Programming Languages and Techniques (CIS1200)

Lecture 14

Mutable State, Aliasing, and the Abstract Stack Machine

Chapters 14 and 15

Announcements (1)

- HW04 available
  - Due in 2 weeks (October 15)
- Fall Break this week
  - No class on Friday
  - Recitations are cancelled for this week
  - No Office Hours from Thu-Sun
- No office hours for Benjamin next Monday (Oct 7)

Review: Mutable State

Records

- By default, all record fields are *immutable*—once initialized, they can never be modified.

```
type point = {x:int; y:int}

let p0 = {x=0; y=0}
;; do_something_with p0
;; print_endline ("p0.x = " ^ (string_of_int p0.x))

let p1 = {x=(p0.x + 1); y=(p0.y + 1)}
;; do_something_with p1
;; print_endline ("p1.x = " ^ (string_of_int p1.x))
```

This will always be 0, no matter what "do_something_with" does

This will always be 1…

Mutable Record Fields

- By default, all record fields are *immutable*—once initialized, they can never be modified.
- OCaml also supports *mutable* fields that can be imperatively updated by the "set" command: `record.field <- val`

note the 'mutable' keyword

```
type point = {mutable x:int; mutable y:int}

let p0 = {x=0; y=0}
(* set the x coord of p0 to 17 *)
;; p0.x <- 17
;; print_endline ("p0.x = " ^ (string_of_int p0.x))

p0.x = 17
```

in-place update of p0.x

Record Update

- Functions can assign to mutable record fields
- Note that the return type of '<−' is unit
  - i.e., it is a *command*

```
type point = {mutable x:int; mutable y:int}

(* a command to shift a point by dx,dy *)
let shift (p:point) (dx:int) (dy:int) : unit =
  p.x <- p.x + dx;
  p.y <- p.y + dy
```

- the result type of `shift` is also unit
  - i.e., `shift` is a *user-defined* command

What answer does the following function produce when called?

```
type point = {mutable x:int; mutable y:int}

let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  p2.x <- 42;
  p1.x
```

1. 17
2. something else
3. sometimes 17 and sometimes something else
4. f is ill typed

ANSWER: 3

The Challenge of Mutable State: Aliasing

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  p2.x <- 42;
  p1.x
```

Consider this call to f:

```
let p0 = {x=0; y=0} in
  f p0 p0
```

Two identifiers are said to be *aliases* if they both name the *same* mutable record. Inside f, the identifiers p1 and p2 might or might not be aliased, depending on which arguments are passed in.

SEE THE COURSE NOTES FOR MORE ON THIS EXAMPLE

The Abstract Stack Machine

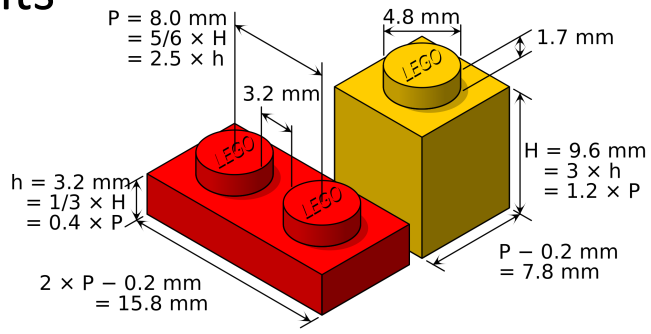
A model of imperative computation

or,

Location, Location, Location!

We Need a New Computation Model

- The simple model of computation we've used so far works well for pure *value-oriented* programming
  - "Observable behavior" of a value is completely determined by its structure
  - Two different calls to the same function with the same arguments always yield the same results
  - These properties justify "replace equals by equals" reasoning

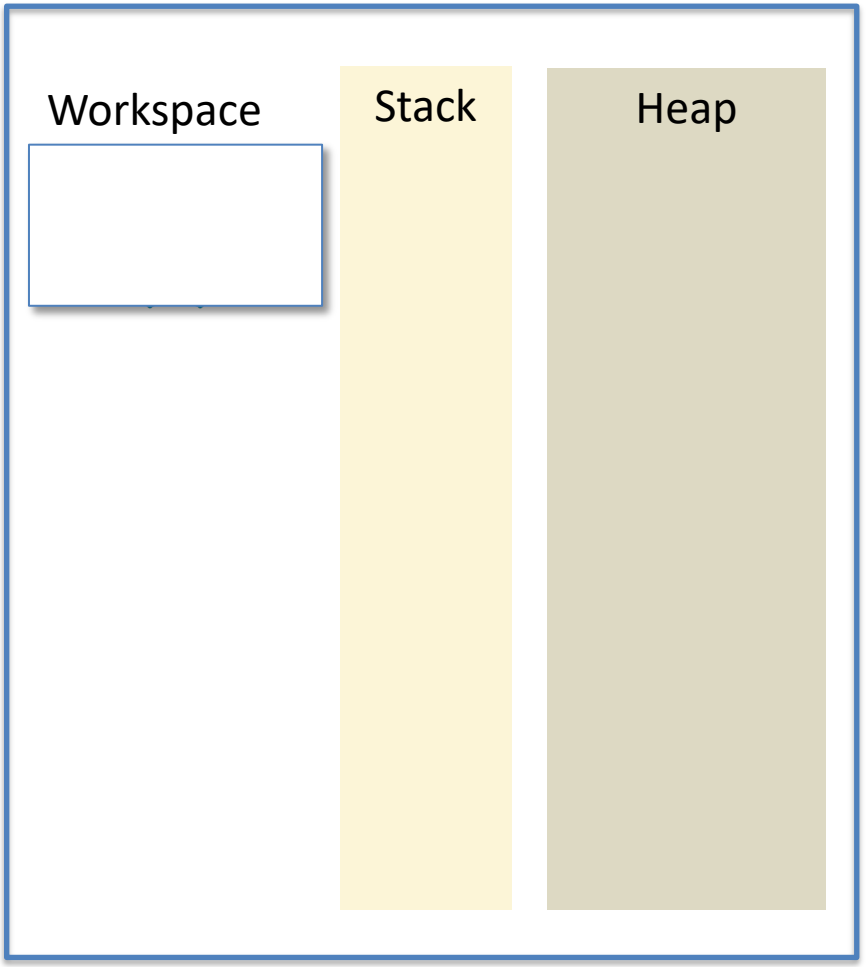


- But with mutable state…
  - The *location* of values matters, not just their structure
  - Results returned by functions are *not* fully determined by their arguments — can also depend on "hidden" mutable state

Abstract Stack Machine

Three “spaces”…

- workspace
  - the expression the computer is currently simplifying
  - abstraction of the CPU
- stack
  - temporary storage for local variables and saved work
  - abstraction of (part of) RAM
- heap
  - storage area for large data structures
  - abstraction of (part of) RAM



Abstract stack machine

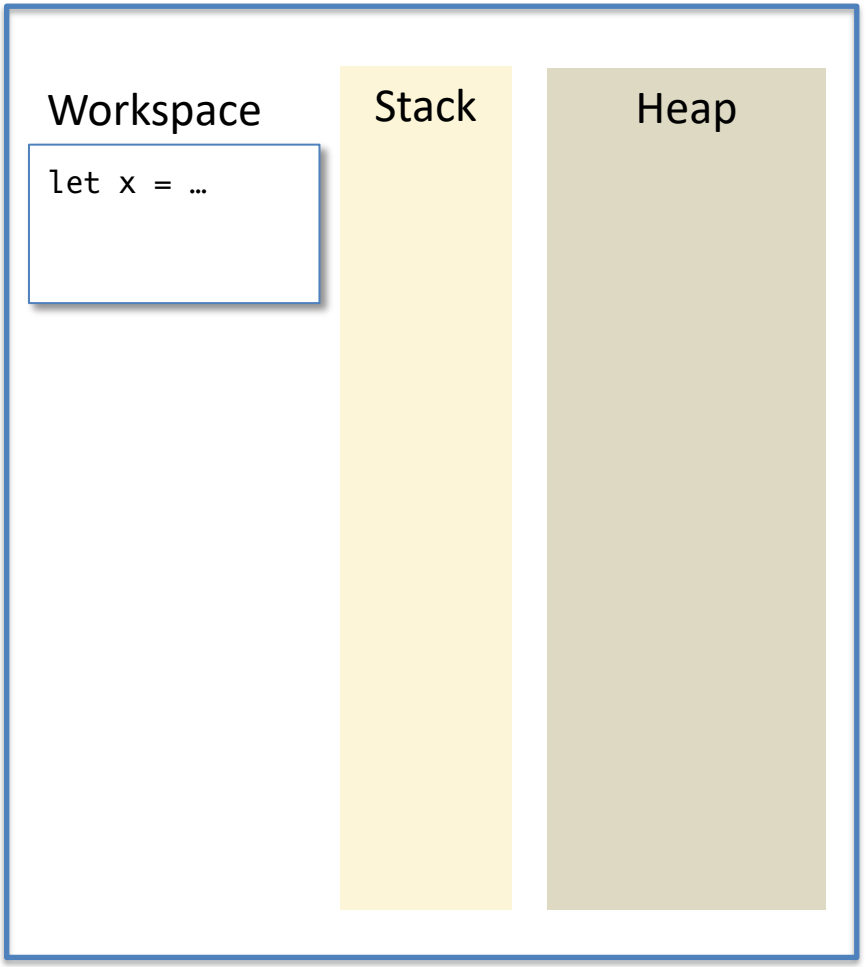
Abstract Stack Machine

Initial state:

- workspace contains whole program
- stack and heap are empty

Machine operation:

- In each step, choose “next part” of the workspace expression and simplify it
- (Sometimes this will change the stack and/or heap)
- Stop when there are no more simplifications to be done



Abstract stack machine

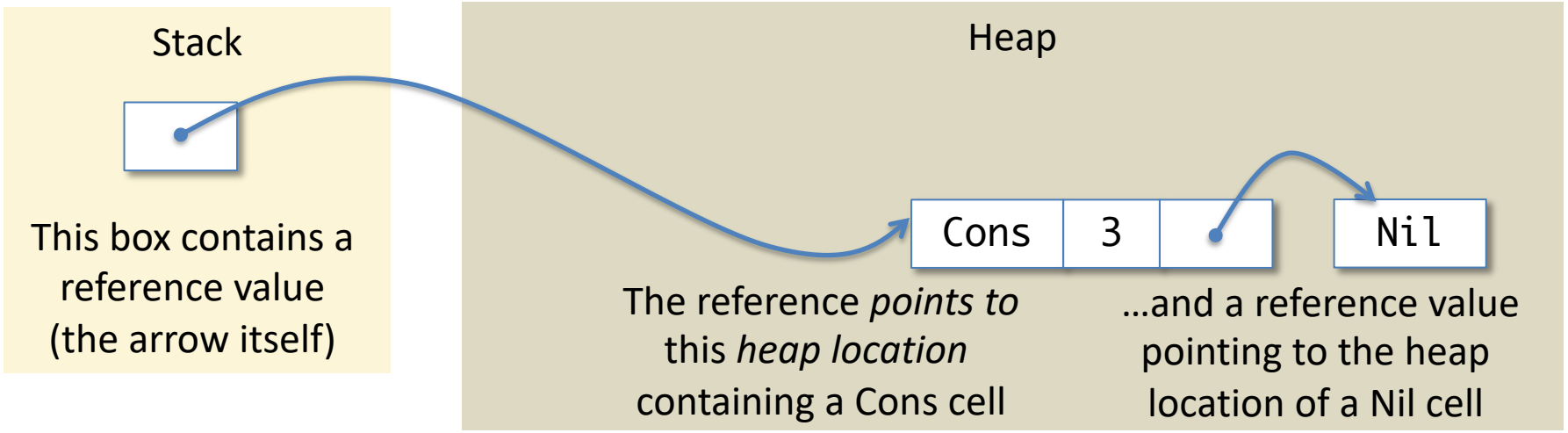
Values and References

A *value* is either:

- a *primitive value* like an integer, or,
- a *reference* to a location in the heap

A reference value is the *address (location)* of data in the heap.

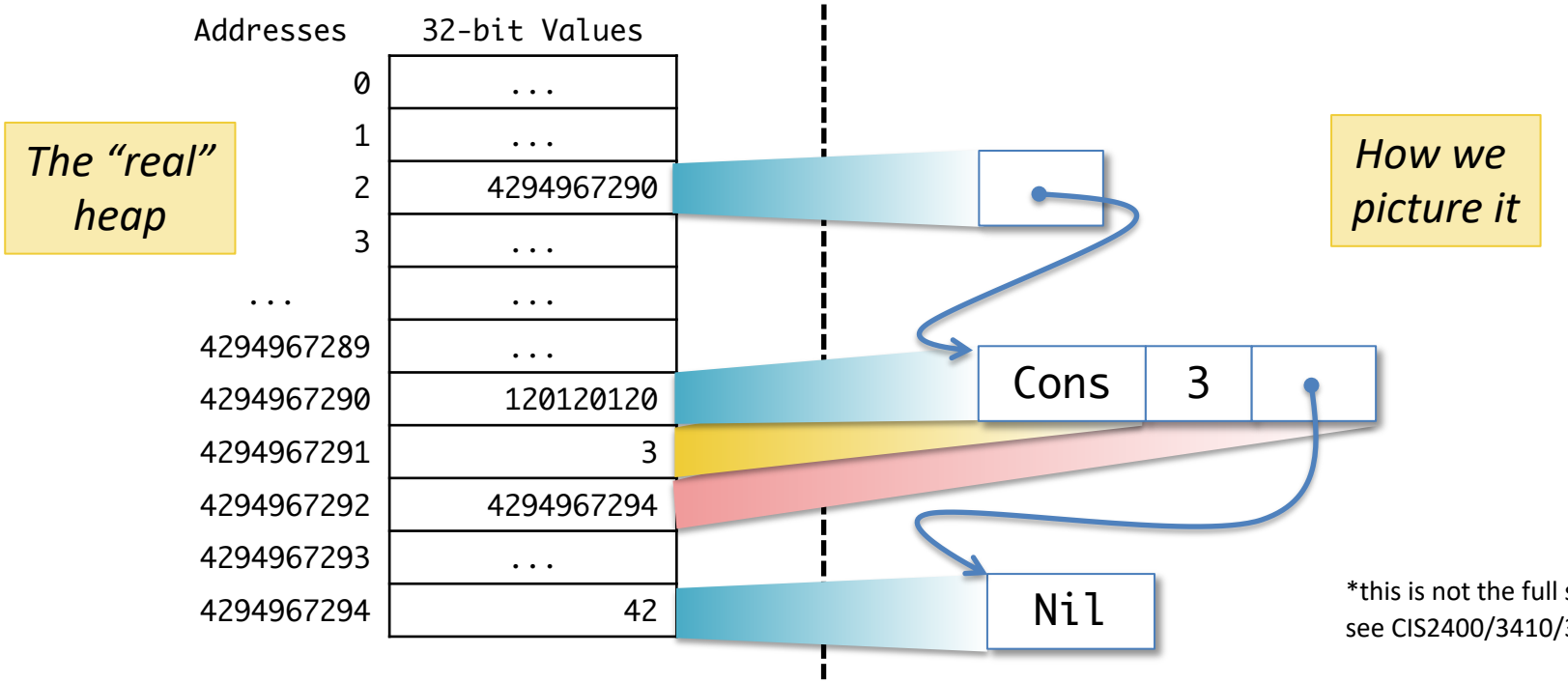
We draw a reference value as an arrow pointing to the data "located at" this address



References are an Abstraction

In a real* computer, the memory consists of an array of 32-bit words, numbered 0 … 2^{32}-1 (for a 32-bit machine)

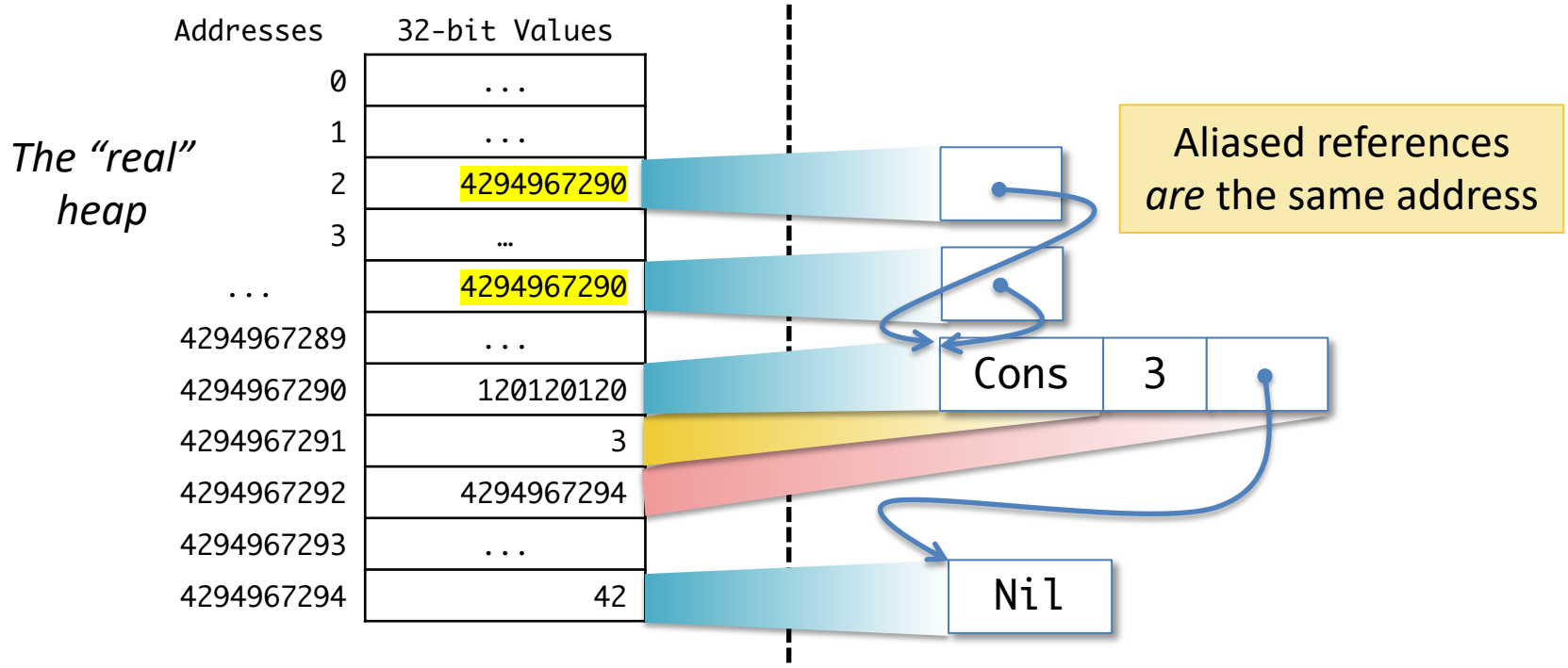
- A *reference* (*pointer*) is an address indicating *where* to look up a value
- Data structures are usually laid out in contiguous blocks of memory
- Constructor tags are just numbers chosen by the compiler
  e.g., `Nil` = 42 and Cons = 120120120



*this is not the full story;
see CIS2400/3410/3800/4710

References are an Abstraction

- Usually, the specific addresses chosen for where to place data don't matter
  - programmers don't want to think at that level of detail
  - *aliasing* (i.e., sharing the same location) is what matters



The ASM: Simplifying variables, operators, `let` expressions, and `if` expressions

Using the stack instead of substitution

Simplification

Workspace

```
let x = 10 + 12 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

```
let x = 10 + 12 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

```
let x = 22 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Simplification

Workspace

```
let x = 22 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

Heap

Instead of *substituting* x with its value in the rest of the program…

Simplification

Workspace

```
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

we *push* a binding for x onto the stack

Simplification

Workspace

```
let y = 2 + x in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Variable x is not a value, so *look it up* in the stack

Simplification

Workspace

```
let y = 2 + 22 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let y = 2 + 22 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let y = 24 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
let y = 24 in
  if x > 23 then 3 else 4
```

Stack

x	22

Heap

Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

x	22

y	24

Heap

Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

x	22
y	24

Heap

Looking up x in the stack proceeds from most recent entries to the least recent entries. Note that the "top" (most recent part) of the stack is drawn on the *bottom* of the diagram.

Simplification

Workspace

```
if 22 > 23 then 3 else 4
```

Stack

x	22

y	24

Heap

Simplification

Workspace

```
if 22 > 23 then 3 else 4
```

Stack

x	22
y	24

Heap

Simplification

Workspace

```
if false then 3 else 4
```

Stack

x	22

y	24

Heap

Simplification

Workspace

```
if false then 3 else 4
```

Stack

x	22

y	24

Heap

Simplification

Workspace

4

Stack

x	22

y	24

Heap

What to simplify next?

- At each step, the ASM finds the leftmost *ready subexpression* in the workspace
- An expression involving a ***primitive operator*** (e.g., "`+`") is *ready* if all its arguments are values
  - Expression is replaced with its result
- A ***let expression*** `let x : t = e in body` is *ready* if `e` is a value
  - A new binding for `x` to `e` is added at the end of the stack
  - let expression is replaced with `body` in the workspace
- A ***variable*** is always *ready*
  - The variable is replaced by its binding in the stack, searching from the most recent bindings (this search can never fail!)
- A ***conditional expression*** `if e then e1 else e2` is *ready* if `e` is either `true` or `false`
  - The workspace is replaced with either `e1` (if `e` is `True`) or `e2` (if e is `False`)

14: Simplifying code on the ASM

0

1

0%

2

0%

3

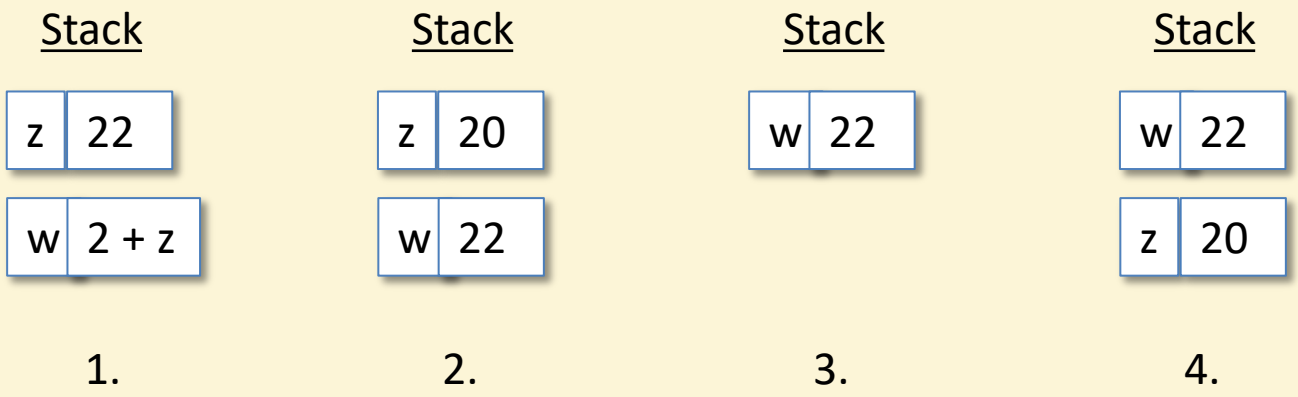
0%

4

0%

What does the Stack look like after simplifying the following code on the workspace?

