Programming Languages and Techniques (CIS120)

Lecture 16

Feb 20, 2013

"Objects" and GUI Design

Announcements

- HW05 is due *tomorrow*, Feb 21 at 11:59:59pm

- Please review course collaboration policy on the syllabus, violations of this policy will have consequences

- Midterm 1 has been graded (scores available online)
  - Solutions posted on the course website
  - View your exams with Ms. Laura Fox, Levine 308
  - Regrade requests submitted in writing

- Midterm Course Feedback: Survey in Labs this week
  - please give us your opinions about the class!

Midterm 1 results

- Stats:
  - median: 90.0
  - mean: 87.27
  - var: 110.68
  - stddev: 10.52
  - max: 100.0 (12)

Great Job!

- Grade ranges (estimate):
  - A 90 – 100
  - B 80 – 90
  - C 70 – 80
  - D 60 – 70

Warning:
Midterm 2 is a killer!

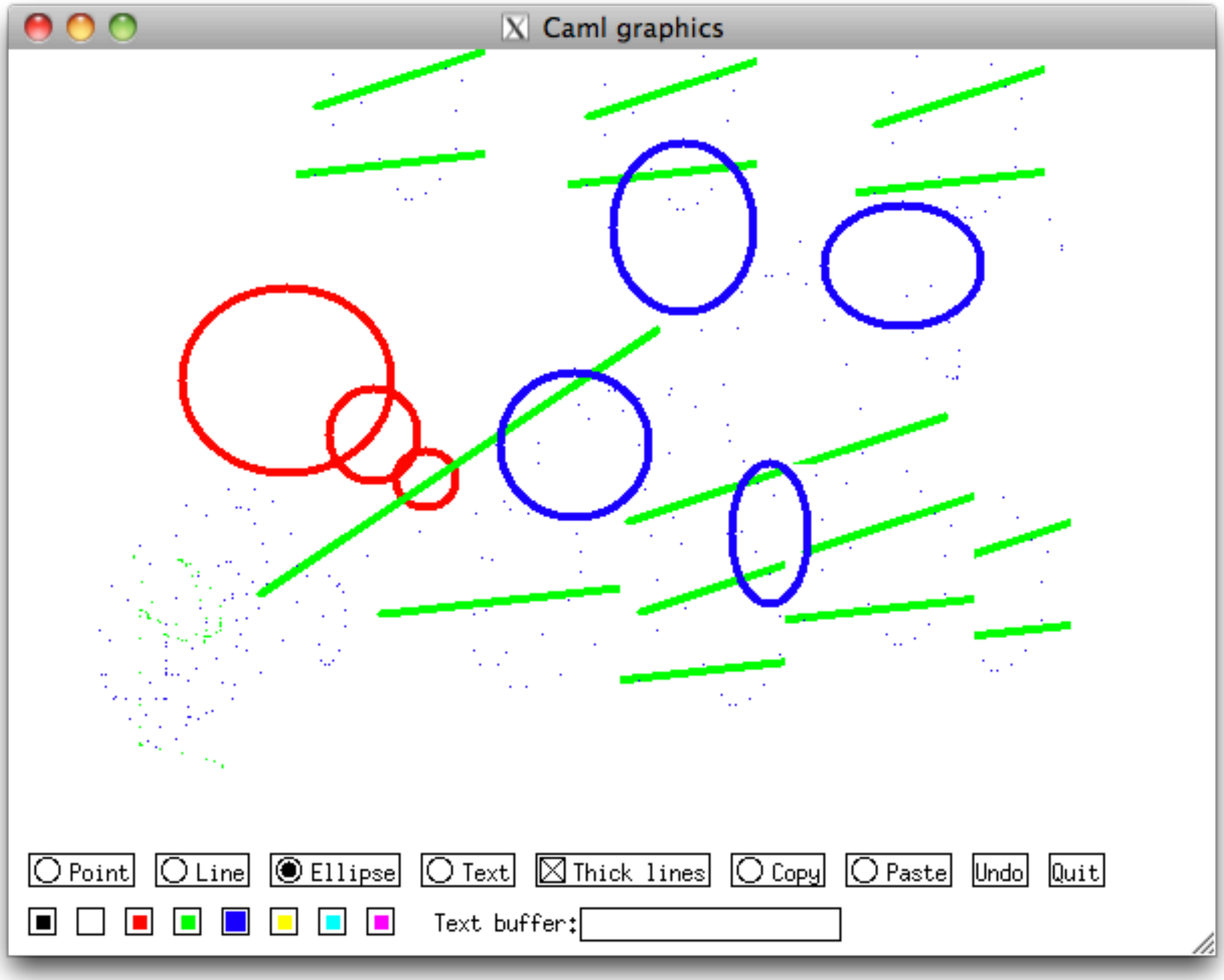
The CIS120 Trajectory

- HW 6: Build a GUI library and client application *from scratch* in OCaml
  - Available Friday
  - Due March 1st
- Several purposes:
  - Show you that you have enough knowledge to do some pretty serious programming
  - Illustrate the *event-driven* programming model
  - Practice with first-class functions and *hidden state*
  - Bridge to object-oriented programming (i.e. Java)
  - Give you a feel for how GUI libraries (like Java's Swing) work
- Afterwards: transition to Java

Demo: GUI Paint Application

HW 6

Building a GUI and GUI Applications



“Objects” and Hidden State

Objects in Java

class declaration

```
public class Counter {

   private int count;

   public Counter () {
     count = 0;
   }

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

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

class name

instance variable

constructor

methods

object creation and use

```
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter();

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

   }
}
```

constructor invocation

method call

What is an Object?

- Object = Instance variables (fields) + Methods
  - Fields = 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 have several *instances*, which are modified independently.

- 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 can be only one counter. If we want another, we need another counter_state value and *another* function.
  - *No encapsulation*: Any other function can modify count, too.

Using Hidden State

- Make a function that creates the counter state and update 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 ()
```

