

Programming Languages and Techniques (CIS120)

Lecture 17

Feb 28th, 2014

“Objects” and GUI Design

Where we're going...

- HW 6: Build a GUI library and client application *from scratch* in OCaml
 - Available soon
 - Due Friday, March 7th
- Read Ch. 18 in lecture notes
- Goals:
 - Apply everything we've seen so far to do some pretty serious programming
 - Illustrate the *event-driven* programming model
 - Practice with *first-class functions* and *hidden state*
 - Give you a feel for how GUI libraries (like Java's Swing) work
 - Bridge to object-oriented programming
- Then: transition to Java

“Objects” and Hidden State

Objects in Java

```
public class Counter {
```

class name

```
private int count;
```

instance variable

```
public Counter () {  
    count = 0;  
}
```

constructor

```
public int incr () {  
    count = count + 1;  
    return count;  
}
```

methods

```
public int decr () {  
    count = count - 1;  
    return count;  
}
```

class declaration

object creation and use

```
public class Main {
```

```
public static void  
main (String[] args) {
```

constructor invocation

```
Counter c = new Counter();
```

```
System.out.println( c.inc() );
```

```
}
```

method call

What is an Object?

- Object = Instance variables (fields) + Methods
 - Field = Mutable record
 - Methods = (Immutable) record of first-class functions that update the fields
- Objects *encapsulate* state when the methods are the **only** way to mutate the fields.
- Objects are first-class.
- Can we get similar behavior in OCaml?

An “incr” function

- This function increments a counter and return its new value each time it is called:

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

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

What is the result of this computation?

```
let mk_incr () : unit -> int =  
  let ctr = {count = 0} in  
  fun () ->  
    ctr.count <- ctr.count + 1;  
    ctr.count  
let incr1 : unit -> int = mk_incr ()  
let incr2 : unit -> int = mk_incr ()  
let _ = incr1 ()  
let _ = incr1 ()  
incr1 ()
```

1. 1
2. 2
3. 3
4. runtime error

What is the result of this computation?

```
let mk_incr () : unit -> int =  
  let ctr = {count = 0} in  
  fun () ->  
    ctr.count <- ctr.count + 1;  
    ctr.count  
let incr1 : unit -> int = mk_incr ()  
let incr2 : unit -> int = mk_incr ()  
let _ = incr1 ()  
let _ = incr2 ()  
incr1 ()
```

1. 1
2. 2
3. 3
4. runtime error

Running mk_incr

Workspace

Stack

Heap

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

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

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

Stack

Heap

```
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 ()
```

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

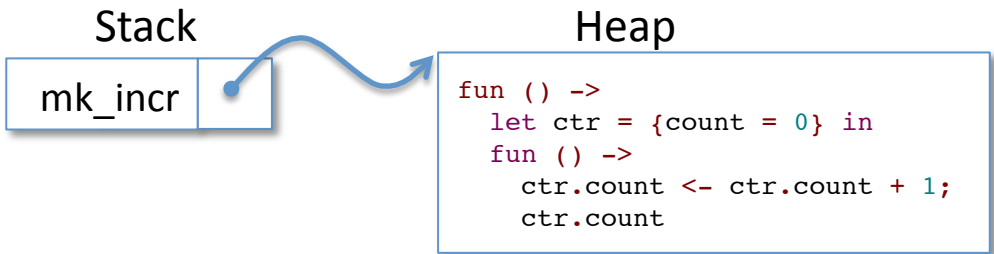
```
let incr1 : unit -> int =  
mk_incr ()
```

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

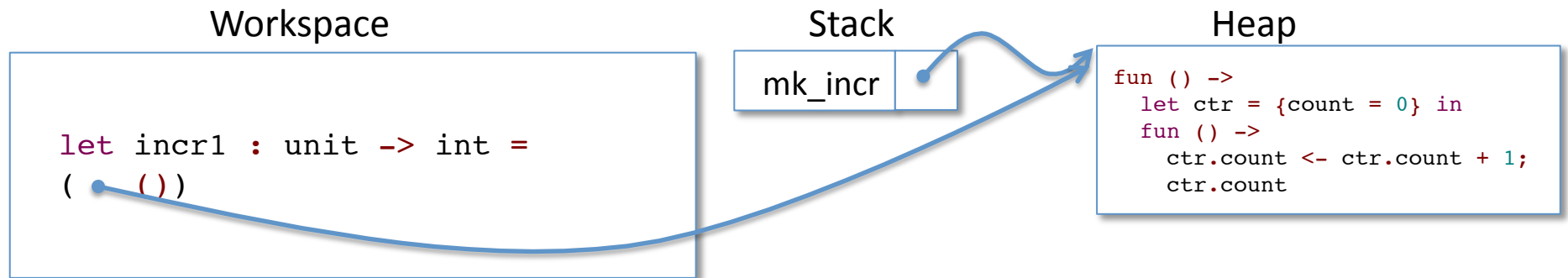
mk_incr



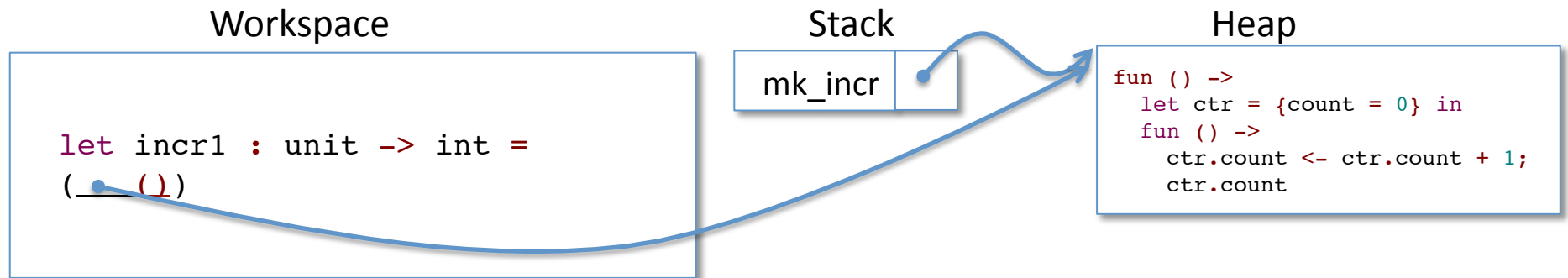
Heap

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


Running mk_incr



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

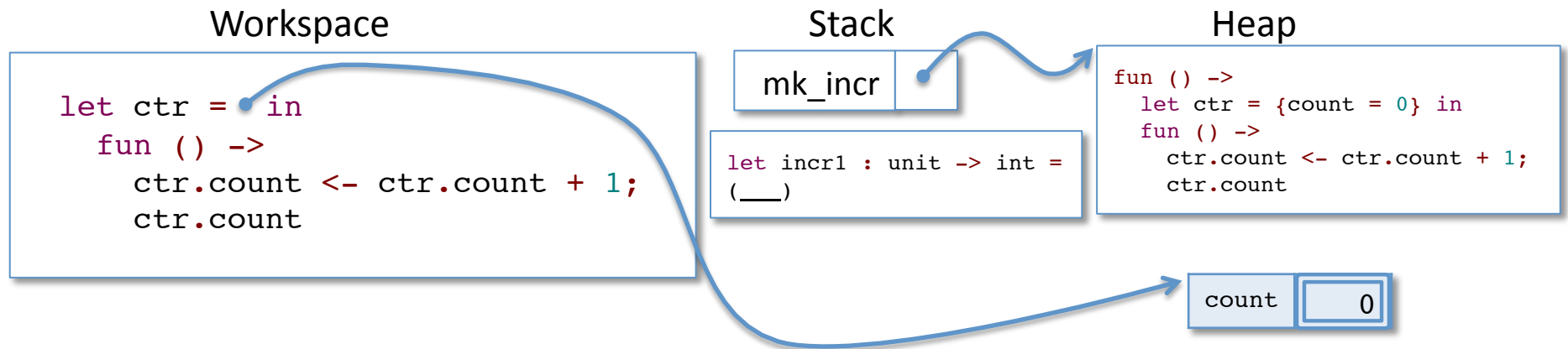
```
let incr1 : unit -> int =
  (___)
```