```
let z = 20 in
let w = 2 + z in
  w
```



1. 2. 3. 4.

ANSWER: 2

14: Simplifying code on the ASM

0

1

0%

2

0%

3

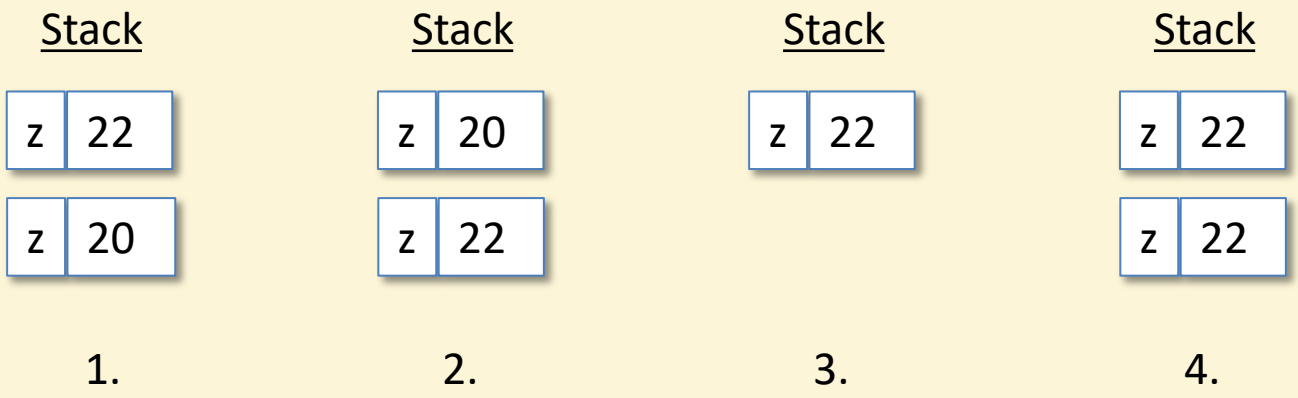
0%

4

0%

What does the Stack look like after simplifying the following code on the workspace?

```
let z = 20 in
let z = 2 + z in
  z
```



ANSWER: 2

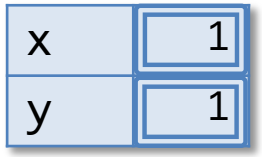
Mutable Records

- The reason for introducing the ASM model is to make heap locations and sharing *explicit*
  - Now we can say what it means to "mutate a heap value *in place*."

```
type point = {mutable x:int; mutable y:int}

let p1 : point = {x=1; y=1}
let p2 : point = p1
let ans : int = (p2.x <- 17; p1.x)
```

- We draw a record in the heap like this:
  - The doubled outlines indicate that those cells are mutable
  - Everything else is **immutable**



A point record in the heap.

Allocate a Record

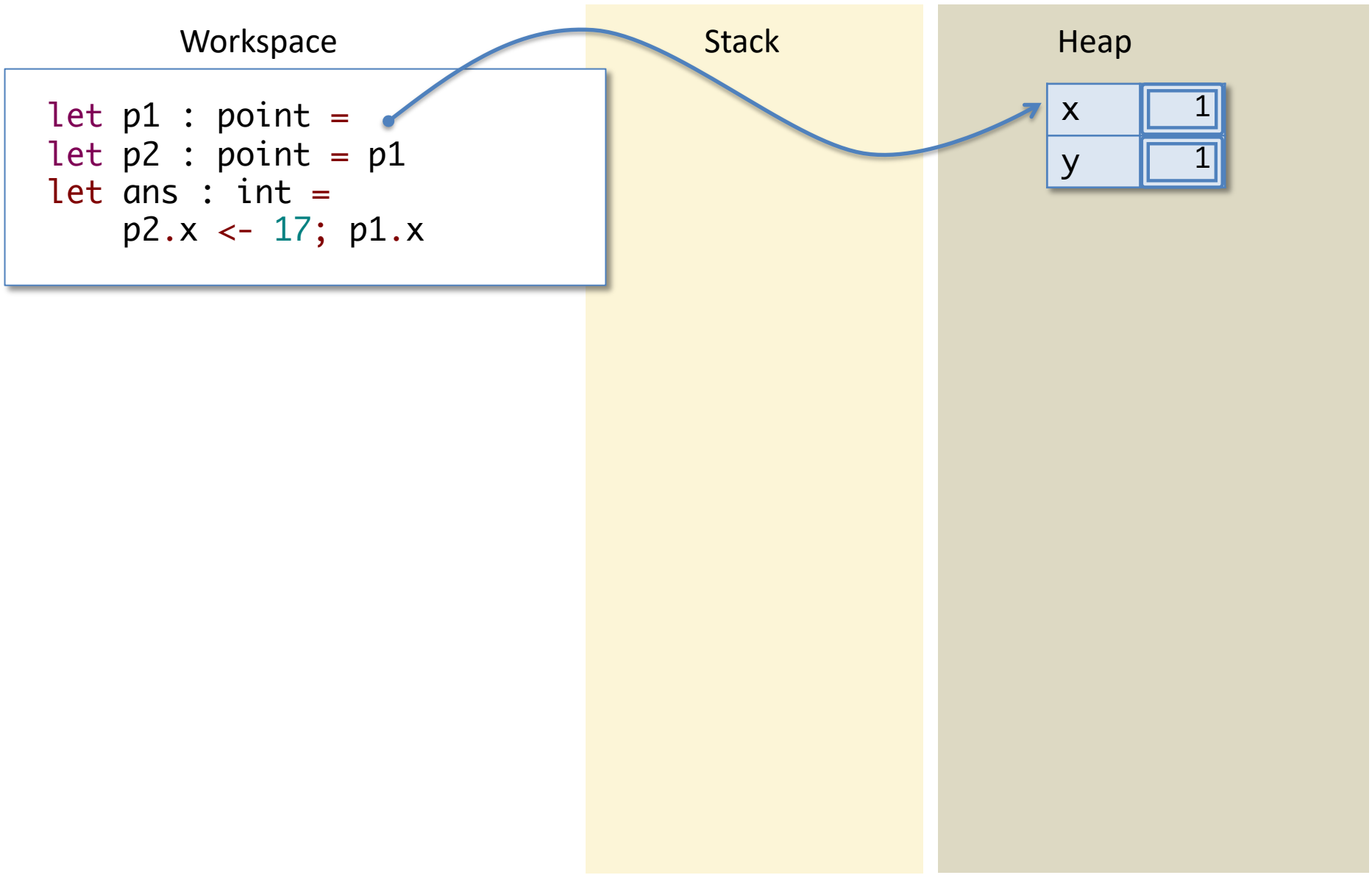
Workspace

```
let p1 : point = {x=1; y=1}
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```

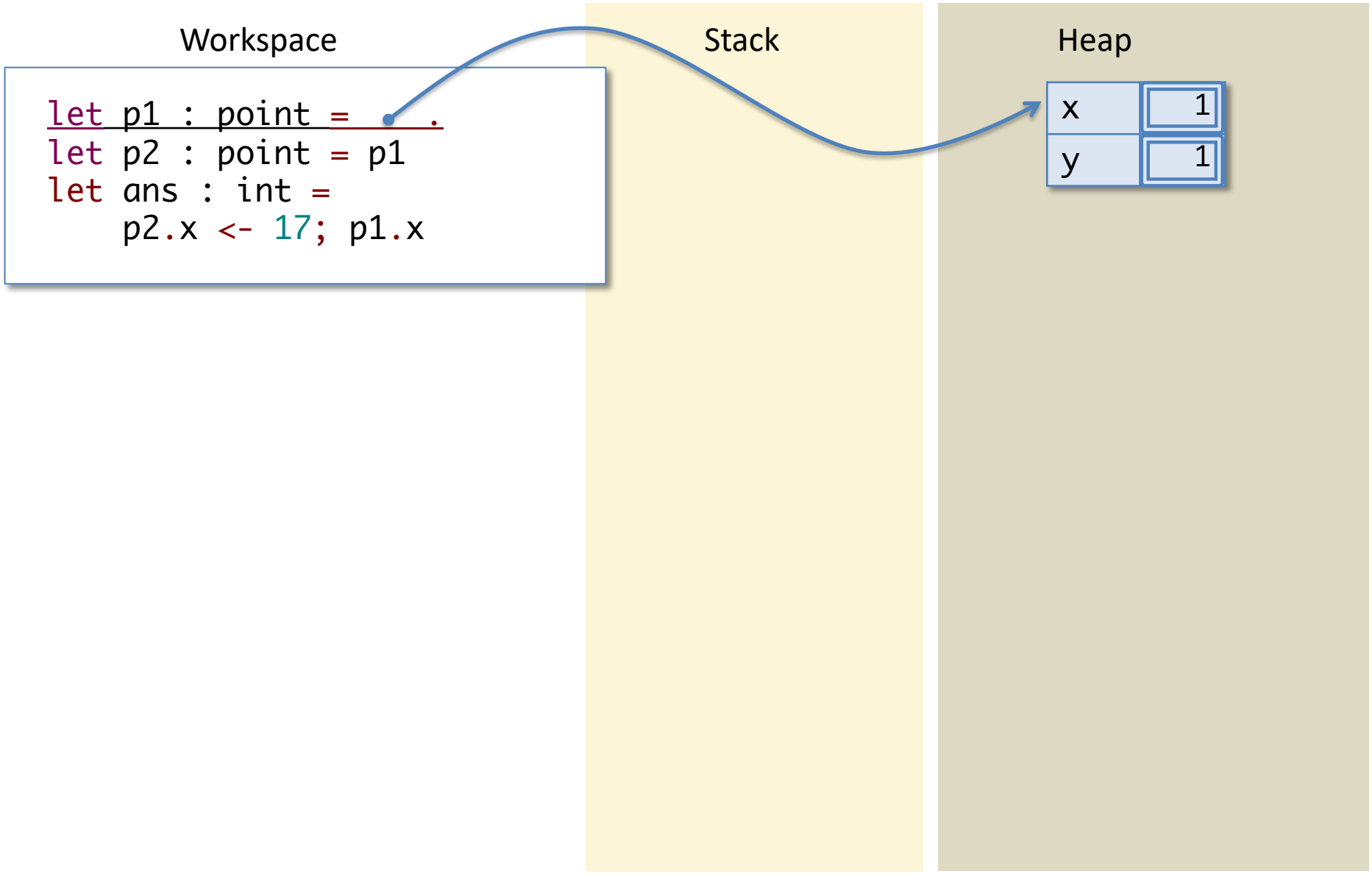
Stack

Heap

Allocate a Record



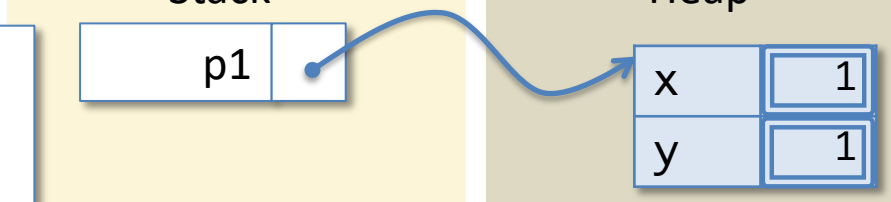
Let Expression



Push p1

Workspace

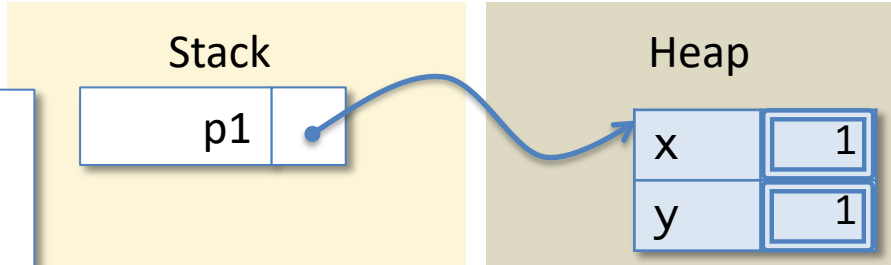
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



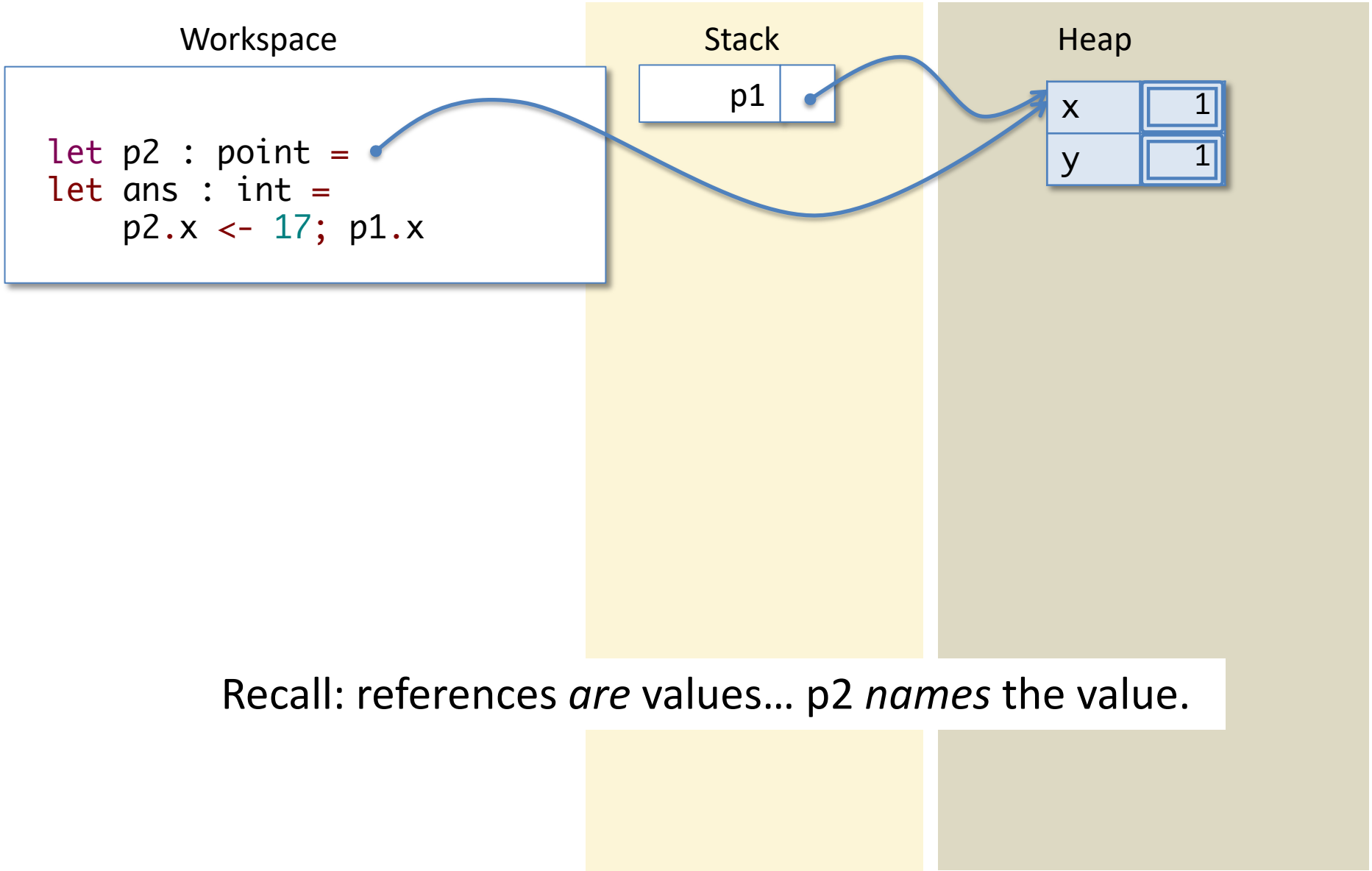
Look Up ‘p1’

Workspace

```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```

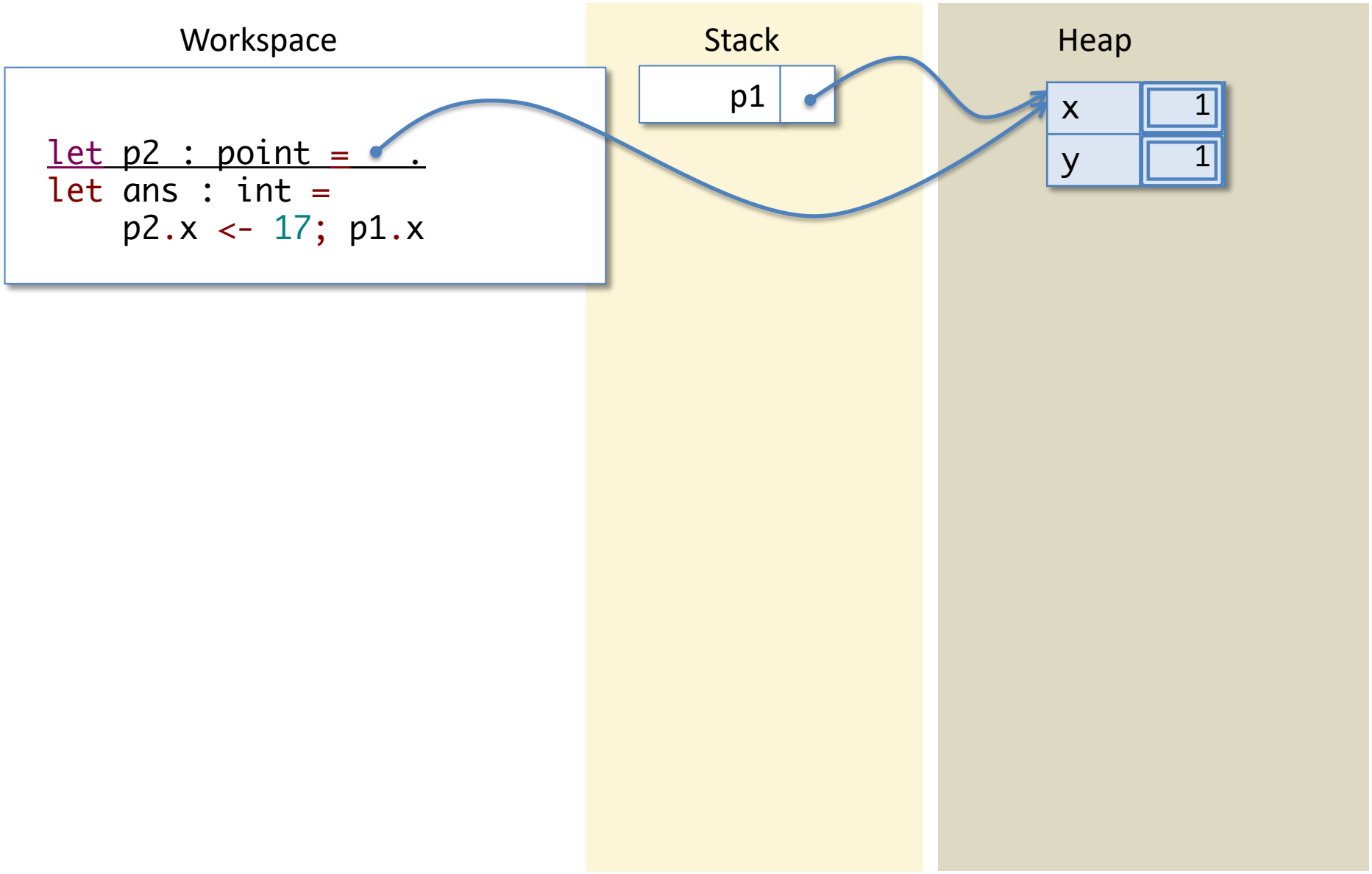


Look Up ‘p1’



Recall: references *are* values… p2 *names* the value.

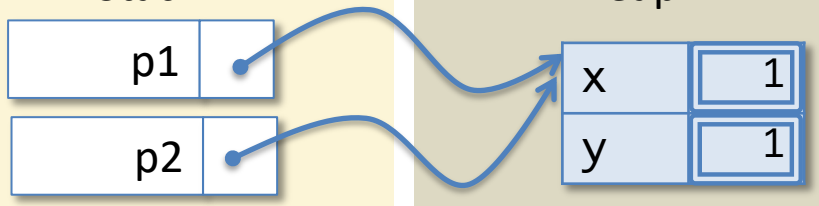
Let Expression



Push p2

Workspace