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

Workspace

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

Stack

mk_incr	

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

Heap

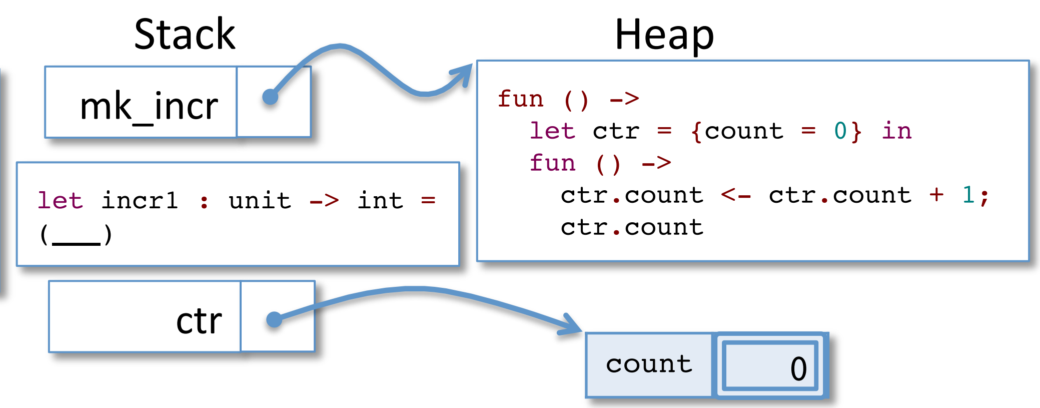
```
fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

count	0

Running mk_incr

Workspace

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



Running mk_incr

Workspace

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

Stack

mk_incr

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

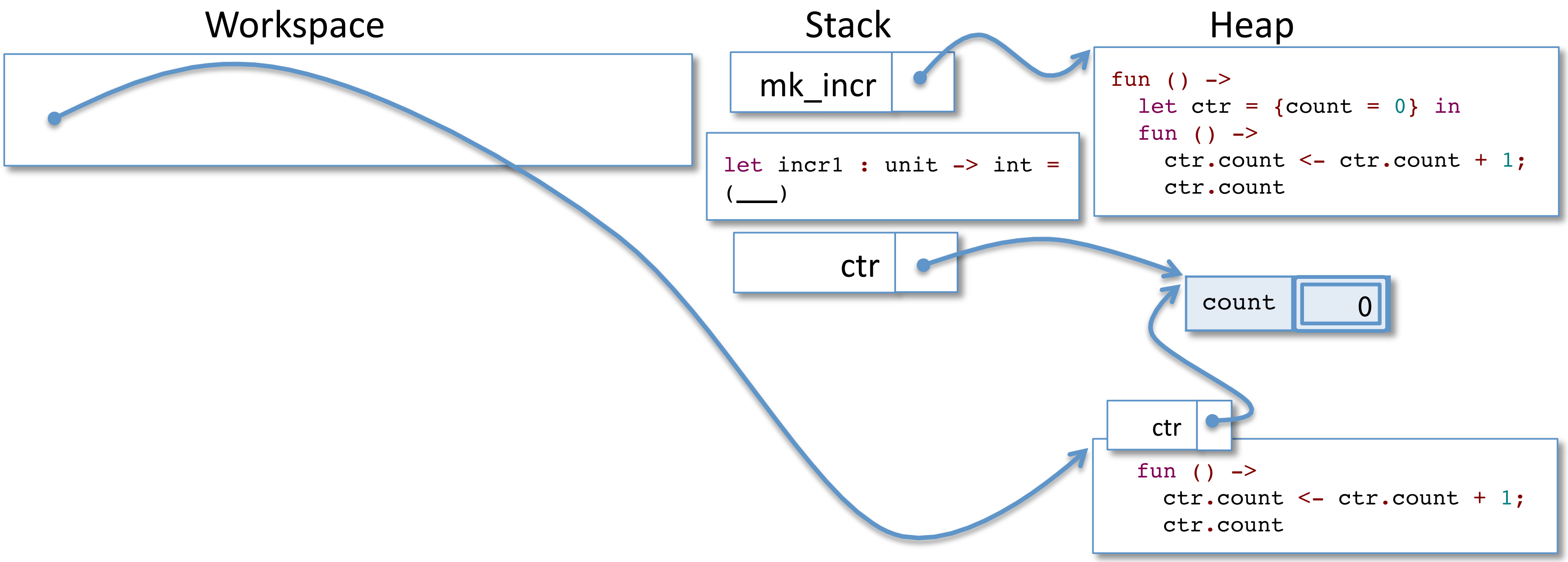
ctr

Heap

