# Programming Languages and Techniques (CIS120)

Lecture 13

Feb 10, 2012

## Abstract Stack Machine

# Announcements

- Homework 4 is available on the web
  - due Monday, February 13th at 11:59:59pm
  - n-body physics simulation
  - start early; see Piazza for discussions

- Midterm 1 will be in class on Wednesday, February 15th
  - LOCATION: LLAB 10
  - Review materials on website
  - Bring questions to lab
  - Review session Tuesday evening
  - Let me know about scheduling problems ASAP

# "with" notation for *copying* records

- Sometimes it is useful to *copy* one record while replacing just a few of its fields:

```
(* using 'with' notation to copy a record but
   change one (or more) fields *)
let cyan = {blue with g=255}
let magenta = {red with b=255}
let yellow = {green with r=255}
```

- Syntax: `{record with field1=val1; field2=val2}`
  - note the '{' and '}'

# Imperative Programming

# *Mutable* Record Fields

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

"in-place" update of p0.x

# Updating records

- Functions can assign to mutable record fields
- Note that the return type of '<–' is unit

```
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
```

# Issue with Mutable State: Aliasing

- What does this function return?

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

```
(* Are you sure? Consider this call to f *)
let ans = f p0 p0
```

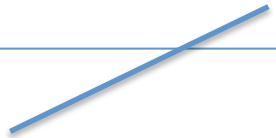
Two identifiers are said to be *aliases* if they both name the *same* mutable record. Inside `f`, `p1` and `p2` might be aliased, depending on which arguments are passed to `f`.

# Aliasing Again

- Does this test pass or fail?

```
let p1 = {x=1; y=1;}
let p2 = p1
;; shift p2 3 4

;; run_test "p1 didn't change"
     (fun () -> (p1.x = 1) && (p1.y = 1))
```



Note: first-class function for testing!

# Reasoning About Mutable State

- Mutable state breaks the simple substitution model!
  - program behaviors become much more difficult to reason about
  - we have to change our mental model of what is going on…
- For example, if we try to use substitution:

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

⟼

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

⟼

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

# Evaluation Cont'd

…

⟶

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

⟶

```
let p1 = {x=1; y=1;}
let p2 = {x=1; y=1;}
let ans = {x=17; y=1;}???; {x=1; y=1;}.x
```

What's going on here?

⟶

```
let p1 = {x=1; y=1;}
let p2 = {x=1; y=1;}
let ans = (); {x=1; y=1;}.x
```

⟶

```
let p1 = {x=1; y=1;}
let p2 = {x=1; y=1;}
let ans = 1
```

This is the *wrong* answer!

# Abstract Stack Machine

# Abstract Machines

- The job of a programming language is to provide some *abstraction* of the underlying hardware
- An *abstract machine* hides the details of the real machine
  - model doesn't depend on the hardware
  - easier to reason about the behavior of programs
- There are lots of ways of visualizing abstract machines
  - e.g. the substitution model we've been using until now
- An Abstract *Stack* Machine
    - is a good way of understanding how recursive functions work
    - gives an accurate picture of how OCaml data structures are shared internally (which helps predict how fast programs will run), and
    - extends smoothly to include imperative features (assignment, pointer manipulation) and objects (for Java)

# Stack Machine

- Three "spaces"
  - workspace
    - contains the expression the computer is currently working with
  - stack
    - temporary storage for variable names
  - heap
    - storage area for large data structures
- Initial state:
  - workspace contains whole program
  - stack and heap are empty
- Machine operation:
  - In each step, find a part of the workspace expression and simplify it (like a step in the substitution semantics)
  - Stop when there are no more simplifications

# Simplification

The abstract machine operates by repeatedly looking for the first (or leftmost) “ready subexpression” in the workspace and simplifying it…

We'll start with an example first, and then define general rules for when subexpressions are "ready" and how they should simplify.

For immutable structures, this stack model is just a complicated way of doing substitution…

… but we need the extra complexity to understand mutable state.

# 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 | Stack | Heap

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

# Simplification

Workspace

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

Stack

| x | 22 |
|---|---|

Heap

# Simplification

Workspace

```
let y = 2 + x 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 = 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

# 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

# Simplification

Workspace

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

Stack

Heap

# Simplification

Workspace

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

Stack

Heap

# Simplification

Workspace

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

Stack

Heap

# Simplification

Workspace | Stack | Heap

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

# Simplification

Workspace

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

Stack

| x | 22 |
|---|---|

Heap

# Simplification

Workspace

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

Stack

| x | 22 |
|---|---|

Heap

# Simplification

Workspace

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

Stack

| x | 22 |
|---|---|

Heap

# Simplification

Workspace

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

Stack

| x | 22 |
|---|---|

Heap

# Simplification

Workspace

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

Stack

| x | 22 |
|---|---|

Heap

# Simplification

Workspace

```
let x = 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 |
|---|---|

| x | 24 |
|---|---|

Heap

# Simplification

Workspace

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

Stack

| x | 22 |
|---|---|

| x | 24 |
|---|---|

Heap

# Simplification

Workspace

```
if 24 > 23 then 3 else 4
```

Stack

| x | 22 |
|---|---|

| x | 24 |
|---|---|

Heap

# Simplification

Workspace

```
if 24 > 23 then 3 else 4
```

Stack

| x | 22 |
|---|---|

| x | 24 |
|---|---|

Heap

# Simplification

Workspace

```
if true then 3 else 4
```

Stack

| x | 22 |
|---|---|

| x | 24 |
|---|---|

Heap

# Simplification

Workspace

```
if true then 3 else 4
```

Stack

| x | 22 |
|---|---|

| x | 24 |
|---|---|

Heap

# Simplification

Workspace

3

Stack

| x | 22 |
|---|---|

| x | 24 |
|---|---|

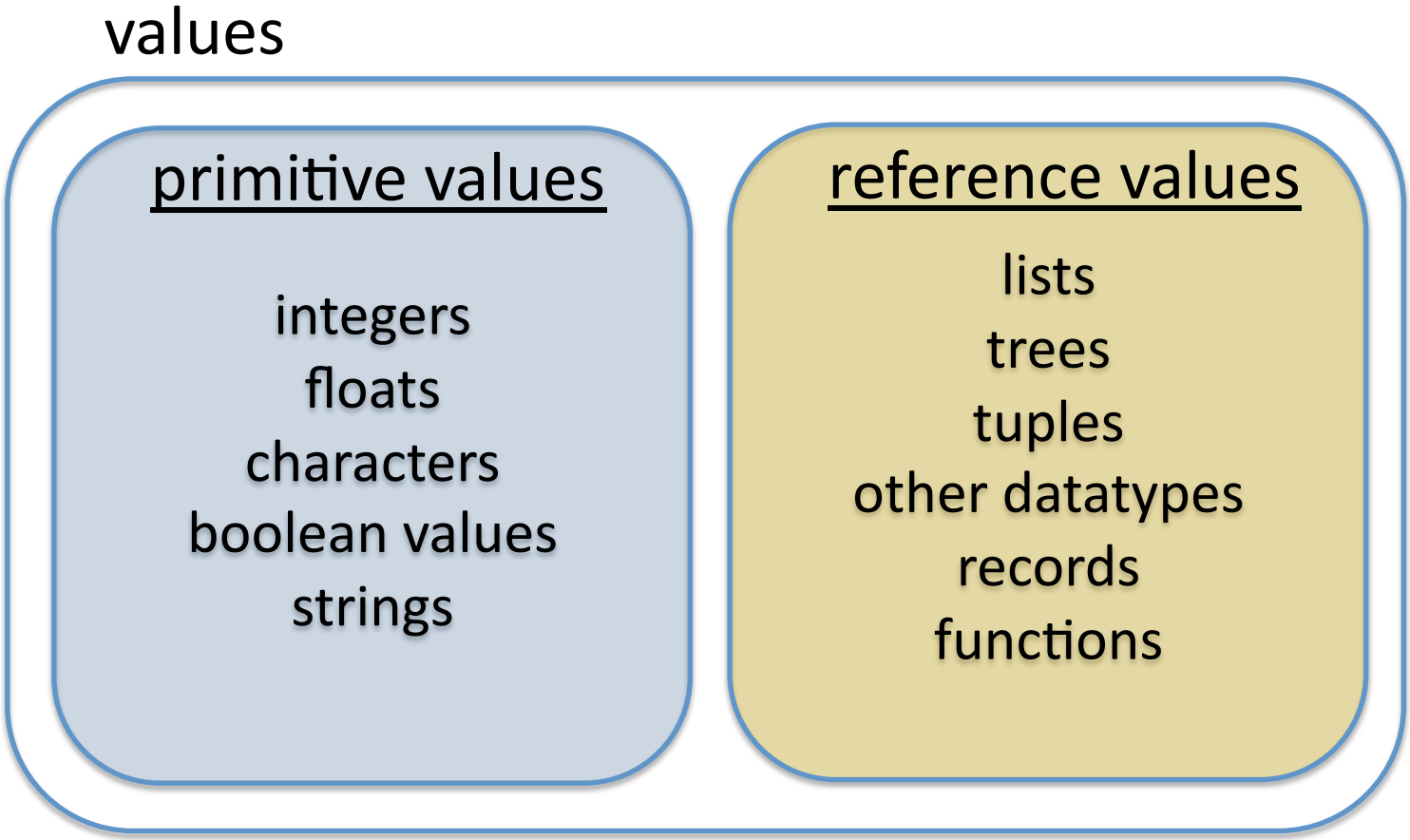
Heap

# Simplification Rules

- A let-expression "`let x = e in body`" is ready if the expression `e` is a *value*
  - it is simplified by adding a binding of `x` to `e` at the end of the stack and leaving body in the workspace
- A variable is always ready
  - it is simplified by replacing it with its value from the stack, where binding lookup goes in order from most recent to least recent
- A primitive operator (like +) is ready if both of its arguments are values
  - it is simplified by replacing it with the result of the operation
- An if expression is ready if the test is true or false
  - if it is true, it is simplified by replacing it with the then branch
  - if it is false, it is simplified by replacing it with the else branch

# Introducing the Heap

- An important property of the ASM is that only *primitive* values are stored on the stack



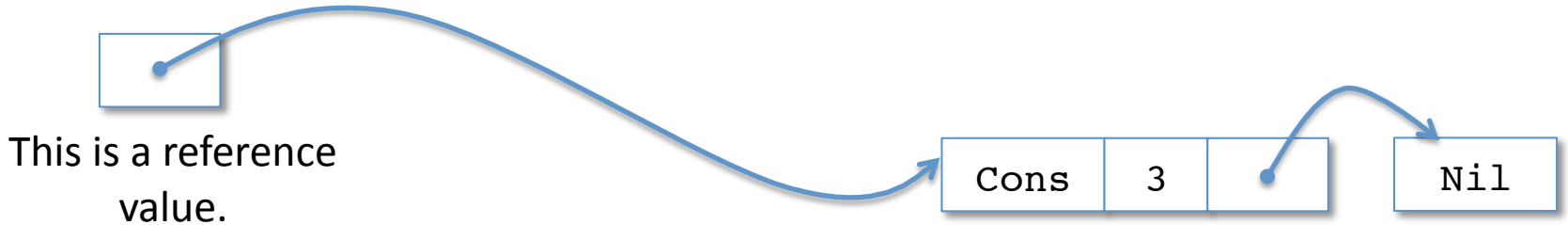
# Values and References

A *value* is either:

- a *primitive* value like an integer or,
- a *reference* (or *pointer*) into the heap

A reference is the *address* (or *location* of) a piece of data in the heap . We draw a reference as an "arrow"

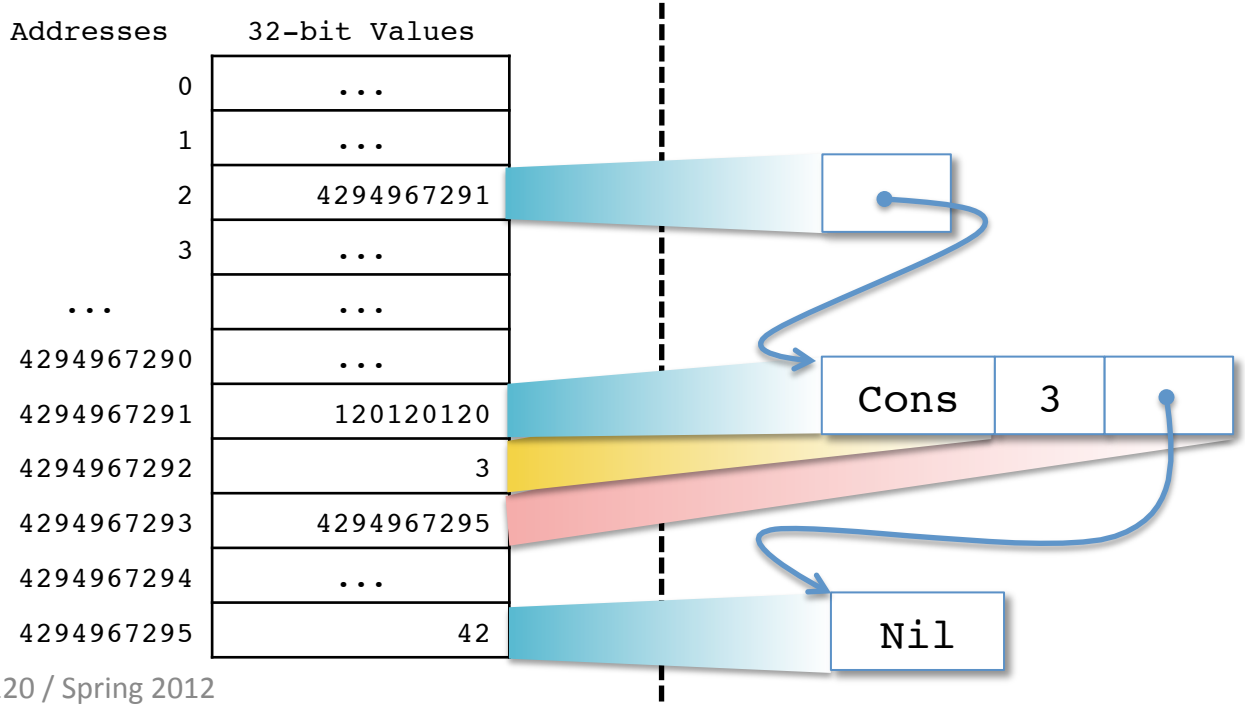
- The start of the arrow is the reference itself (i.e. the address).
- The arrow "points" to the value located at the reference's address.



# References as 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 is just an address that tells you where to lookup a value
  - Datastructures 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

The "real" heap.




How we picture it.

# Data in the Heap

The heap contains a few kinds of data:

- a *cell*, labeled by a datatype constructor, and containing arguments of the constructor (e.g. see previous slides)
  - The constructor name takes up some heap space
  - The arguments are themselves values

- a *record*, containing a value for each of its fields
  - the field names themselves don't take up any space, but we mark them anyway to help us do bookkeeping

- a *function*, written*

$$\texttt{fun } (x_1\texttt{:}t_1)\ldots(x_n\texttt{:}t_n)\texttt{ -> e}$$

*For now, we consider only *top level* functions – nested functions require changes to the model

# Simplifying Datatypes

- A datatype constructor (like `Nil` or `Cons`) is ready if all its arguments are values
- It is simplified by:
  - creating a new heap cell labeled with the constructor and containing the argument values*
  - replacing the constructor expression in the workspace by a pointer to this heap cell

*Note: in OCaml, using a datatype constructor causes some space to be automatically allocated on the heap. Other languages have different mechanisms for accomplishing this: for example, the keyword 'new' in Java works similarly (as we'll see in a few weeks).

# Simplification

Workspace | Stack | Heap

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

# Simplification

Workspace | Stack | Heap

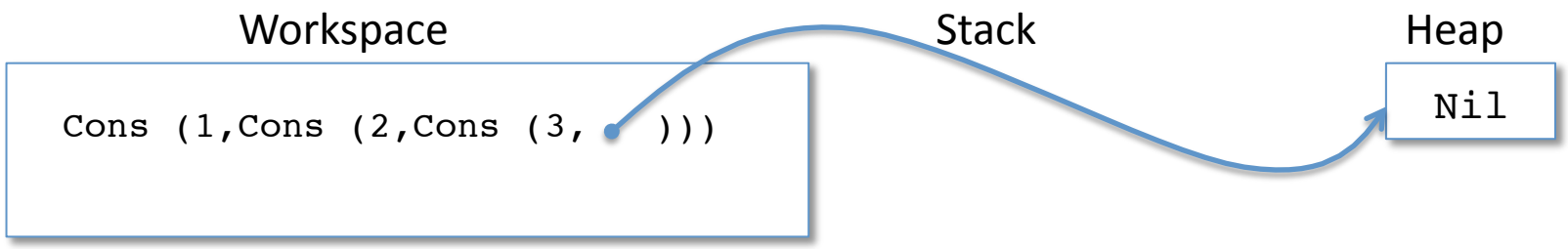
```
Cons (1,Cons (2,Cons (3,Nil)))
```

# Simplification

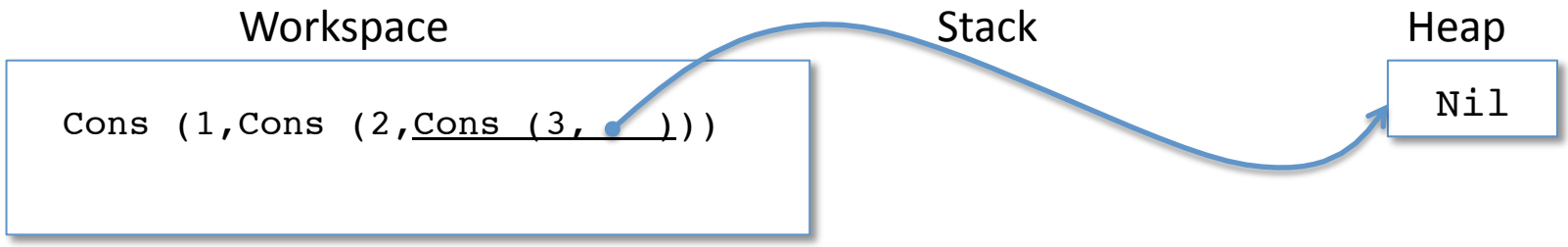
Workspace | Stack | Heap

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

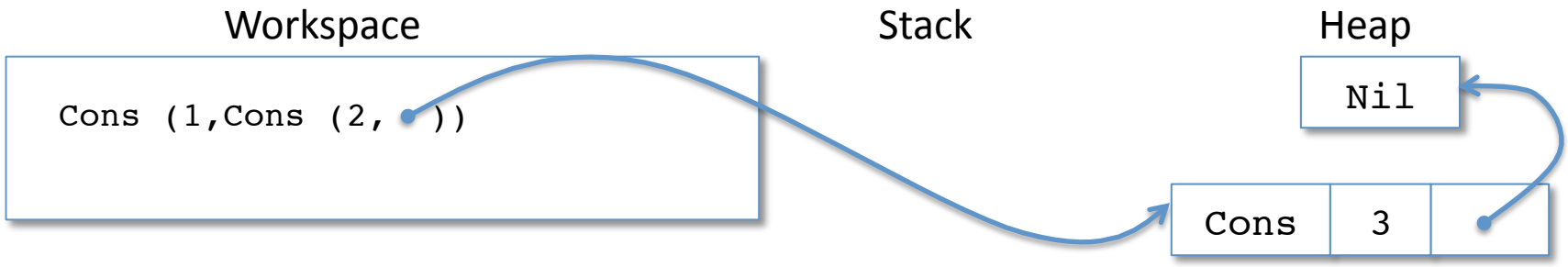
# Simplification



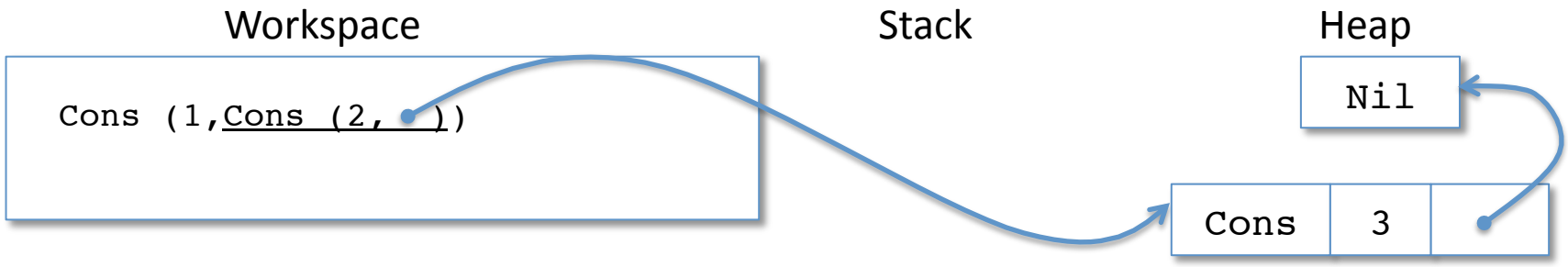
# Simplification



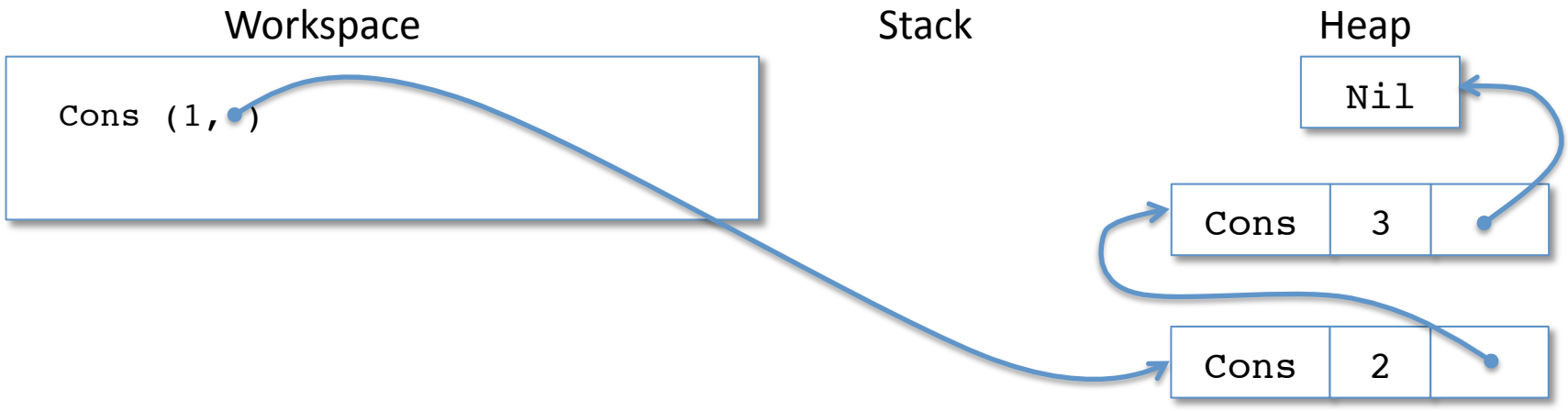
# Simplification



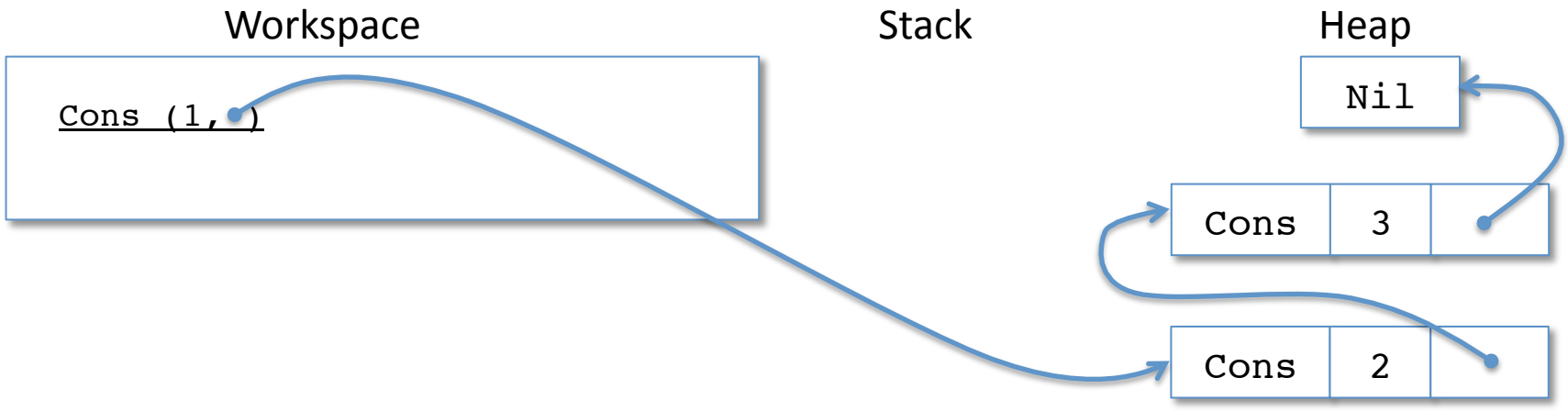
# Simplification



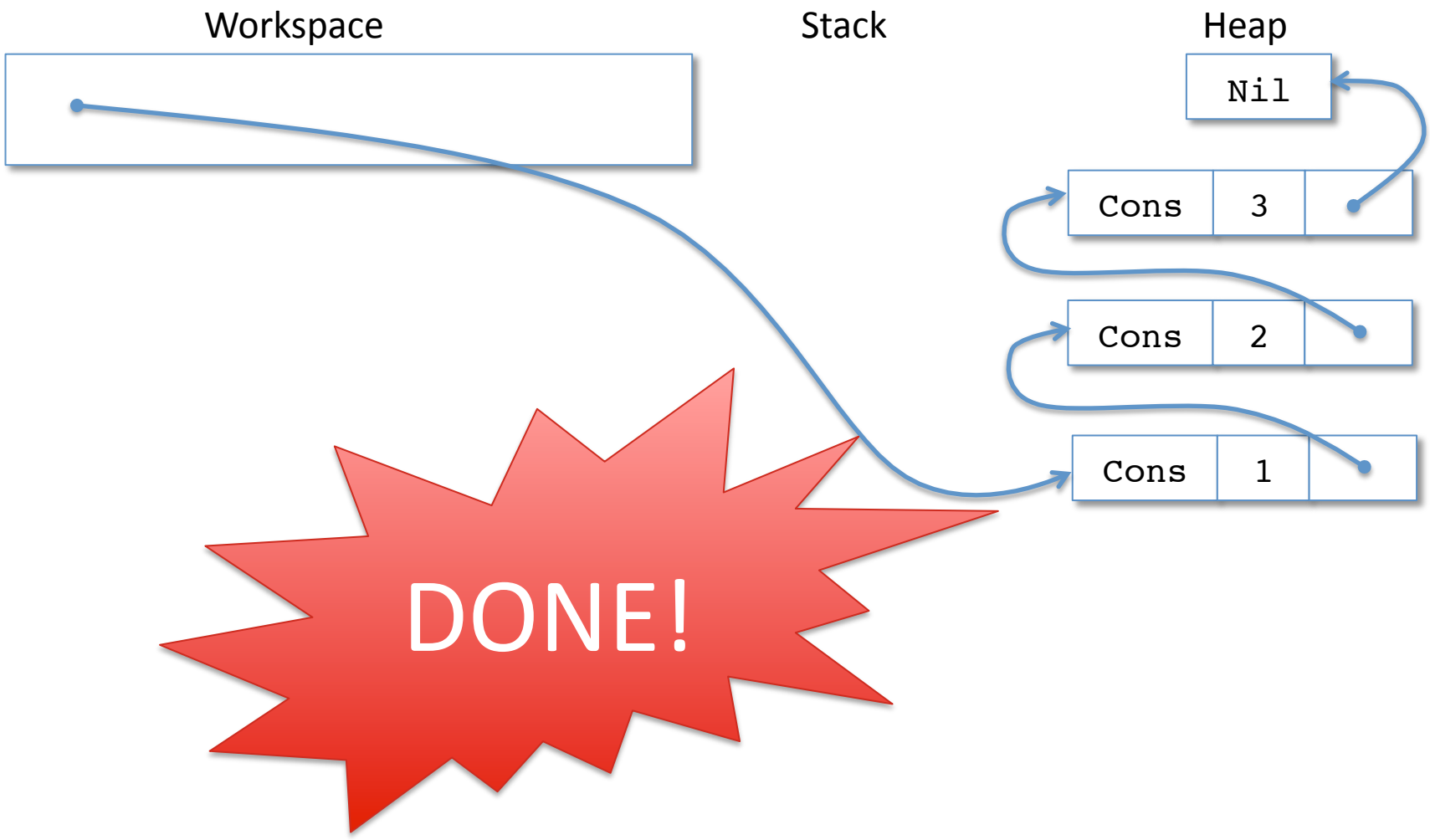
# Simplification



# Simplification



# Simplification



# Simplifying Match

- A match expression

```
begin match e with
  | pat1 -> branch1
  | …
  | patn -> branchn
end
```

  is ready if e is a value
  - Note that e will always be a pointer to a constructor cell in the heap
  - This expression is simplified by finding the first pattern $\texttt{pat}_i$ that matches the cell and adding new bindings for the pattern variables (to the parts of `e` that line up) to the end of the stack
  - replacing the whole match expression in the workspace with the corresponding $\texttt{branch}_i$

# Function Simplification

Workspace | Stack | Heap

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

# Function Simplification

Workspace | Stack | Heap

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

# Function Simplification

Workspace | Stack | Heap

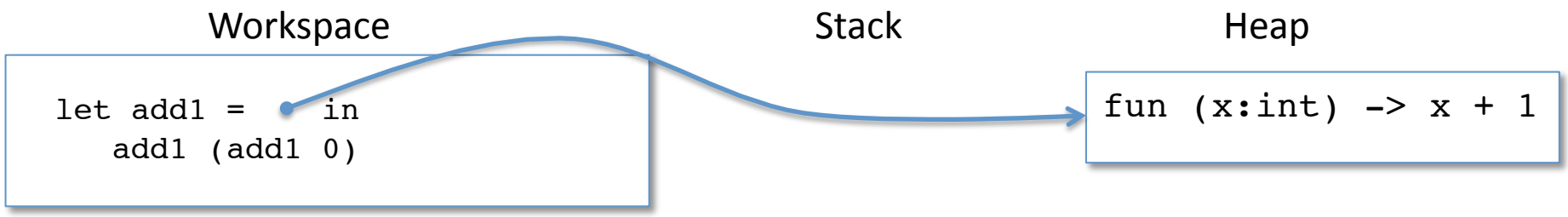
```
let add1 : int ->  int =
  fun (x:int) -> x + 1 in
add1 (add1 0)
```