```
let ans : int =
    p2.x <- 17; p1.x
```

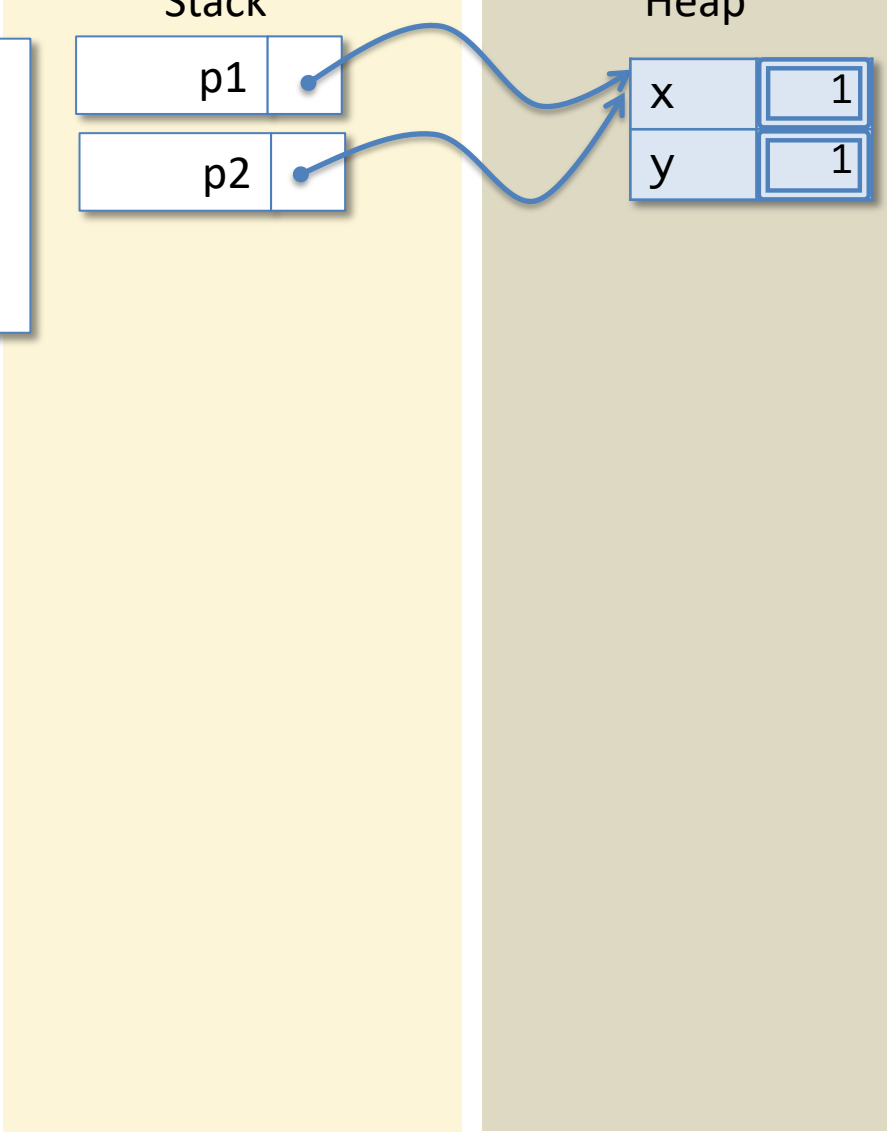


Now p1 and p2 are references to the *same* heap record.
They are *aliases* – two different names for the *same location*.

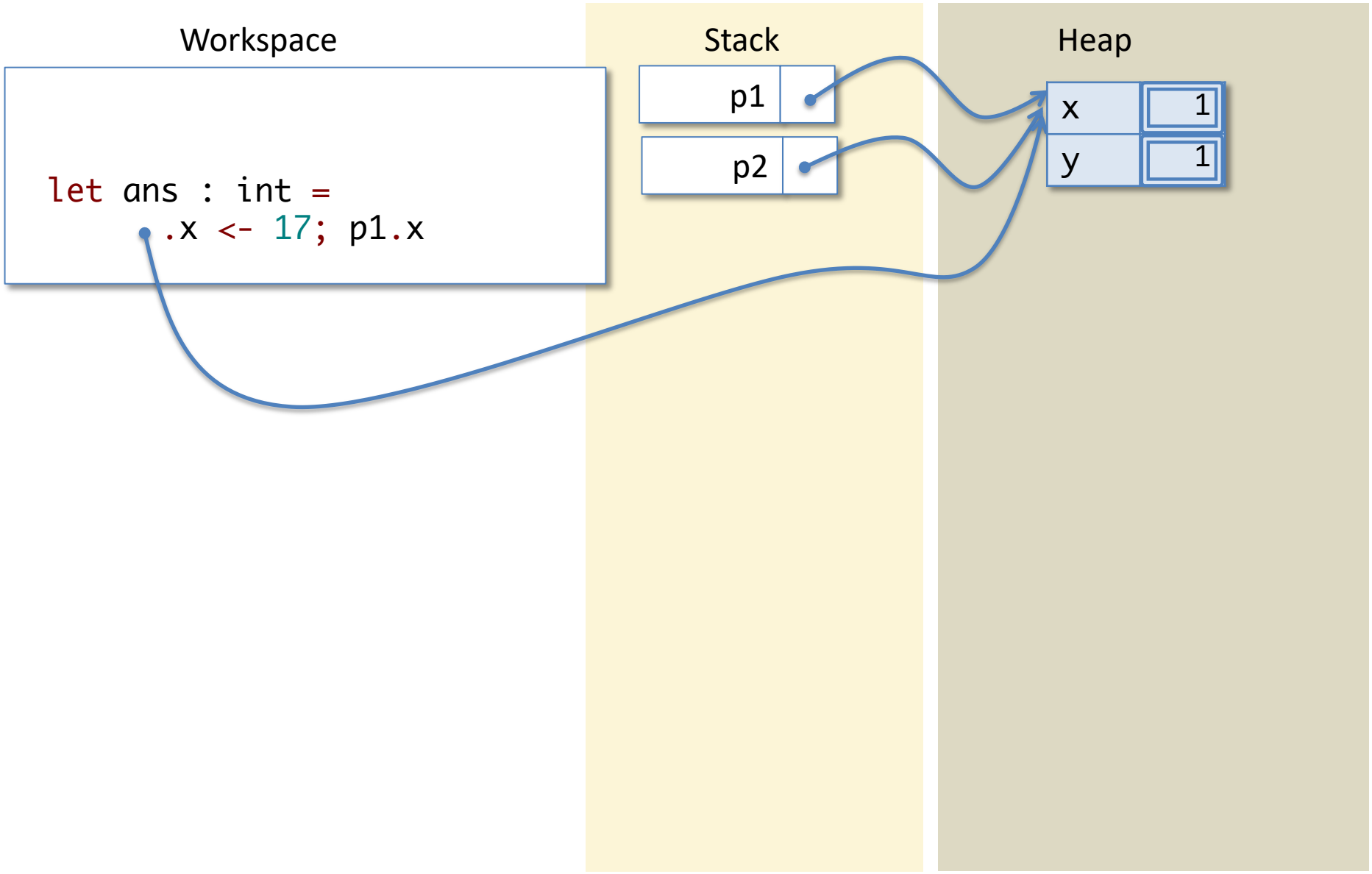
Look Up ‘p2’

Workspace

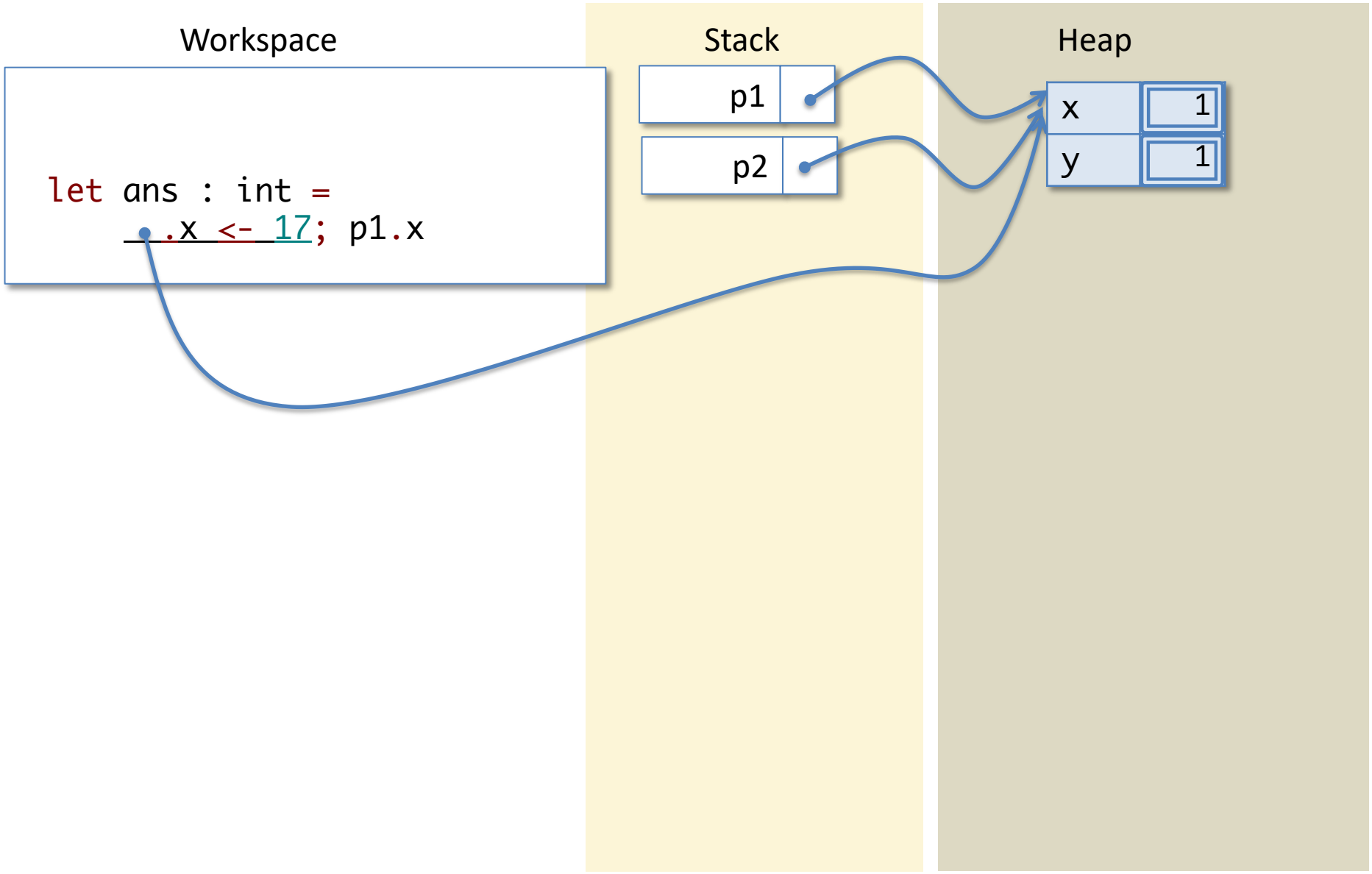
```
let ans : int =
    p2.x <- 17; p1.x
```



Look Up ‘p2’



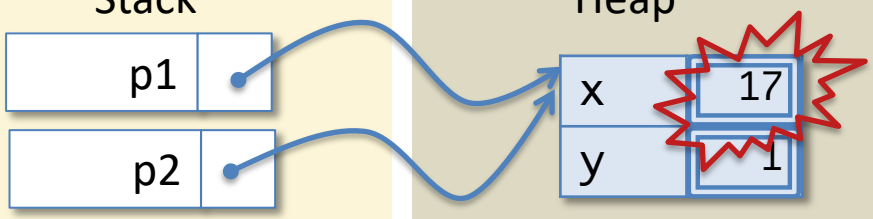
Assign to x field



Assign to x field

Workspace

```
let ans : int =
    (); p1.x
```

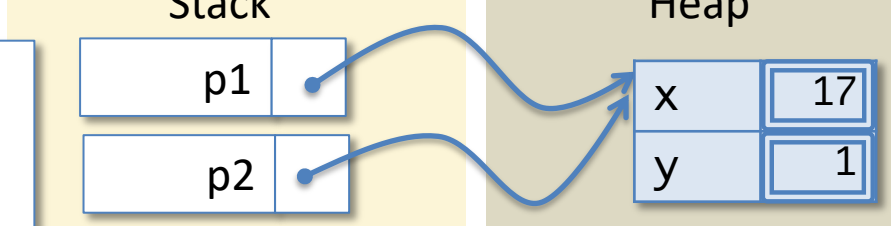


This is the step in which the ‘imperative’ update occurs. The mutable field x has been modified in place to contain the value 17.

Sequence ‘;’ Discards Unit

Workspace

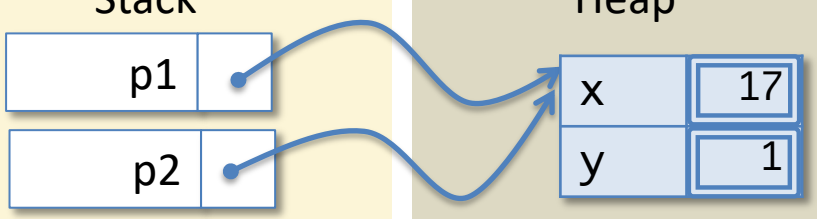
```
let ans : int =
    (); p1.x
```



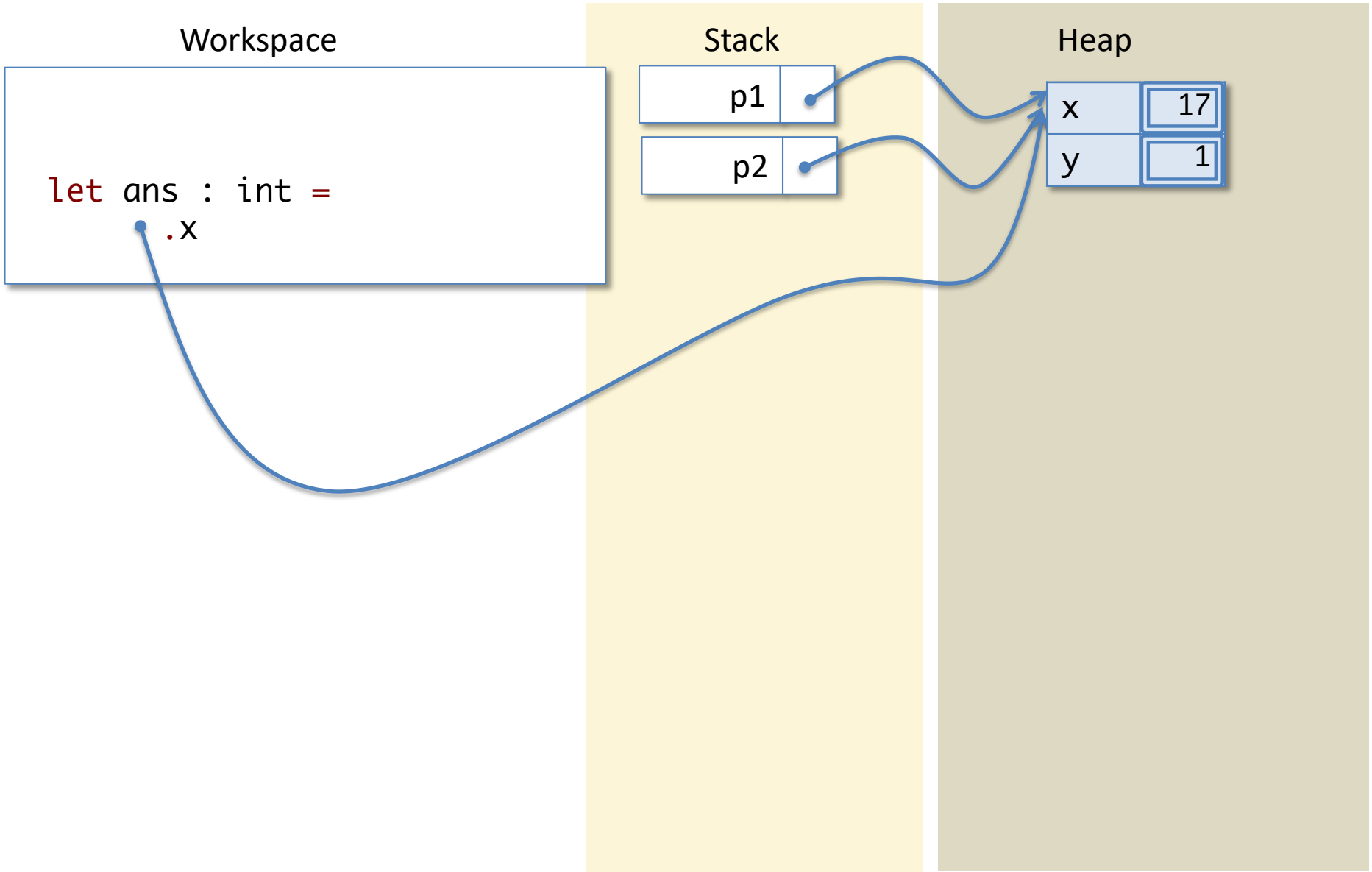
Look Up ‘p1’

Workspace

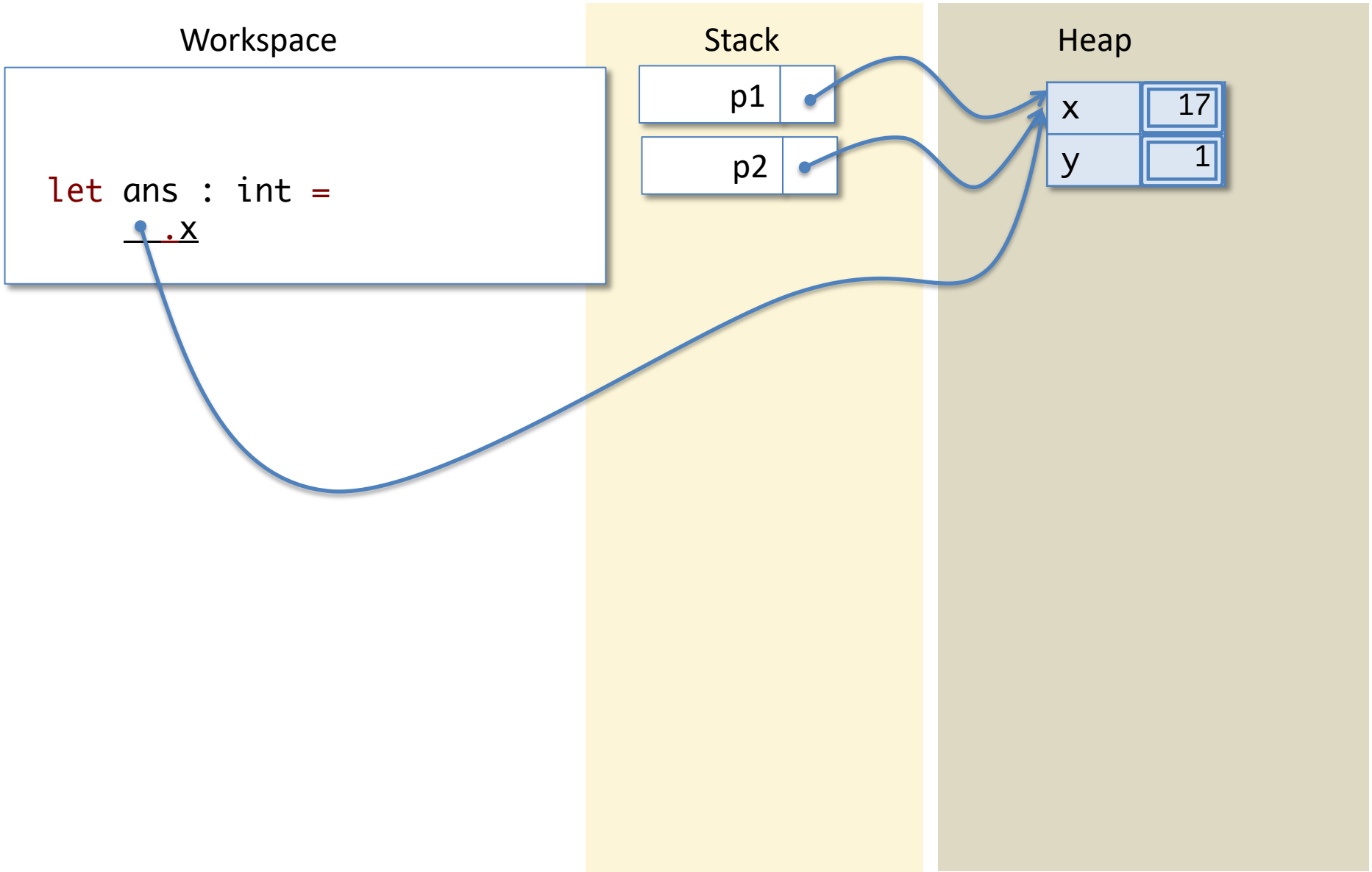
```
let ans : int =
    p1.x
```



Look Up ‘p1’



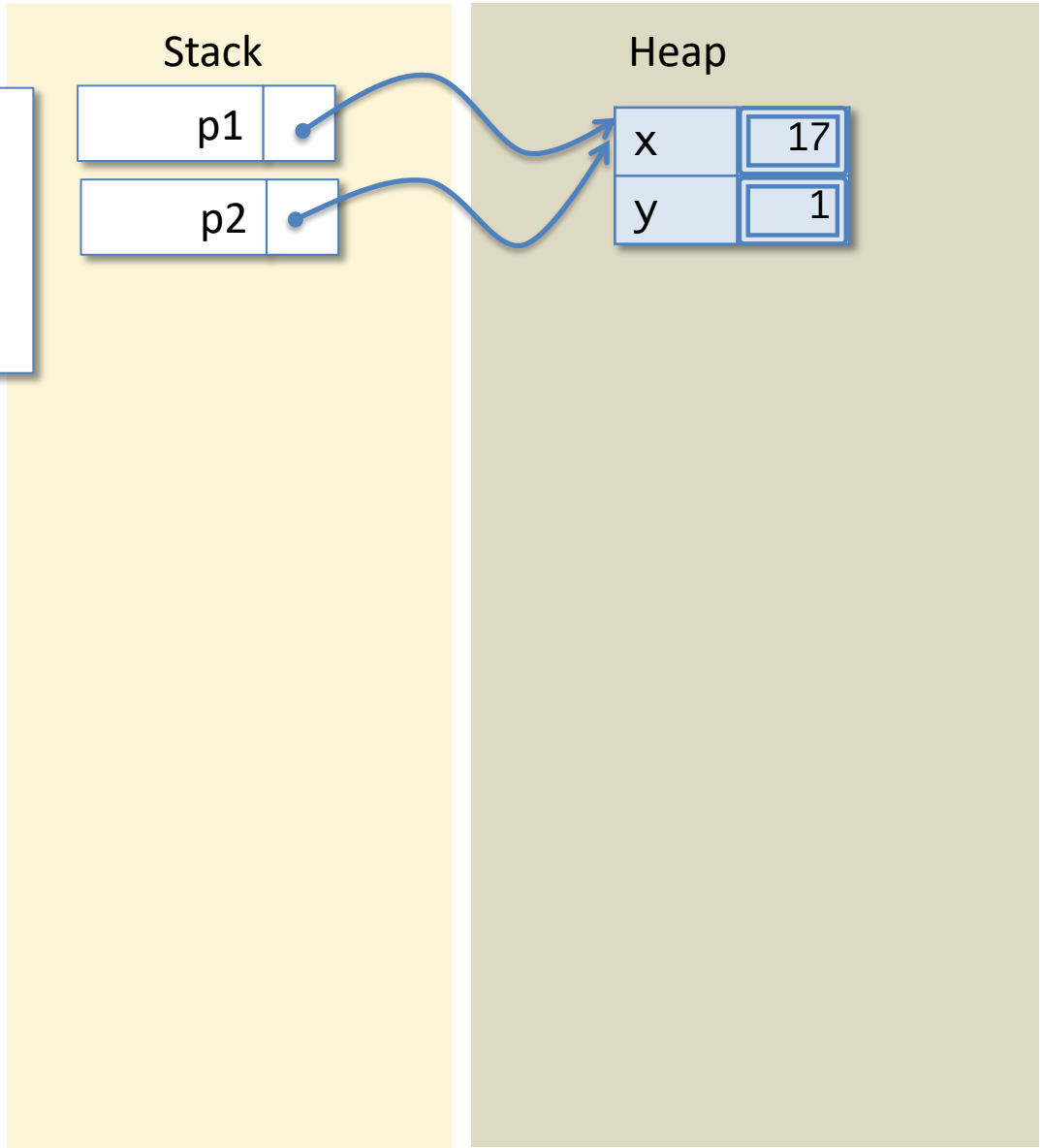
Project the ‘x’ field



Project the ‘x’ field

Workspace

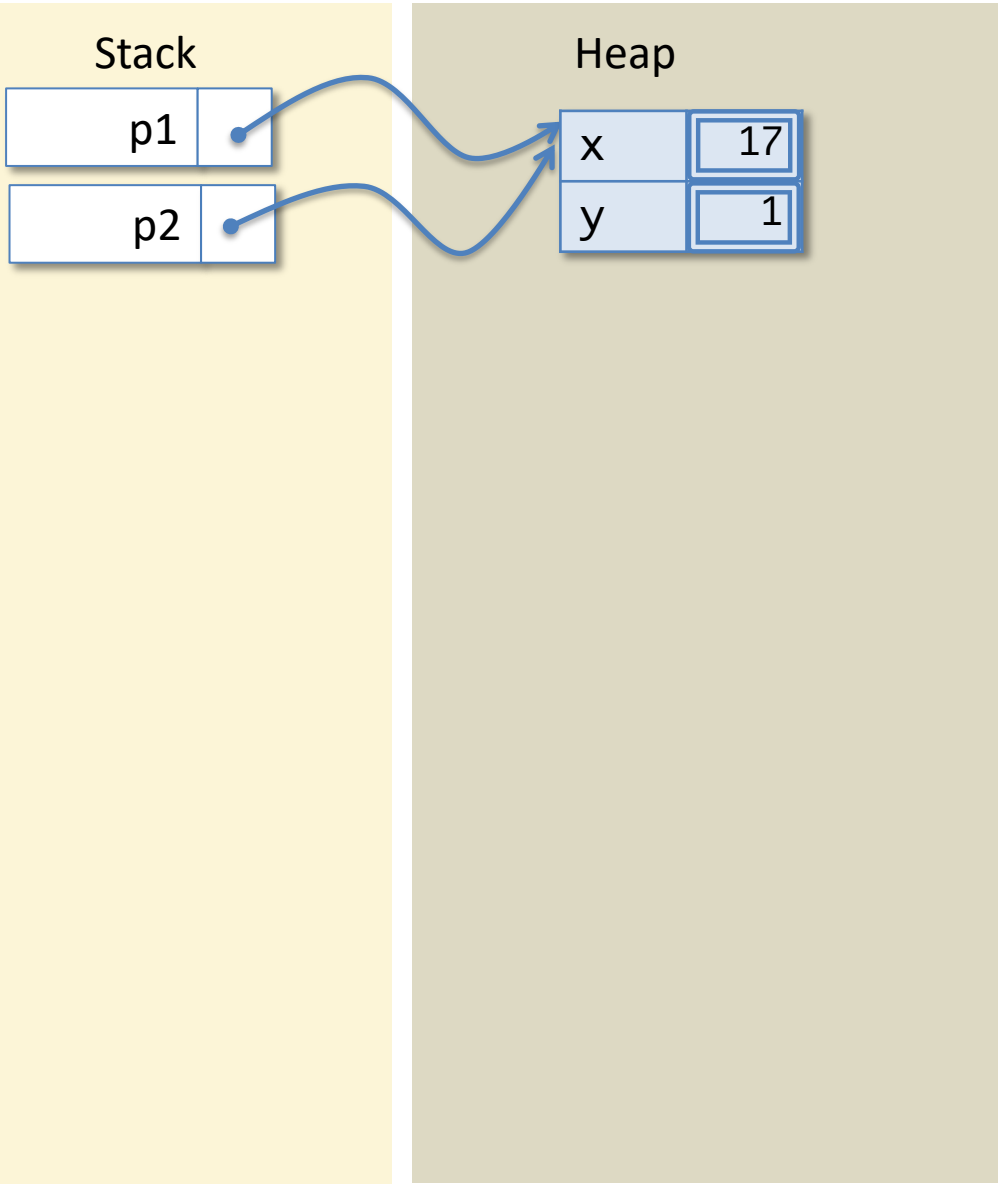
```
let ans : int =
    17
```



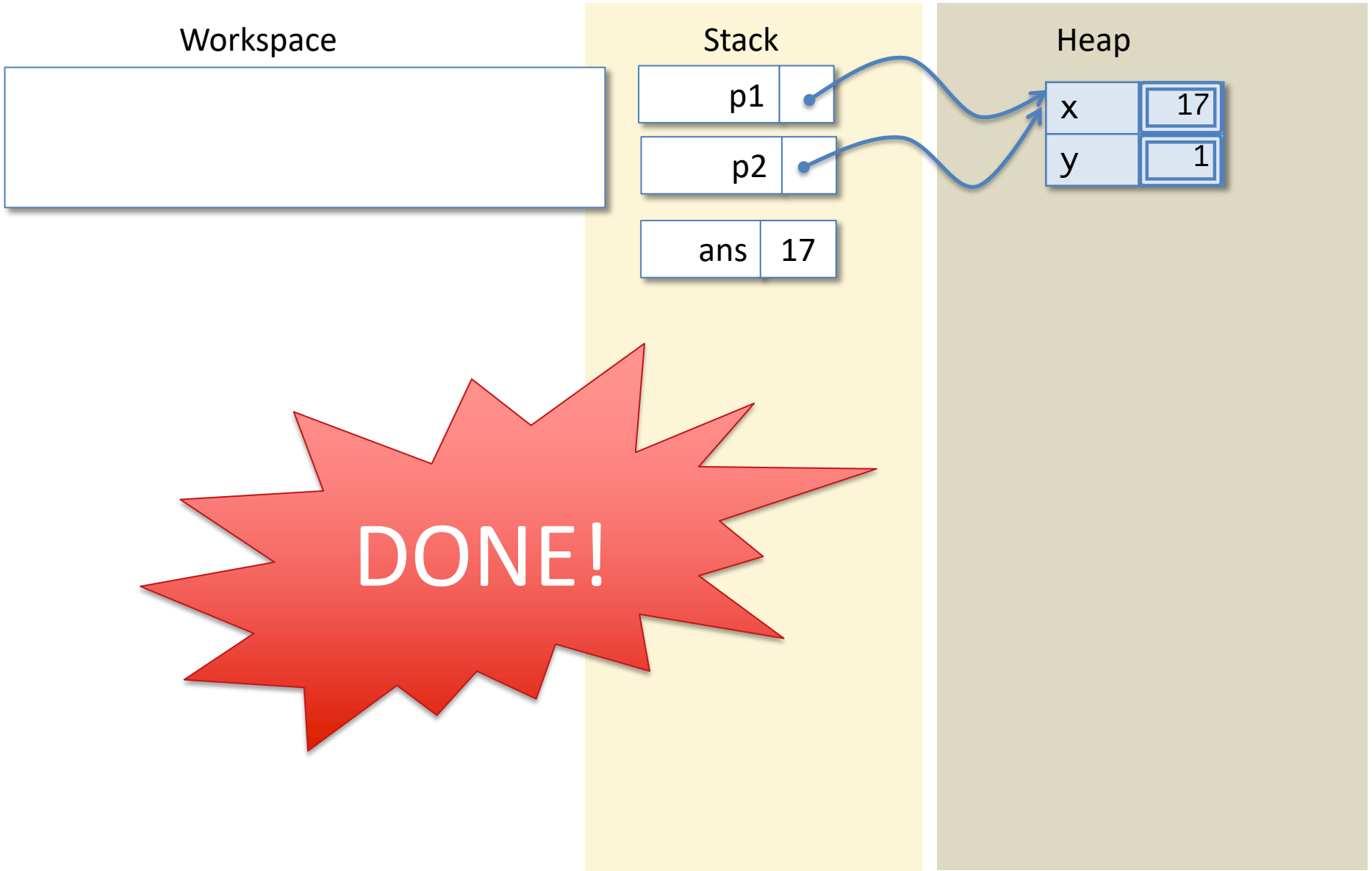
Let Expression

Workspace