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

count 0

Local Functions

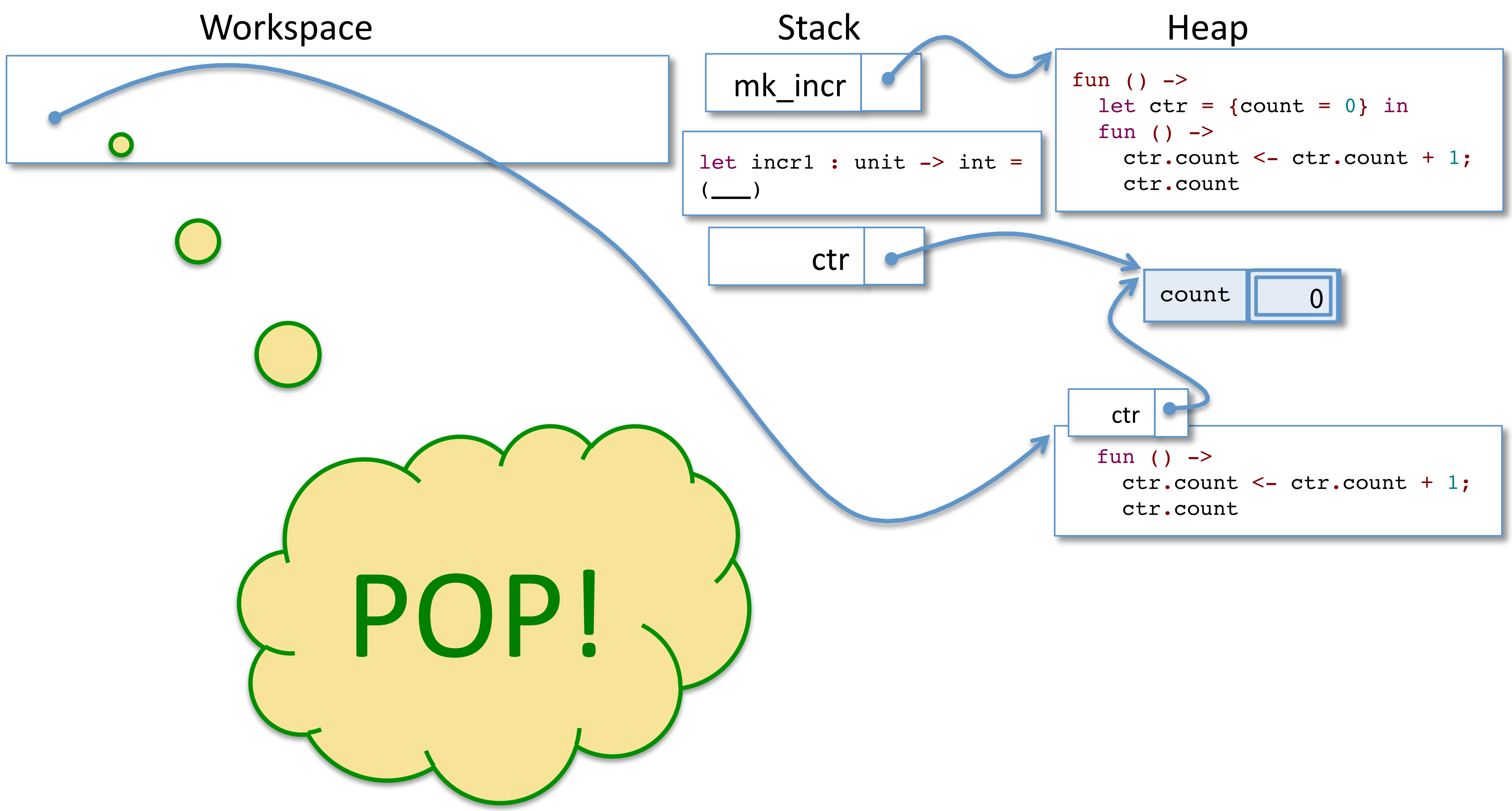


NOTE: We need one refinement to handle local functions. Why?

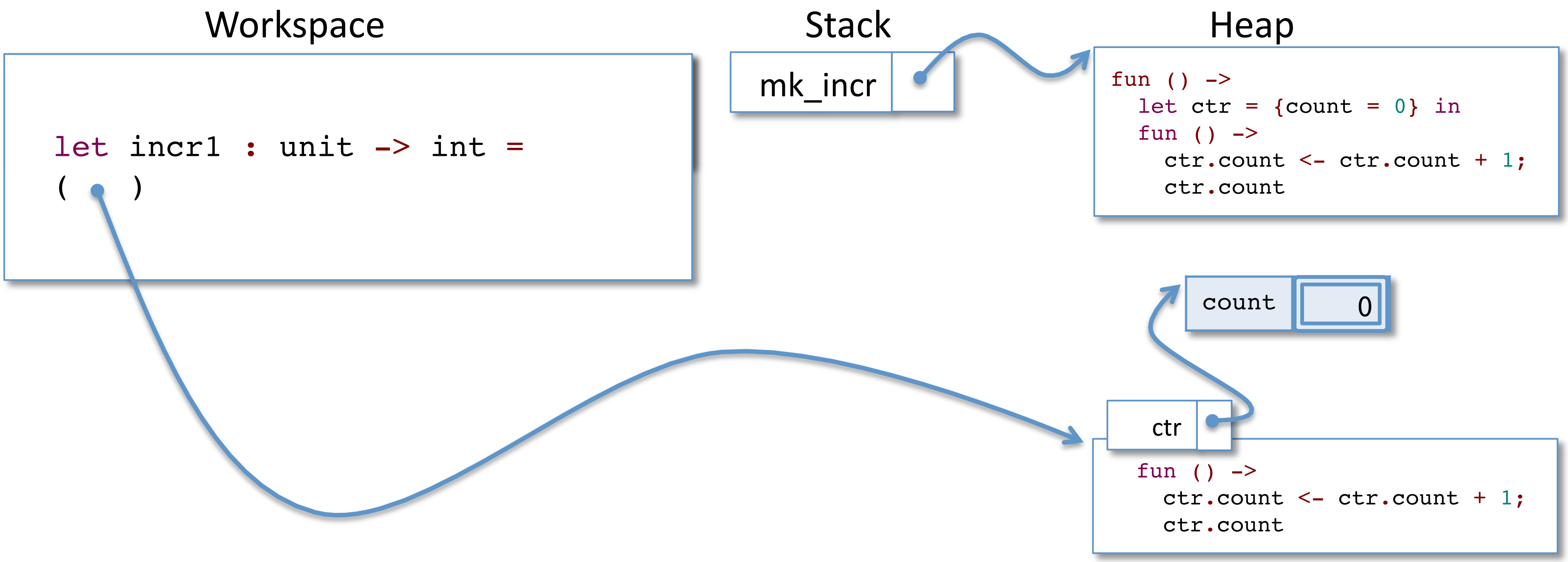
The function mentions “ctr”, which is on the stack (but about to be popped off)…

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

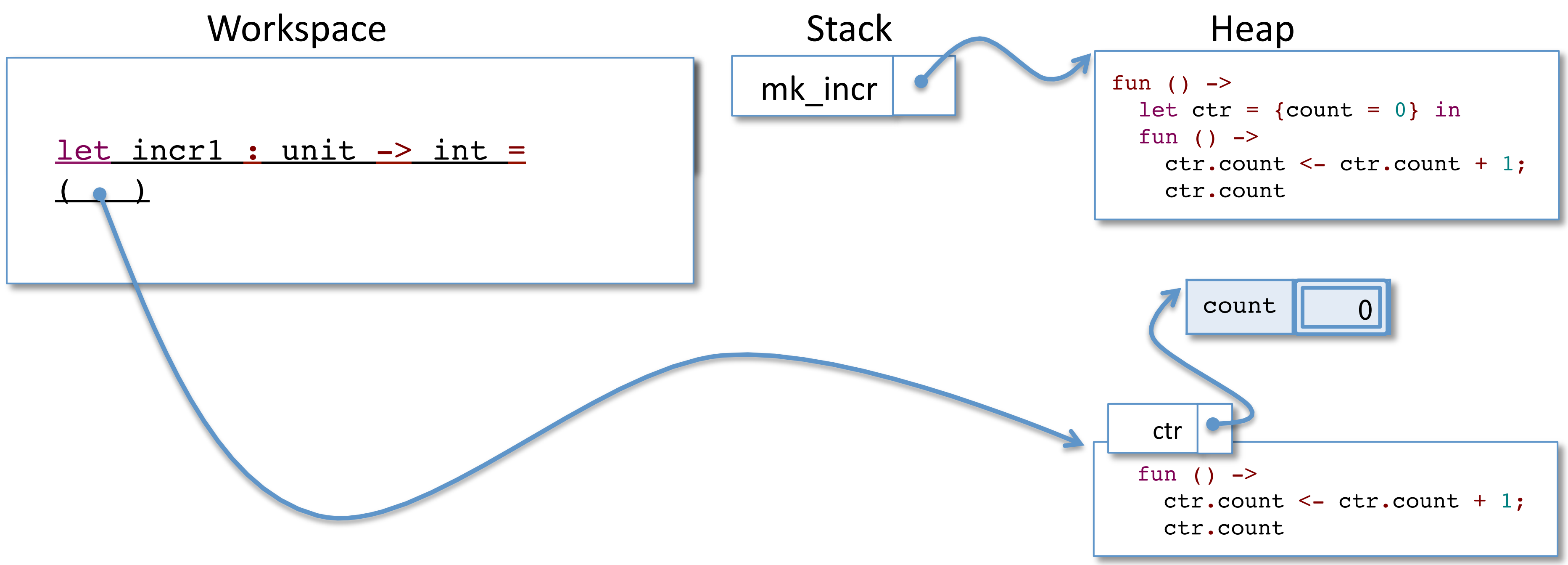
Local Functions



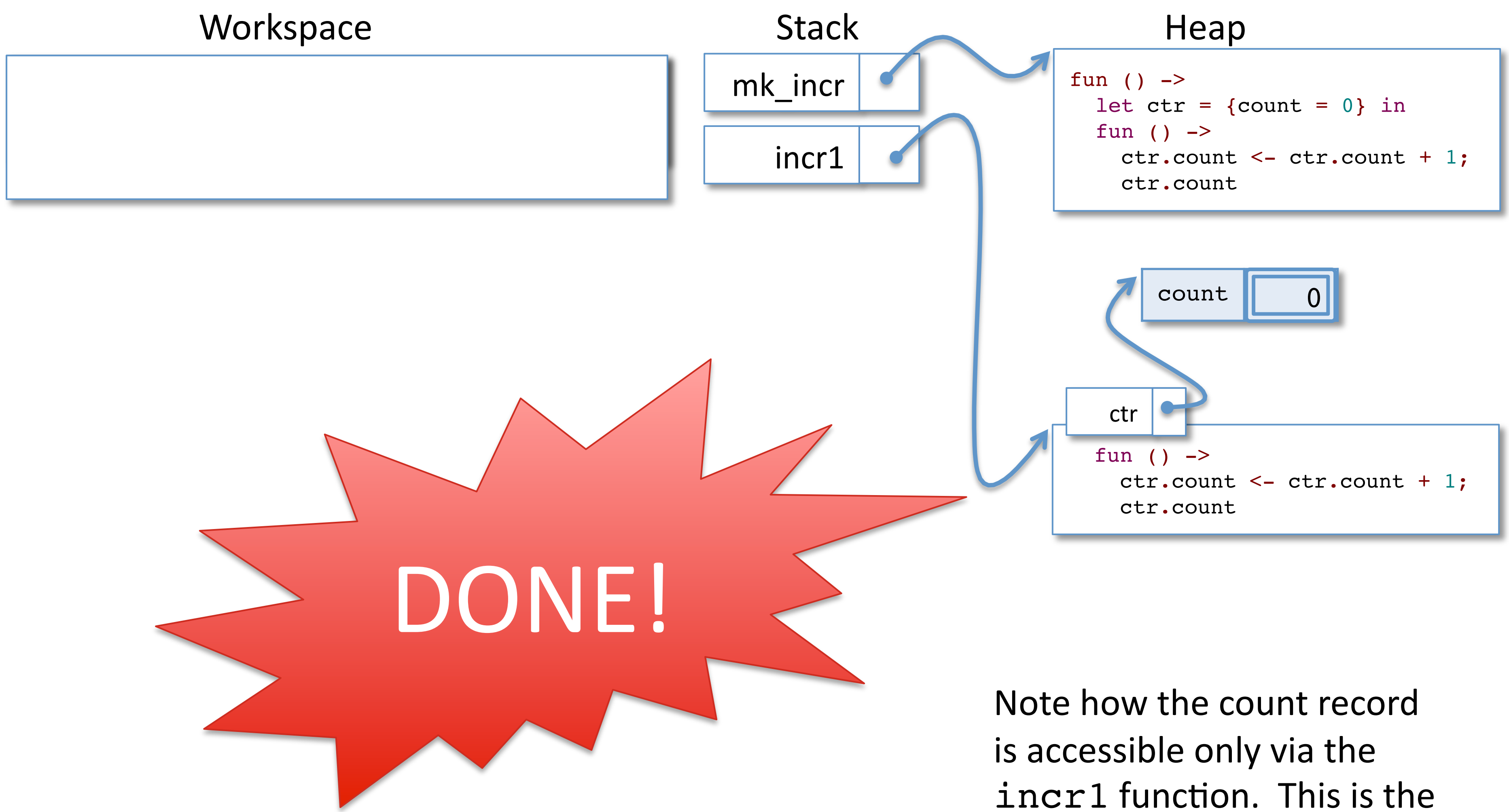
Local Functions



Local Functions

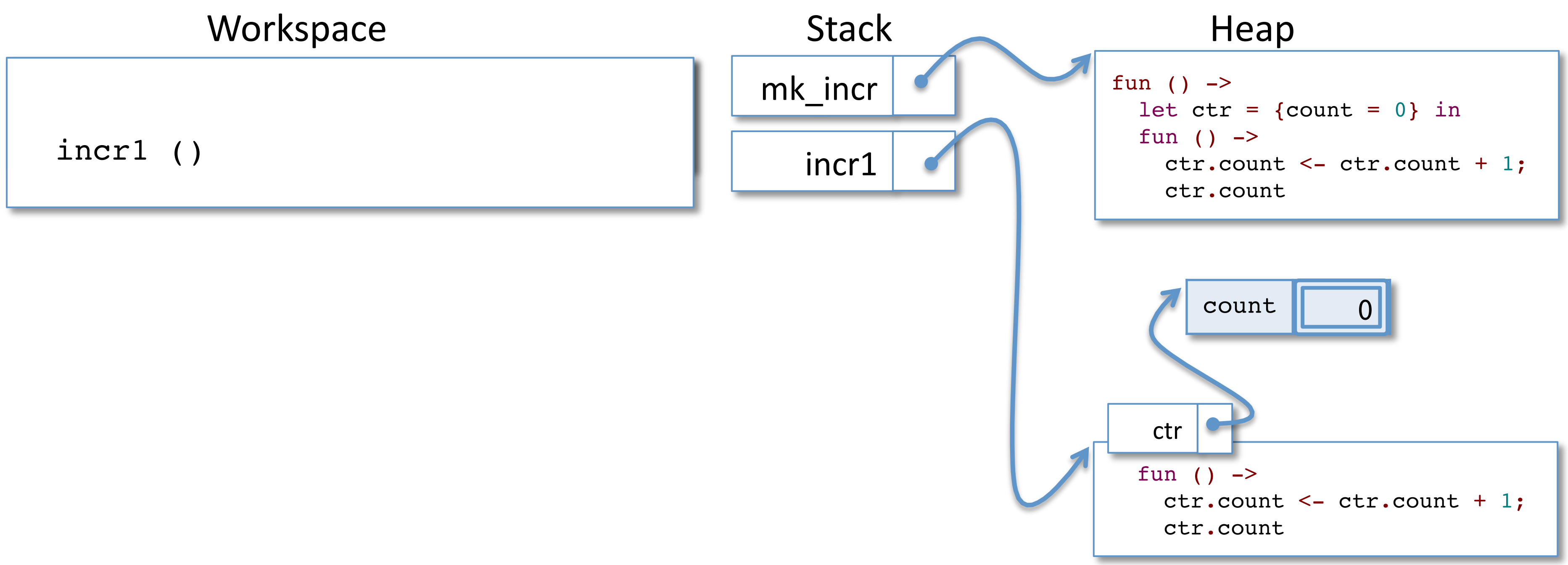


Local Functions

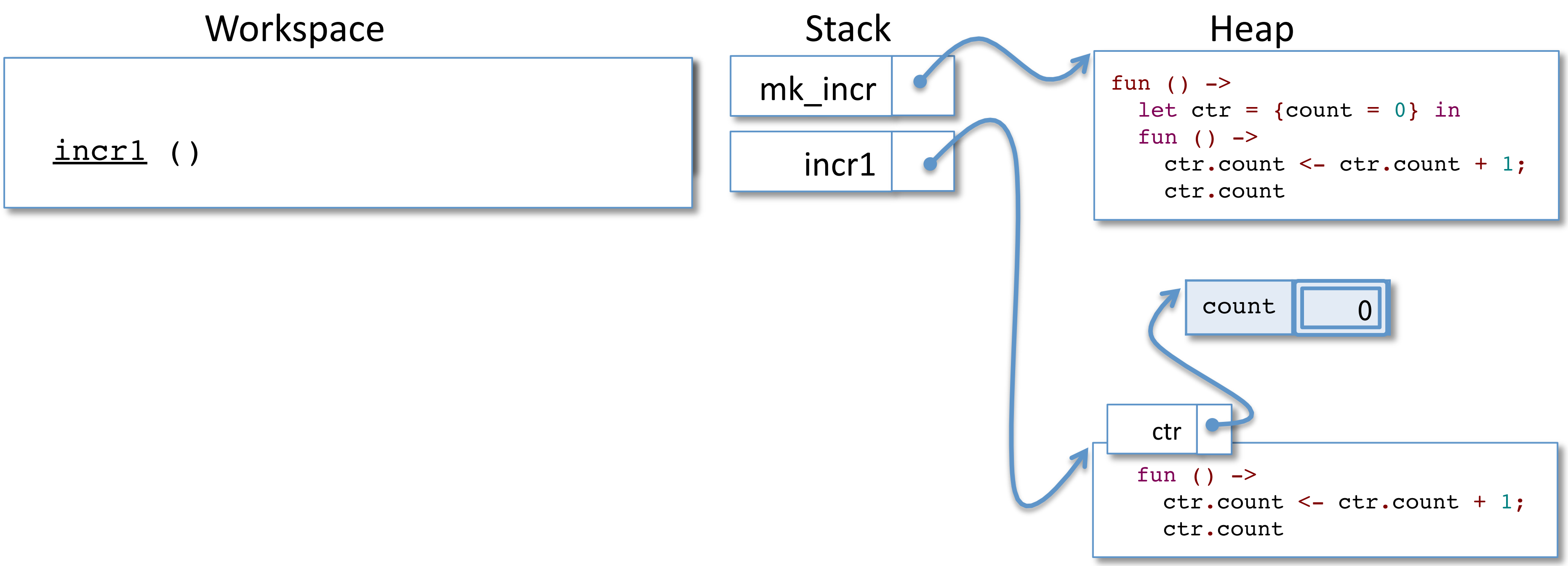


