Programming Languages and Techniques (CIS120)

Lecture 16

Feb 20, 2012

"Objects" and GUI Design

Announcements

- HW05 is due *Thursday* at 11:59:59pm

- Weirich OH moved from today to Wednesday (3:30-5PM)

- Midterm 1 has been graded. Grades and solutions are online.
  – You may *look* at your exam starting today
  – Laura Fox in Levine 308, 8:30-4:00 daily

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

Taking Stock: CIS120 so far…

- General Design Strategy
  - Understand the problem / Formulate the interface / Generate Tests / Write the code (refine; apply the design strategy recursively)
- Recursive, generic datatypes
  - lists, trees, etc.
  - Design pattern: recursion
- Several styles of abstraction:
  - Generic functions (map, fold)
  - Abstract datatypes via modules and interfaces (set, finite map, queues)
- Imperative data structures
  - Mutable records in the heap
  - Queues

Demo: GUI Paint Application

The CIS120 Trajectory

- Coming up: putting it all together
  - Build a GUI library and client application *from scratch* in OCaml
- Several purposes:
  - Show you that you have enough knowledge to do some pretty serious programming (just more of the same).
  - Work through a more interesting design process
  - Illustrate the *event-driven* programming model
  - Practice with first-class functions and *hidden state*
  - Motivate the features of object-oriented languages like Java
  - Give you a feel for how real GUI libraries (like Java's Swing) work
- Afterwards: transition to Java

Hidden State

What is an Object?

- Object = Fields + Methods
  - Fields = Mutable record
  - Methods = Functions that mutate the fields

- Objects *encapsulate* state when the methods are the ***only*** way to mutate the fields.

- Objects are first-class. Can have several *instances* of the state, 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.

Adding a ‘decr’ function

- This function will decrement the *same* counter state that incr works with:
  - They share a reference to ctr, so mutations in one affect value read in the other.

```
(* decr shares the same state as incr *)
let decr () : int =
  ctr.count <- ctr.count - 1;
  ctr.count
```

- Drawback: there can be only one global counter state.
  - How can we do better?

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
```

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 =
(___)
```

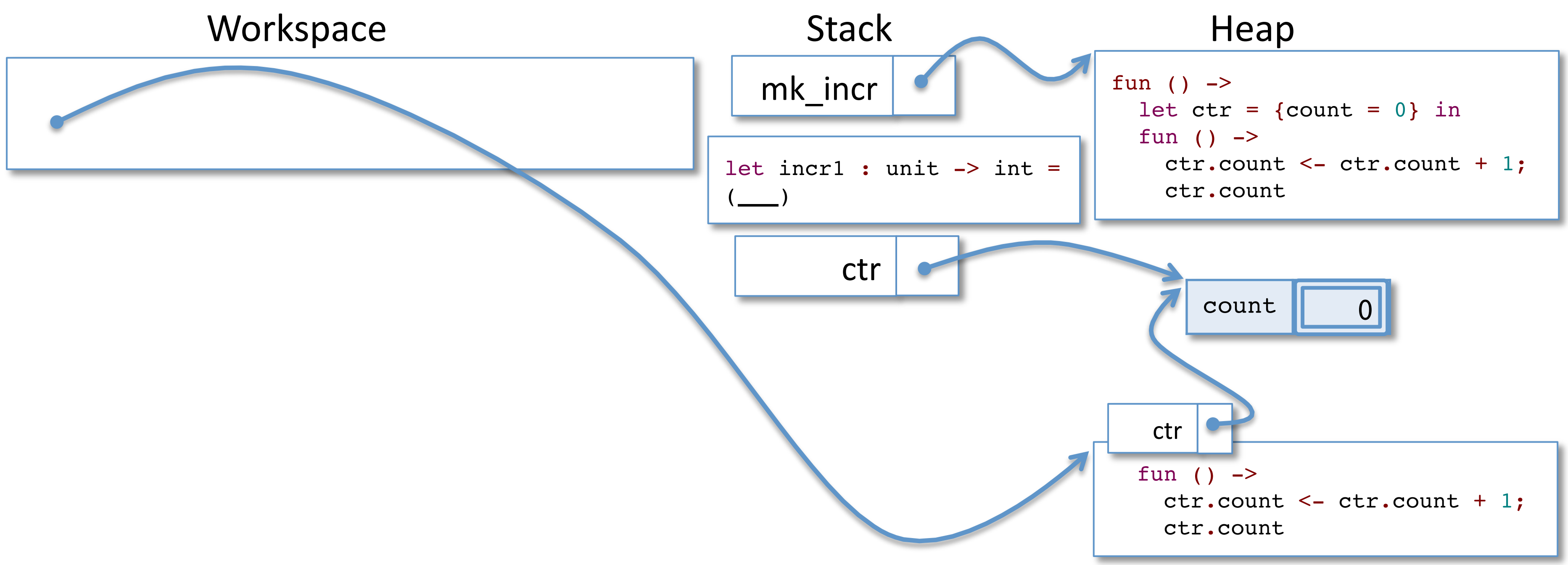
ctr

Heap

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

count 0

Local Functions

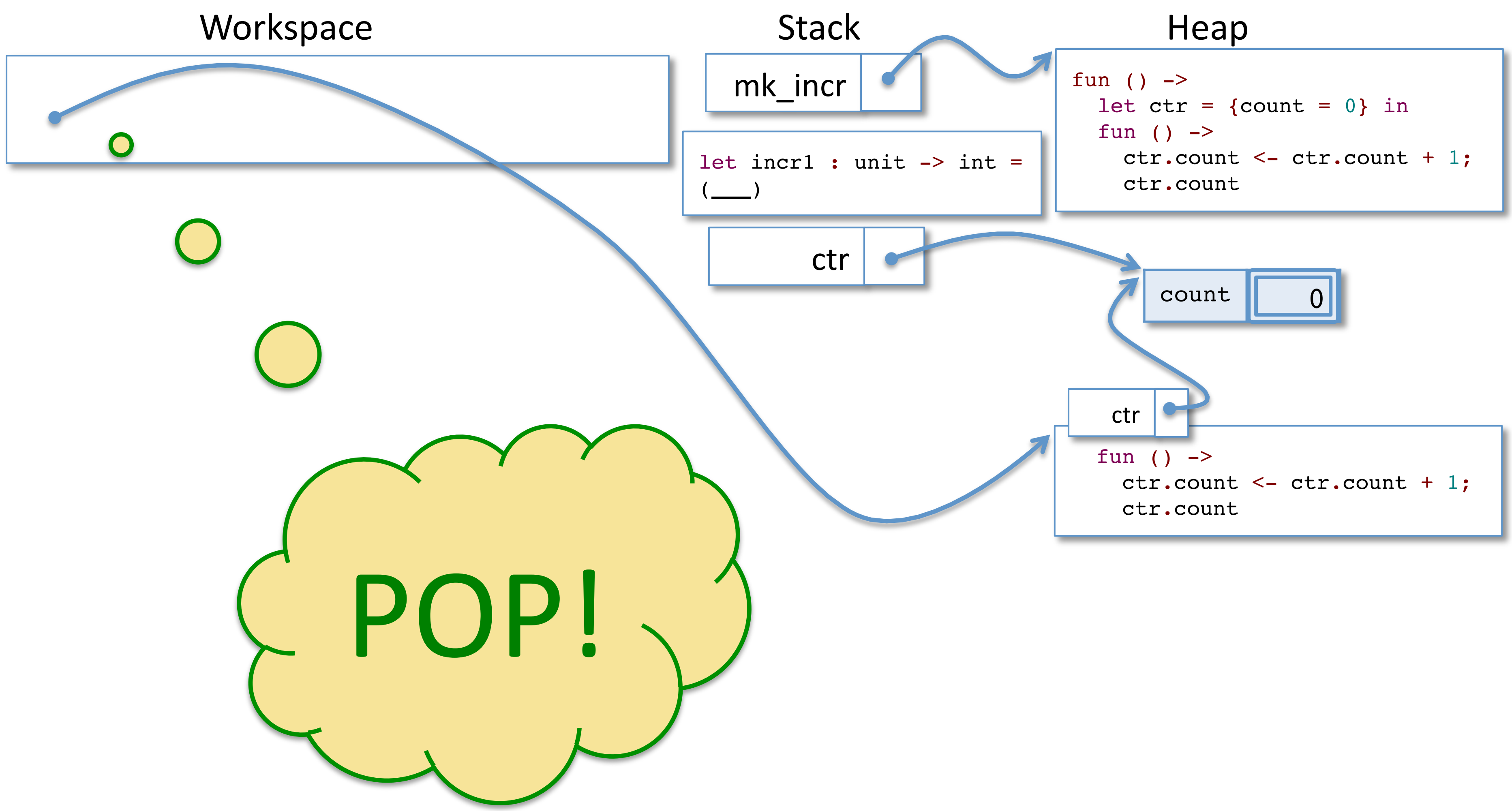


NOTE: We need one refinement to handle local functions. Why?

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

Workspace

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

Stack

mk_incr	

Heap

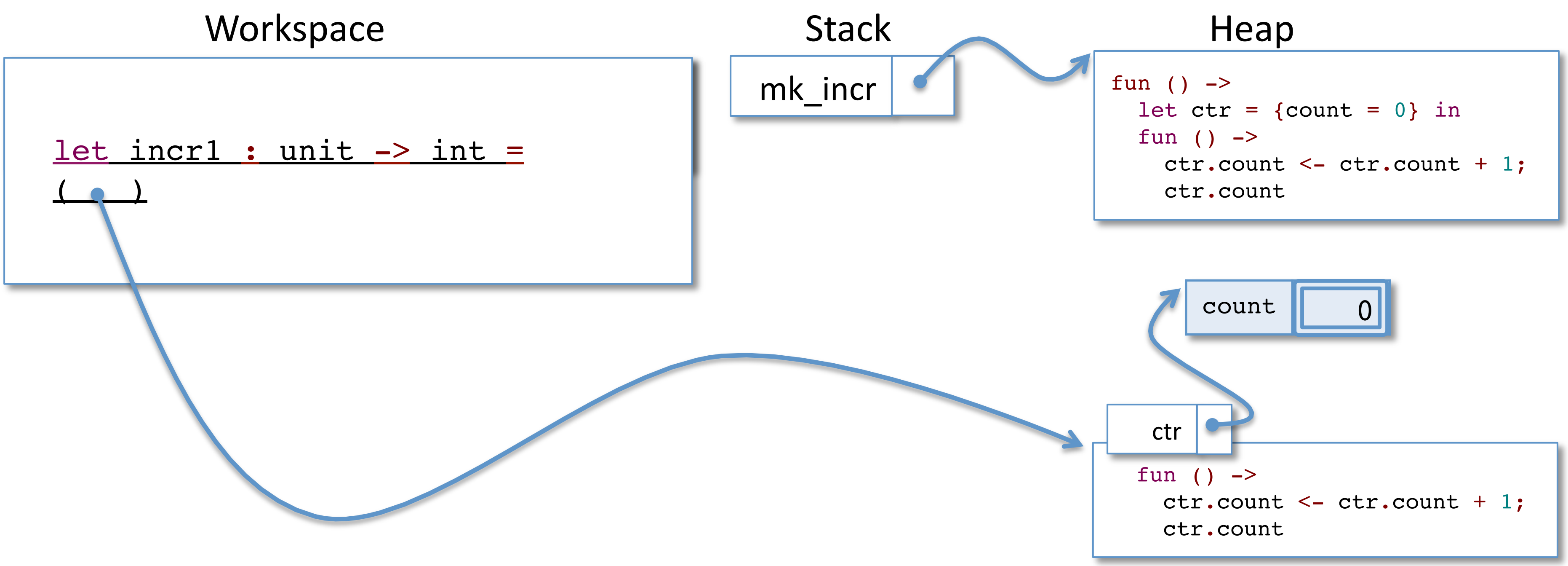
```
fun () ->
  let ctr = {count = 0} in
  fun () ->
    ctr.count <- ctr.count + 1;
    ctr.count
```