# Function Simplification

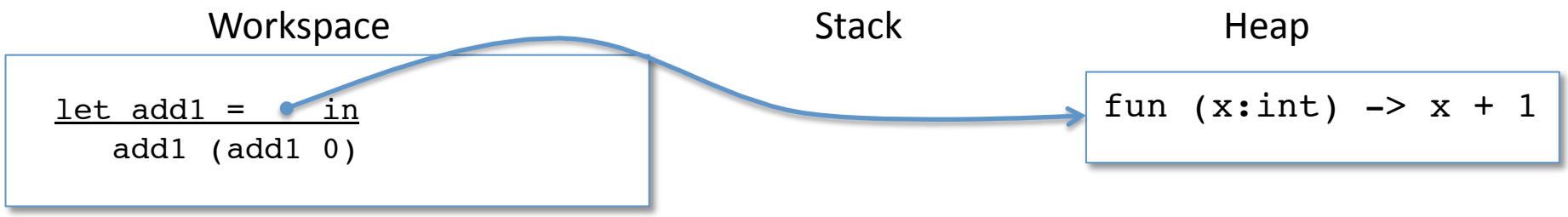
Workspace | Stack | Heap

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

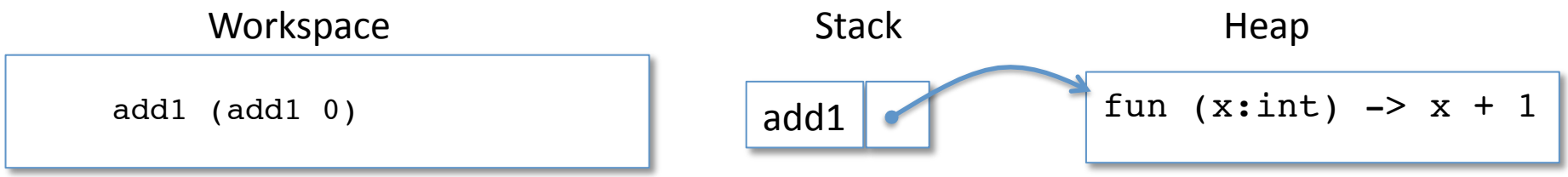
# Function Simplification



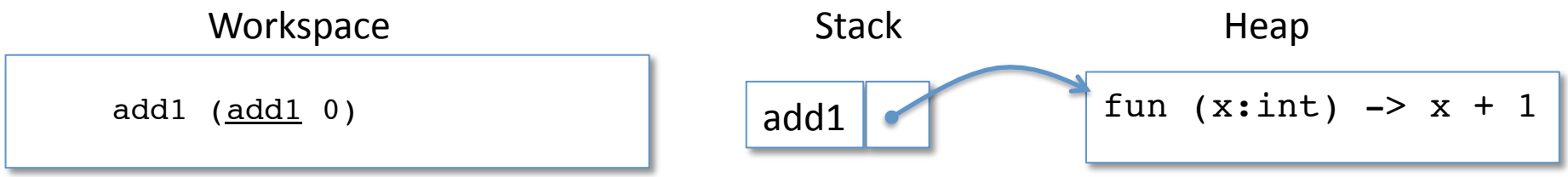
# Function Simplification



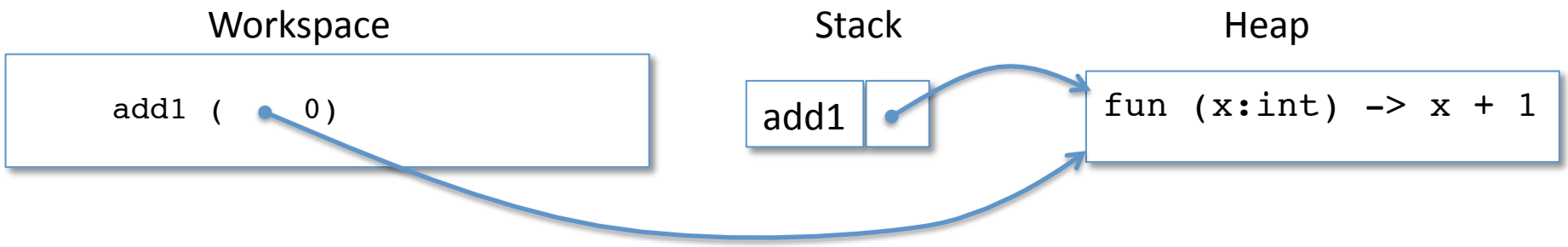
# Function Simplification



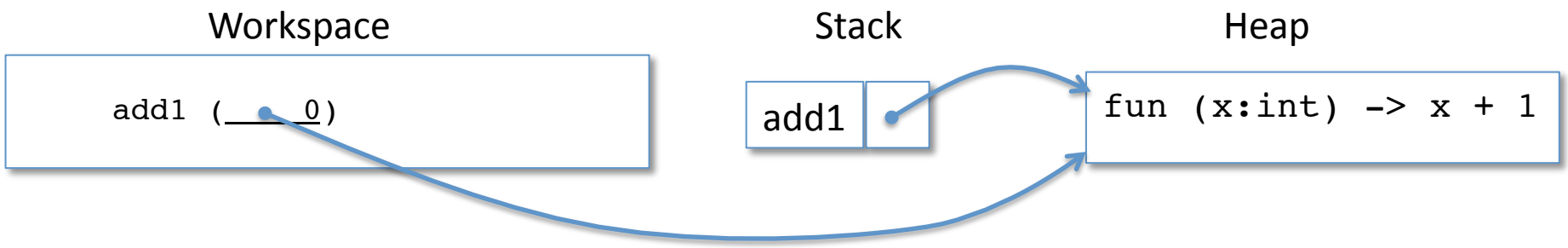
# Function Simplification



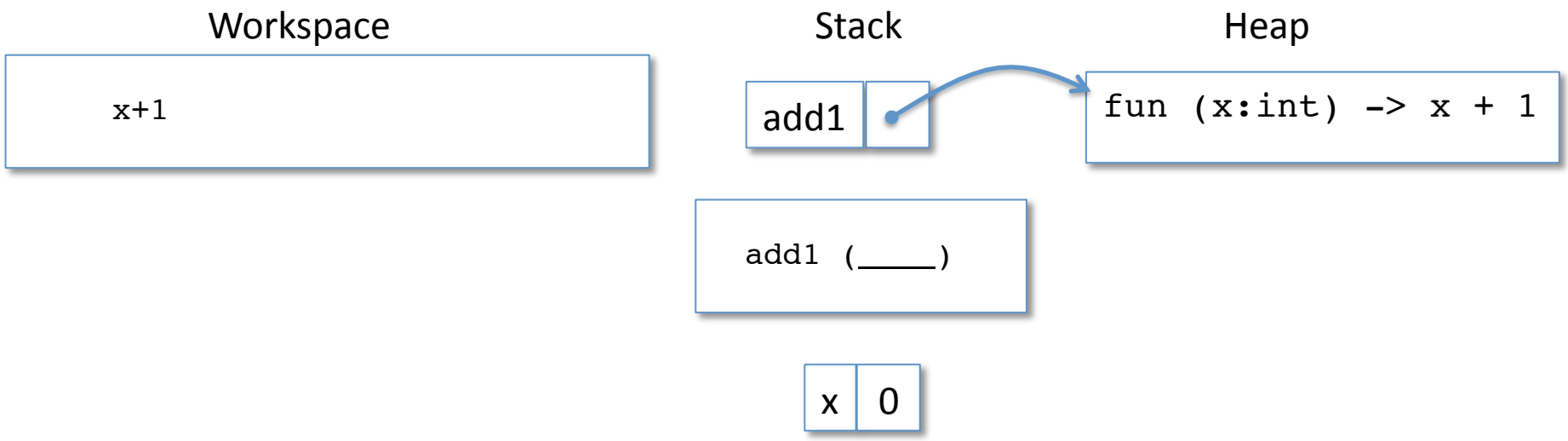
# Function Simplification



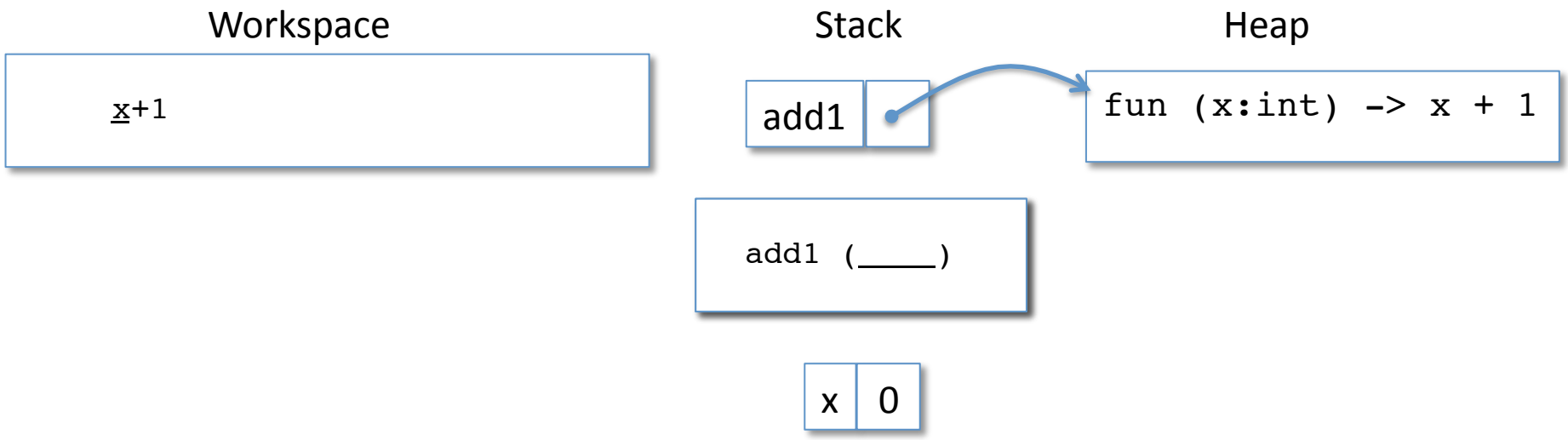
# Function Simplification



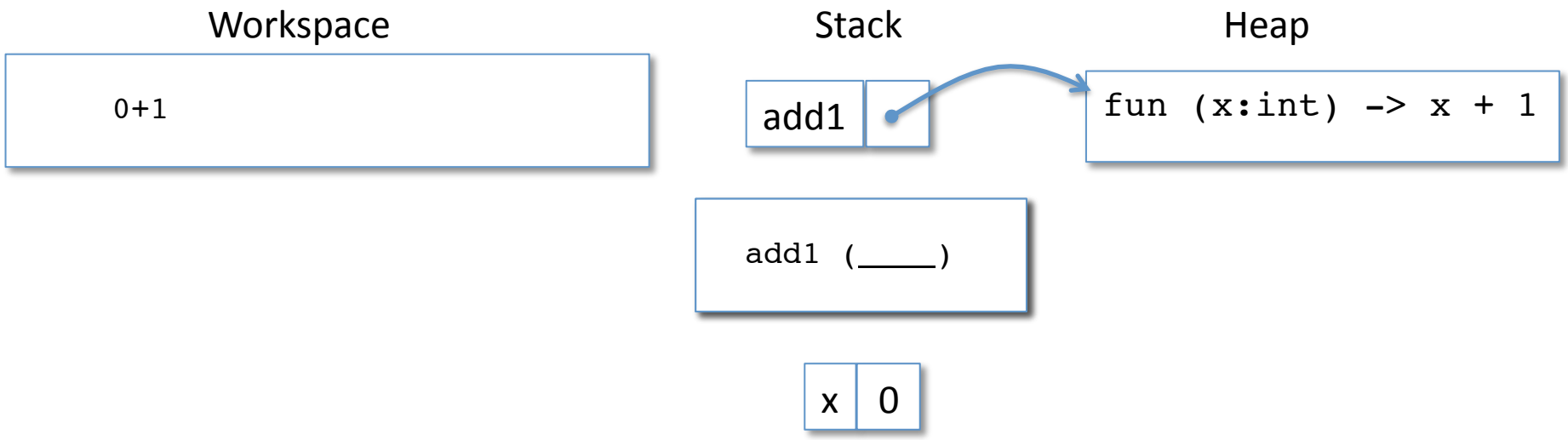
# Do the Call, Saving the Workspace



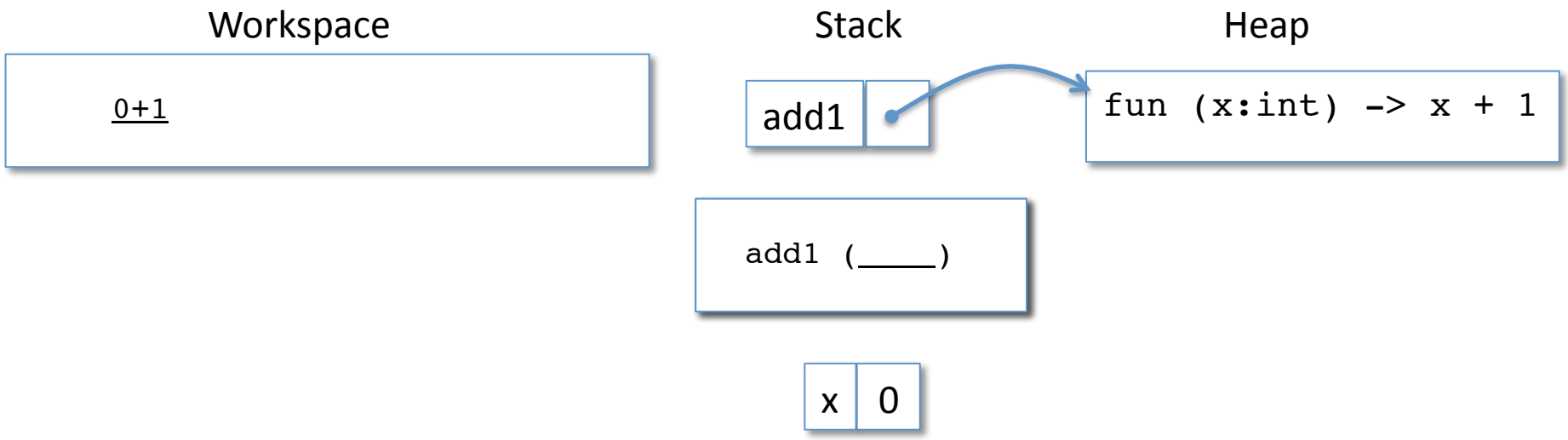
# Function Simplification



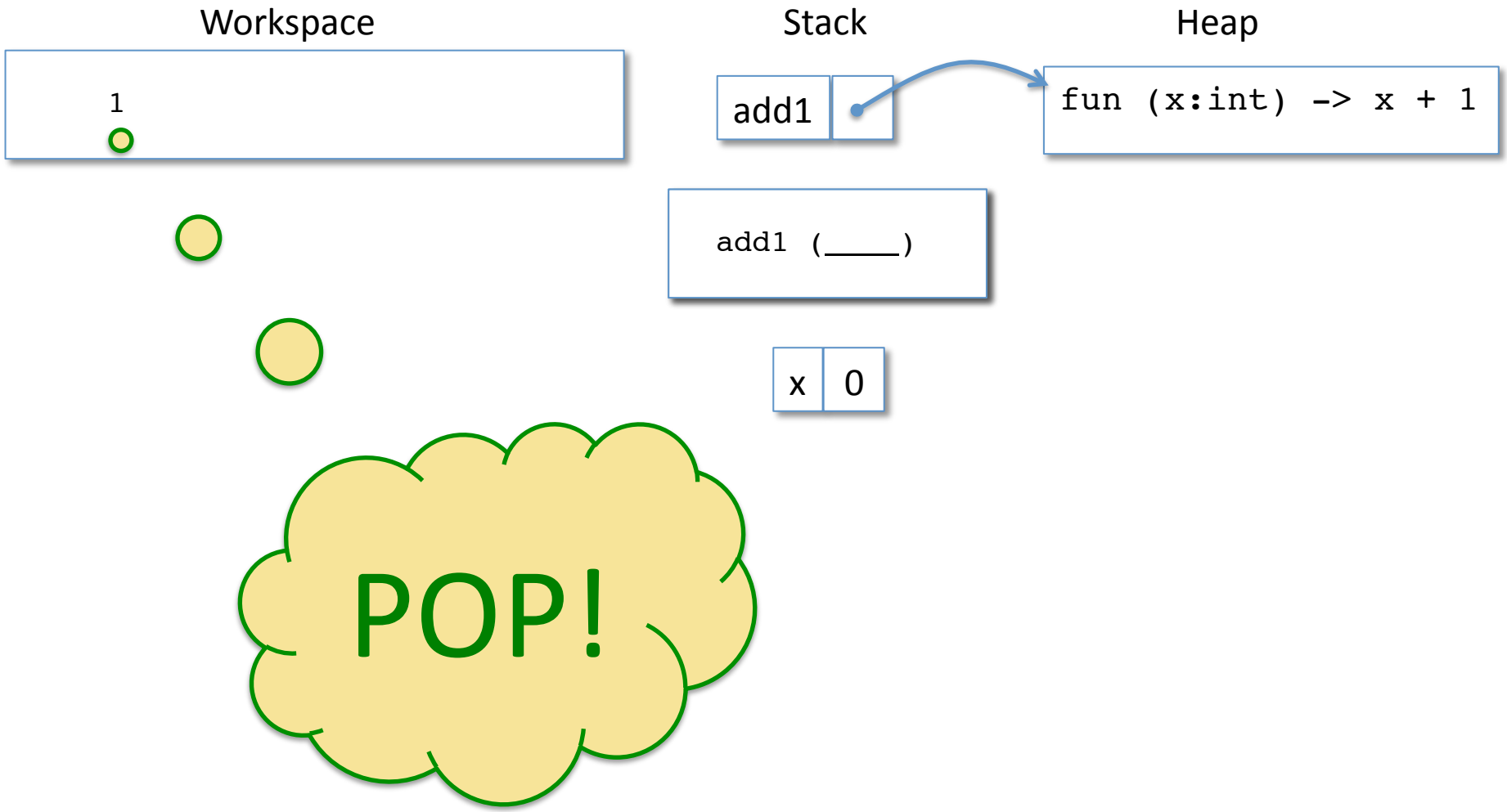
# Function Simplification



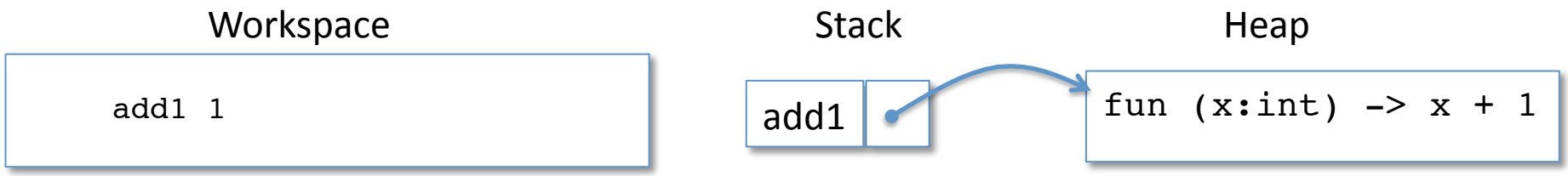
# Function Simplification



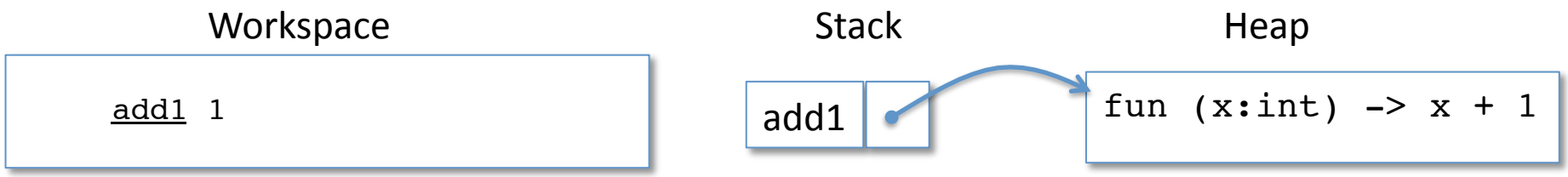
# Function Simplification



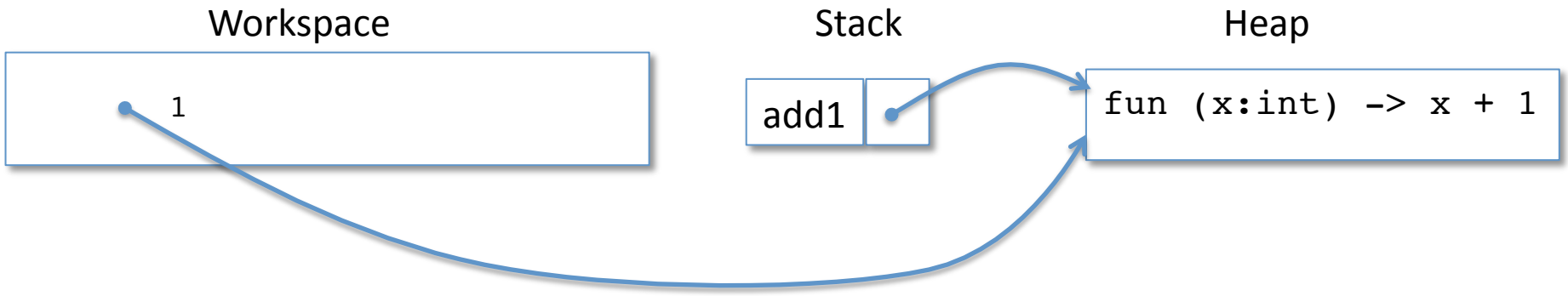
# Function Simplification



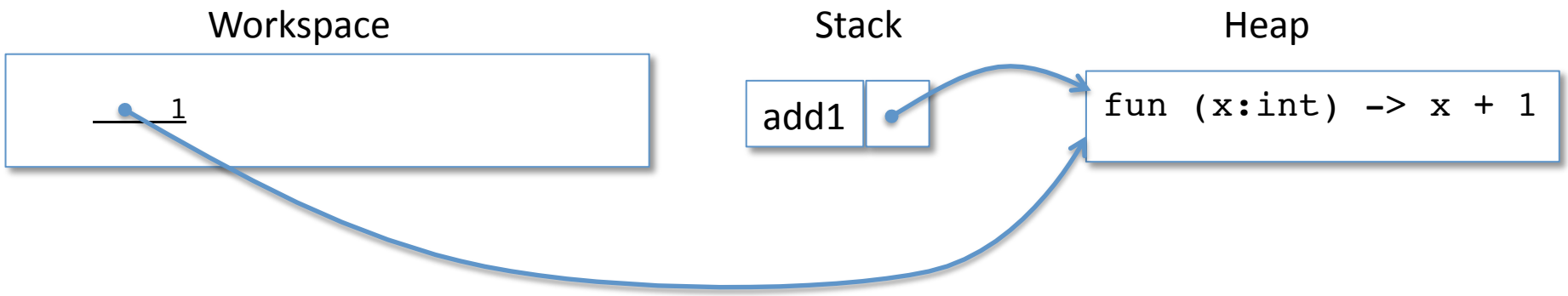
# Function Simplification



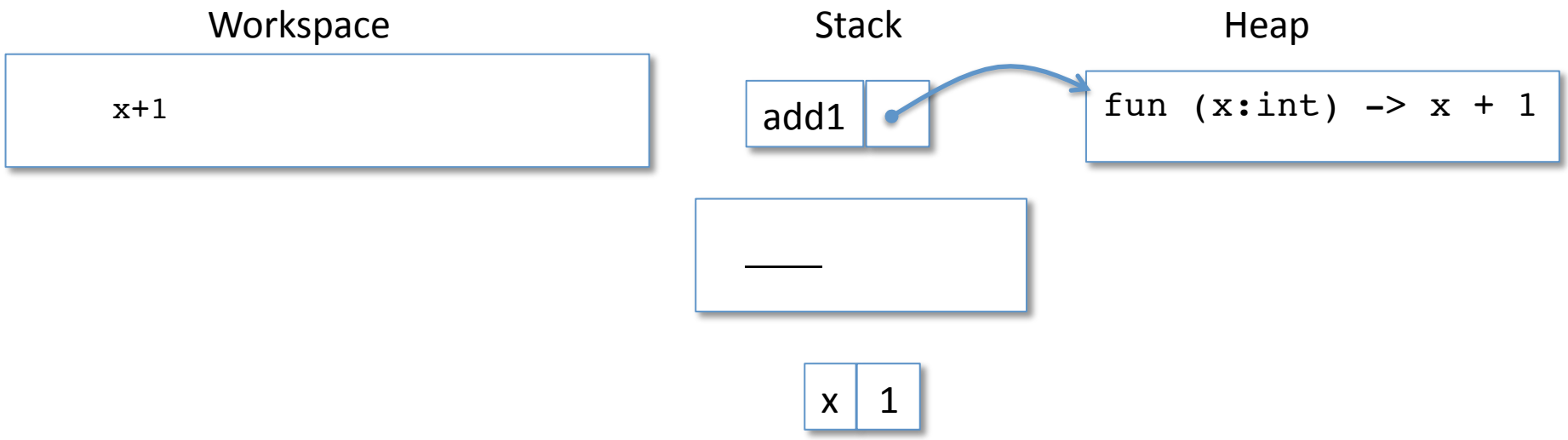
# Function Simplification



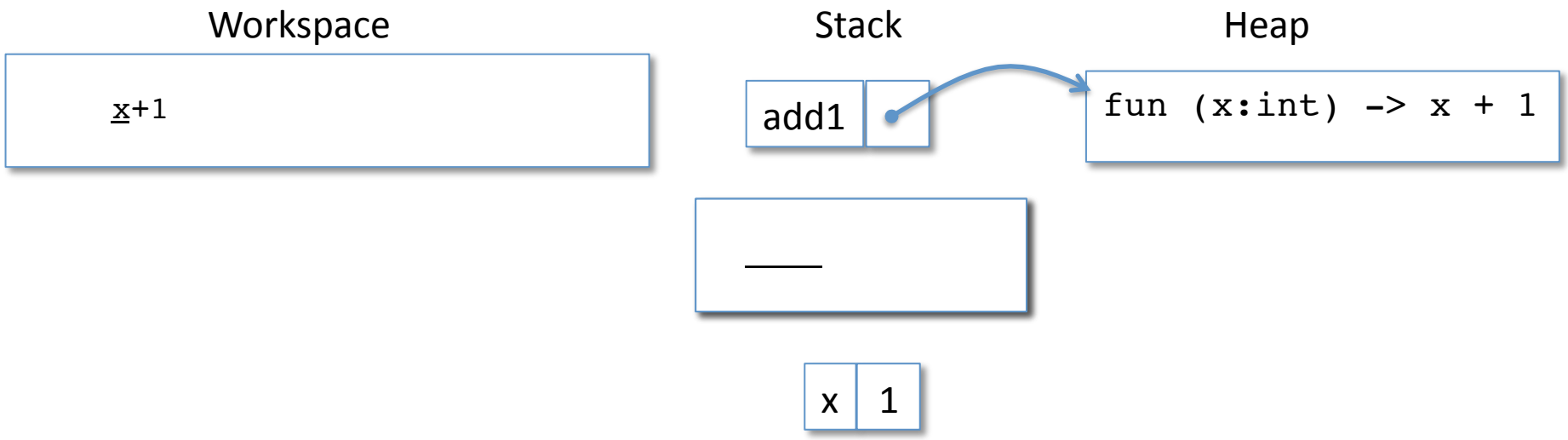
# Function Simplification



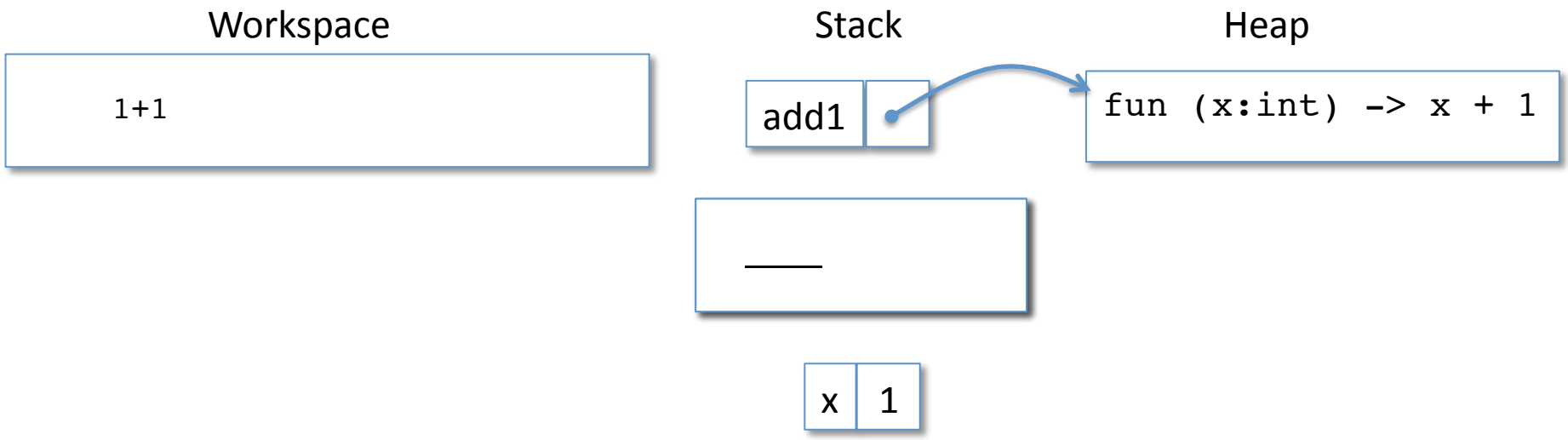
# Function Simplification



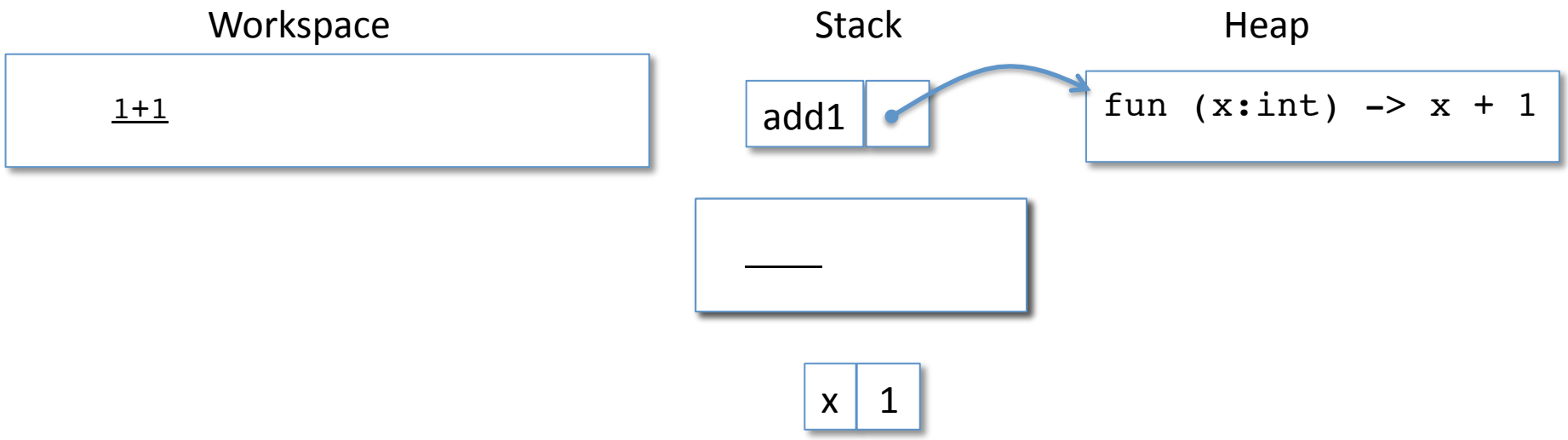
# Function Simplification



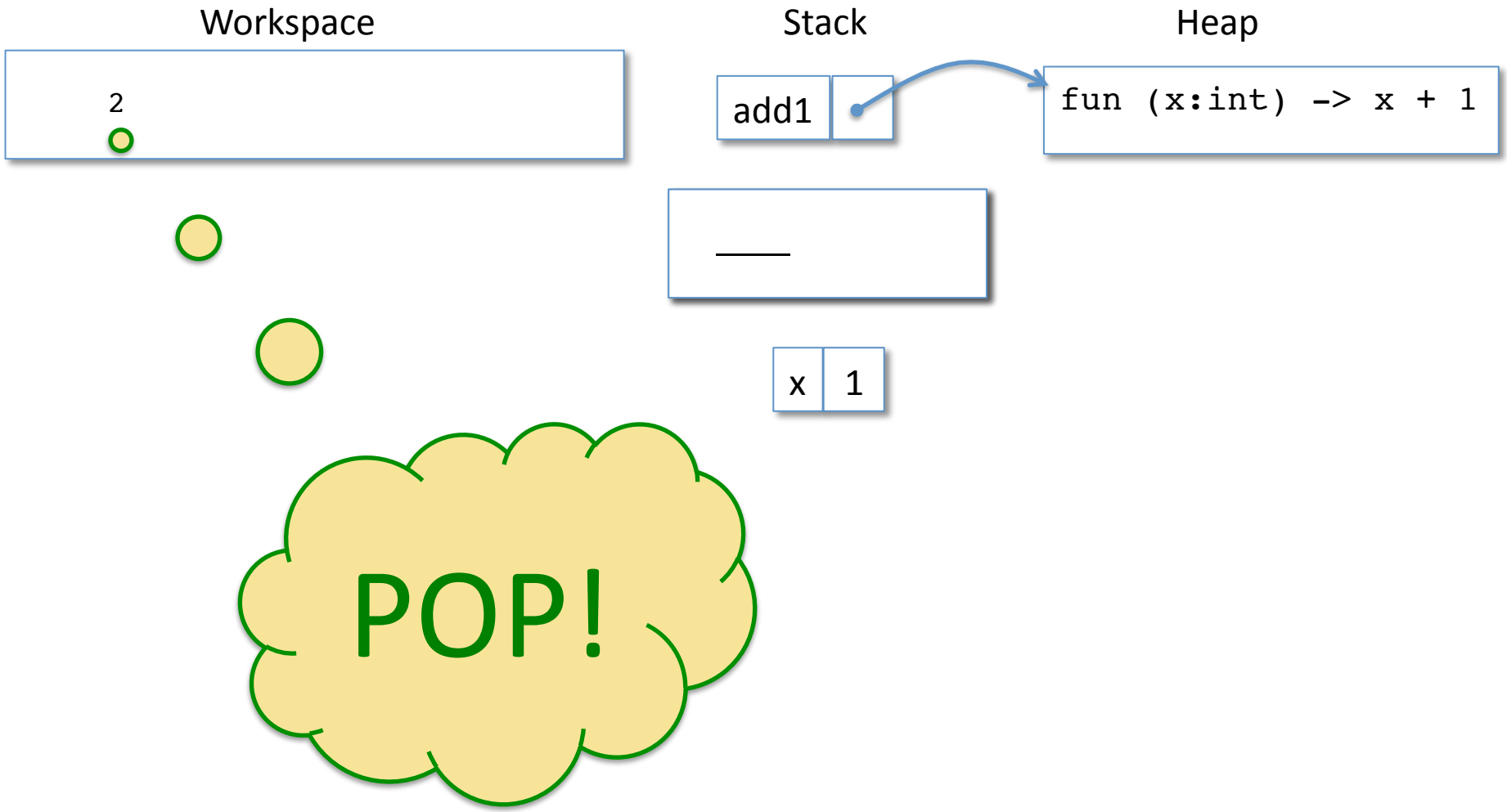
# Function Simplification



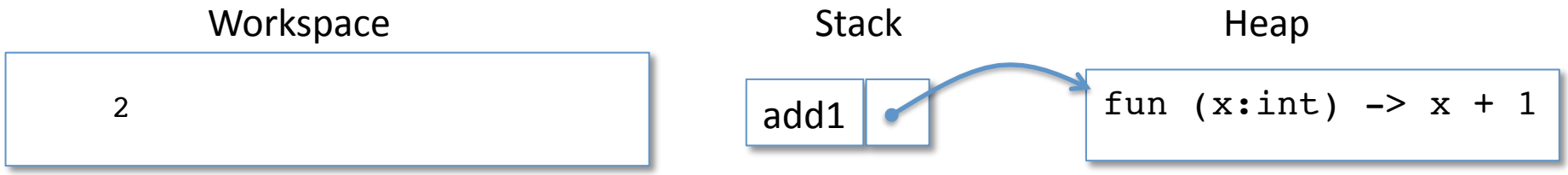
# Function Simplification



# Function Simplification



# Function Simplification




# Simplifying Functions

- A function definition "let rec f ($x_1$:$t_1$)…($x_n$:$t_n$) = e in body" is always ready.
  - It is simplified by replacing it with "let f = fun (x:$t_1$)…(x:$t_n$) = e in body"

- A function "fun ($x_1$:$t_1$)…($x_n$:$t_n$) = 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.