Note how the count record is accessible only via the `incr1` function. This is the sense in which the state is "local" to `incr1`.

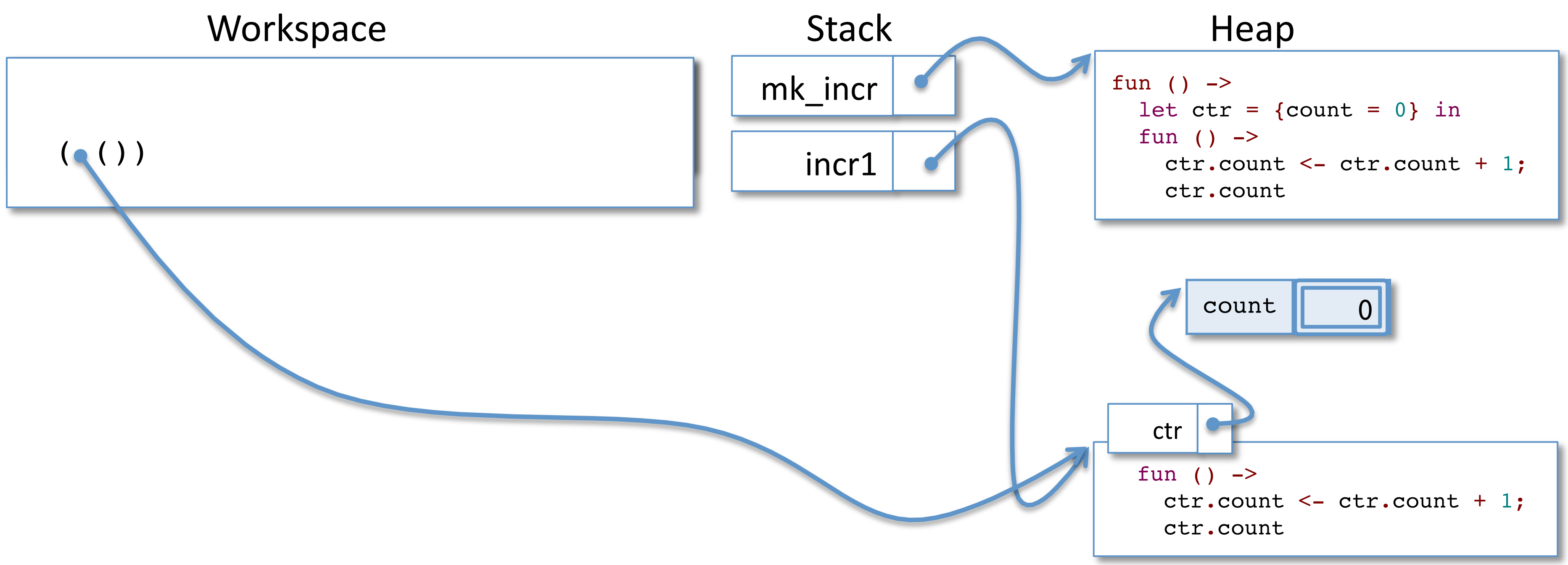
Now let's run "`incr1 ()`"



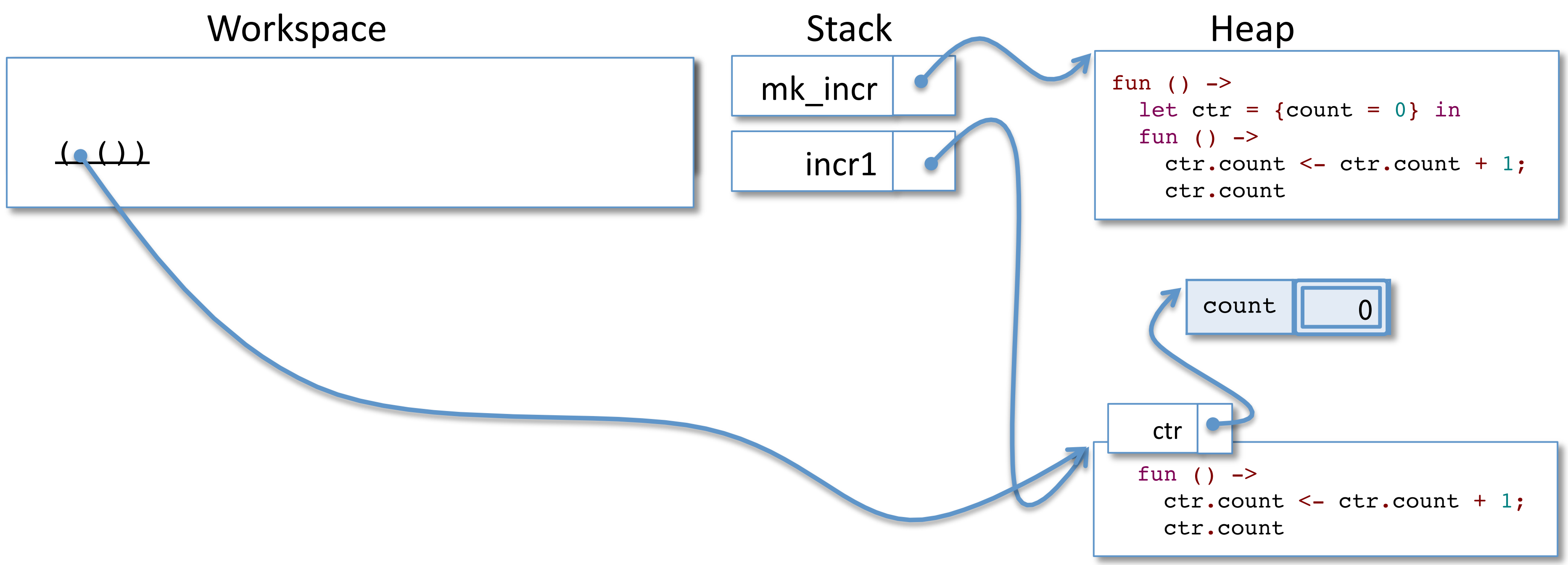
Now let's run "`incr1 ()`"



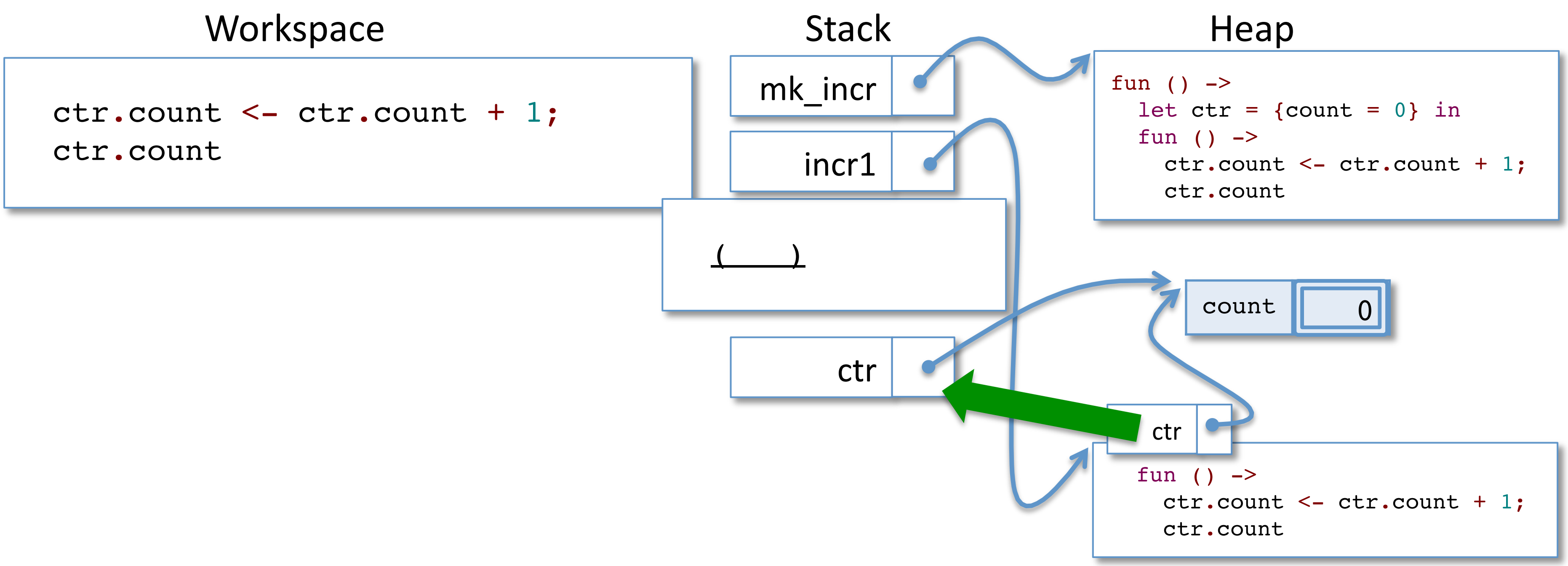
Now let's run "`incr1 ()`"



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



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

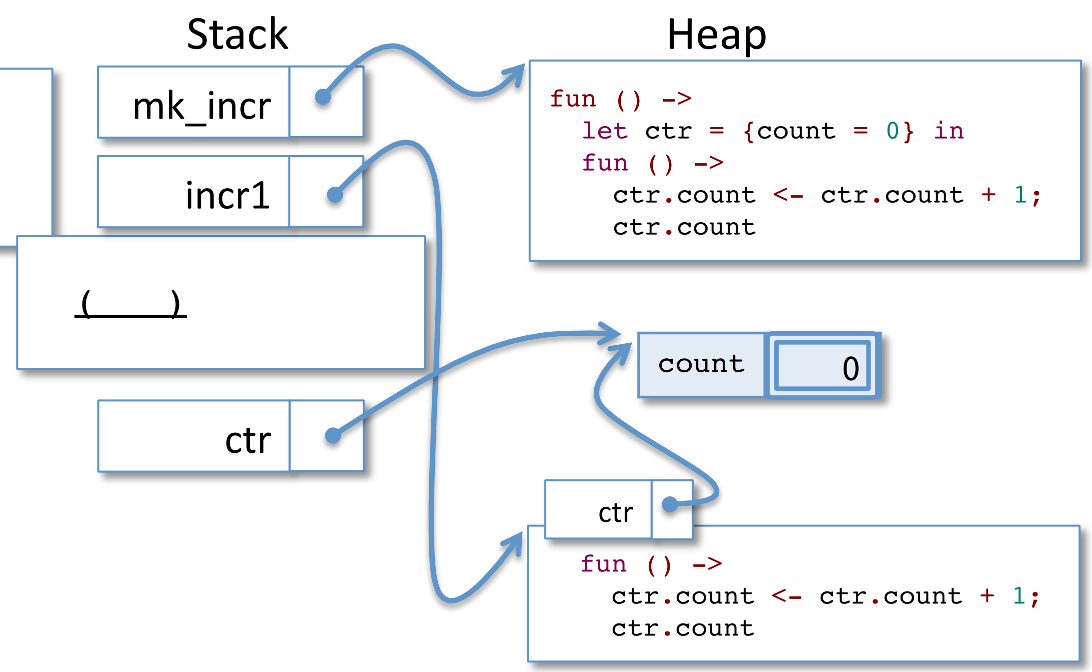


NOTE: Since the function had some saved stack bindings, we add them to the stack at the same time that we put the code in the workspace.

