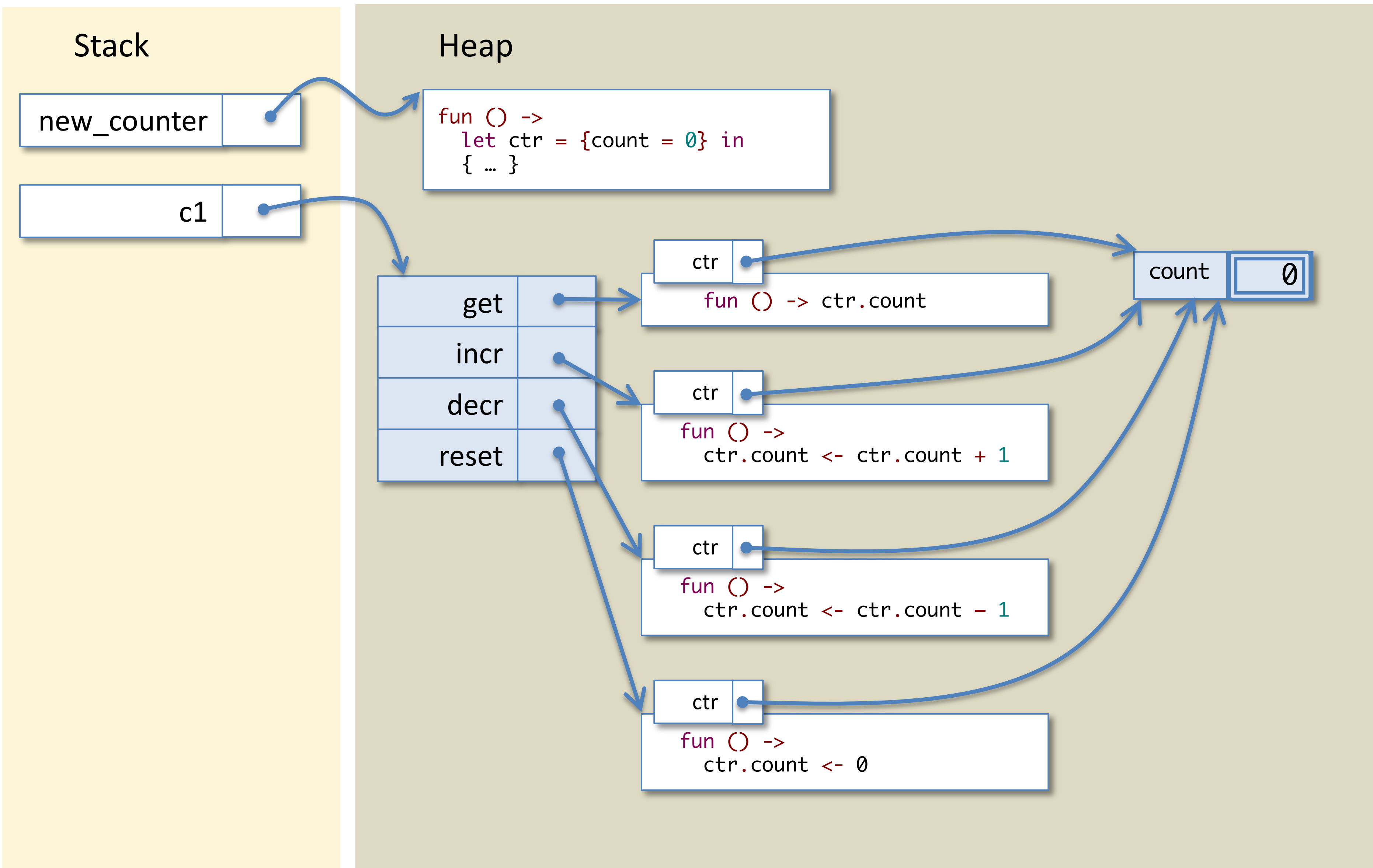
- encapsulated mutable state (*fields*)
- operations that manipulate that state (*methods*)

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

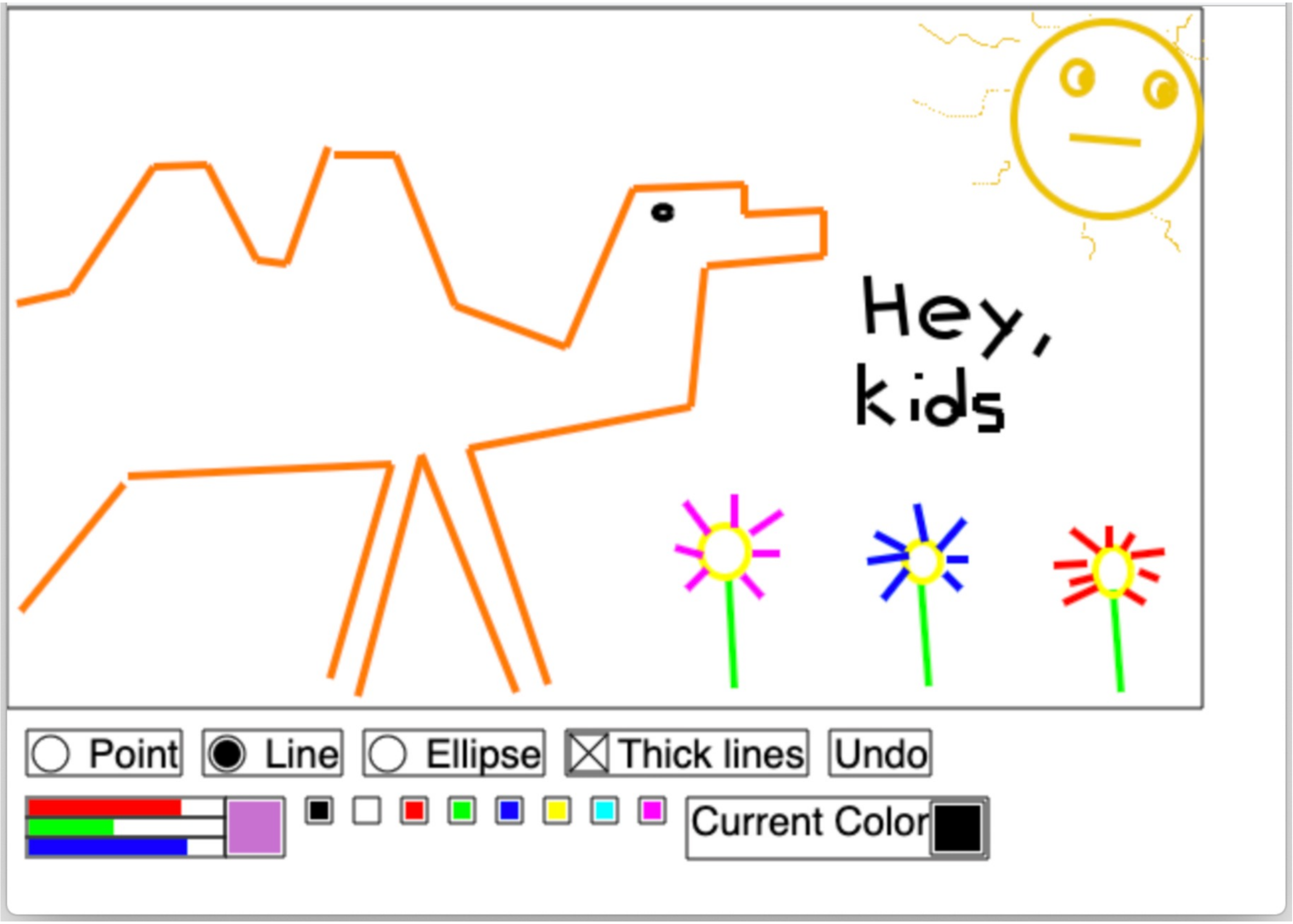
- Goals:
  - Practice with *first-class functions* and *hidden state (Ch 17)*
  - Bridge to object-oriented programming in Java
  - Illustrate the *event-driven programming* model
  - Give a feel for how GUI libraries (like Java's Swing) are put together
  - Apply everything we've seen so far to do some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