count	0

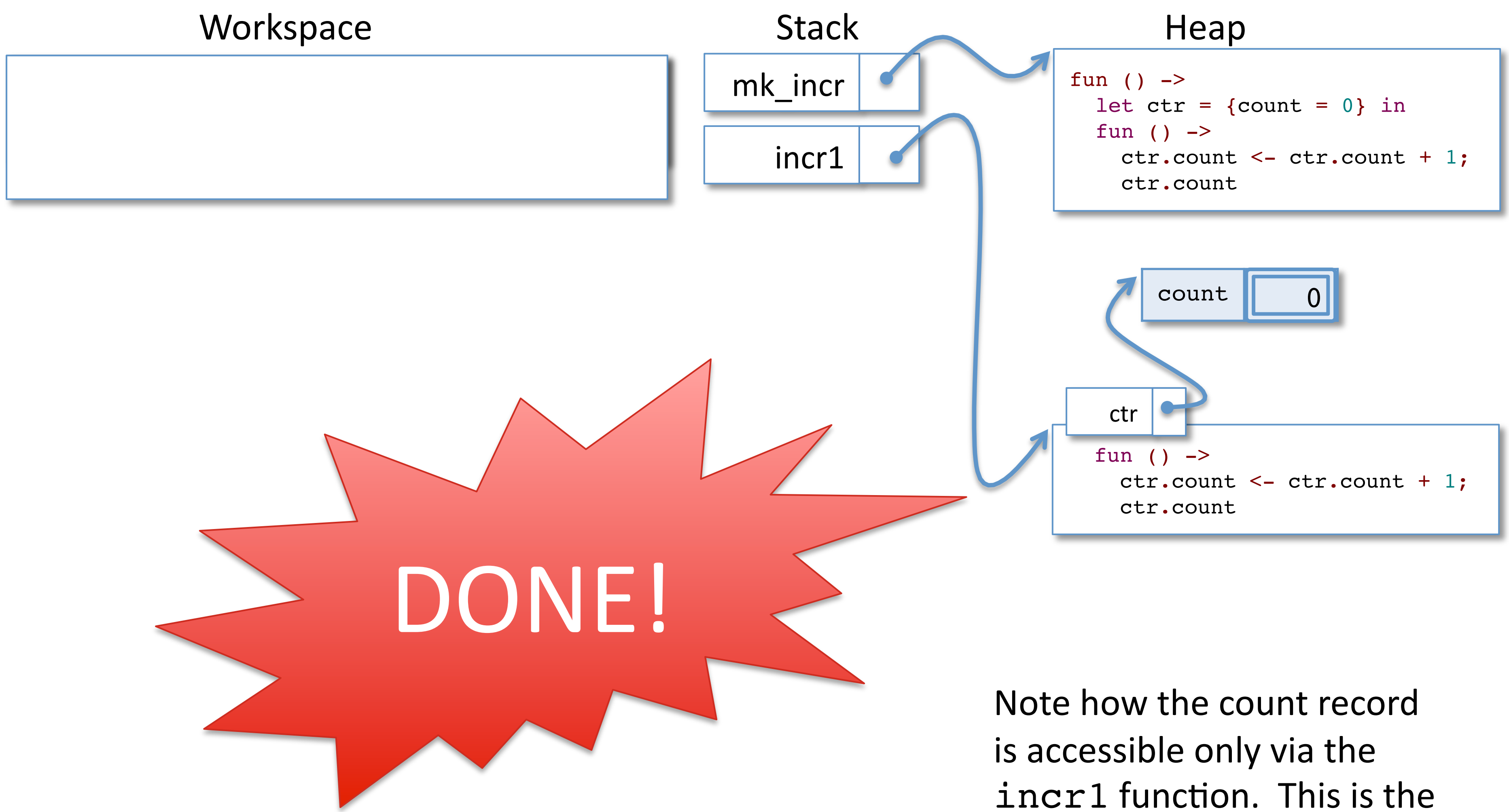
ctr	

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

Local Functions

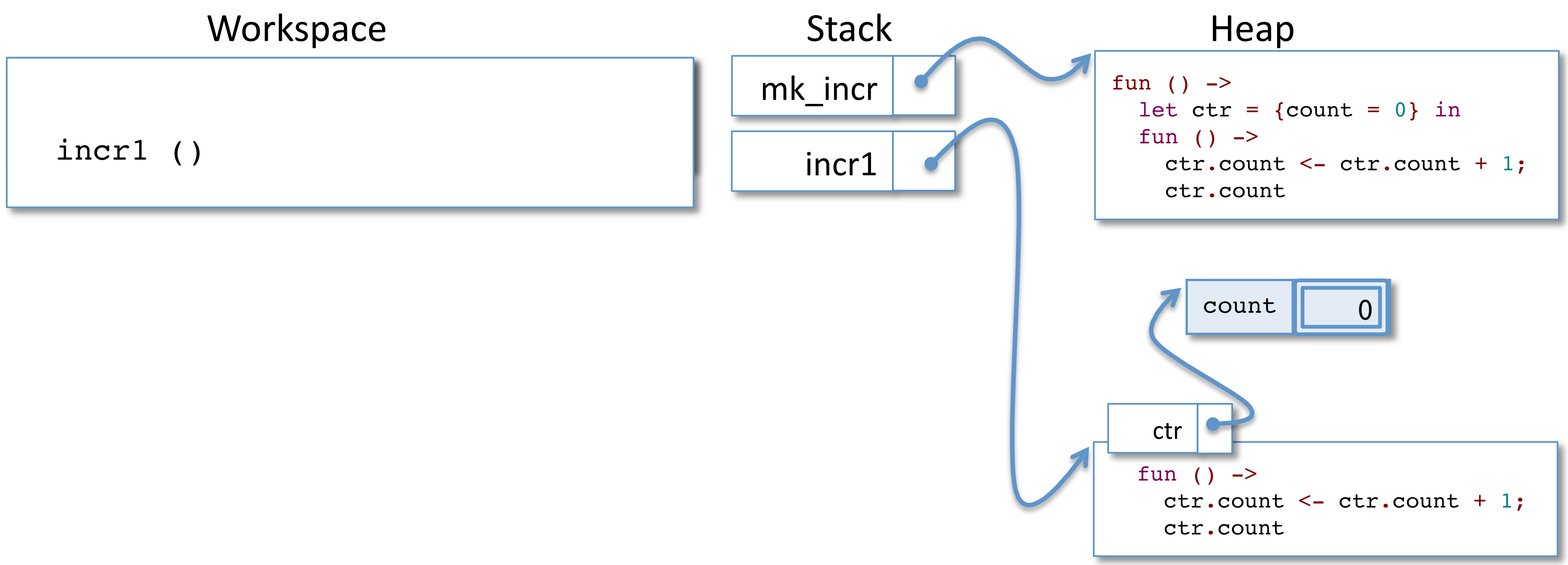


Local Functions

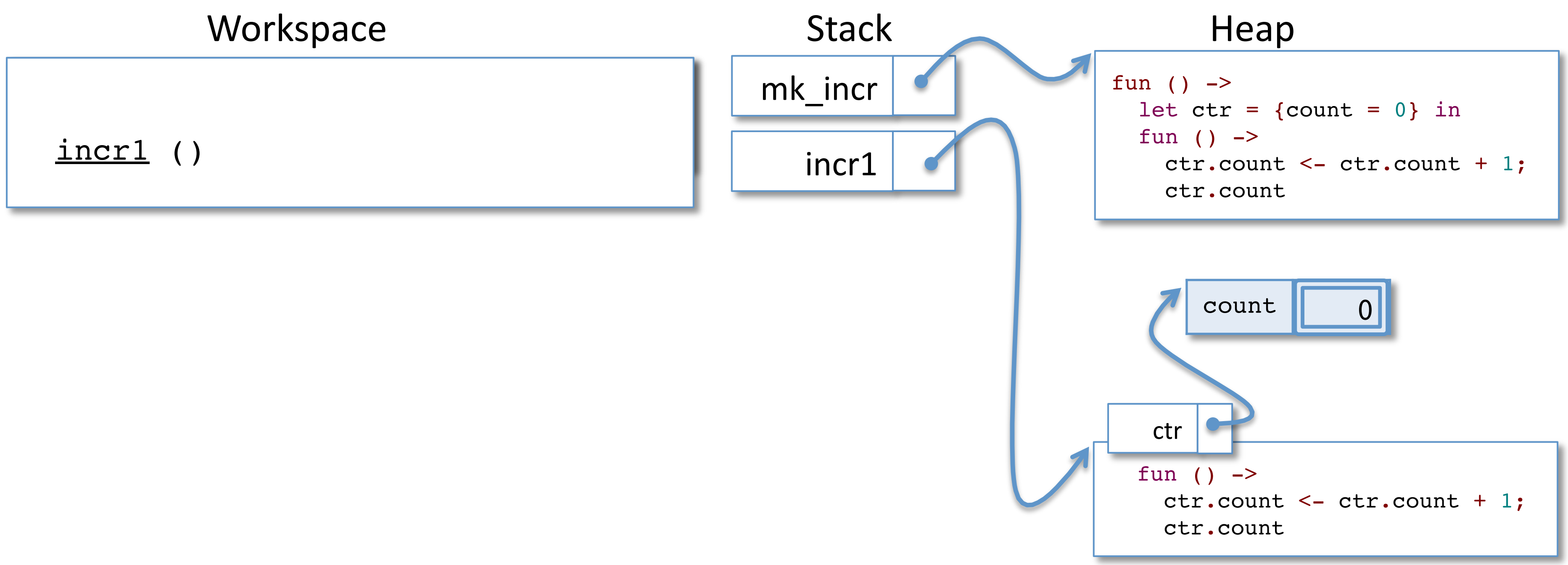


Note how the count record is accessible only via the `incr1` function. This is the sense in which the state is "local" to `incr1`.

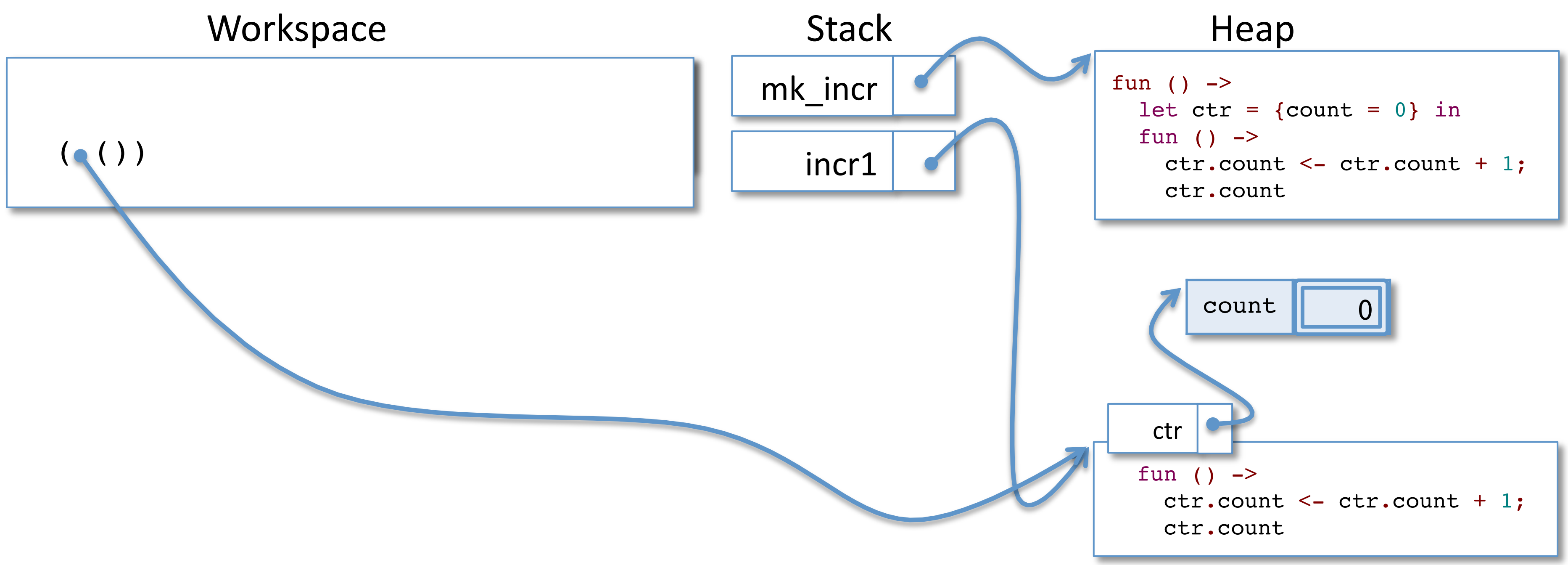
Now let's run "`incr1 ()`"