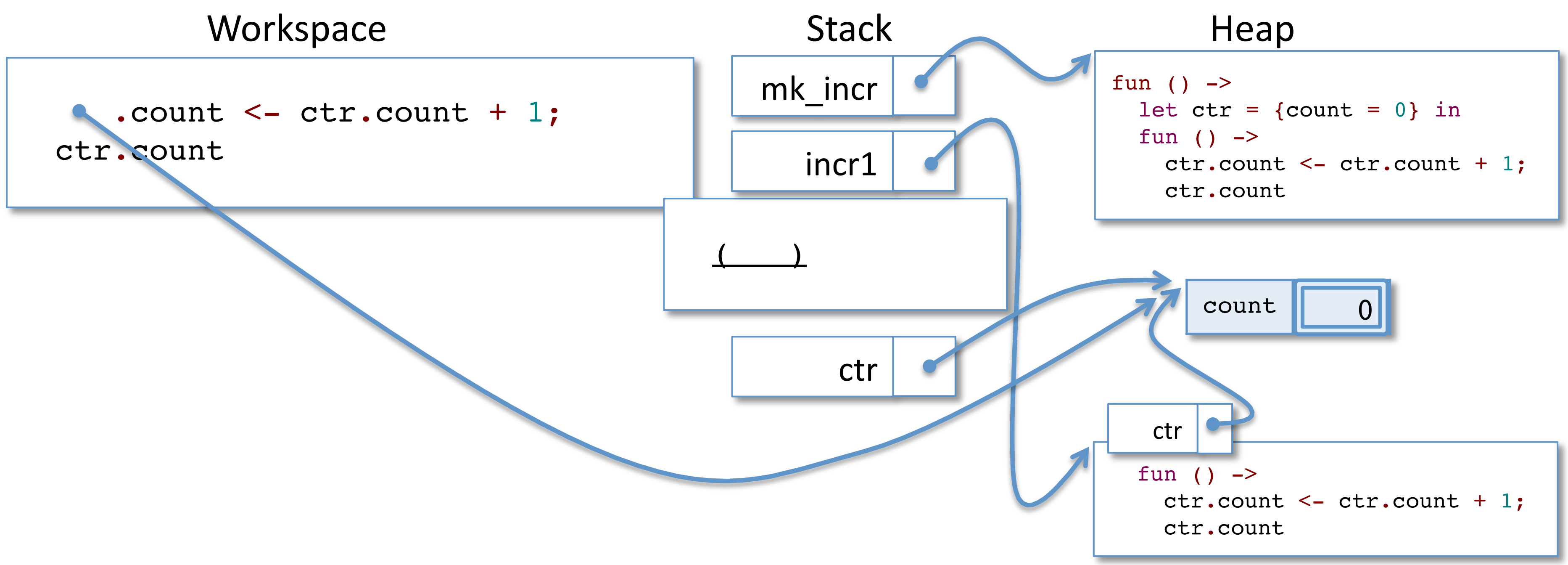
Now let's run "`incr1 ()`"

Workspace

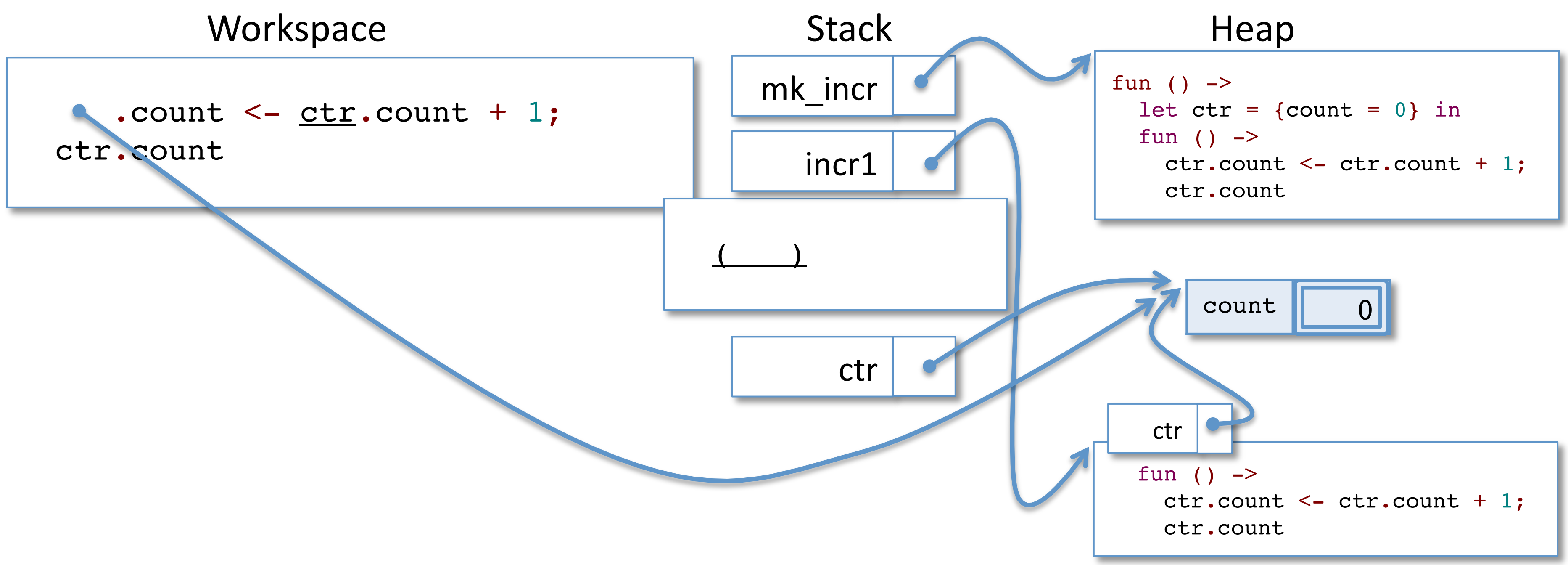
```
ctr.count <- ctr.count + 1;
ctr.count
```



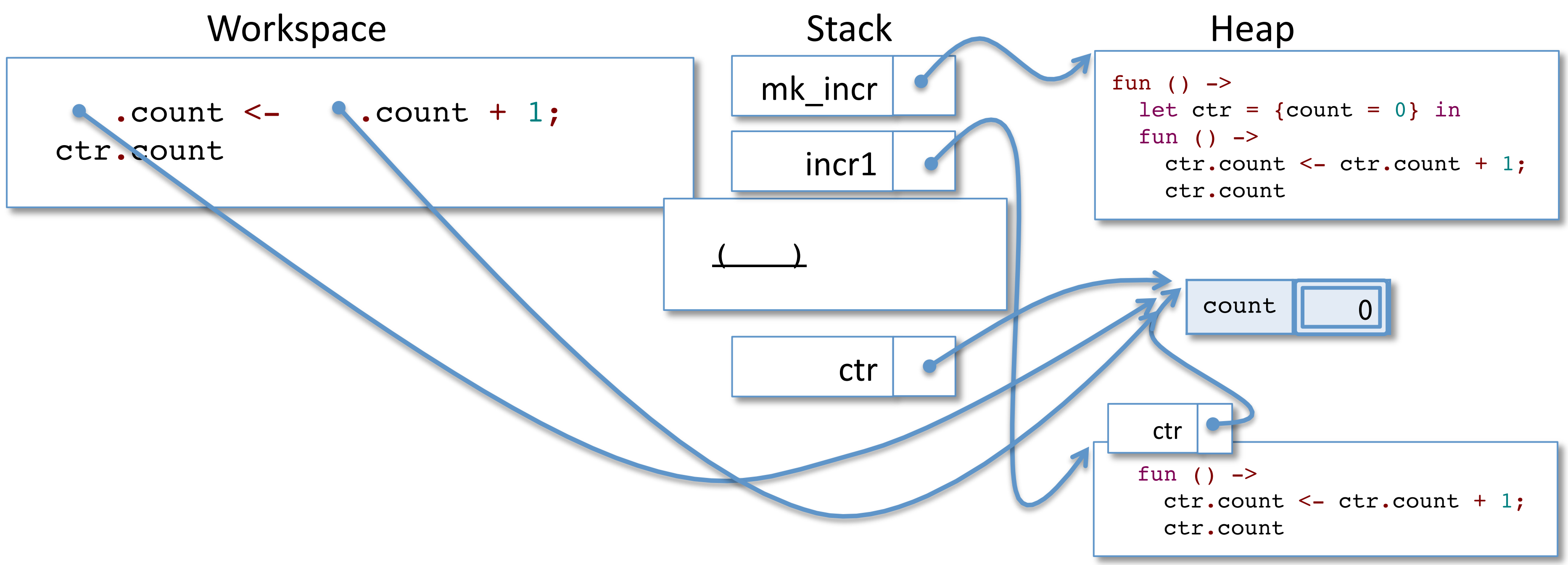
Now let's run "`incr1 ()`"



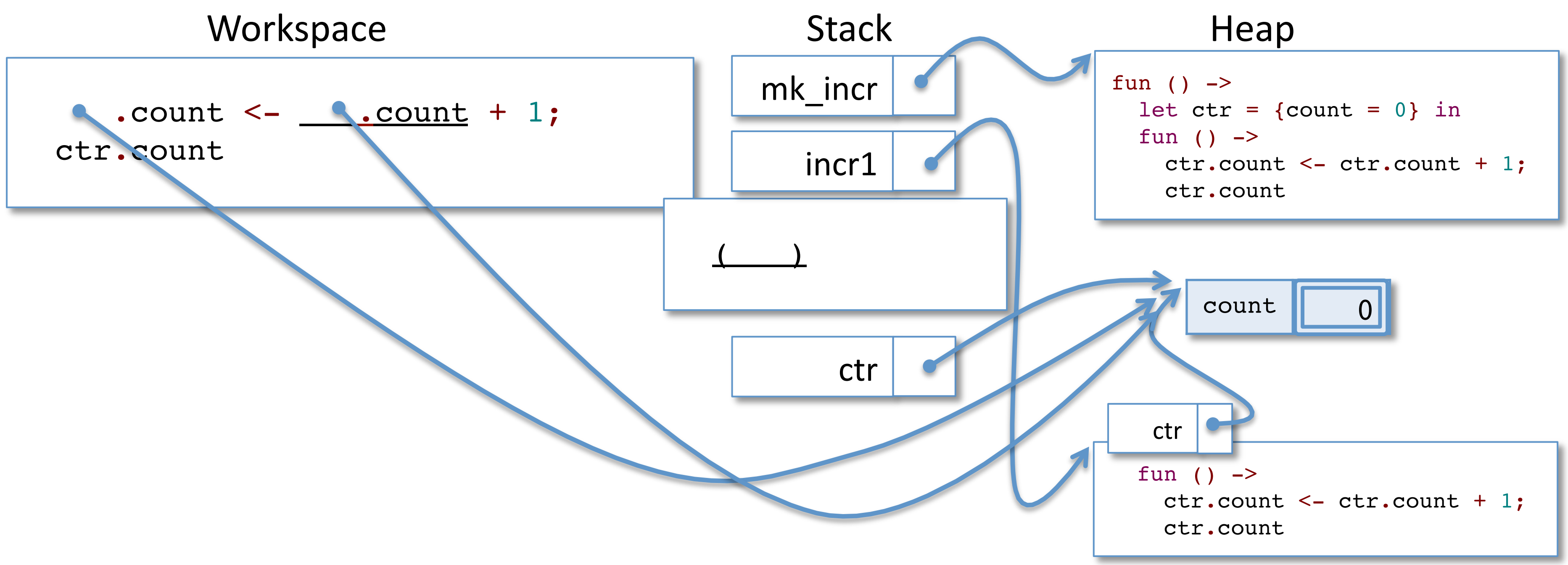
Now let's run "`incr1 ()`"



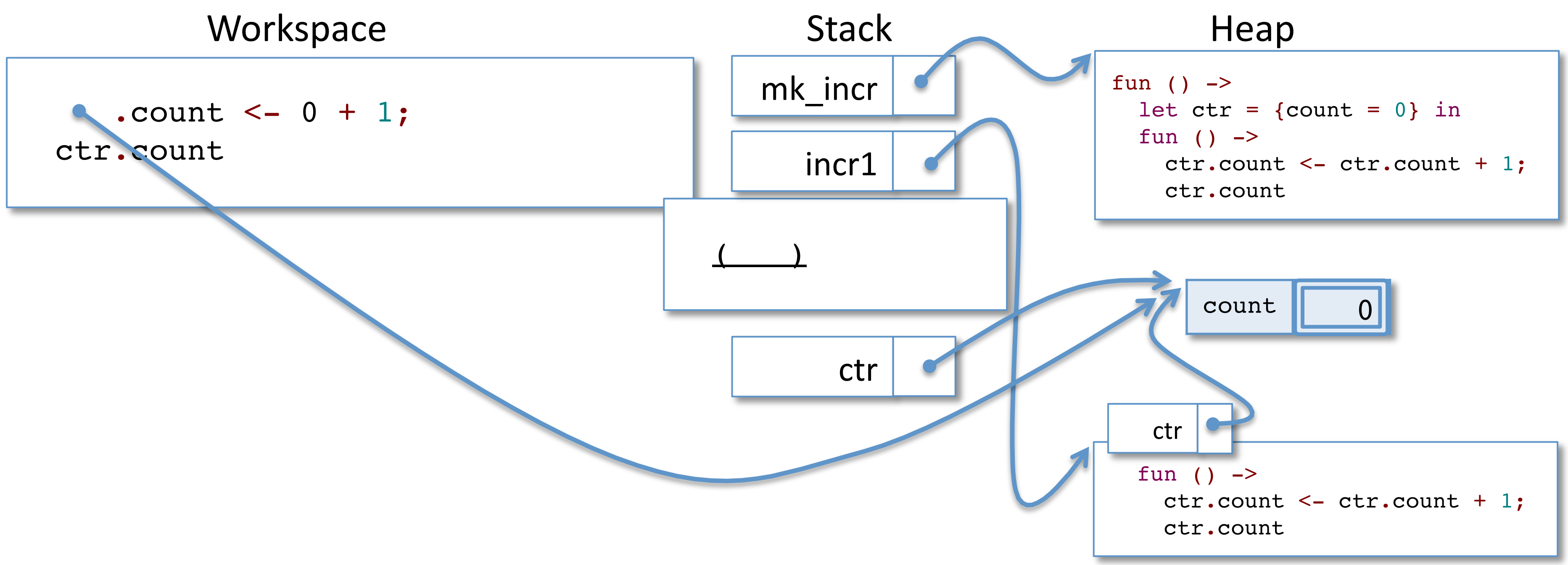
Now let's run "incr1 ()"



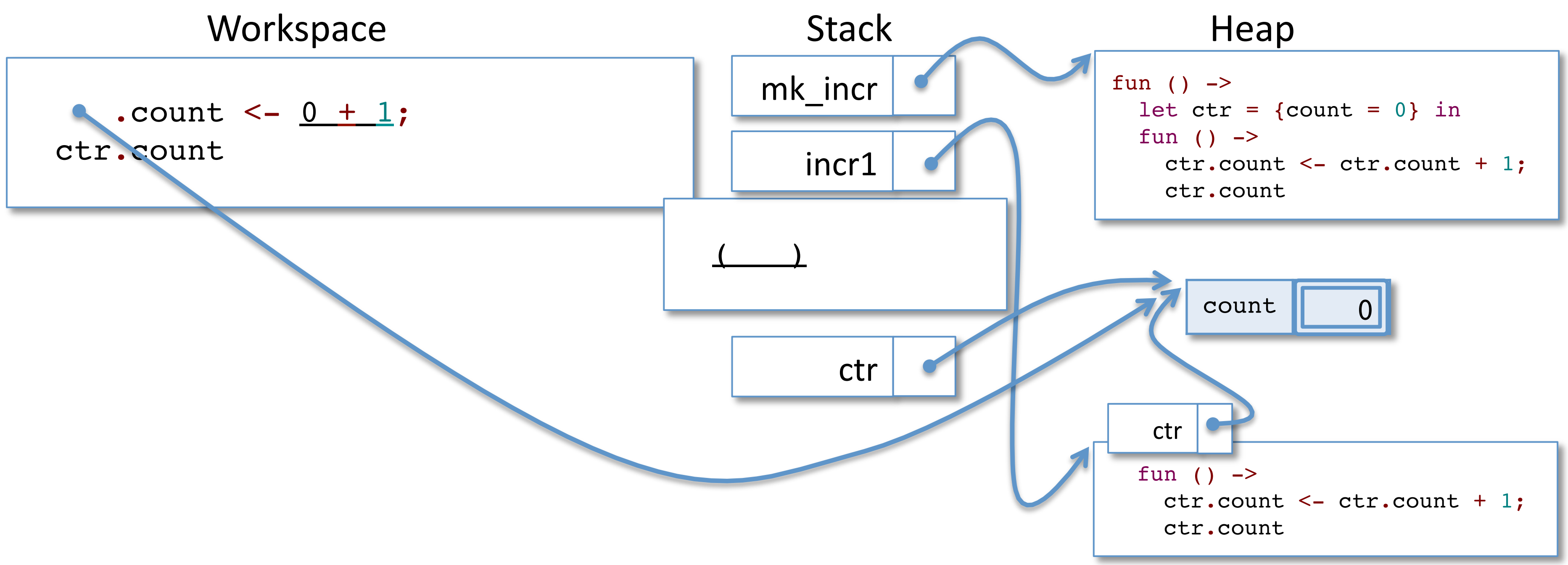
Now let's run "`incr1 ()`"



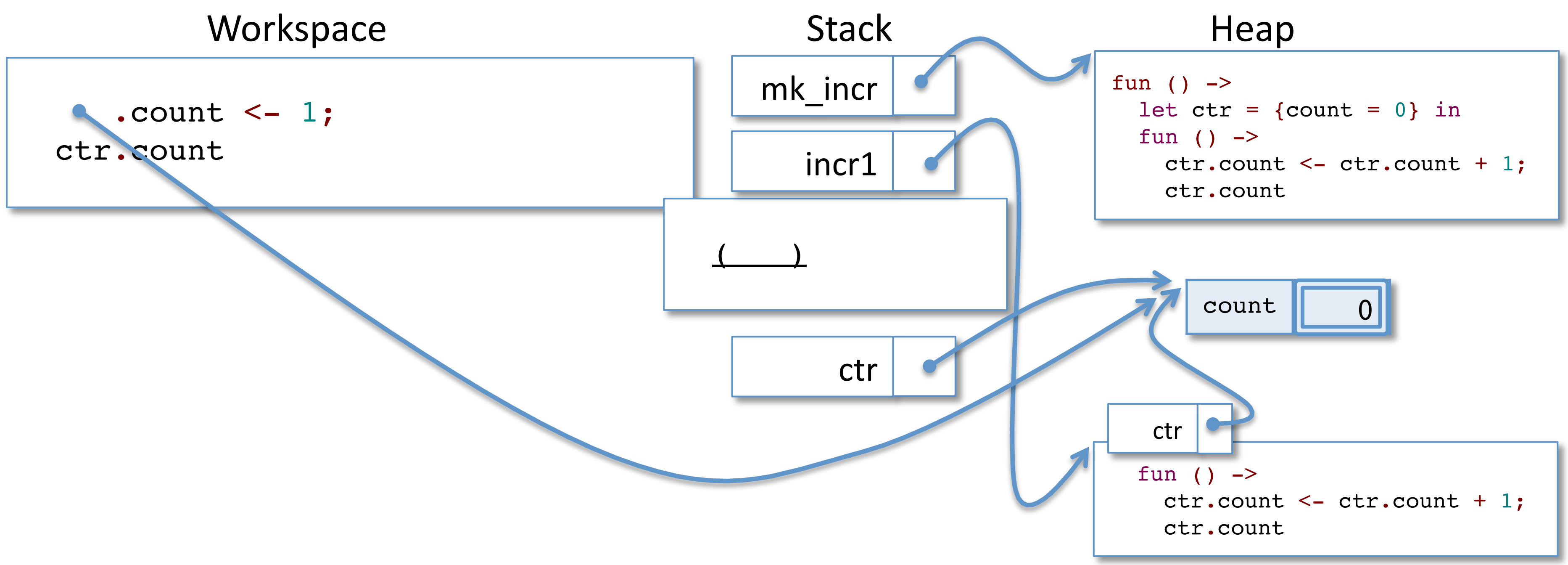
Now let's run "`incr1 ()`"



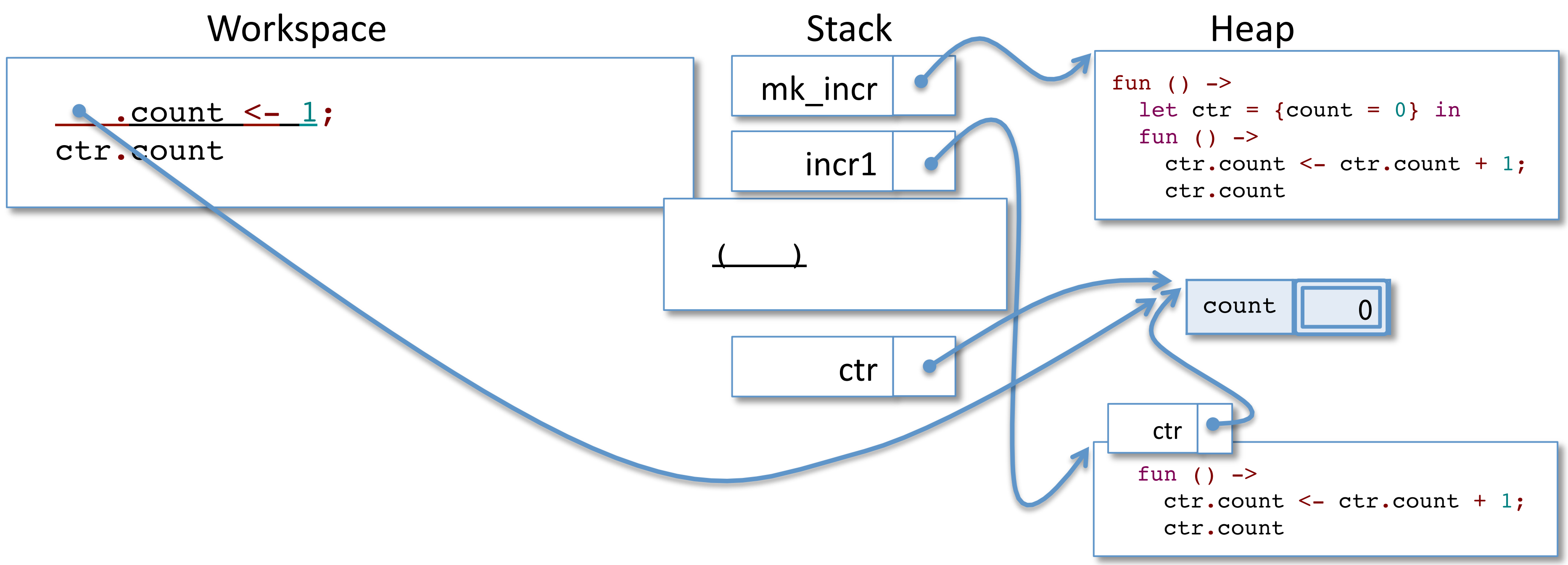