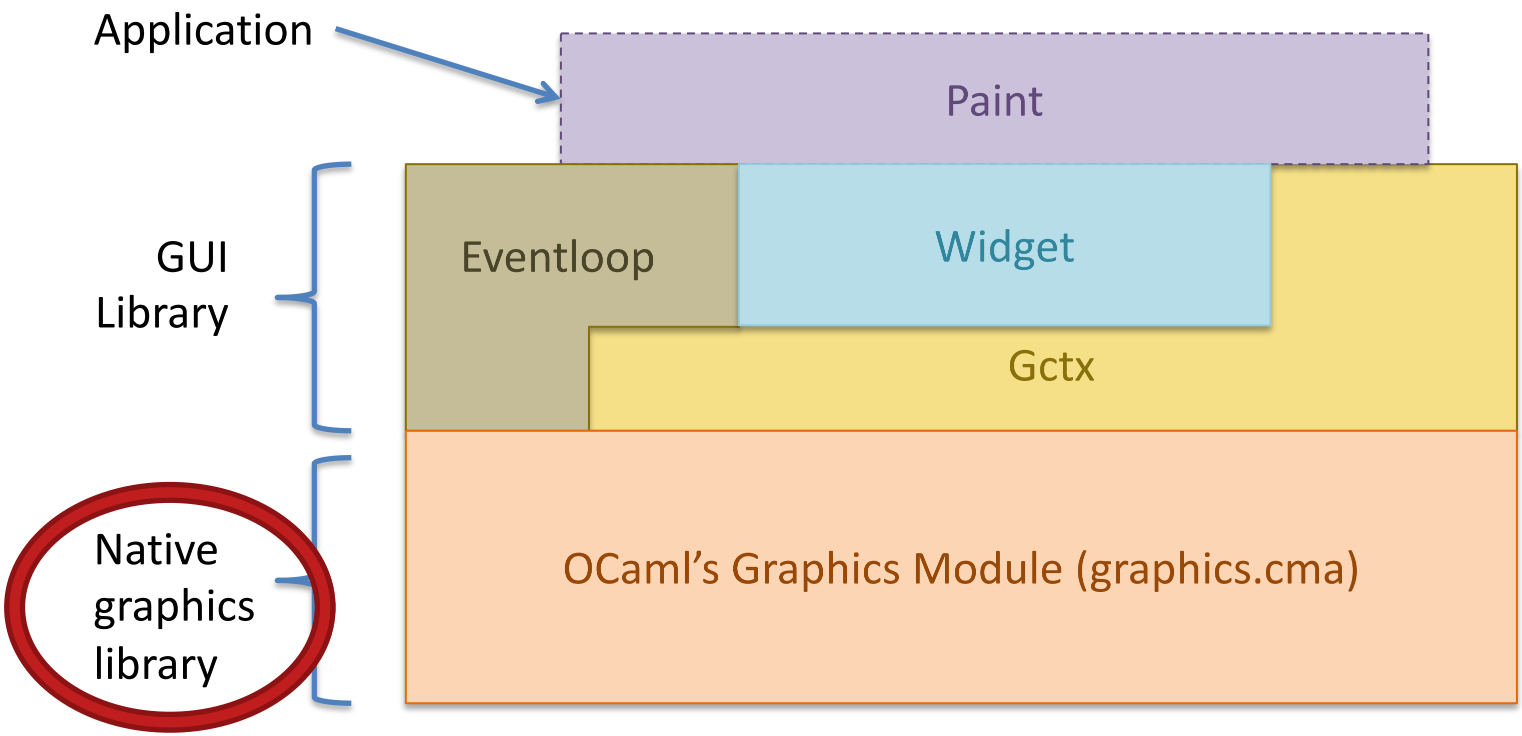


Step #1: Understand the Problem

- There are two separate parts of this homework: an *application* (Paint) and a *GUI library* (several files) used to build the application
- 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?
- Goal: The library should be *reusable*. It should be useful for other applications besides Paint.

Step #2, Interfaces: Project Architecture*

*program snippets will be 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.

Starting point: The low-level `Graphics` module

- OCaml's `Graphics` library provides *very basic* primitives for:
  - Creating an area in the screen for graphics
  - 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: https://ocaml.github.io/graphics/graphics/Graphics/

- How do we go from that to a full-blown GUI library?

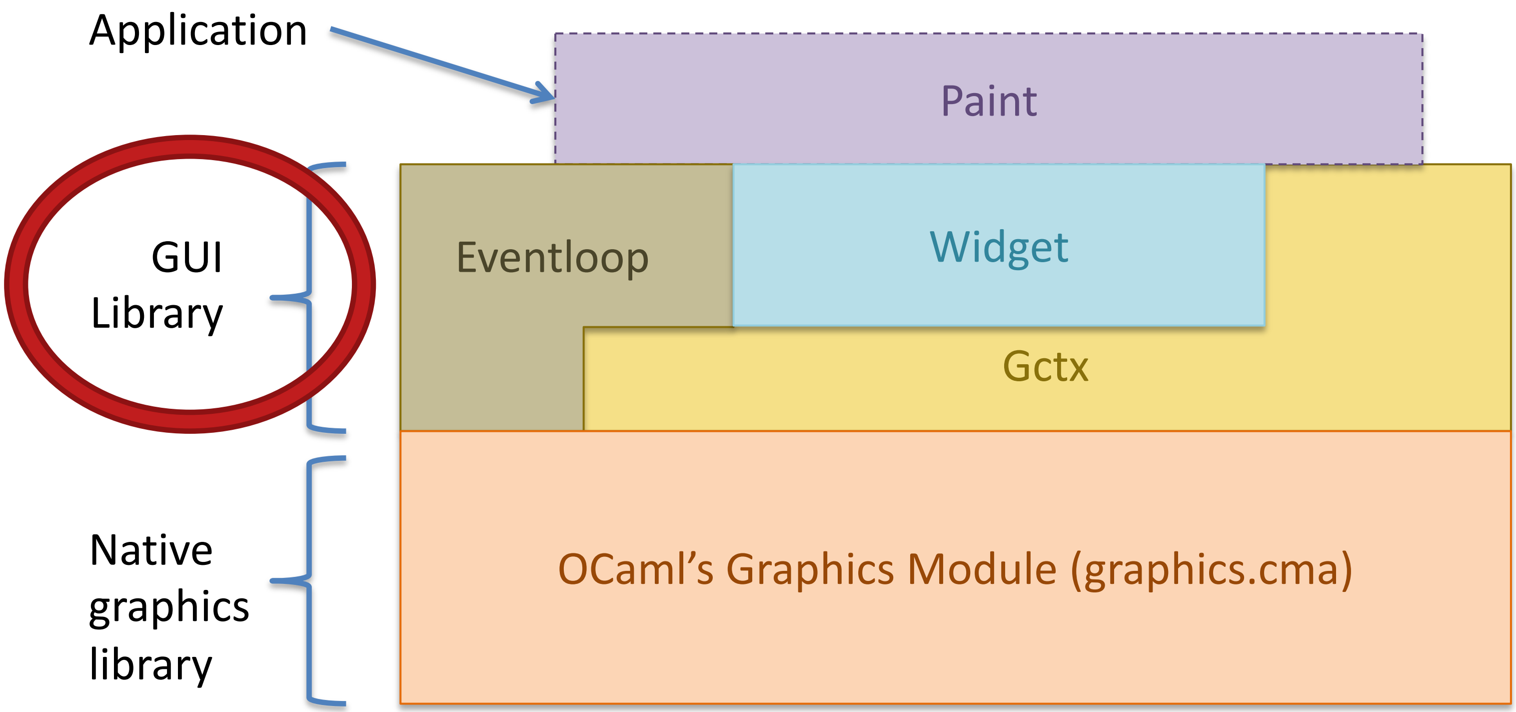
GUI Library Design

Abstractions for graphical interfaces

See: GUI Demo Code project on Codio

Interfaces: Project Architecture*

*program snippets will be 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.

Modules only call functions defined in libraries immediately below.

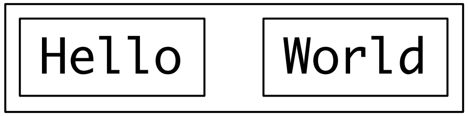
GUI terminology – Widget*

- Basic element of GUIs: examples include buttons, checkboxes, windows, textboxes, canvases, scrollbars, labels
- Every widget
  - knows how to repaint itself
  - knows how to handle events like mouse clicks
  - can calculate its size (width * height)

Simplified!

```
type widget = {
  repaint: unit -> unit;
  handle: event -> unit;
  size: unit -> int*int
}
```

- May be composed of other sub-widgets, for laying out complex interfaces



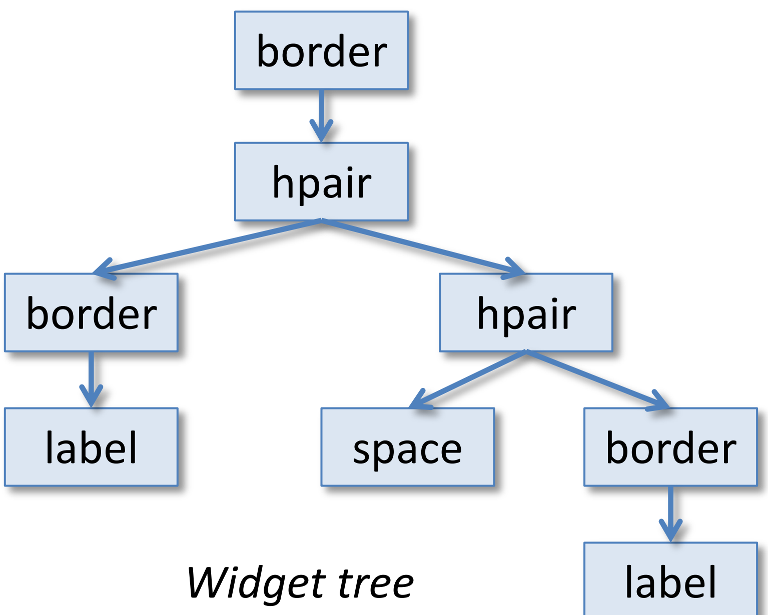
*Each GUI library uses its own naming convention for what we call "widgets." Java Swing calls them "Components"; iOS UIKit calls them "UIViews"; WINAPI, GTK+, X11's widgets, etc....