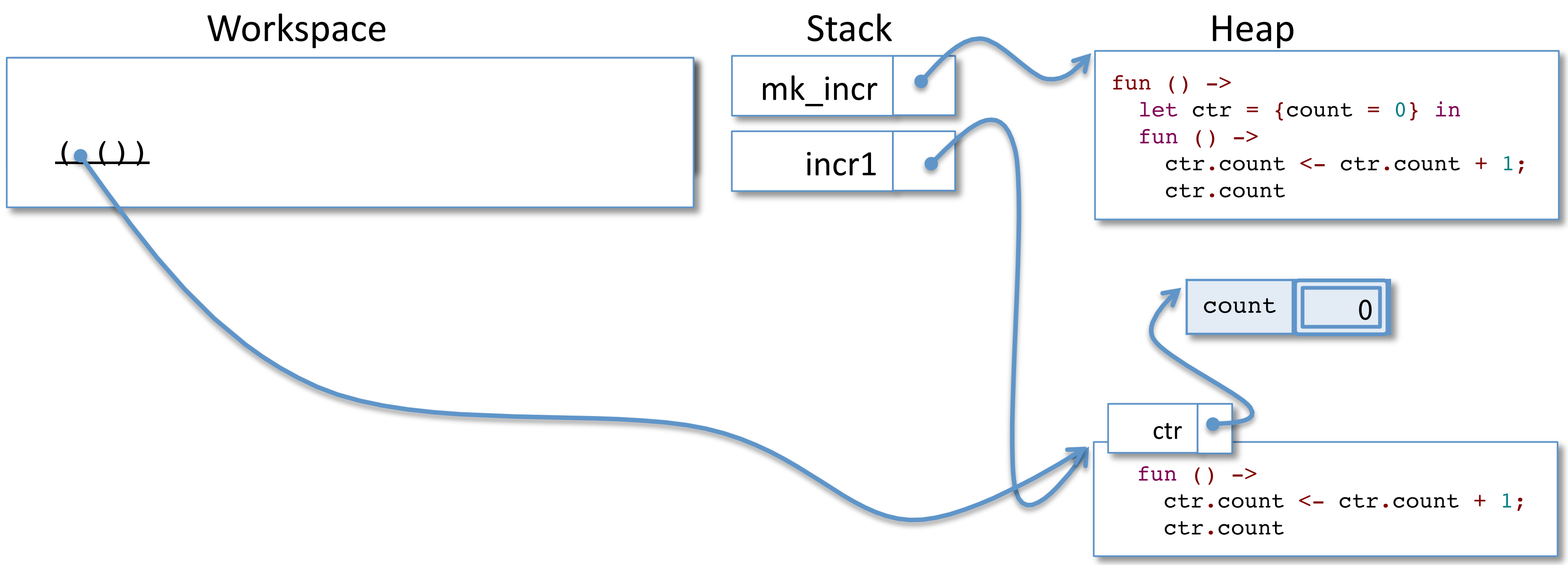
Now let's run "`incr1 ()`"



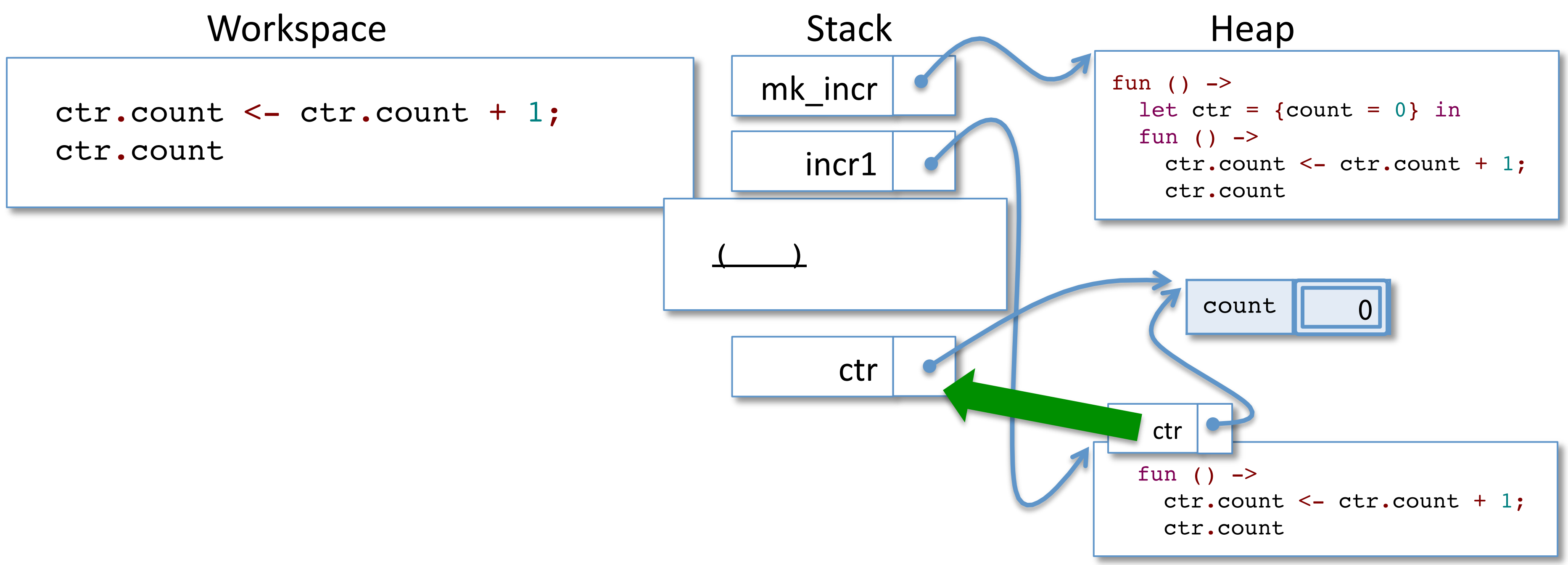
Now let's run "`incr1 ()`"



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

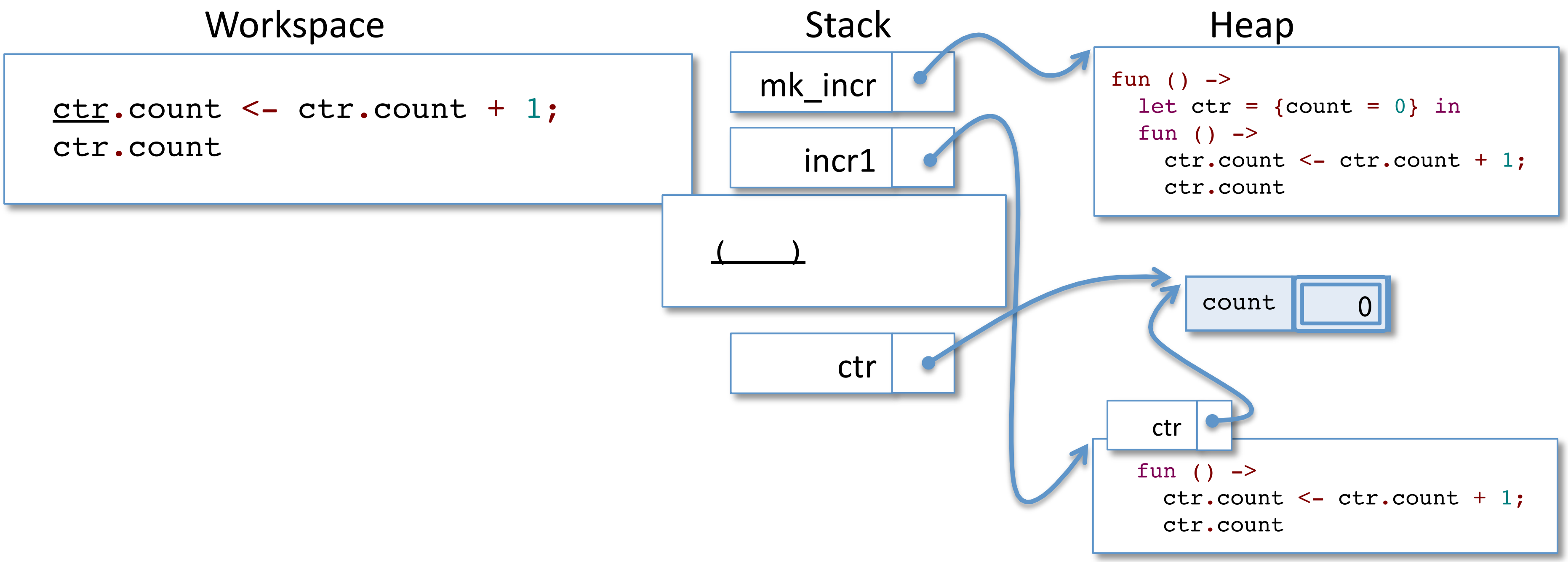


Now let's run "incr1 ()"

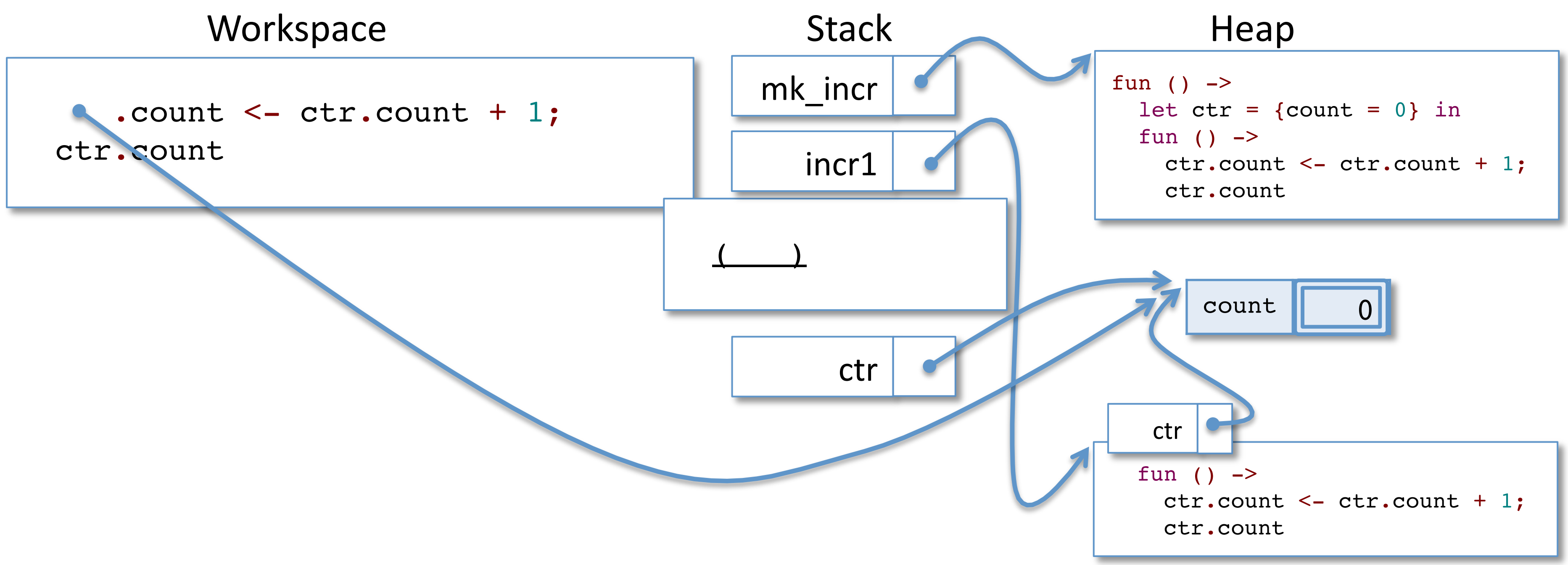


NOTE: Since the function had some saved stack bindings, we add them to the stack at the same time that we put the code in the workspace.