Now let's run "`incr1 ()`"



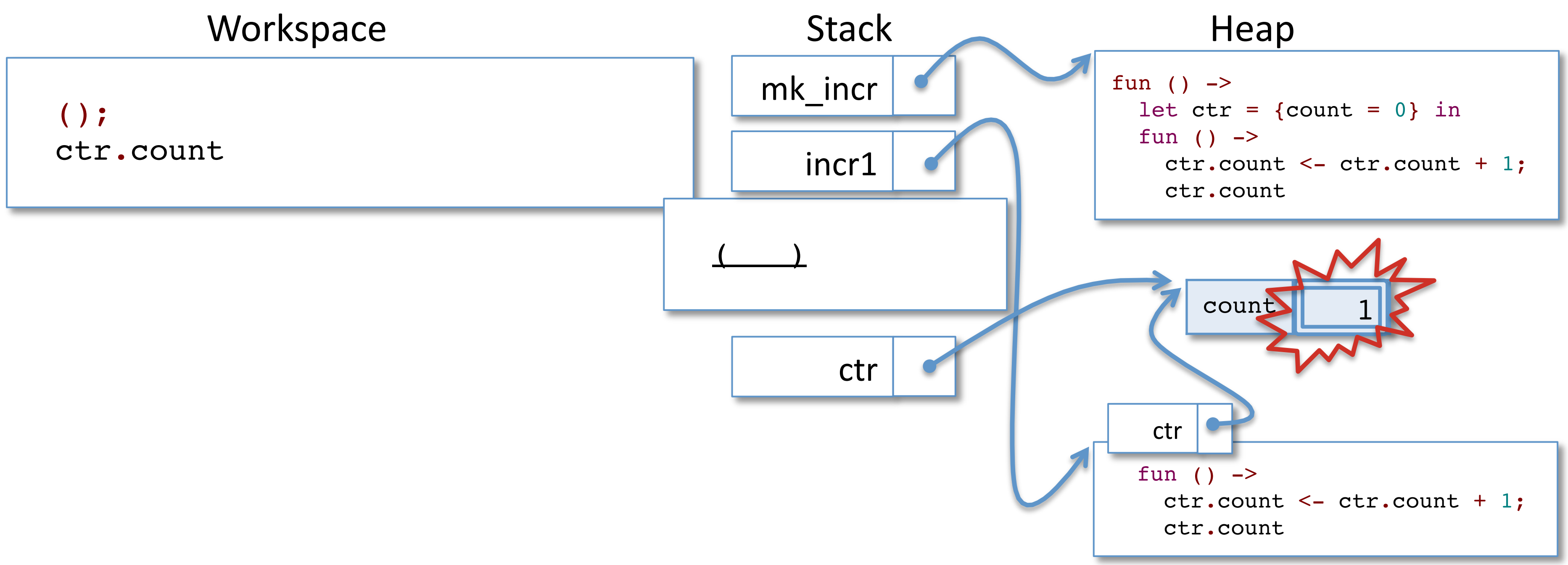
Now let's run "`incr1 ()`"



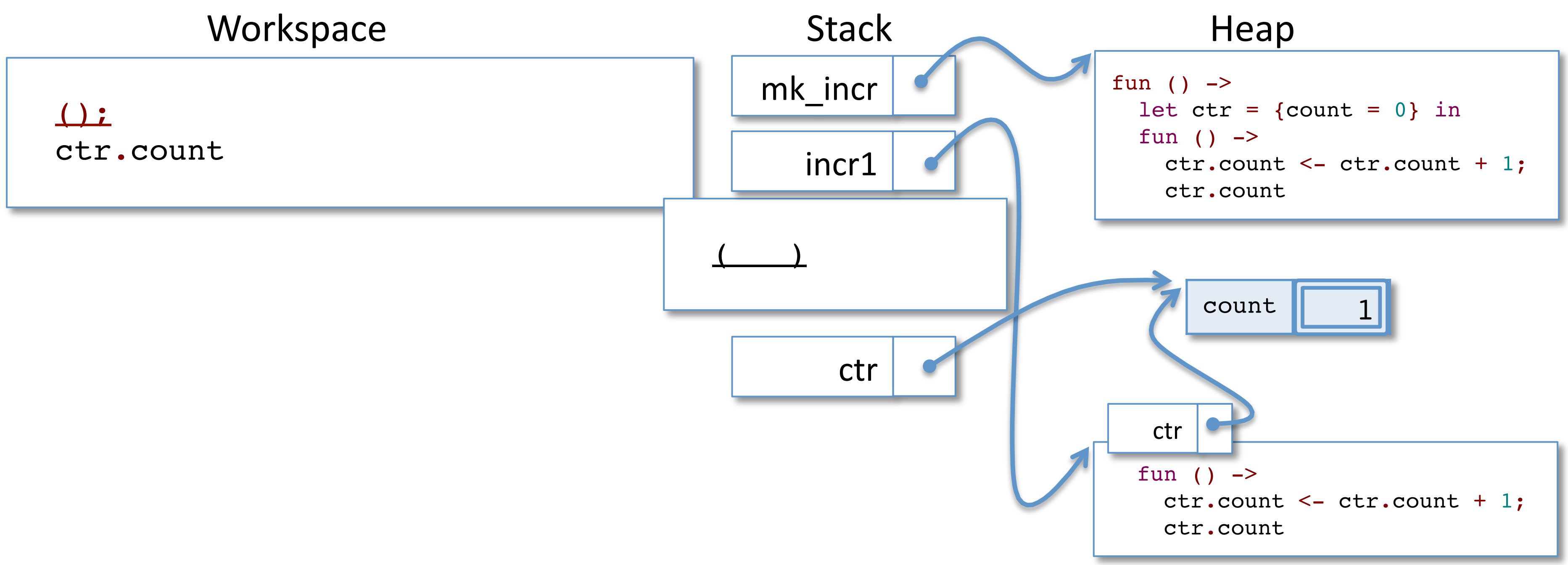
Now let's run "`incr1 ()`"



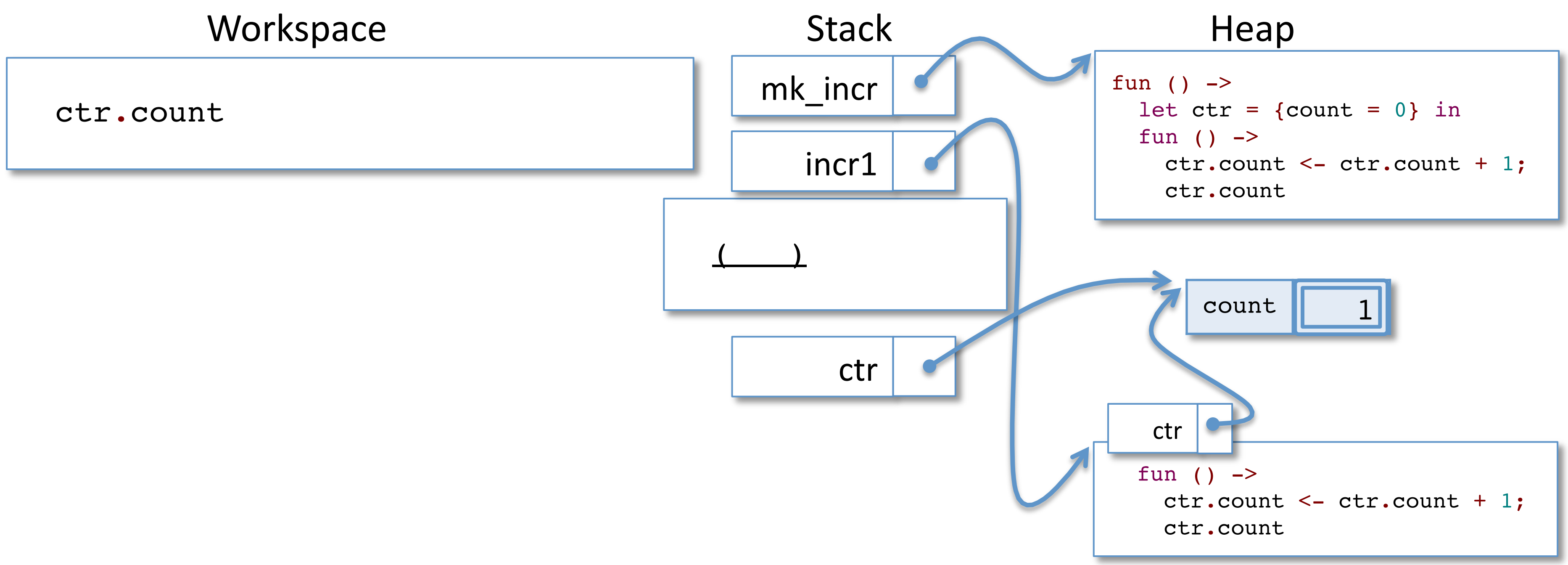
Now let's run "`incr1 ()`"



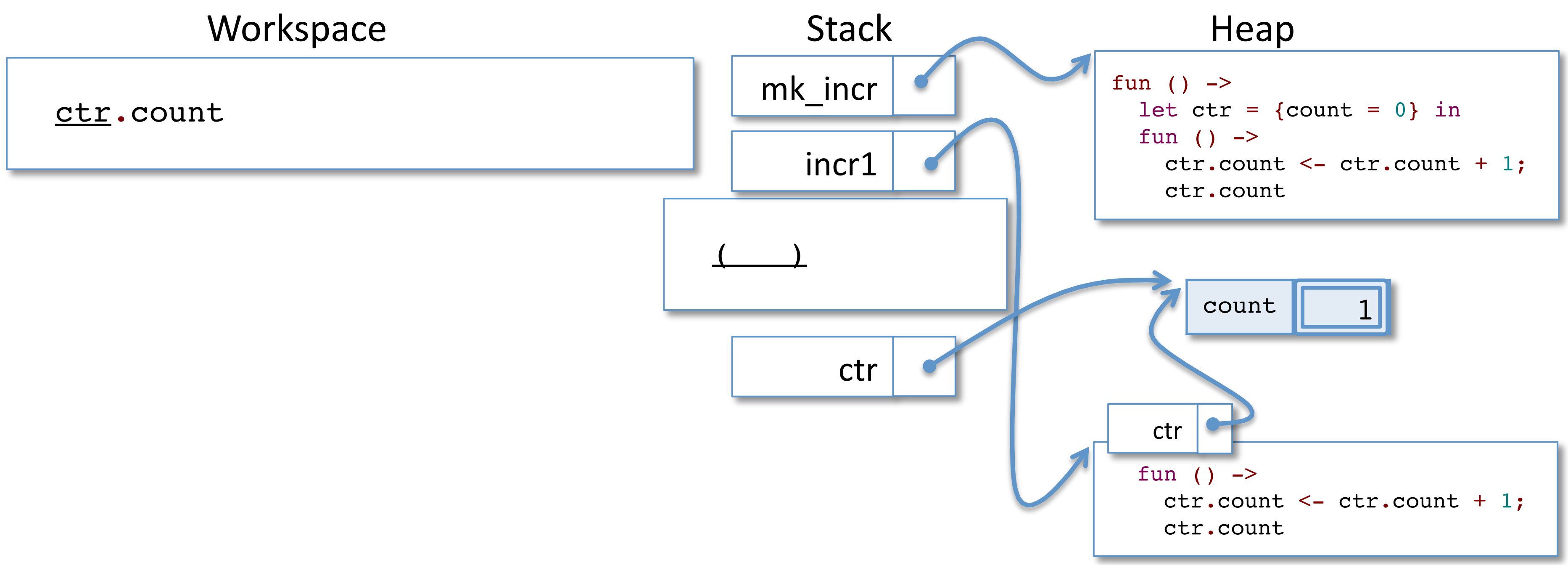
Now let's run "incr1 ()"



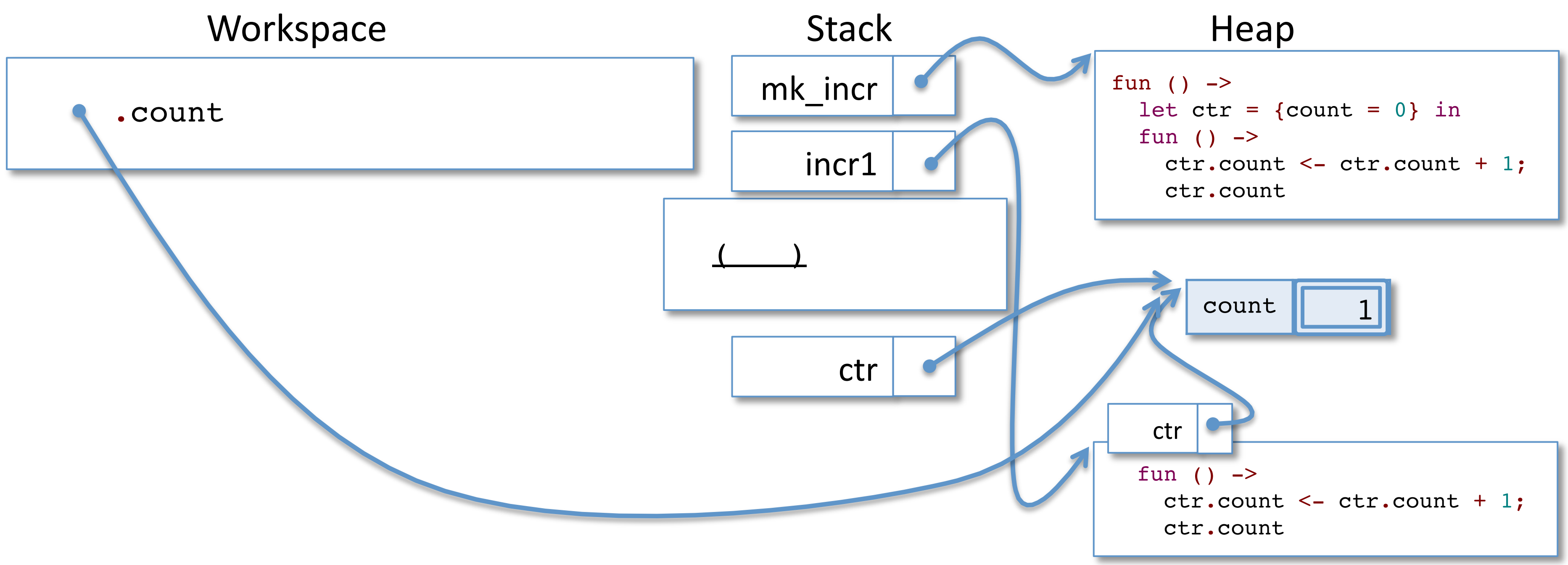
Now let's run "`incr1 ()`"



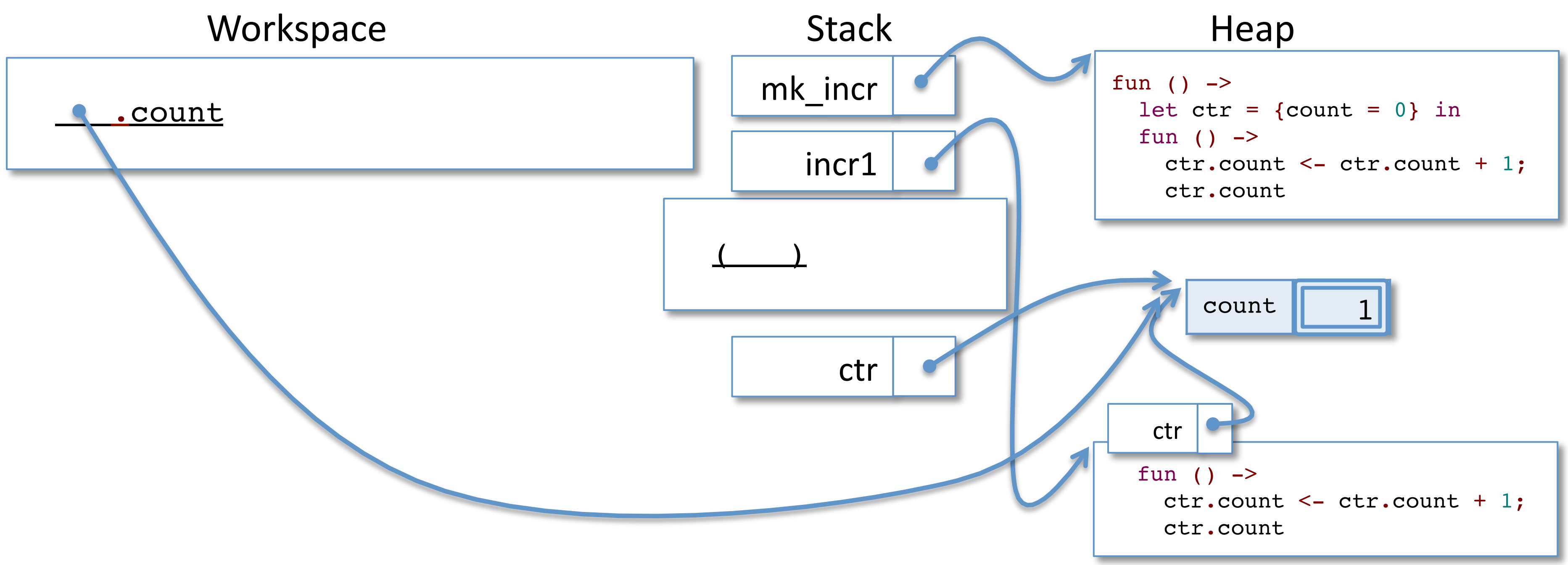
Now let's run "`incr1 ()`"



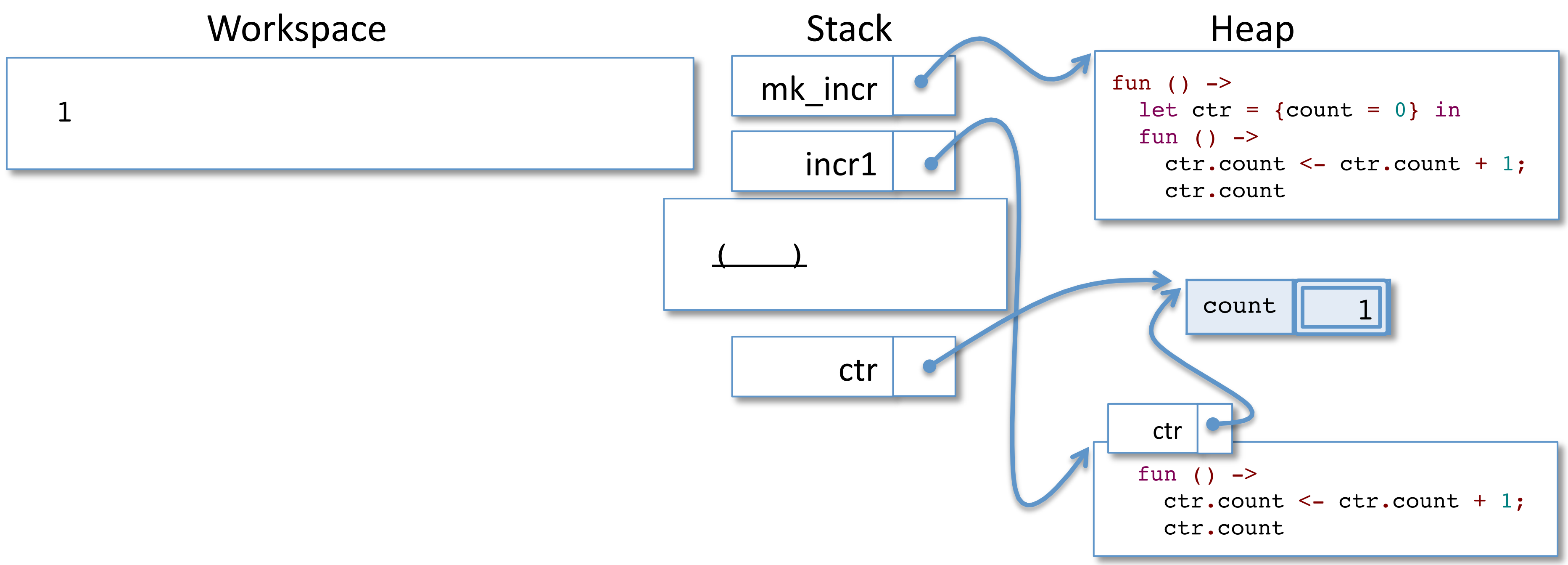
Now let's run "`incr1 ()`"



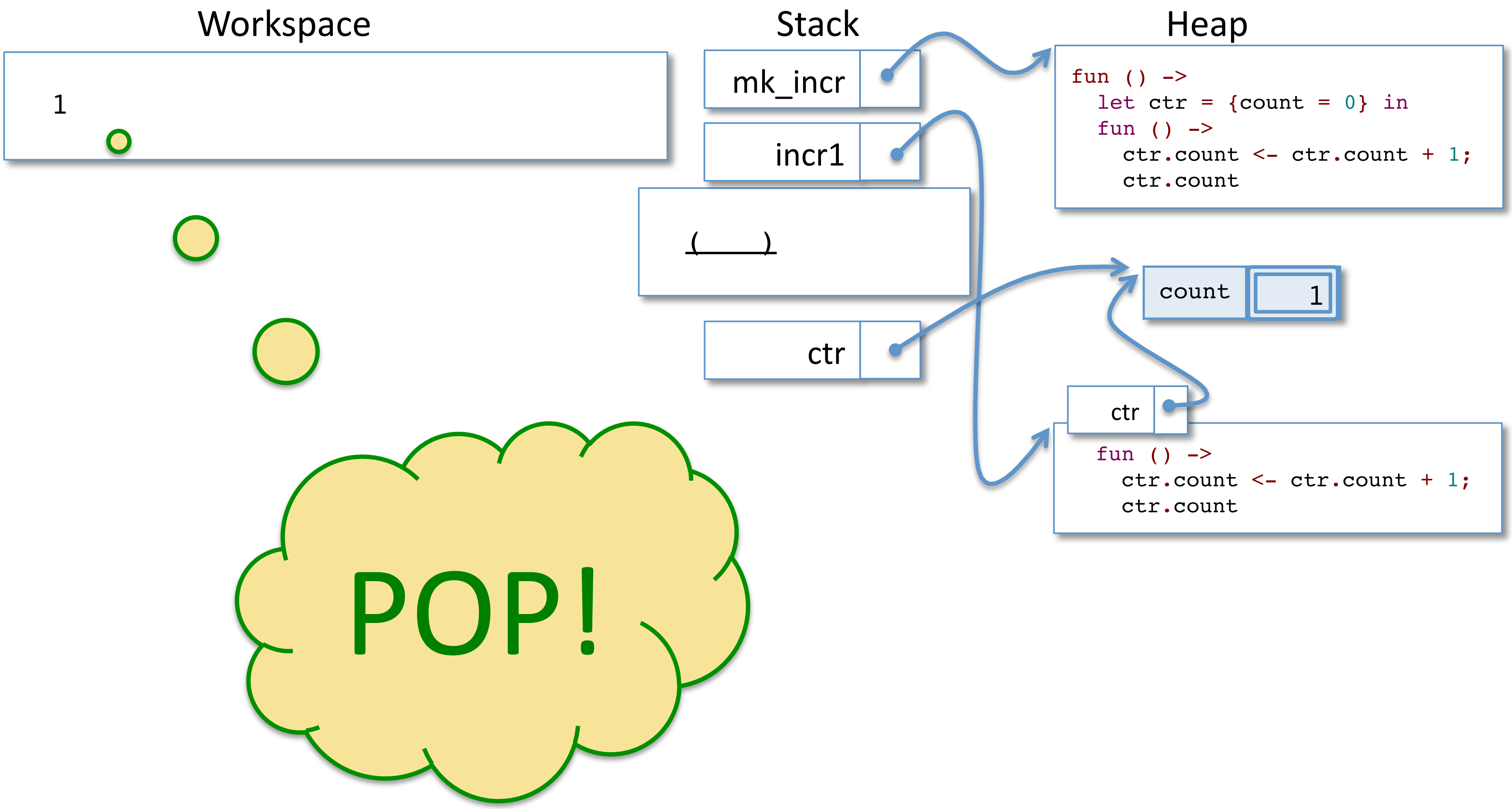
Now let's run "`incr1 ()`"