A “Hello World” application

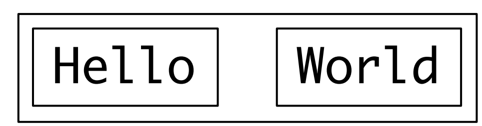
hello.ml

```
(* Create some simple label (string) widgets *)
let l1 : widget = label "Hello"
let l2 : widget = label "World"

(* Compose them horizontally, adding some borders *)
let h : widget =
    border (hpair (border l1)
                  (hpair (space (10,10)) (border l2)))
```



Widget tree



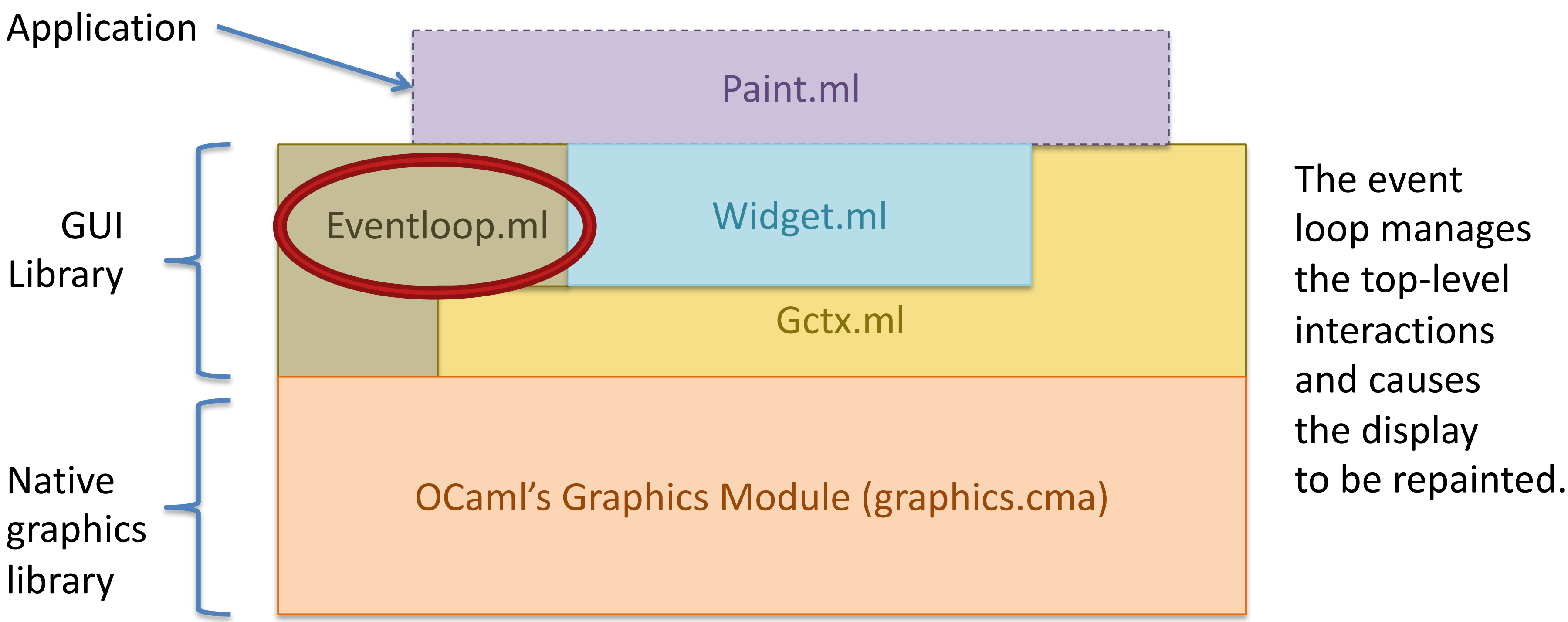
On the screen

Module: EventLoop

Top-level driver

GUI Architecture

- The eventloop is the main "driver" of a GUI application
  - For now: focus on how widgets are drawn on the screen
  - Later: deal with event handling



GUI terminology: “event loop”

- Main loop for all GUI applications (simplified)
  - “run” function takes top-level widget w as argument, containing all other widgets in the application.

```
let run (w:widget) : unit =
  w.repaint () ;            …draw the widget the first time
  Graphics.loop             …wait for user input (mouse click, etc)
    (fun e ->
       clear_graph ();
       w.handle e;          …inform widget about the event…
       w.repaint ()         …update the widget's appearance…
    )
```

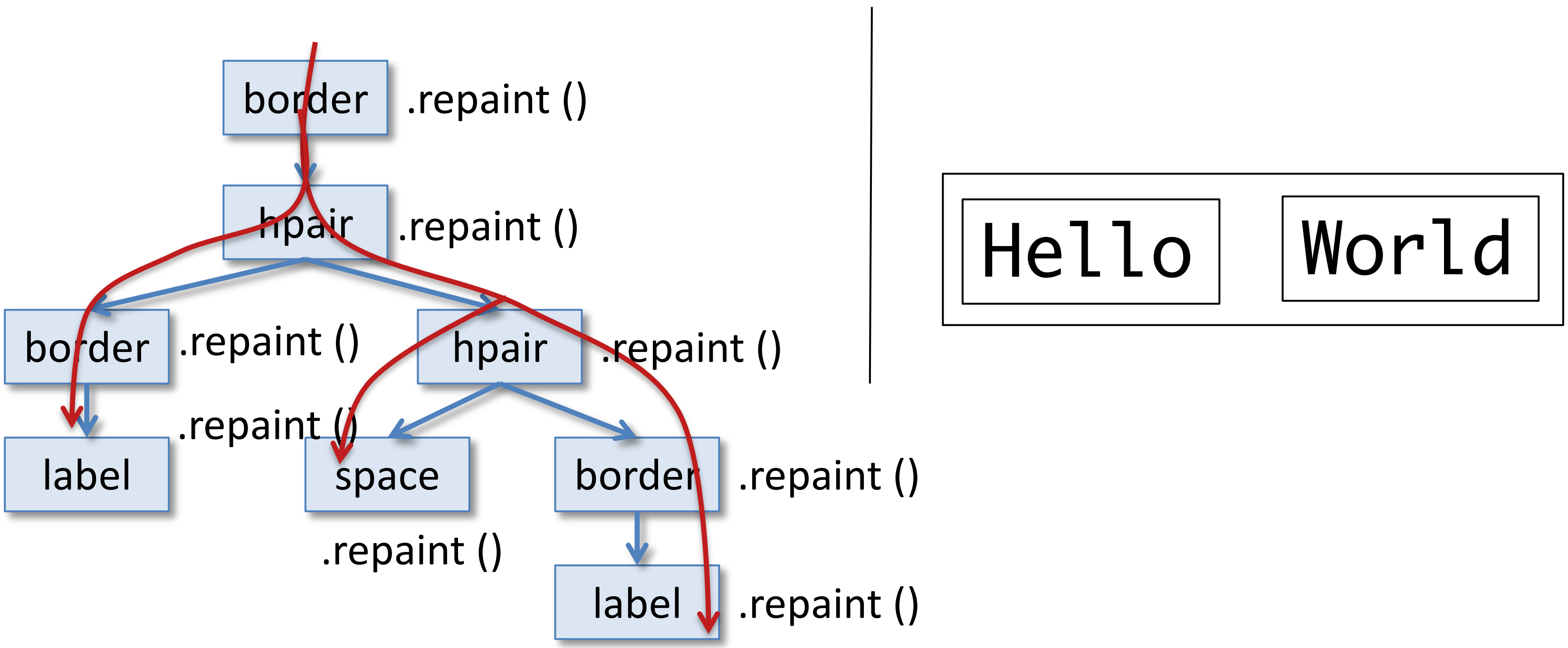
Eventloop (simplified)

```
let rec loop (f: event -> unit) : unit =
  let e = wait_next_event () in
  f e;
  loop f
```

