Programming Languages and Techniques (CIS1200)

Lecture 17

Iteration and Tail Recursion

Hidden State, Objects

Chapter 17

Announcements (1)

- Midterm 1 Grades and Solutions available
  - Submit Regrade requests via Gradescope by Friday, 10/18
- HW04 due next Tuesday (at 11.59pm)
- Final Exam
  - Tuesday, December 17th, 12-2pm

Announcements (2)

- Democracy Assignment (Due November 6th)
  - If you are eligible to vote in this election, submit a photograph of yourself at your polling place on November 5th or, if appropriate, mailing off your mail-in ballot, etc..
  - If you are not eligible to vote in this election, then either:
    - Submit a photo of yourself supporting someone else in voting (e.g., accompanying them to their polling place), or
    - Submit a short essay (1/2 page to 1 page) on a topic connected to democracy and voting -- e.g., a summary of the challenges of fully electronic voting, or a description of how voting works in your country of origin.

- Those who complete the voting assignment will receive **two late days** for this semester's programming assignments

Review: Data Structure for Mutable Queues

```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

type 'a queue = { mutable head : 'a qnode option;
                  mutable tail : 'a qnode option }
```

- Mutable Queue *invariant*

Either:
(1) `head` and `tail` are both `None` (i.e. the queue is empty)
or
(2) `head` is `Some n1`, `tail` is `Some n2` and
- `n2` is reachable from `n1` by following 'next' pointers
- `n2.next` is `None`

- Each queue operation may *assume* that the invariant holds of its inputs and must *ensure* that the invariant holds when it's done.

Mutable Queues: Queue Length

working with singly linked data structures

Queue Length

Suppose we want to extend the interface with a length function:

```
module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue
  …

  (* Get the length of the queue *)
  val length : 'a queue -> int
end
```

How can we implement it?

length (recursively)