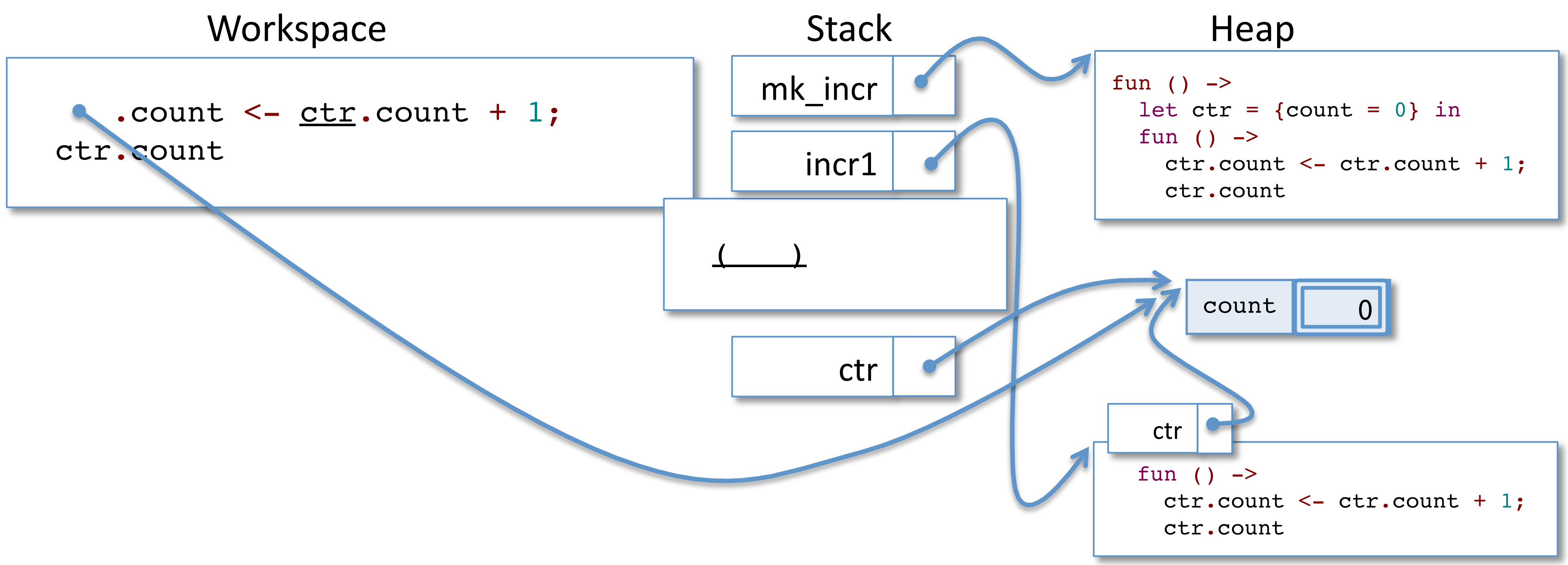
Now let's run "`incr1 ()`"



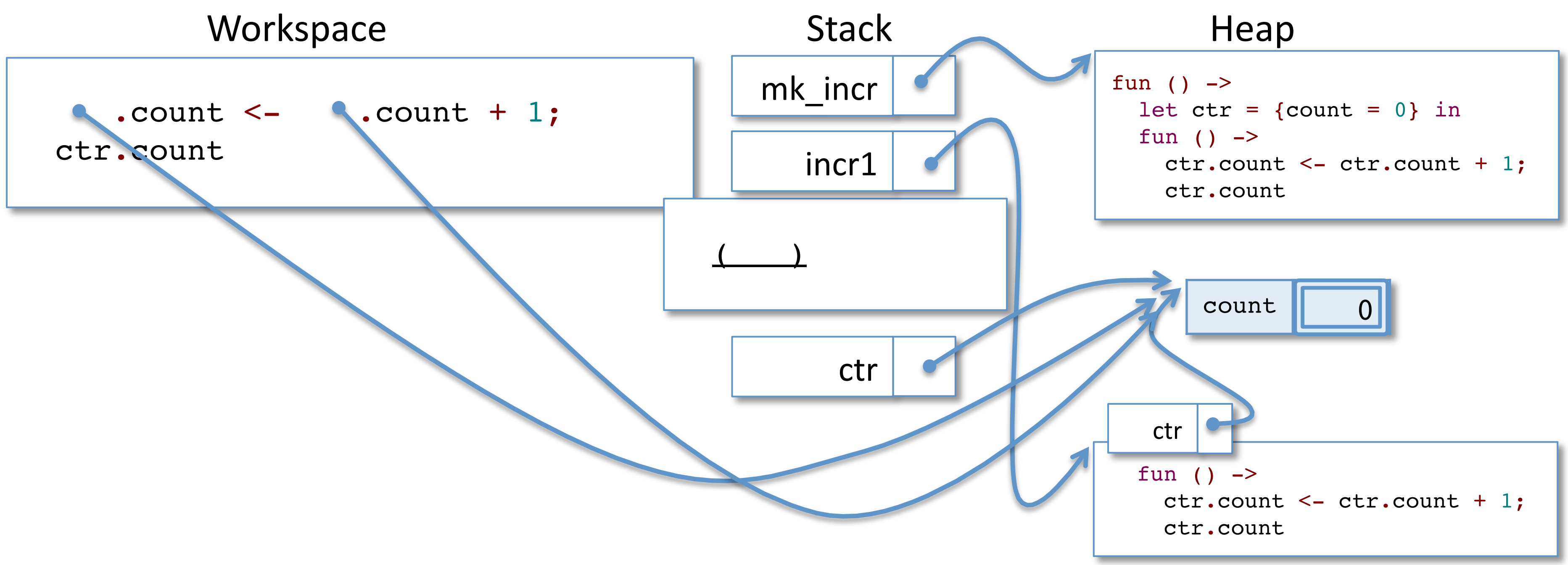
Now let's run "incr1 ()"



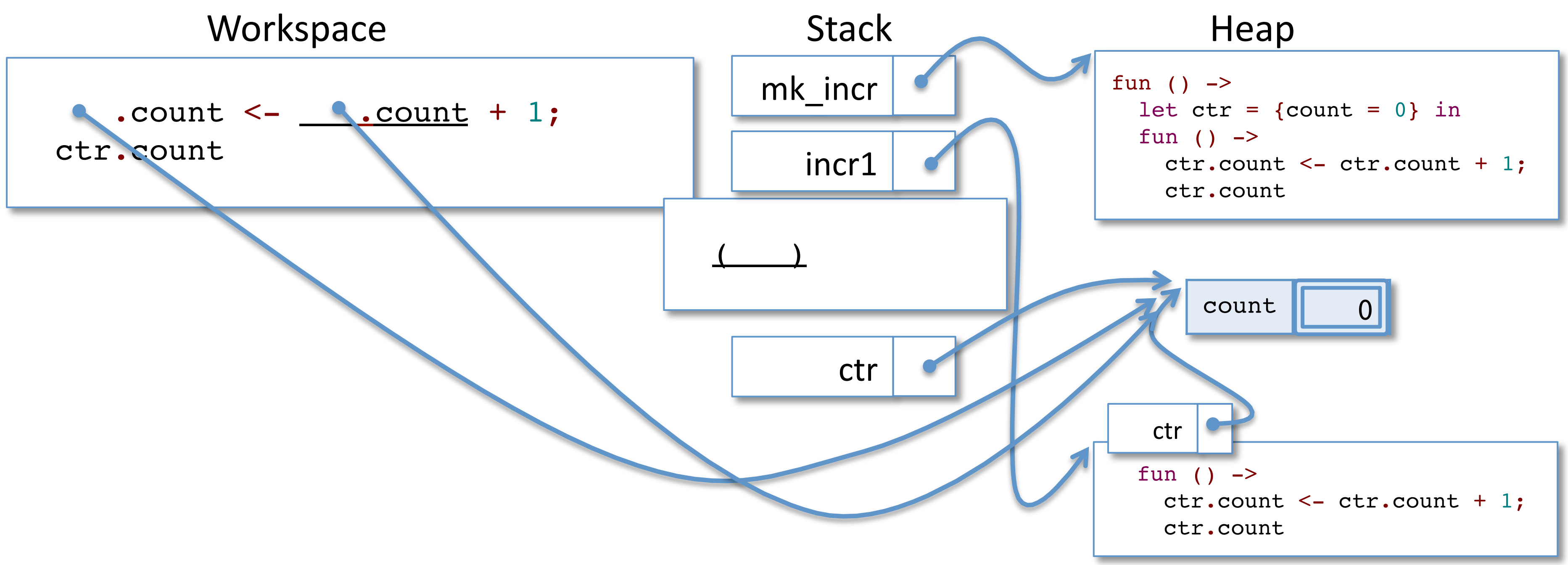
Now let's run "`incr1 ()`"



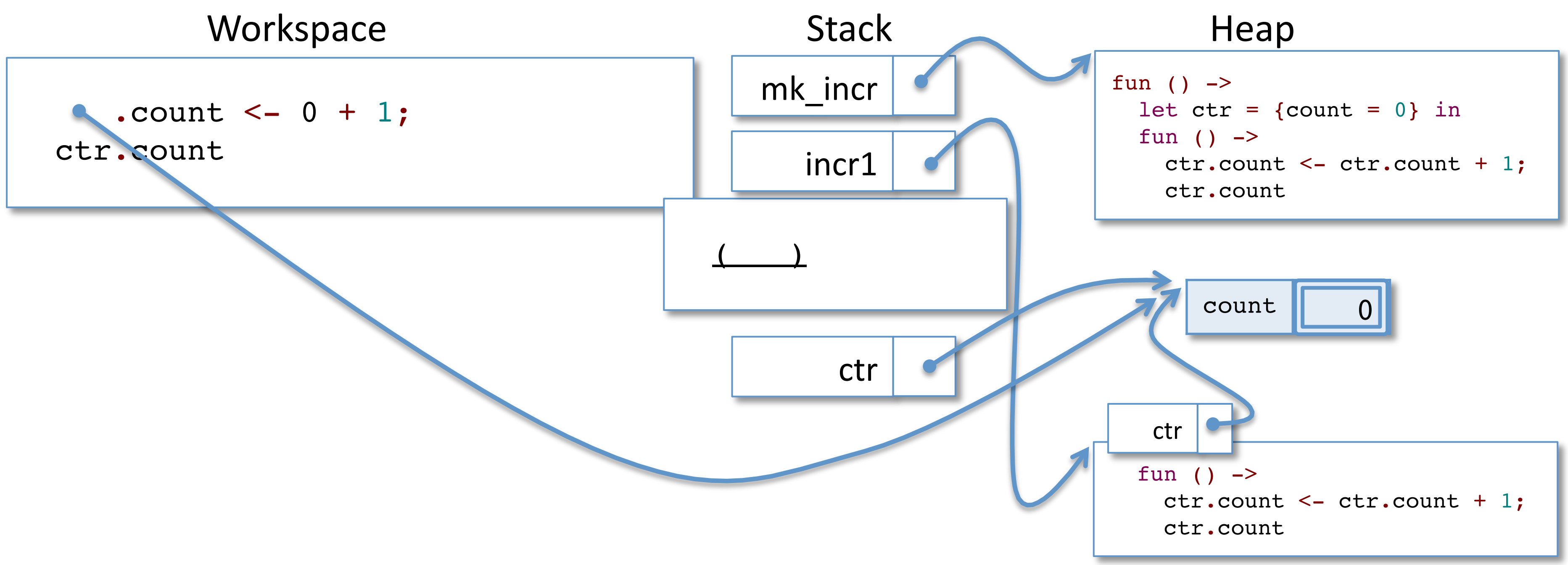
Now let's run "`incr1 ()`"



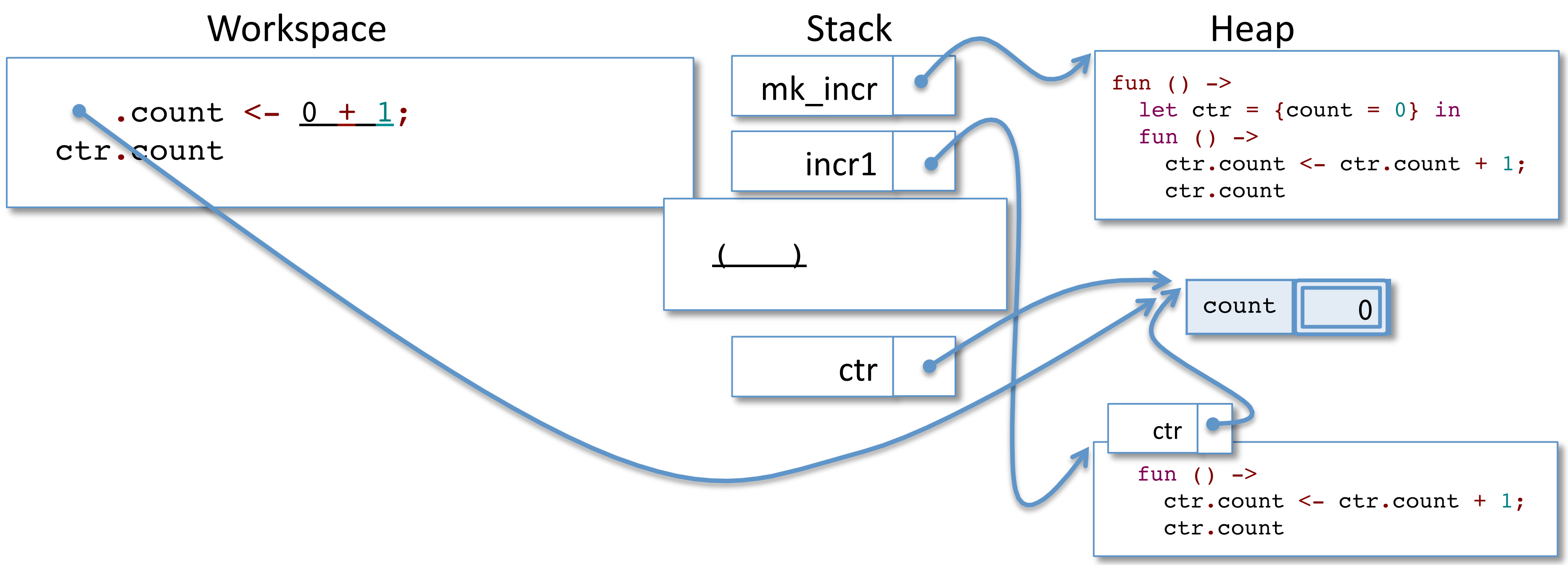
Now let's run "`incr1 ()`"