```
let ans : int =
    17
```



Push ans



14: What answer does the following function produce when called?

17

0%

42

0%

sometimes 17 and sometimes 42

0%

f is ill typed

0%

What answer does the following function produce when called?

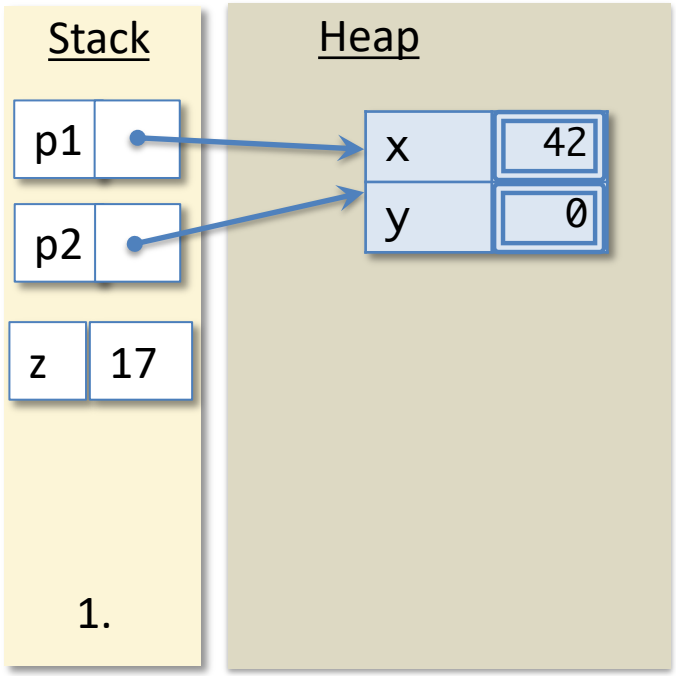
```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  let z = p1.x in
  p2.x <- 42;
  z
```

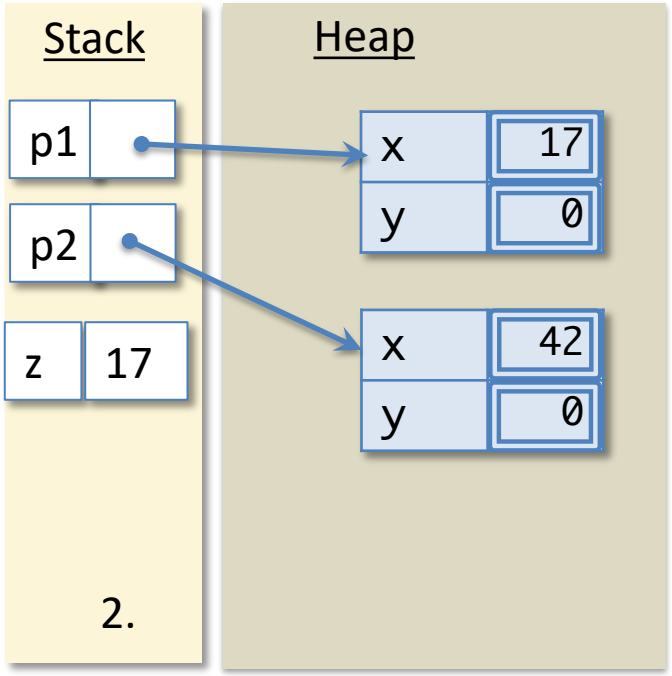
1. 17
2. 42
3. sometimes 17 and sometimes 42
4. f is ill typed

Answer: 1

What do the Stack and Heap look like after simplifying the following code on the workspace?

```
let p1 = {x=0; y=0} in
let p2 = p1 in
p1.x <- 17;
let z = p1.x in
p2.x <- 42;
p1.x
```





Answer: 1

References and Equality

= vs. ==

Reference Equality

- Suppose we have two counters. Are they at the same location?

```
type counter = { mutable count : int }
let c1 : counter = …
let c2 : counter = …
```

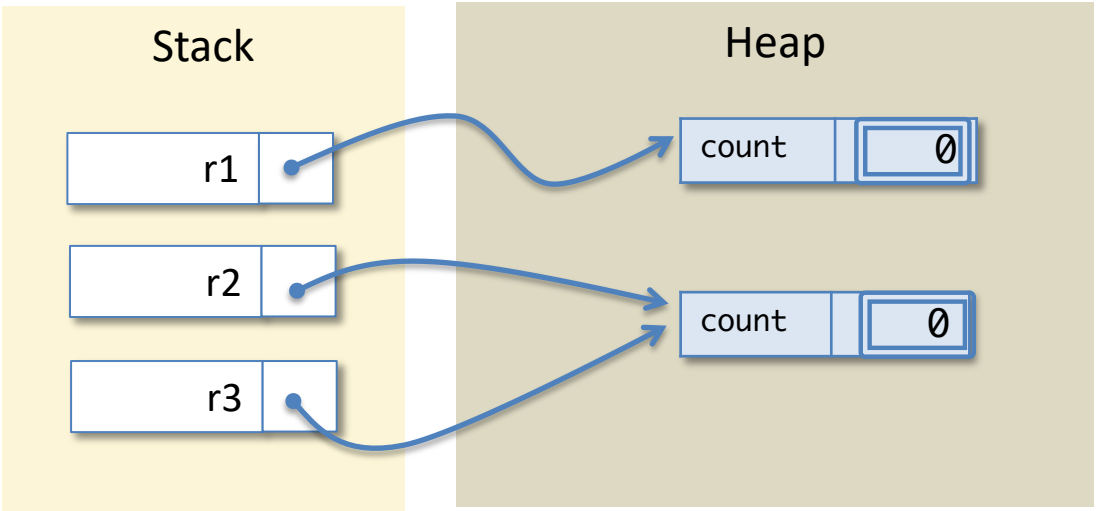
  - We could increment one and see whether the other's value changes.
  - But we could also just test whether the references are **aliases**.

- OCaml uses '==' to mean *reference* equality:
  - two reference values are '==' if they point to the same location in the heap; so:

```
r2 == r3

not (r1 == r2)

r1 = r2
```



Structural vs. Reference Equality

- *Structural (in)equality*: `v1 = v2` `v1 <> v2`
  - recursively traverses over the *structure* of the data, comparing the two values' components for structural equality
  - function values cannot be compared structurally
  - structural equality can go into an infinite loop on cyclic structures
  - appropriate for comparing *immutable* datatypes

- *Reference (in)equality*: `v1 == v2` `v1 != v2`
  - Only looks at where the two references point in the heap
  - function values are only equal to themselves
  - even if v1 = v2, we may not have v1 == v2
  - appropriate for comparing *mutable* datatypes

14: What is the result of evaluating the following expression?

0

true

0%

false

0%

runtime error

0%

compile-time error

0%

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = p1 in

p1 = p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

14: What is the result of evaluating the following expression?

0

true

0%

false

0%

runtime error

0%

compile-time error

0%

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = p1 in

p1 == p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = { x = 0; y = 0 } in

p1 == p2
```

1. true
2. false

Answer: false

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = { x = 0; y = 0 } in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 = l2
```

1. true
2. false

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = p1 in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 == l2
```

1. true
2. false

Answer: false

ASM: Lists and datatypes

Tracking the space usage of *immutable* data structures

Simplification

Workspace

```
1::2::3::[]
```

Stack

Heap

For uniformity, we'll pretend lists are declared like this:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

Simplification

Workspace

```
Cons (1,Cons (2,Cons (3,Nil)))
```

Stack

Heap

For uniformity, we'll pretend lists are declared like this:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

Simplification

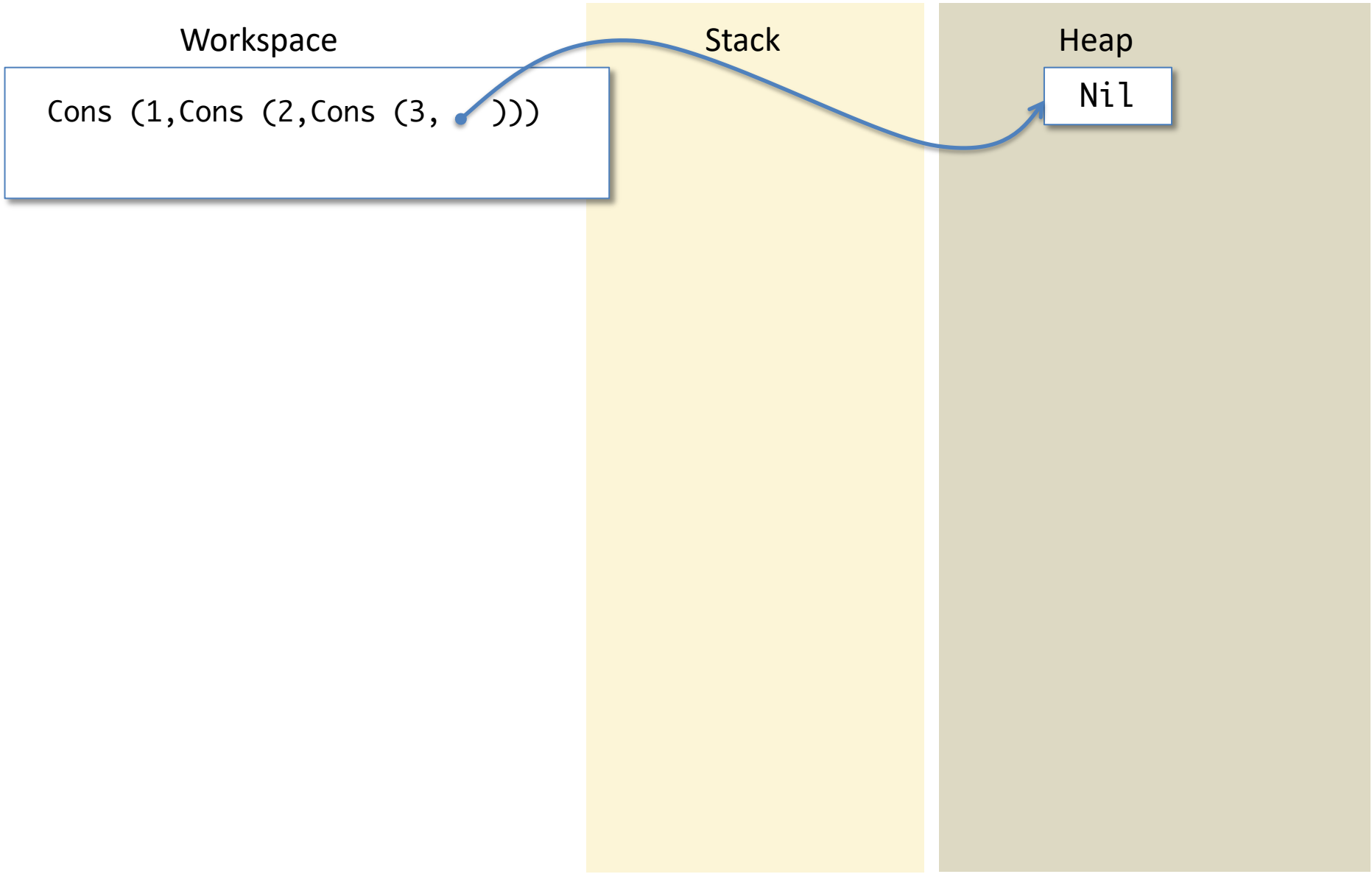
Workspace

```
Cons (1,Cons (2,Cons (3,Nil)))
```