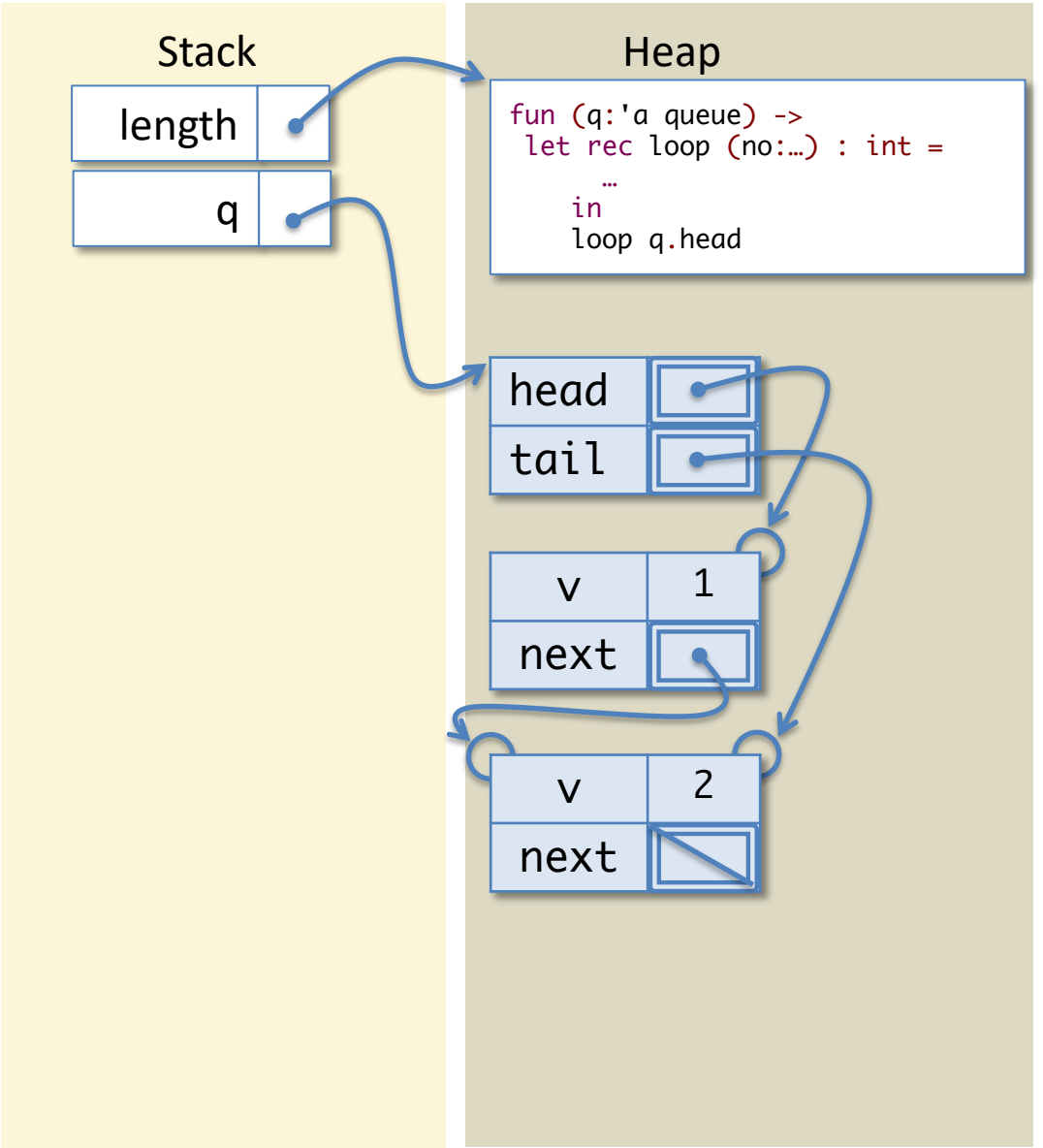
```
(* Calculate the length of the queue recursively *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