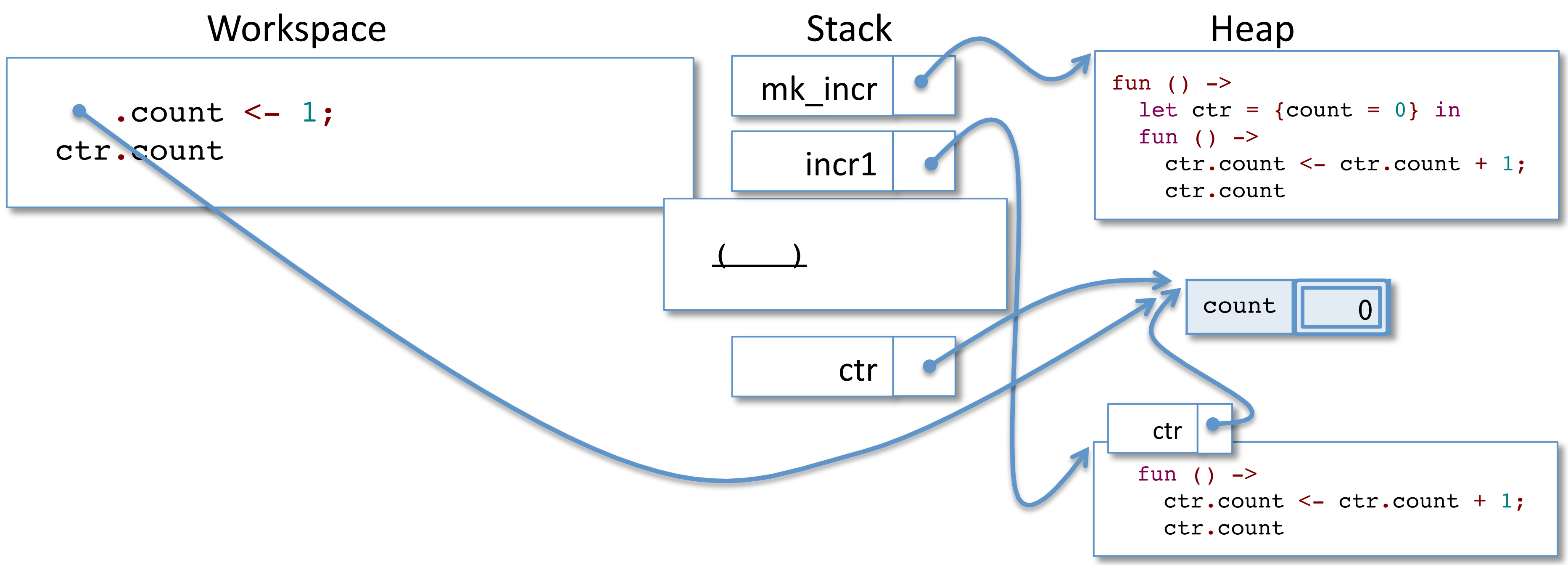
Now let's run "`incr1 ()`"



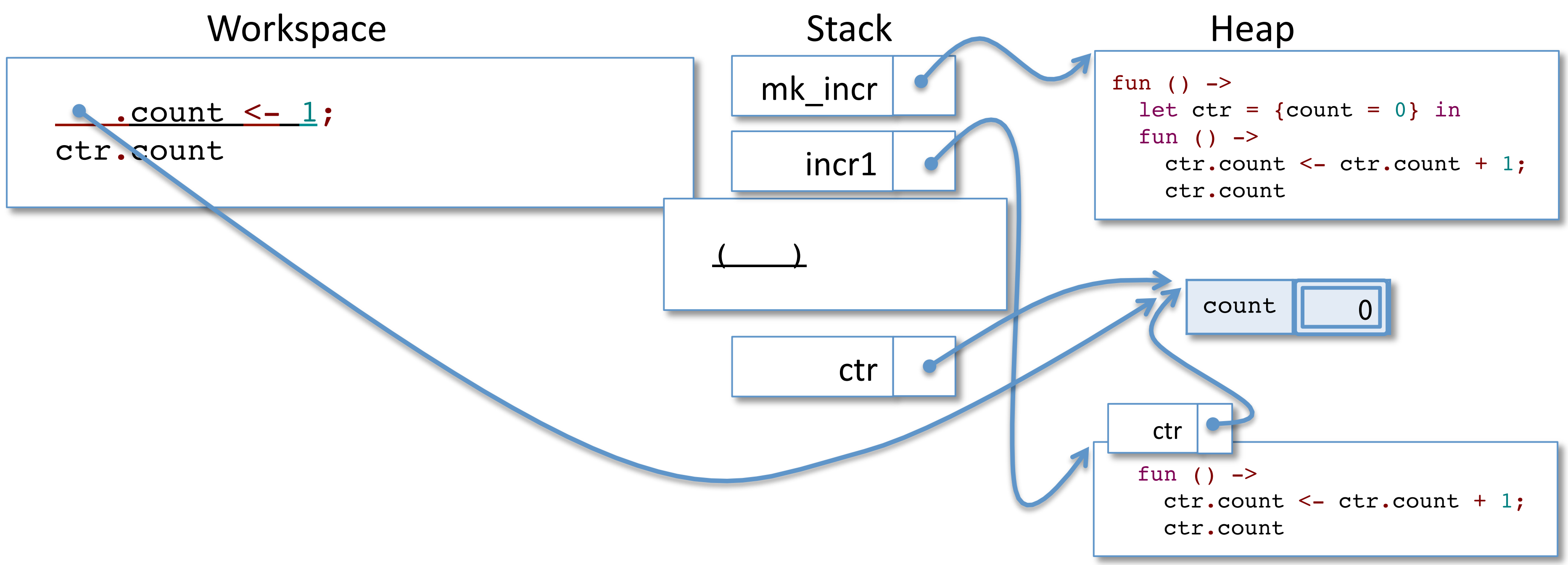
Now let's run "`incr1 ()`"



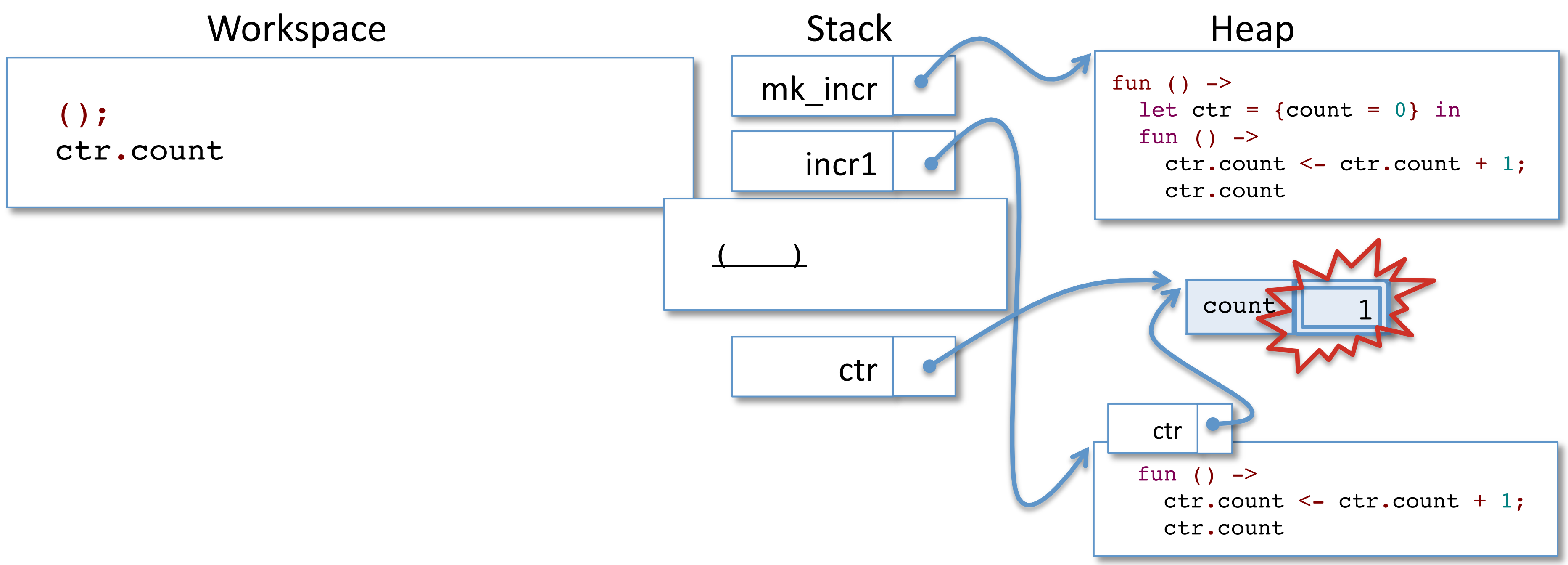
Now let’s run “incr1 ()”



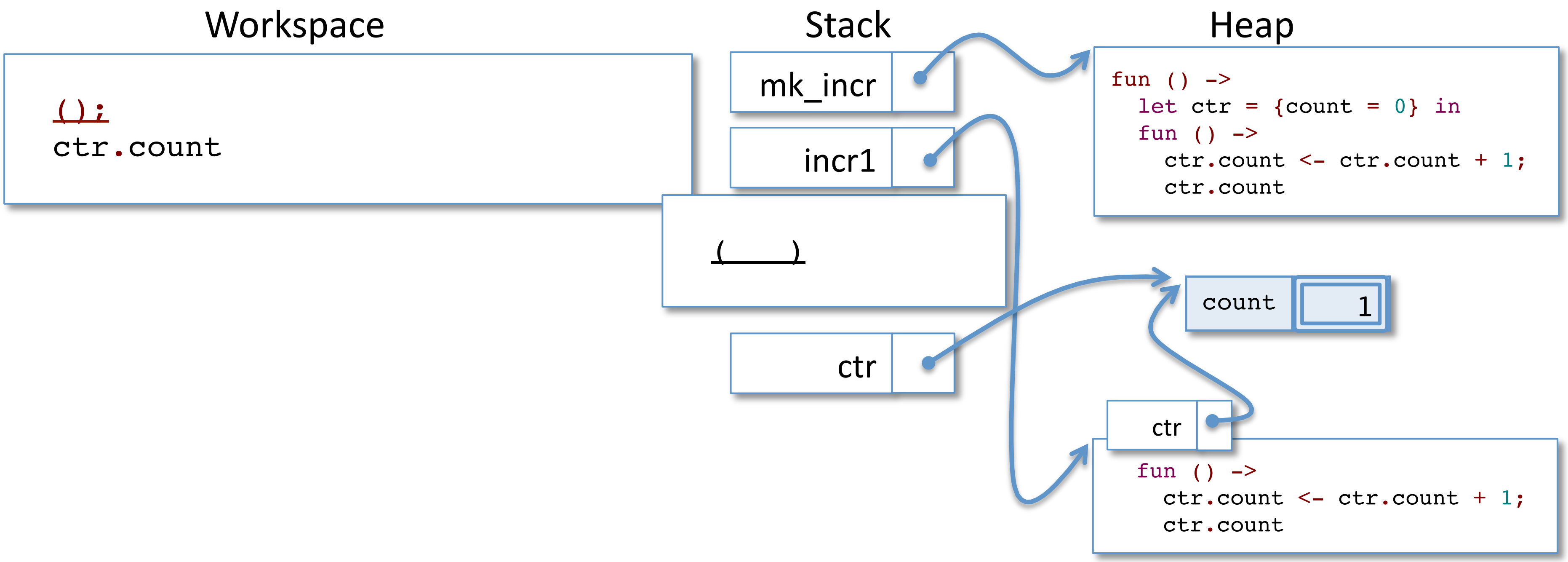
Now let's run "`incr1 ()`"