# Recursive Function Example

```
let rec append (l1: 'a list) (l2: 'a list) : 'a list =
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
  end in

let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in

append a b
```

# Simplification

Workspace

```
let rec append (l1: 'a list)
    (l2: 'a list) : 'a list =
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
      Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

Heap

# Function Definition

Workspace

```
let rec append (l1: 'a list)
    (l2: 'a list) : 'a list =
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

Heap

# Rewrite to a “fun”

Workspace

```
let append =
  fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
      Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

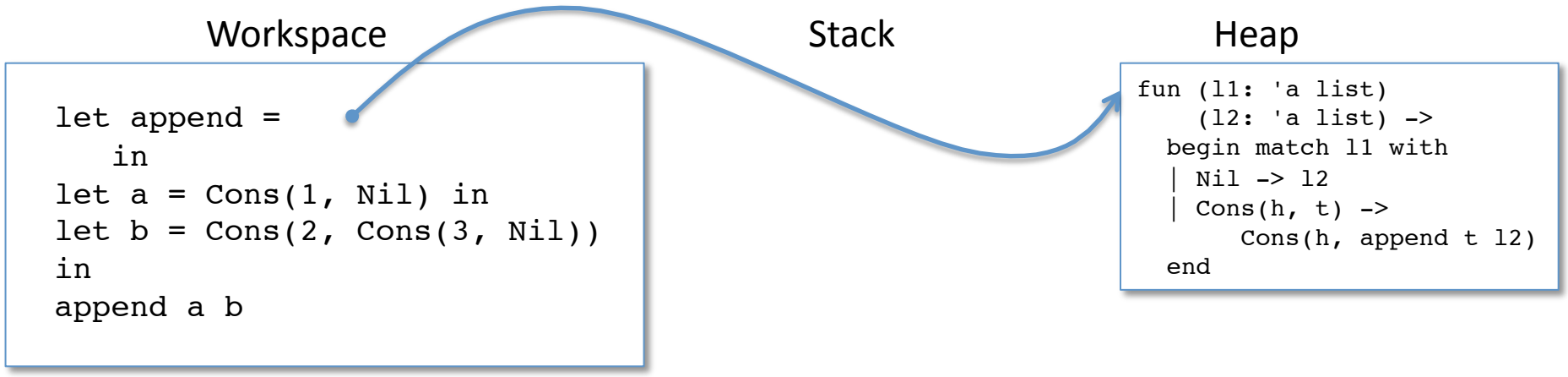
Heap

# Function Expression

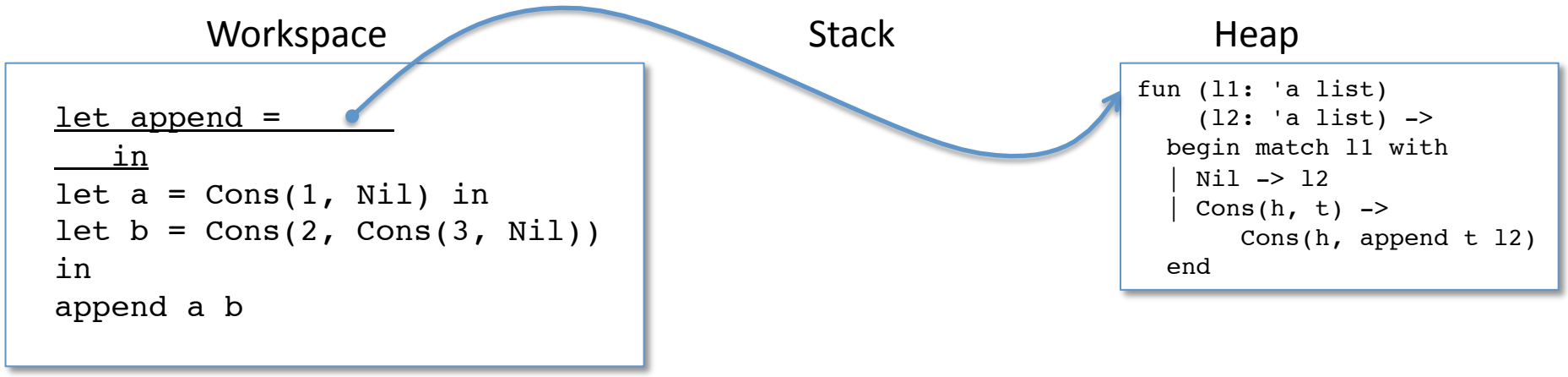
Workspace | Stack | Heap

```
let append =
  fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
      Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

# Copy to the Heap, Replace w/Reference



# Let Expression



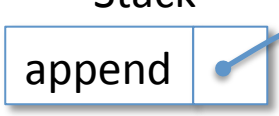

Note that the reference to a function in the heap is a value.

# Create a Stack Binding

Workspace

```
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack




Heap

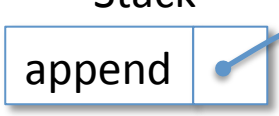
```
fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
      Cons(h, append t l2)
  end
```

# Allocate a Nil cell

Workspace

```
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

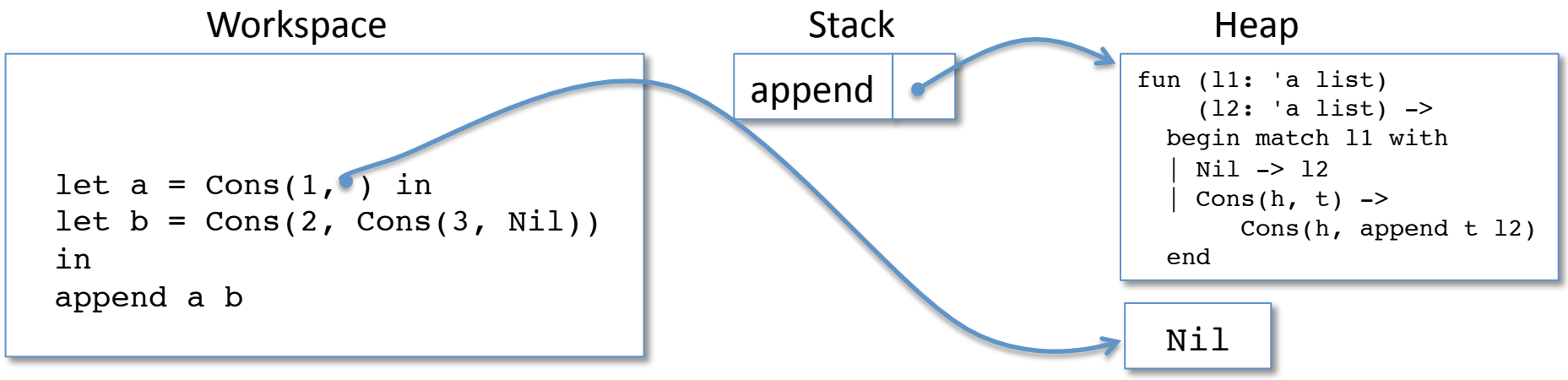
Stack



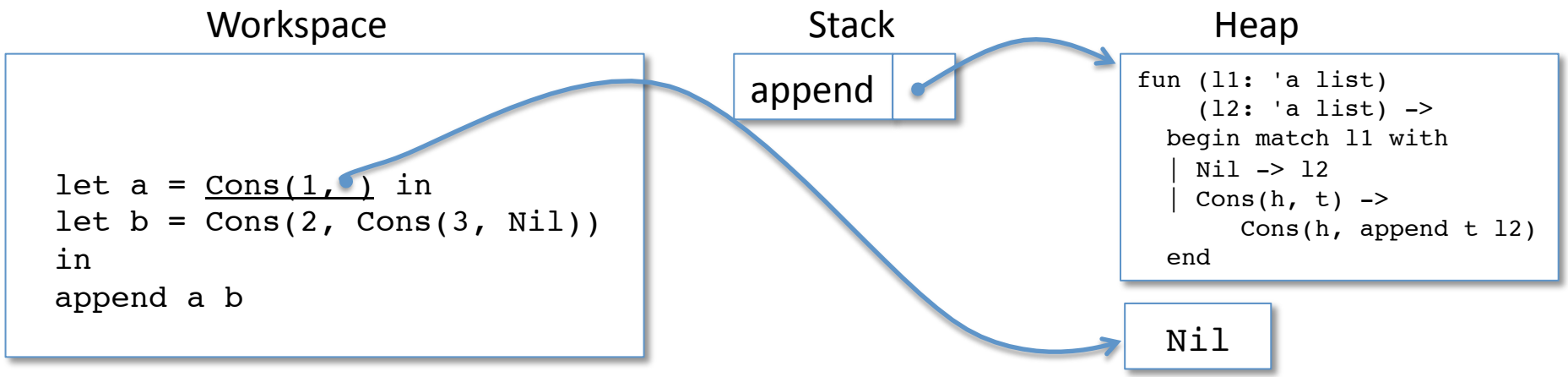

Heap

```
fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
      Cons(h, append t l2)
  end
```

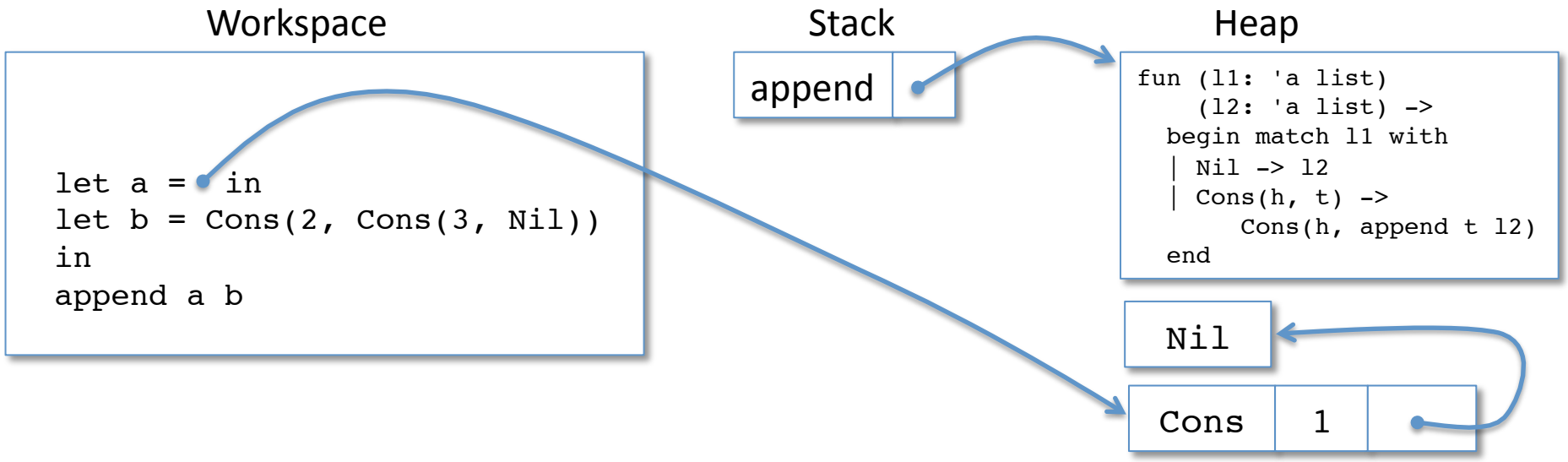
# Allocate a Nil cell



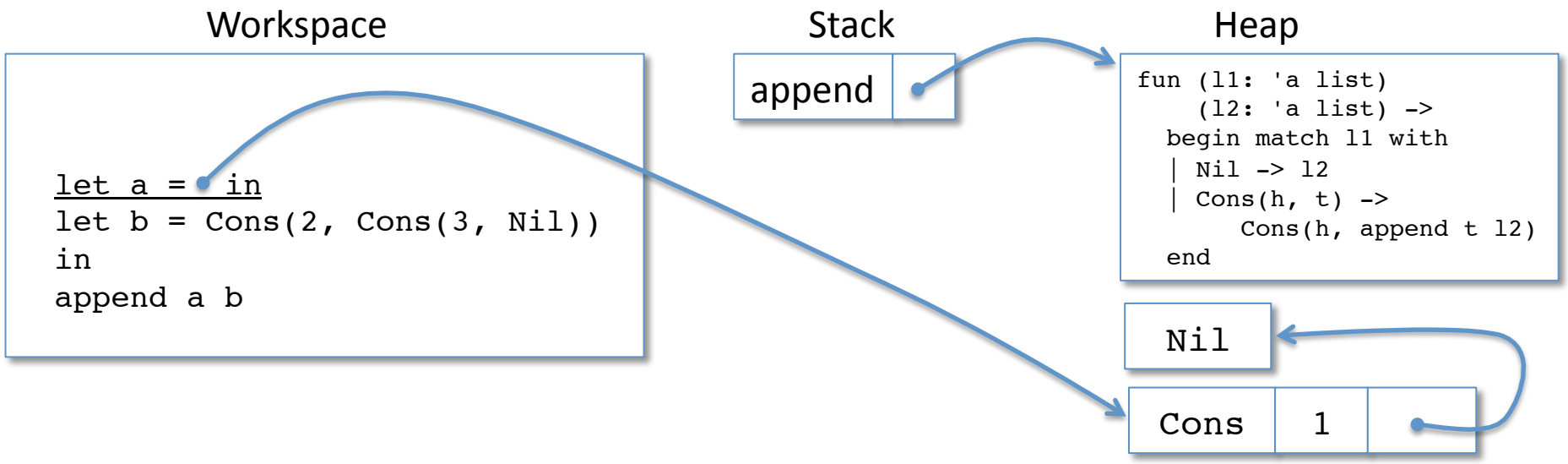
# Allocate a Cons cell



# Allocate a Cons cell



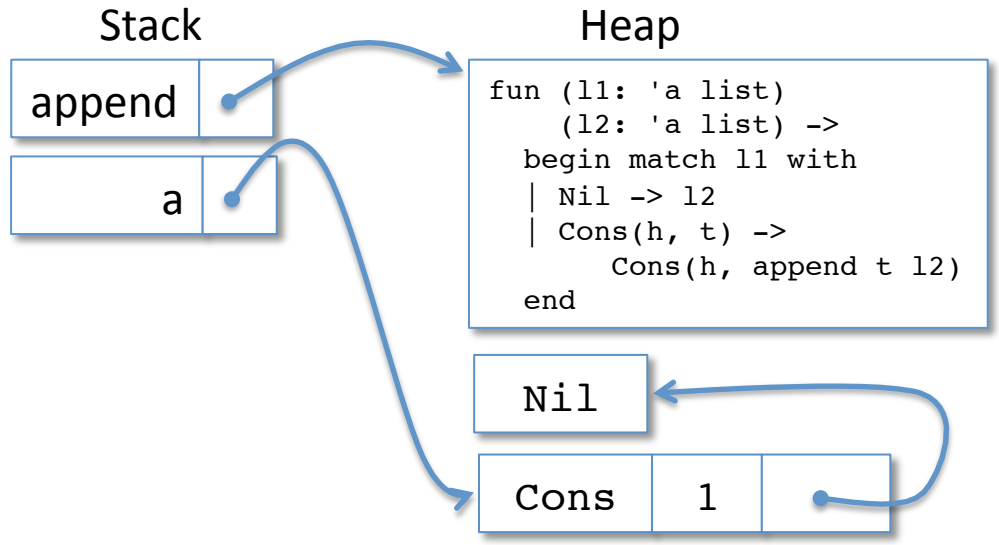
# Let Expression



# Create a Stack Binding

Workspace

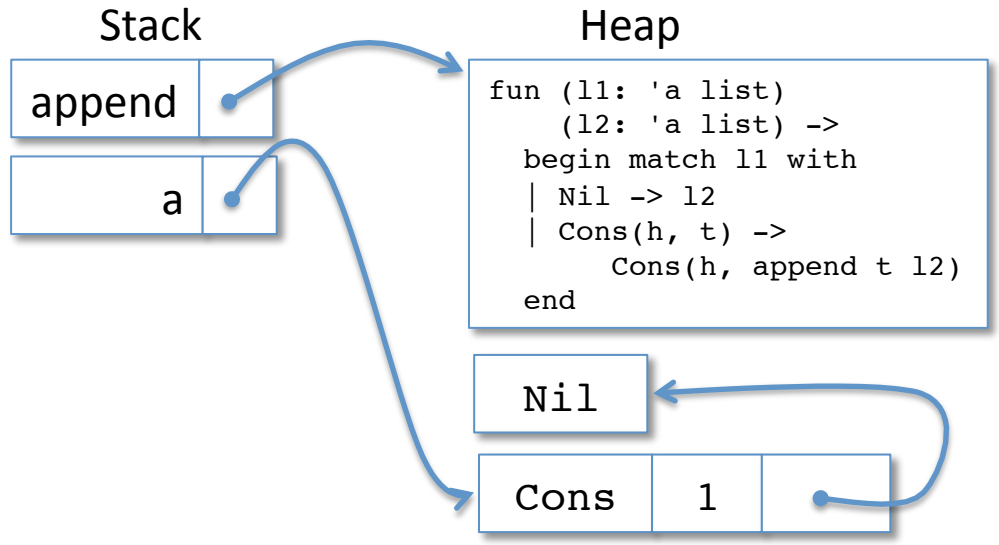
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```



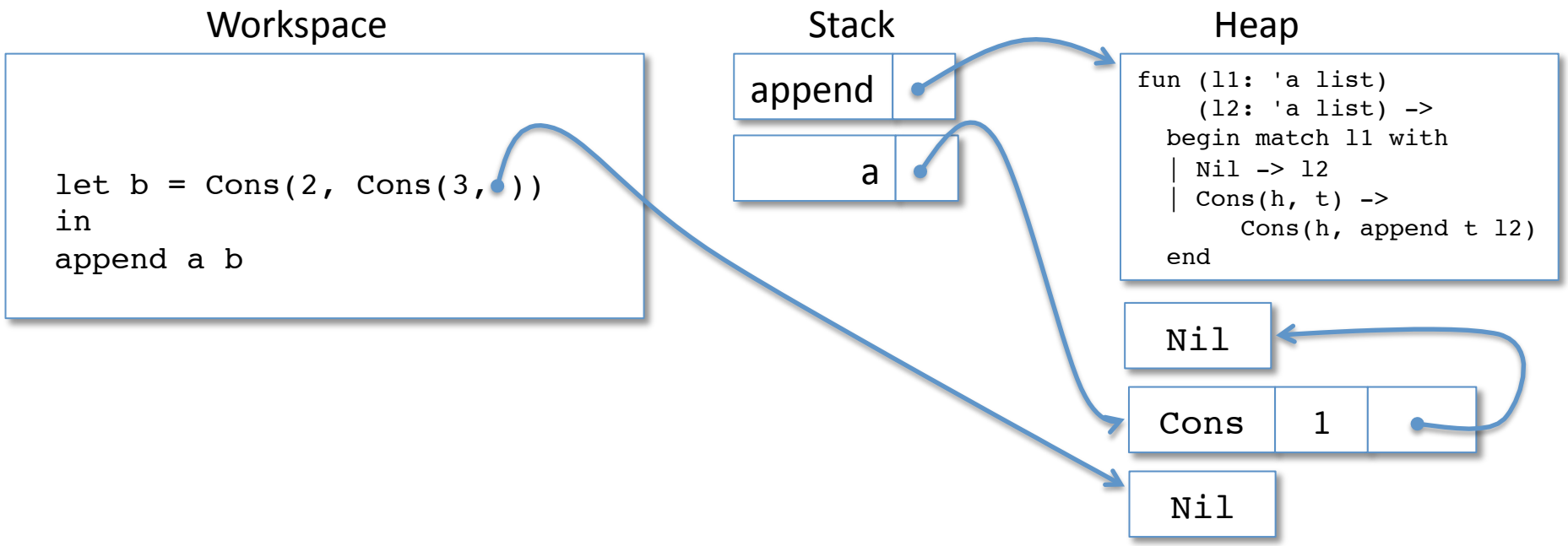
# Allocate a Nil cell

Workspace

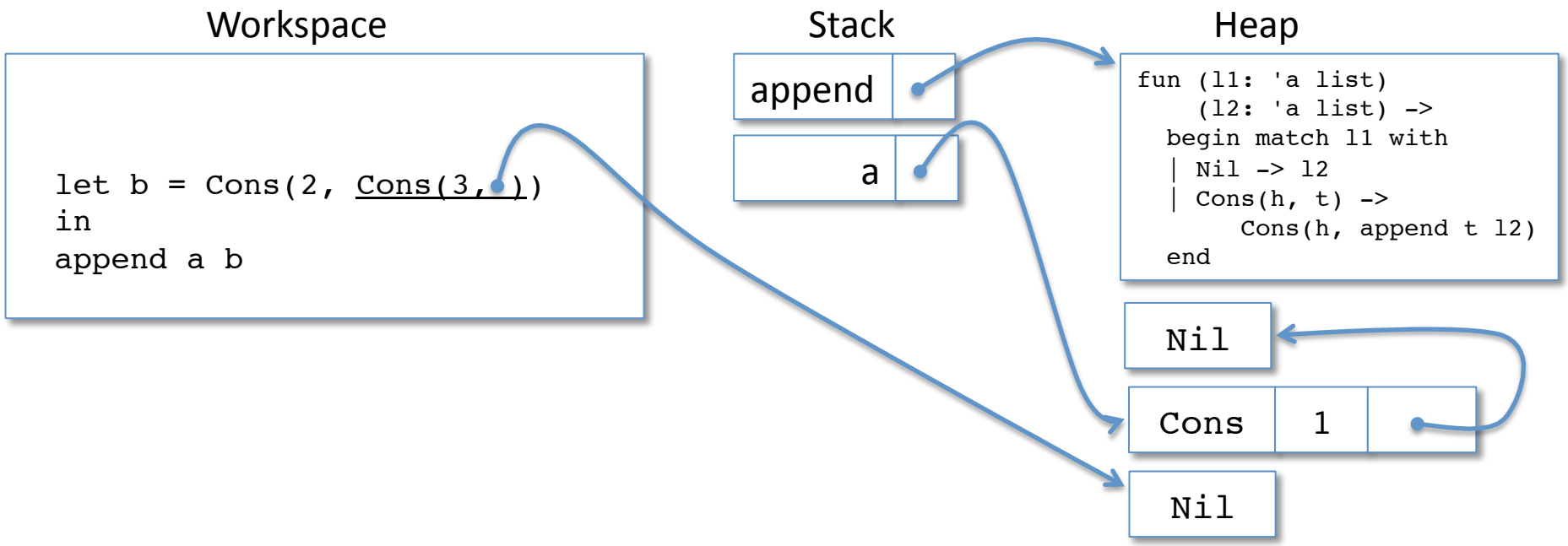
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```