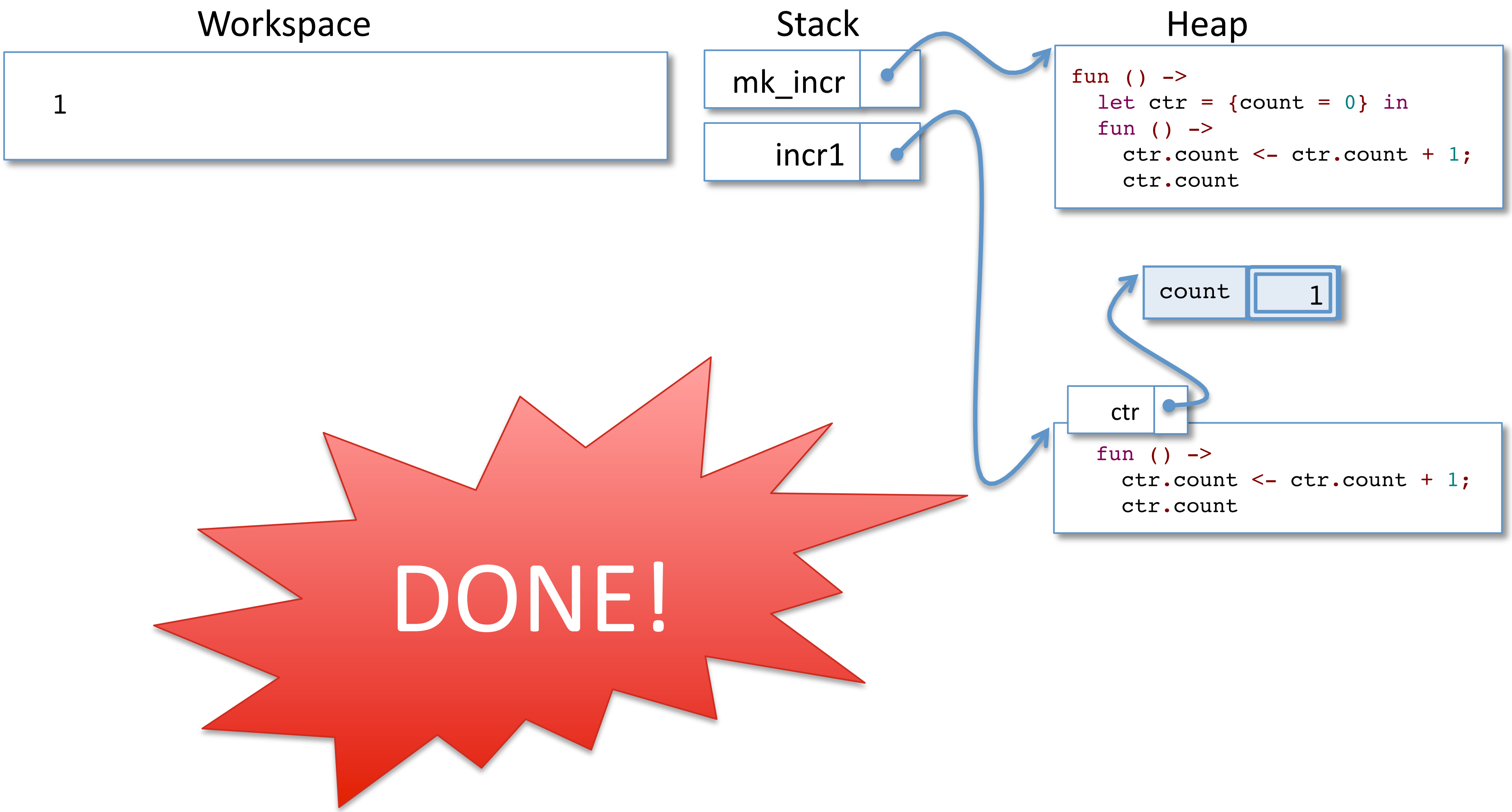
Now let's run "`incr1 ()`"



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



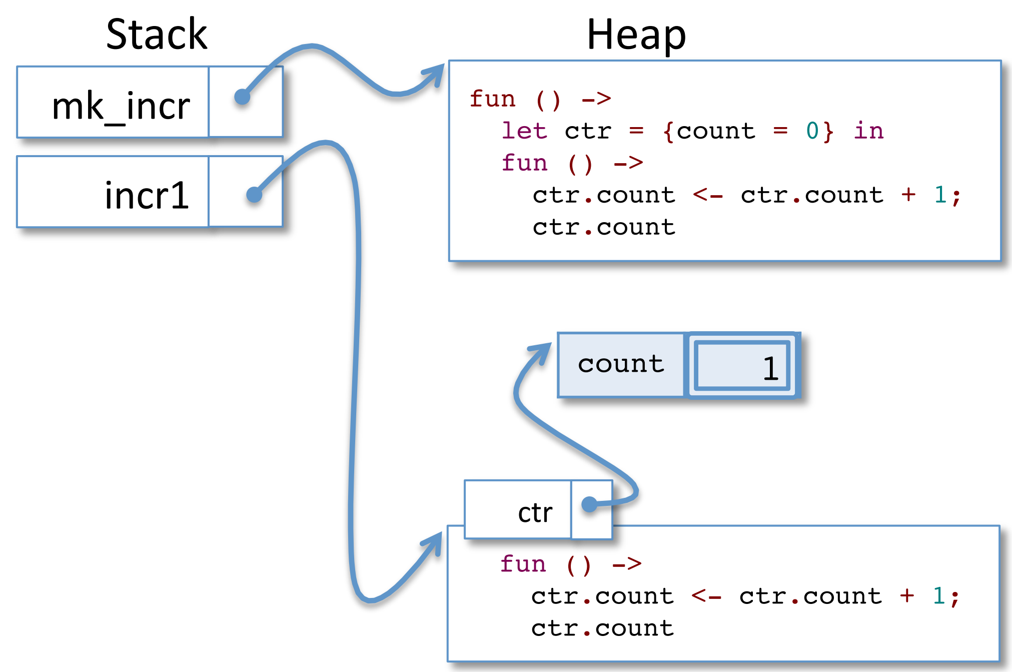
Now let's run "`incr1 ()`"



Now Let's run `mk_incr` again

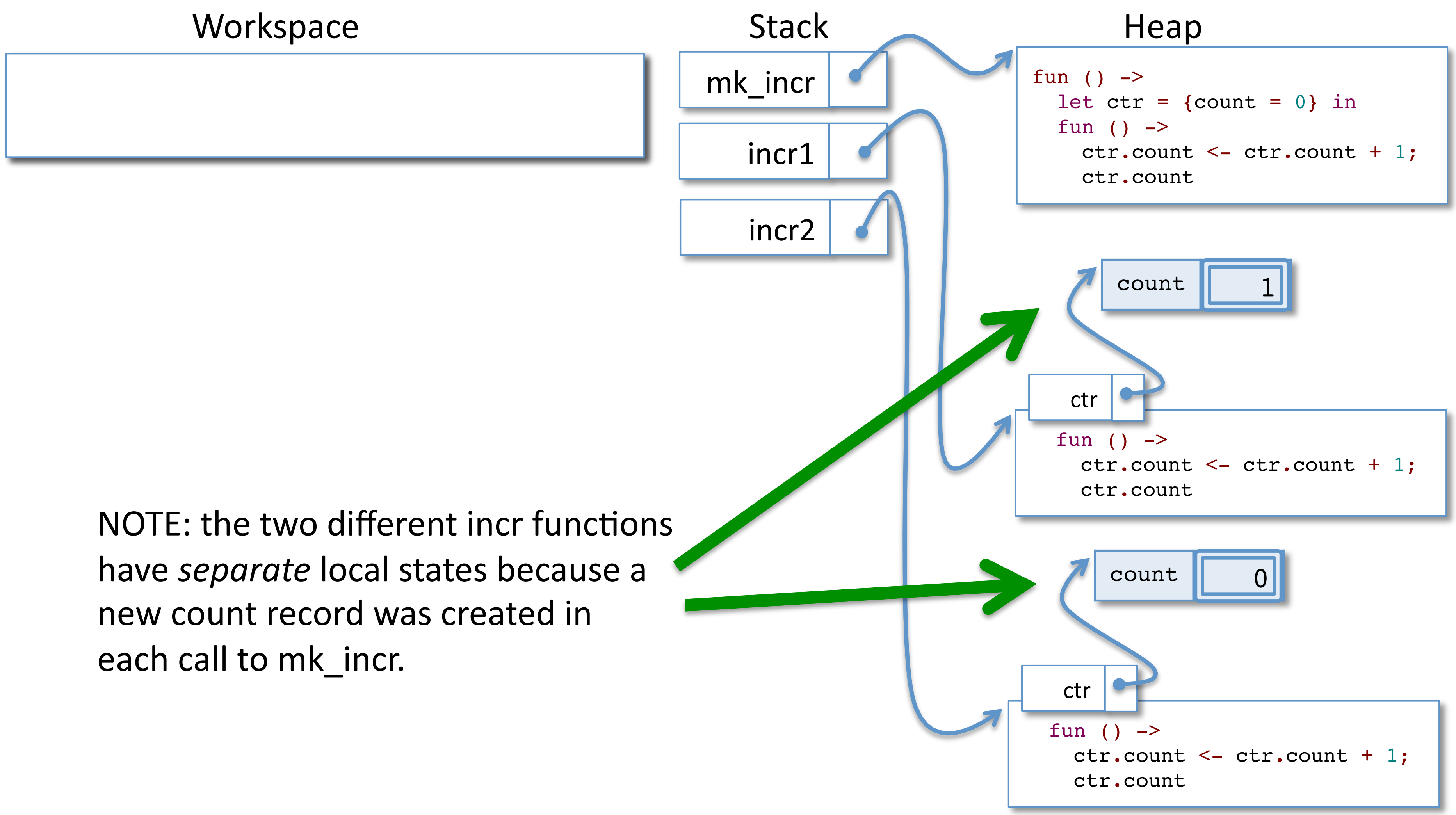
Workspace

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



…time passes…

After creating incr2



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

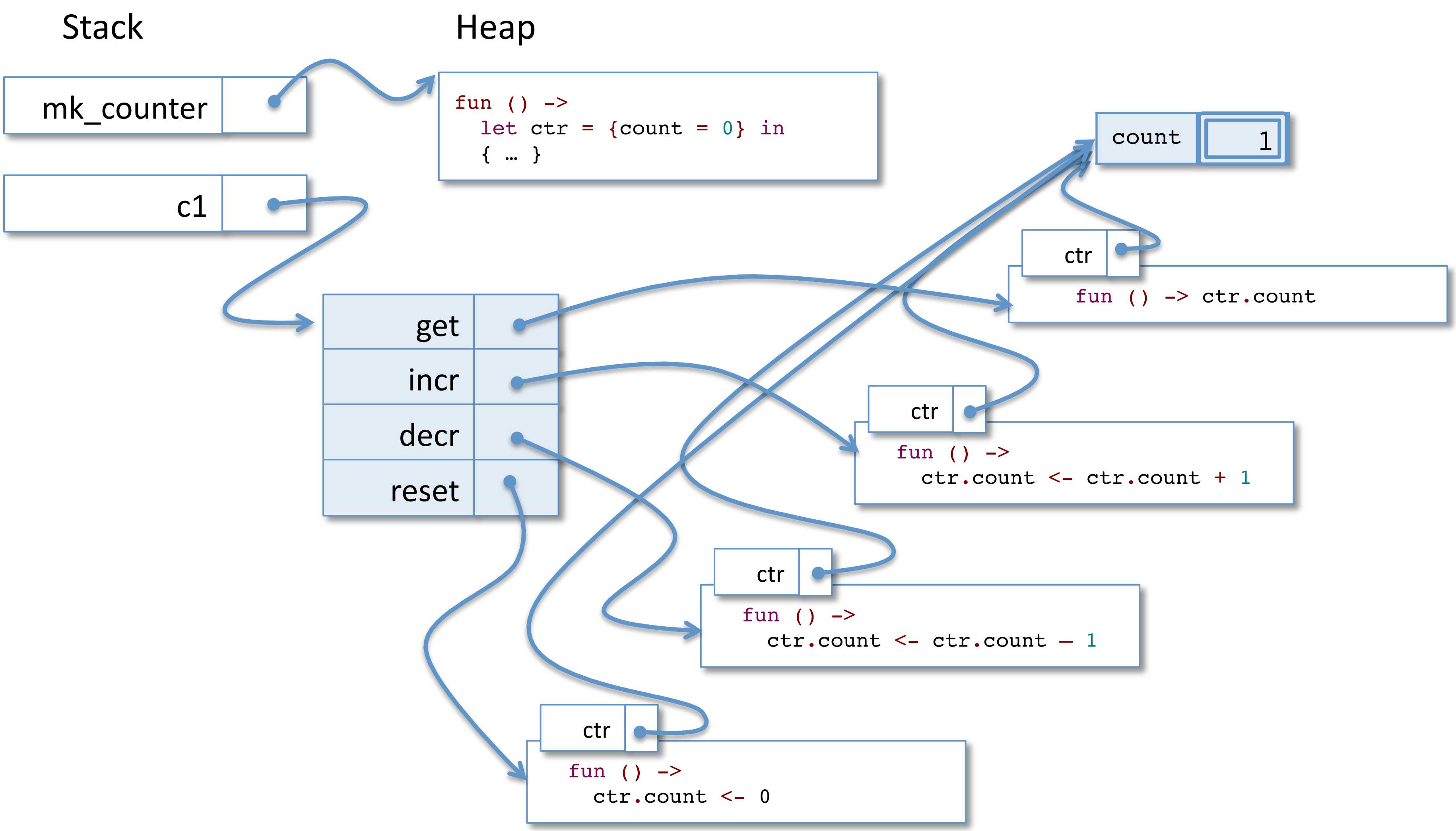
One step further

- 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 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 ()
```