- This code for `length` uses a helper function, `loop`:
  - the correctness depends crucially on the queue invariant
  - (what happens if we pass in a bogus q that is cyclic?)
- The height of the ASM stack is proportional to the length of the queue
  - That seems inefficient… why should it take so much space?

Evaluating length

Workspace

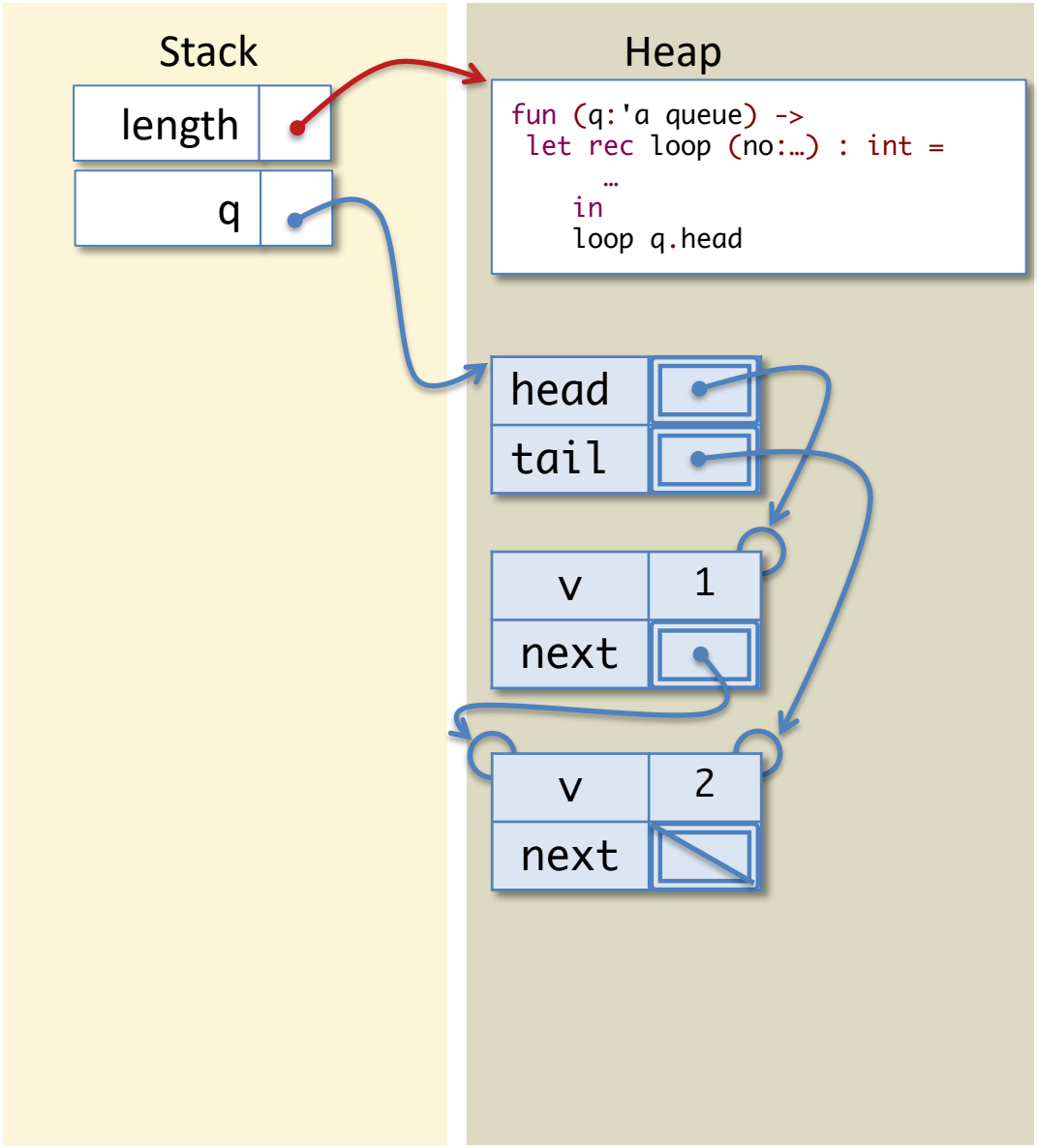
```
length q
```



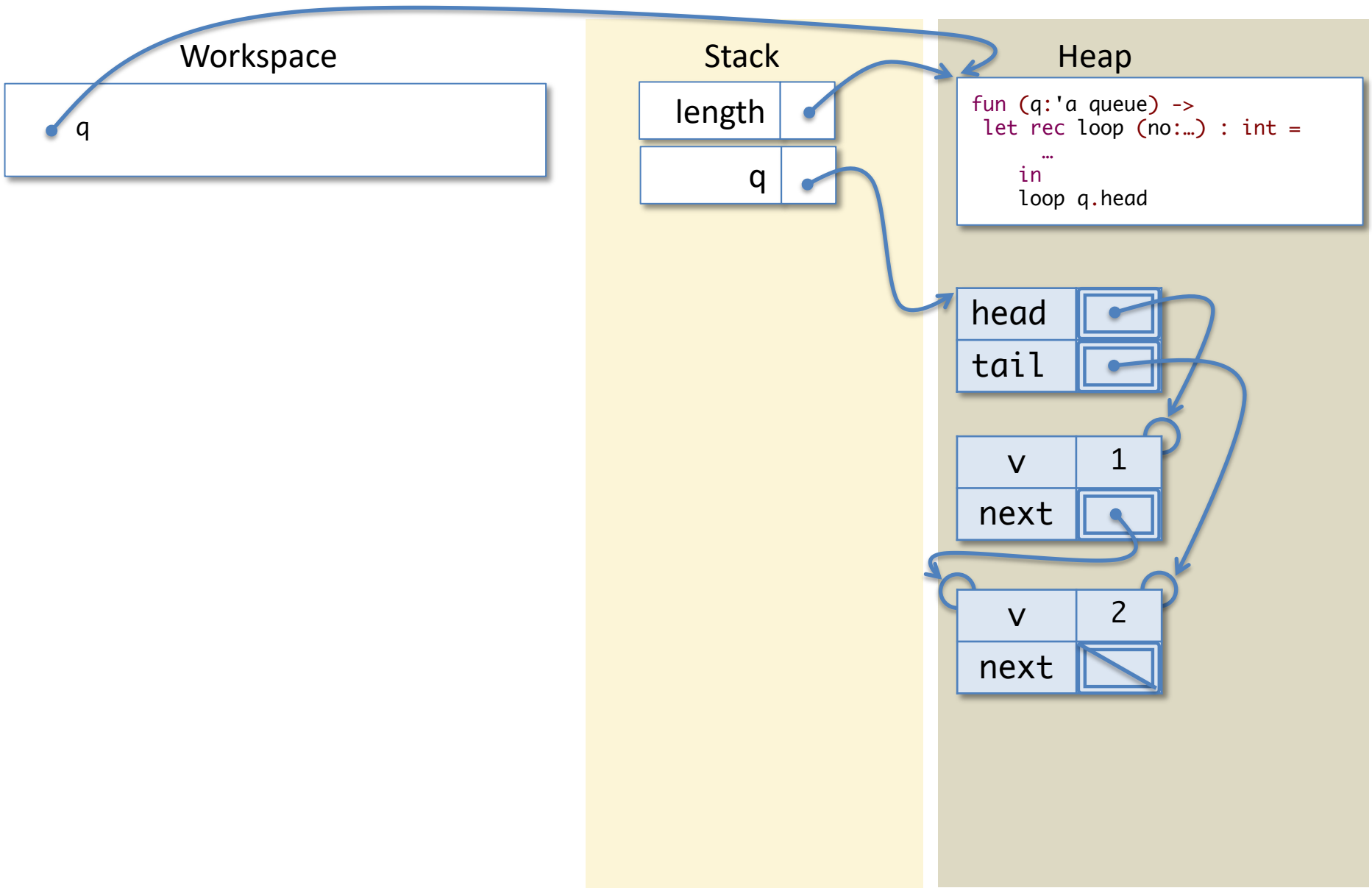
Evaluating length

Workspace

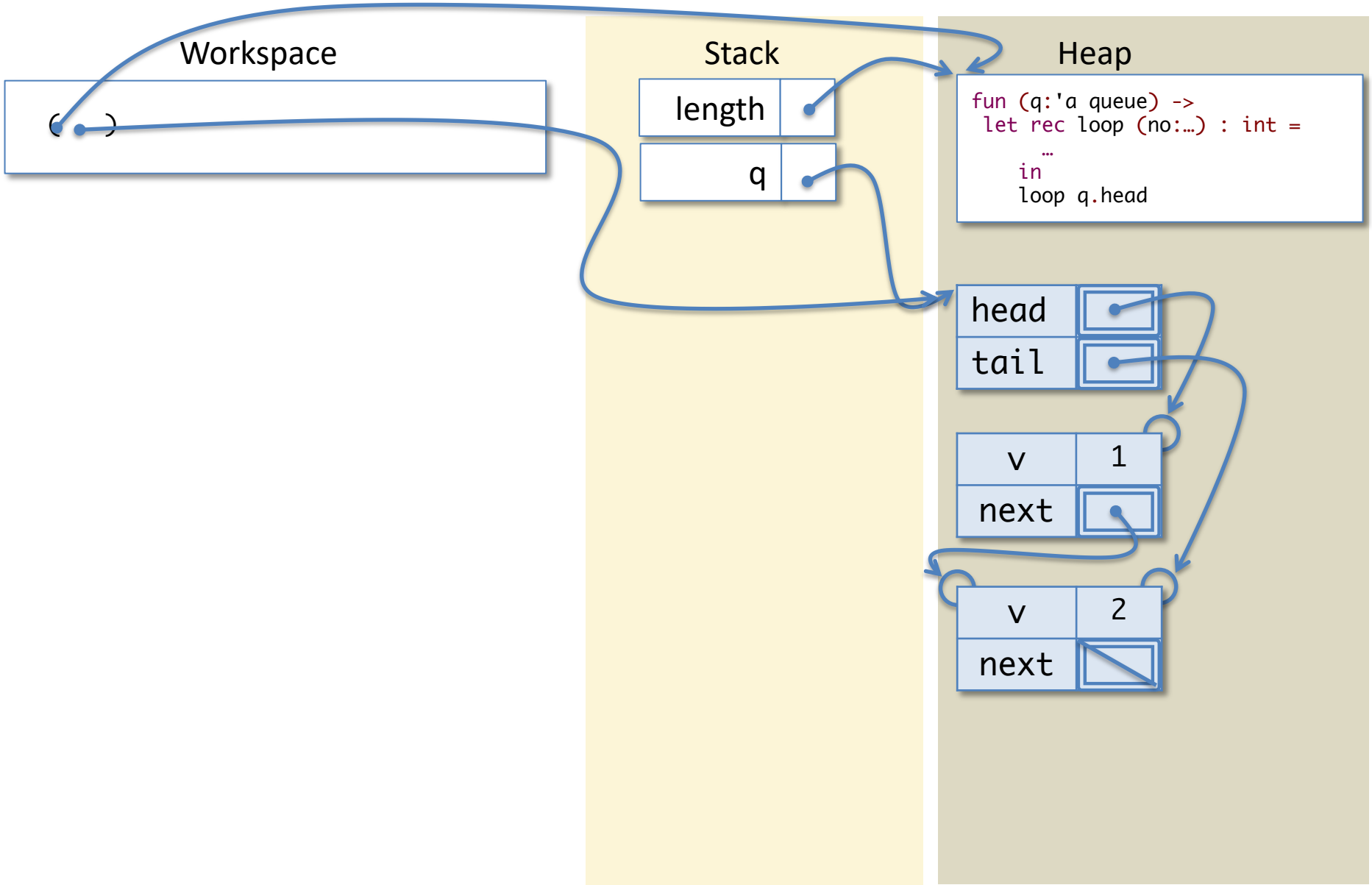