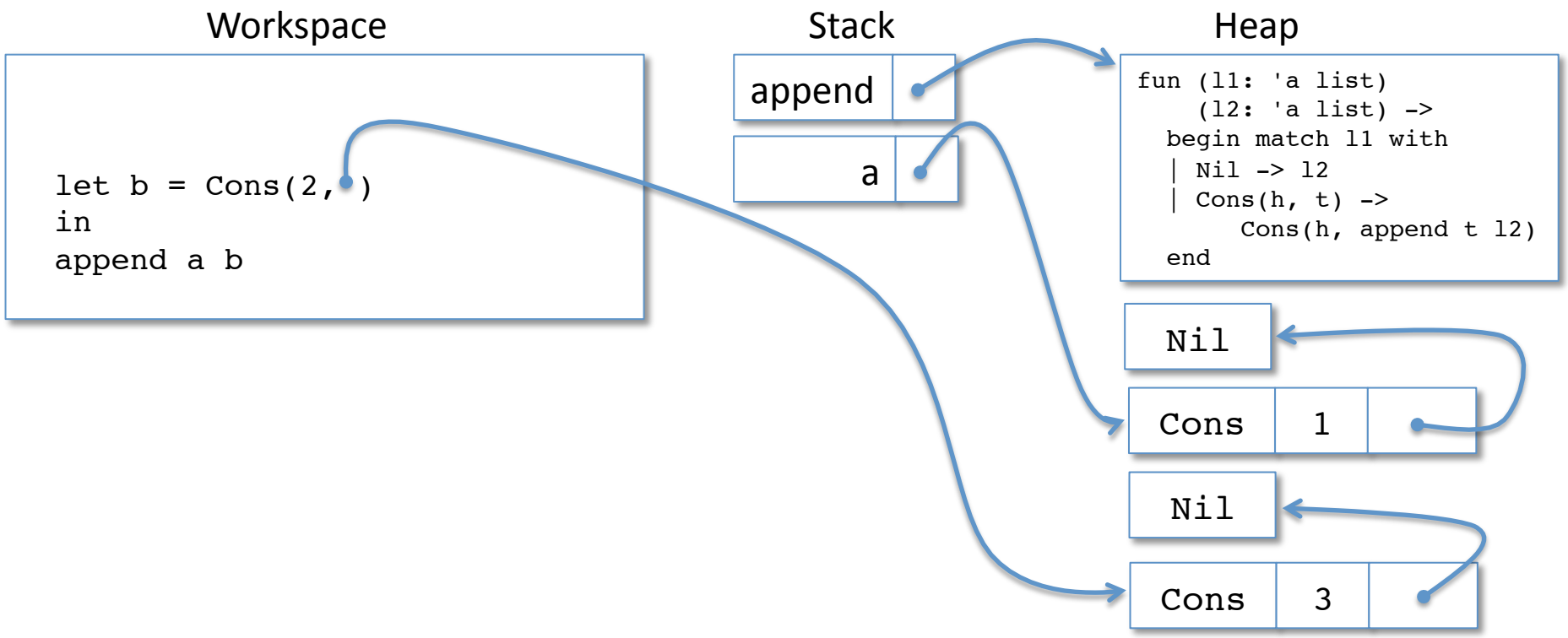
# Allocate a Nil cell



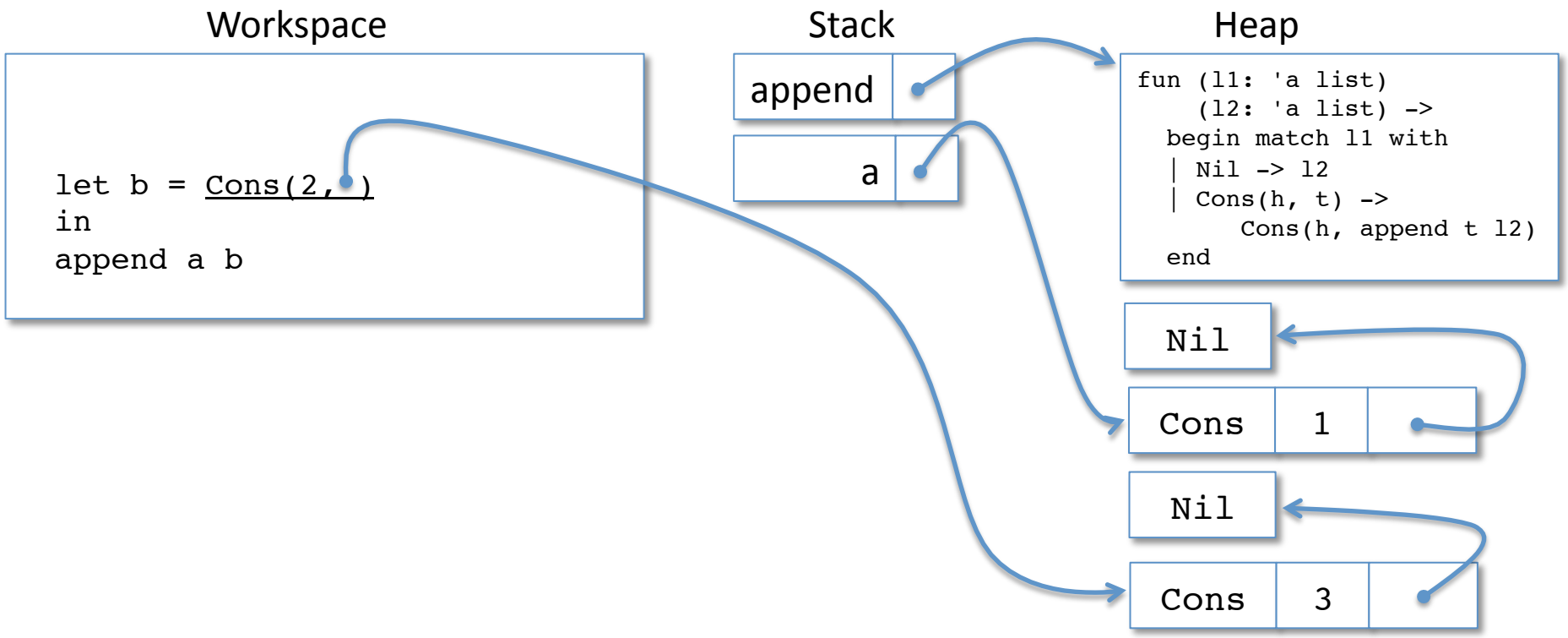
# Allocate a Cons cell



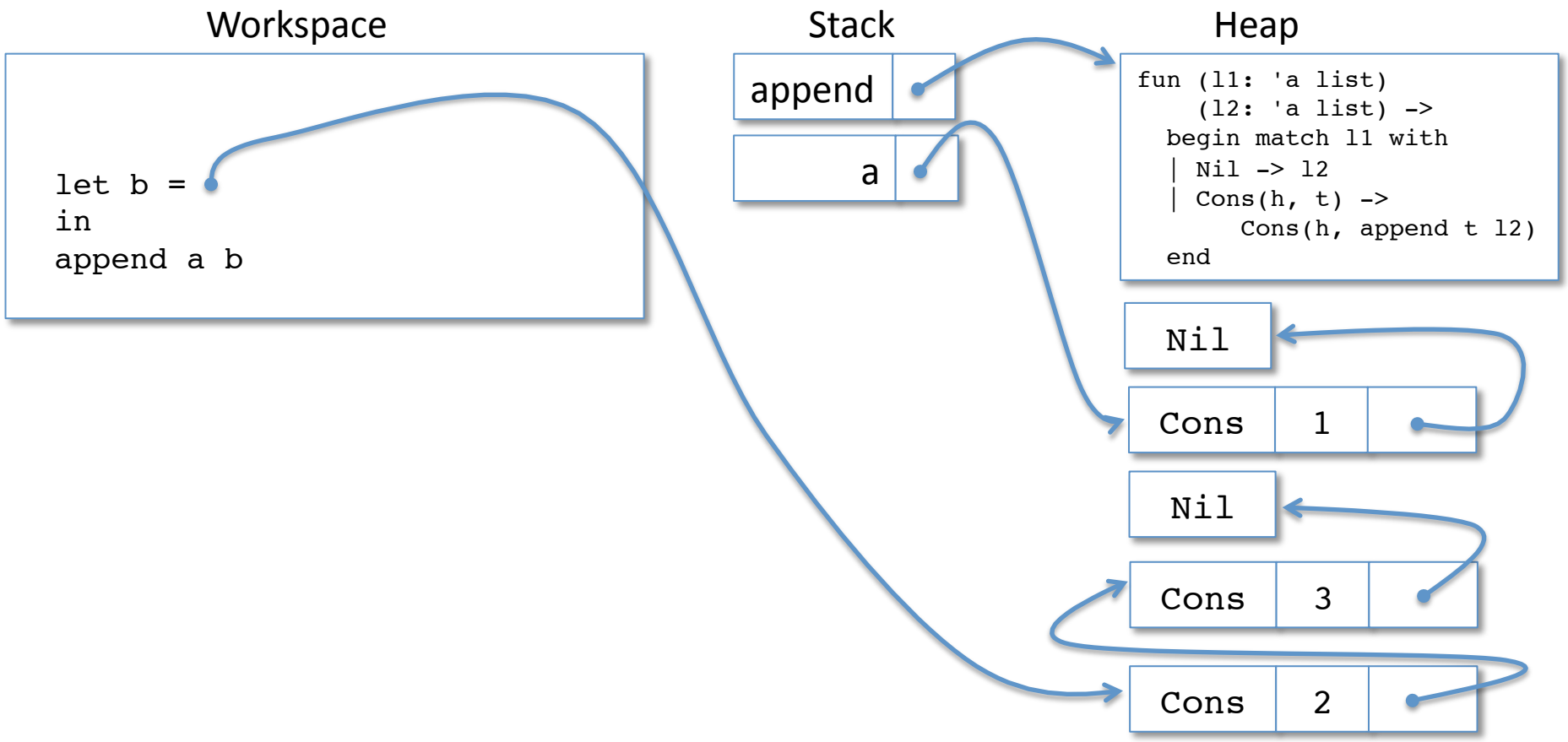
# Allocate a Cons cell



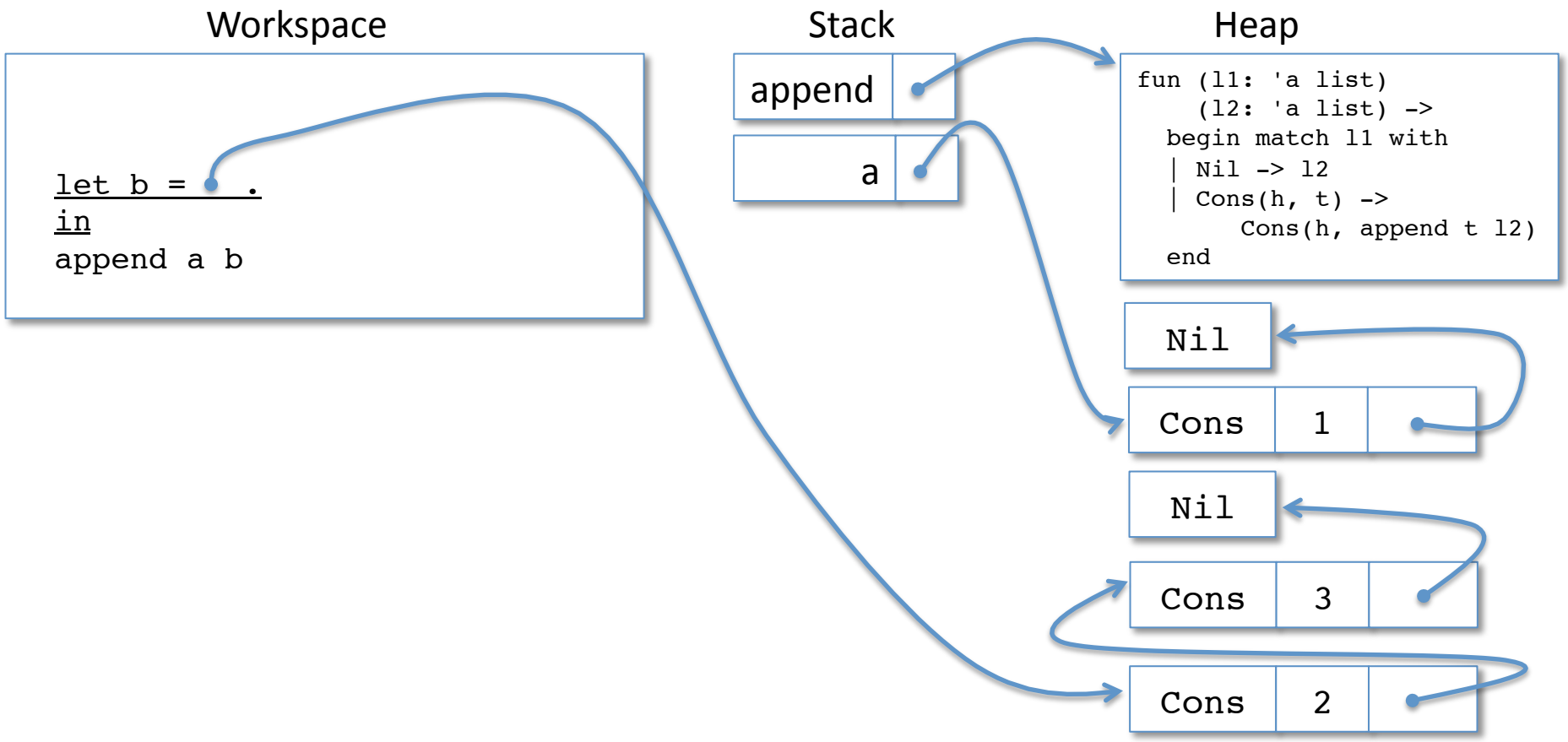
# Allocate a Cons cell



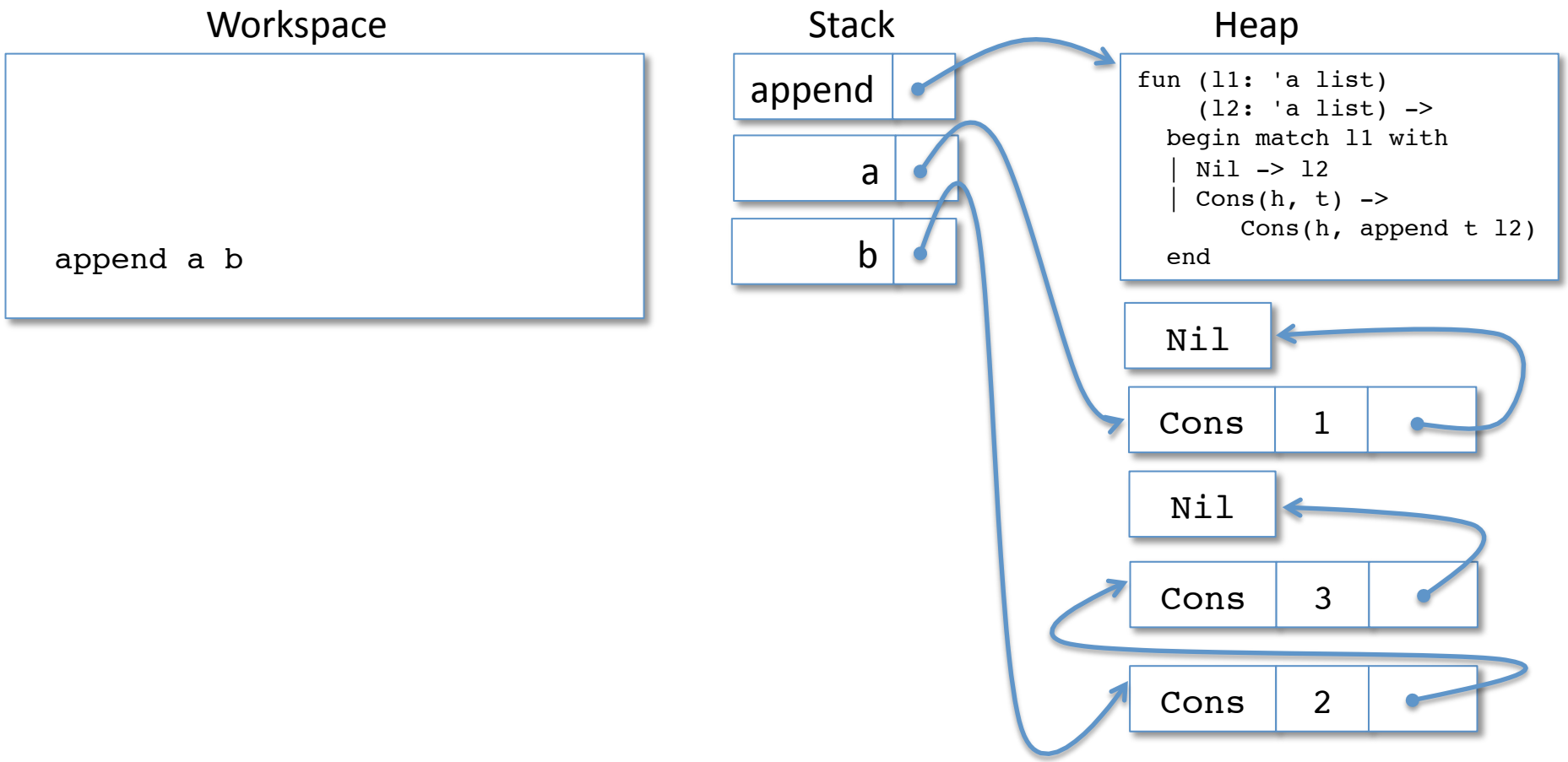
# Allocate a Cons cell



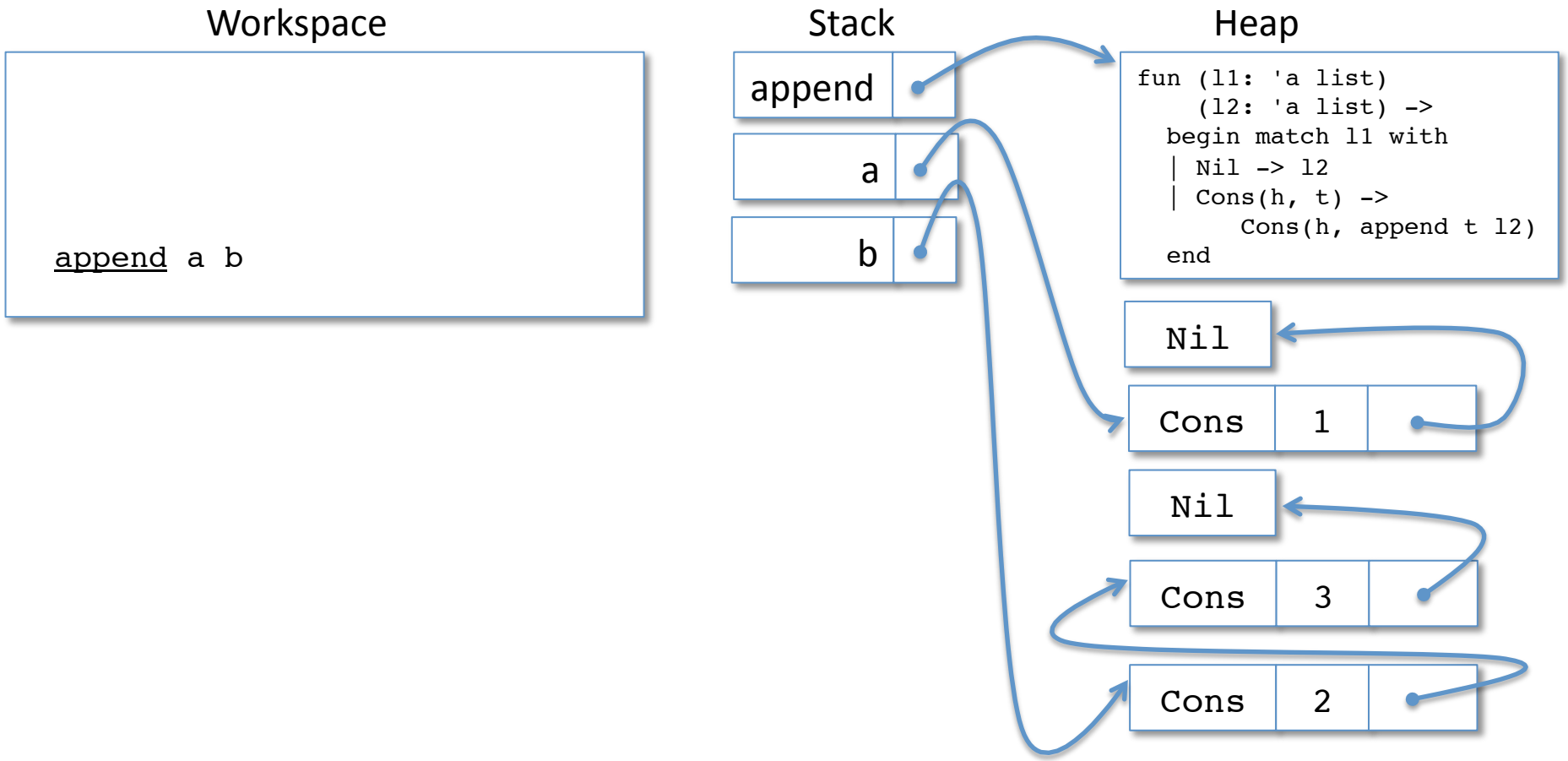
# Let Expression



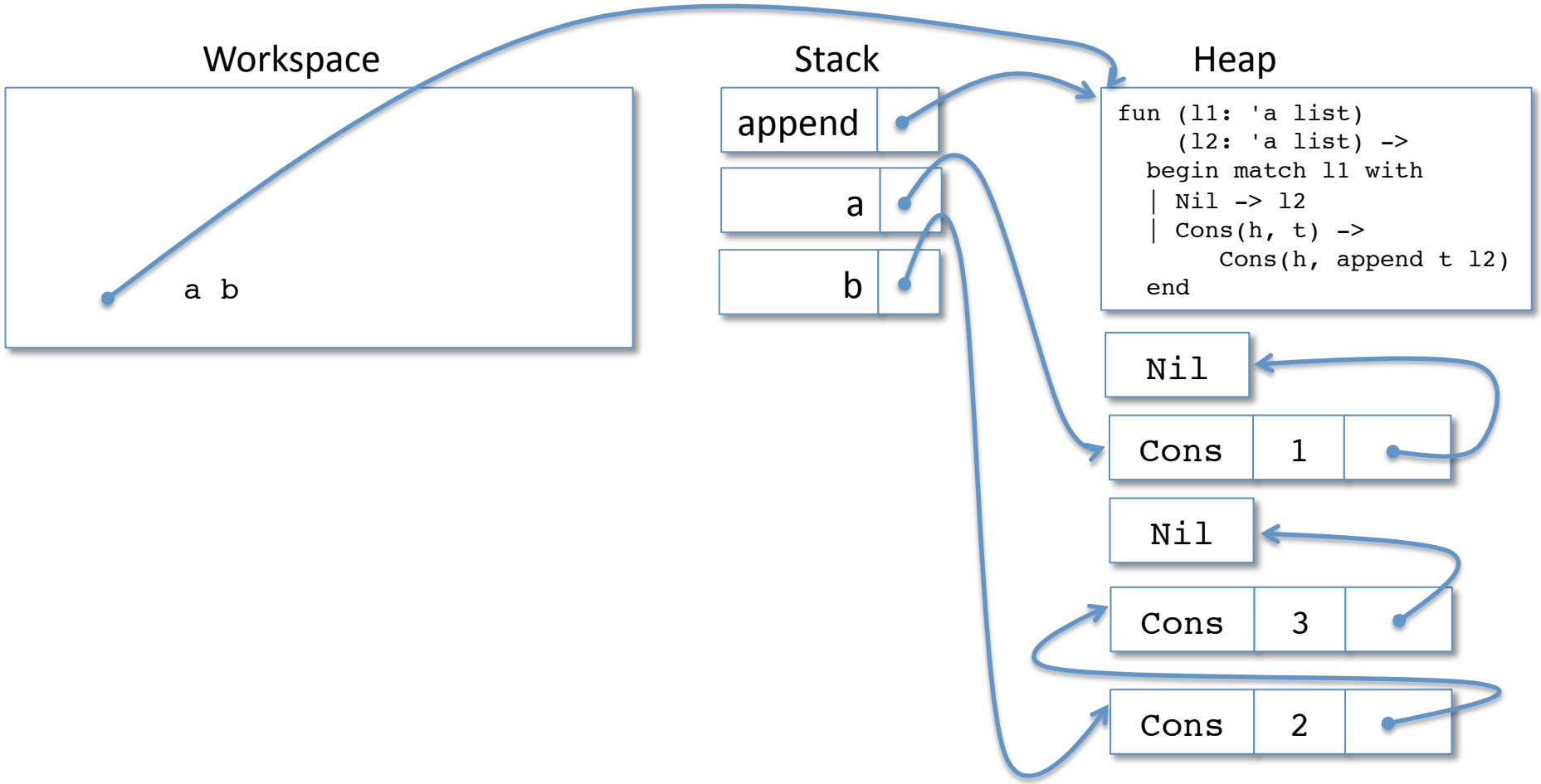
# Create a Stack Binding



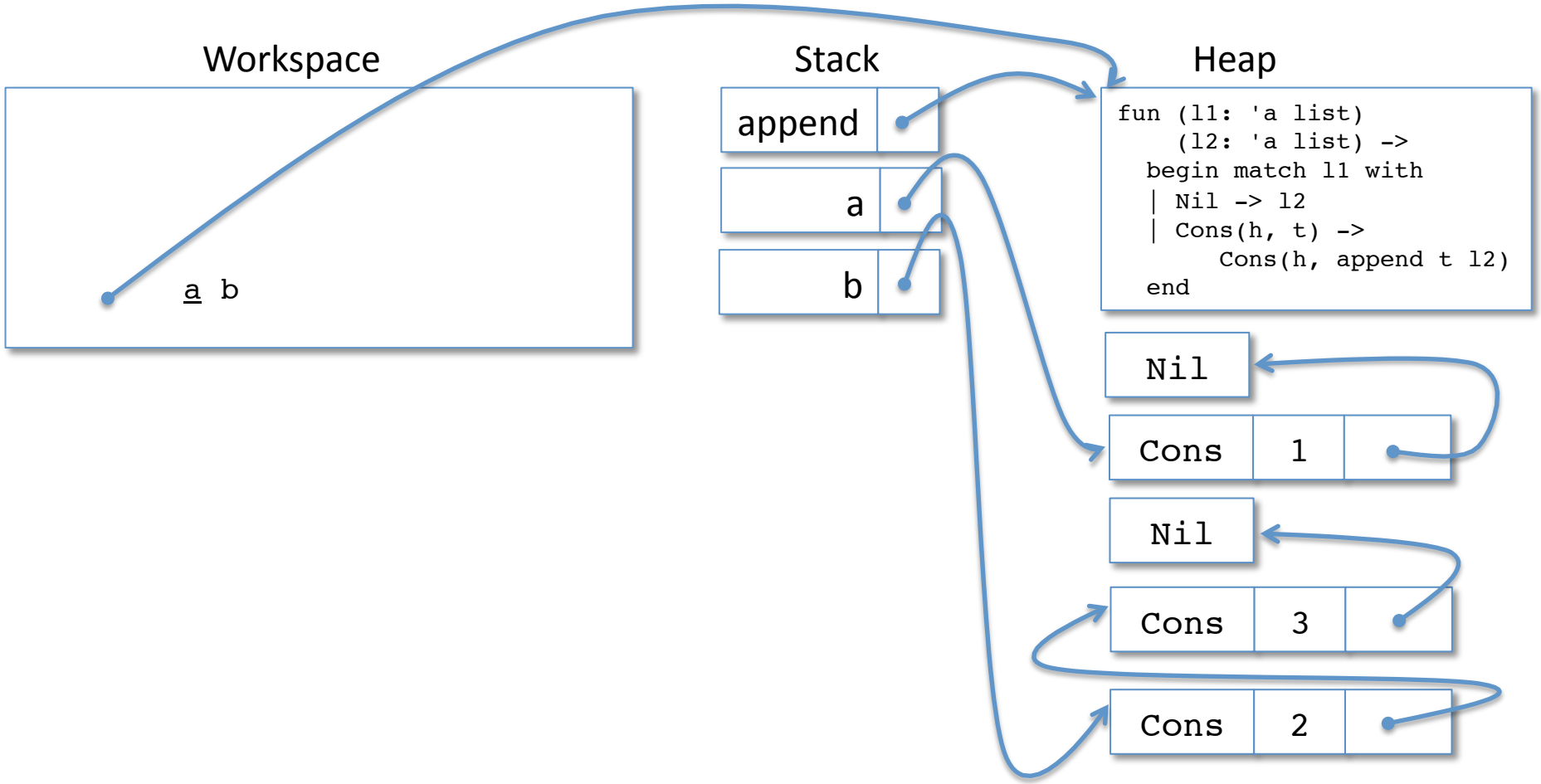
# Lookup ‘append’



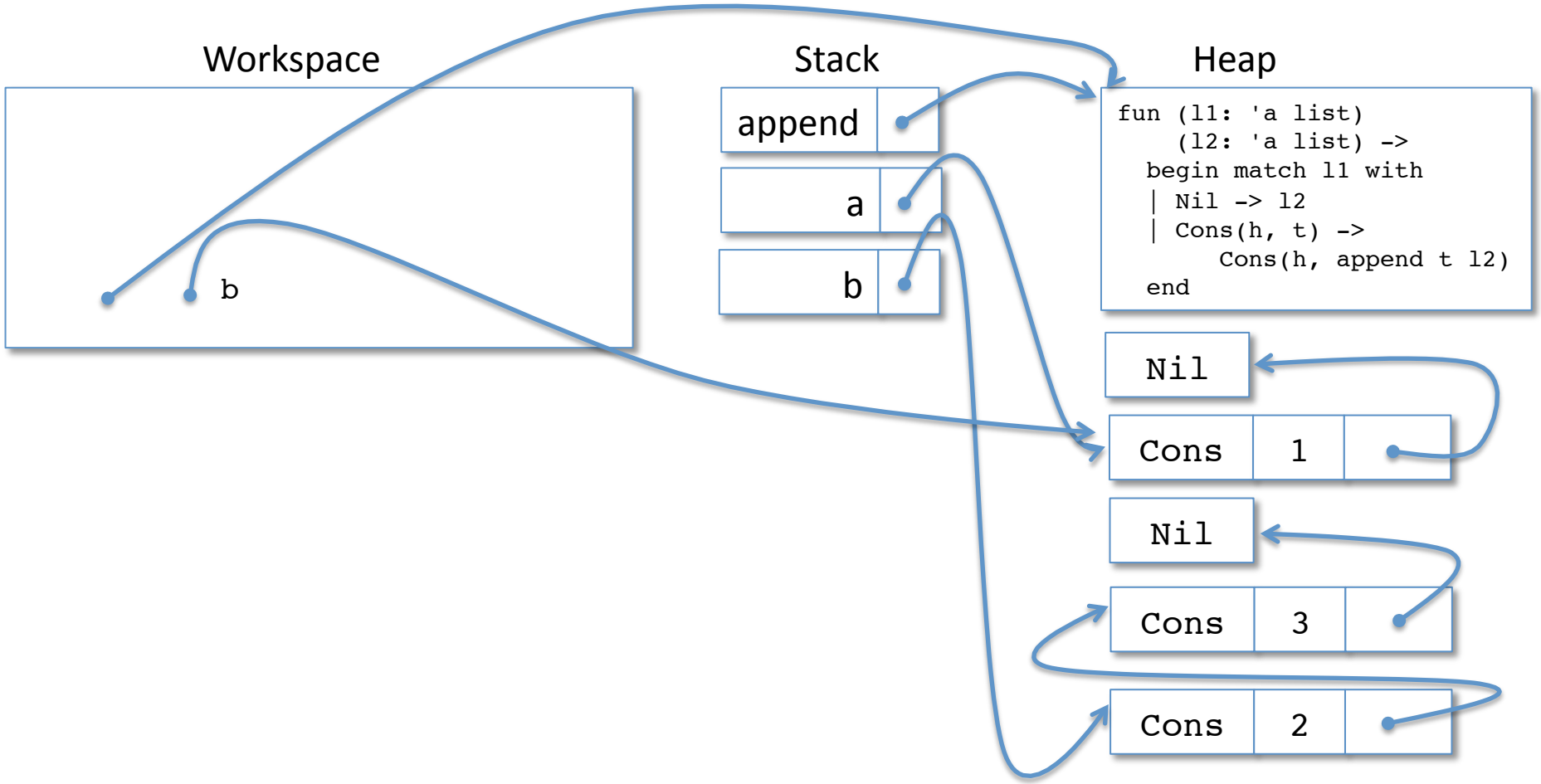
# Lookup ‘append’



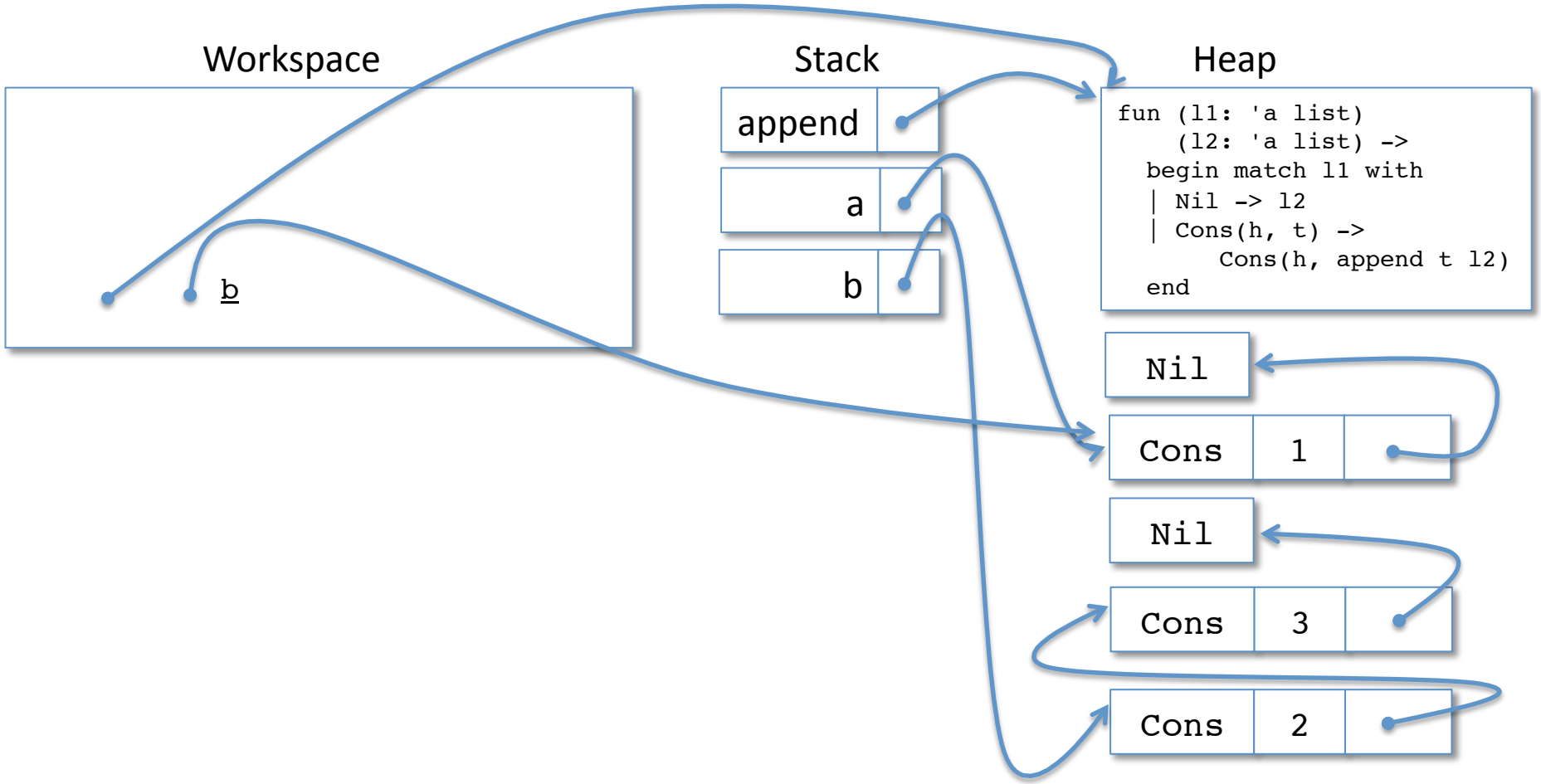
# Lookup ‘a’


Workspace
a b
Stack
append
a
b
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

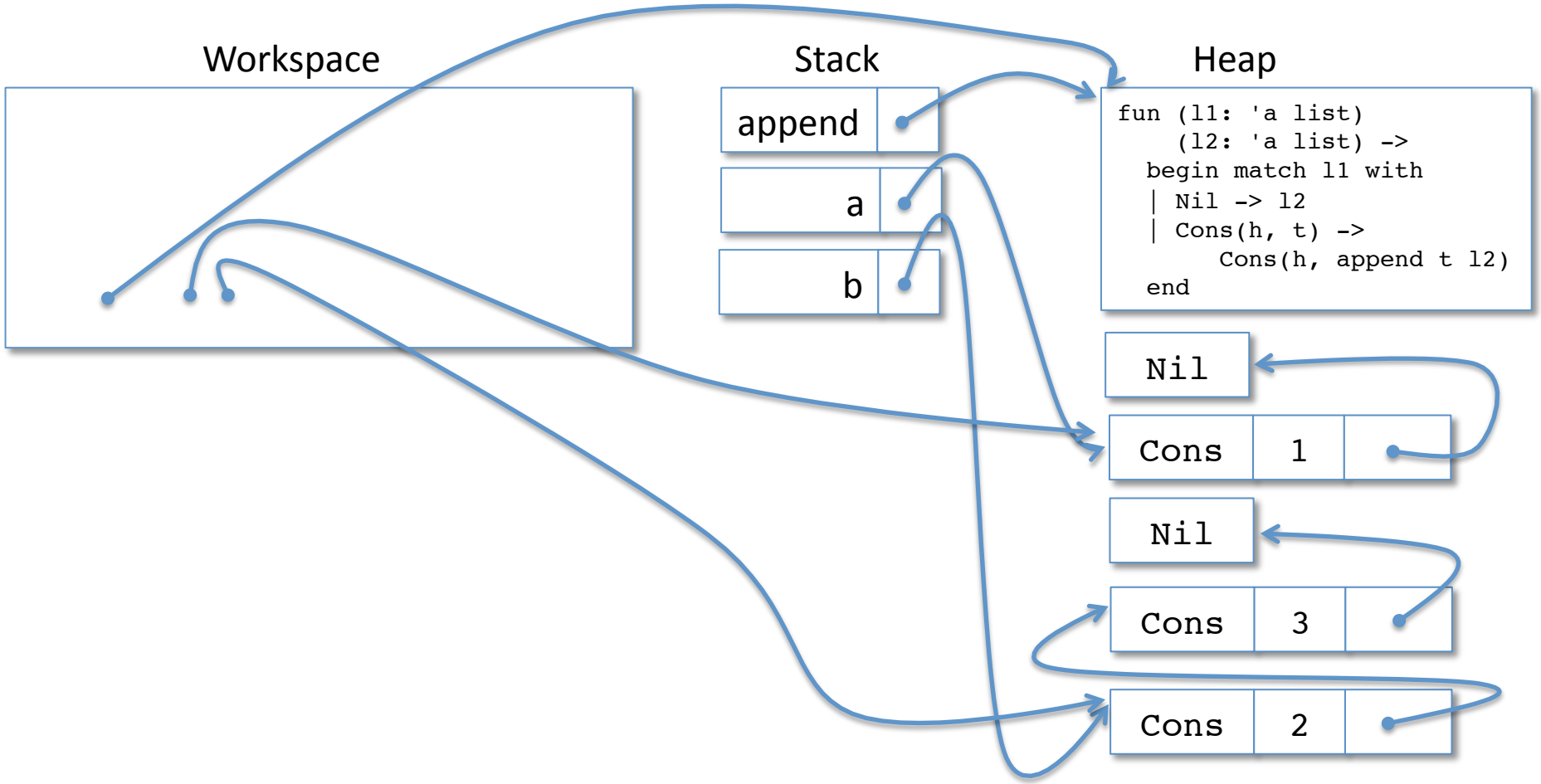
# Lookup ‘a’


Workspace
b
Stack
append
a
b
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons
1
Nil
Cons
3
Cons
2

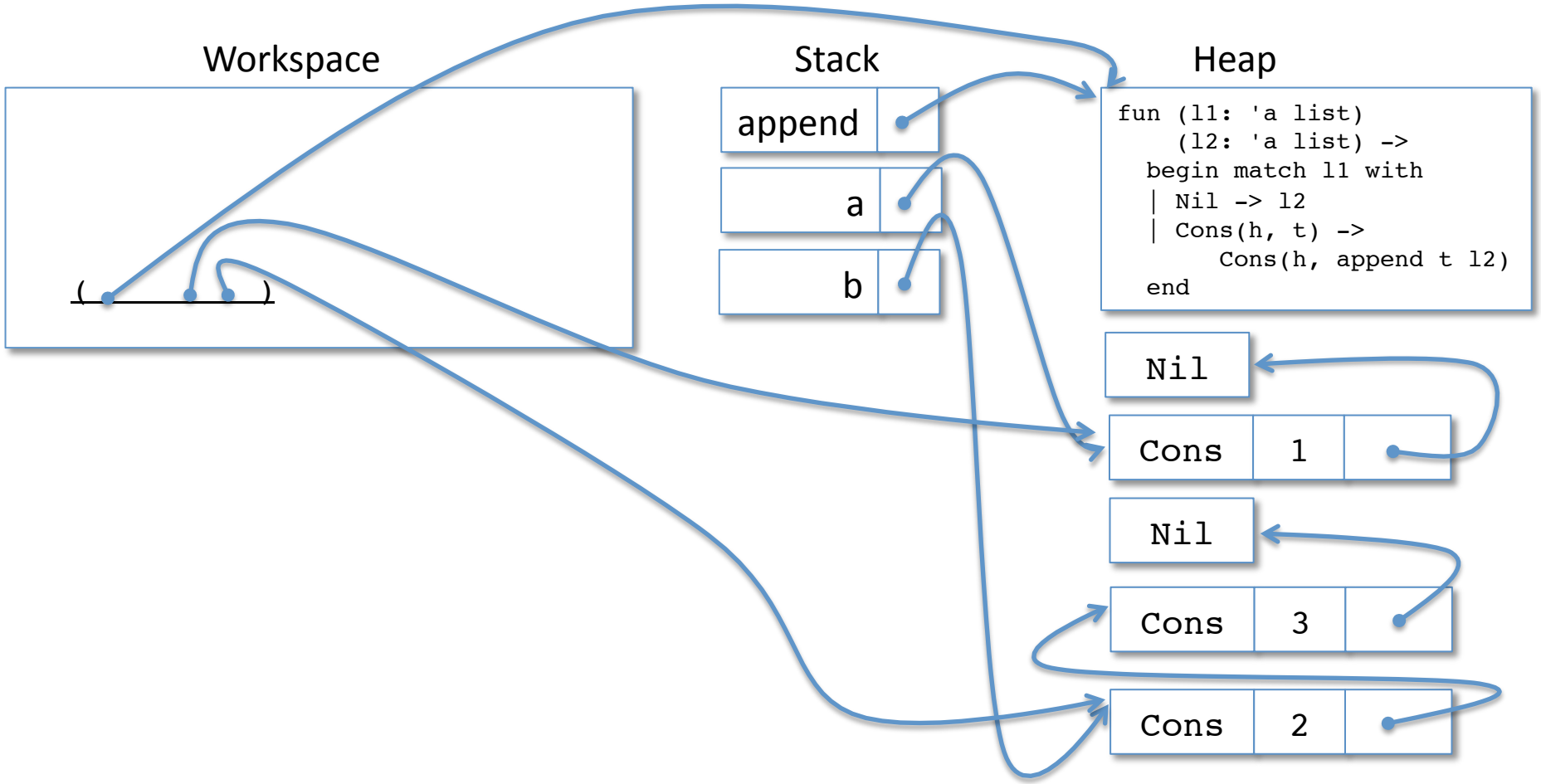
# Lookup ‘b’


Workspace
b
Stack
append
a
b
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons
1
Nil
Cons
3
Cons
2

# Lookup ‘b’


Workspace
Stack
Heap
append
a
b
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