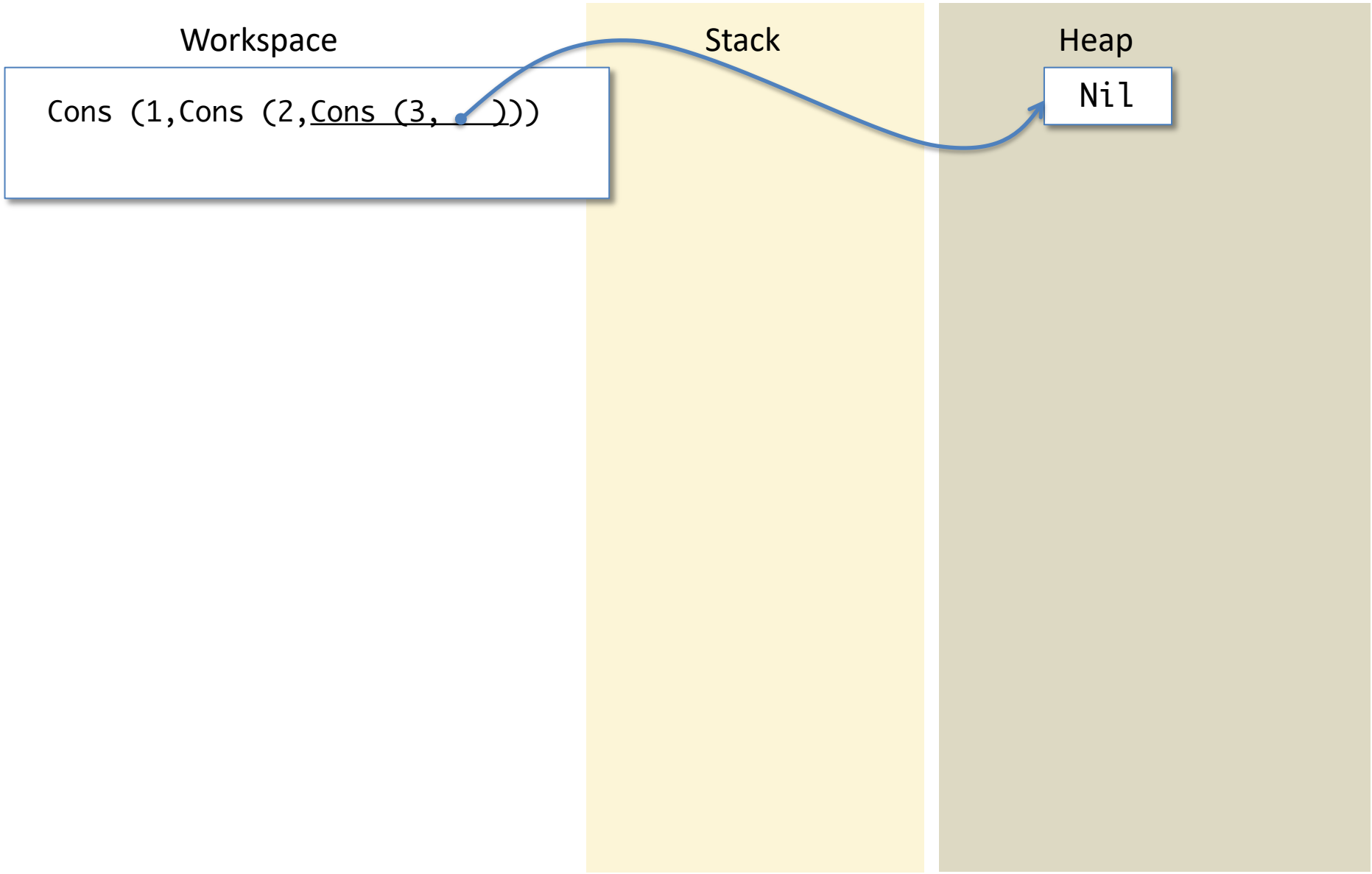
Stack

Heap

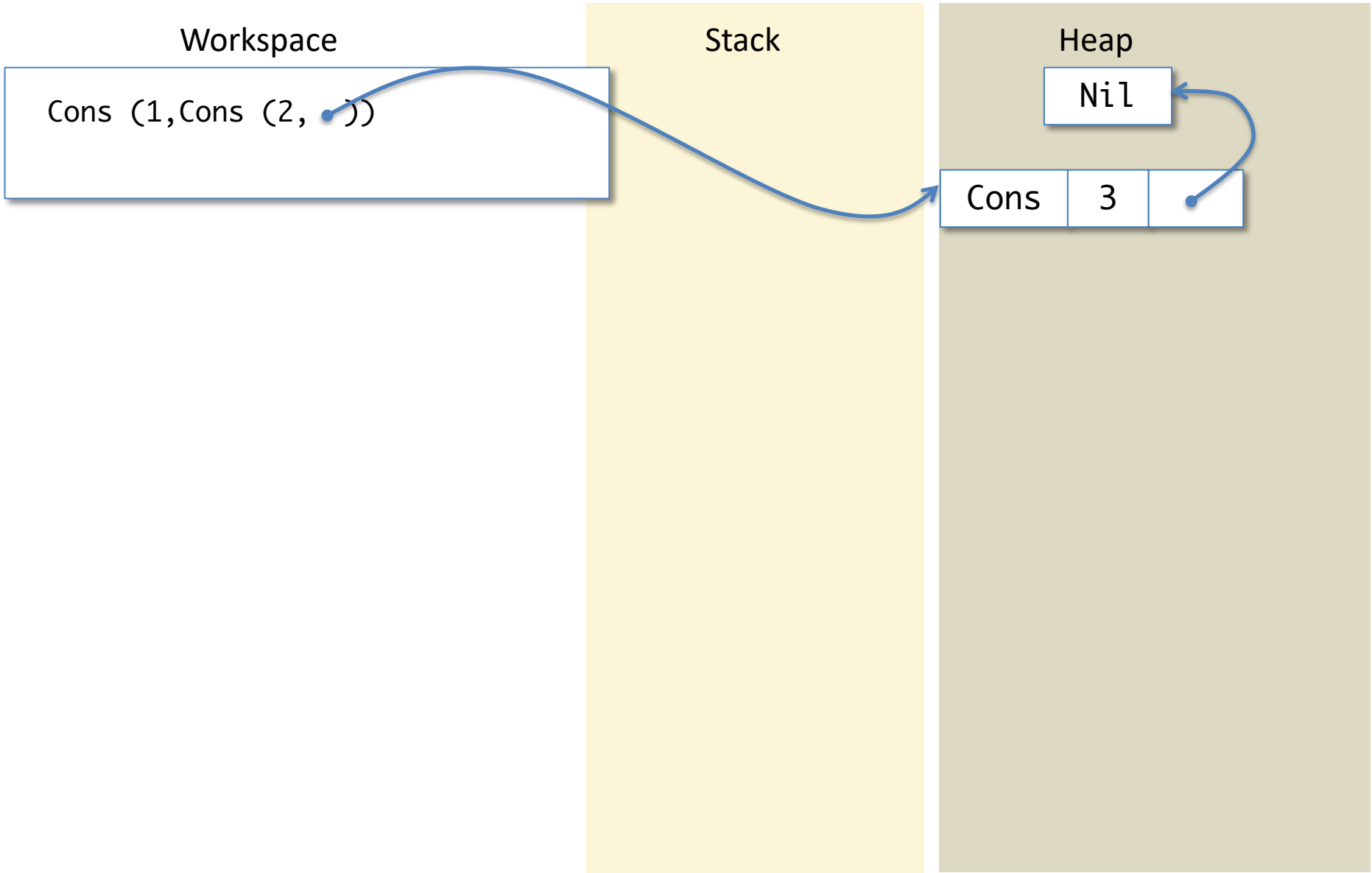
Simplification



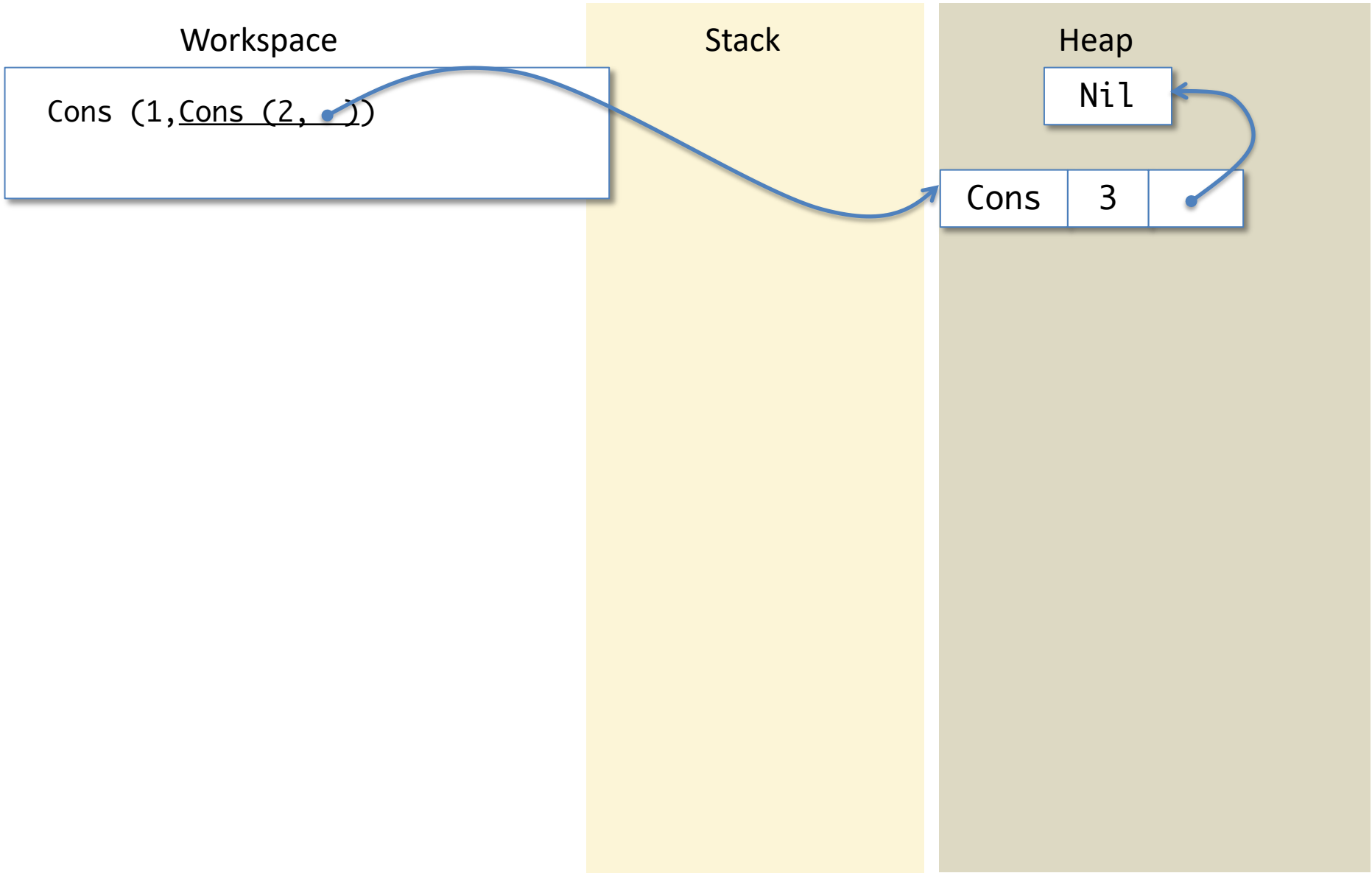
Simplification



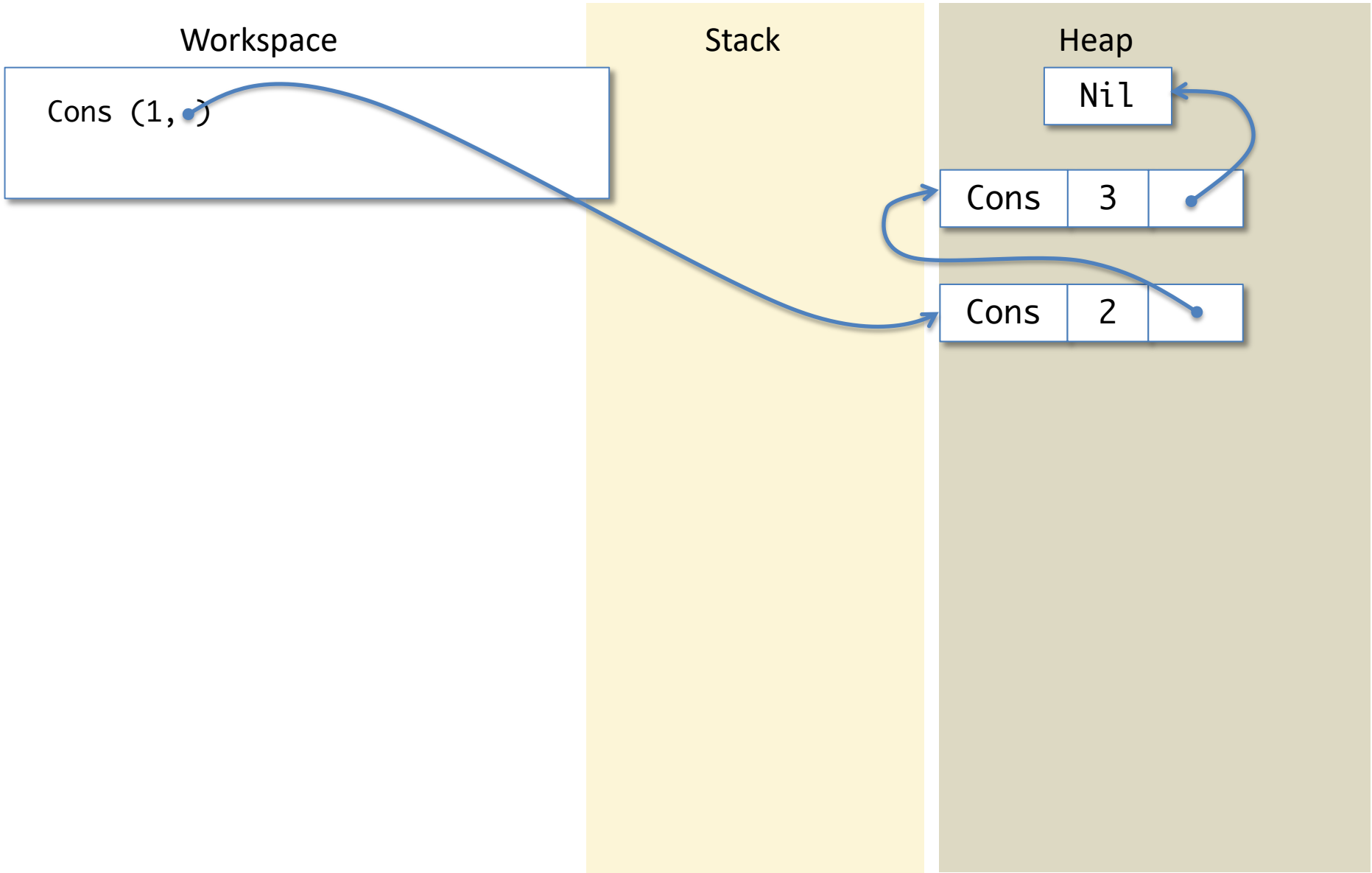
Simplification



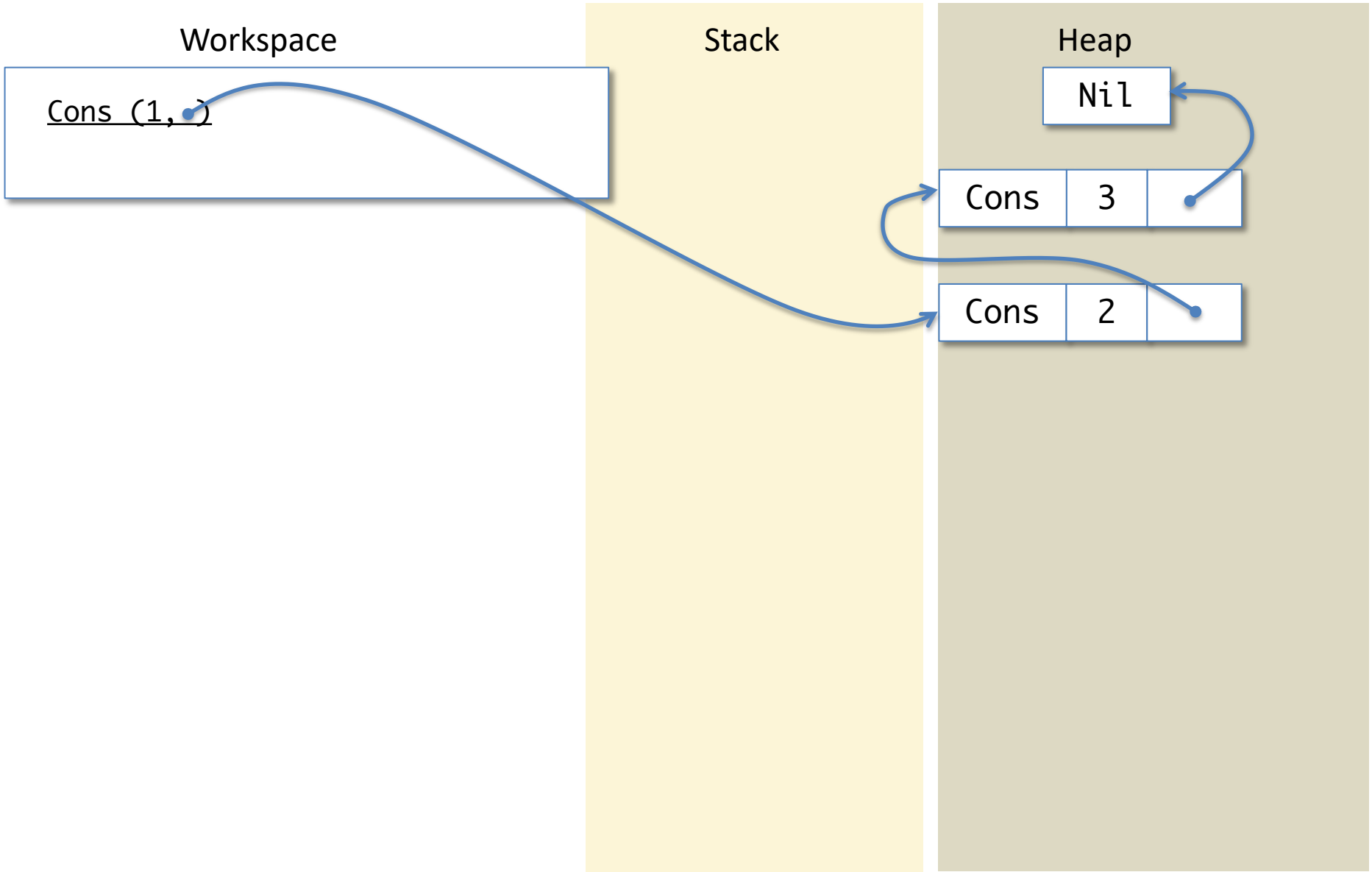
Simplification



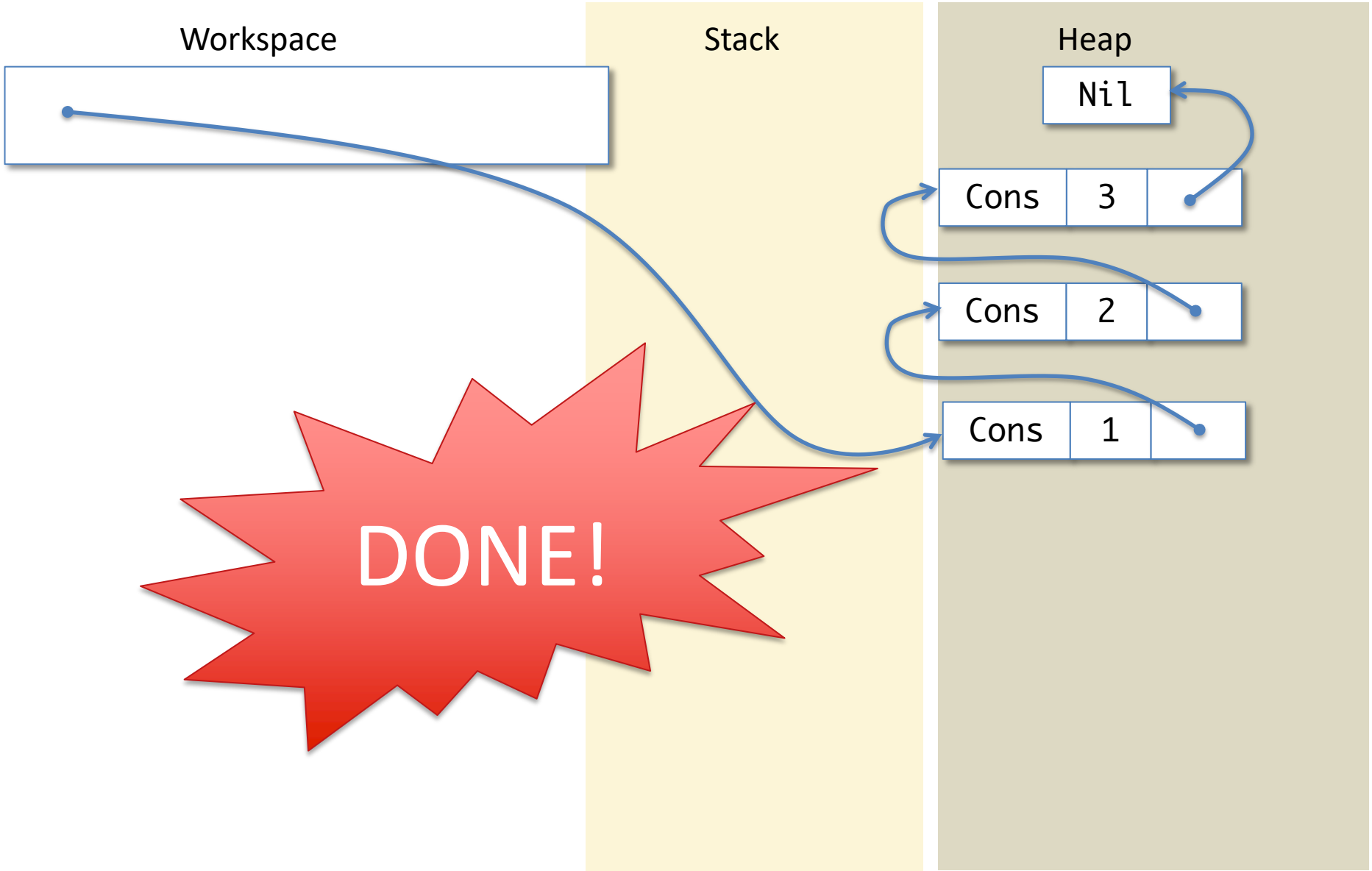
Simplification



Simplification



Simplification



15: Simplifying code on the ASM

0

1

0%

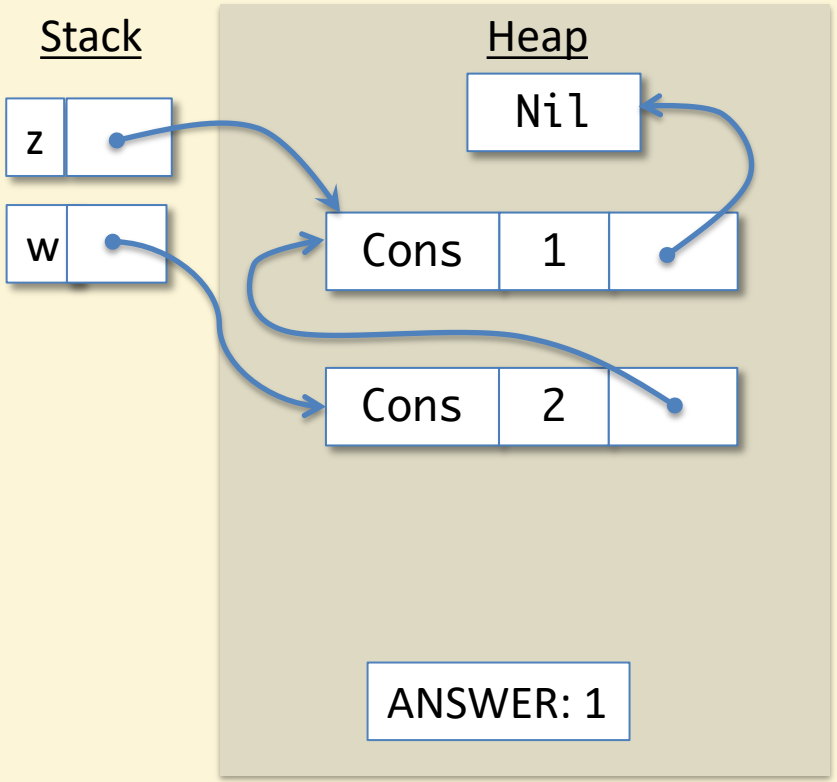
2

0%

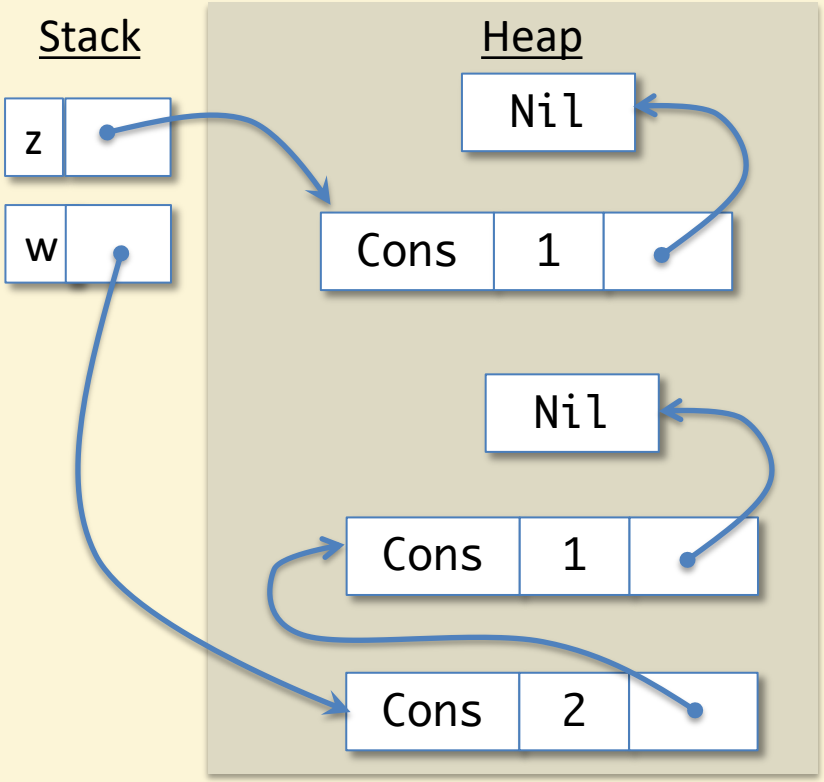
What do the Stack and Heap look like after simplifying the following code on the workspace?

```
let z = Cons (1, Nil) in
let w = Cons (2, z) in
    w
```

1.



2.



An Optimization

- Datatype constructors that carry no extra information can be treated as “small” values.
- Examples:

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list
```

```
type ‘a option =
| None
| Some of ‘a
```

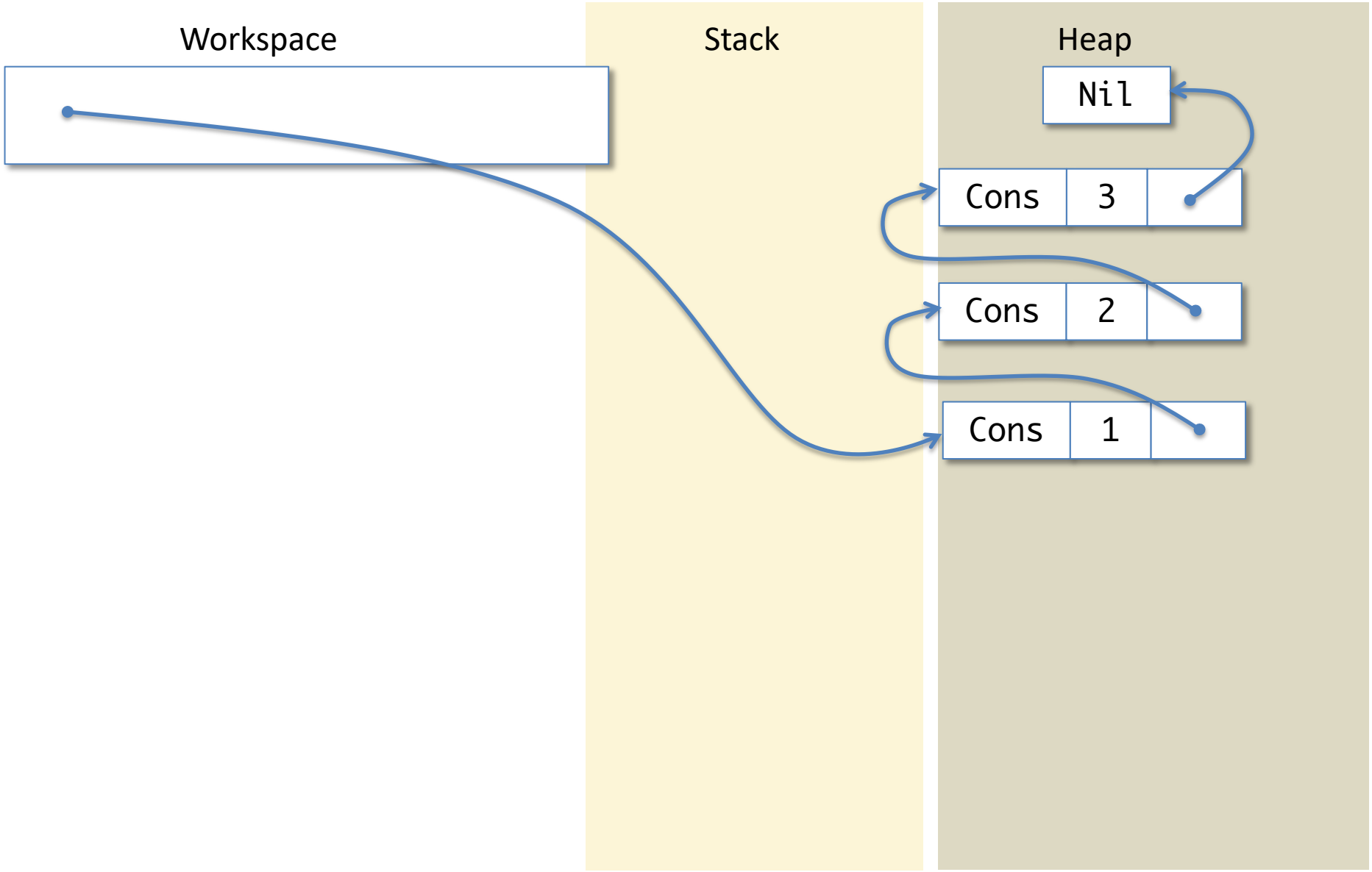
```
type ‘a tree =
| Empty
| Node of ‘a tree * ‘a * ‘a tree
```

- They can be placed directly in the stack.
- They don’t require a reference in the heap.

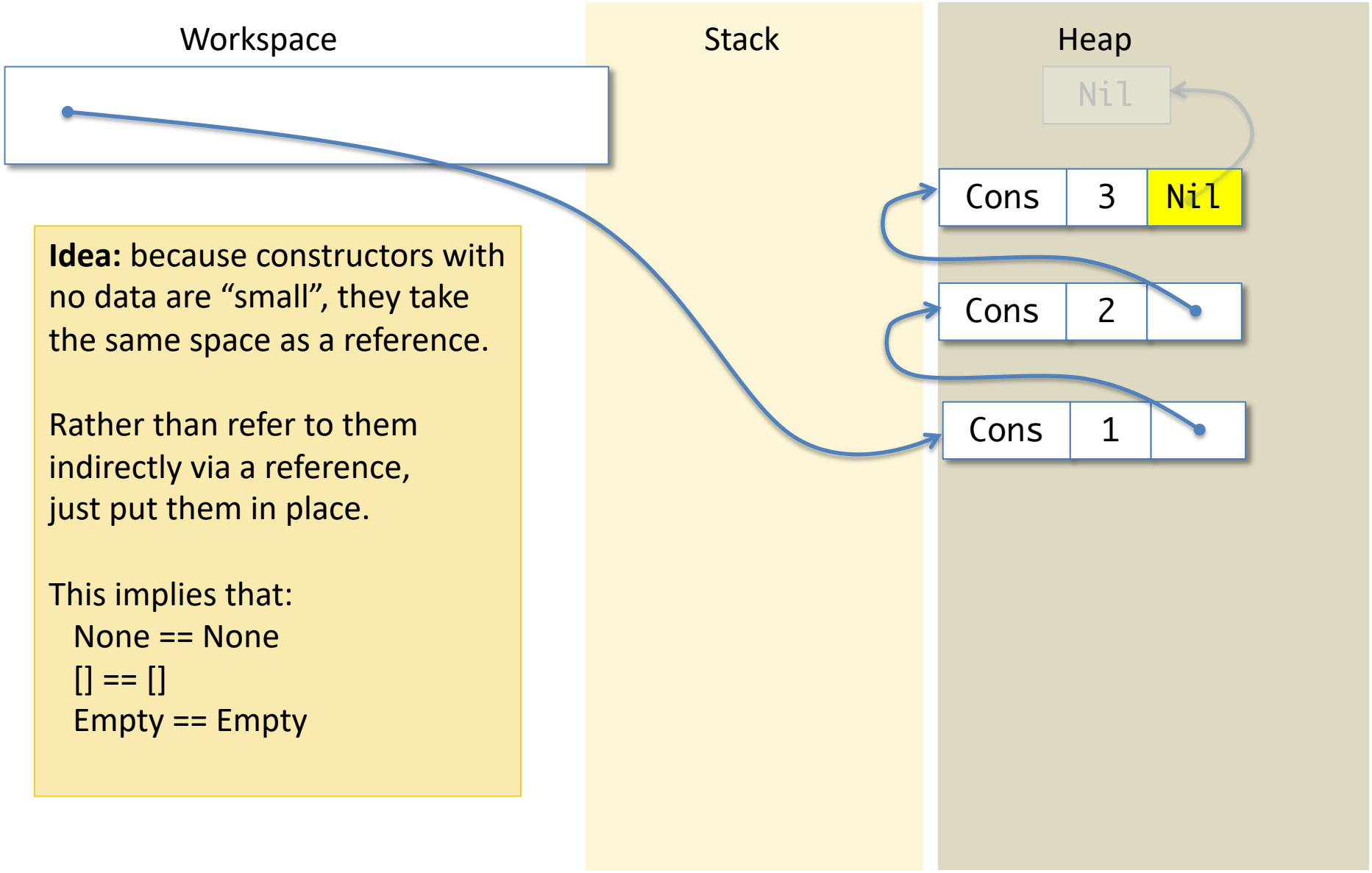
} Saves space!

- N.b.: This optimization affects reference equality.

Example Optimization



Example Optimization



Idea: because constructors with no data are "small", they take the same space as a reference.

Rather than refer to them indirectly via a reference, just put them in place.

This implies that:

None == None

[] == []

Empty == Empty

ASM: functions

Tracking the space usage of function calls

Function Simplification

Workspace