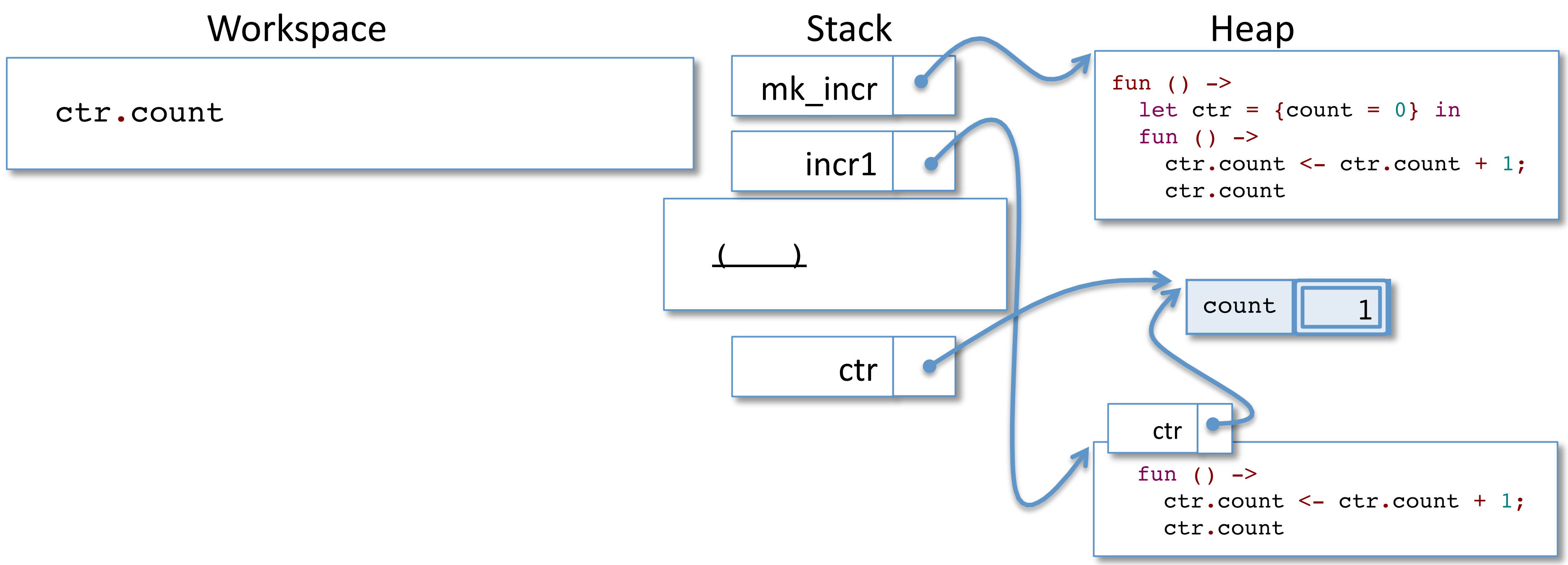
Now let's run "`incr1 ()`"



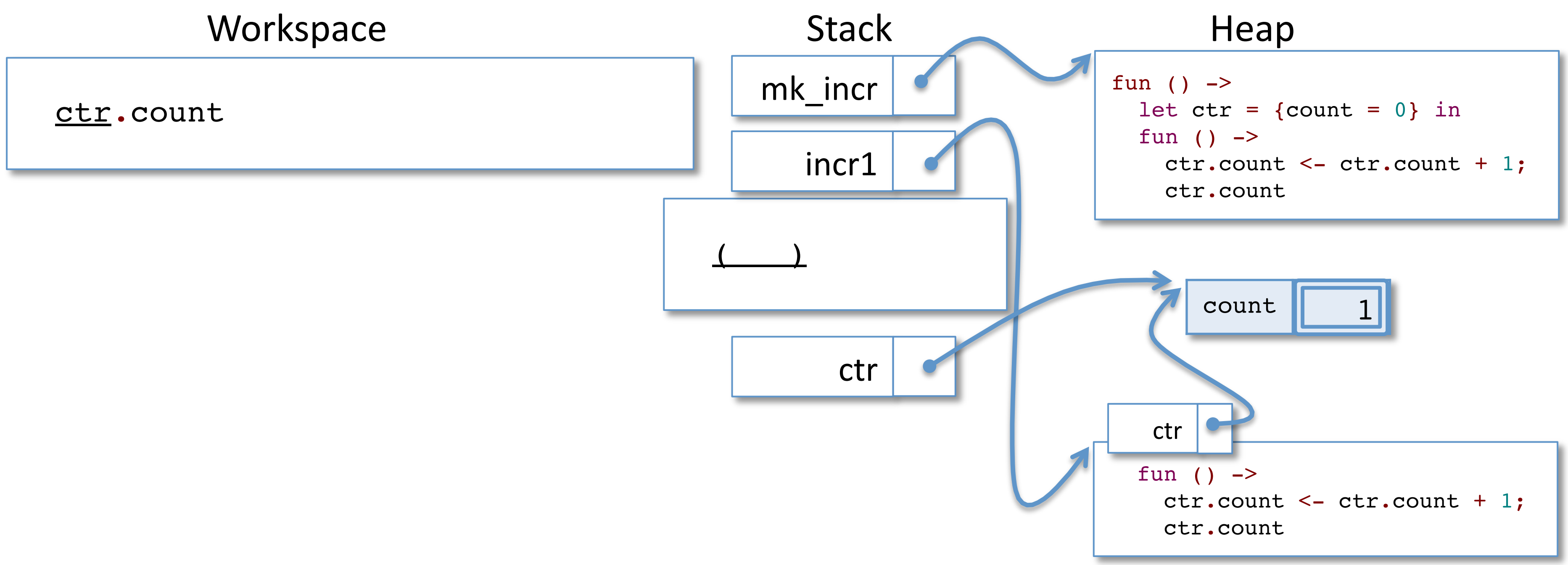
Now let's run "`incr1 ()`"



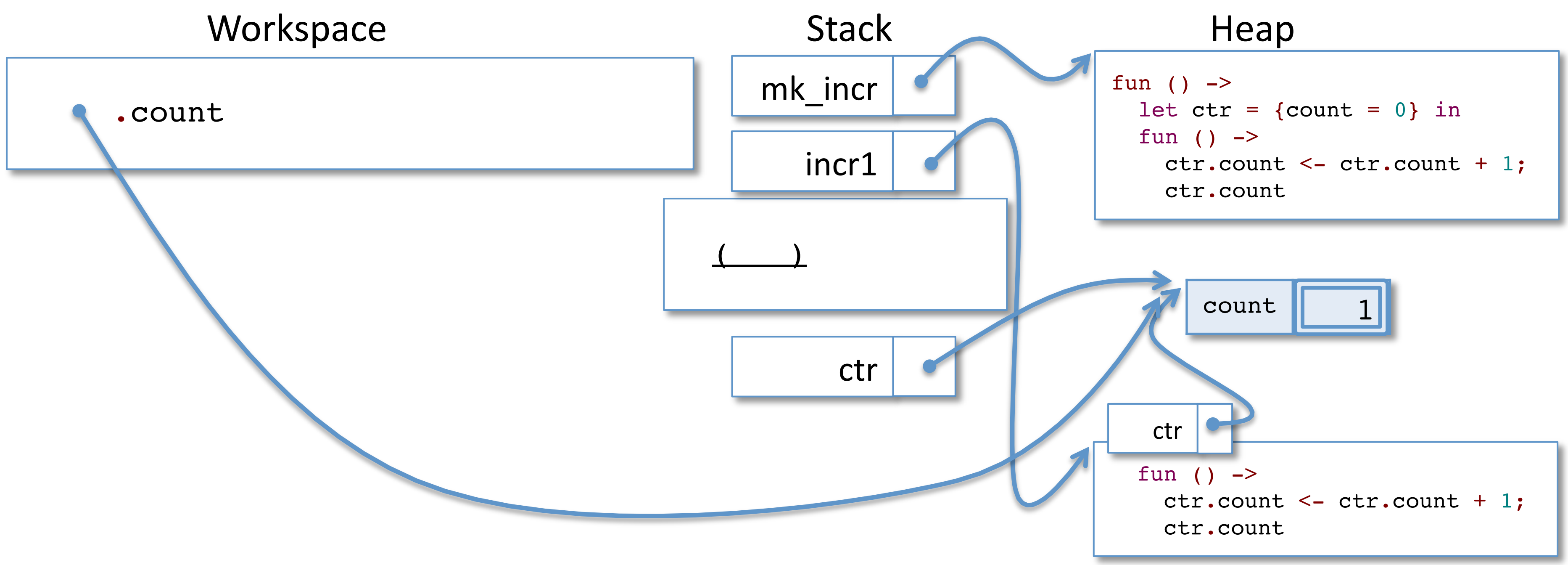
Now let's run "`incr1 ()`"



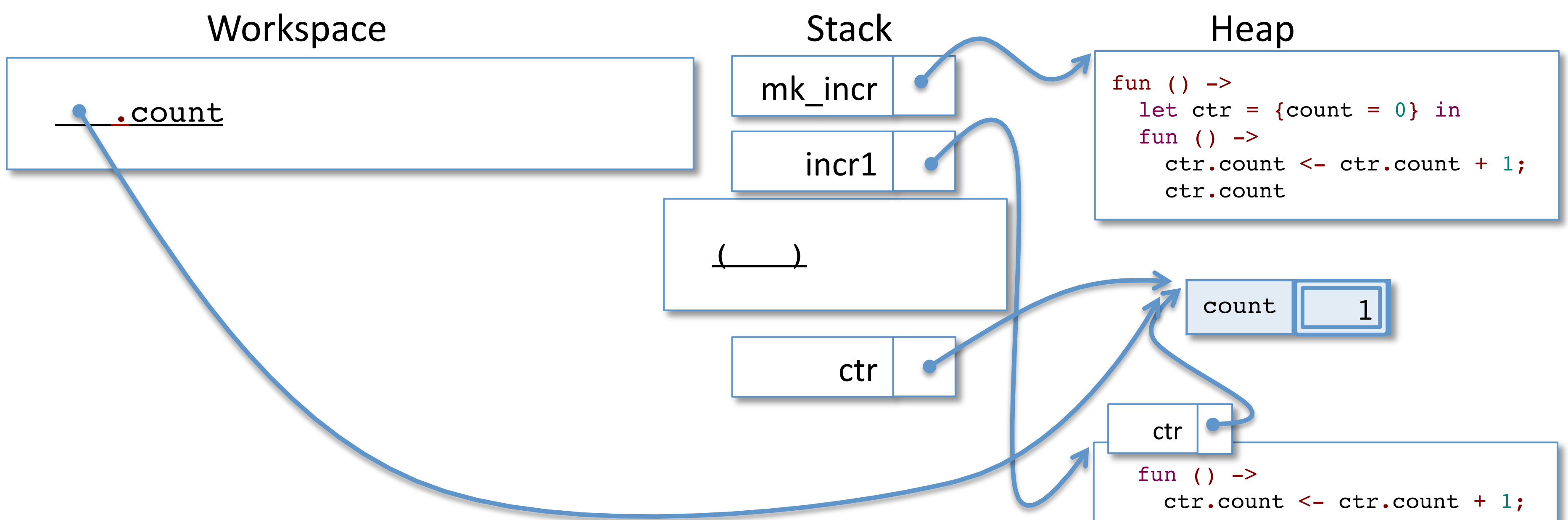
Now let's run "incr1 ()"