Heap

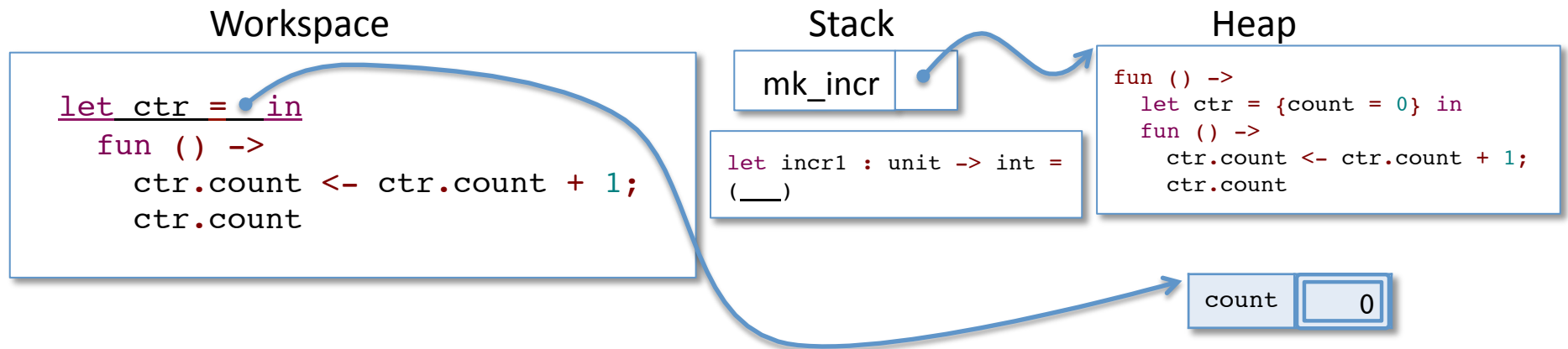
```
fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```



Running mk_incr



Running mk_incr



Running mk_incr

Workspace

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

Stack

mk_incr

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

ctr

Heap

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

count

0

Running mk_incr

Workspace

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

Stack

mk_incr

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

ctr

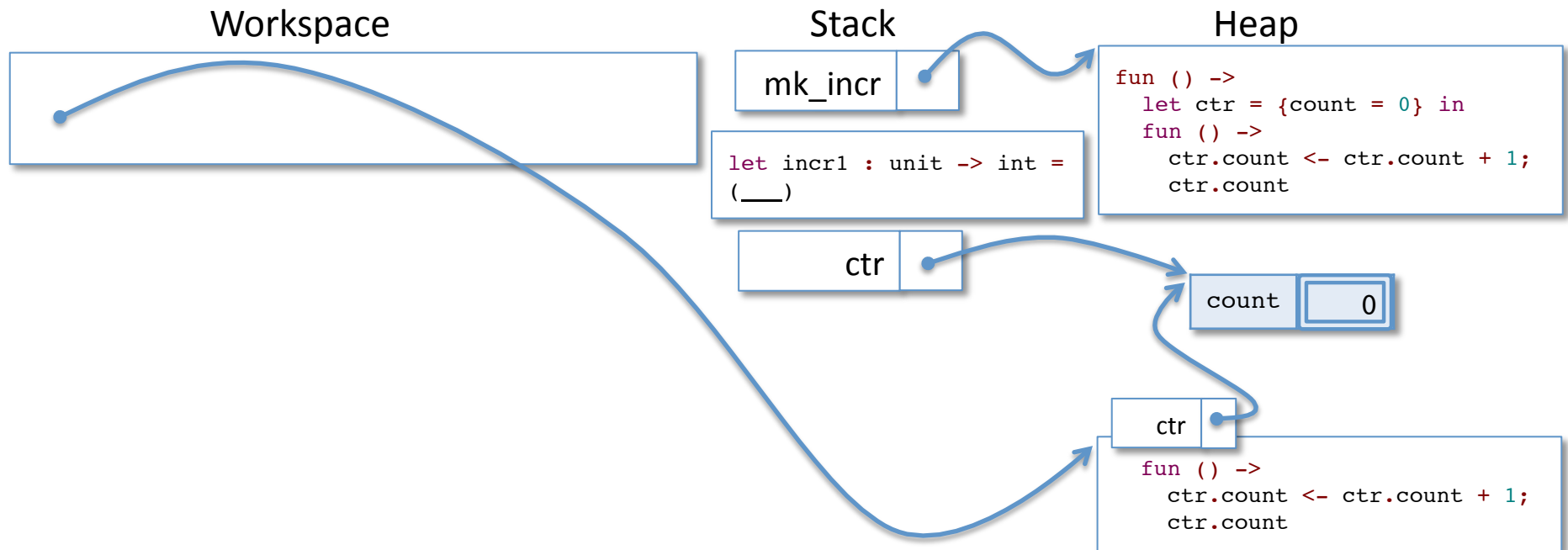
Heap

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