```
let add1 (x : int) : int =
  x + 1 in
add1 (add1 0)
```

Stack

Heap

Function Simplification

Workspace

```
let add1 (x : int) : int =
  x + 1 in
add1 (add1 0)
```

Stack

Heap

First step: replace declaration of add1 with more primitive version

Function Simplification

Workspace

```
let add1 : int ->  int =
  fun (x:int) -> x + 1 in
add1 (add1 0)
```

Stack

Heap

Function Simplification

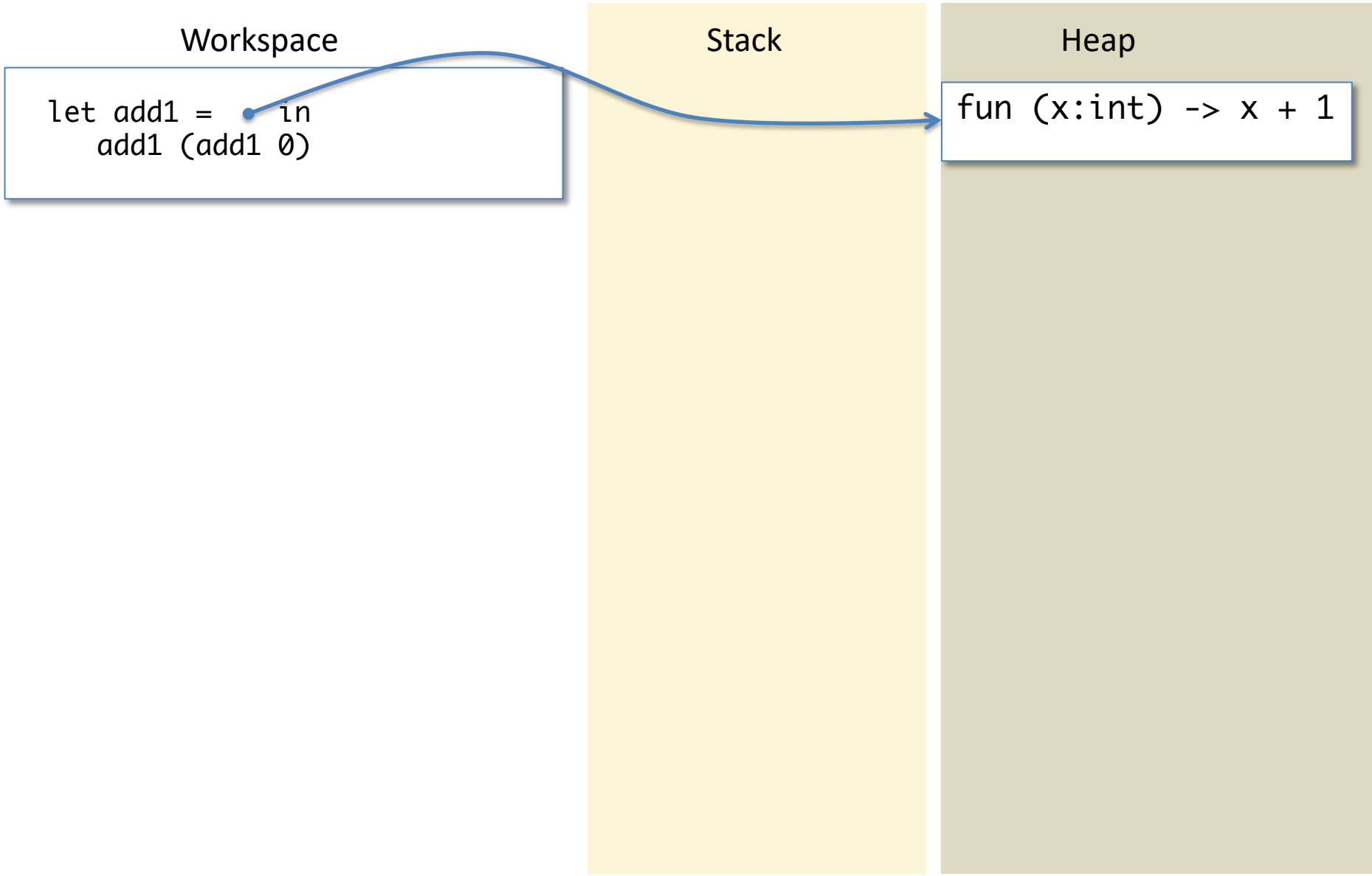
Workspace

```
let add1 : int ->  int =
  fun (x:int) -> x + 1 in
add1 (add1 0)
```

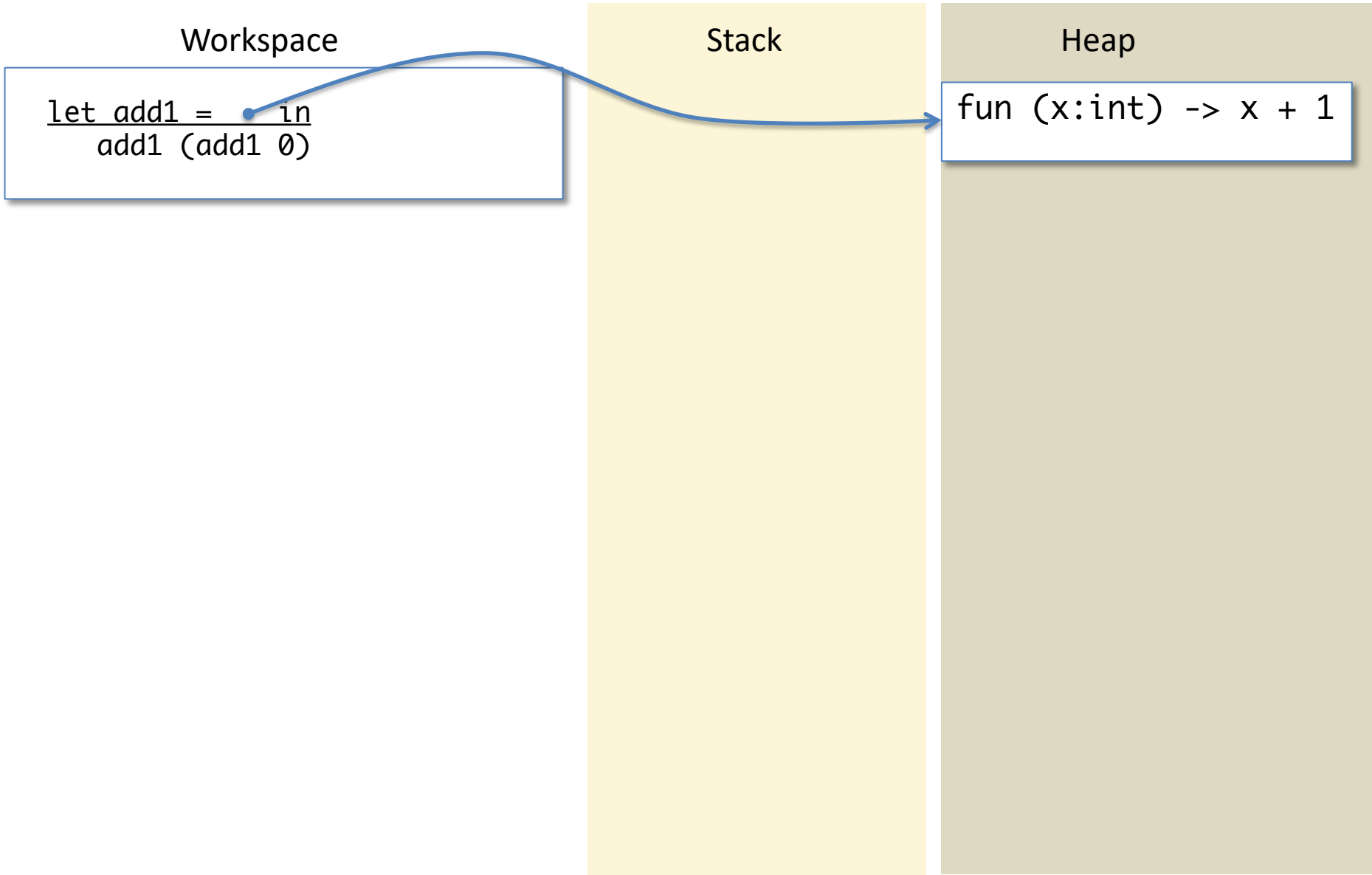
Stack

Heap

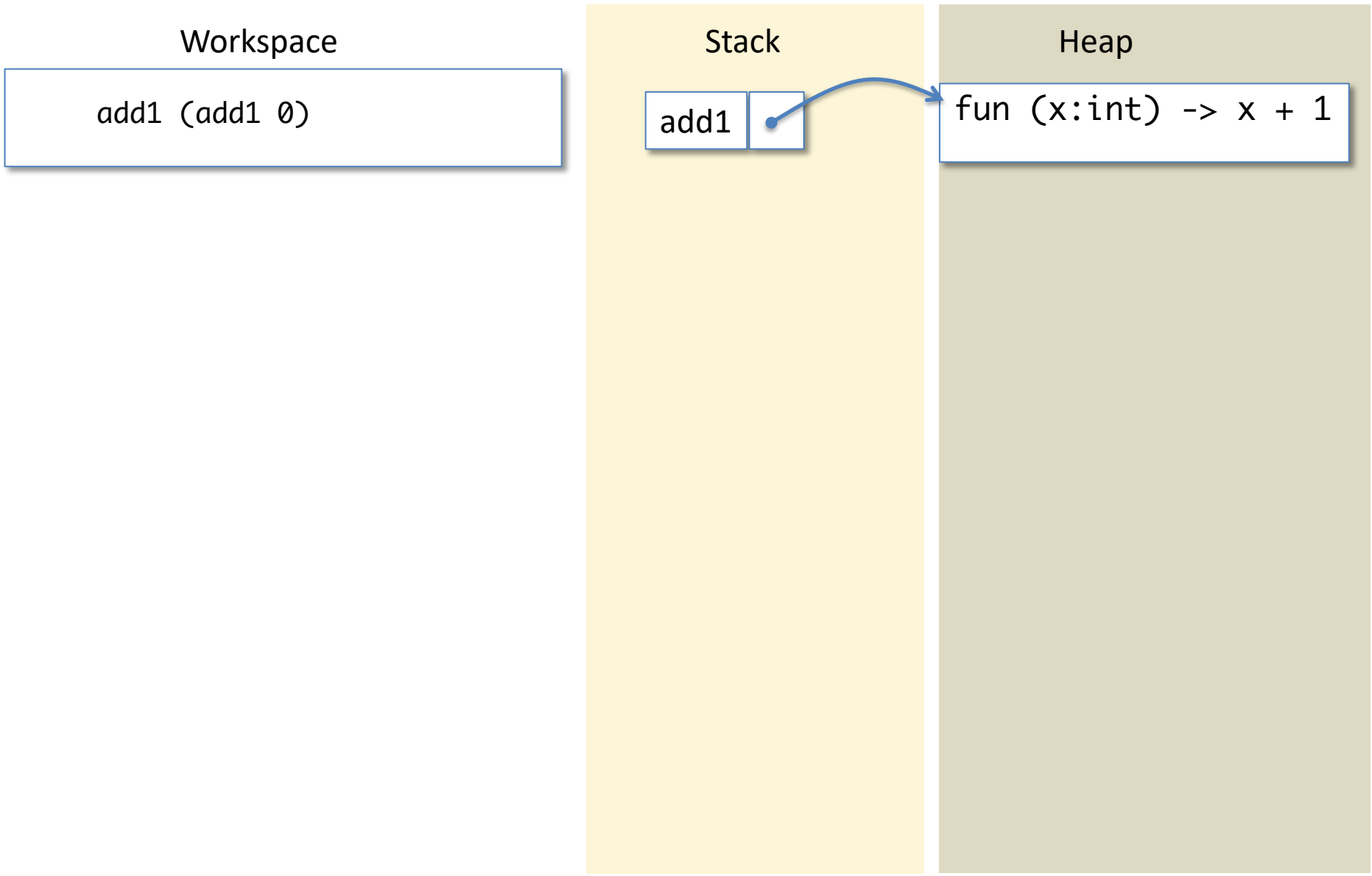
Function Simplification



Function Simplification



Function Simplification

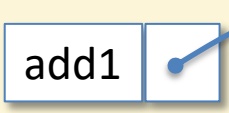


Function Simplification

Workspace

```
add1 (add1 0)
```

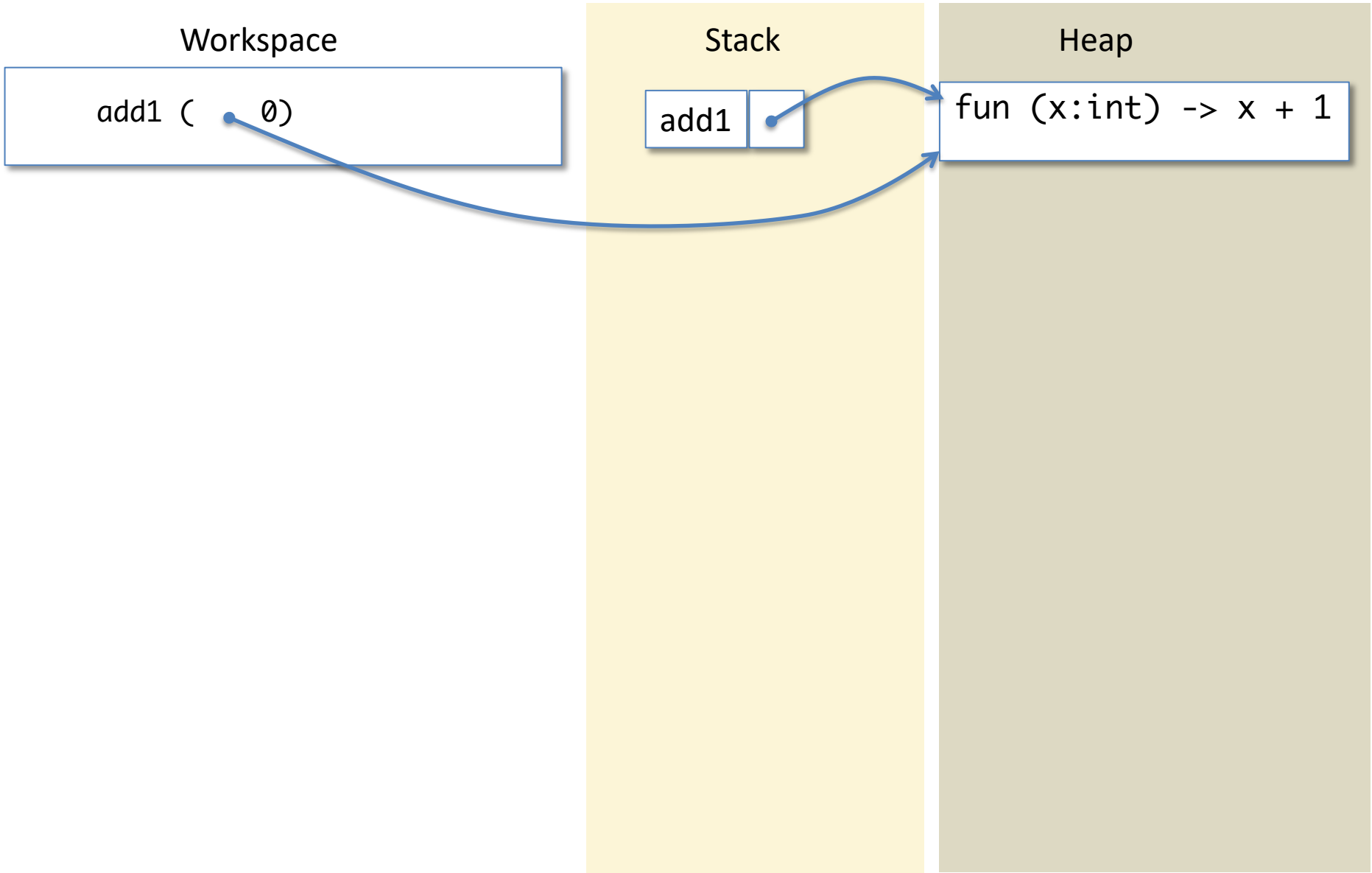
Stack



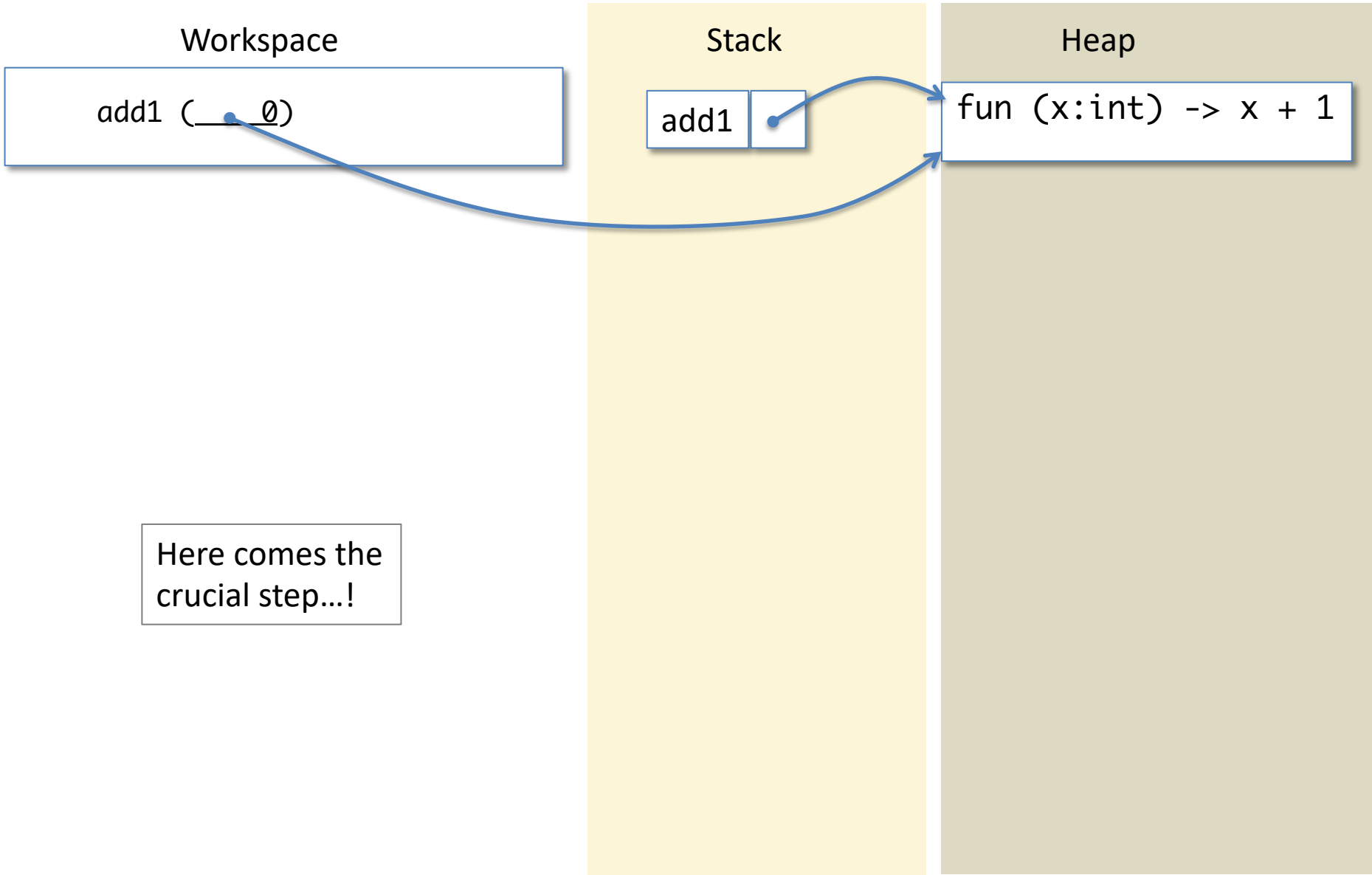
Heap