Graphics

Drawing: Containers

Container widgets propagate `repaint` *commands to their children:*



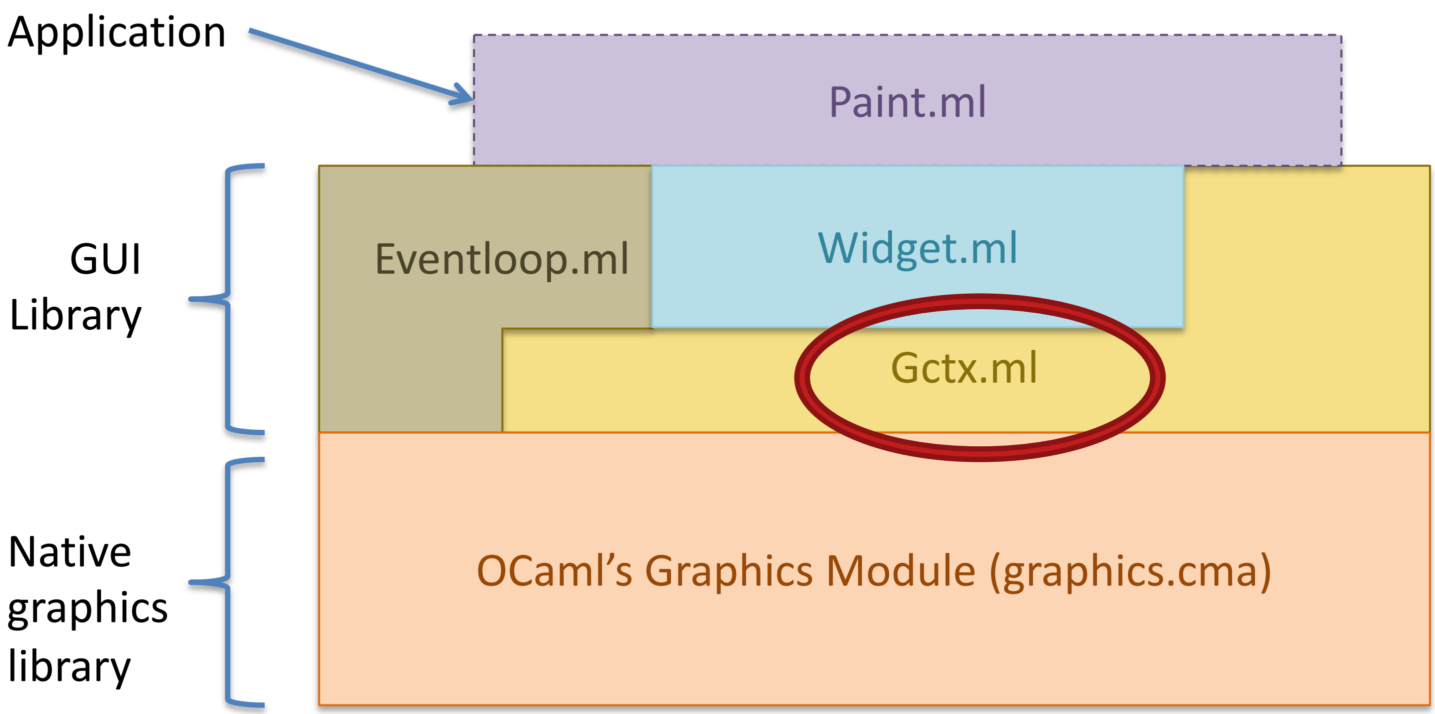
Challenge: The label widget's repaint function draws text in two different places. How can we make this code *location independent*?

Module: Gctx

“Contextualizes” graphics operations

Challenge: Widget Layout

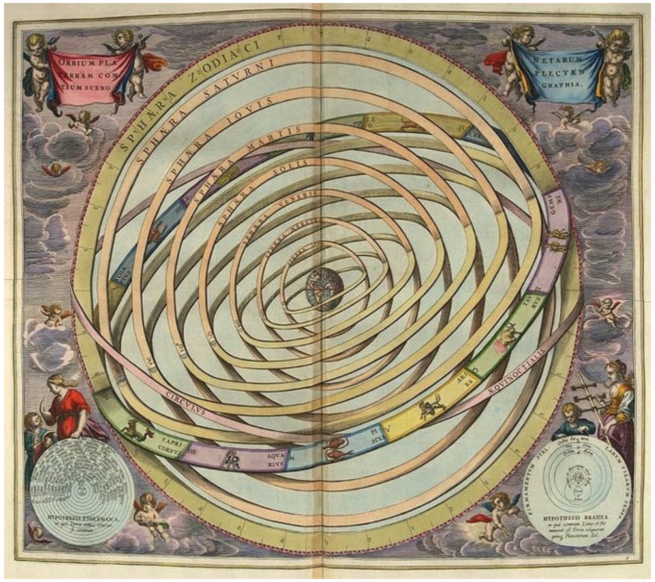
- Widgets are "things drawn on the screen". How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to a widget's current position



The graphics context isolates the widgets from the Graphics module.

GUI terminology – Graphics Context

- Translates coordinates
  - *Flips* between OCaml and "standard" coordinates so origin is top-left
  - *Translates* coordinates so all widgets can pretend that they are at the origin



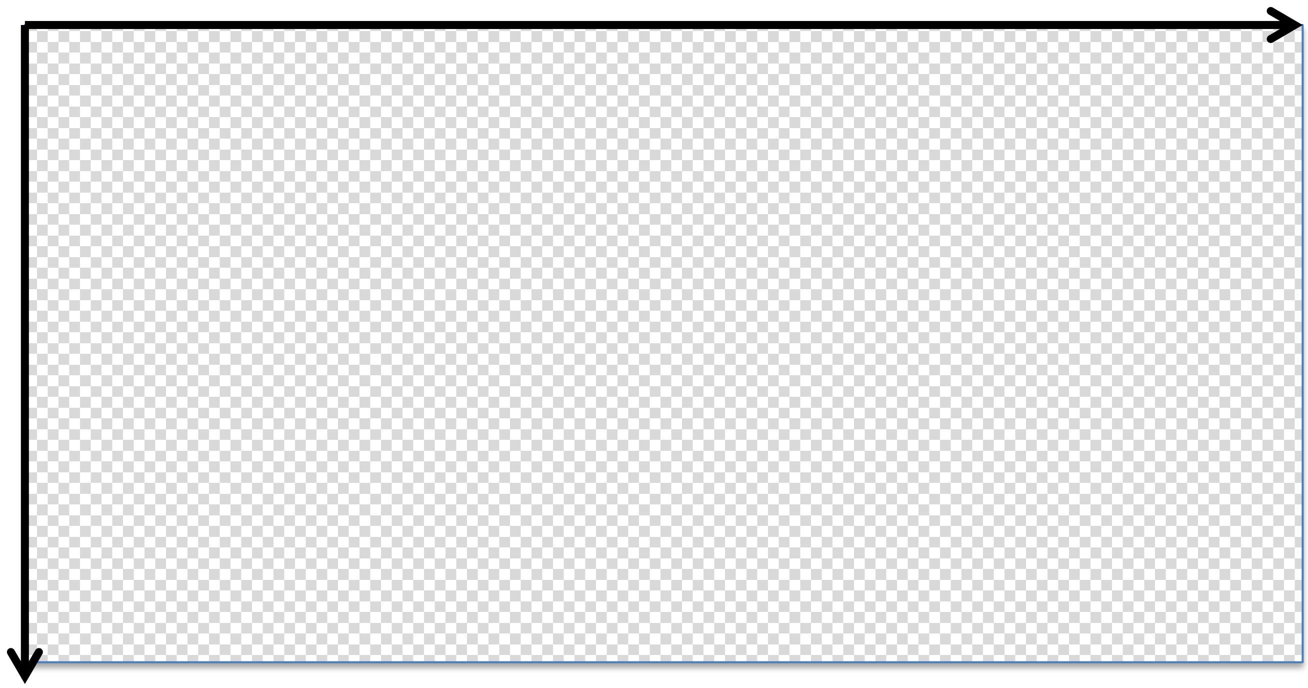
- Also carries information about the way things should be drawn
  - color
  - line width
- "Task 0" in the homework helps you understand the interaction between Gctx and OCaml's Graphics module