count

0

Local Functions

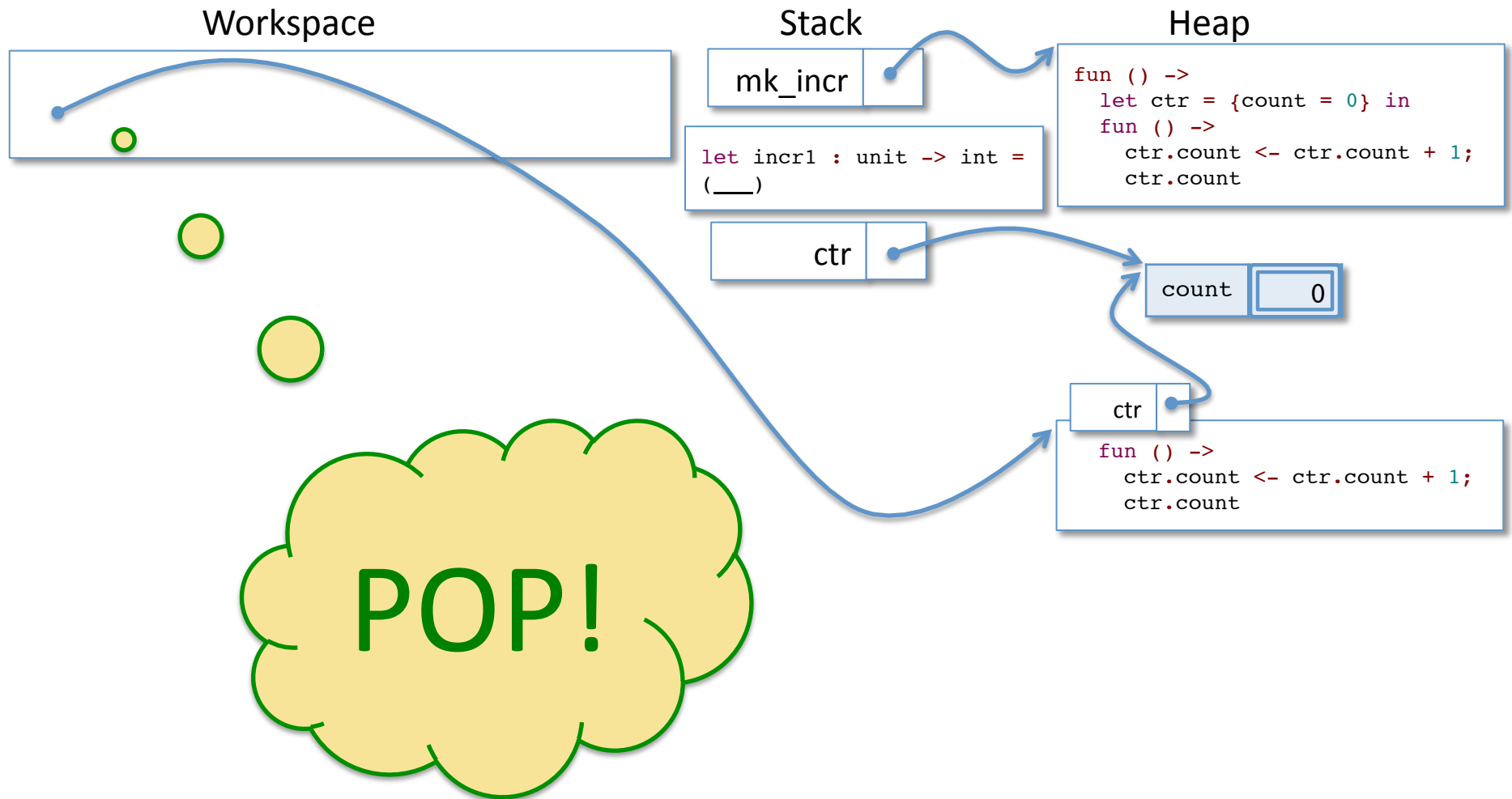


NOTE: We need one refinement of the ASM model to handle local functions. Why?

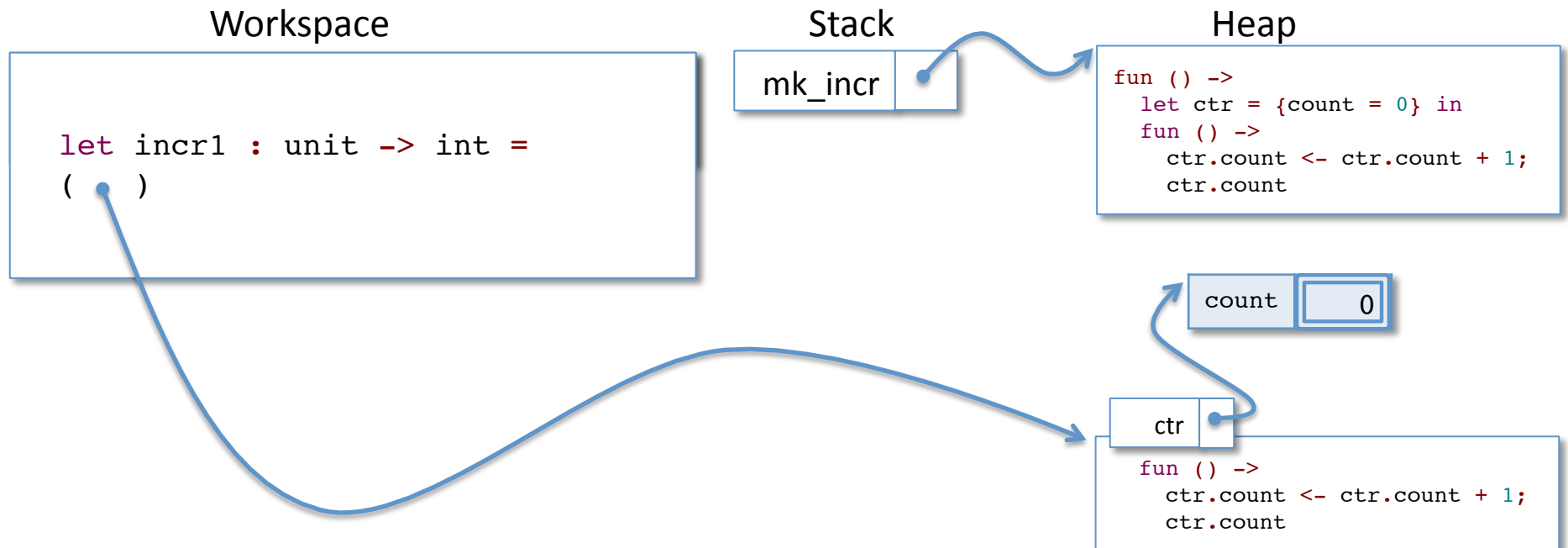
The function mentions “ctr”, which is on the stack (but about to be popped off)...

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

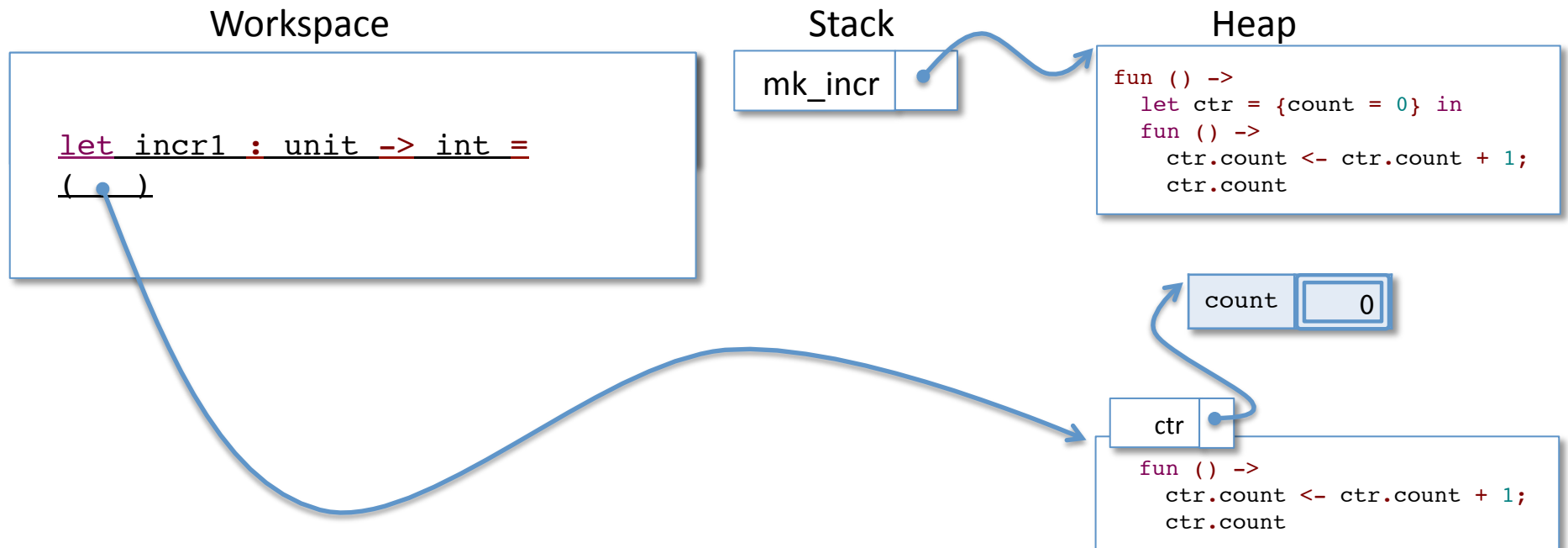
Local Functions



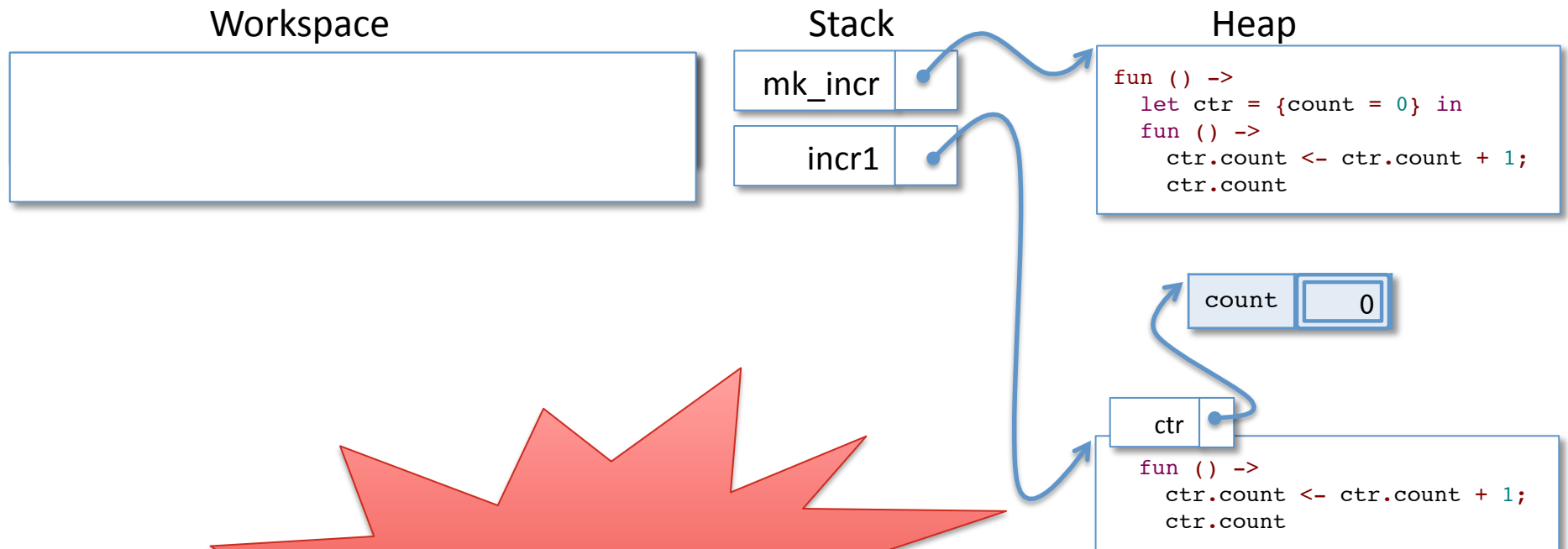
Local Functions



Local Functions

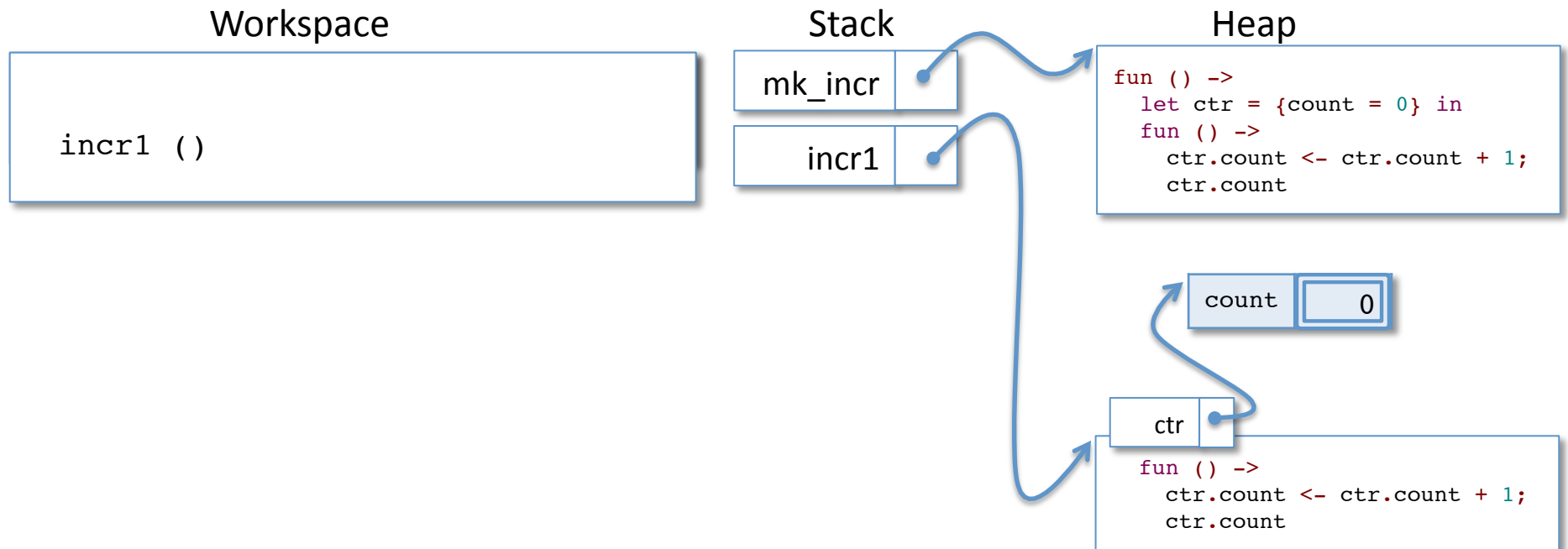


Local Functions

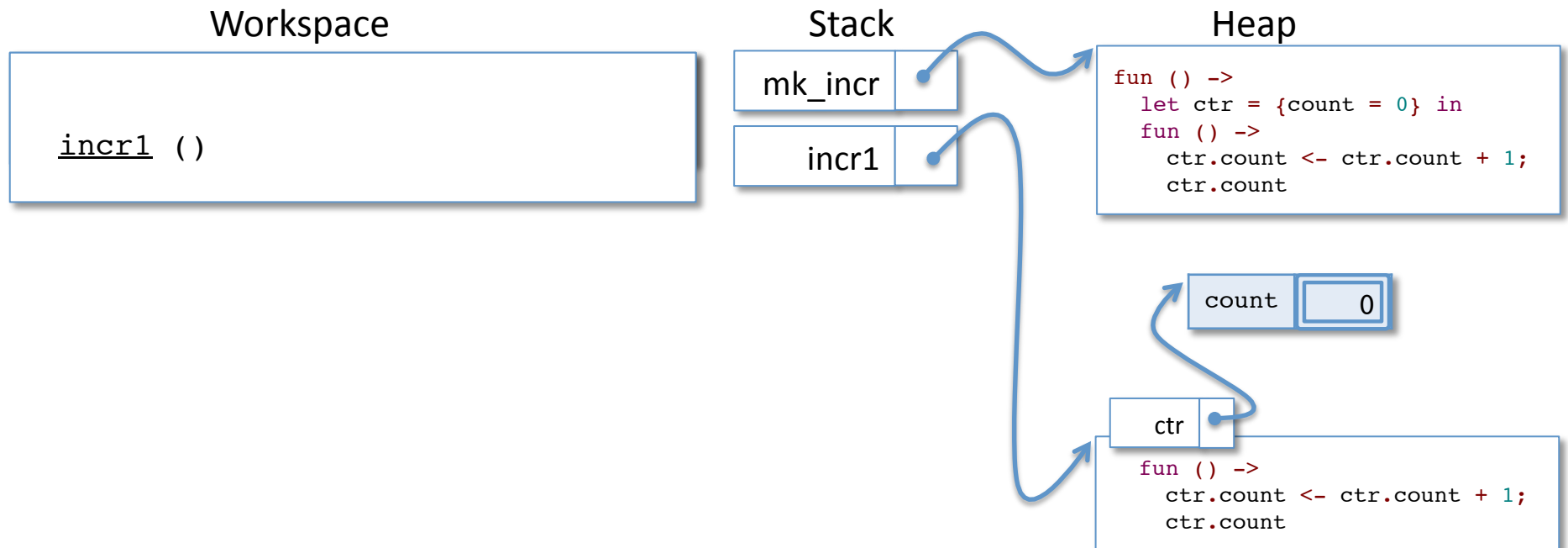


Note how the count record is accessible only via the `incr1` function. This is the sense in which the state is “local” to `incr1`.