```
fun (x:int) -> x + 1
```

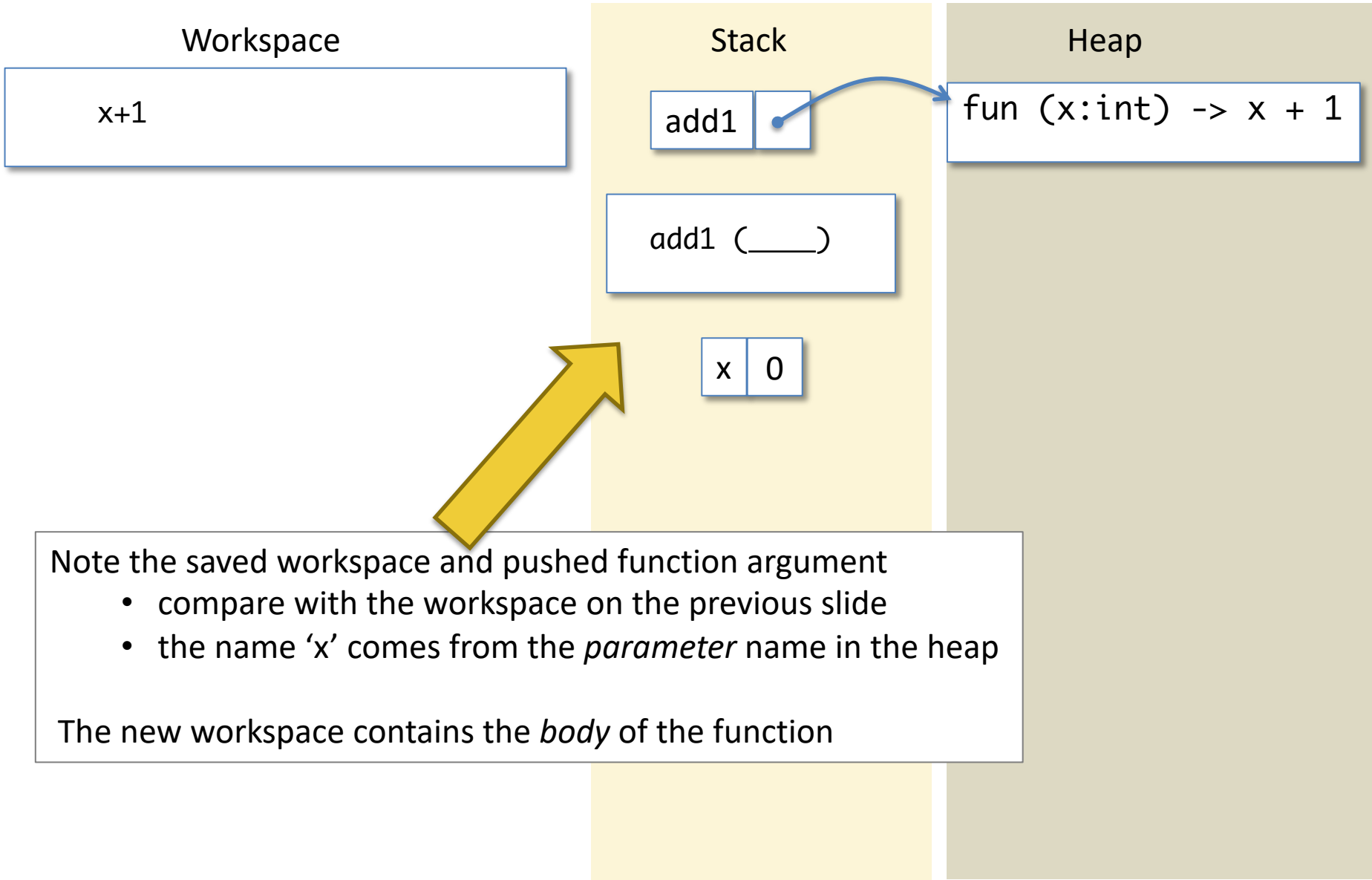
Function Simplification



Function Simplification



Do the Call, Saving the Workspace

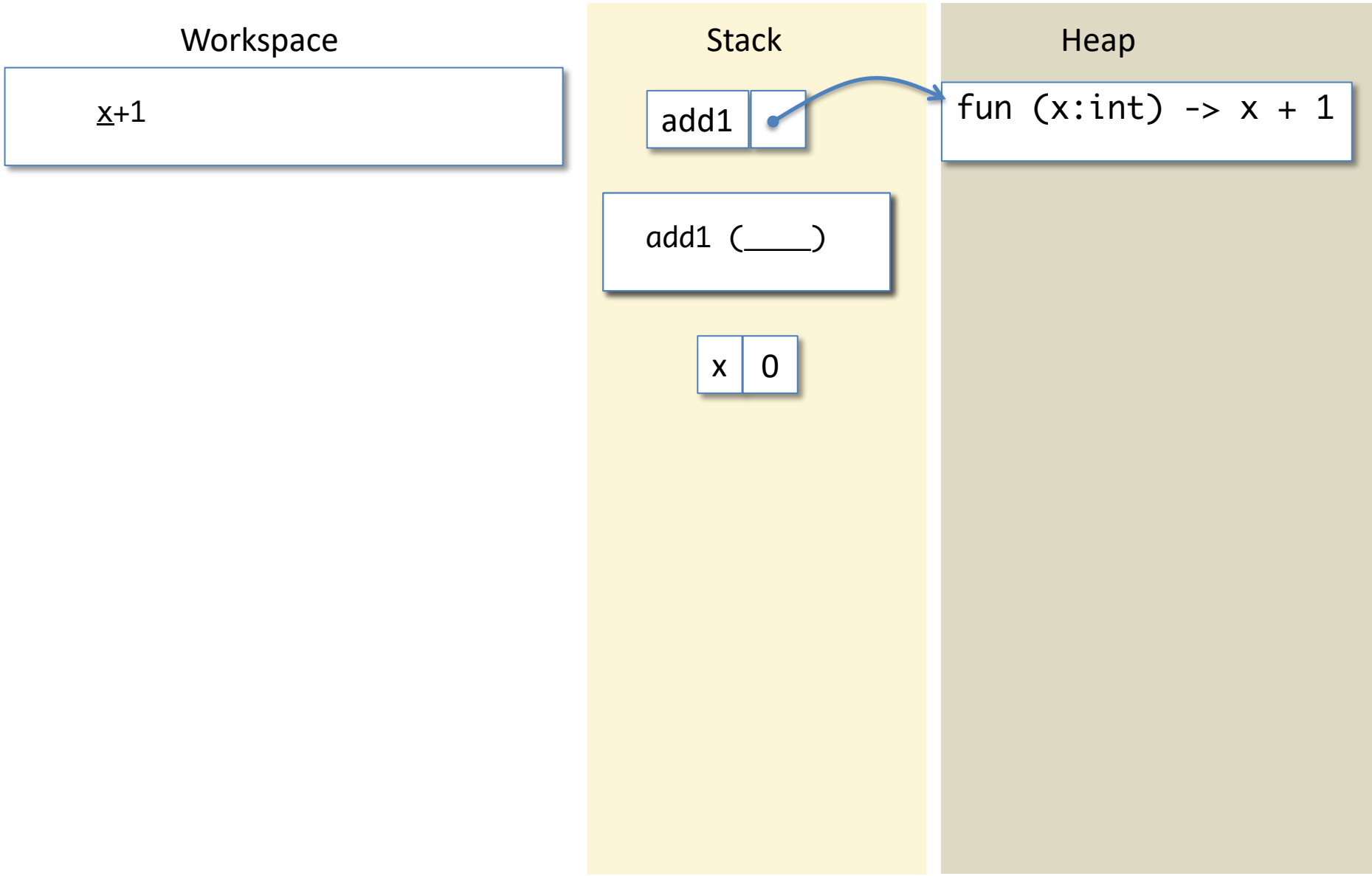


Note the saved workspace and pushed function argument

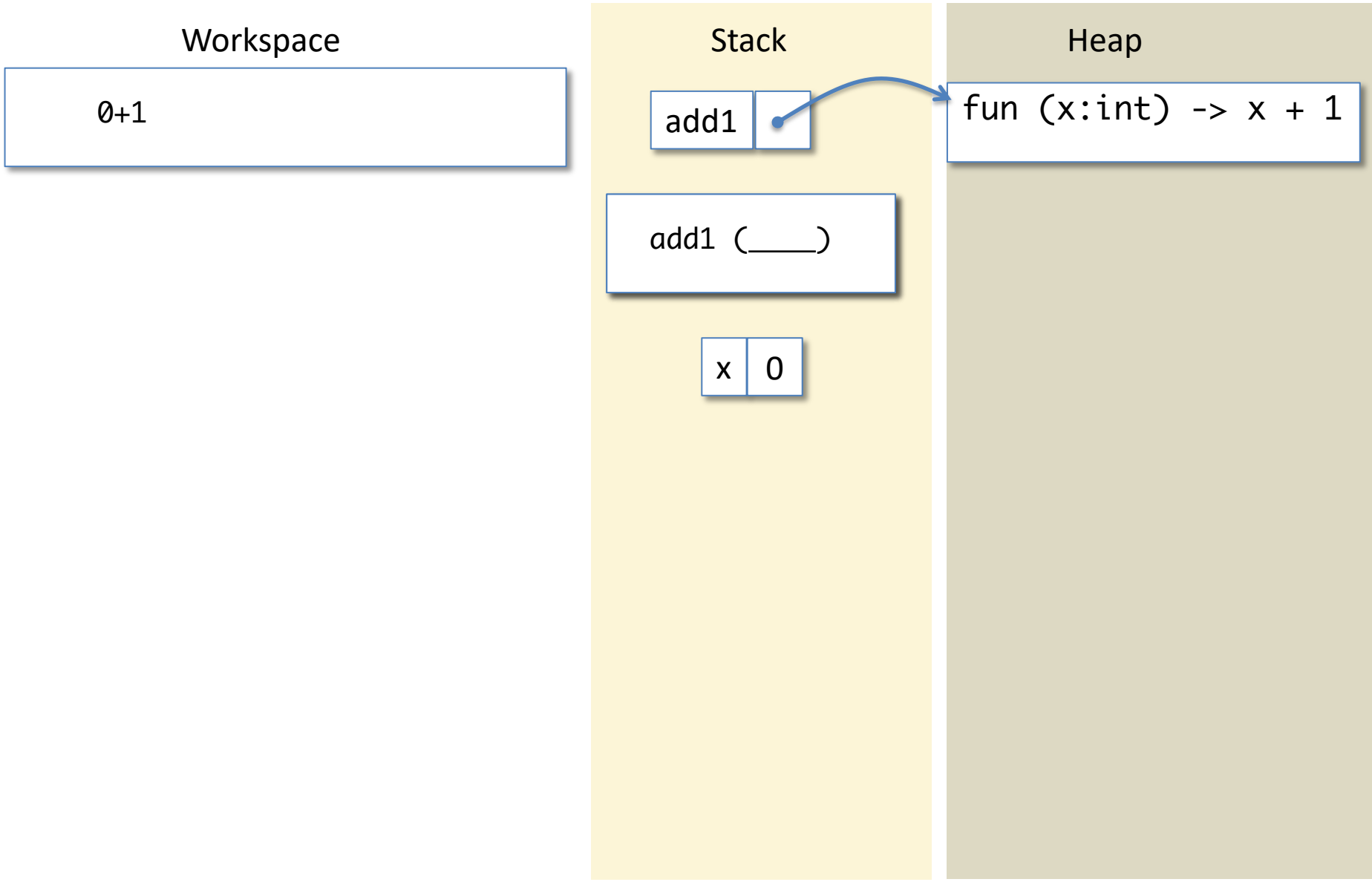
- compare with the workspace on the previous slide
- the name ‘x’ comes from the *parameter* name in the heap

The new workspace contains the *body* of the function

Function Simplification



Function Simplification



Function Simplification

Workspace

```
0+1
```

Stack

add1

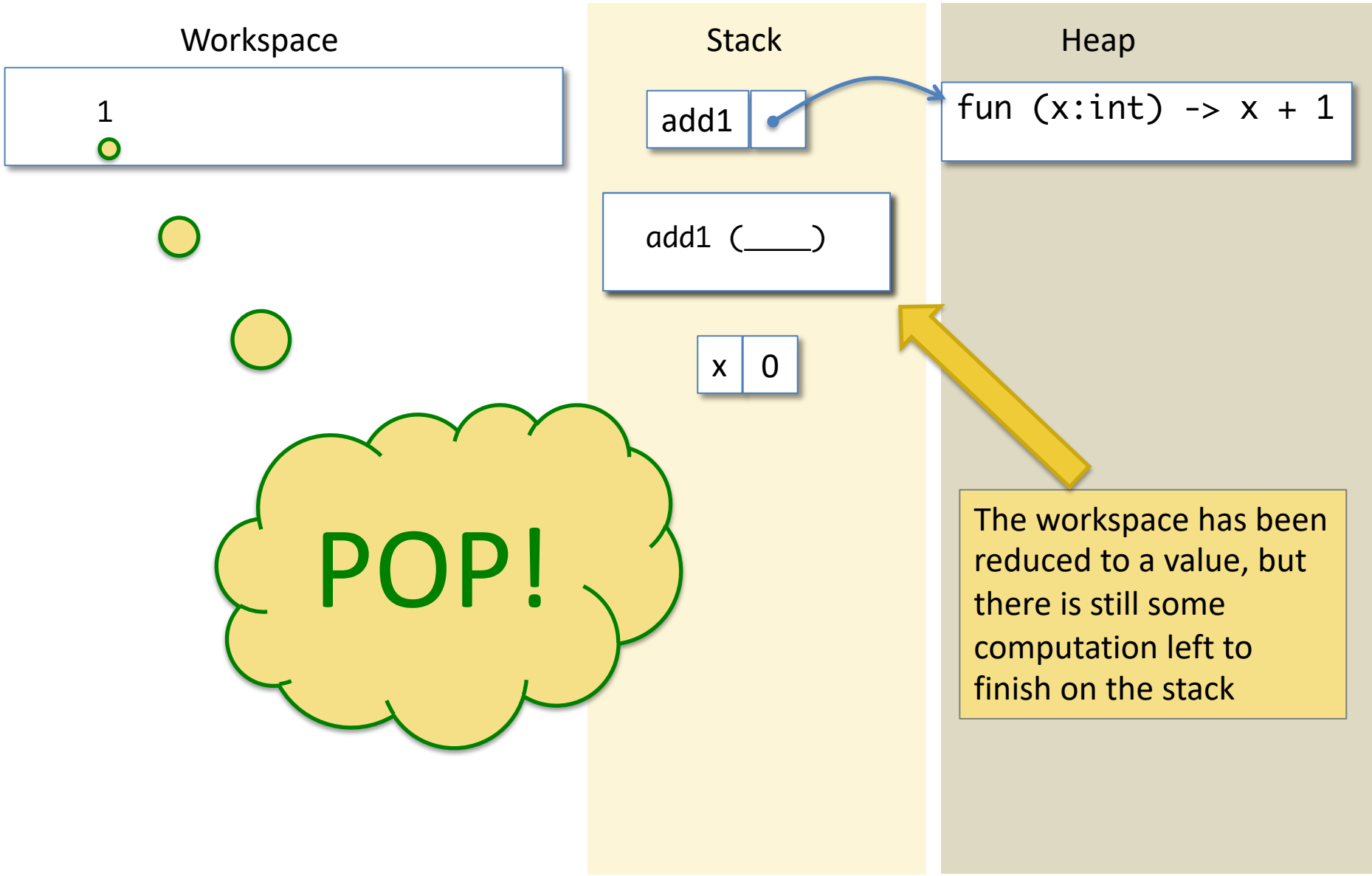
```
add1 (____)
```

x	0

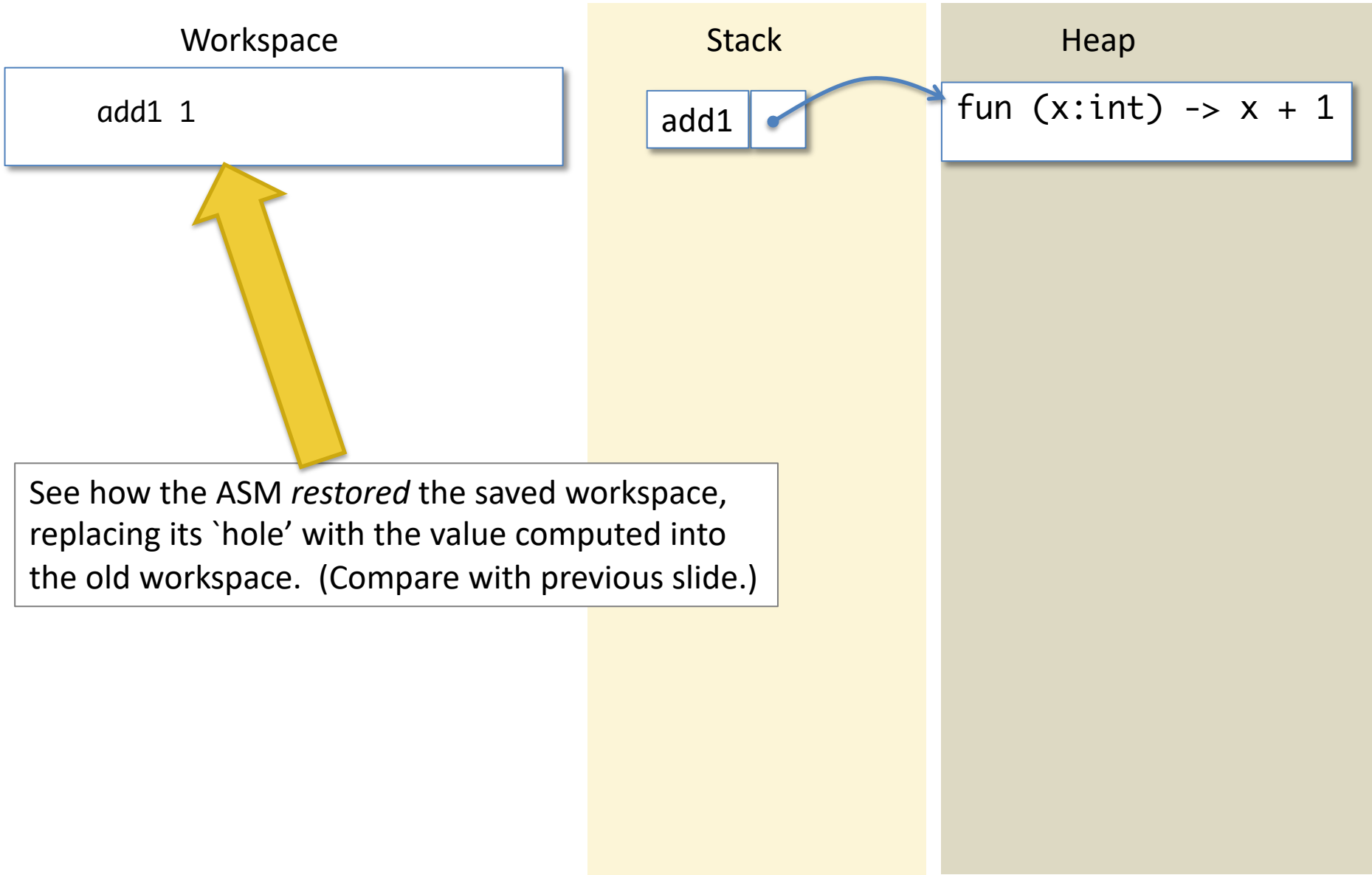
Heap