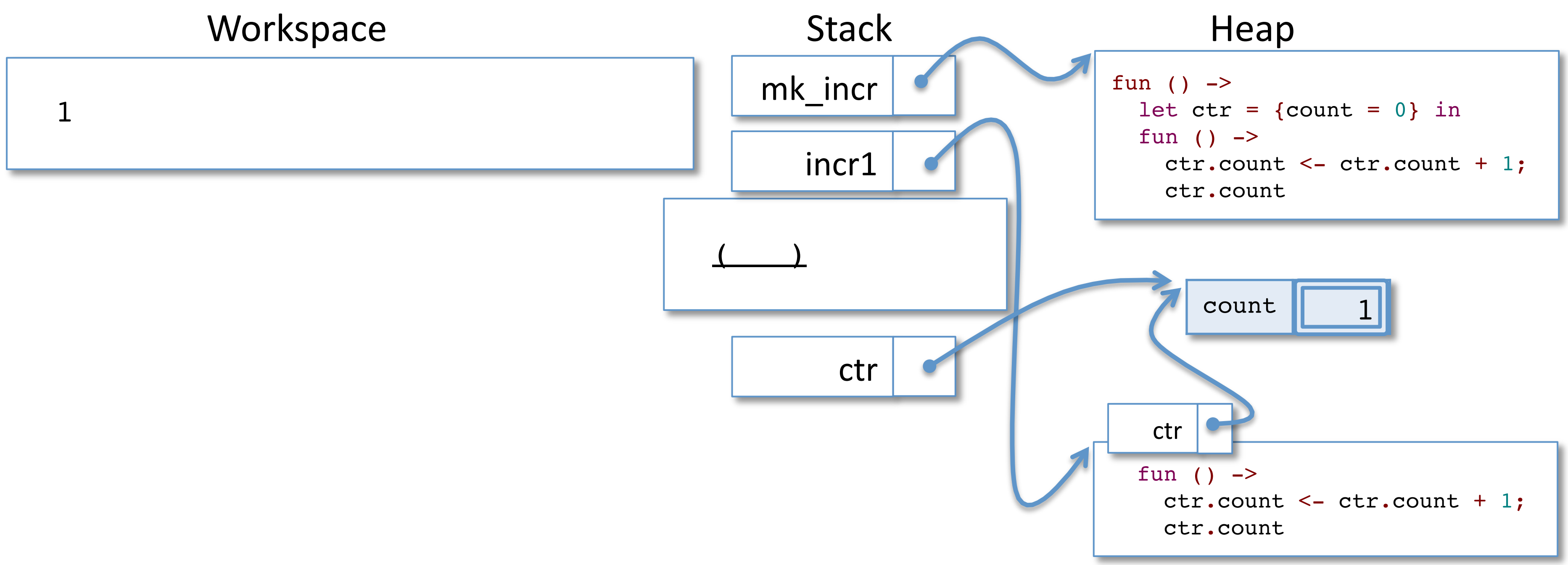
Now let's run "`incr1 ()`"



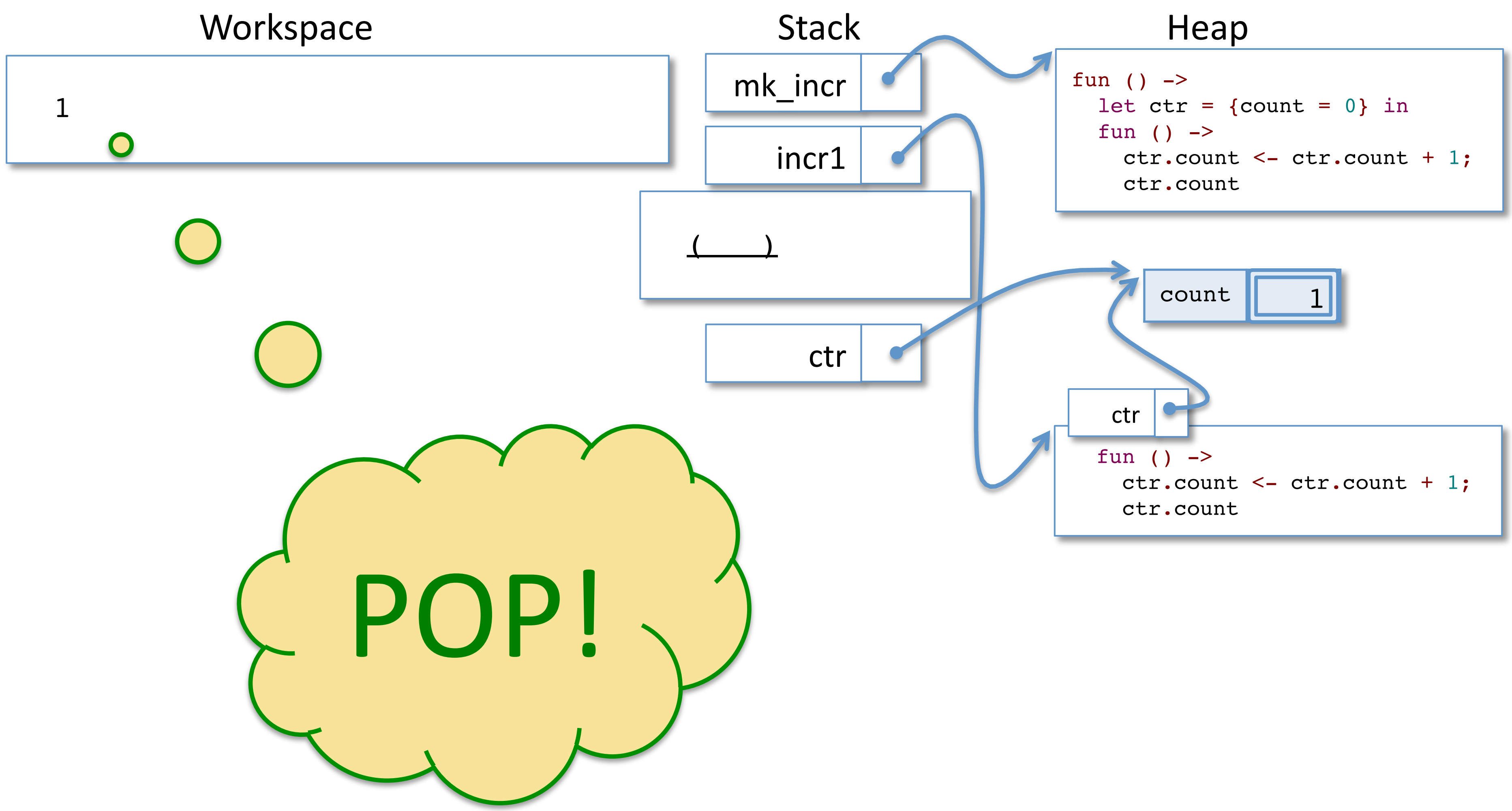
Now let's run "`incr1 ()`"



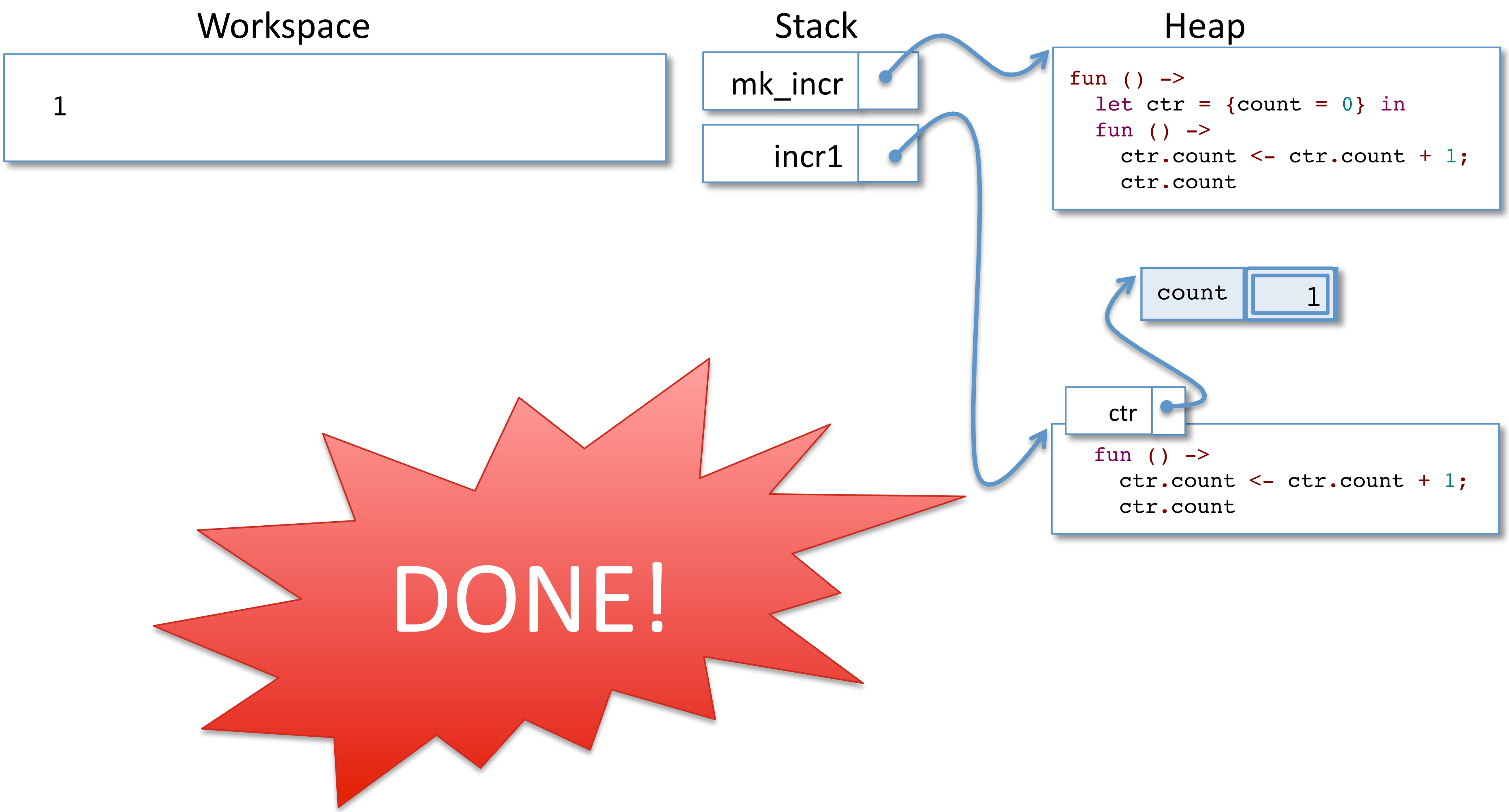
Now let's run "`incr1 ()`"



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



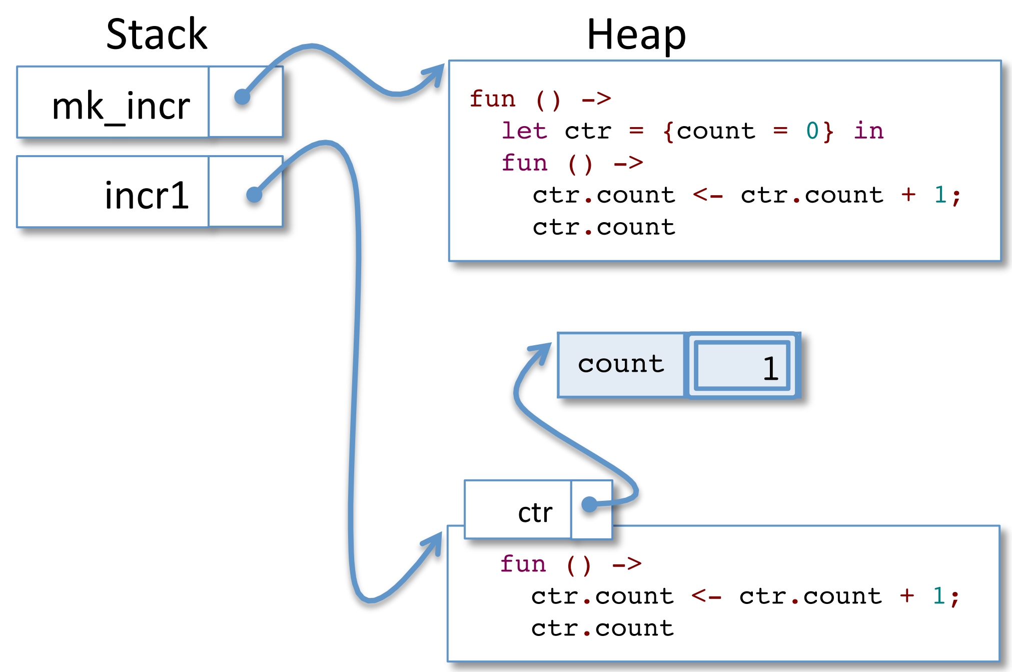
Now let's run "`incr1 ()`"