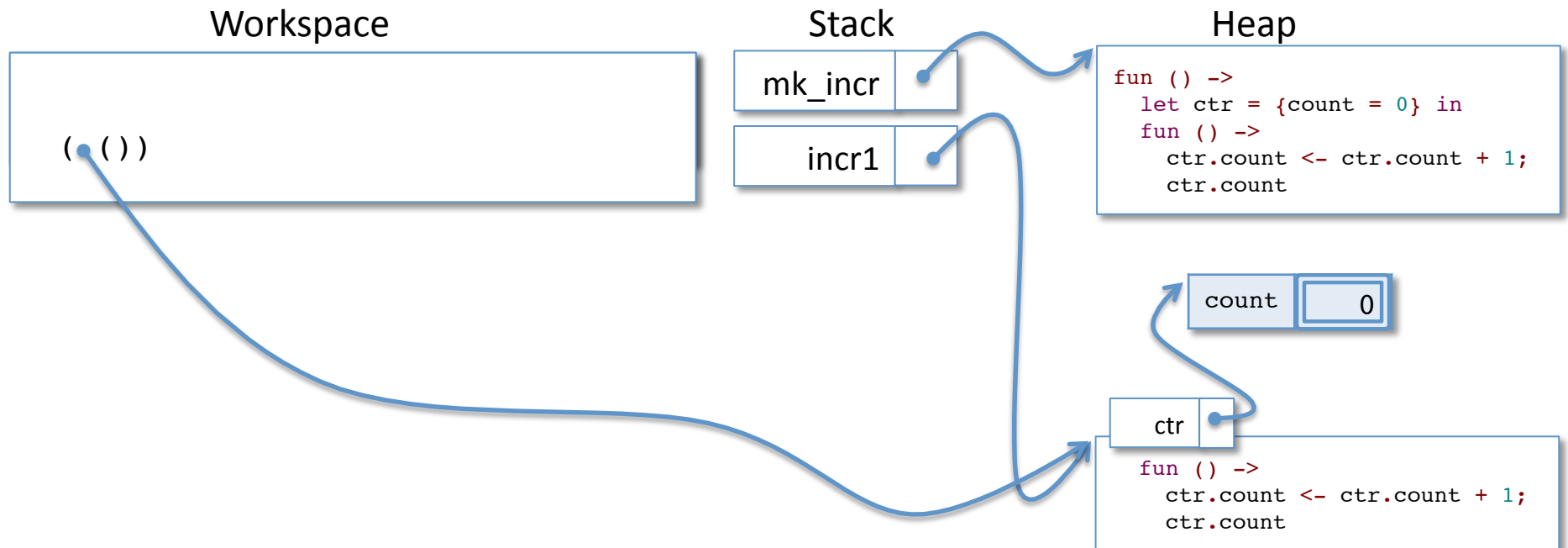
Now let's run "incr1 ()"



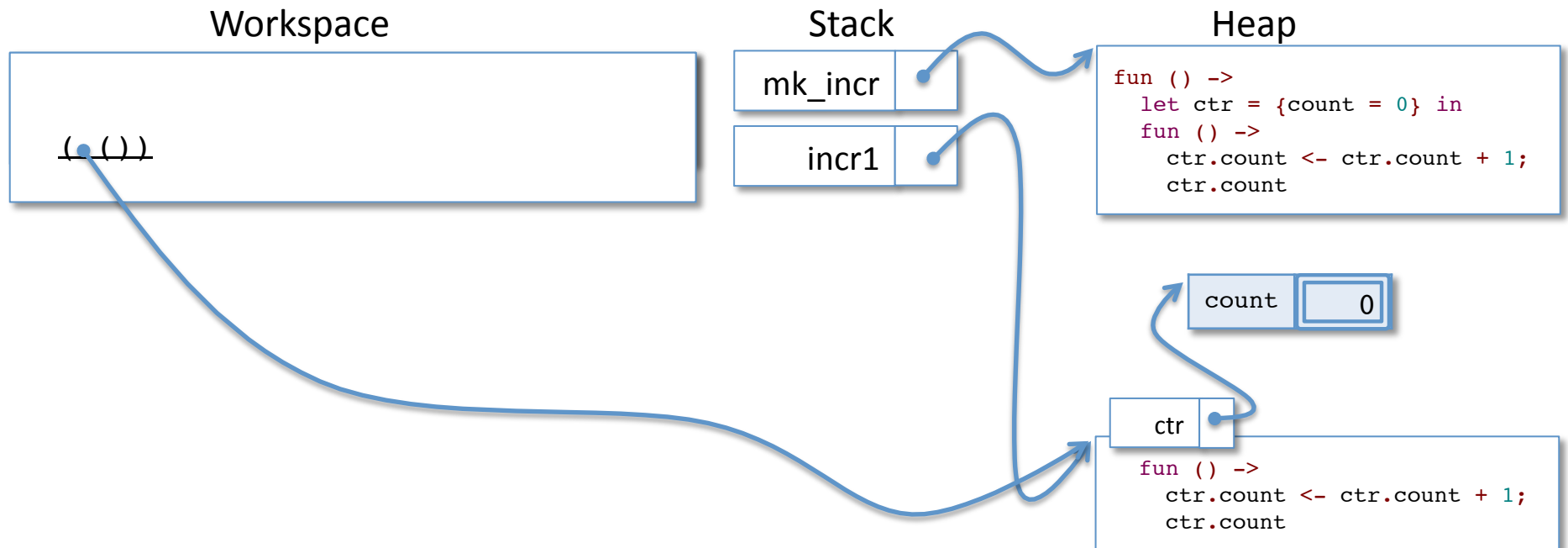
Now let's run "incr1 ()"



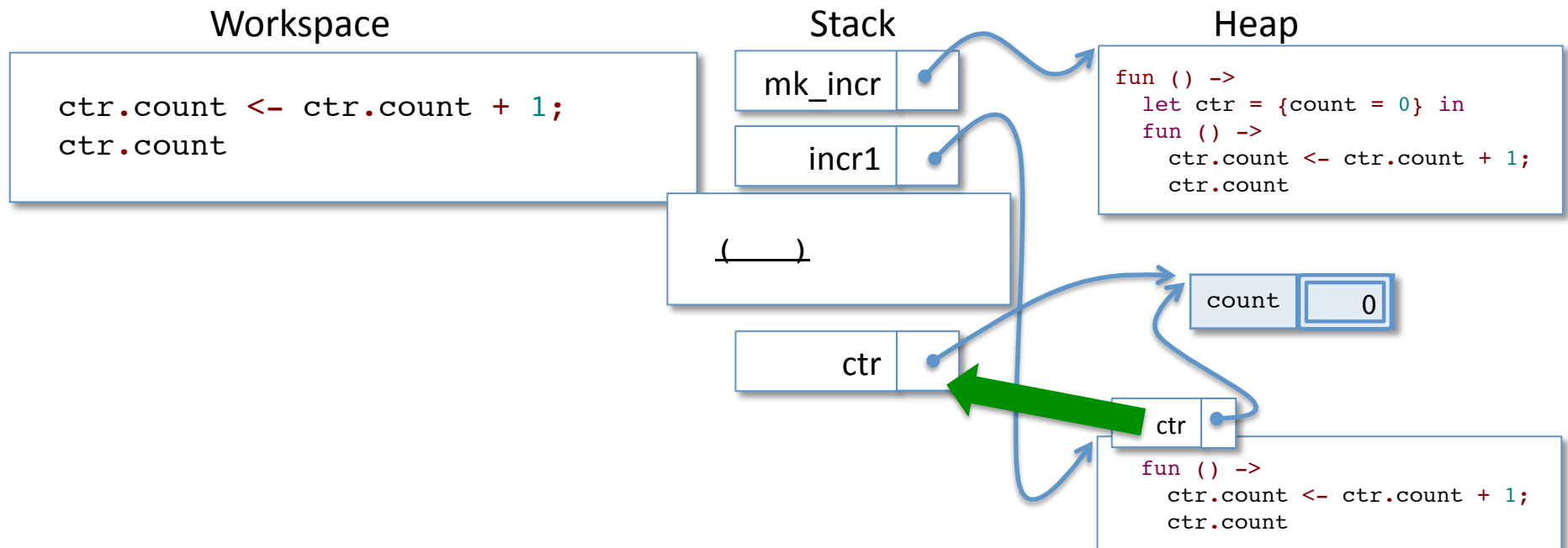
Now let's run "incr1 ()"



Now let's run "incr1 ()"

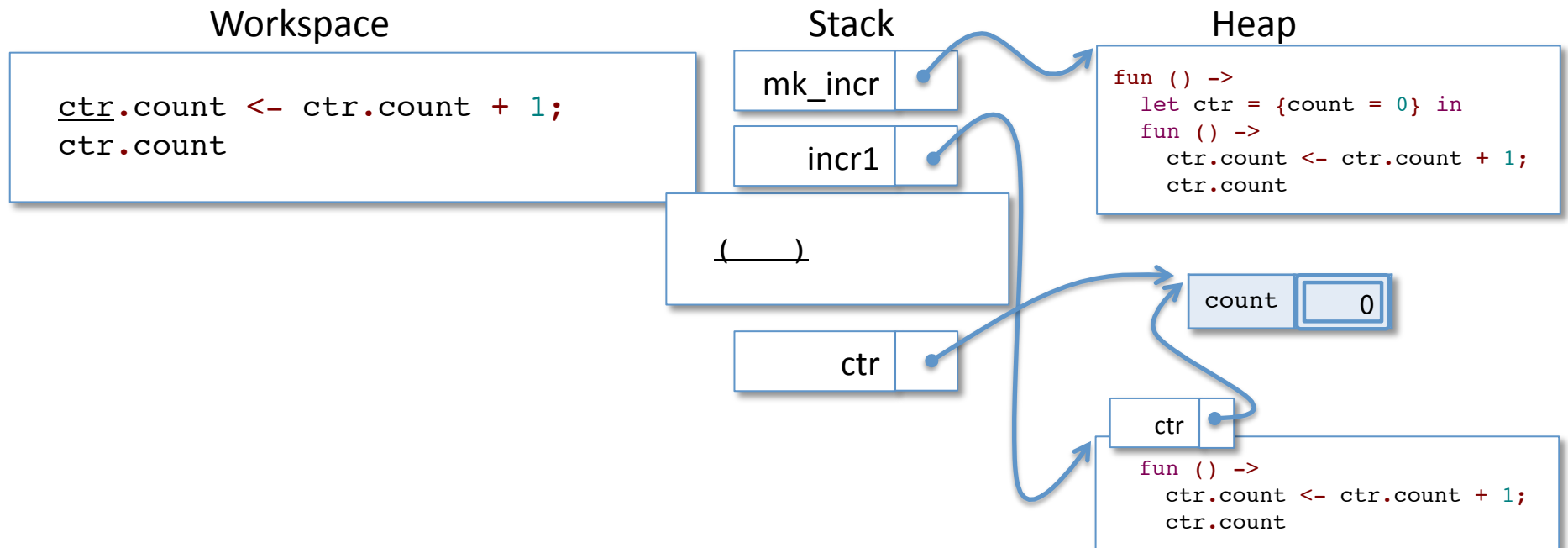


Now let's run "incr1 ()"

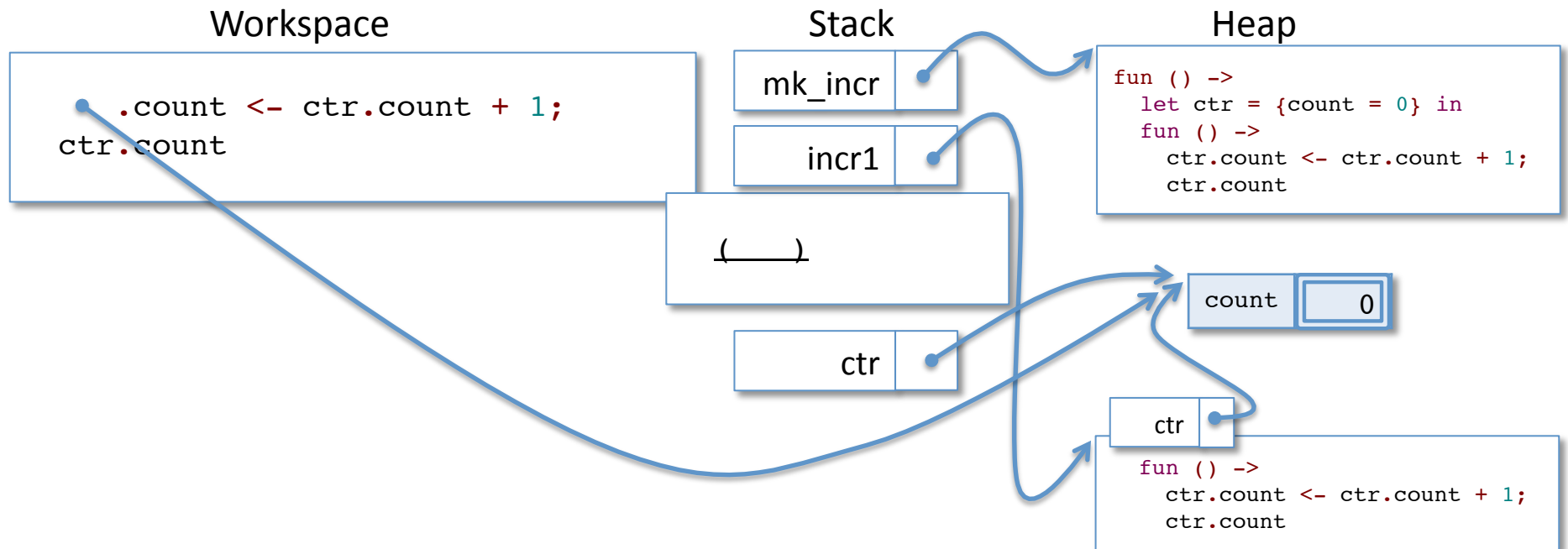


NOTE: Since the function had some saved stack bindings, we add them to the stack at the same time that we put the code in the workspace.

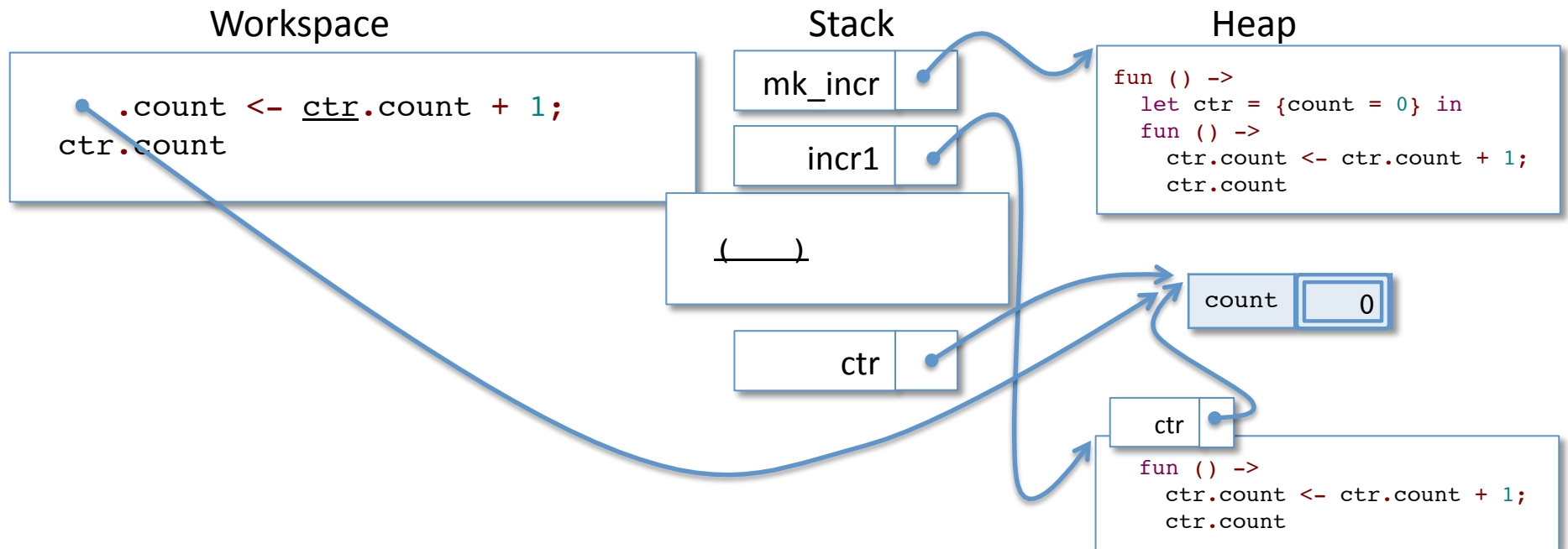
Now let's run "incr1 ()"



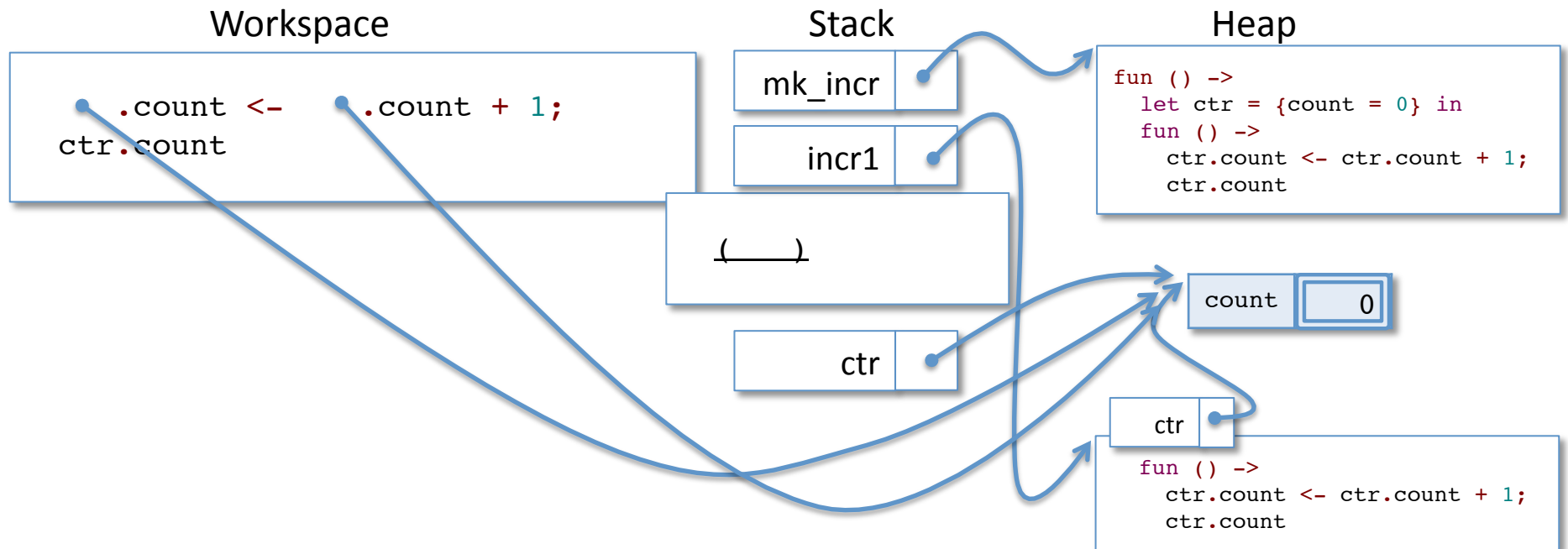