```
length q
```



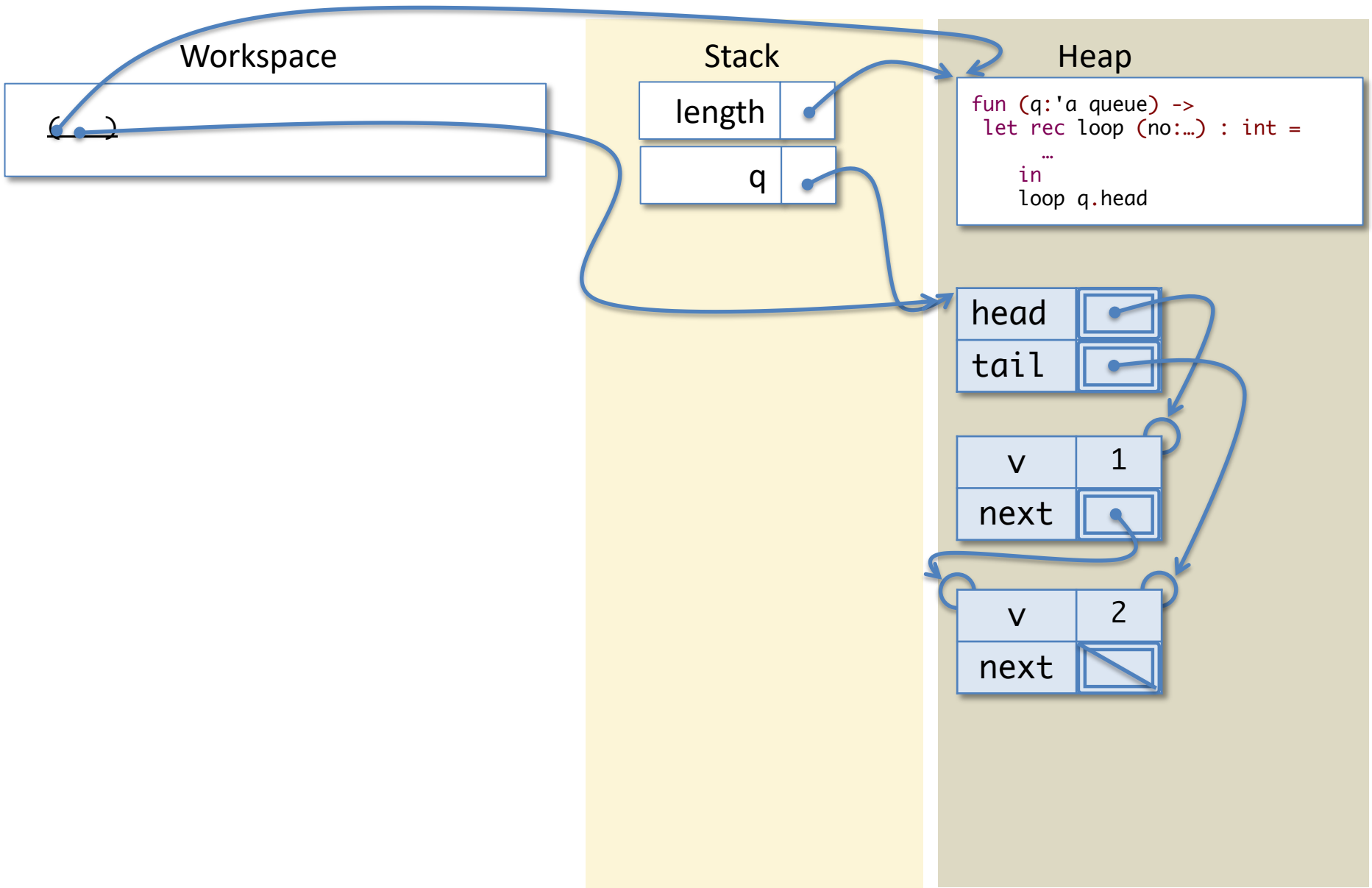
Evaluating length



Evaluating length



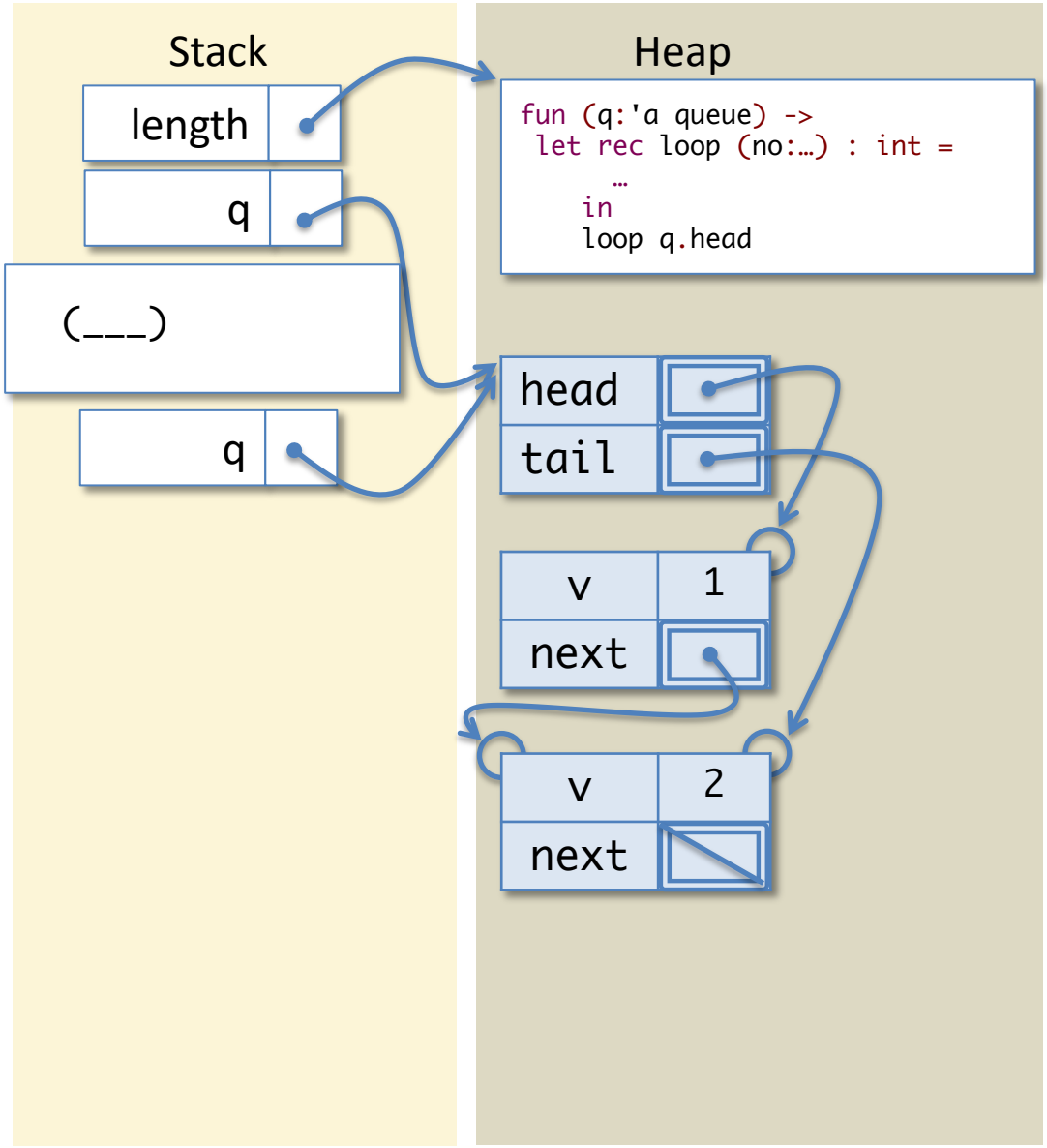
Evaluating length