Graphics Contexts

This top box is a picture of the whole window.

```
let top  = Gctx.top_level in
```

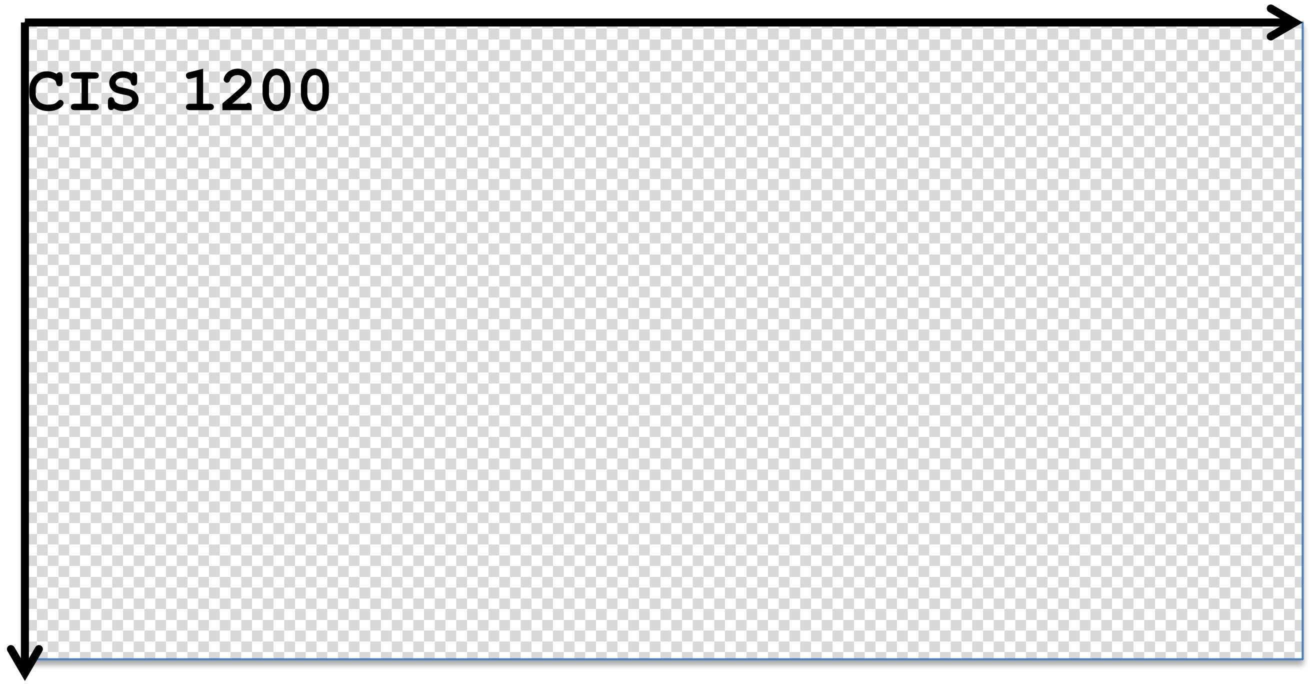
Graphics Contexts



```
let top  = Gctx.top_level
```

The top graphics context represents a coordinate system anchored at (0,0), with current pen color of black.

Graphics Contexts




```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"
```

Drawing a string at (0,10) in this context positions it on the left edge and 10 pixels down.
The string is drawn in black.

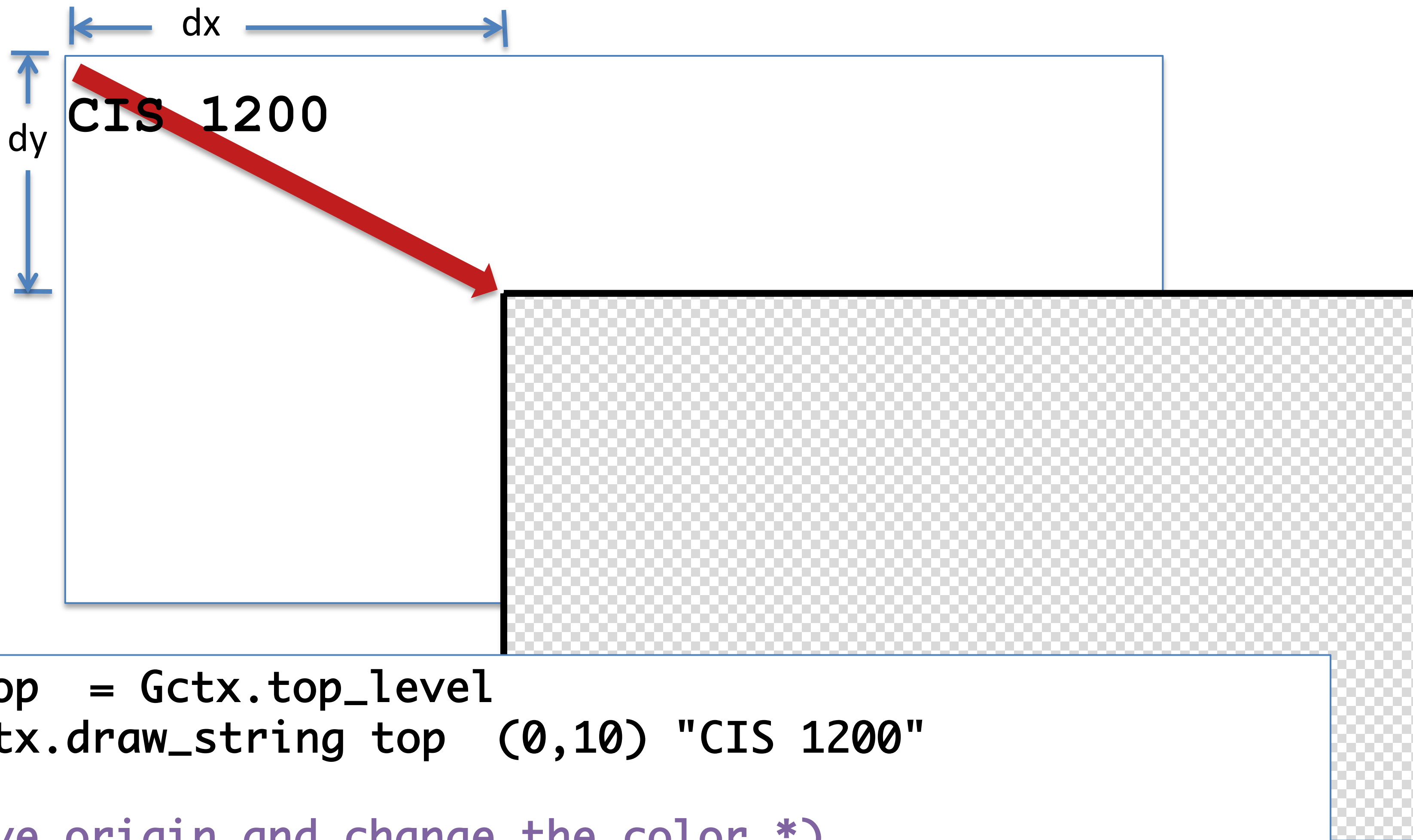
Graphics Contexts


```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 120(

(* move origin and change the color *)
let nctx = Gctx.with_color
              (Gctx.translate top (dx,dy)) red
```

Translating the gctx has the effect of shifting the origin relative to the old origin.