Now let's run "incr1 ()"



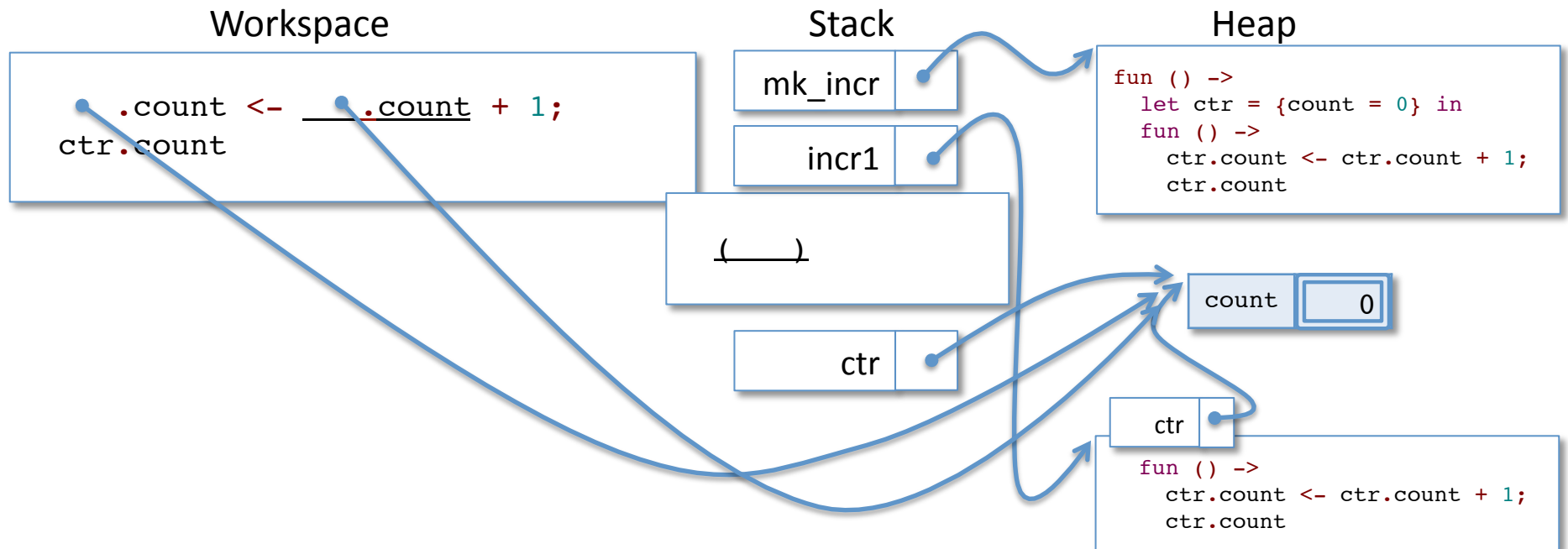
Now let's run "incr1 ()"



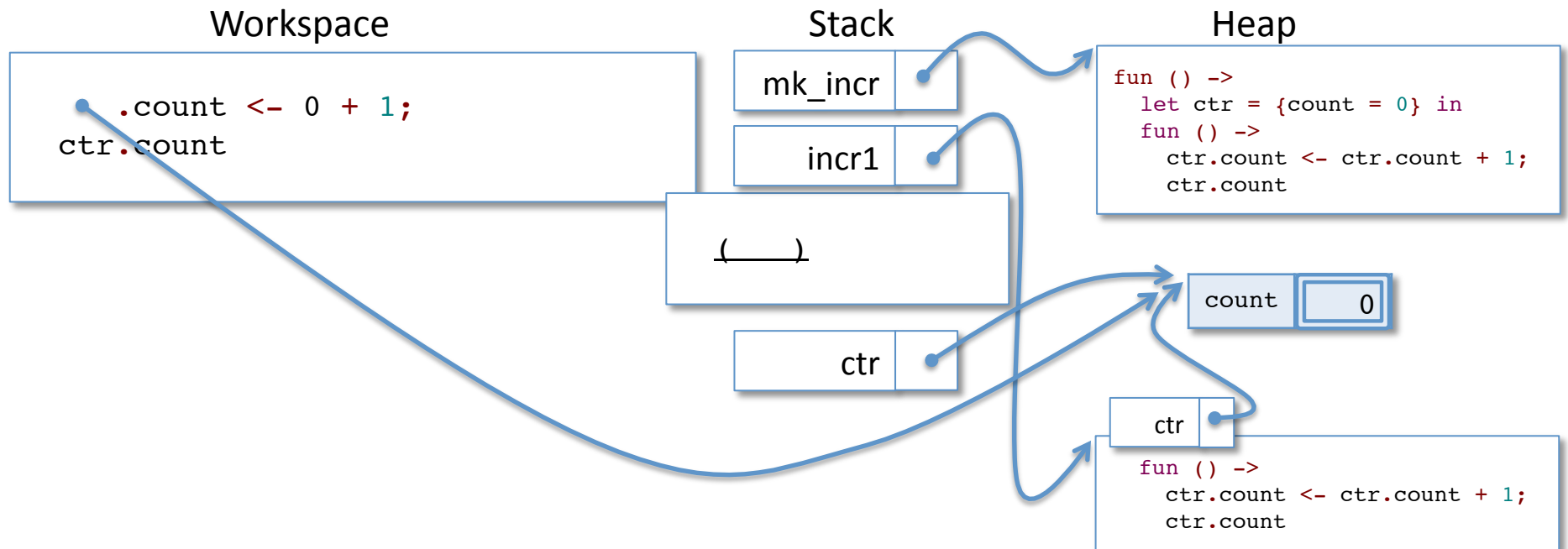
Now let's run "incr1 ()"



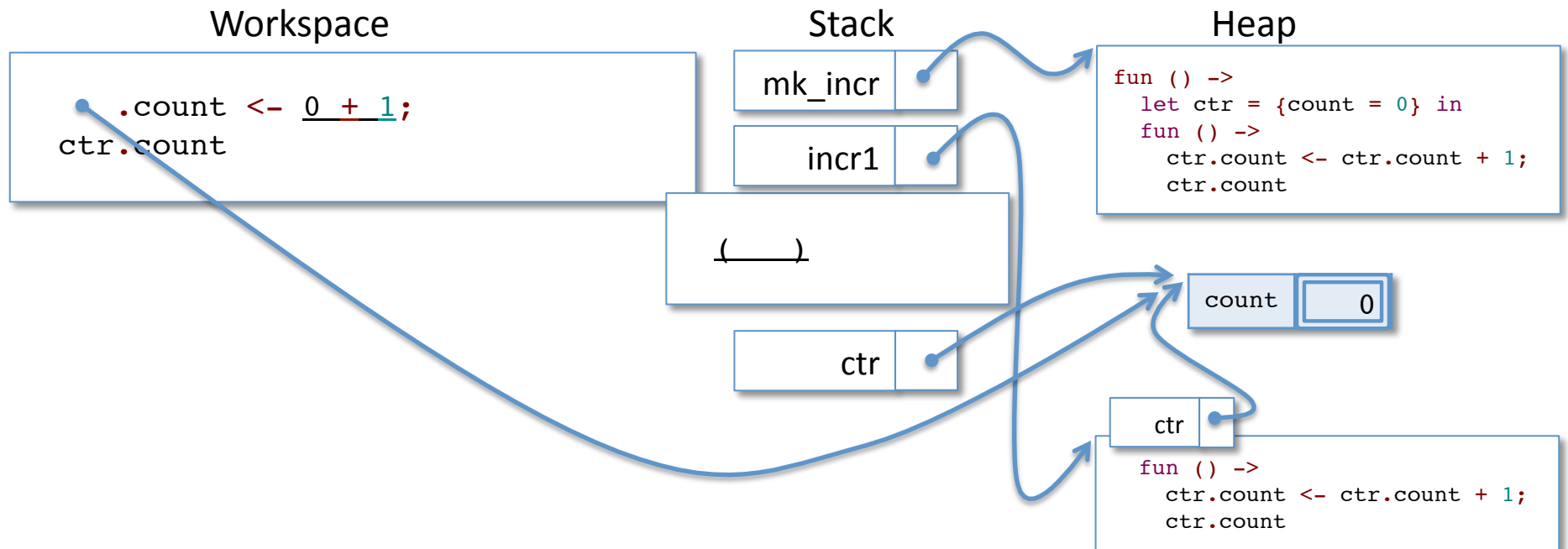
Now let's run "incr1 ()"



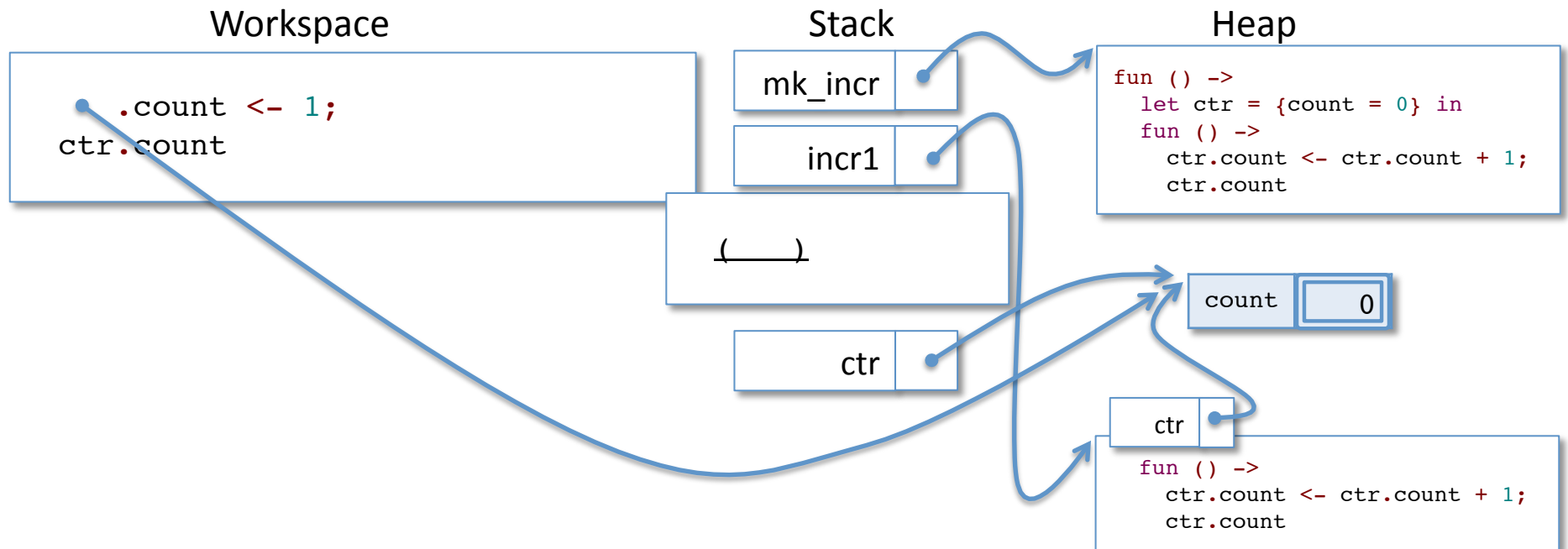
Now let's run "incr1 ()"



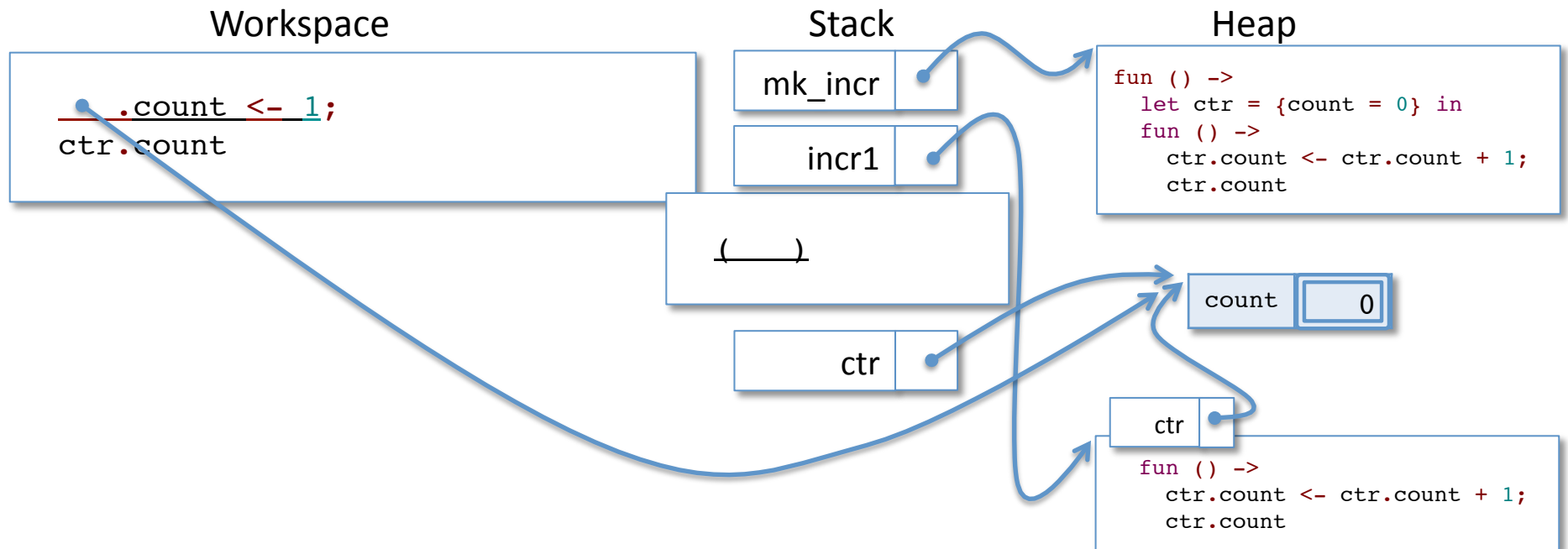
Now let's run "incr1 ()"



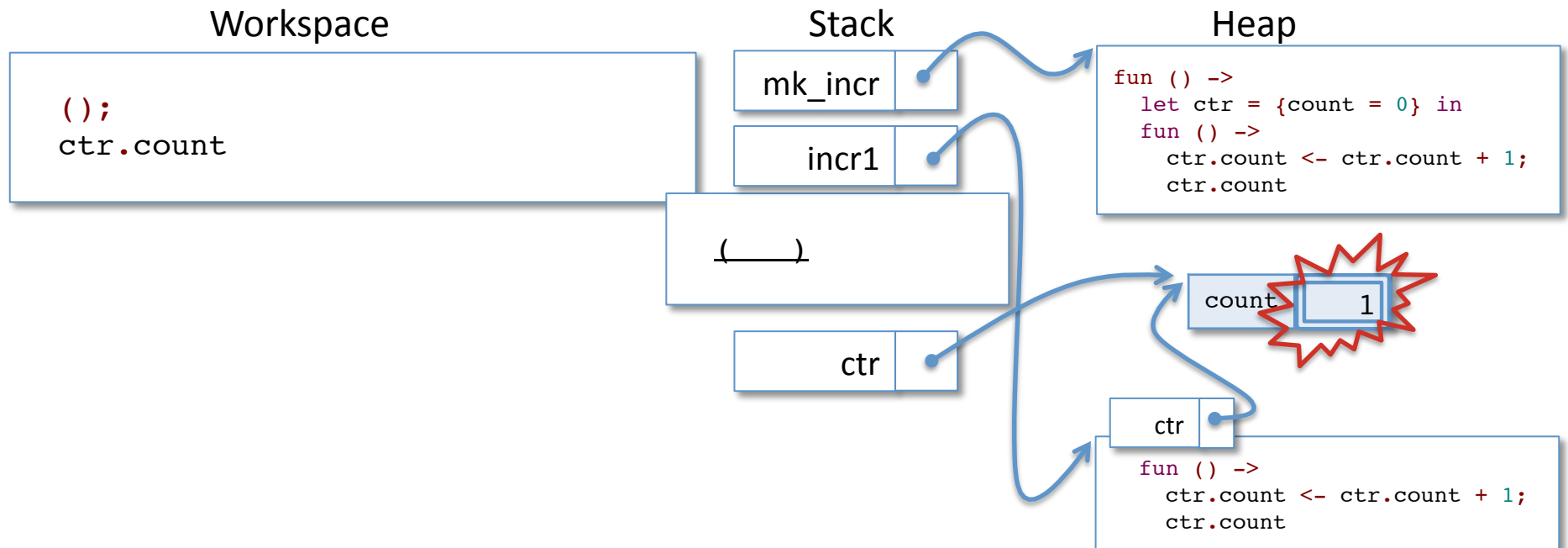
Now let's run "incr1 ()"



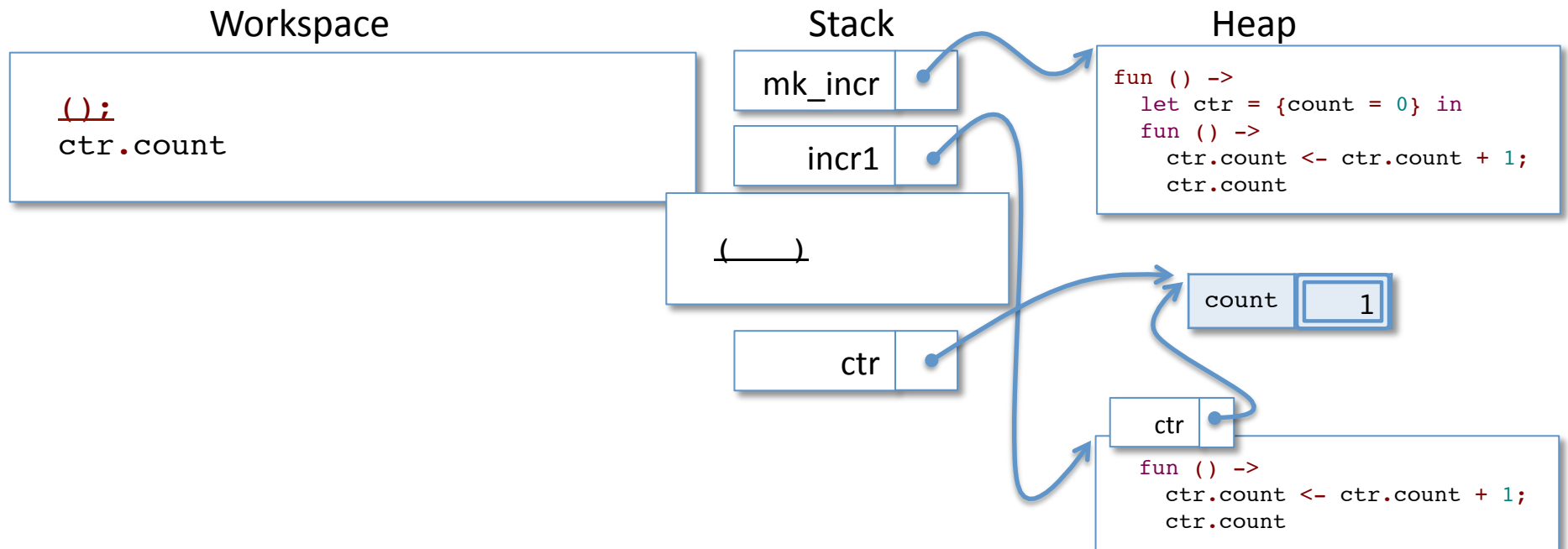
Now let's run "incr1 ()"



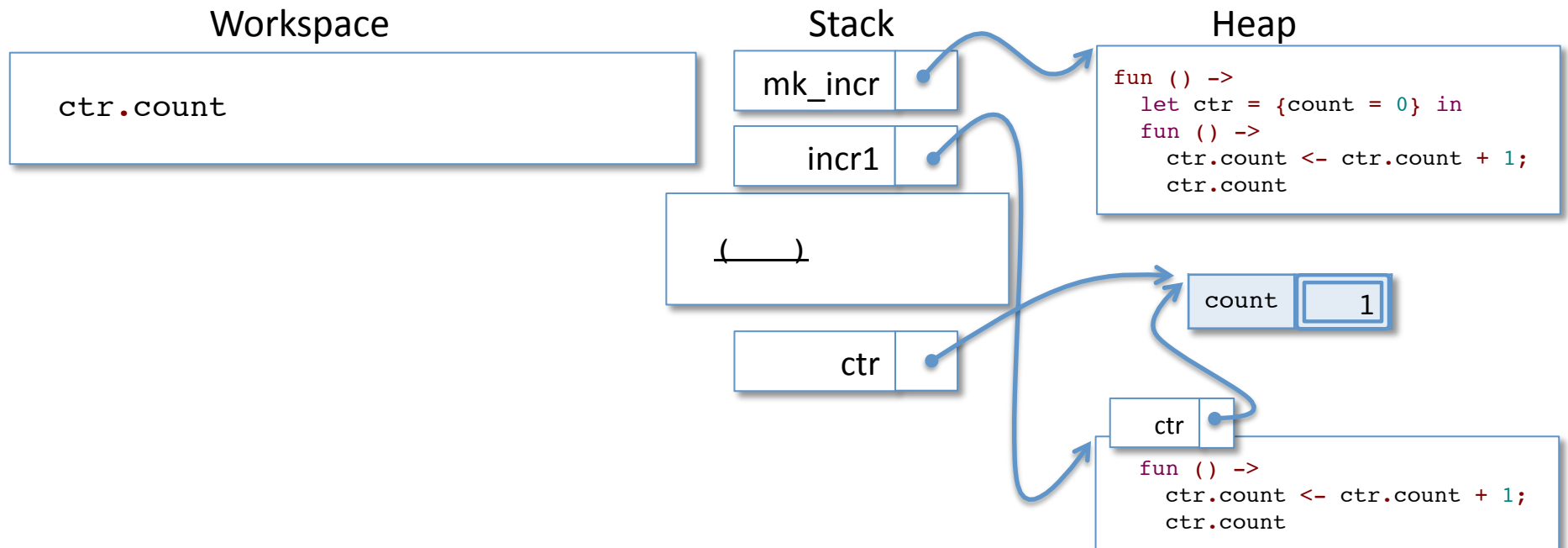
Now let's run "incr1 ()"



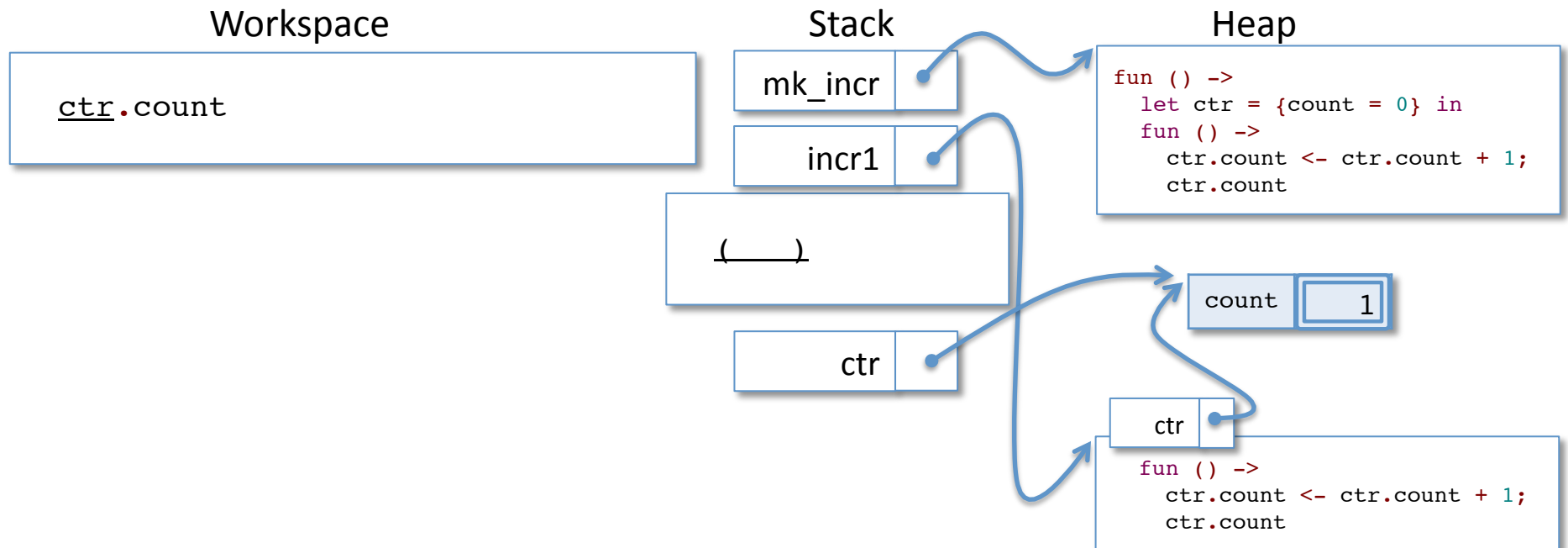