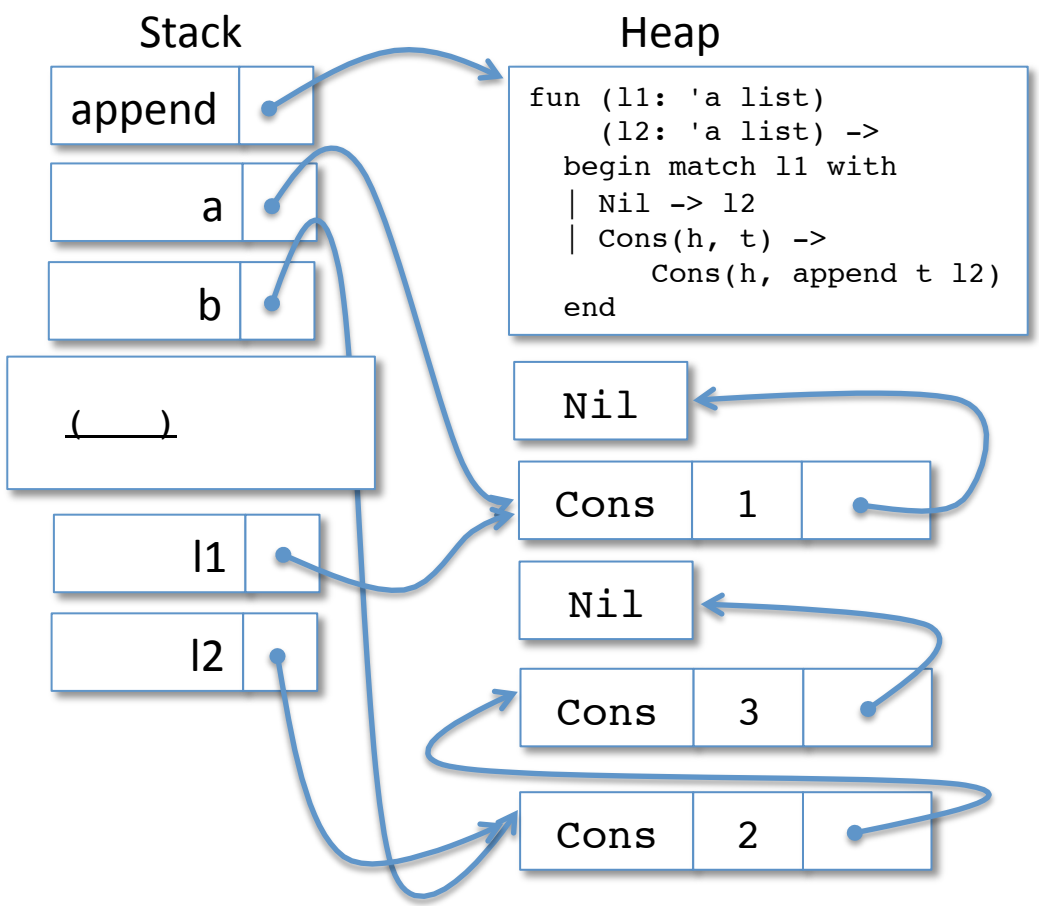
# Do the Function call



# Save Workspace; push l1, l2

Workspace

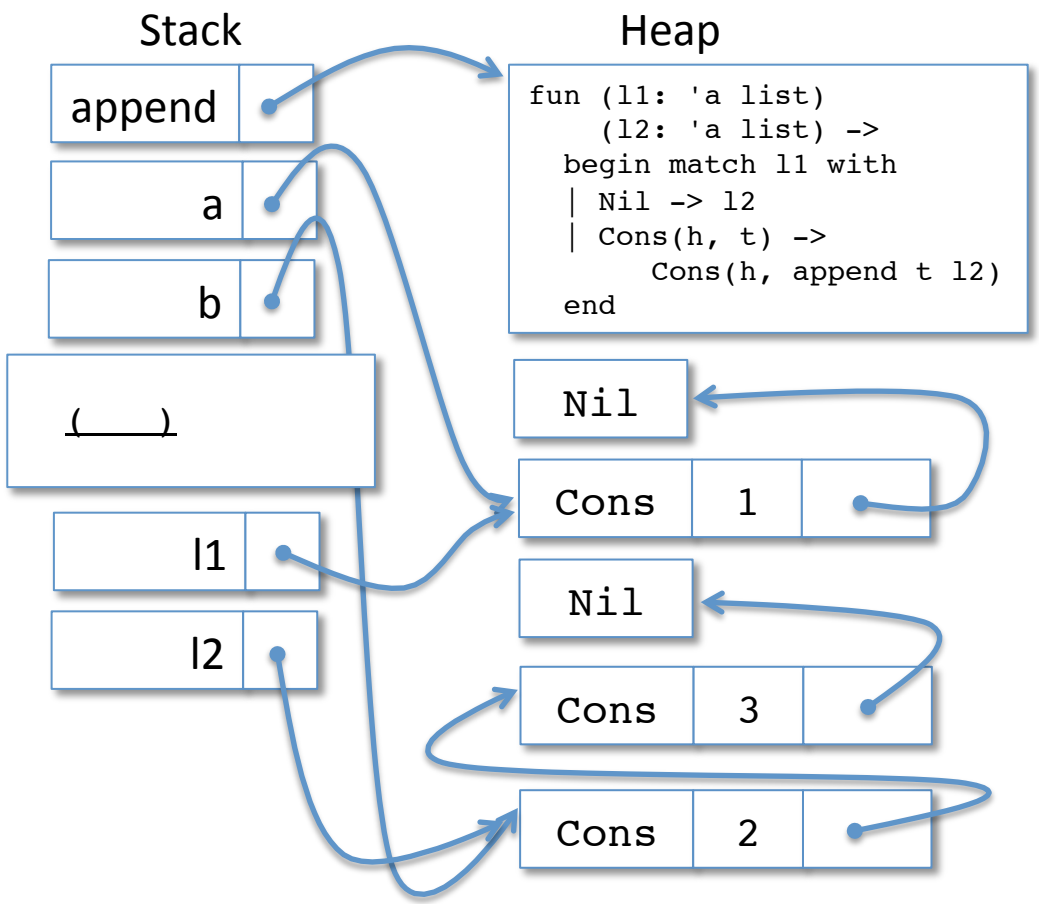
```
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
      Cons(h, append t l2)
  end
```



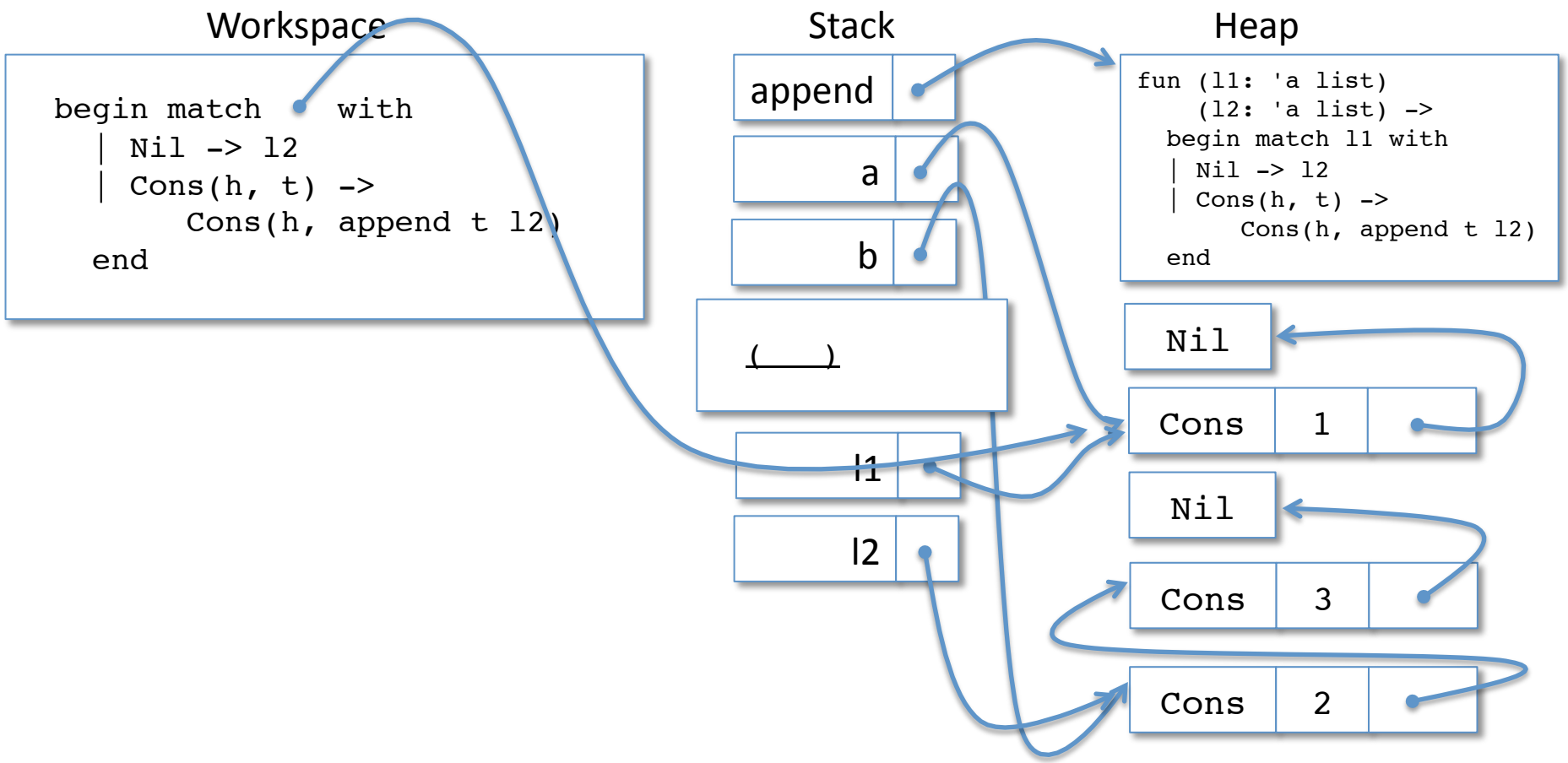
# Lookup l1

Workspace

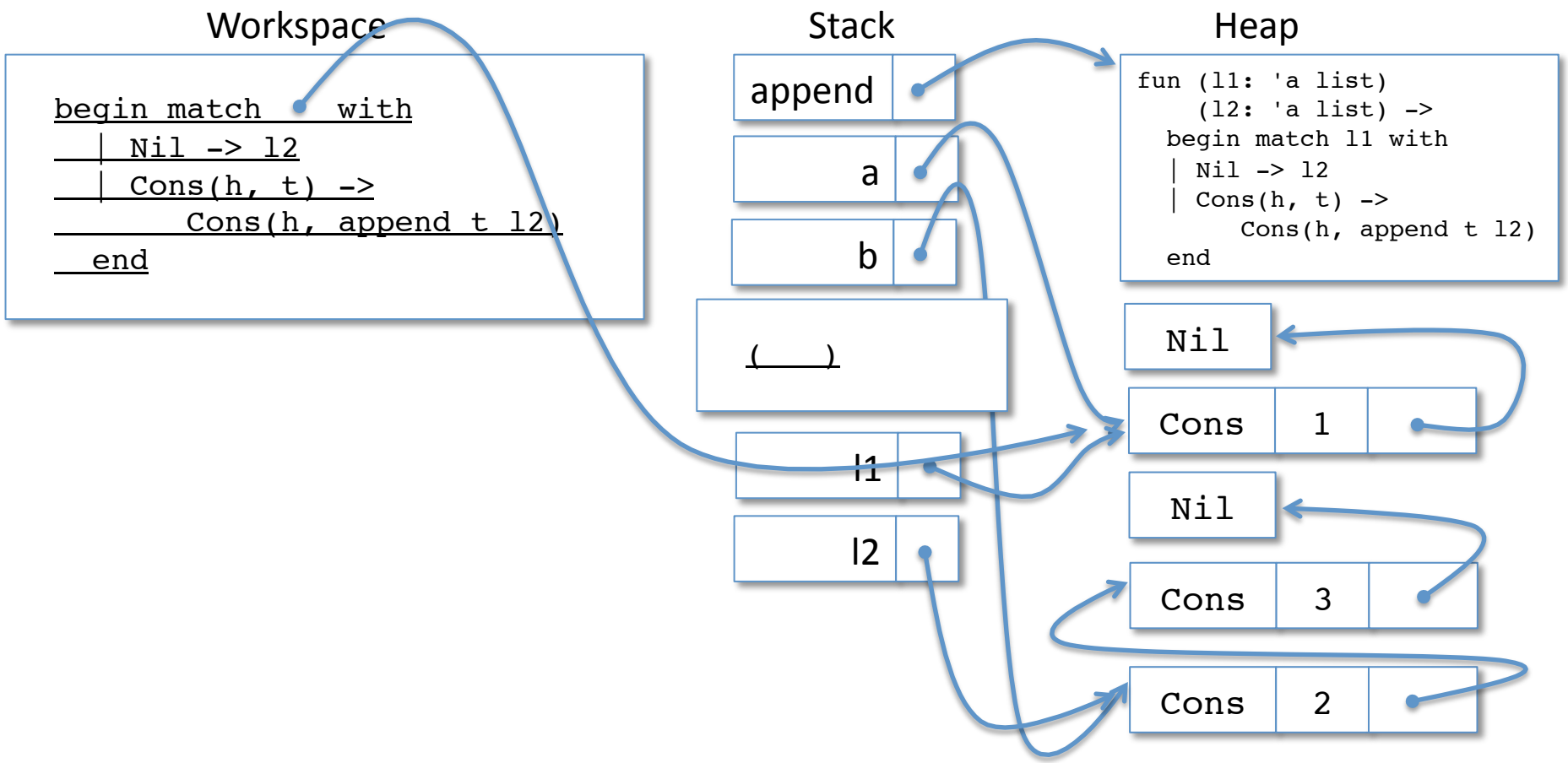
```
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
      Cons(h, append t l2)
  end
```



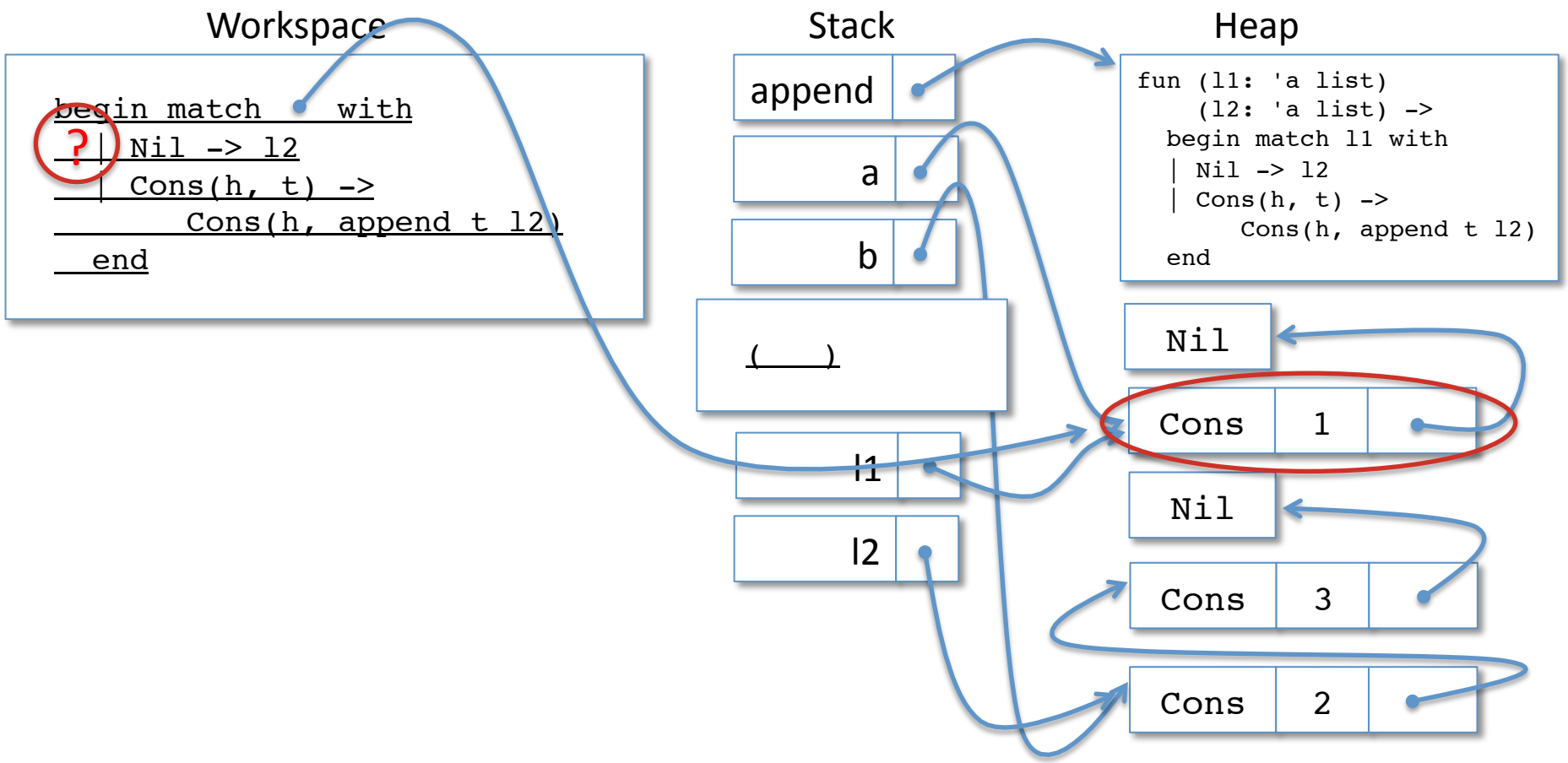
# Lookup l1


Workspace
begin match with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Stack
append
a
b
( )
l1
l2
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

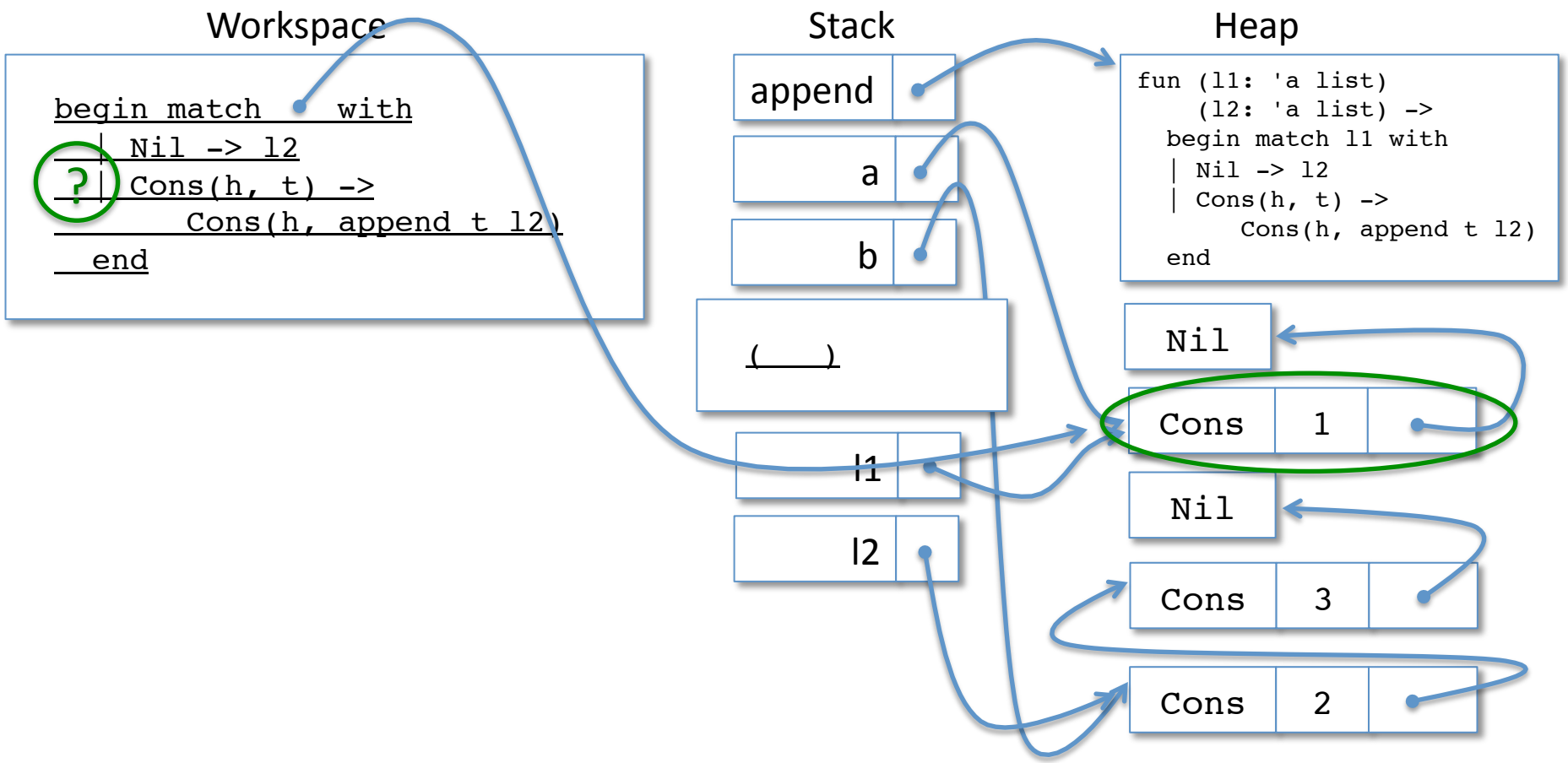
# Match Expression



# Nil case Doesn't Match



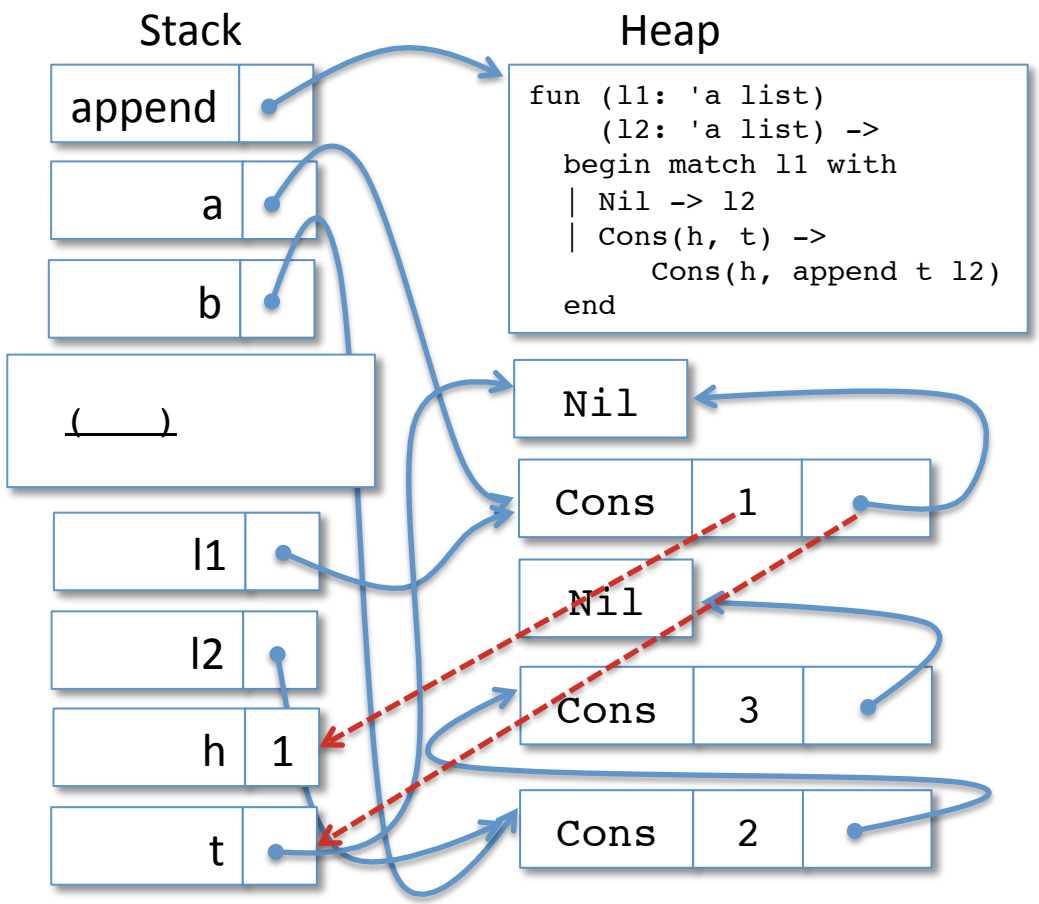
# Cons case *Does* Match



# Simplify the Branch: push h, t

Workspace

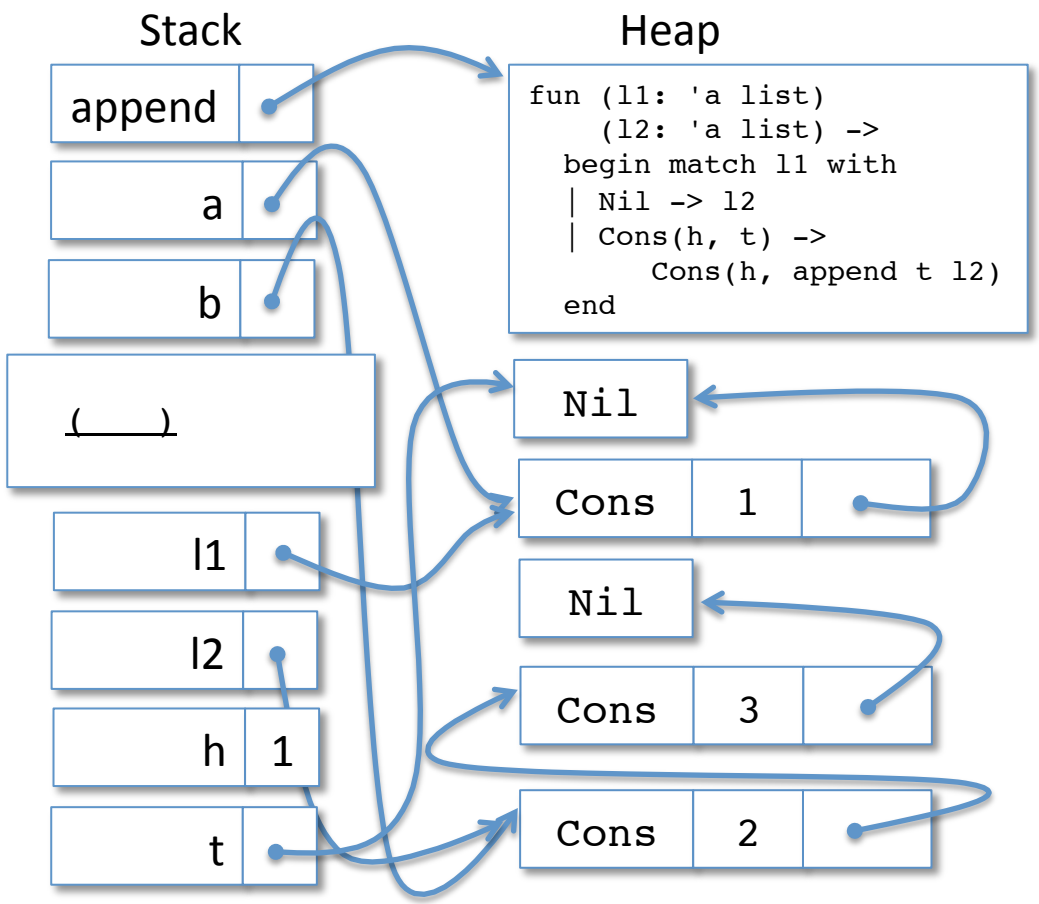
```
Cons(h, append t l2)
```



# Lookup ‘h’

Workspace

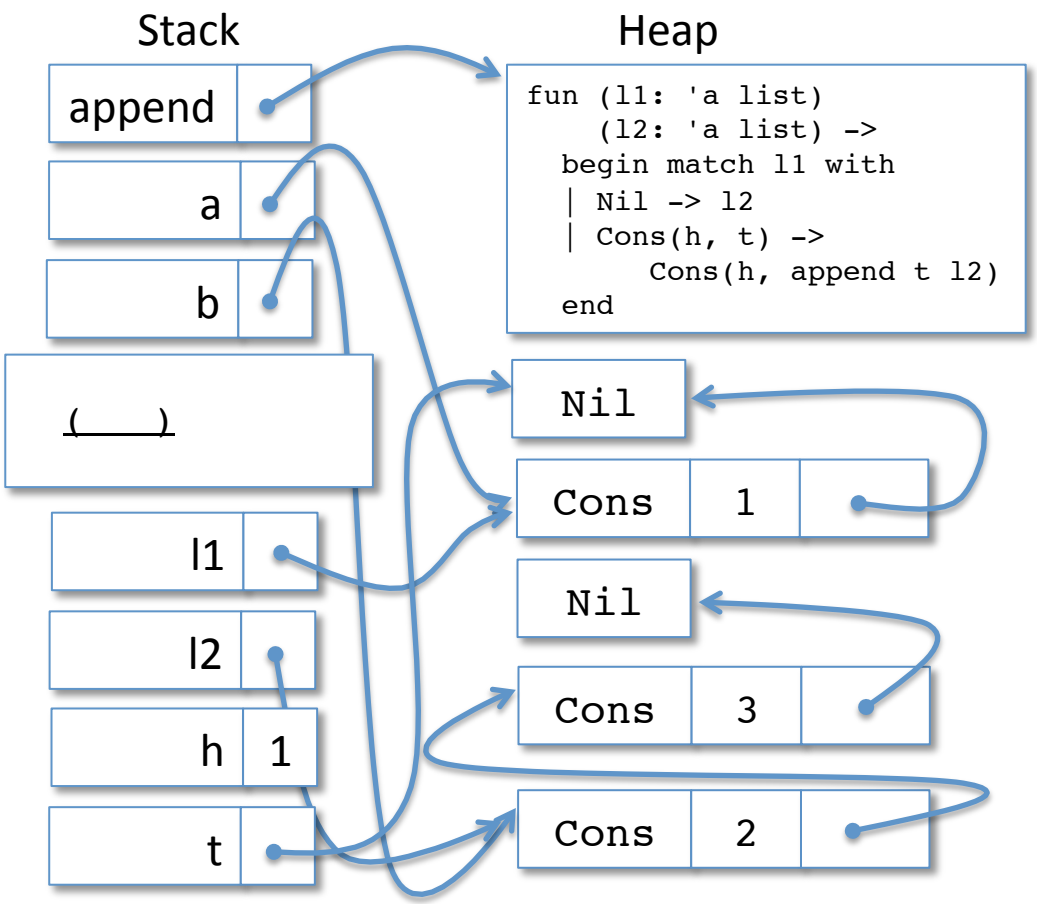
```
Cons(h, append t l2)
```



# Lookup ‘h’

Workspace

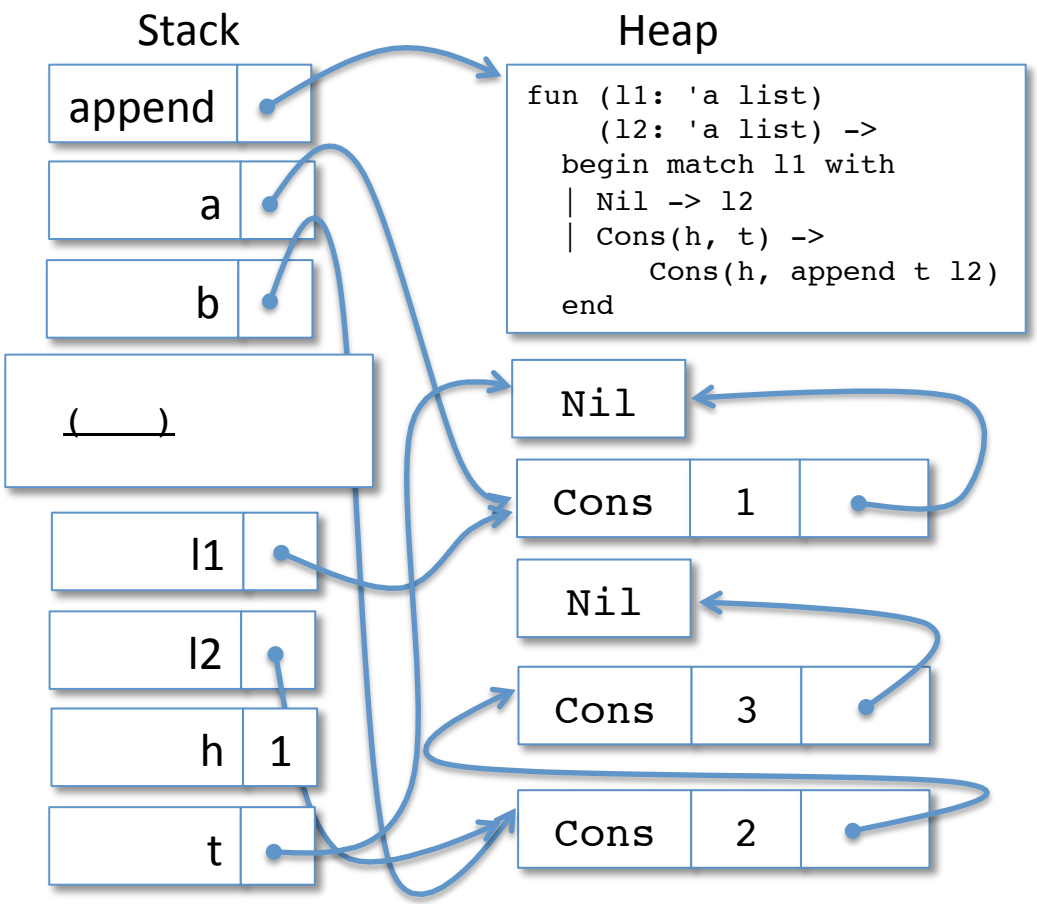
```
Cons(1, append t l2)
```



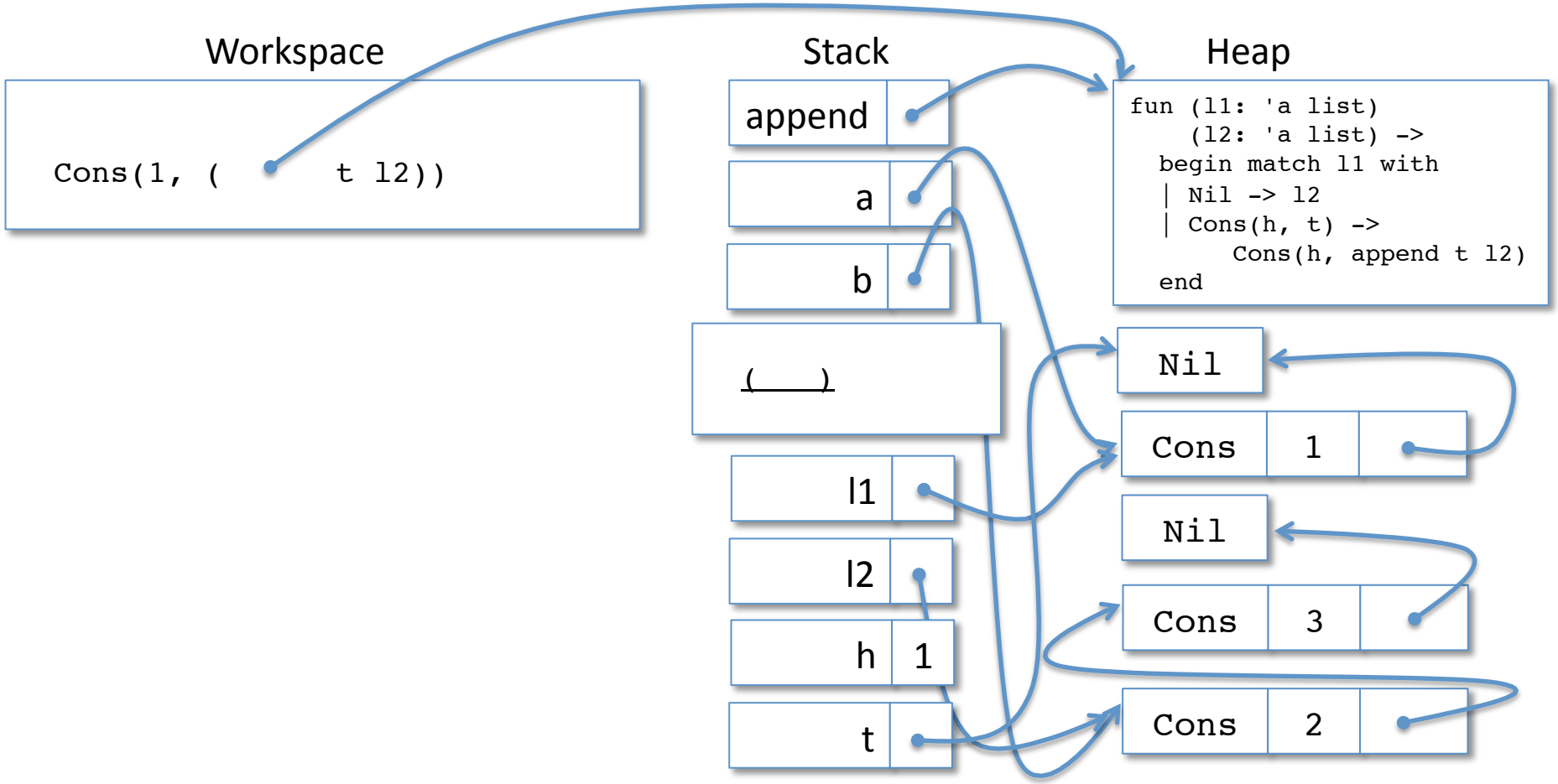
# Lookup ‘append’