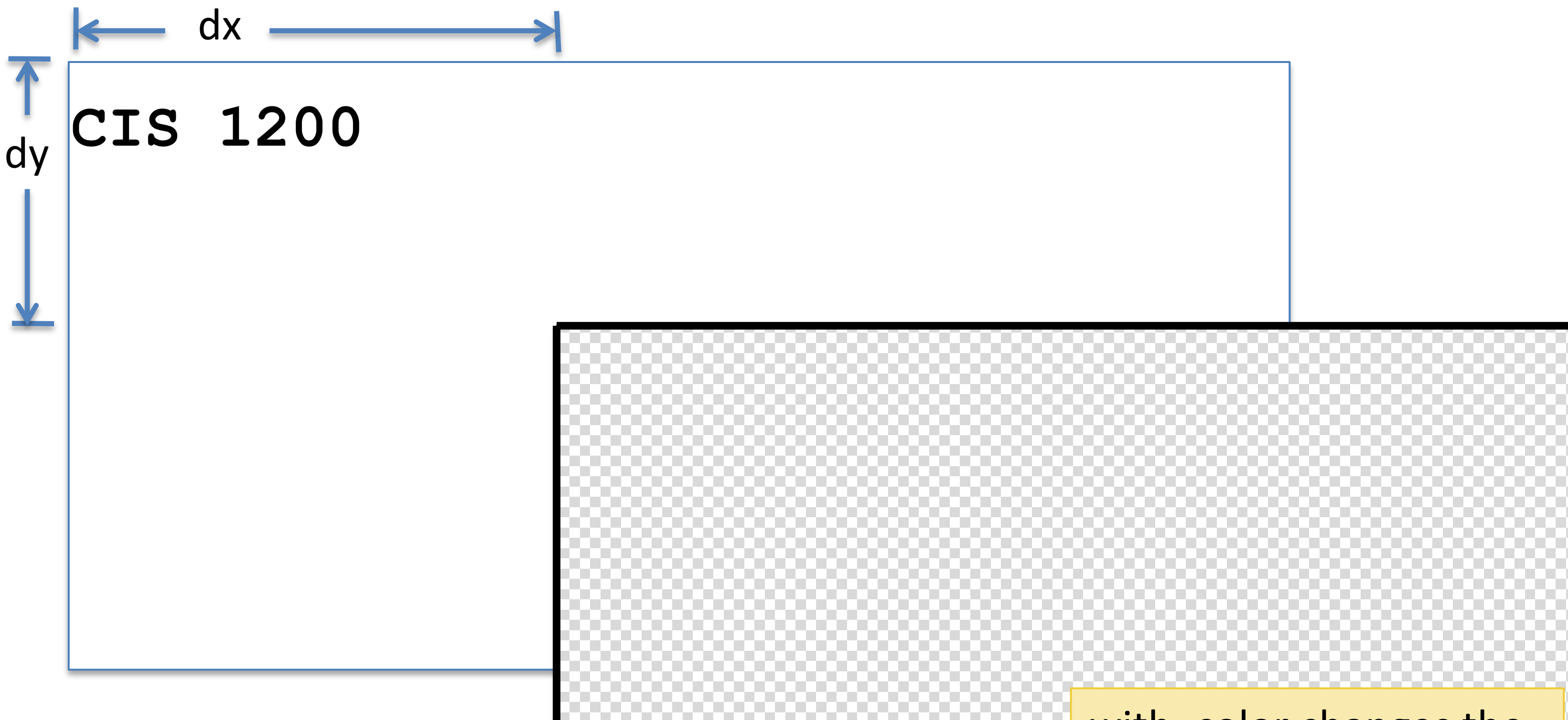
Graphics Contexts




```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"

(* move origin and change the color *)
let nctx = Gctx.with_color
              (Gctx.translate top (dx,dy)) red
```

Graphics Contexts

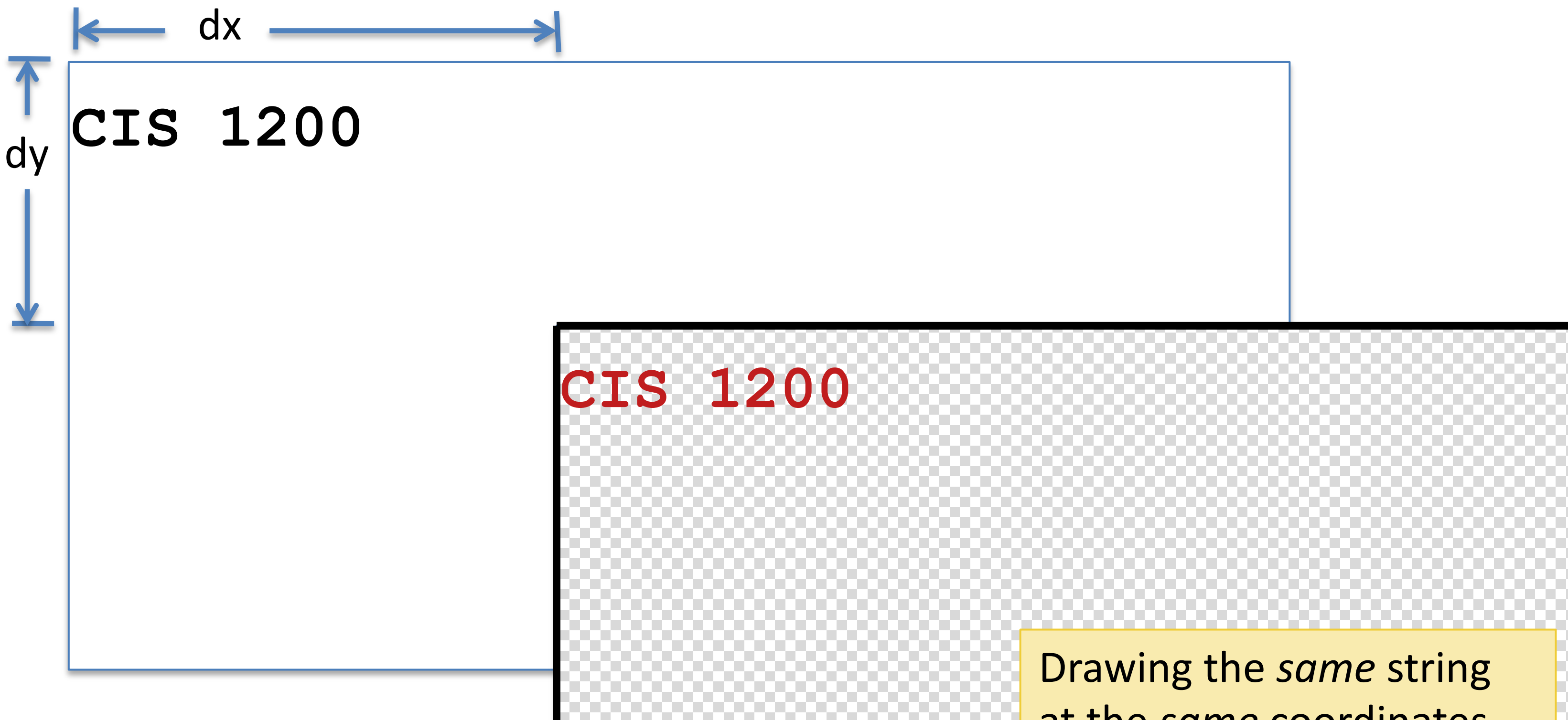


with_color changes the current drawing color…

```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"

(* move origin and change the color *)
let nctx = Gctx.with_color
              (Gctx.translate top (dx,dy)) red
```

Graphics Contexts



Drawing the *same* string at the *same* coordinates in the new context causes it to display at a translated location and in the new color.

```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"

(* move origin and change the color *)
let nctx = Gctx.with_color
               (Gctx.translate top (dx,dy)) red
;; Gctx.draw_string nctx (0,10) "CIS 1200"
```

Graphics Contexts

The graphics contexts aren't displayed anywhere, they only serve as frames of reference…

```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200

(* move origin and change the color *)
let nctx = Gctx.with_color
               (Gctx.translate top (dx,dy)) red
;; Gctx.draw_string nctx (0,10) "CIS 1200"
```

18: Which of the following can we fill in for ??? to obtain the result shown?

0

Gctx.translate top (dx,0)

0%

Gctx.translate top (0,-dy)

0%

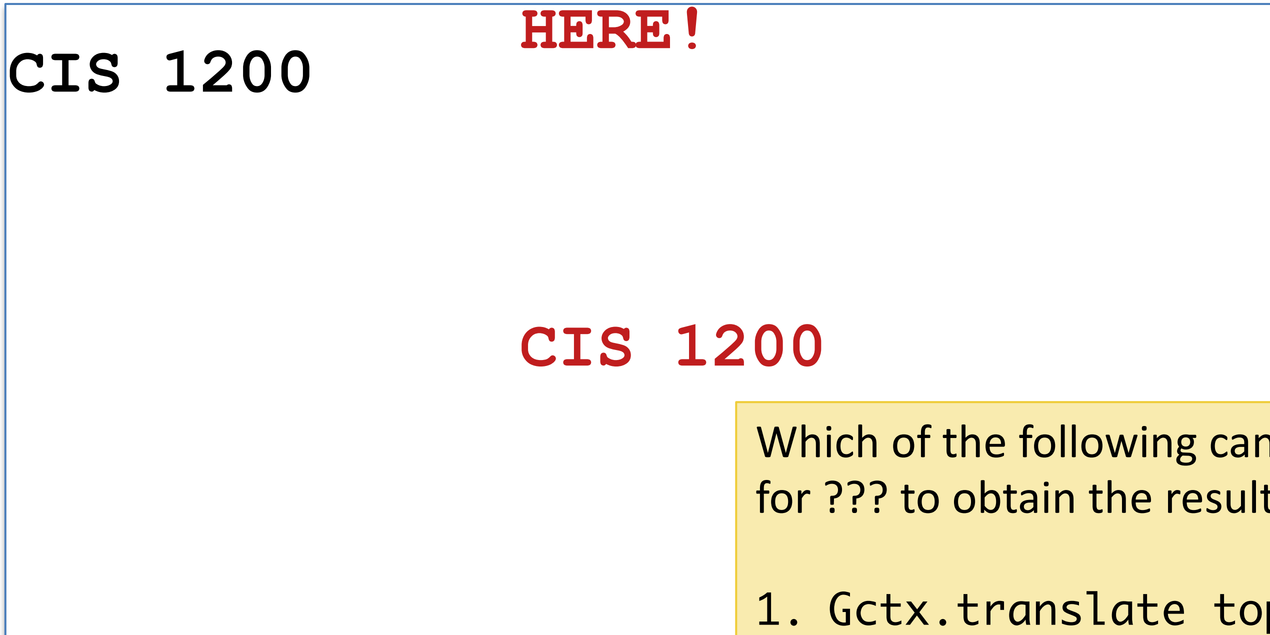
Gctx.translate nctx (dx,0)