Now let's run "incr1 ()"



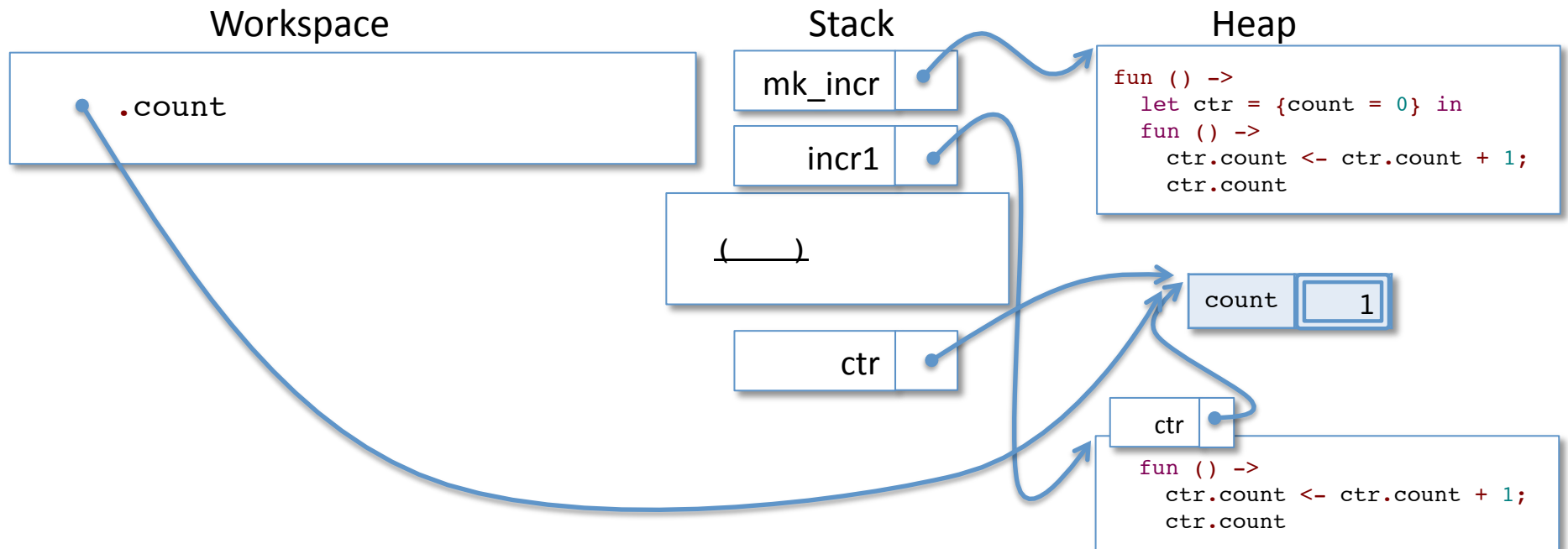
Now let's run "incr1 ()"



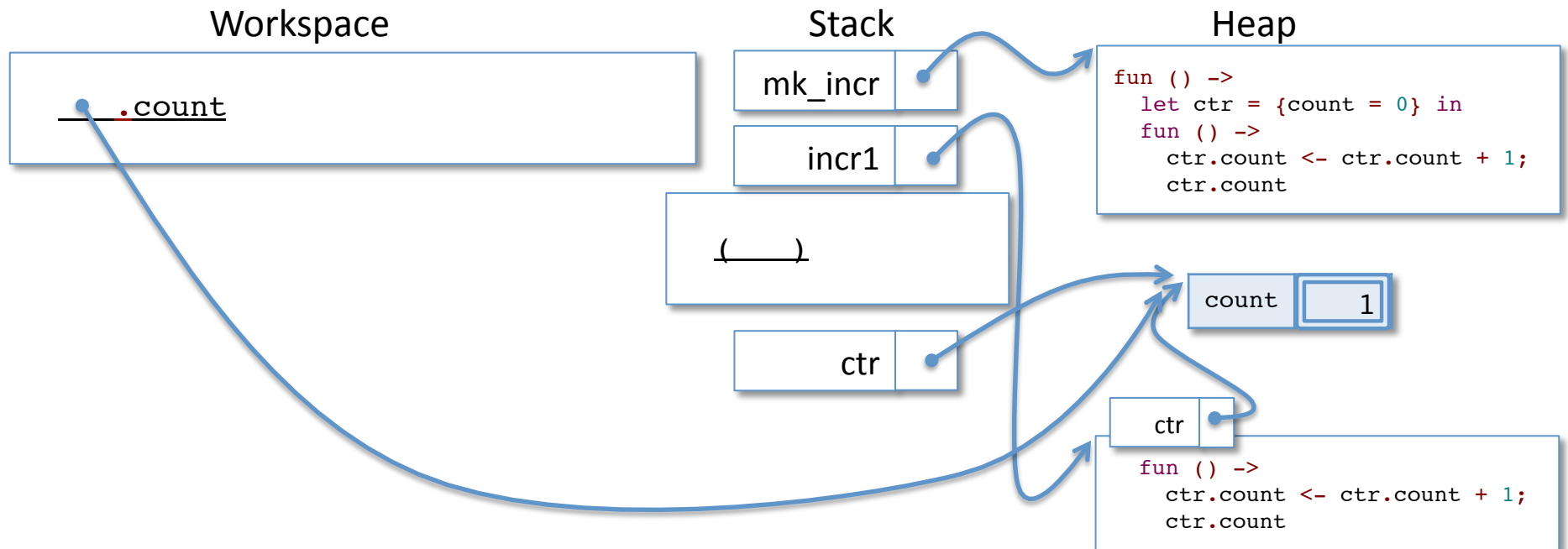
Now let's run "incr1 ()"



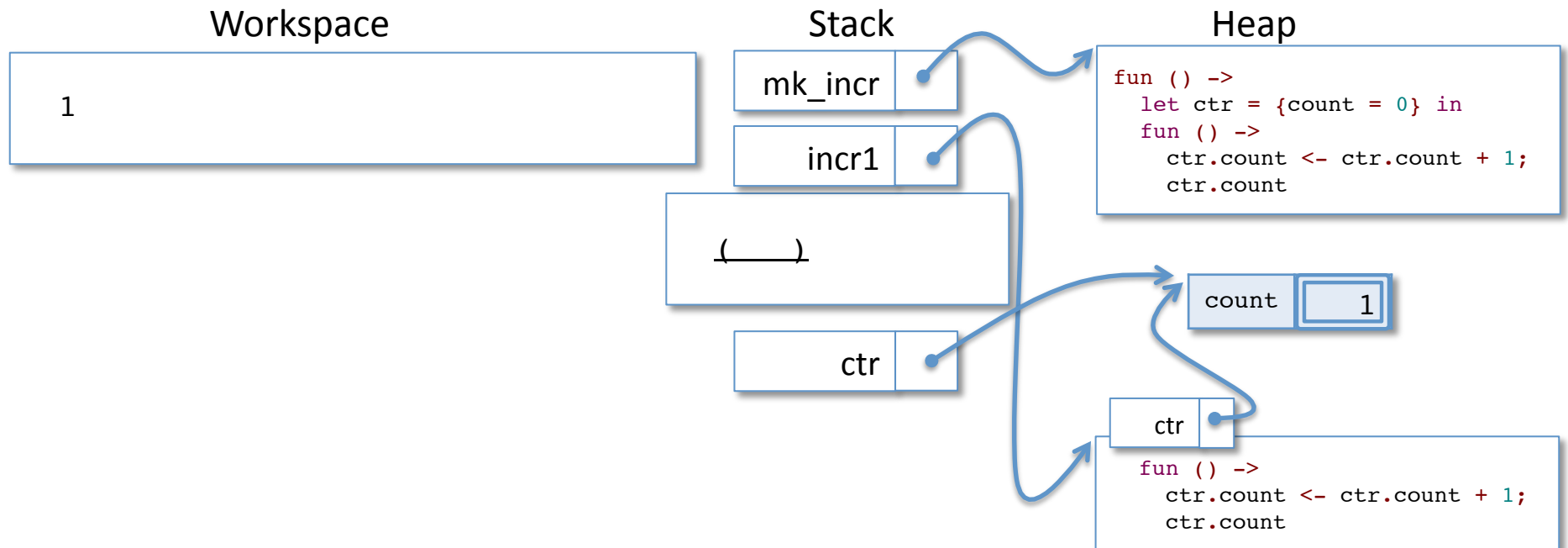
Now let's run "incr1 ()"



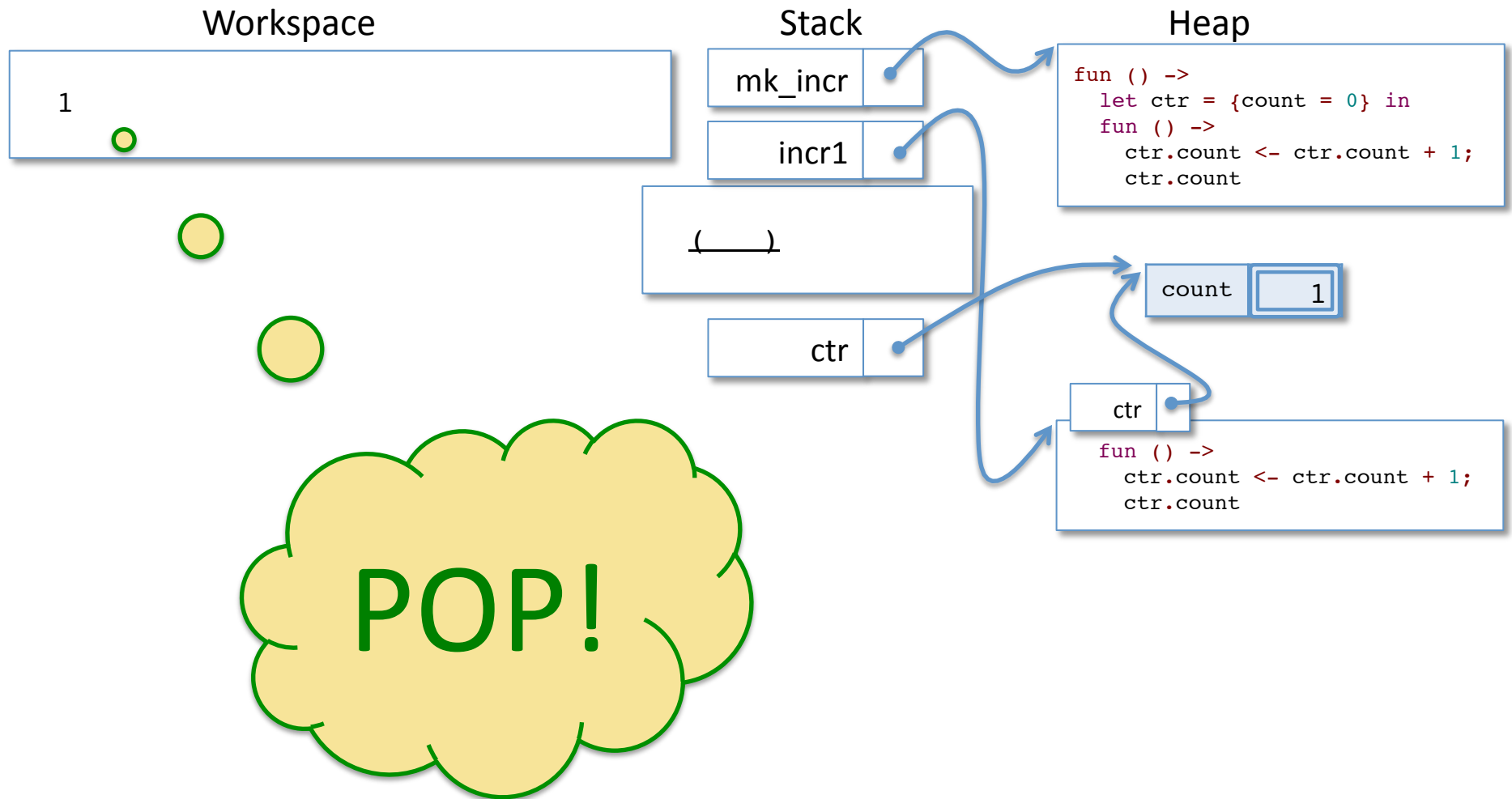
Now let's run "incr1 ()"



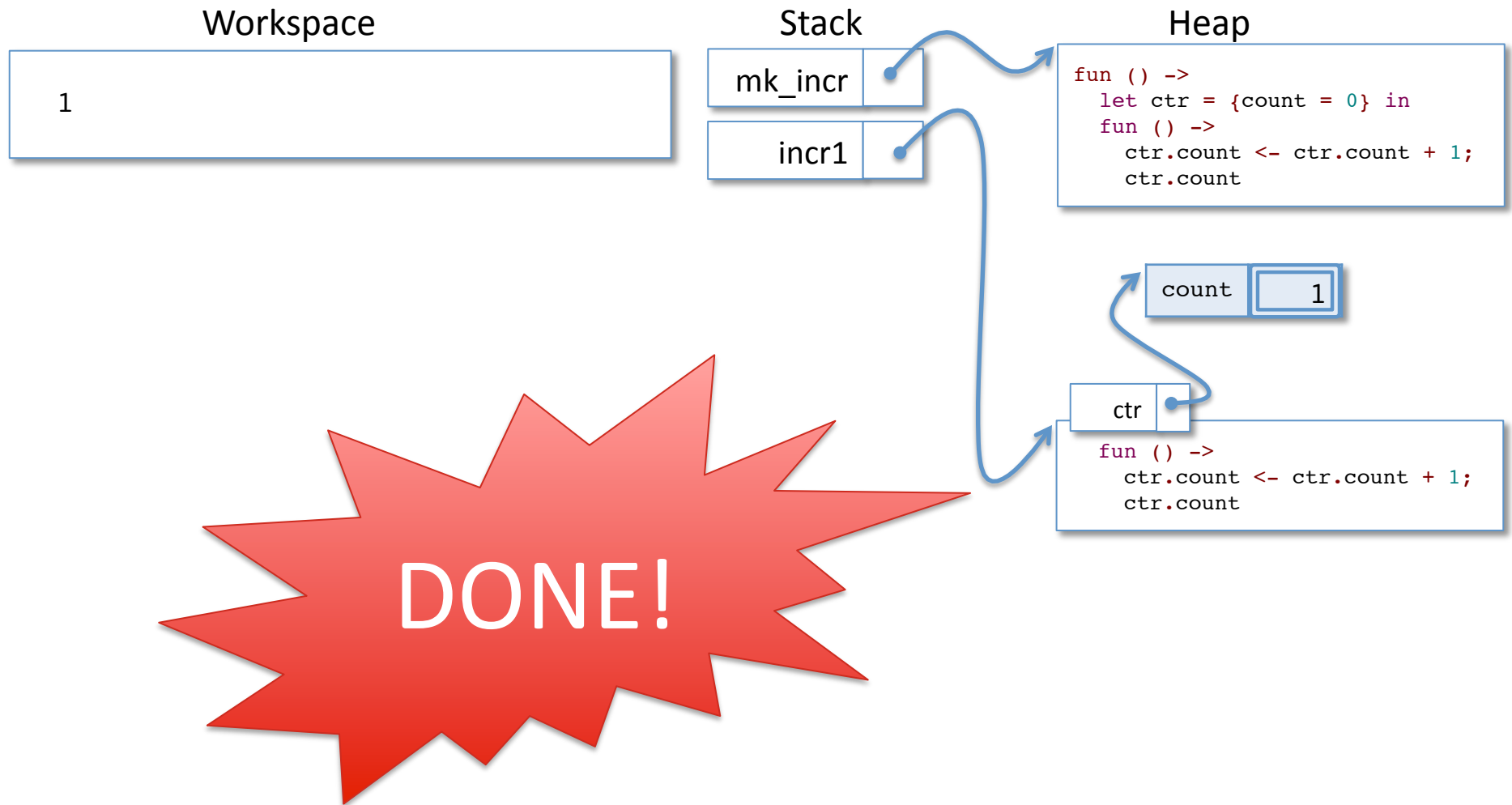
Now let's run "incr1 ()"



Now let's run "incr1 ()"



Now let's run "incr1 ()"



Now Let's run `mk_incr` again

Workspace

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

Stack

mk_incr

incr1

Heap

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

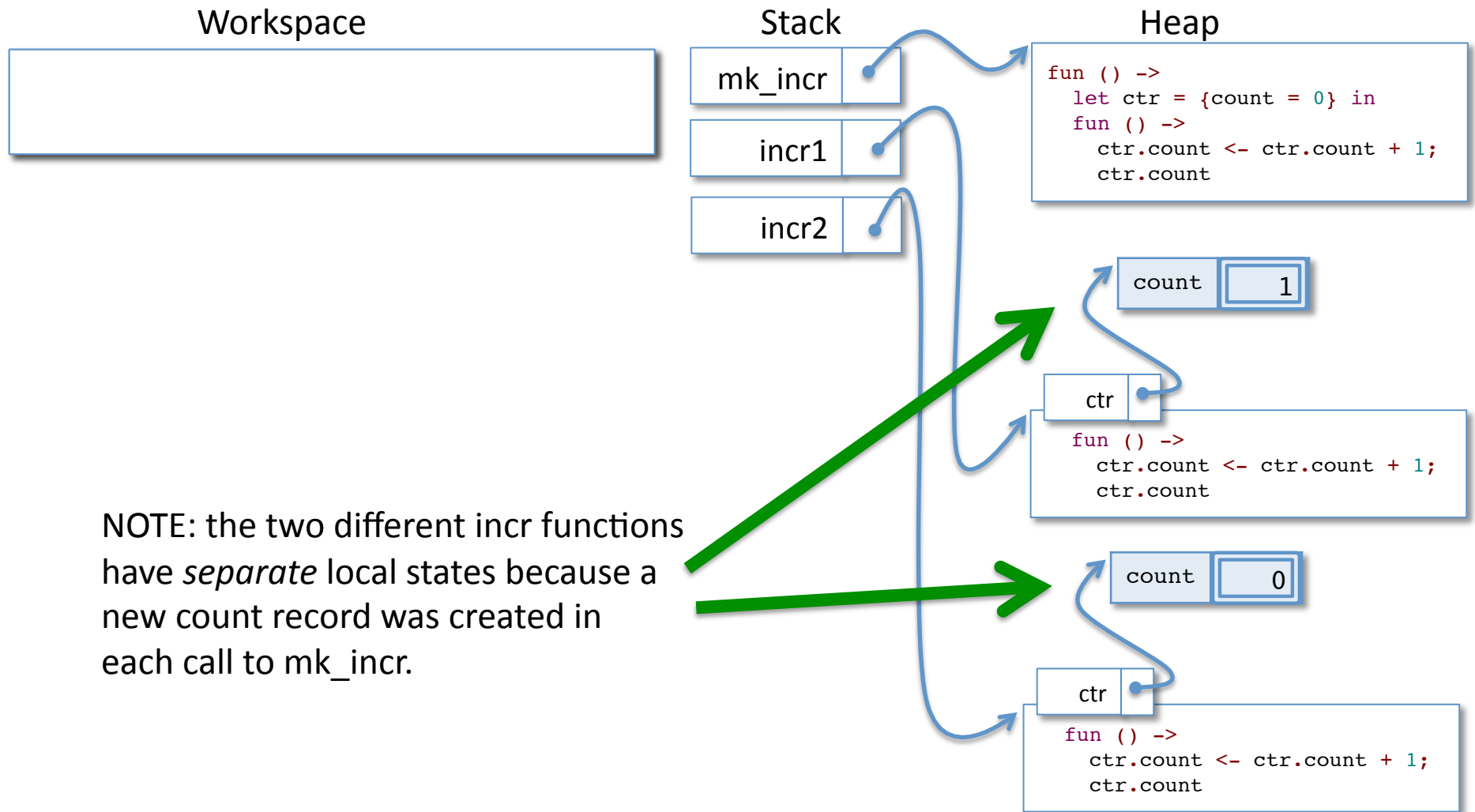