Evaluating length

Workspace

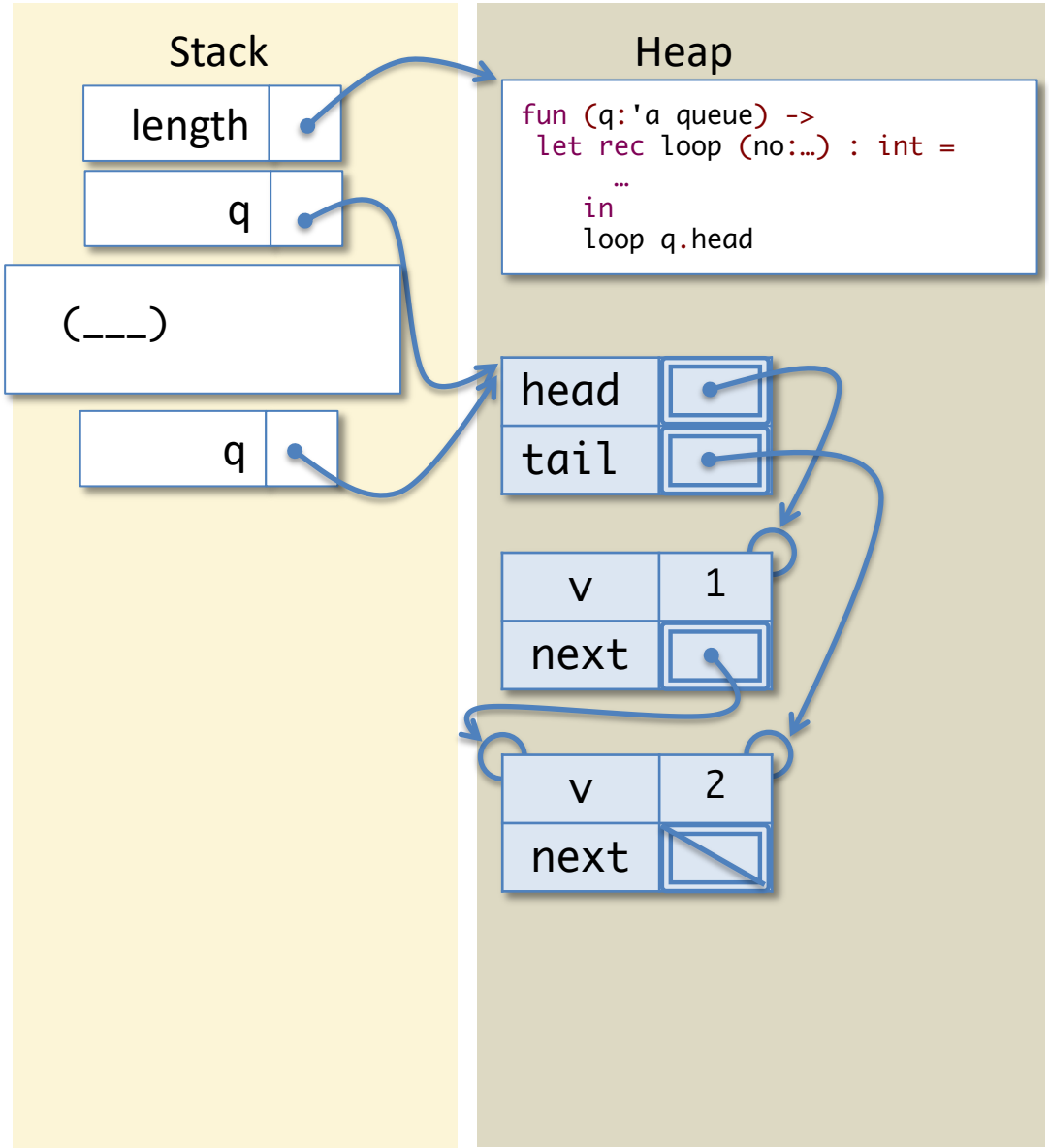
```
let rec loop (no: …) : int =
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