Workspace

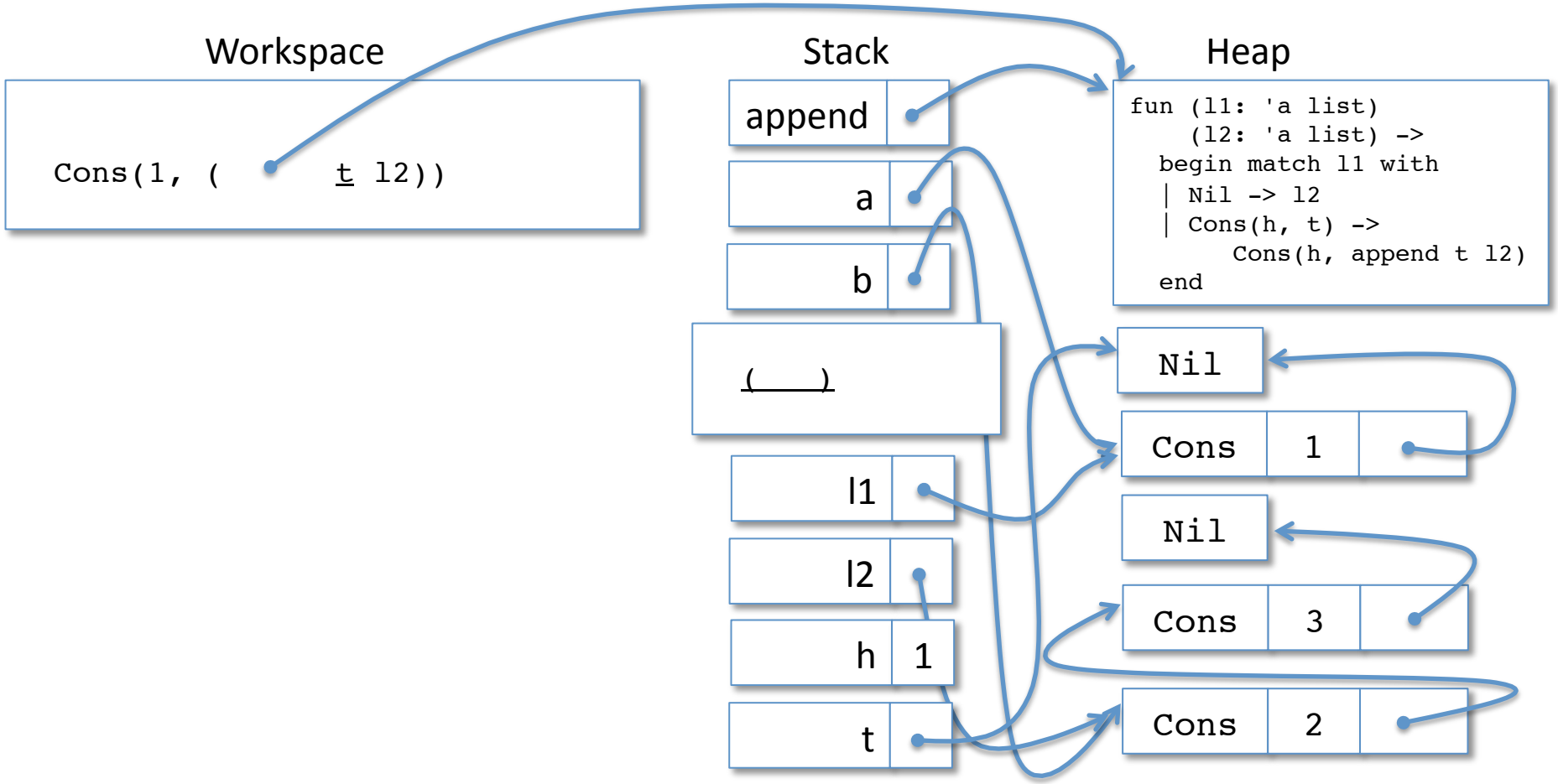
```
Cons(1, append t l2)
```



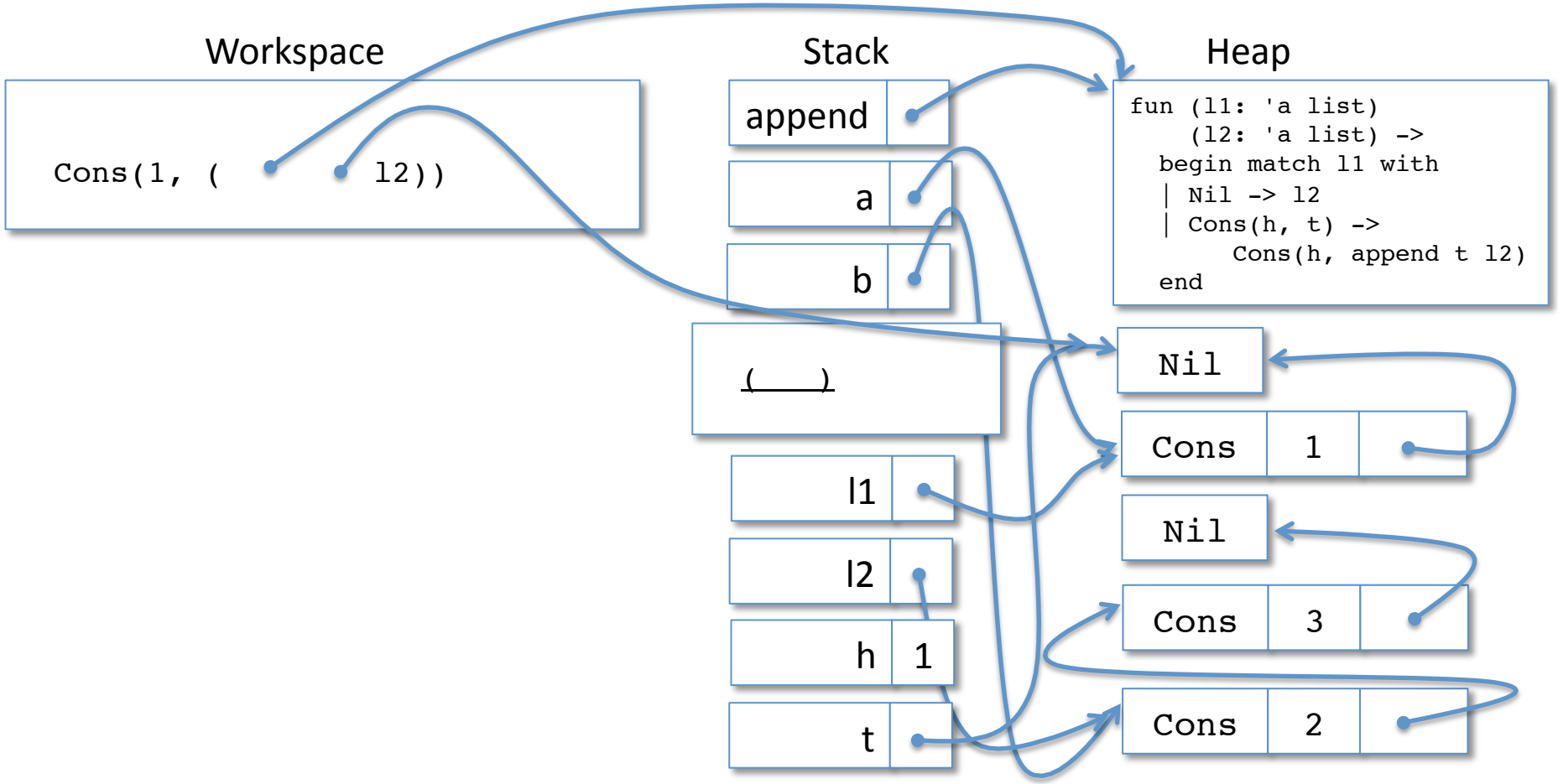
# Lookup ‘append’



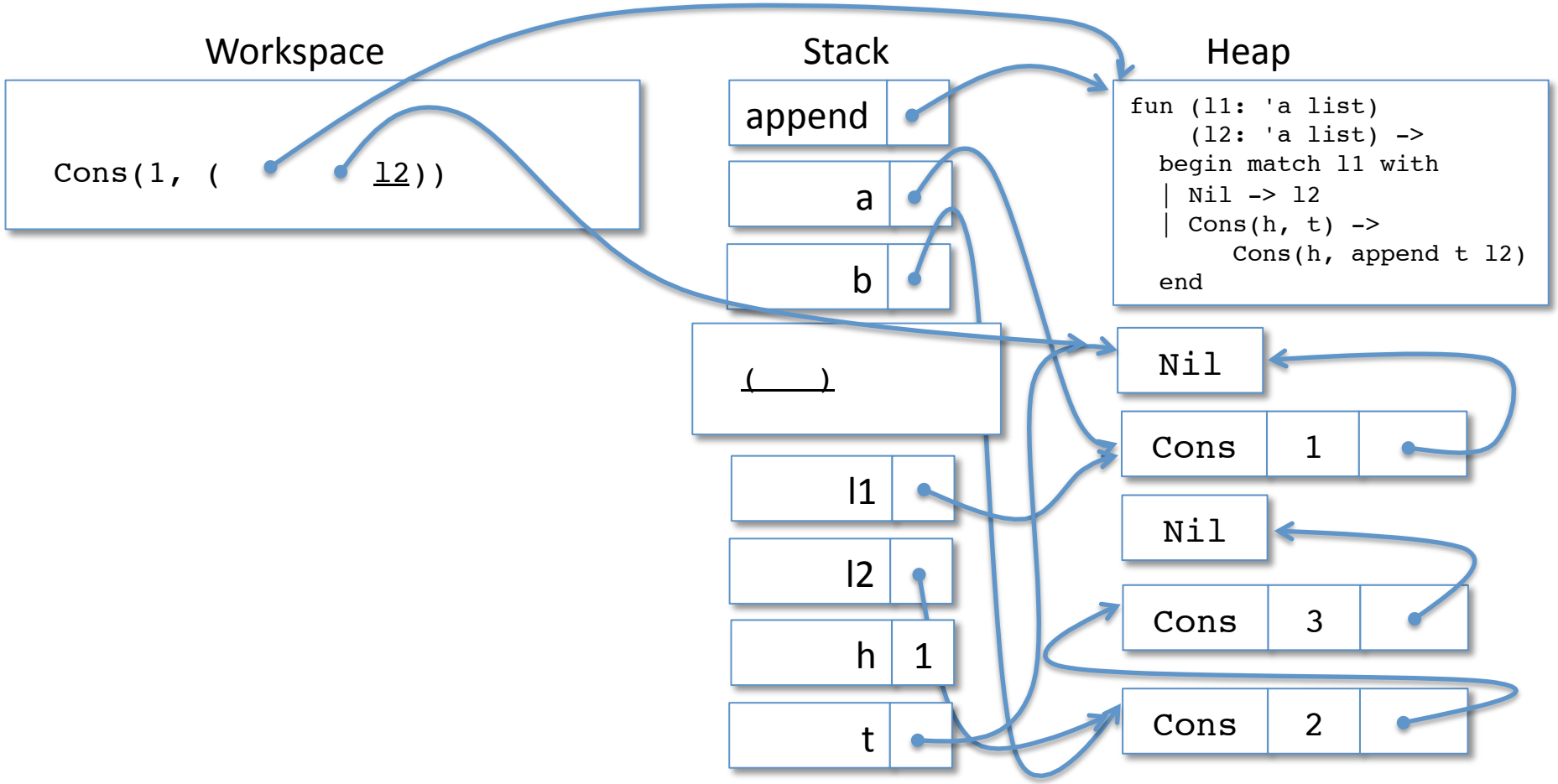
# Lookup ‘t’


Workspace
Cons(1, ( t l2))
Stack
append
a
b
( )
l1
l2
h 1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

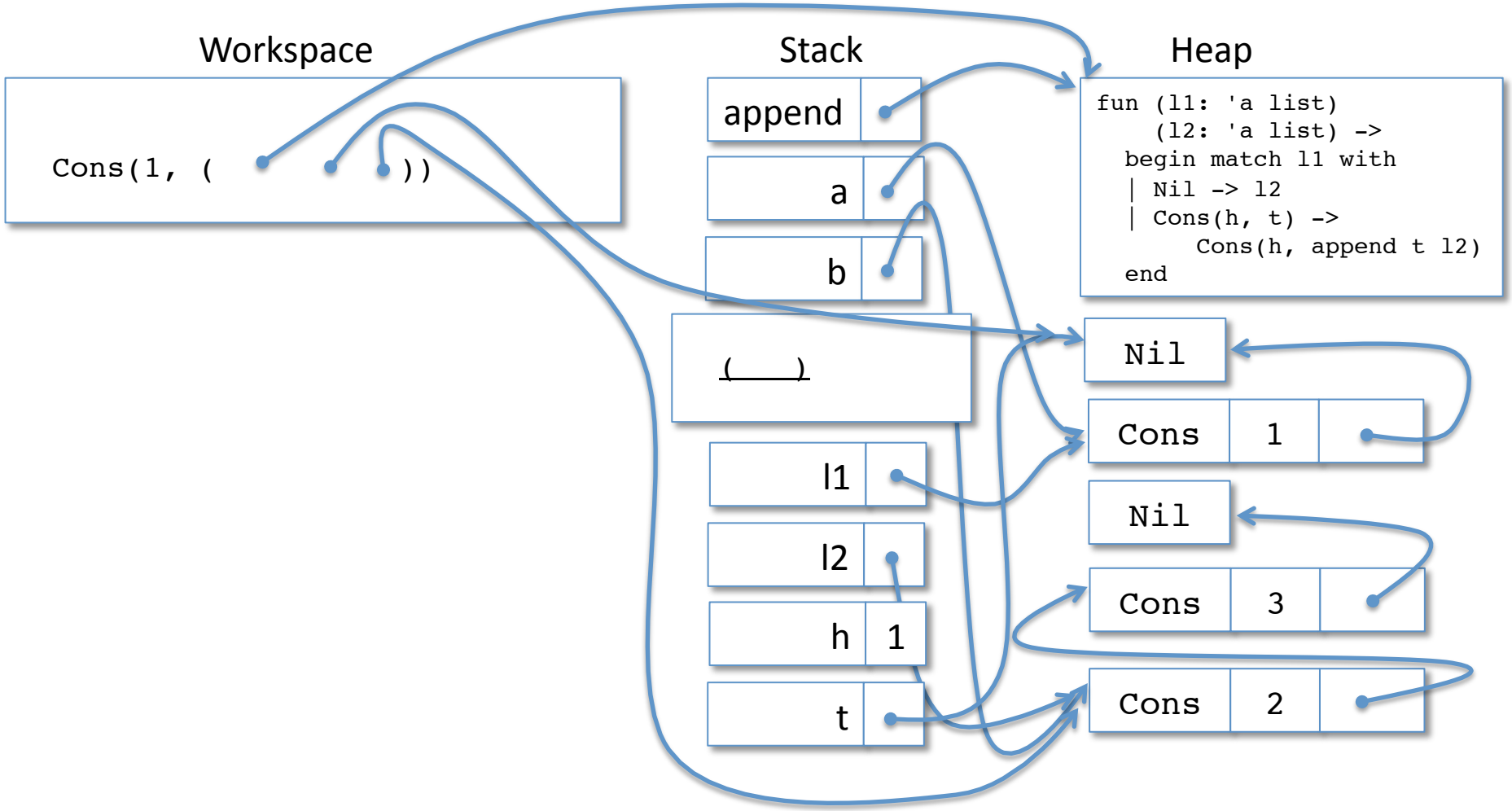
# Lookup ‘t’


Workspace
Cons(1, (
l2))
Stack
append
a
b
( )
l1
l2
h
1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons
1
Nil
Cons
3
Cons
2

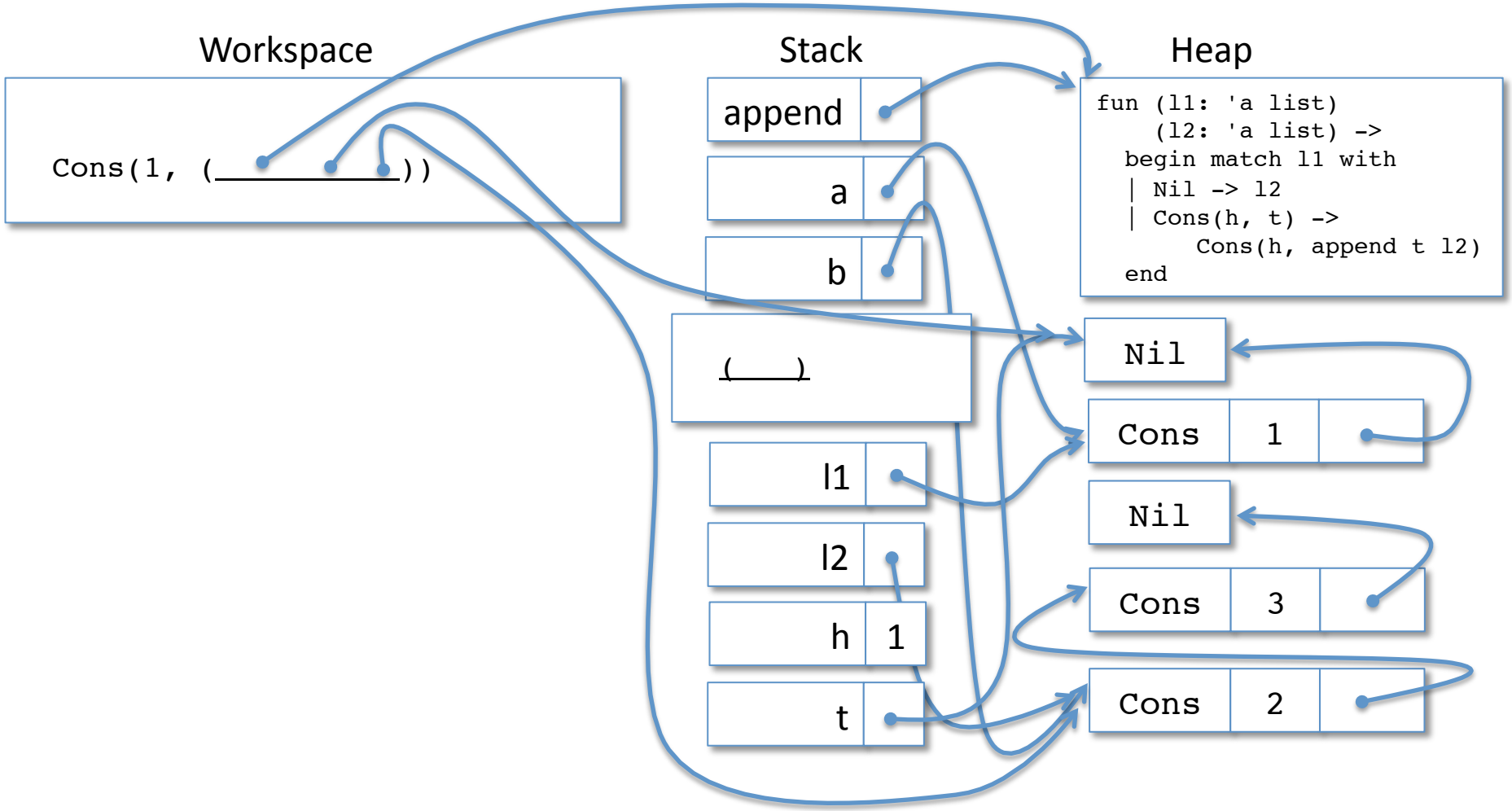
# Lookup ‘l2’


Workspace
Cons(1, ( l2))
Stack
append
a
b
( )
l1
l2
h 1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

# Lookup ‘l2’


Workspace
Cons(1, ( ))
Stack
append
a
b
( )
l1
l2
h 1
t
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

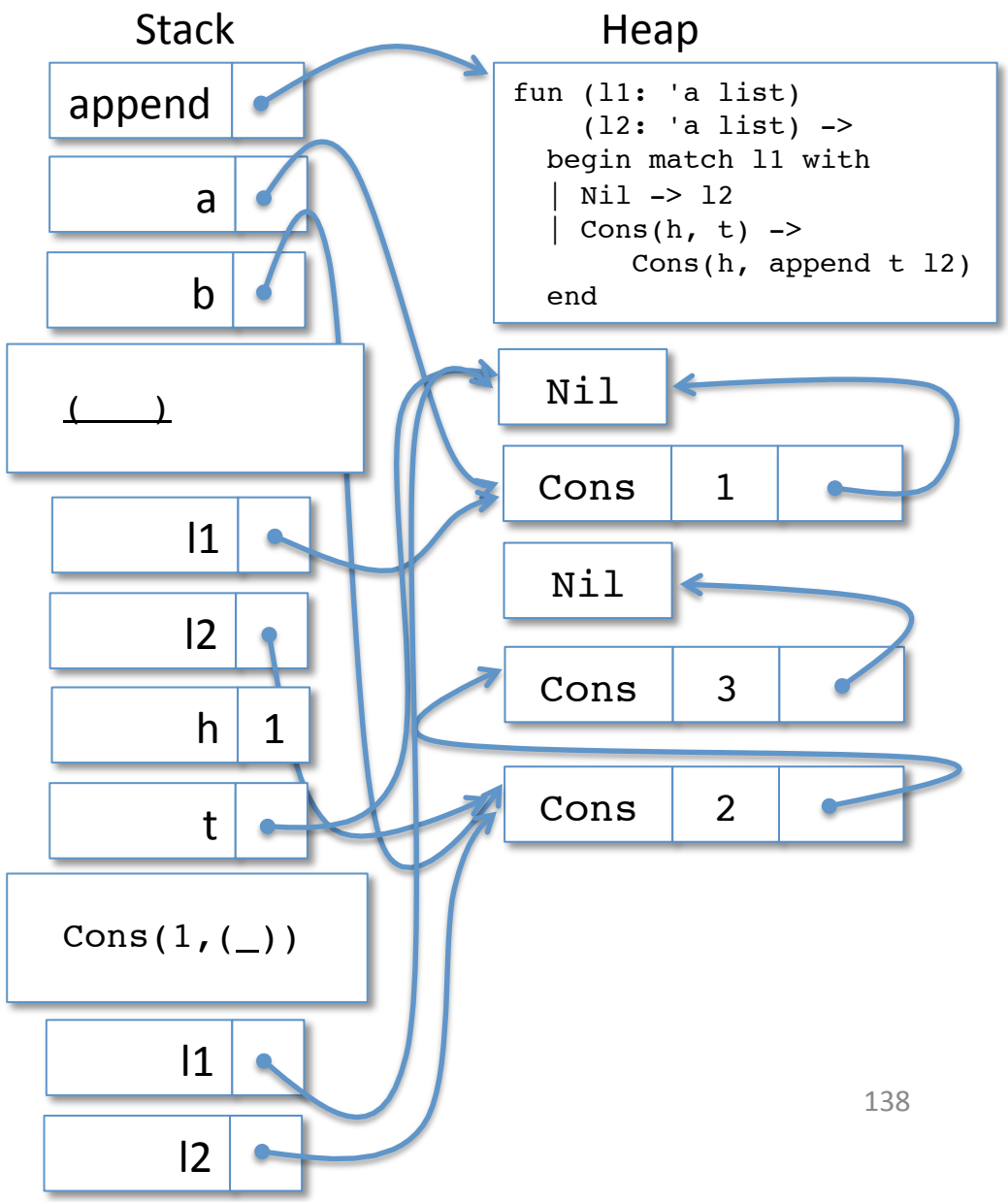
# Do the Function Call



# Save the Workspace; push l1, l2

Workspace

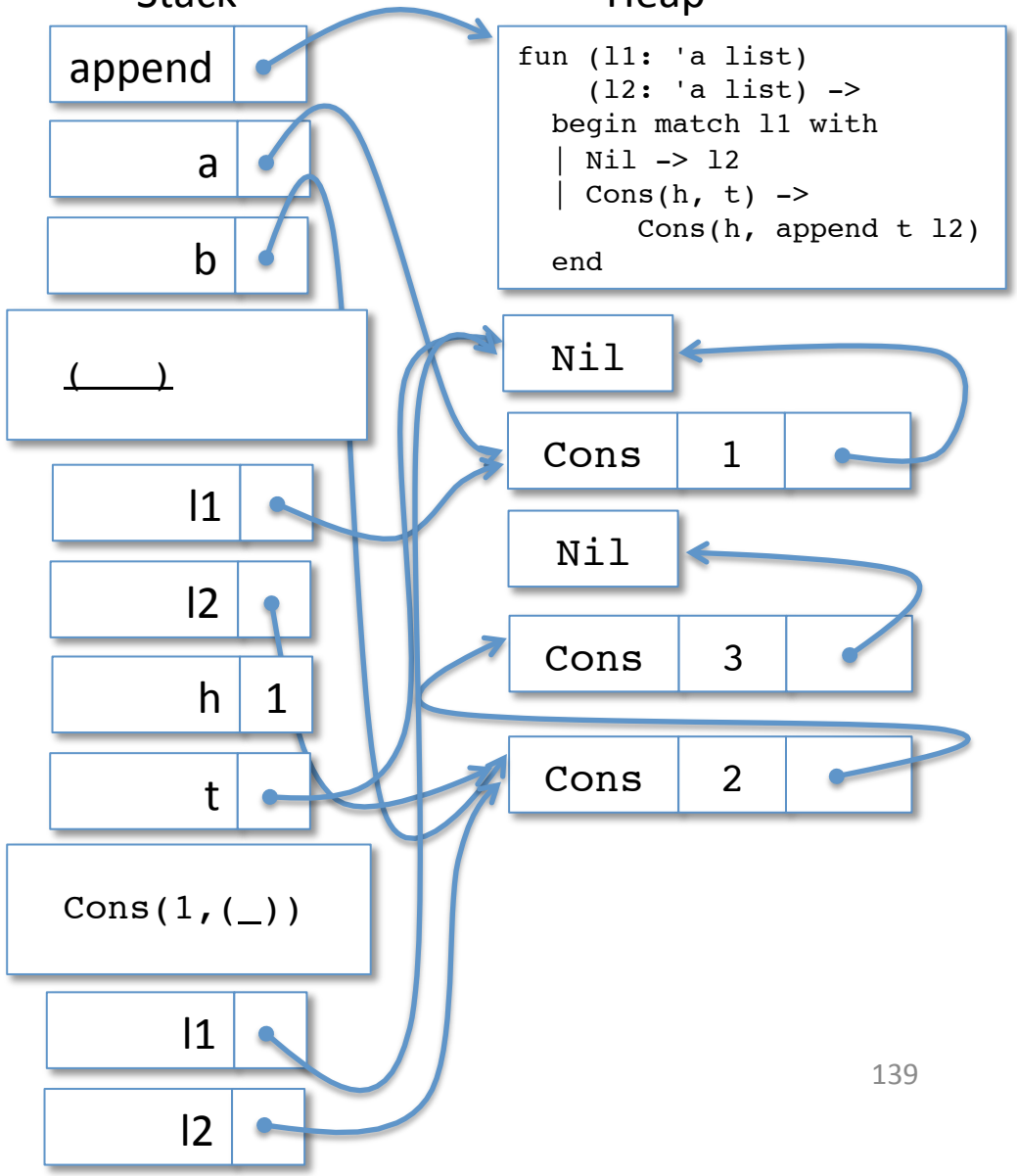
```
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
 end
```



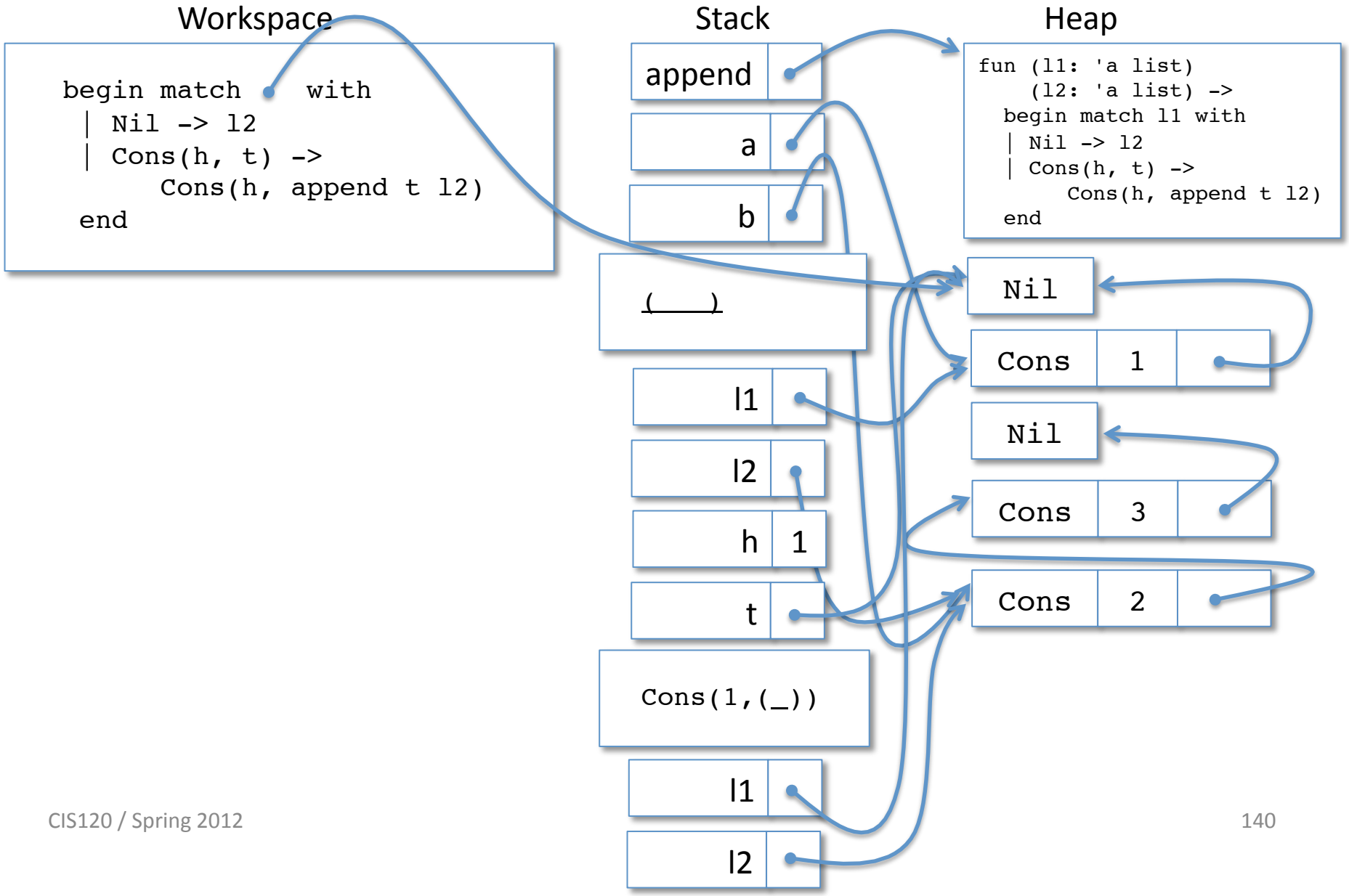
# Lookup ‘l1’

Workspace

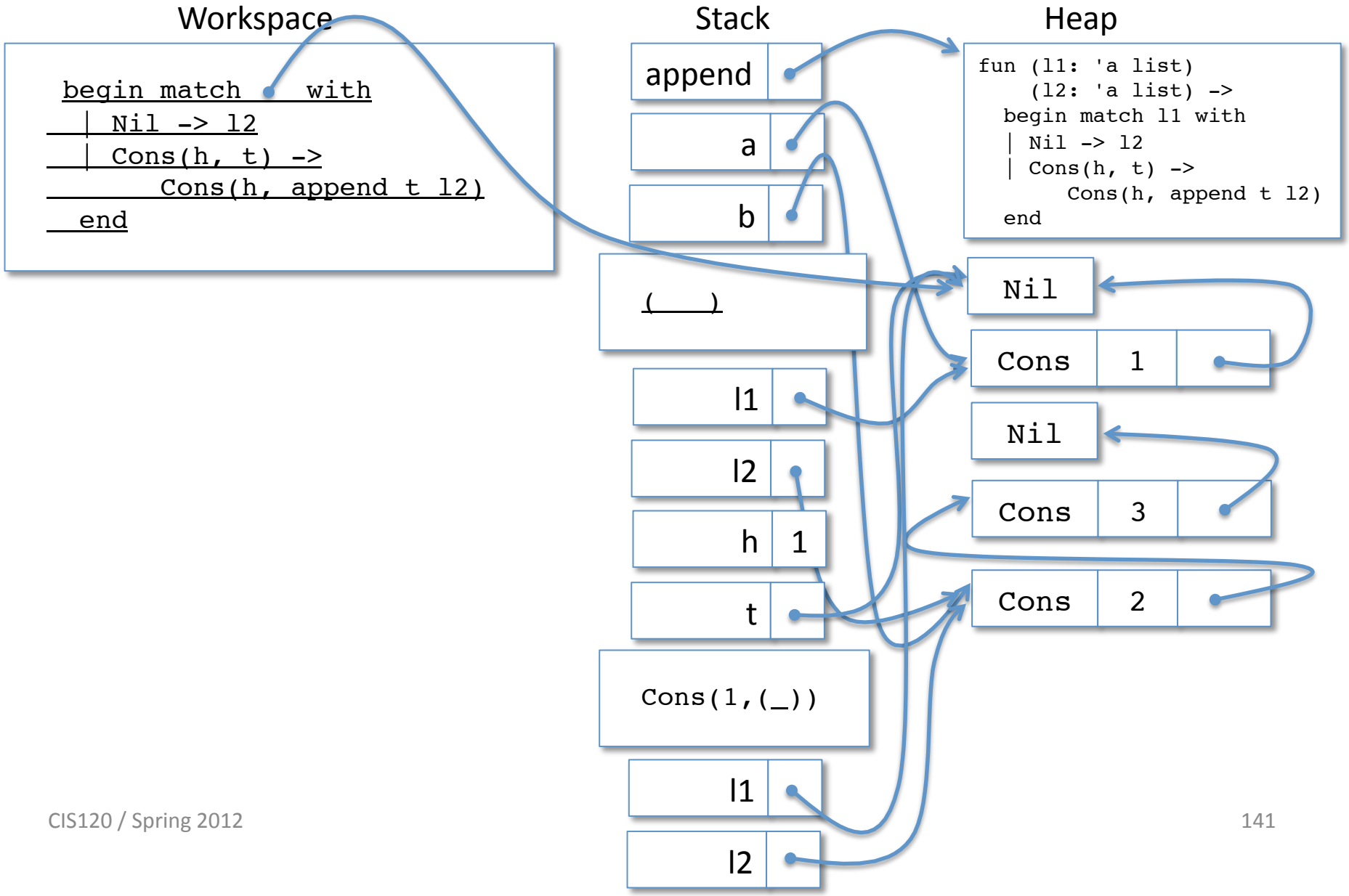
```
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
 end
```