count 1

ctr

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

...time passes...

After creating incr2



One step further

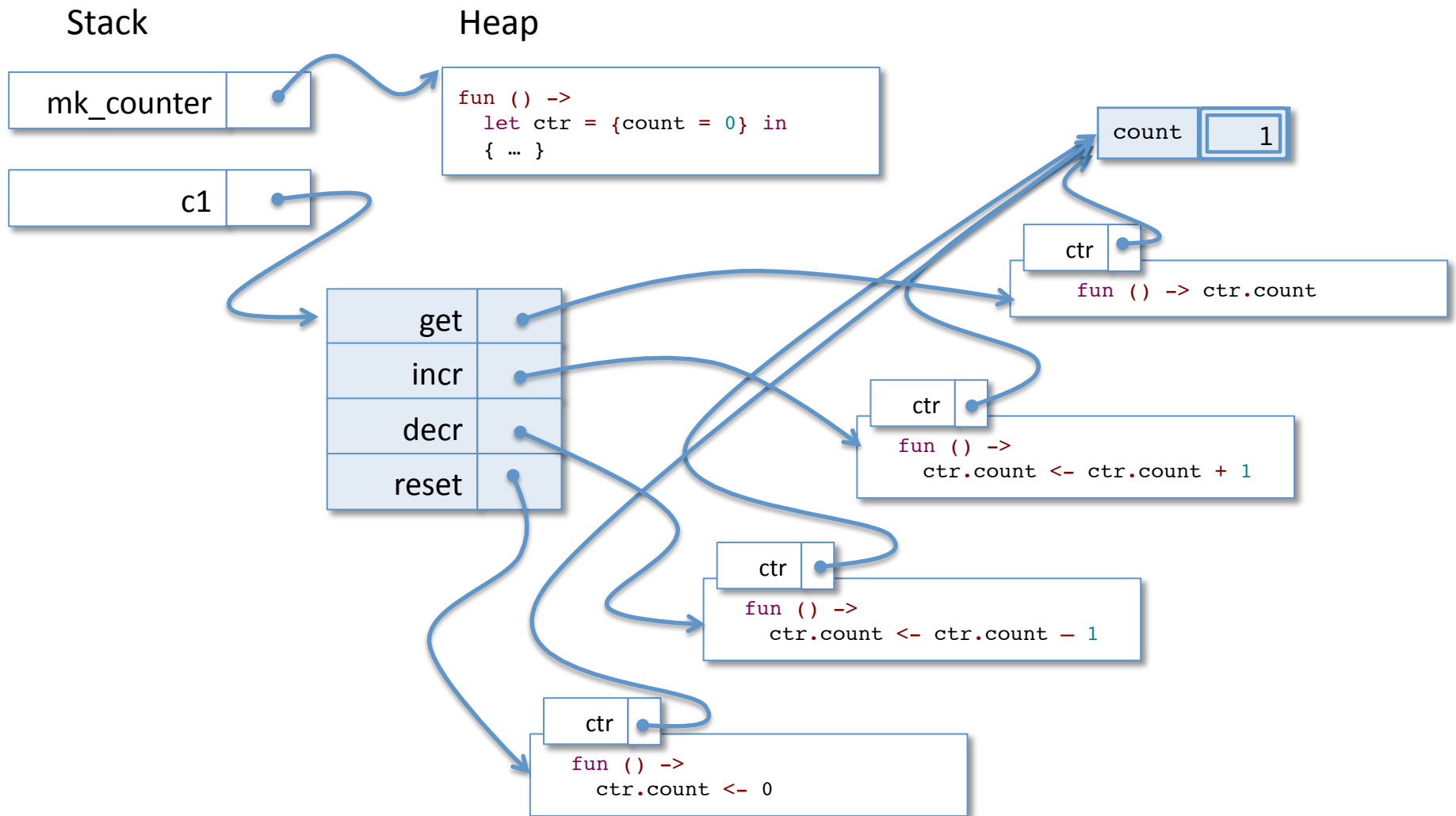
- `mk_incr` shows us how to create different instance of local state so that we can have several different counters.
- What if we want to bundle together *several* operations that share the same local state?
 - e.g. `incr` and `decr` operations that work on the same counter

A Counter *Object*

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

(* Create a counter object with hidden state: *)
let mk_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) ;}
```