0%

Gctx.translate nctx (0,-dy)

0%

Graphics Contexts



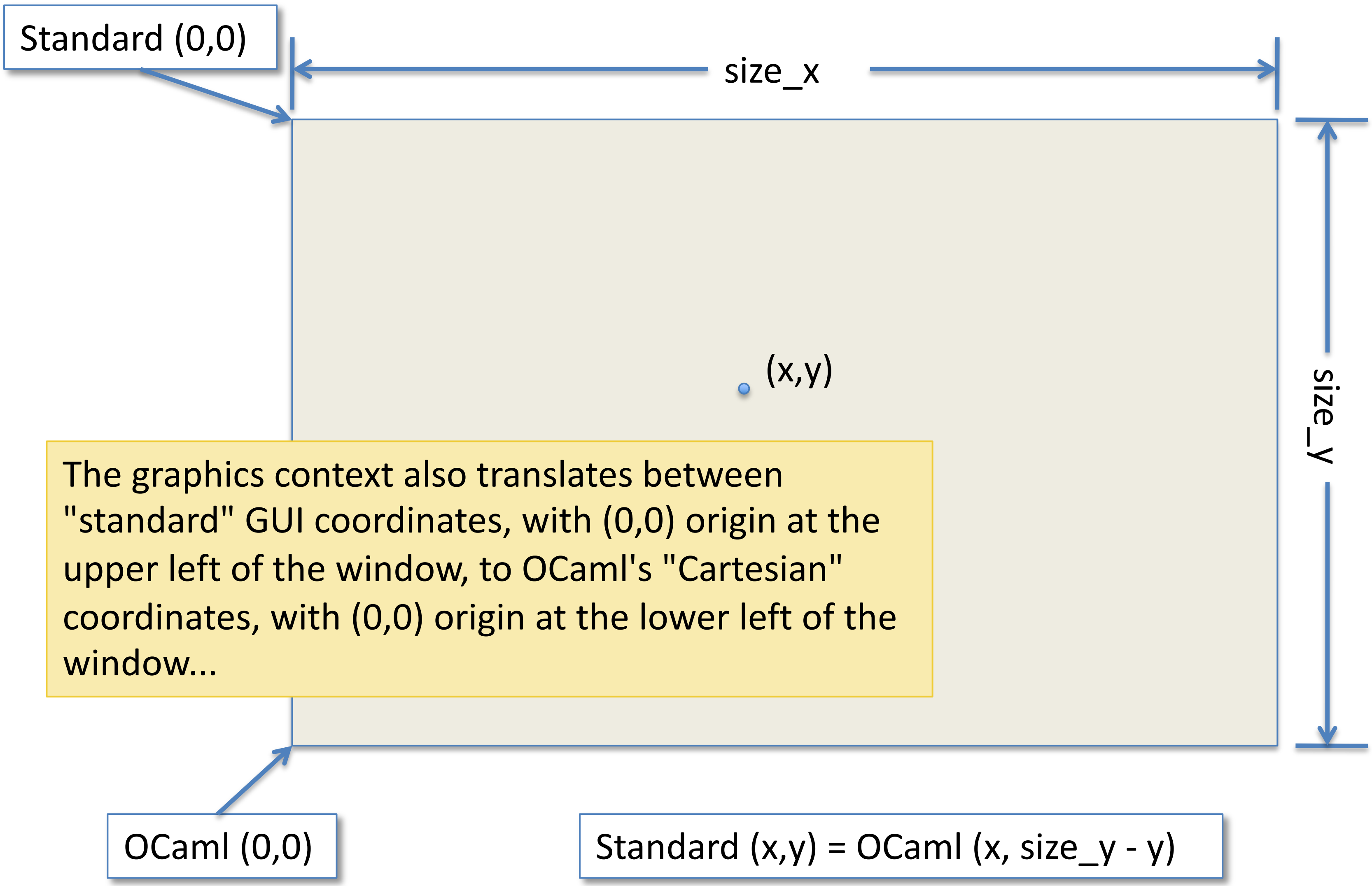
Which of the following can we fill in for ??? to obtain the result shown?

1. Gctx.translate top (dx,0)
2. Gctx.translate top (0,-dy)
3. Gctx.translate nctx (dx,0)
4. Gctx.translate nctx (0,-dy)

```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"
let nctx = Gctx.with_color
              (Gctx.translate top (dx,dy)) red
;; Gctx.draw_string nctx (0,10) "CIS 1200"
let ctx3 = ???
;; Gctx.draw_string ctx3 (0,0) "HERE!"
```

Answer: 4

OCaml vs. “Standard” Coordinates



The graphics context also translates between "standard" GUI coordinates, with (0,0) origin at the upper left of the window, to OCaml's "Cartesian" coordinates, with (0,0) origin at the lower left of the window...

Module Gctx

```
(** The main (abstract) type of graphics contexts. *)
type gctx

(** The top-level graphics context *)
val top_level : gctx

(** A widget-relative position *)
type position = int * int

(** Display text at the given (relative) position *)
val draw_string : gctx -> position -> string -> unit
(** Draw a line between the two specified positions *)
val draw_line : gctx -> position -> position -> unit

(** Produce a new gctx shifted by (dx,dy) *)
val translate : gctx -> int * int -> gctx
(** Produce a new gctx with a different pen color *)
val with_color : gctx -> color -> gctx
```

Widget Layout

Building blocks of GUI applications

see simpleWidget.ml in GUI Demo Code project

Simple Widgets

simpleWidget.mli

```
(* An interface for simple GUI widgets *)
type widget = {
   repaint : Gctx.gctx -> unit;
   size    : unit -> (int * int)
}
val label  : string -> widget
val space  : int * int -> widget
val border : widget -> widget
val hpair  : widget -> widget -> widget
val canvas : int * int -> (Gctx.gctx -> unit) -> widget
```

- You can ask a simple widget to repaint itself
  - Repainting is relative to a graphics context
- You can ask a simple widget to tell you its size
- (For now, we ignore event handling...)

Widget Examples

```
(* A simple widget that puts some text on the screen *)
let label (s:string) : widget =
{
  repaint = (fun (g:Gctx.gctx) -> Gctx.draw_string g (0,0) s);
  size = (fun () -> Gctx.text_size s)
}
```

simpleWidget.ml

```
(* A "blank" area widget -- it just takes up space *)
let space ((w,h):int*int) : widget =
{
  repaint = (fun (_:Gctx.gctx) -> ());
  size = (fun () -> (w,h))
}
```

simpleWidget.ml

The canvas Widget

- Region of the screen that can be drawn upon
- Has a fixed width and height
- Parameterized by a repaint method "r"
  - …which can directly use the Gctx drawing routines to draw on the canvas