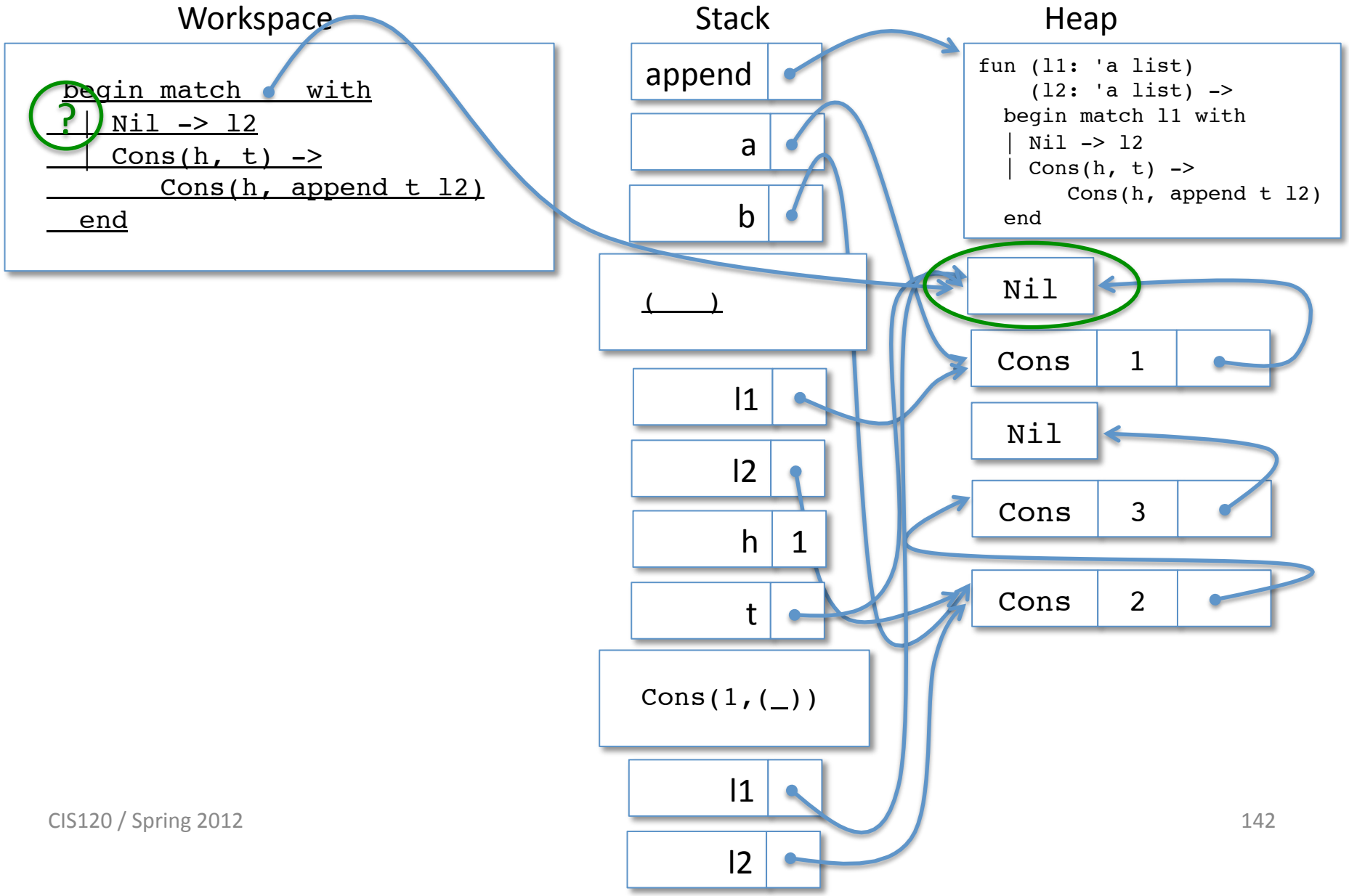
# Lookup ‘l1’


Workspace
begin match with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Stack
append
a
b
( )
l1
l2
h 1
t
Cons(1,(_))
l1
l2
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

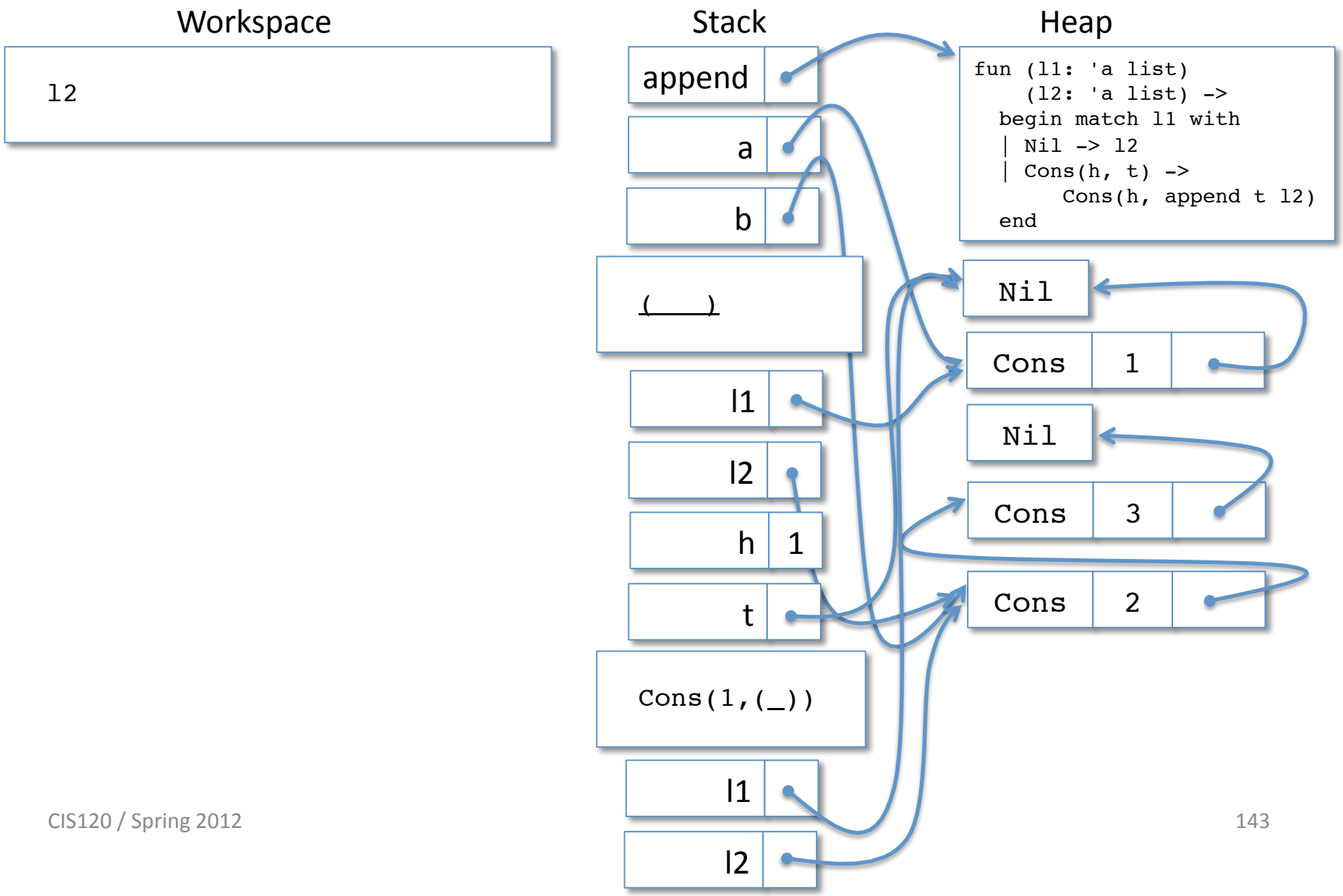
# Match Expression



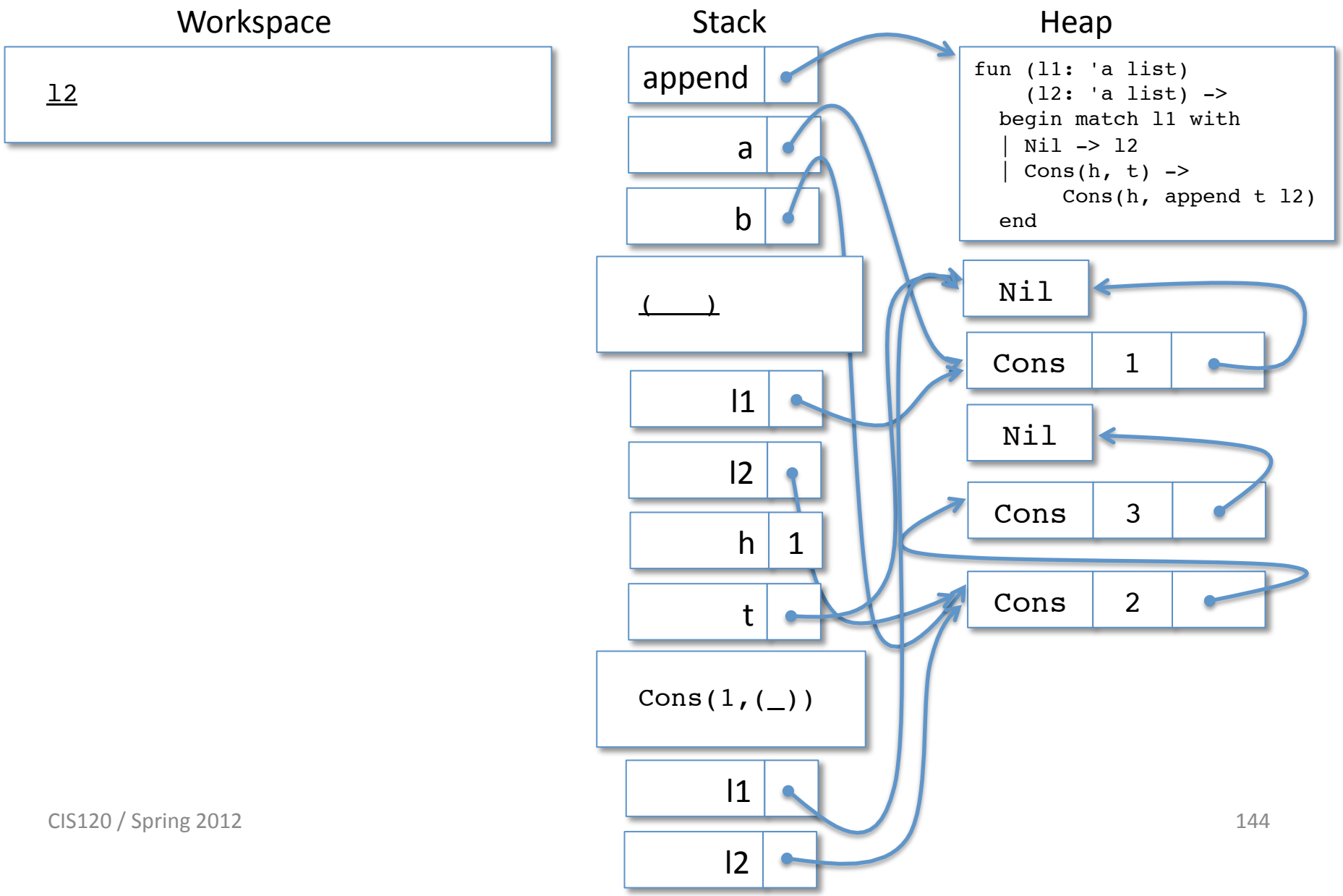
# The Nil case Matches



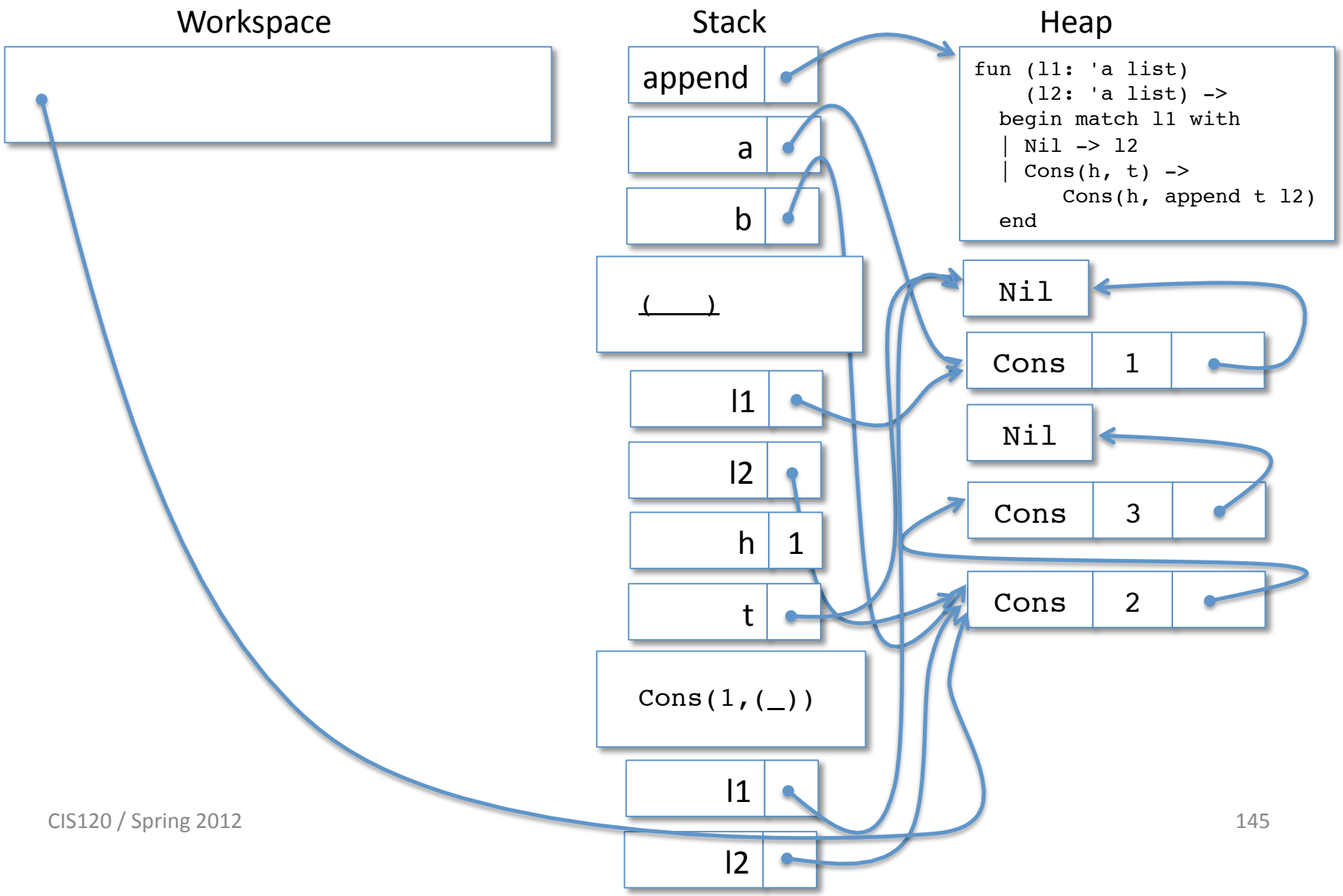
# Simplify the Branch (nothing to push)



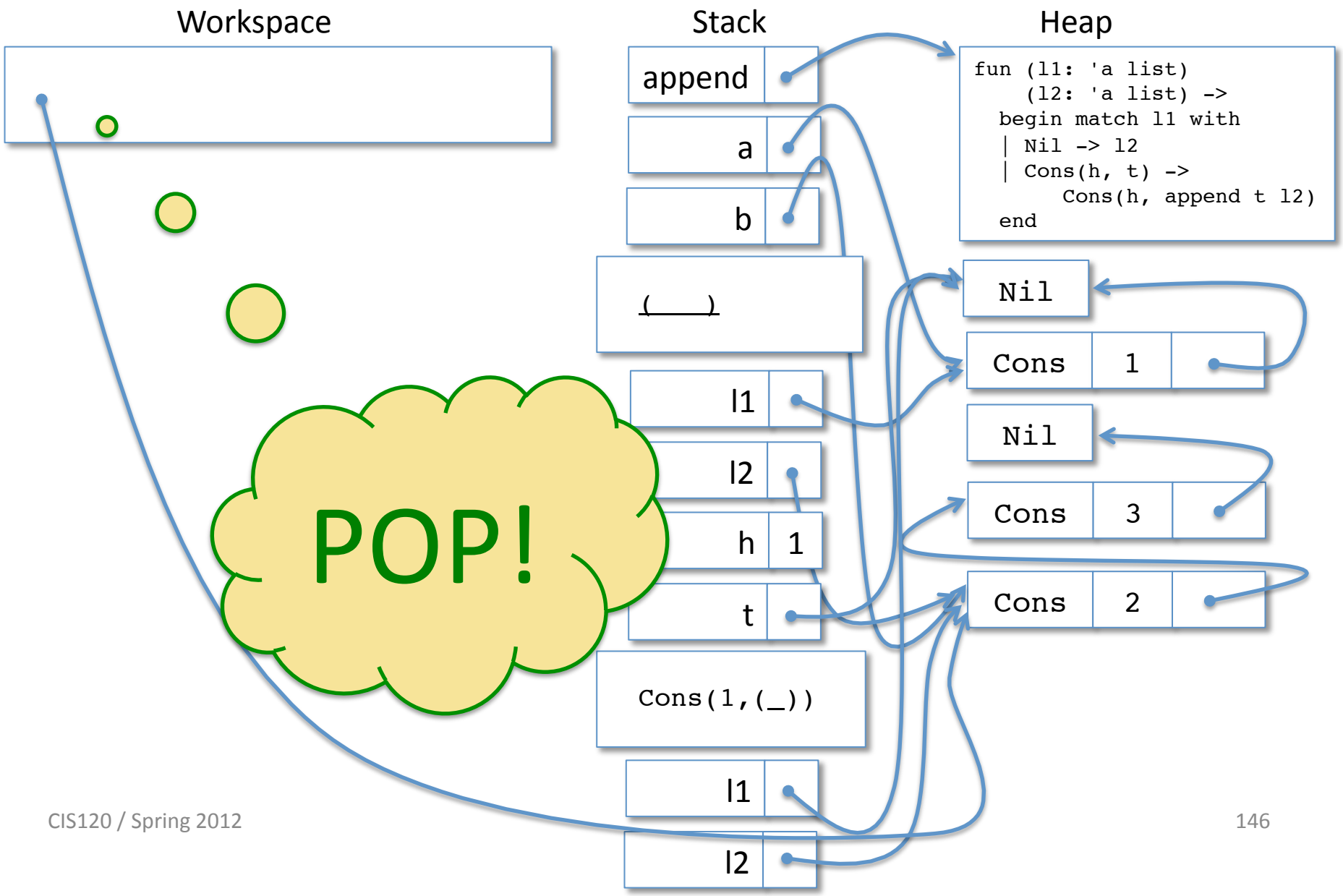
# Lookup ‘l2’


Workspace
l2
Stack
append
a
b
( _ _ _ )
l1
l2
h
1
t
Cons(1,(_))
l1
l2
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons
1
Nil
Cons
3
Cons
2

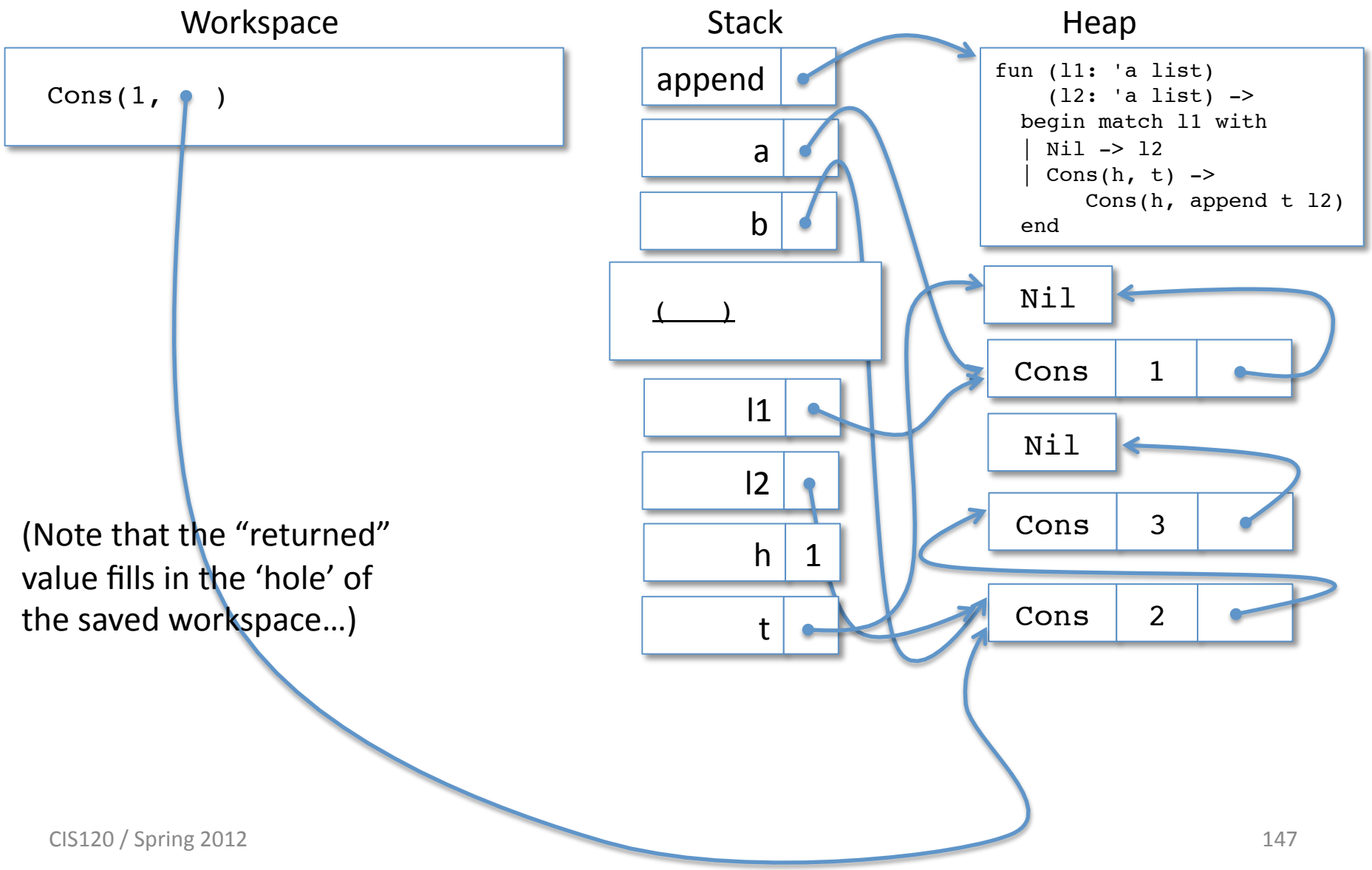
# Lookup ‘l2’


Workspace
Stack
Heap
append
a
b
( )
l1
l2
h 1
t
Cons(1,(_))
l1
l2
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

# Done! Pop stack to last Workspace

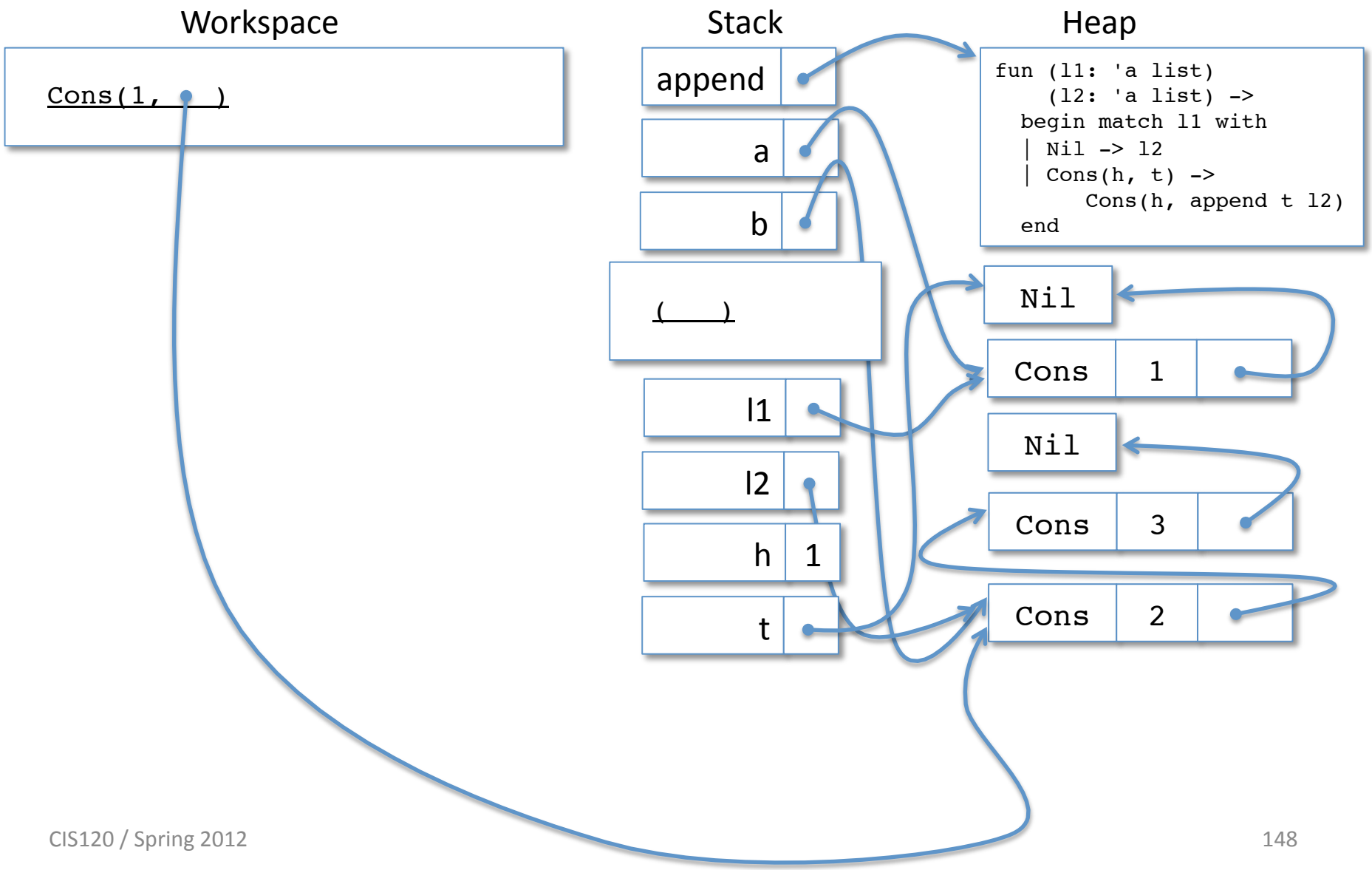


# Done! Pop stack to last Workspace

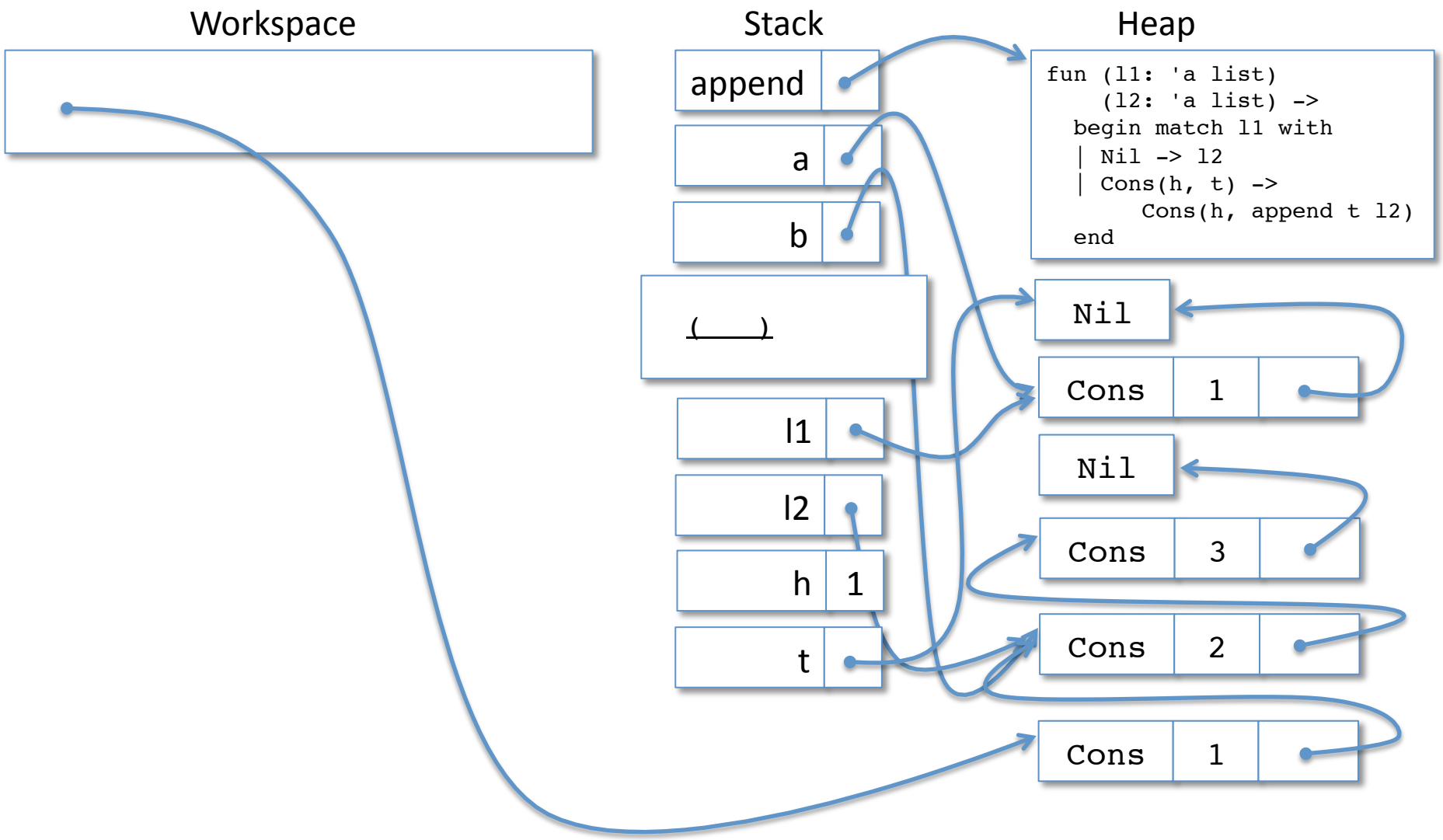



(Note that the "returned" value fills in the 'hole' of the saved workspace…)