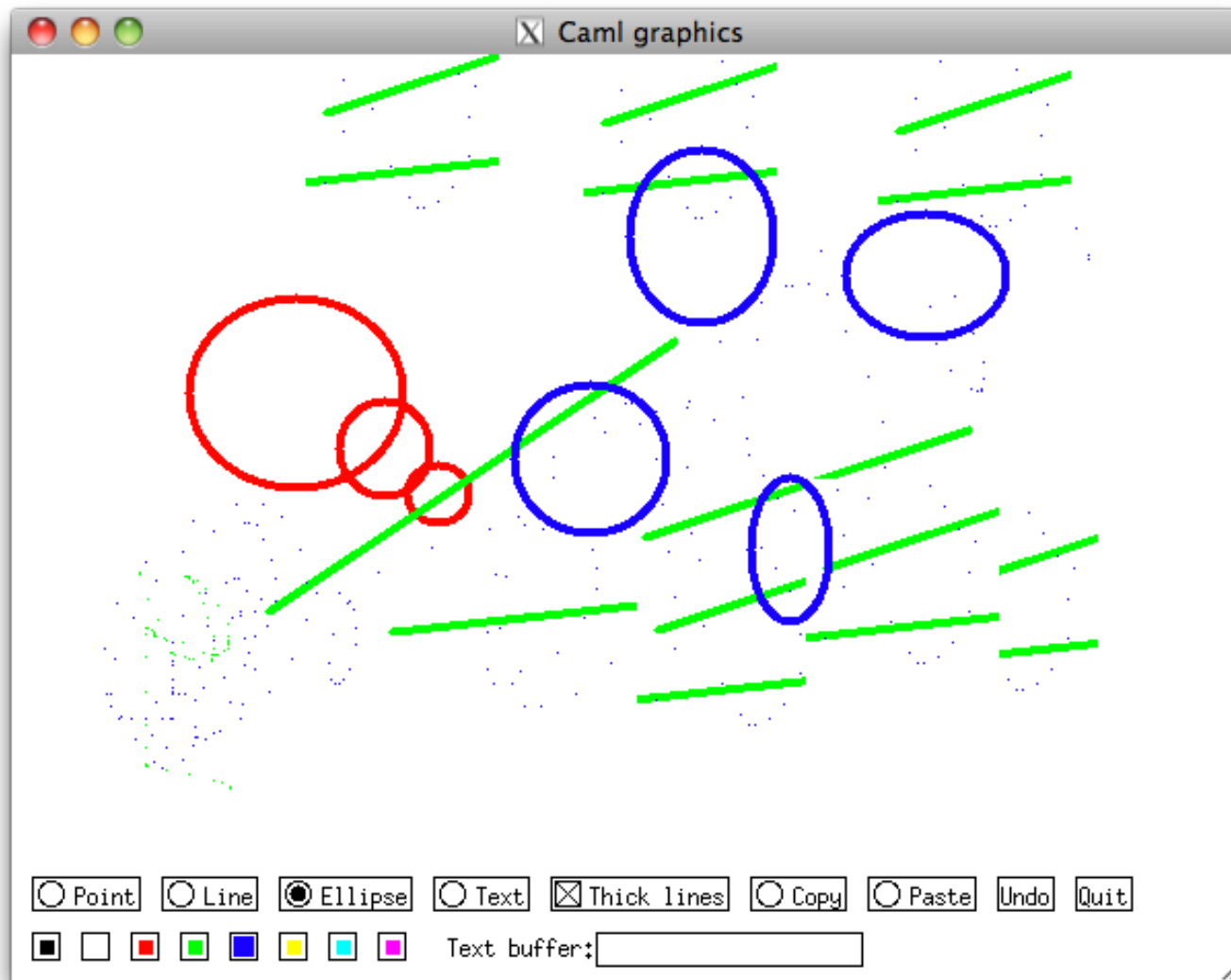
```
let c1 = mk_counter ()
```



GUI Design

putting objects to work

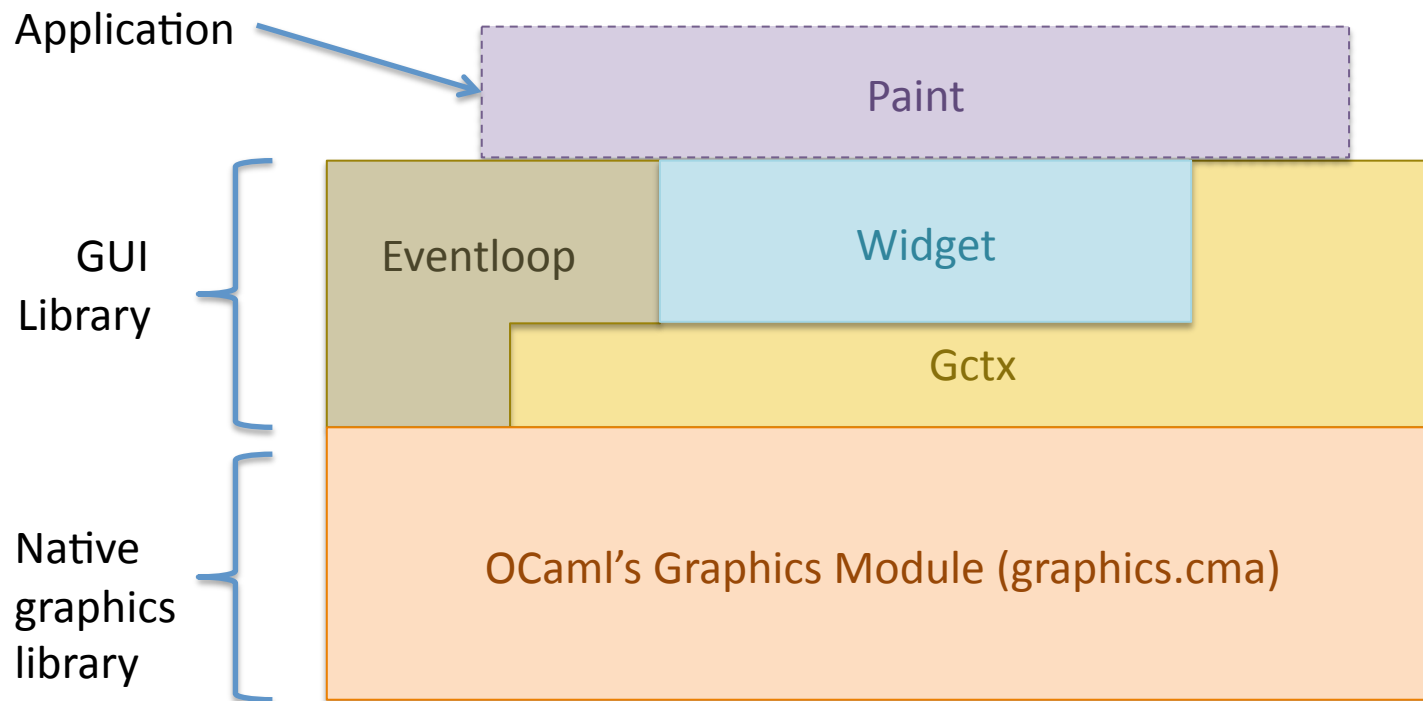
Building a GUI library



Step #1: Understand the Problem

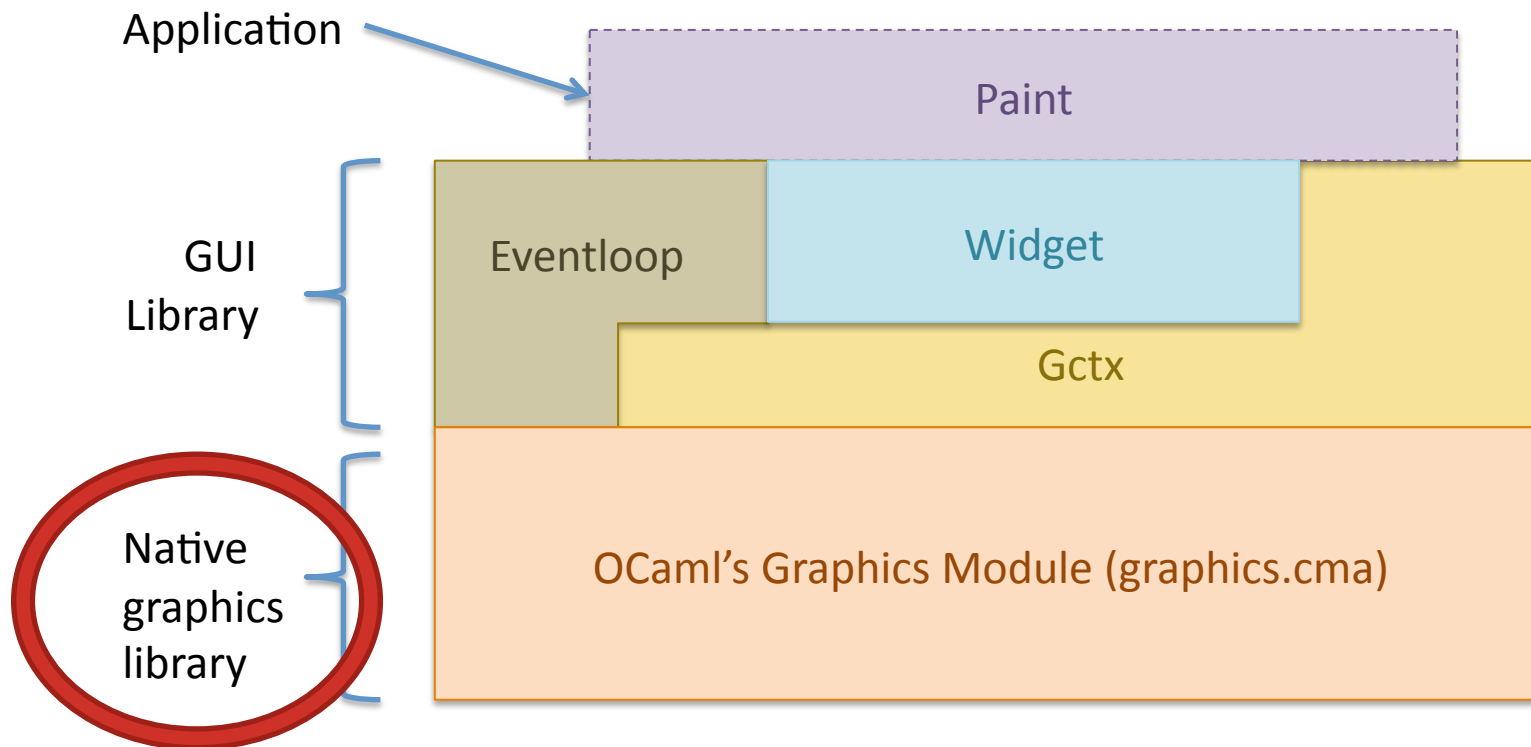
- We don't want to build just one graphical application: we want to make sure that our code is *reusable*.
- What are the concepts involved in GUI libraries and how do they relate to each other?
- How can we separate the various concerns on the project?

Project Architecture



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

Project Architecture



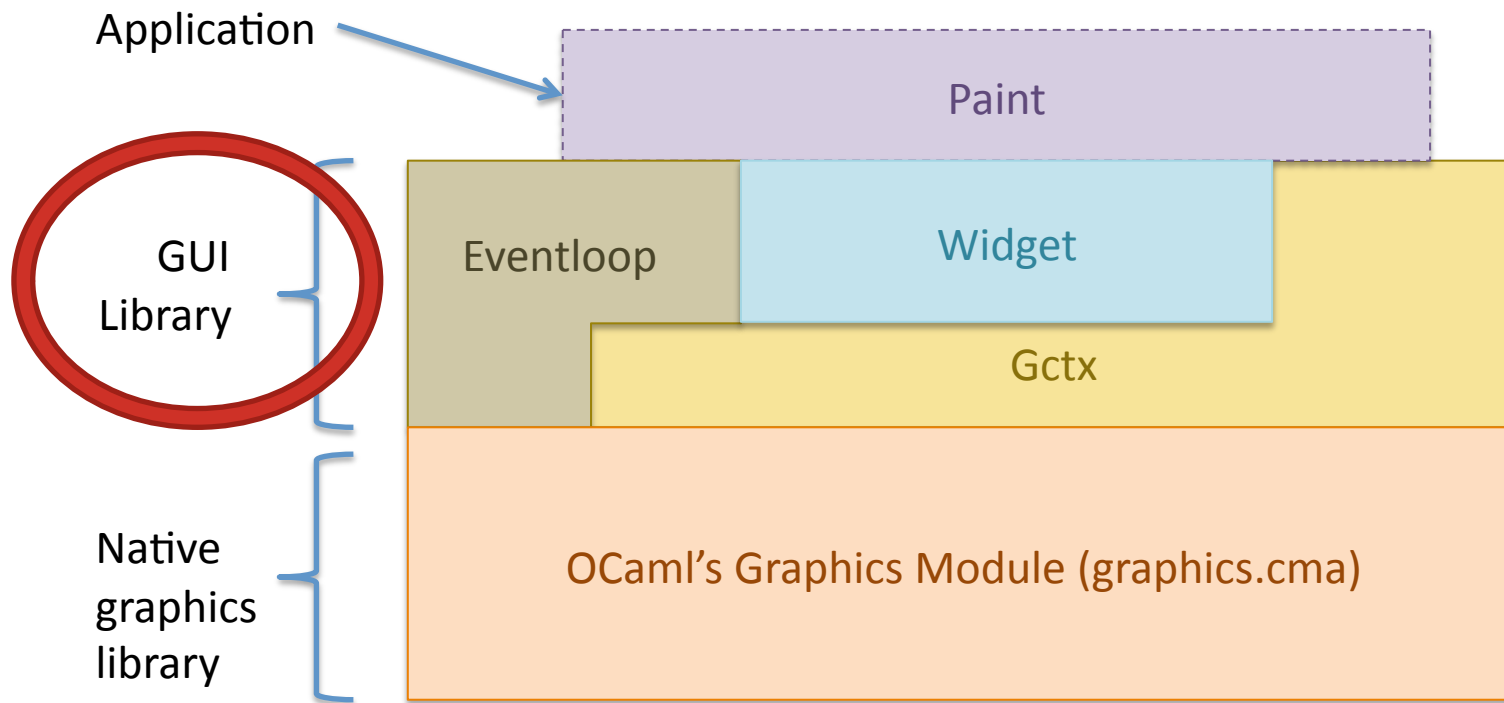
Goal of the GUI library: provide a consistent layer of abstraction *between* the application (Paint) and the Graphics module.

Designing a GUI library

- 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://www.seas.upenn.edu/~cis120/current/ocaml-3.12-manual/libref/Graphics.html>
- How do we go from that to a functioning, reusable GUI library?

*Pragmatic note: when compiling a program that uses the Graphics module, add `graphics.cmxa` (for native compilation) or `graphics.cma` (for bytecode compilation) to OCaml Build Flags under the Projects>Properties dialog in Eclipse.

Project Architecture



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

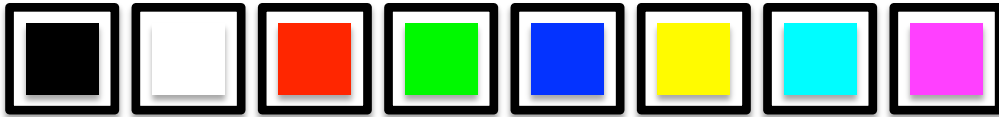
- Main loop of any GUI application

```
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 it 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.

Container Widgets for layout



```
let color_toolbar : Widget.t = 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.

- Challenge: How can we make it so that the functions that draw widgets (buttons, check boxes, text, etc.) in **different places** on the window are location independent?

Challenge: Widget Layout

- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a graphics context to make drawing primitives *relative* to the widget’s local coordinates.