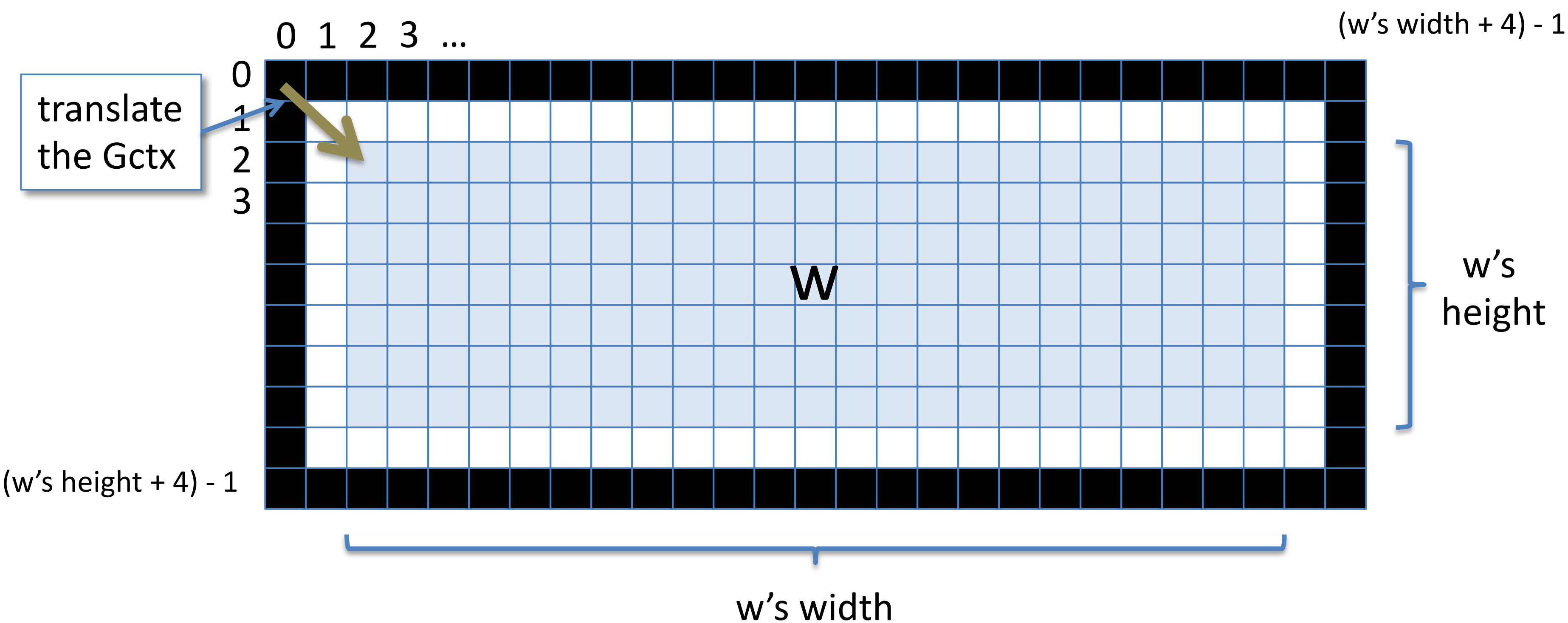
```
let canvas ((w,h):int*int) (r: Gctx.gctx -> unit) : widget =
{
  repaint = r;
  size = (fun () -> (w,h))
}
```

simpleWidget.ml

Nested Widgets

Containers and Composition

The Border Widget Container



- `let b = border w`
- Draws a one-pixel wide border around contained widget w
- b's size is slightly larger than w's (+4 pixels in each dimension)
- b's repaint method must call w's repaint method
- When b asks w to repaint, b must *translate* the Gctx.t to (2,2) to account for the displacement of w from b's origin

The Border Widget

simpleWidget.ml

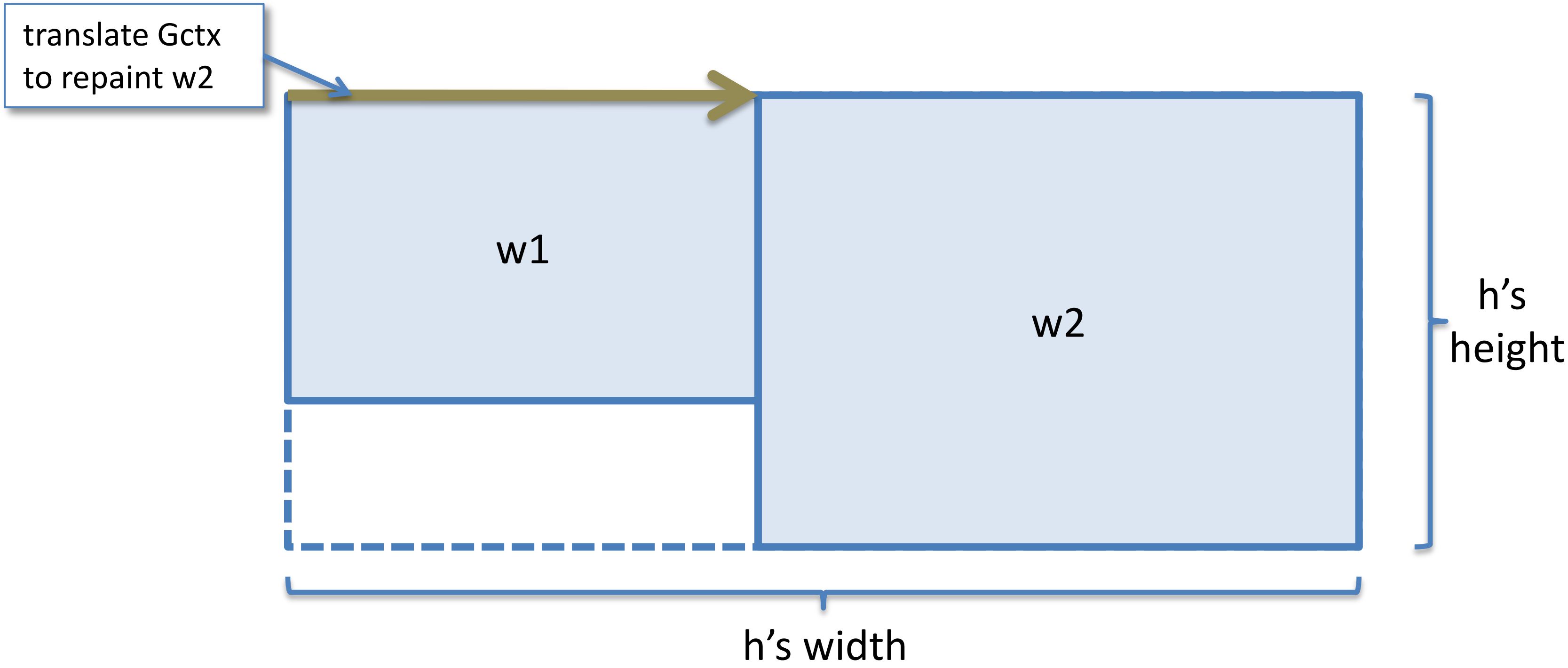
```
let border (w:widget):widget =
{
repaint = (fun (g:Gctx.gctx) ->
  let (width,height) = w.size () in
  let x = width + 3 in
  let y = height + 3 in
  Gctx.draw_line g (0,0) (x,0);
  Gctx.draw_line g (0,0) (0,y);
  Gctx.draw_line g (x,0) (x,y);
  Gctx.draw_line g (0,y) (x,y);
  let gw = Gctx.translate g (2,2) in
  w.repaint gw);

size = (fun () ->
  let (width,height) = w.size () in
  (width+4, height+4))
}
```

Draw the border

Display the interior

The hpair Widget Container



- `let h = hpair w1 w2`
- Creates a horizontally adjacent pair of widgets
- Aligns them by their top edges
  - Must translate the Gctx when repainting w2
- Size is the *sum* of their widths and *max* of their heights

The hpair Widget

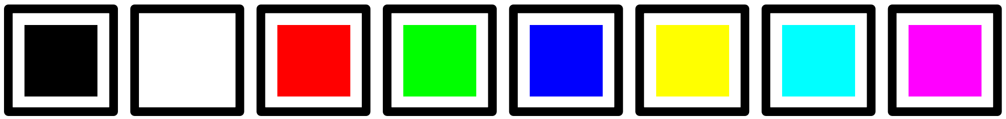
simpleWidget.ml

```
let hpair (w1: widget) (w2: widget) : widget =
  {
    repaint = (fun (g: Gctx.gctx) ->
              let (x1, _) = w1.size () in begin
                w1.repaint g;
                w2.repaint (Gctx.translate g (x1,0))
                (* Note translation of the Gctx *)
              end);

    size = (fun () ->
              let (x1, y1) = w1.size () in
              let (x2, y2) = w2.size () in
              (x1 + x2, max y1 y2))
  }
```

Translate the Gctx to shift w2's position relative to widget-local origin.

Container Widgets for layout



```
let color_toolbar : widget = hlist
   [ color_button black;   spacer;
     color_button white;   spacer;
     color_button red;     spacer;
     color_button green;   spacer;
     color_button blue;    spacer;
     color_button yellow;  spacer;
     color_button cyan;    spacer;
     color_button magenta]
```

paint.ml

hlist is a container widget. It takes a list of widgets and turns them into a single one by laying them out horizontally (using hpair).

What's Next?

- You should be set to work on the first parts of HW05
- Coming up: How do widgets handle events??
- How to we compose widgets into a larger application like the paint program?