```
fun (x:int) -> x + 1
```

Function Simplification



Function Simplification

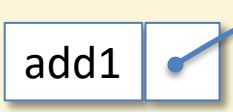


Function Simplification

Workspace

add1 1

Stack

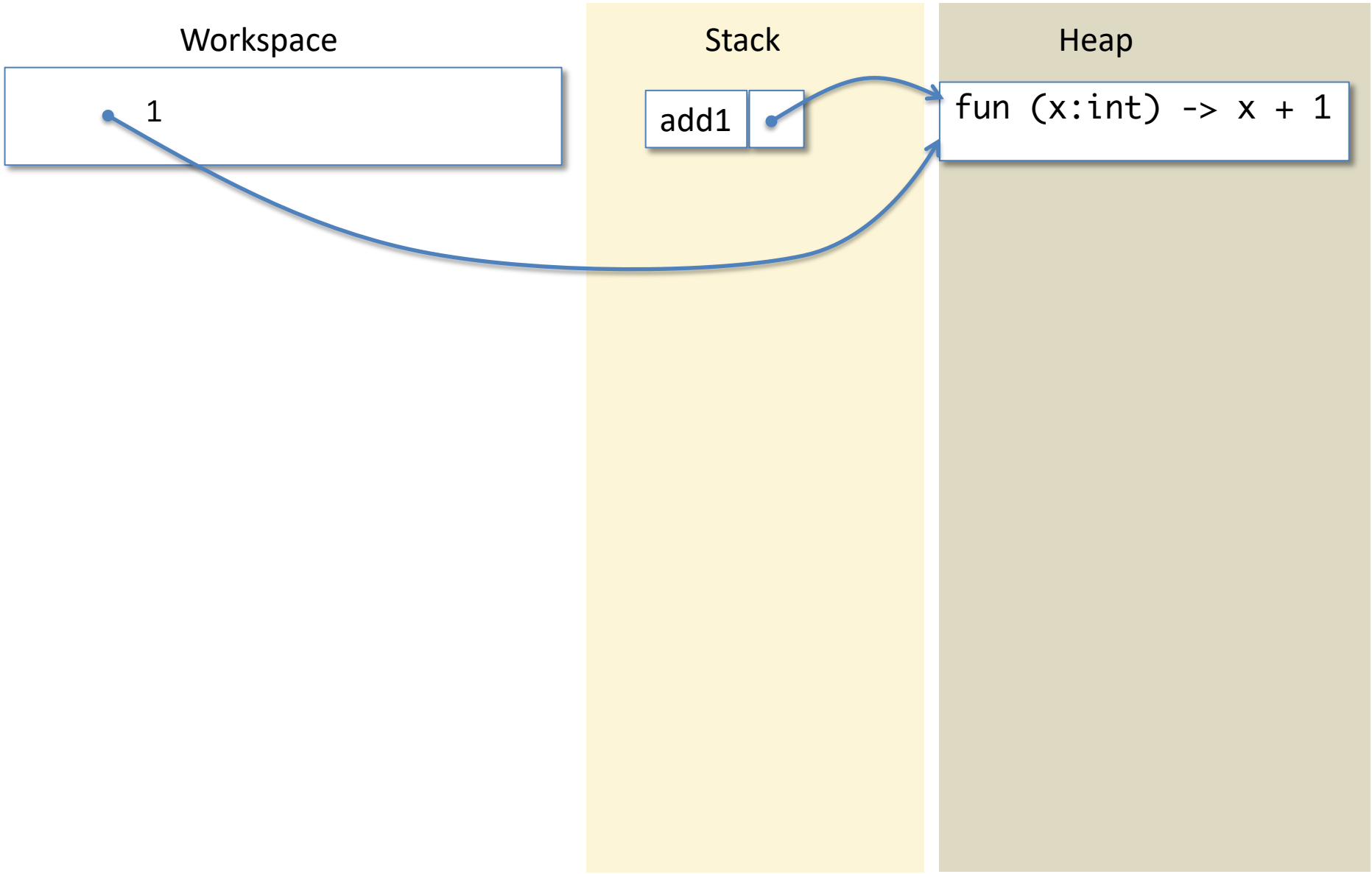


Heap

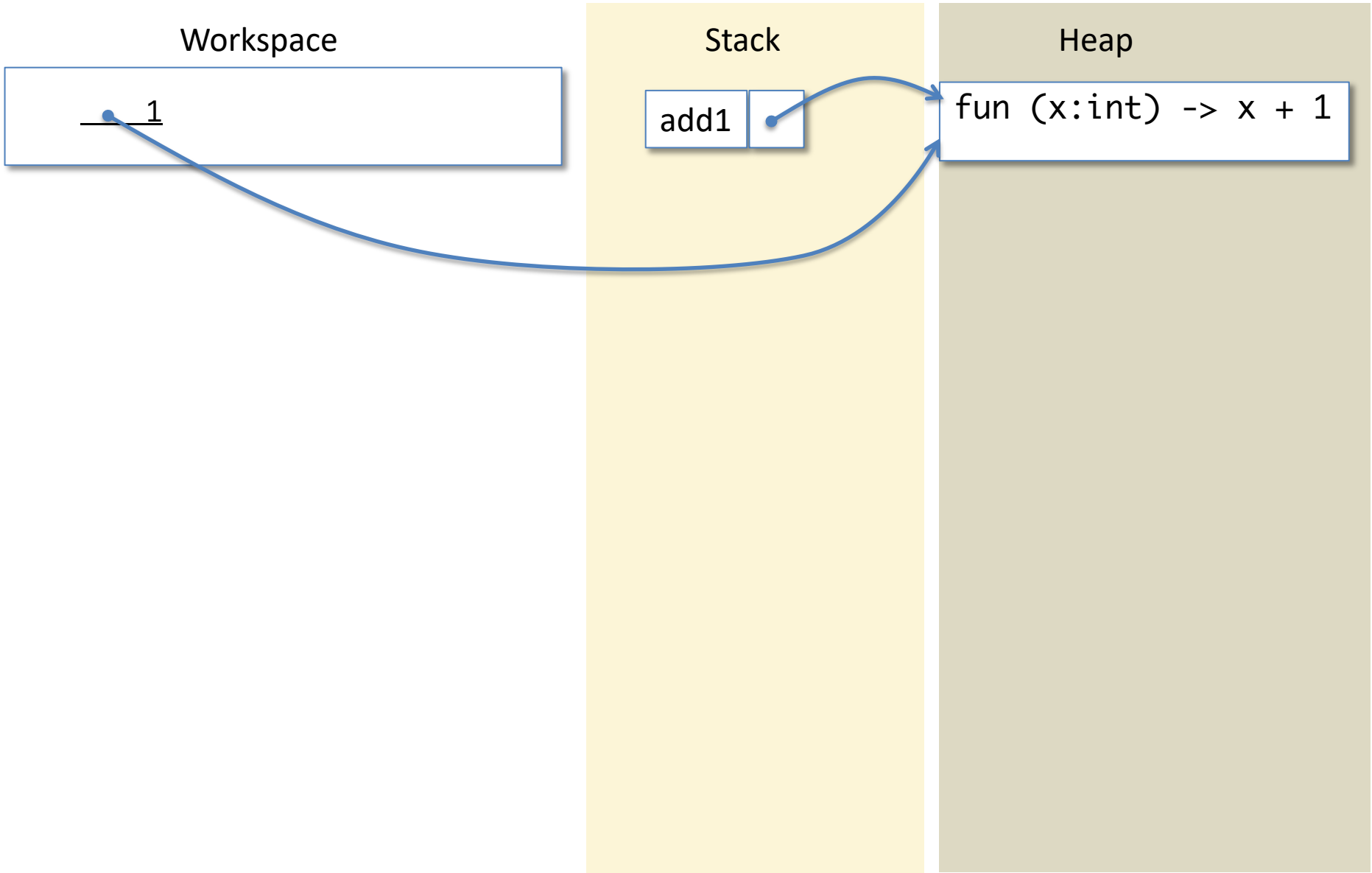
```
fun (x:int) -> x + 1
```

Now we have to do it all over again for the second invocation of add1…

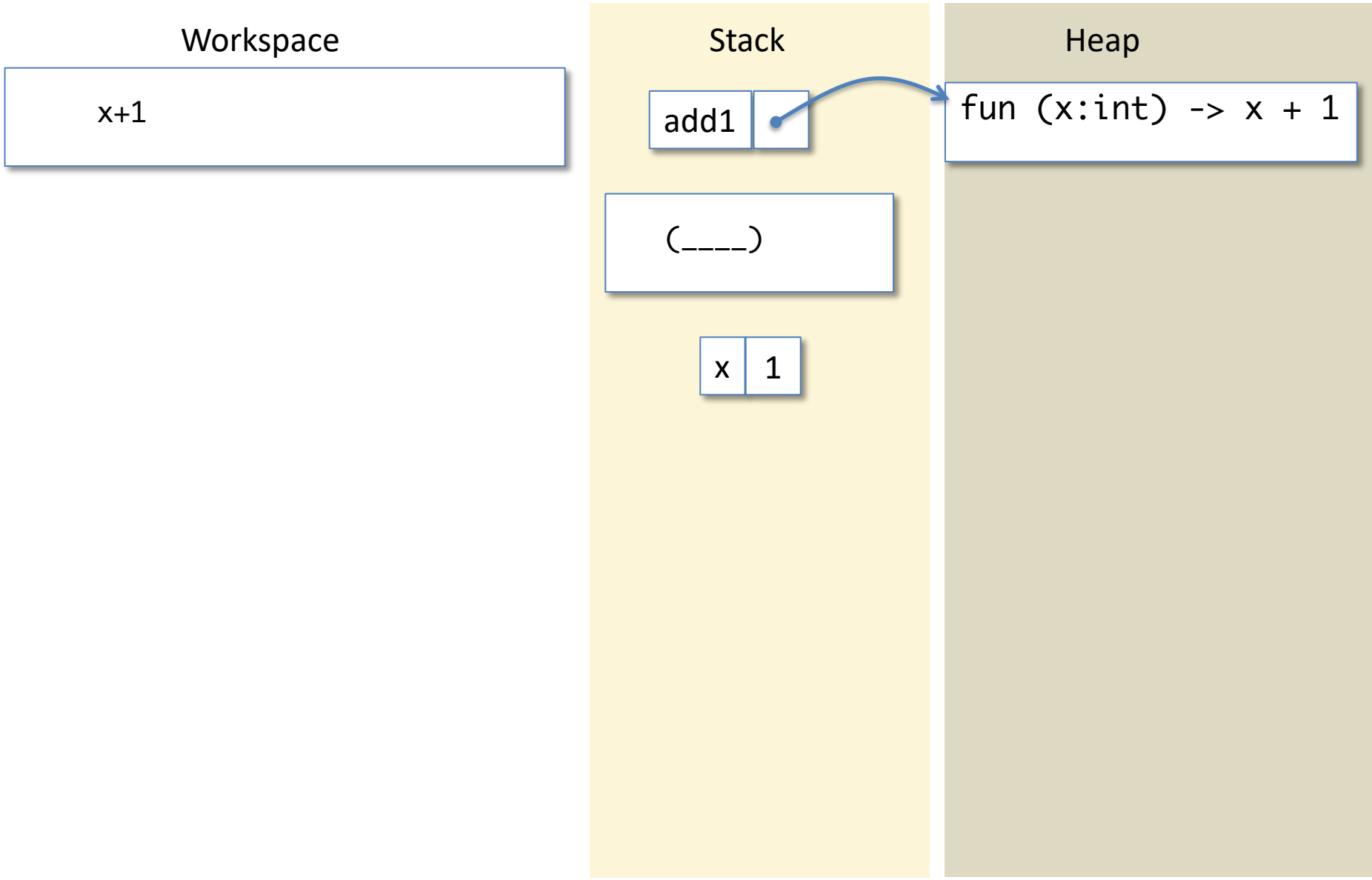
Function Simplification



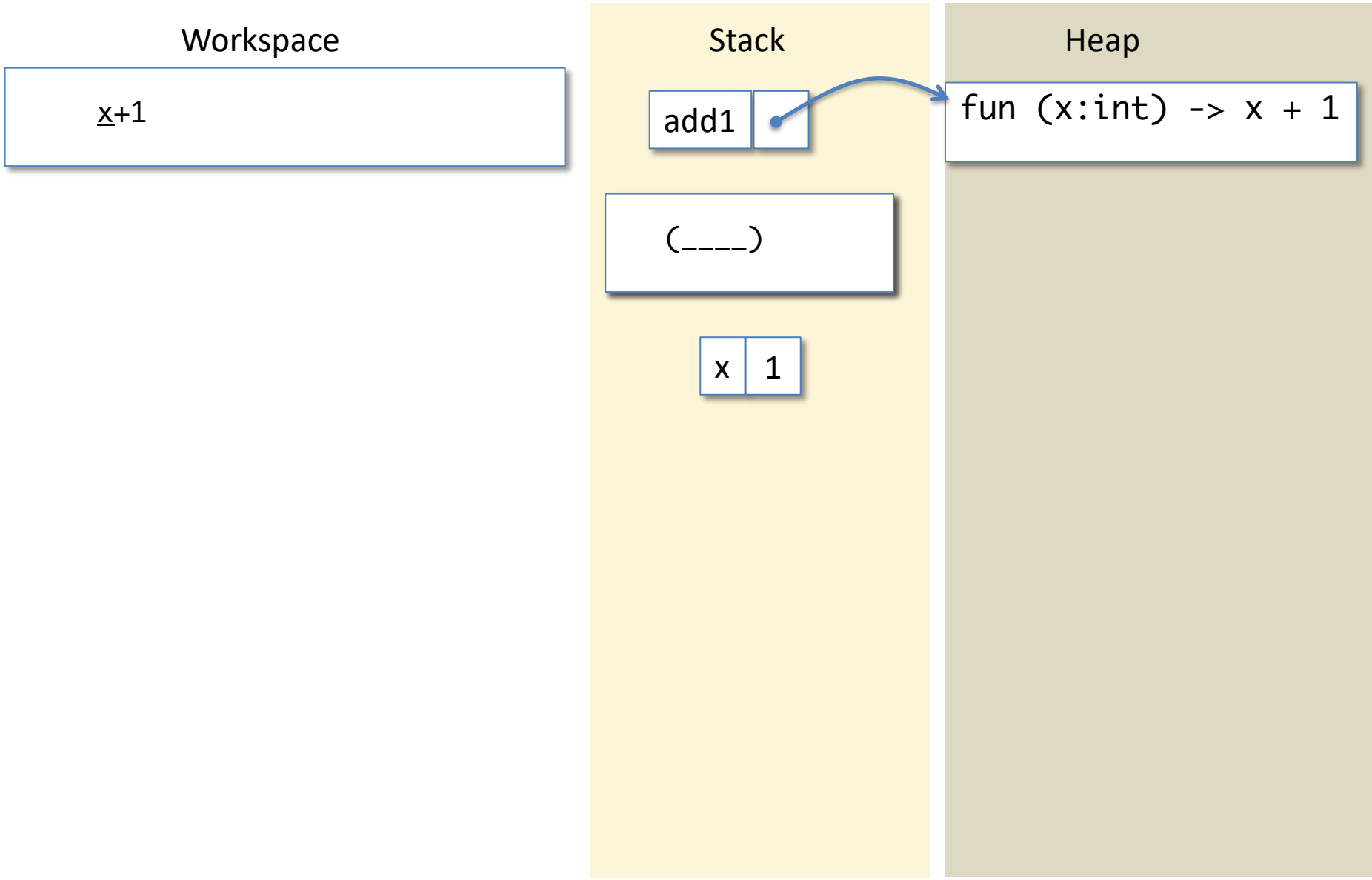
Function Simplification



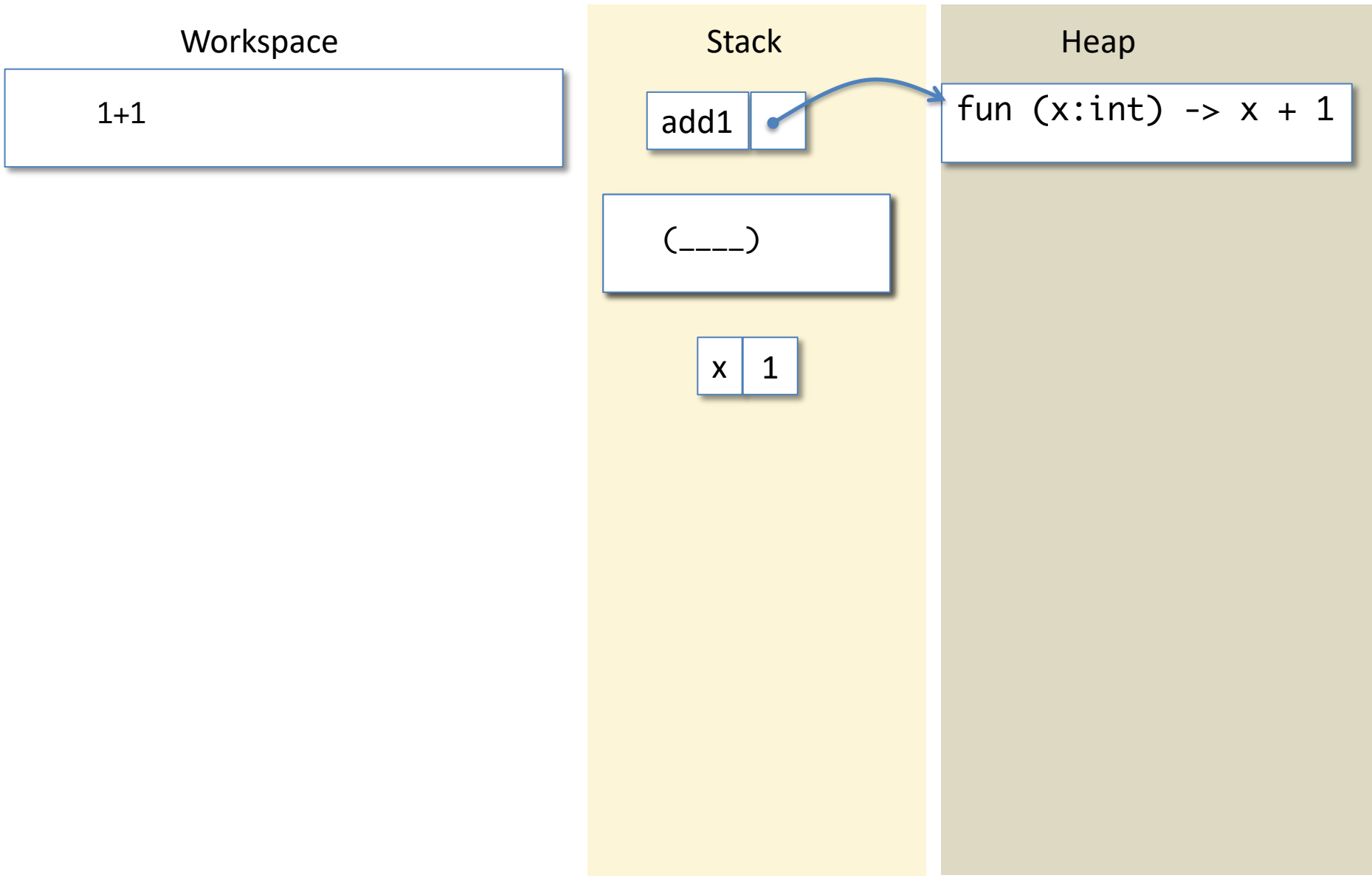
Function Simplification



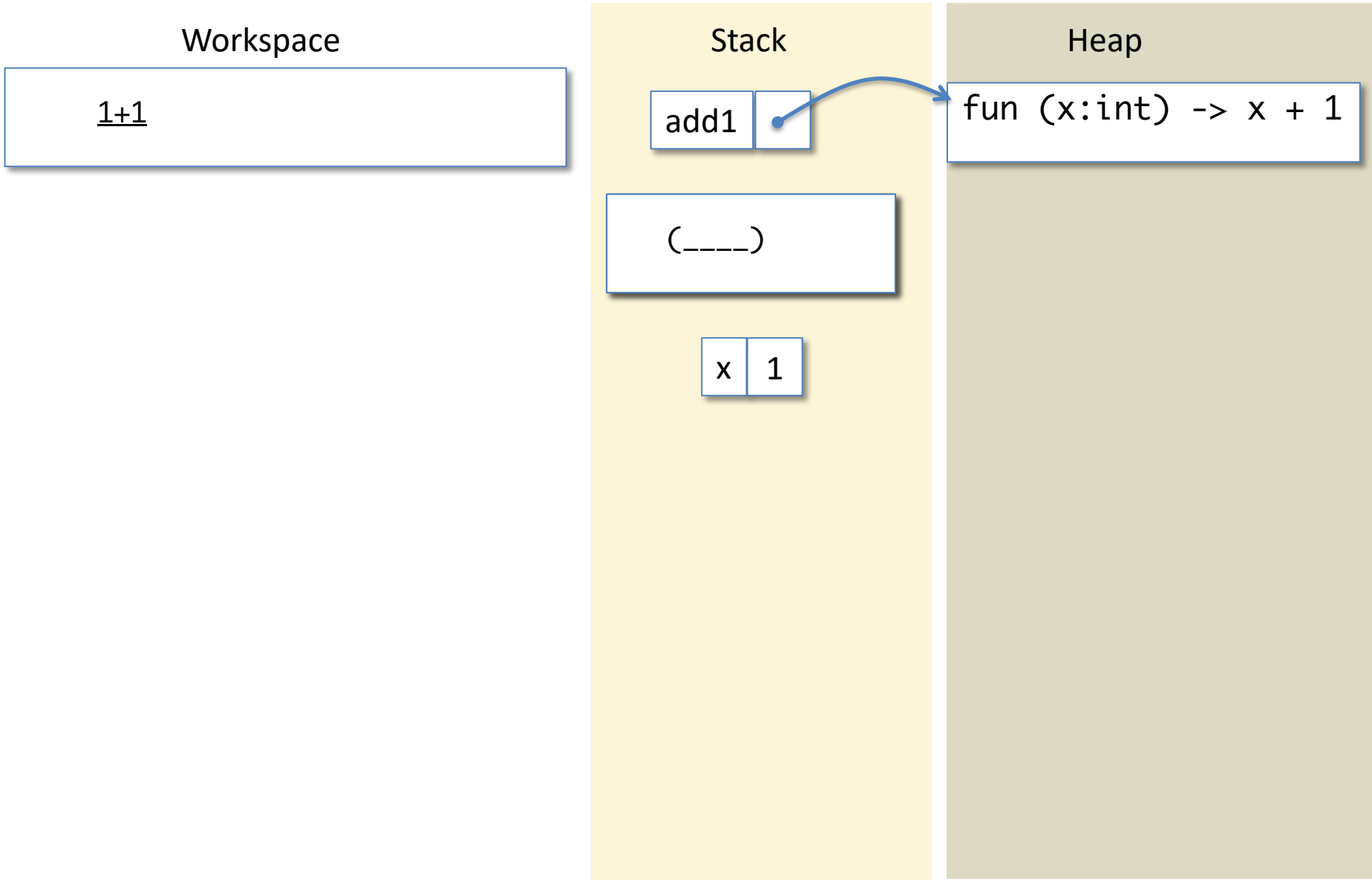
Function Simplification



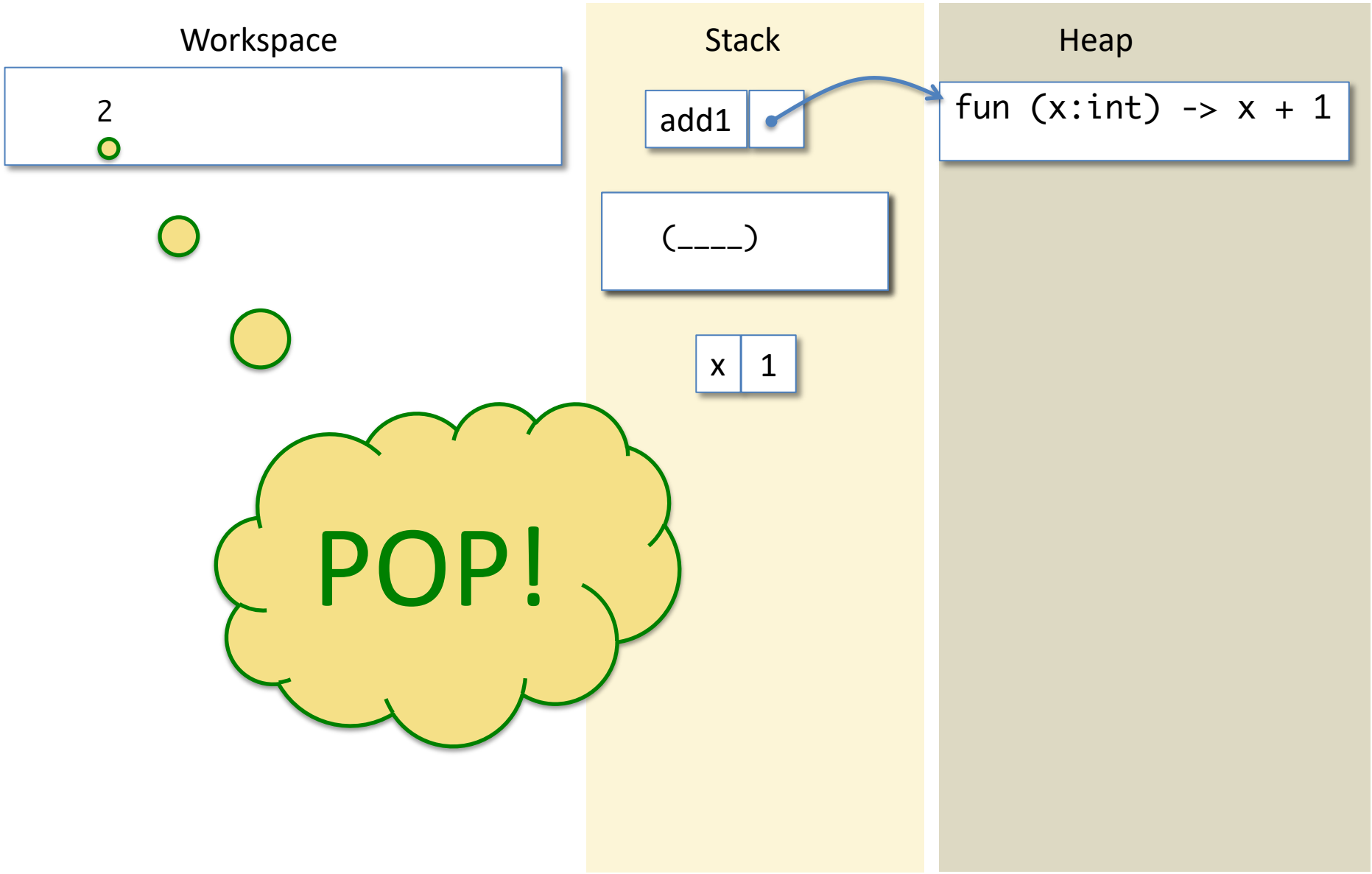
Function Simplification



Function Simplification



Function Simplification



Function Simplification

Simplifying Functions

- A function definition "let f $(x_1:t_1)\ldots(x_n:t_n)$ = e in body" is always ready.
  - It is simplified by replacing it with "let f = fun $(x:t_1)\ldots(x:t_n)$ = e in body"
- A function "fun $(x_1:t_1)\ldots(x_n:t_n)$ = e" is always ready.
  - It is simplified by moving the function to the heap and replacing the function expression with a pointer to that heap data.
- A function *call* is ready if the function and its arguments are all values
  - it is simplified by
    - saving the current workspace contents on the stack
    - adding bindings for the function's parameter variables (to the actual argument values) to the end of the stack
    - copying the function's body to the workspace

Function Completion

- When the workspace contains just a single value, we *pop the stack* by removing everything back to (and including) the last saved workspace contents.

- The value currently in the workspace is substituted for the function application expression in the saved workspace contents, which are put back into the workspace.

- If there aren't any saved workspace contents in the stack, the whole computation is finished and the value in the workspace is its final result.