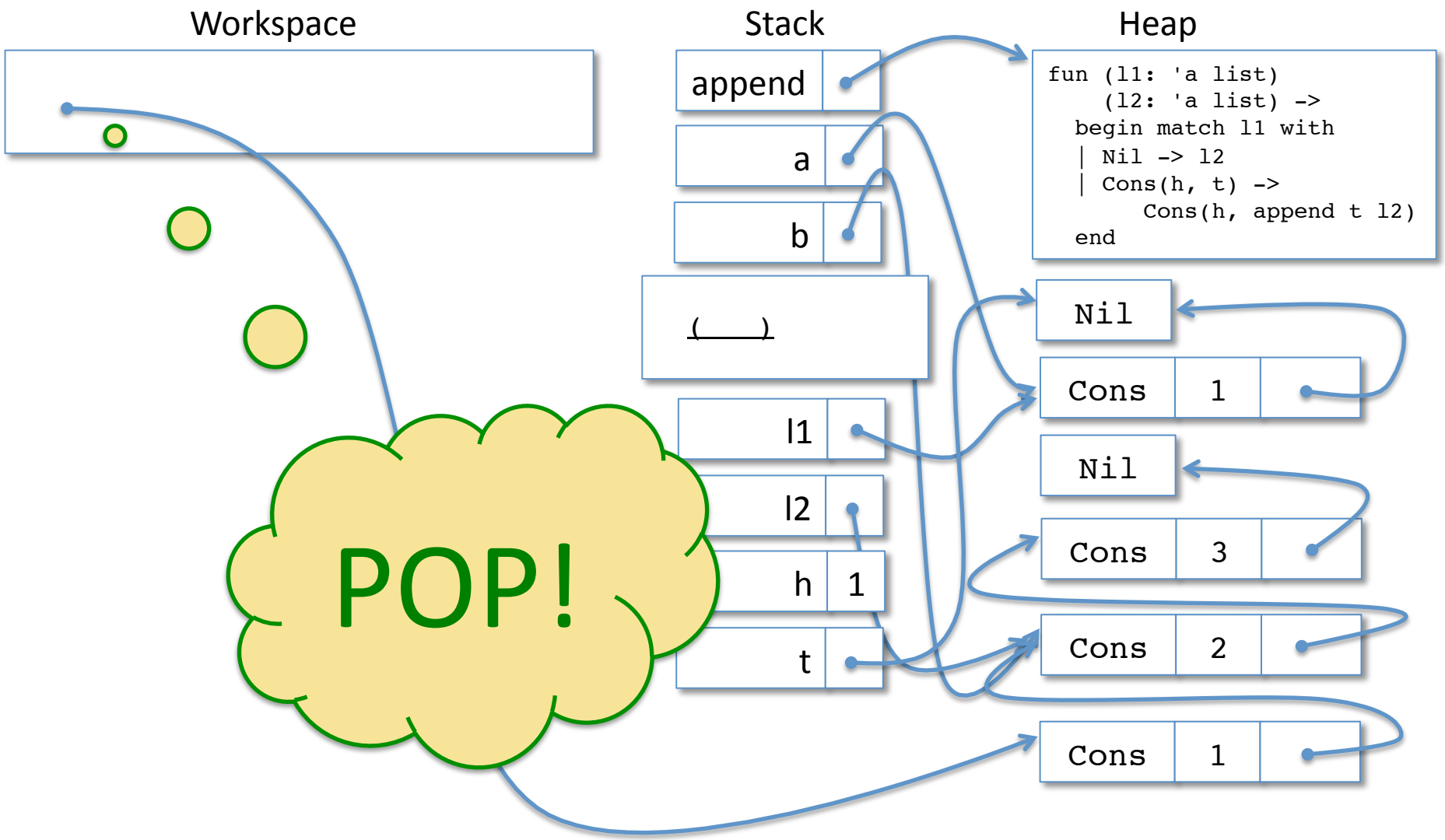
# Allocate a Cons cell



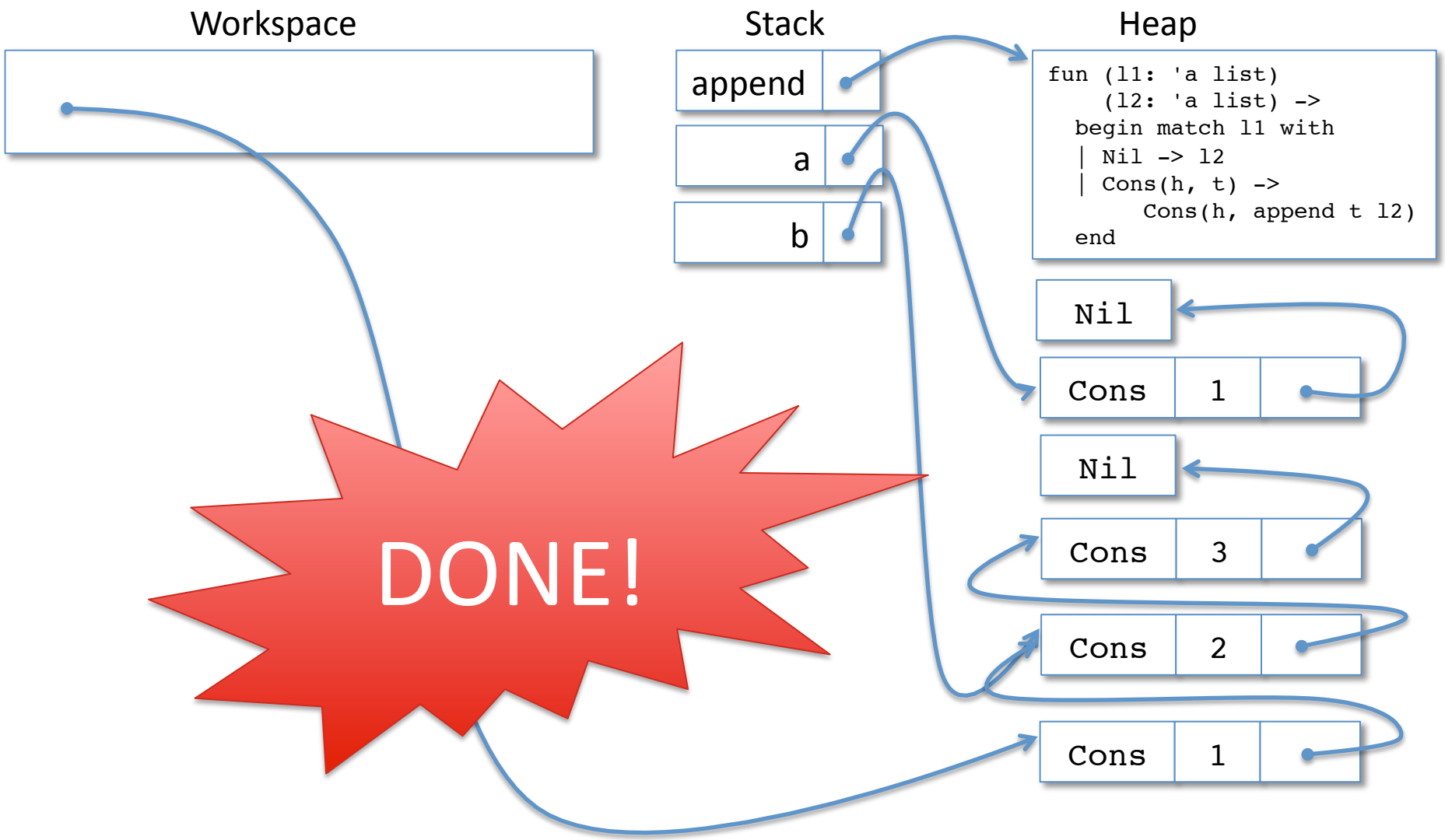
# Allocate a Cons cell



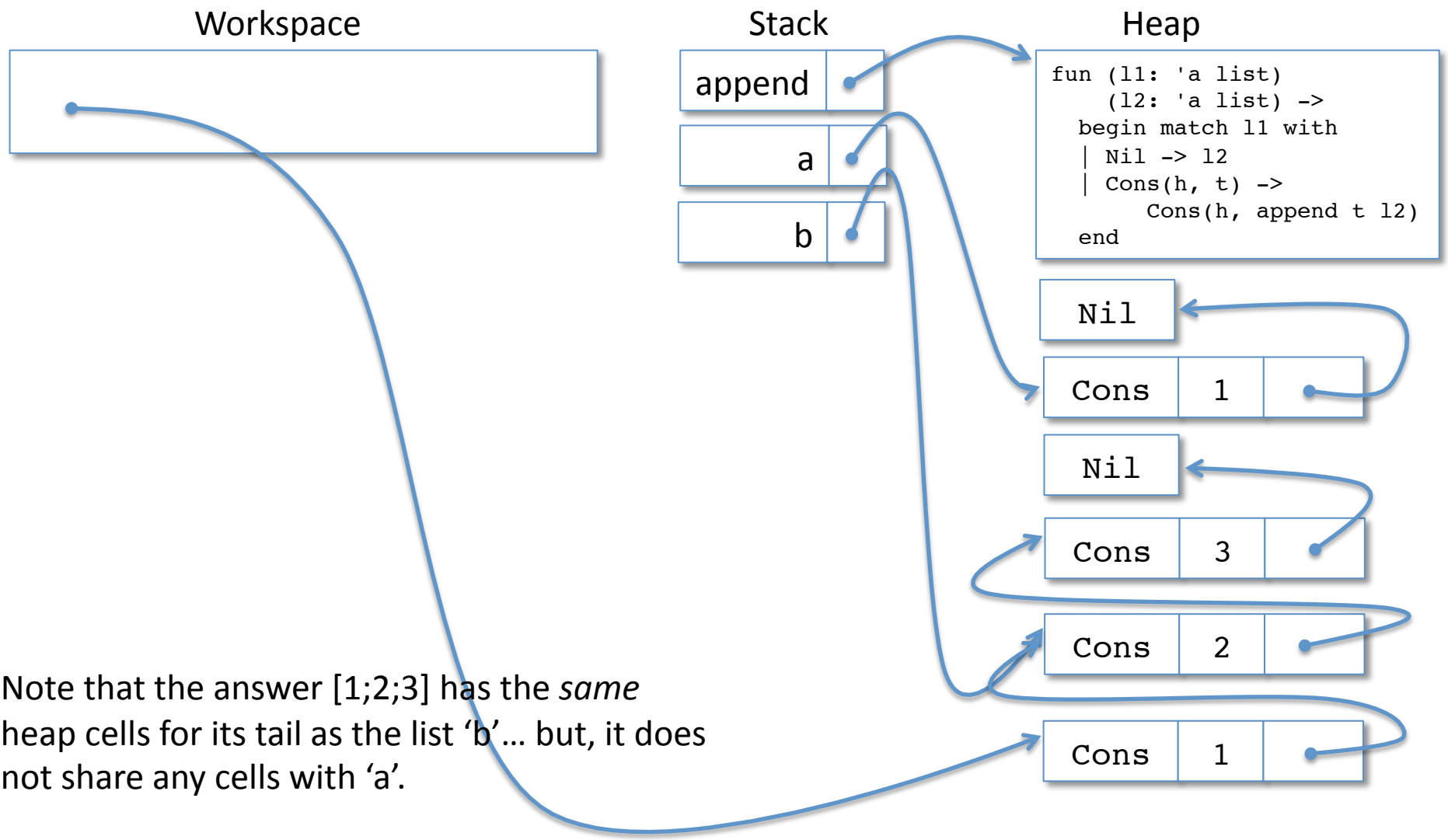
# Done! Pop stack to last Workspace



# Done! (PHEW!)


Workspace
Stack
Heap
append
a
b
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, append t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2
Cons 1
DONE!

# Done! (PHEW!)




Note that the answer [1;2;3] has the *same* heap cells for its tail as the list 'b'… but, it does not share any cells with 'a'.

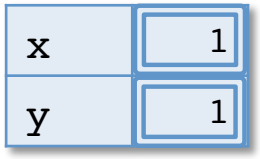
# Mutable Records

- We had to go through all this abstract stack stuff to make the model of 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
  - (remember that field names don't actually take up any space)




A point record in the heap.

# Allocate a Record

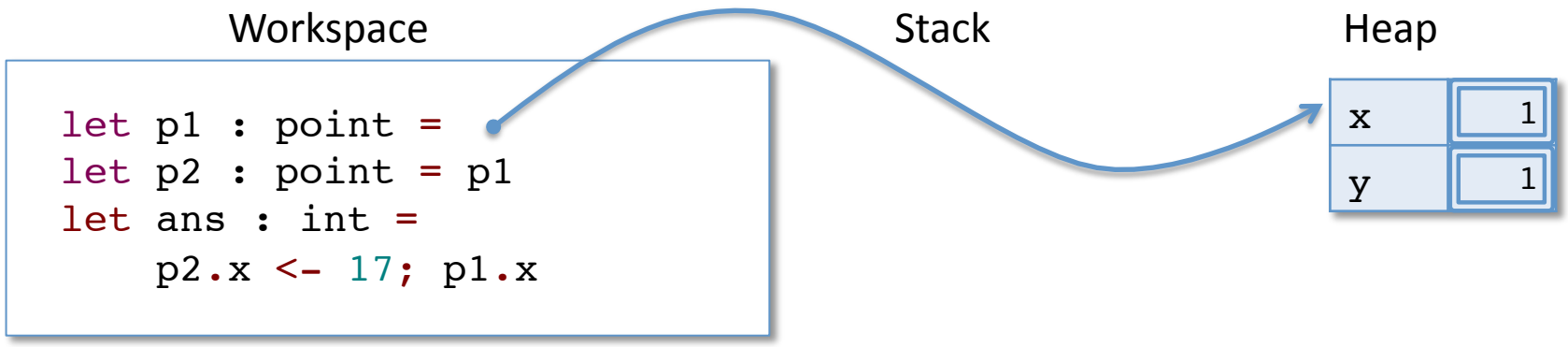
Workspace

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

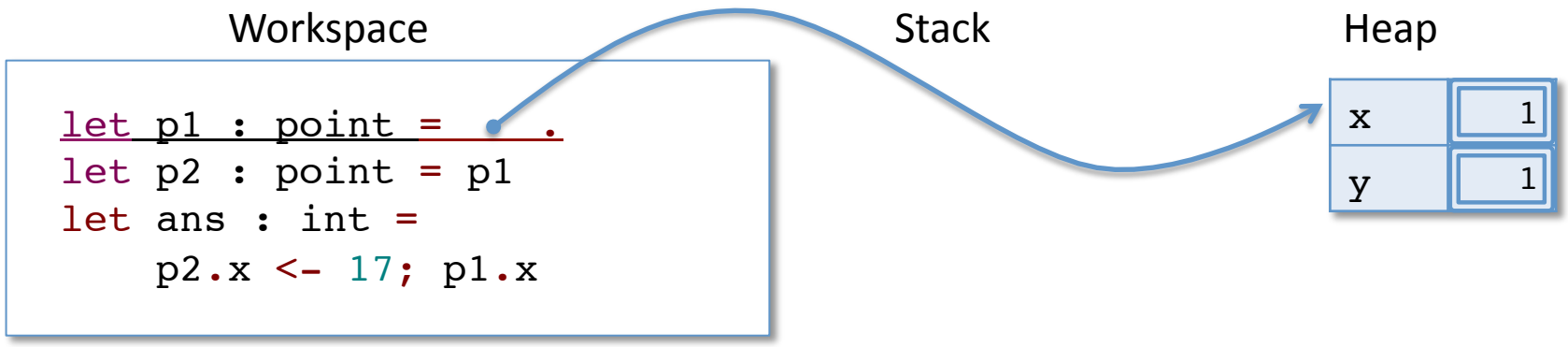
Stack

Heap

# Allocate a Record



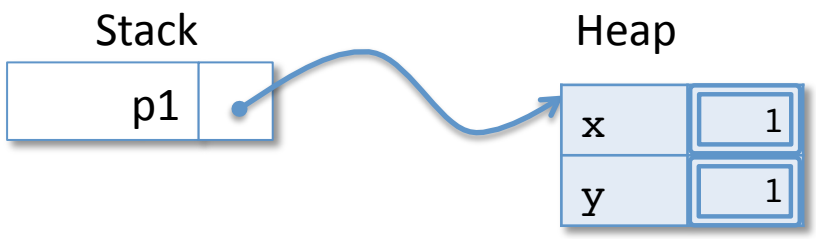
# Let Expression



# Push p1

Workspace

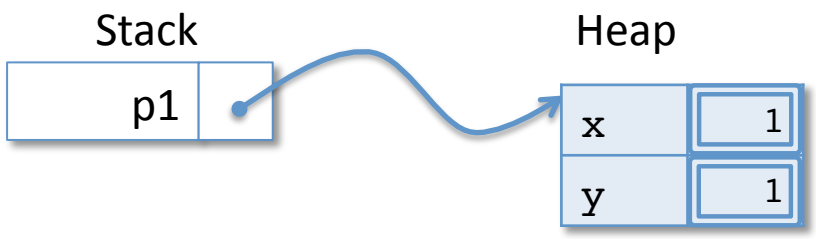
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



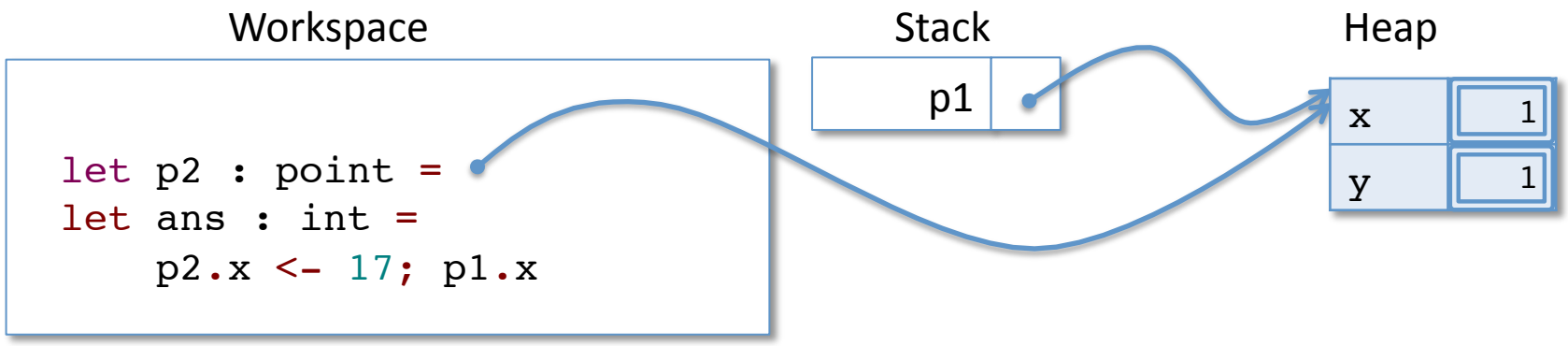
# Lookup ‘p1’

Workspace

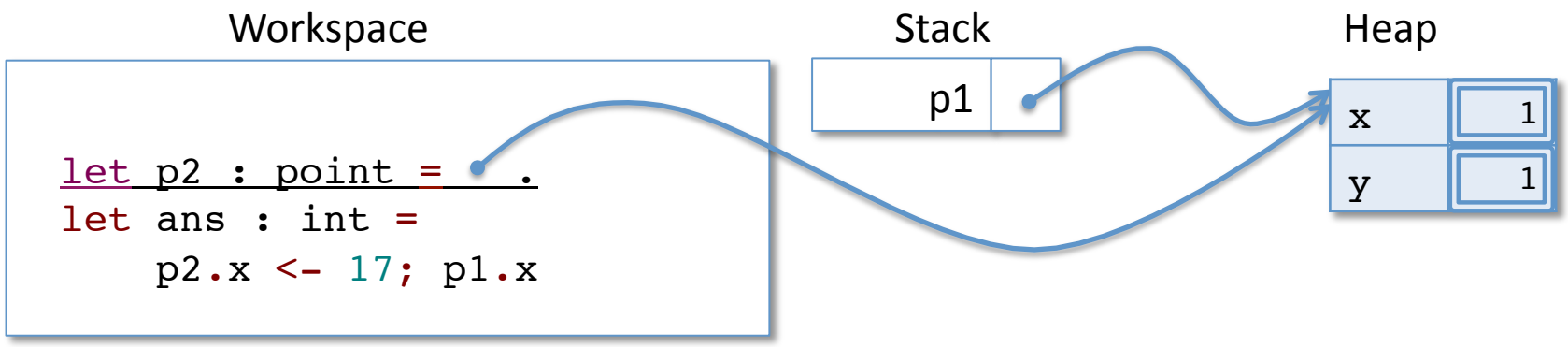
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



# Lookup ‘p1’


Workspace
Stack
Heap
let p2 : point =
let ans : int =
p2.x <- 17; p1.x
p1
x
1
y
1

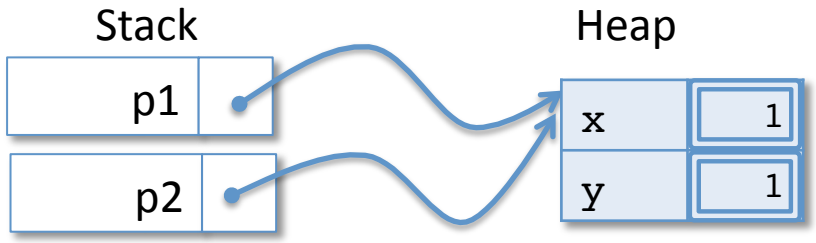
# Let Expression



# Push p2

Workspace

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

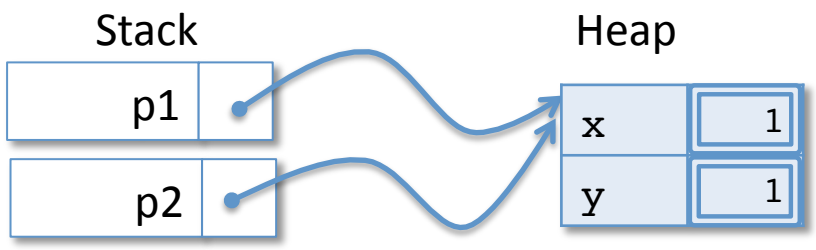



Note: p1 and p2 are references to the *same* heap record.
They are *aliases* – two different names for the *same* thing.

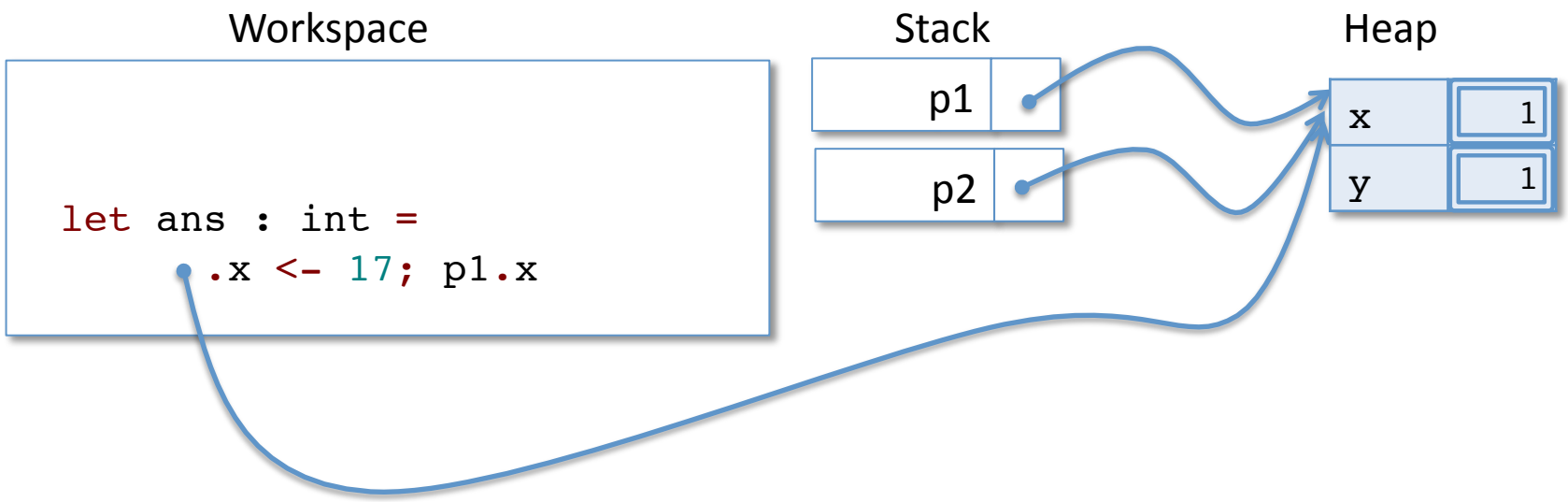
# Lookup ‘p2’

Workspace

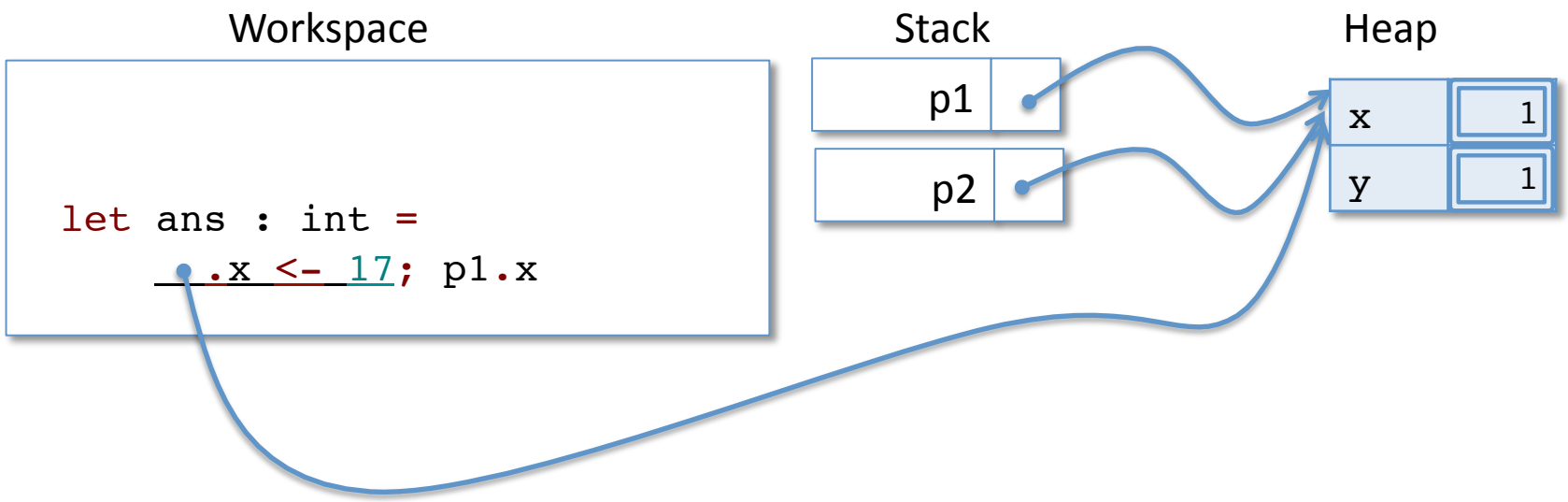
```
let ans : int =
    p2.x <- 17; p1.x
```



# Lookup ‘p2’



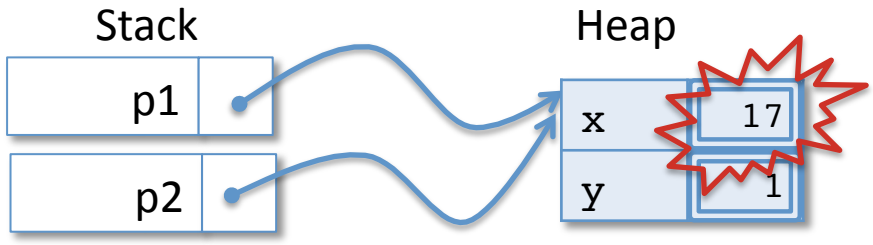
# Assign to x field



# Assign to x field

Workspace

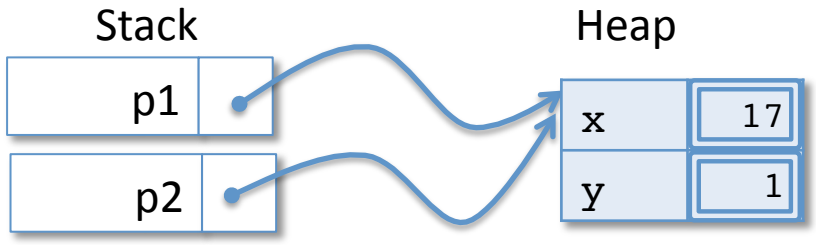
```
let ans : int =
    (); p1.x
```



# Sequence ';' Discards Unit

Workspace

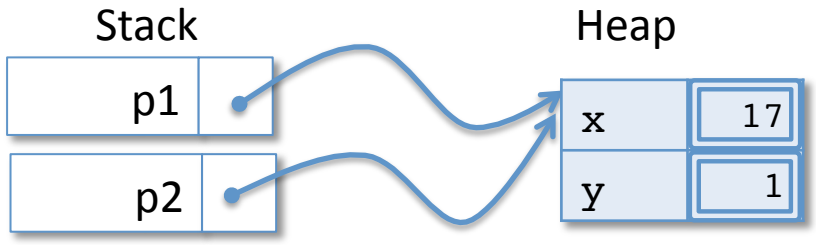
```
let ans : int =
    (); p1.x
```



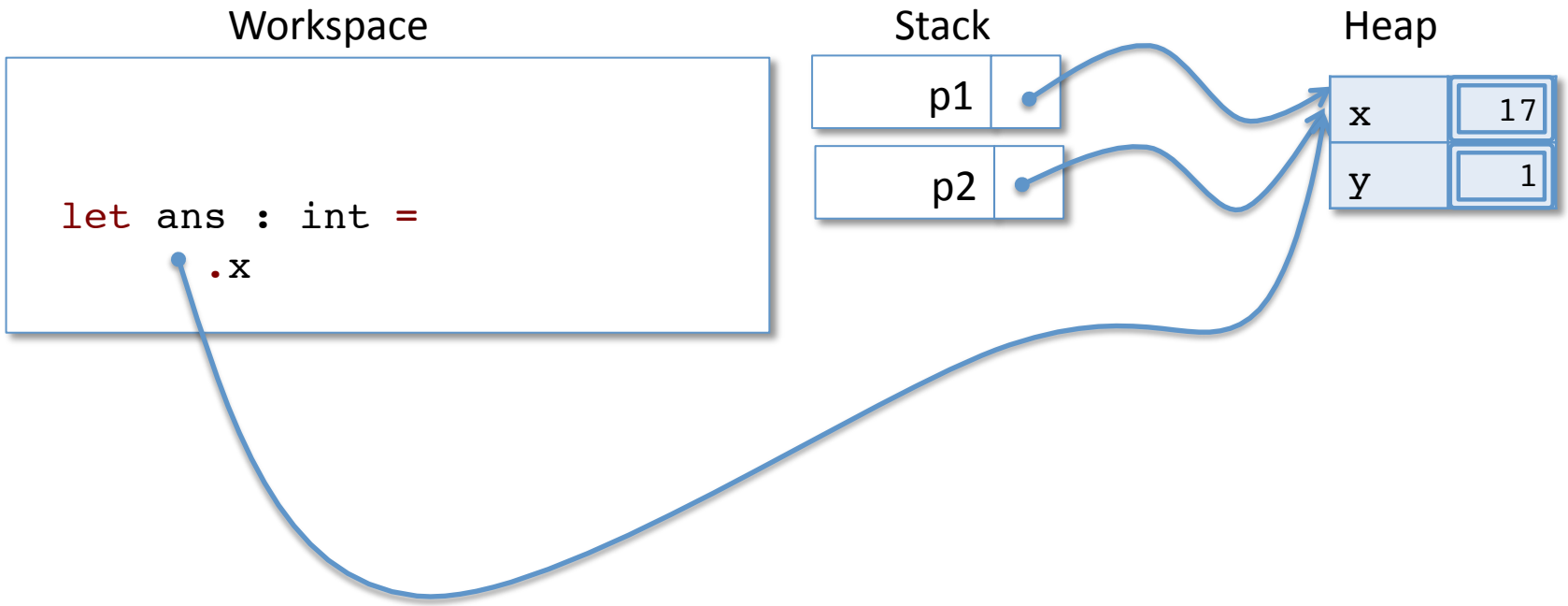
# Lookup ‘p1’

Workspace

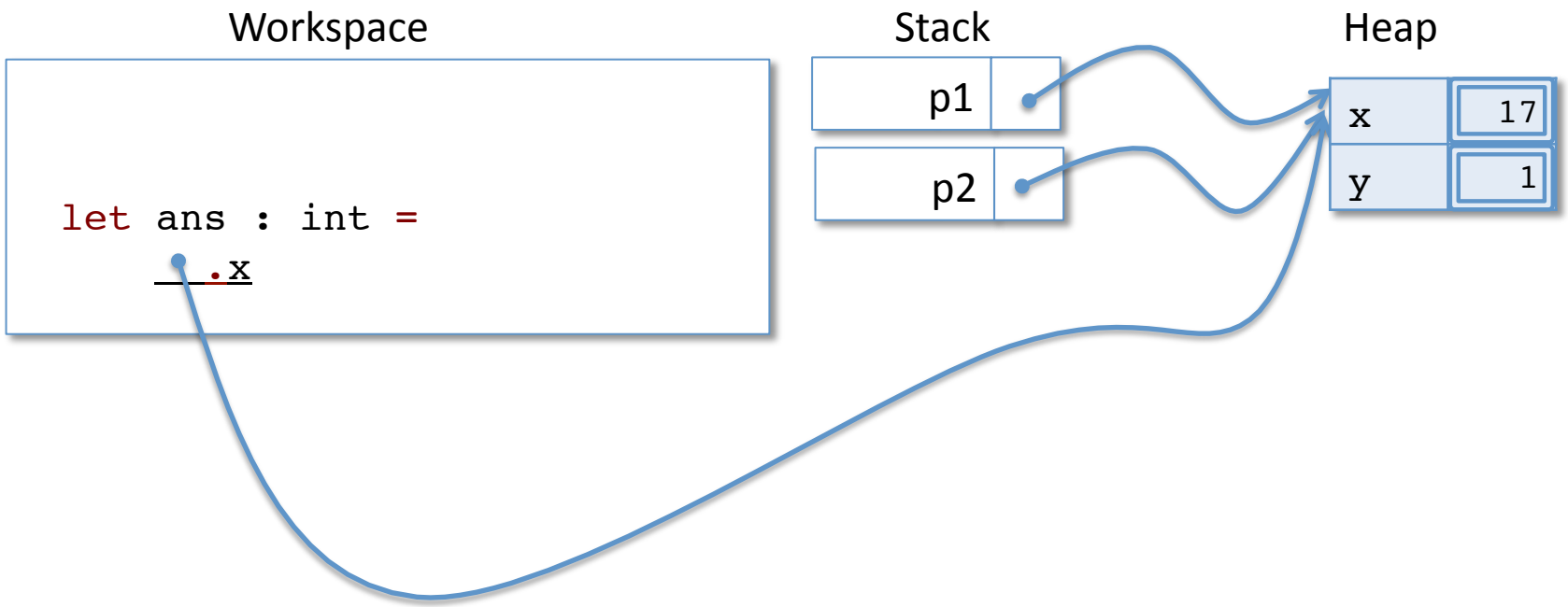
```
let ans : int =
    p1.x
```



# Lookup ‘p1’



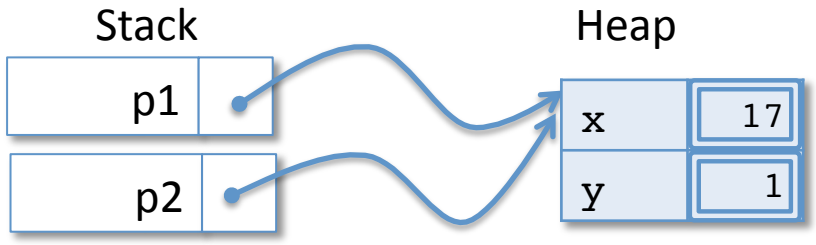
# Project the ‘x’ field



# Project the ‘x’ field

Workspace

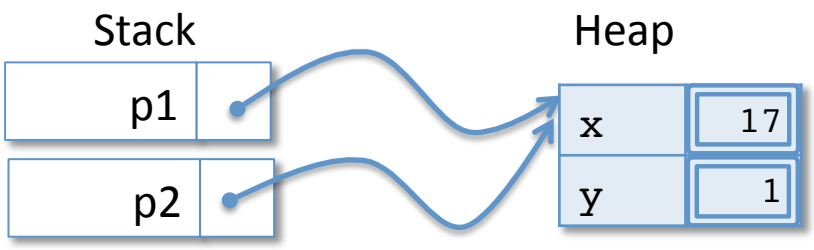
```
let ans : int =
    17
```



# Let Expression

Workspace

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



# Push ans