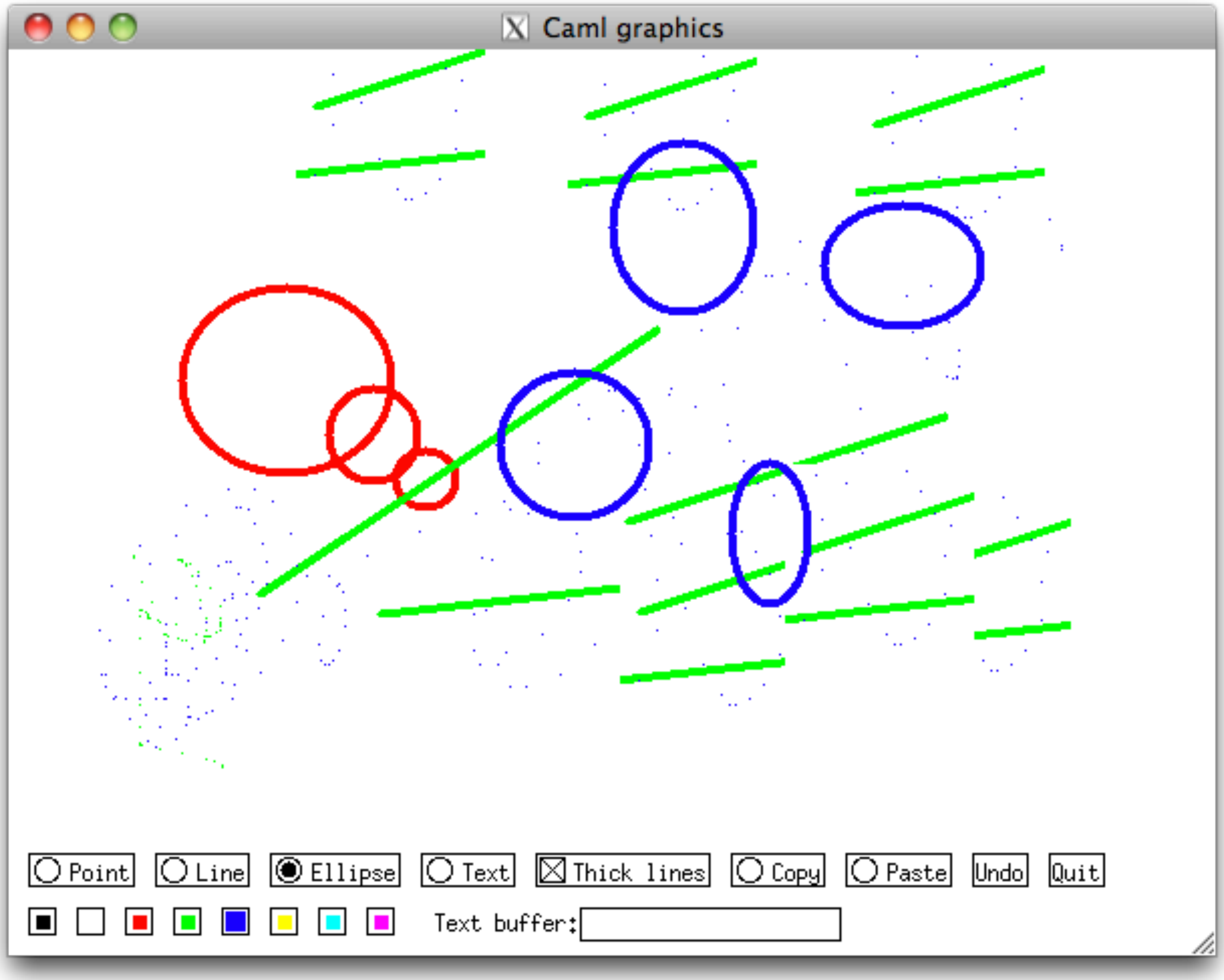
Using the Counter Objects

```
(* 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 = mk_counter ()
let c2 = mk_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

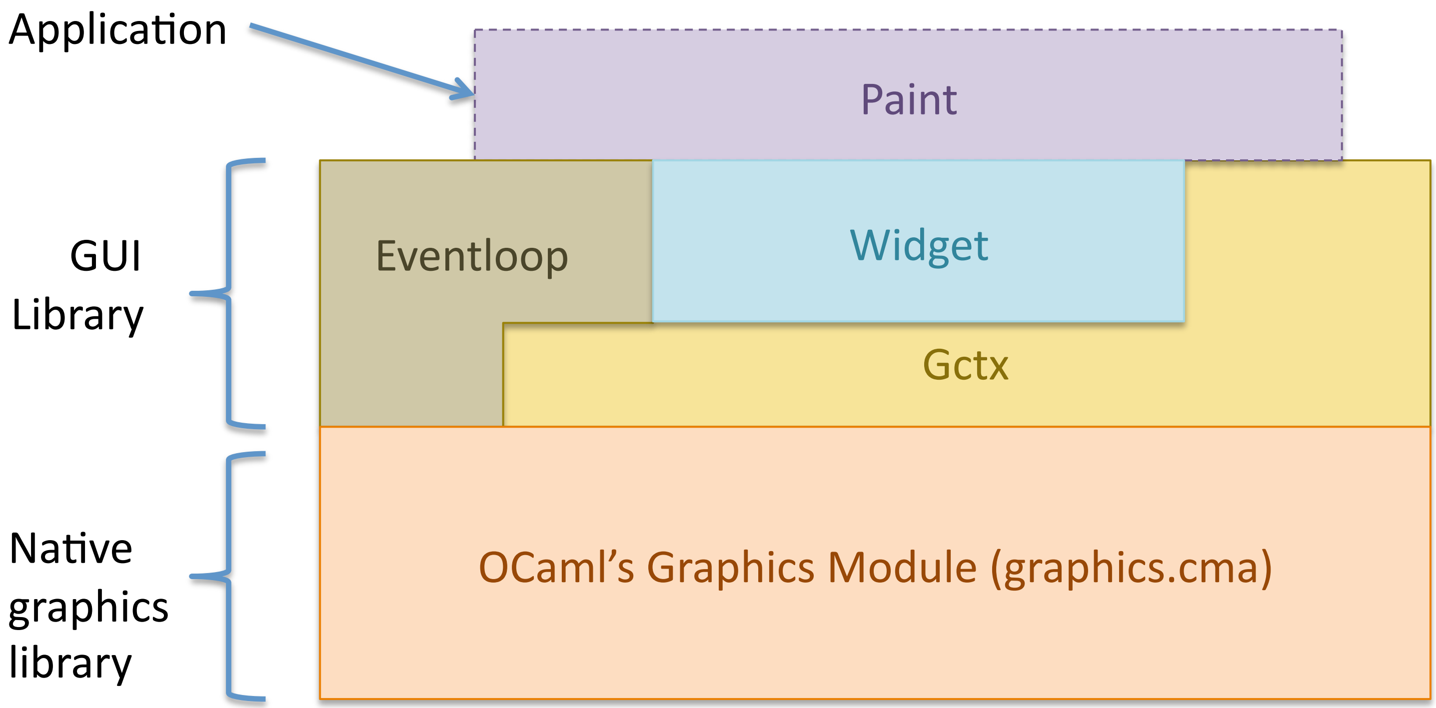
Building a GUI and GUI Applications



Step #1: Understand the Problem

- We don't want to create 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



Designing an OCaml GUI library

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.

OCaml vs. *Standard* Coordinates