Now Let’s run `mk_incr` again

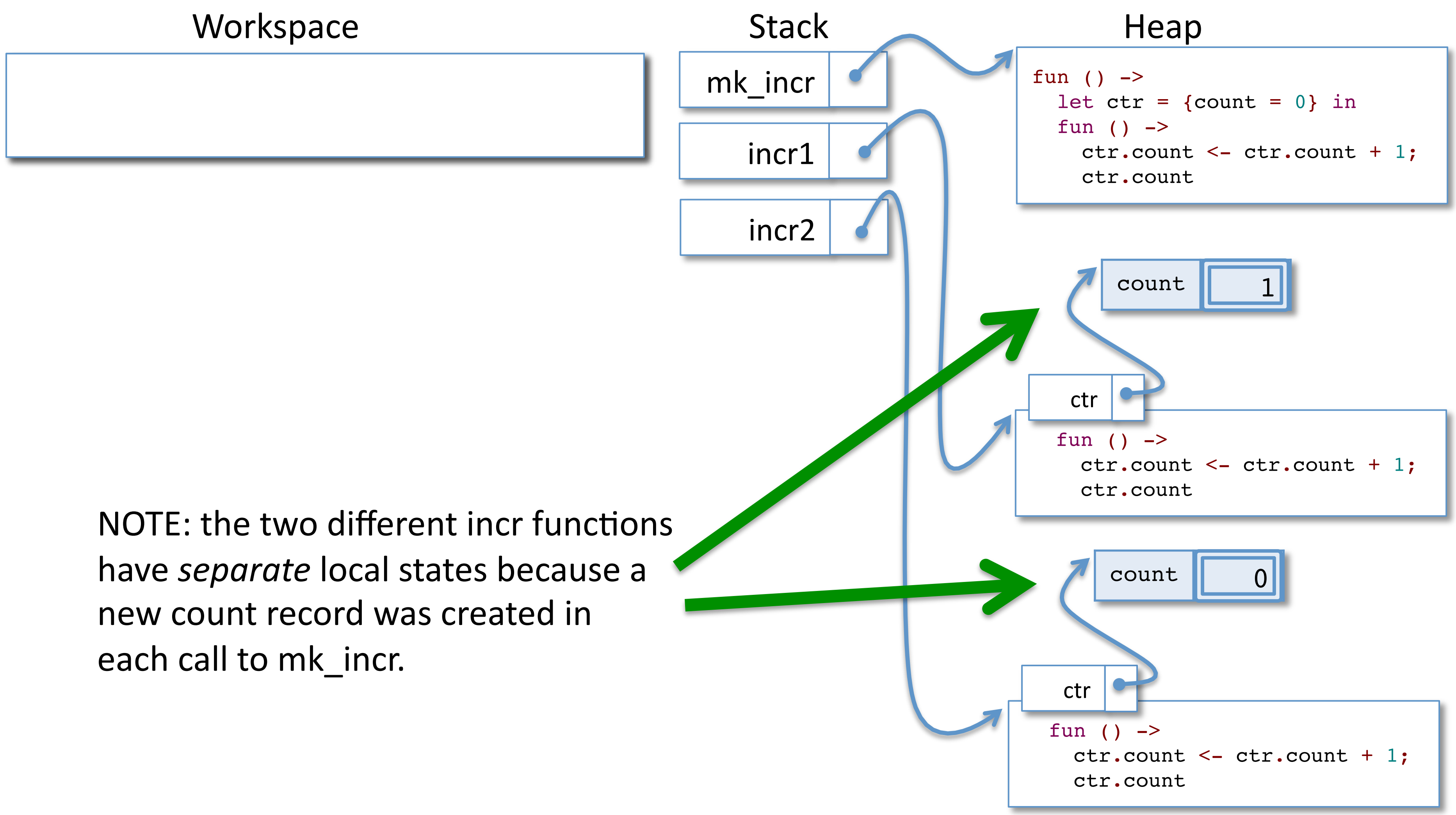
Workspace

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



…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

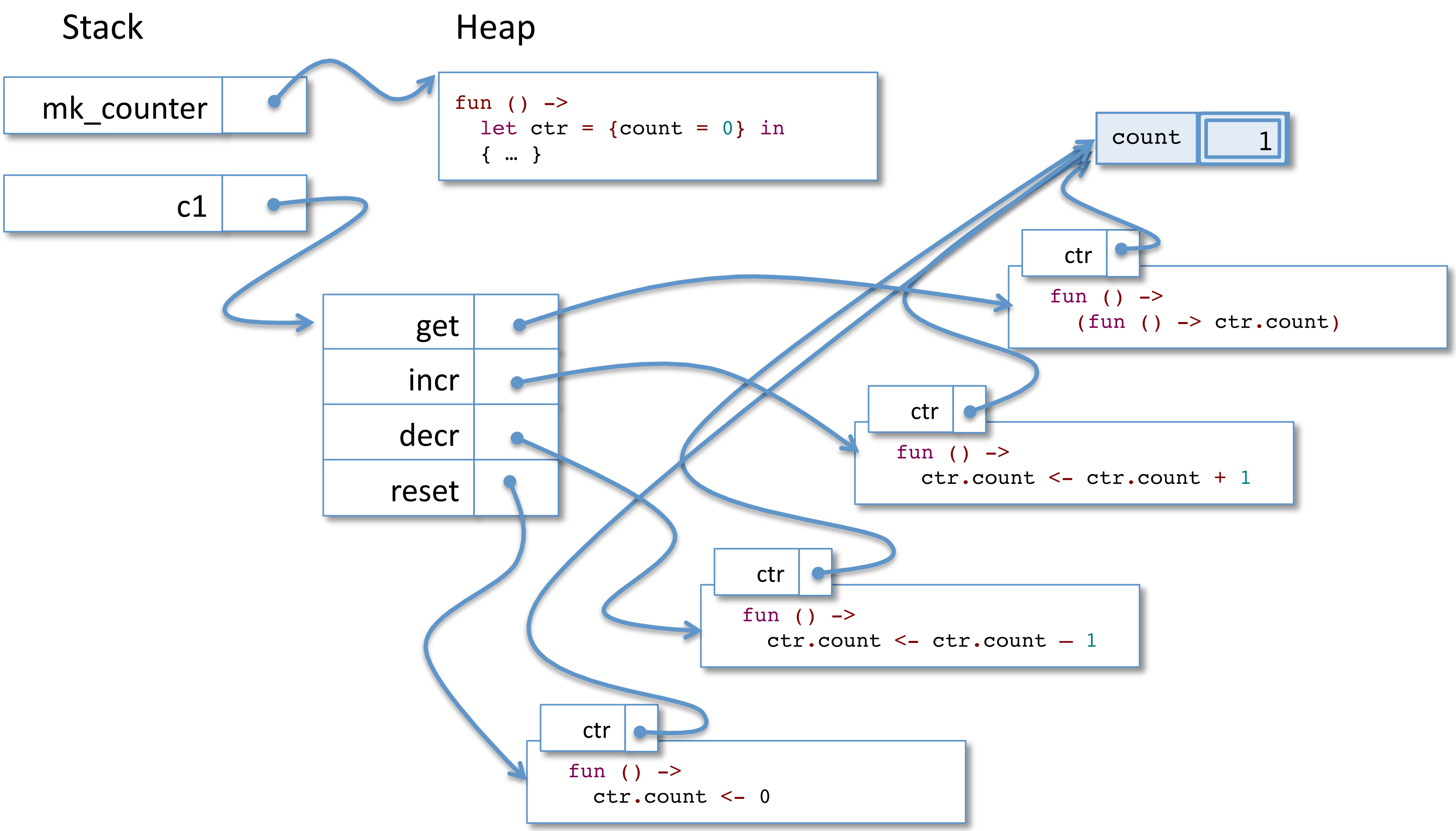
- mk_incr shows us how to create different instance of local state so that we can have several different counters.
- What if we want to bundle together several operations that share the same local state?
  - e.g. incr and decr operations that work on the same counter

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 ()))
```

The `'a ref` type

- It is common to need a reference to just one thing.
  - e.g. the counter state was just one integer
- …but the only mutable things in OCaml are fields of records.
- OCaml provides some syntactic sugar to help:

```
(* this type is built in to OCaml *)
type 'a ref = {mutable contents : 'a}

ref e      means    {contents = e}
!e         means    e.contents
e := v     means    e.contents <- v
```

Using ref notation

```
(* Implement counters using ref notation like this: *)
let mk_counter () =
  let c = ref 0 in
  {get   =  (fun () -> !c);
   incr =   (fun () -> c := (!c + 1));
   decr =   (fun () -> c := (!c - 1));
   reset = (fun () -> c := 0)}
```

Designing a GUI library

Designing a GUI

- 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 Graphics library documentation
- http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html

Step #1: Understand the Problem

- How do we go from the simple library to a functioning GUI application?
- What are the concepts involved in GUI libraries and how do they relate to each other?

GUI terminology

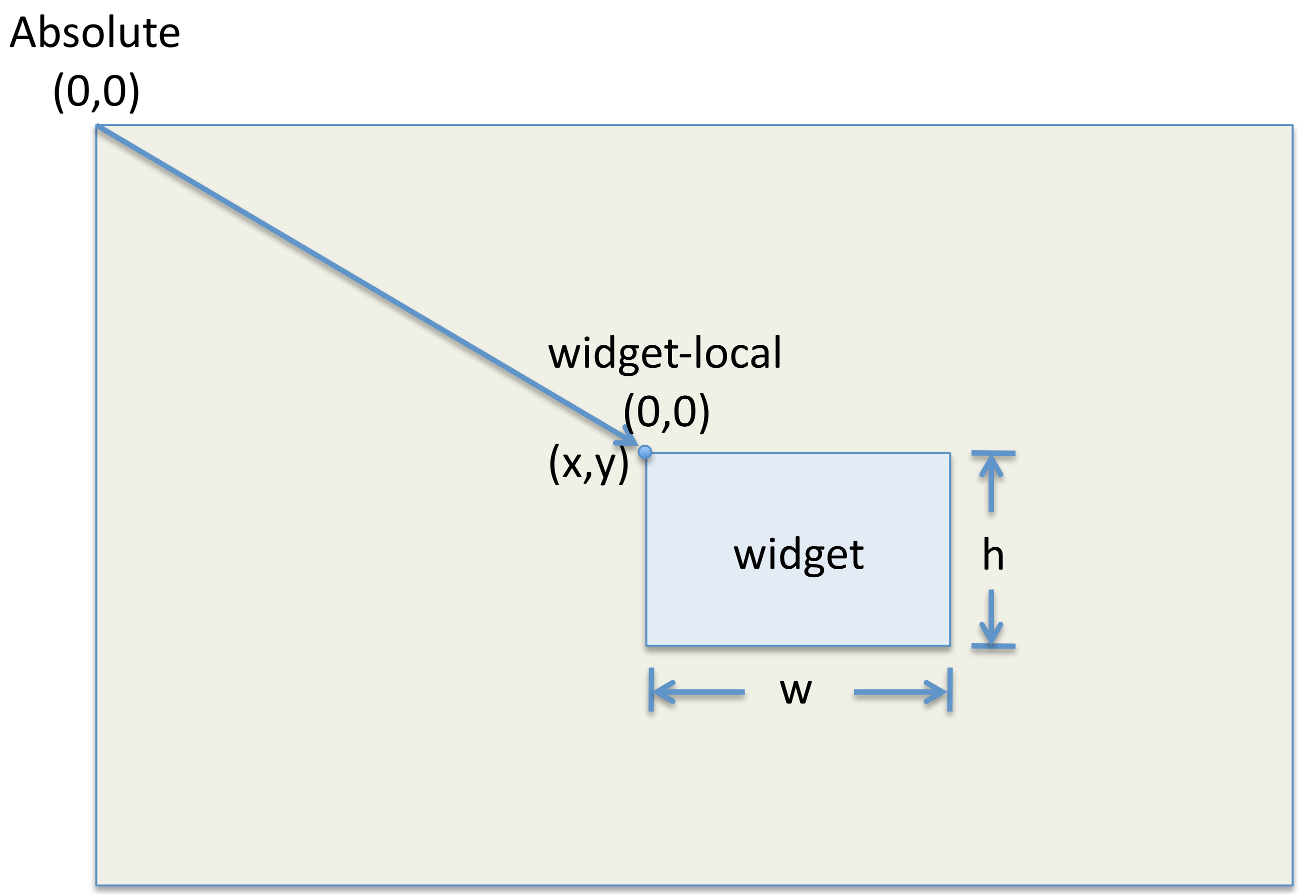
- Widget
  - Basic element of GUIs: buttons, checkboxes, windows, textboxes, canvases, scrollbars, labels
  - All have a position on the screen and know how to display themselves
  - May be composed of other widgets

First-class function

Design Challenge #1: Abstracting Layout

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

Graphics Contexts



A graphics context Gctx.t represents a position within the window, relative to which the widget-local coordinates should be interpreted. We can add additional context information that should be "inherited" by children widgets (e.g. current pen color).

Gctx.mli (excerpt)

```
type t  (** The main (abstract) type of graphics contexts *)

(** Creates a fresh Gctx.t *)
val create : unit -> t
(** Produce a new Gctx.t shifted  by (dx,dy) *)
val translate : t -> int * int -> t

(** A widget-relative position *)
type position = int * int
(** A width and height paired together. *)
type dimension = int * int

(** Display text at the given (widget-local) position *)
val draw_string : t -> position -> string -> unit
(** Calculates the size of a text when rendered. *)
val text_size : t -> string -> dimension
```