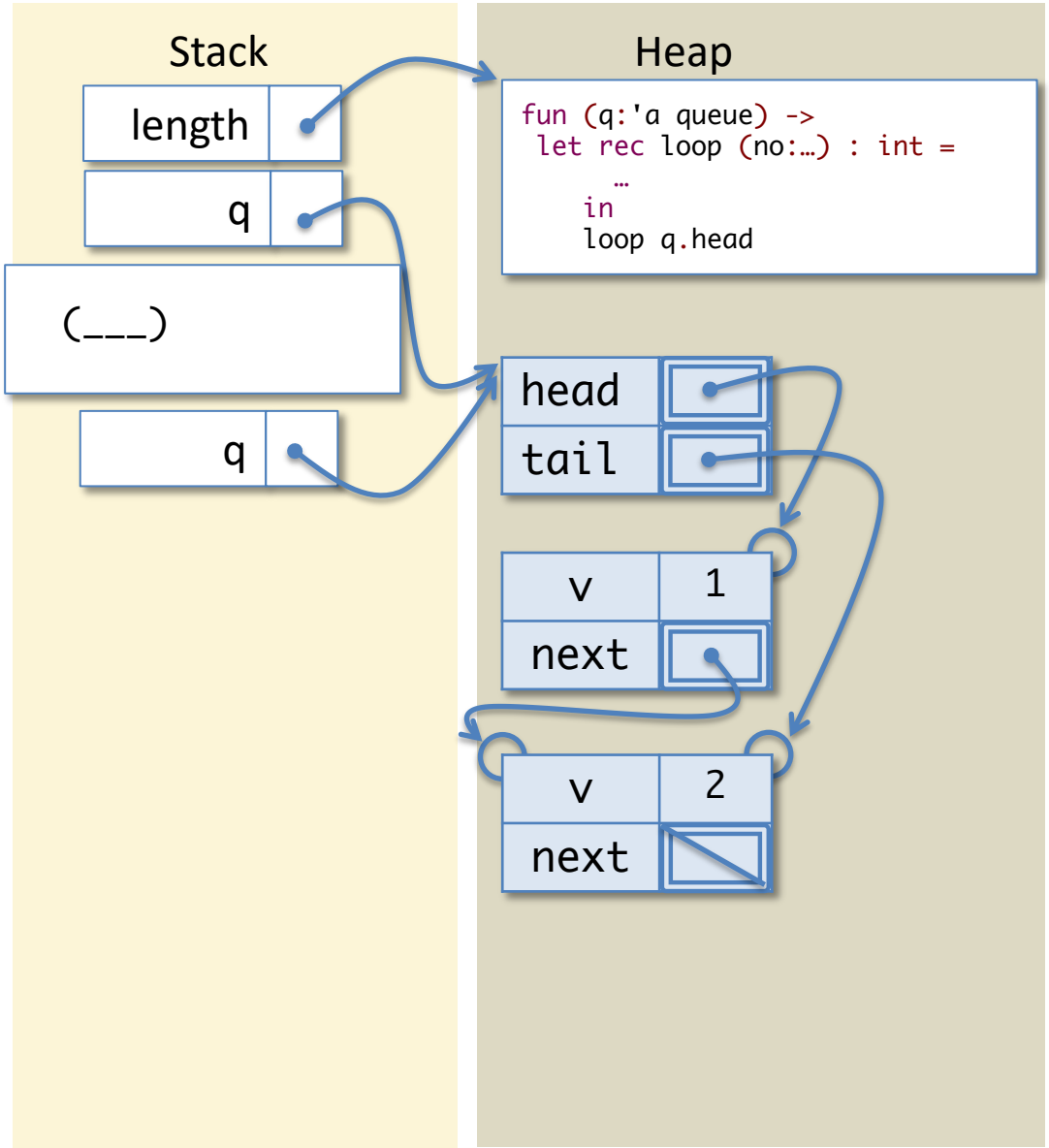
```
let rec loop = fun (no: …) ->
      begin match no with
        | None -> 0
        | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

Workspace

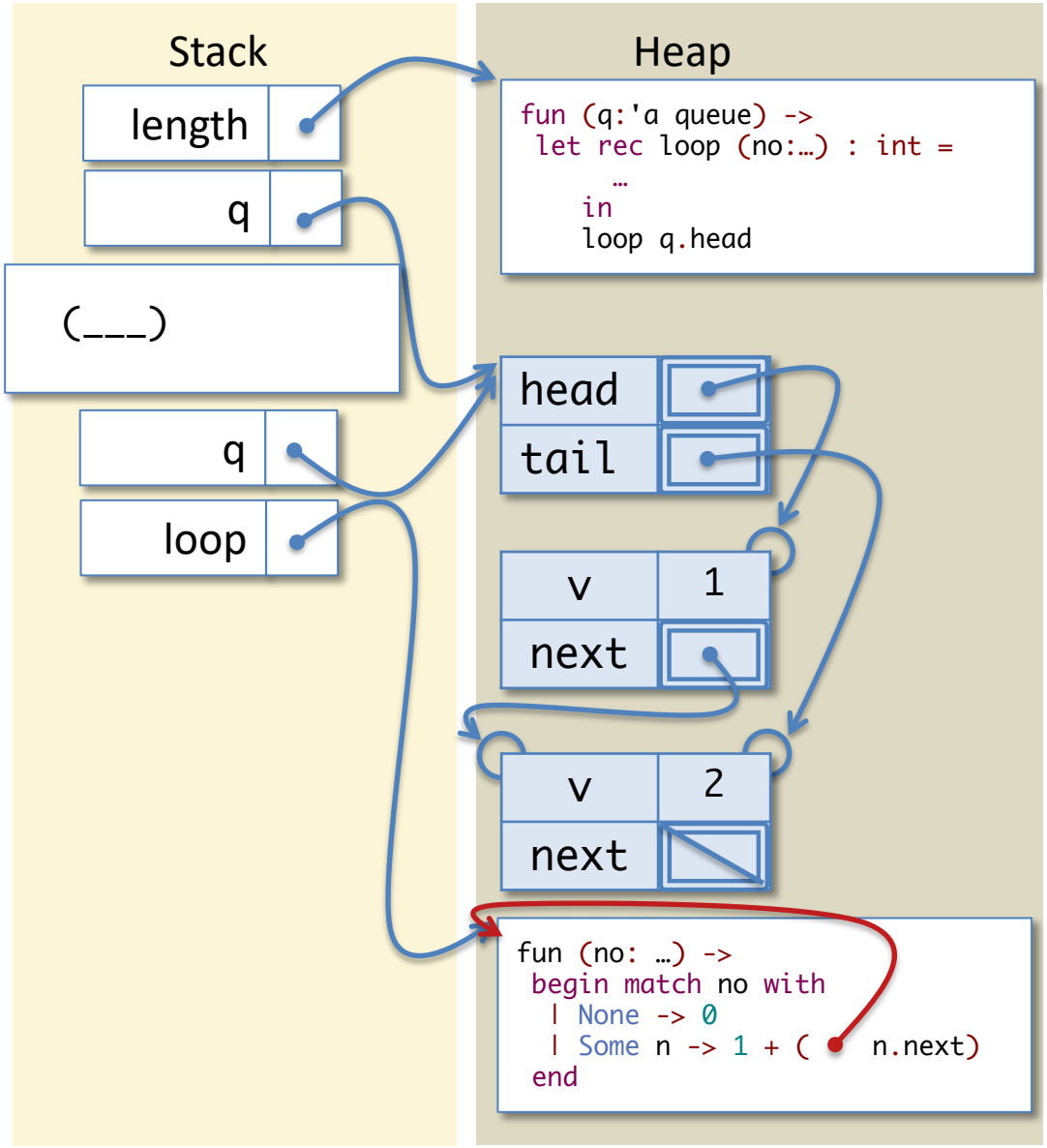
```
let loop = fun (no: …) ->
      begin match no with
          | None -> 0
          | Some n -> 1 + (loop n.next)
      end
    in
    loop q.head
```



Evaluating length

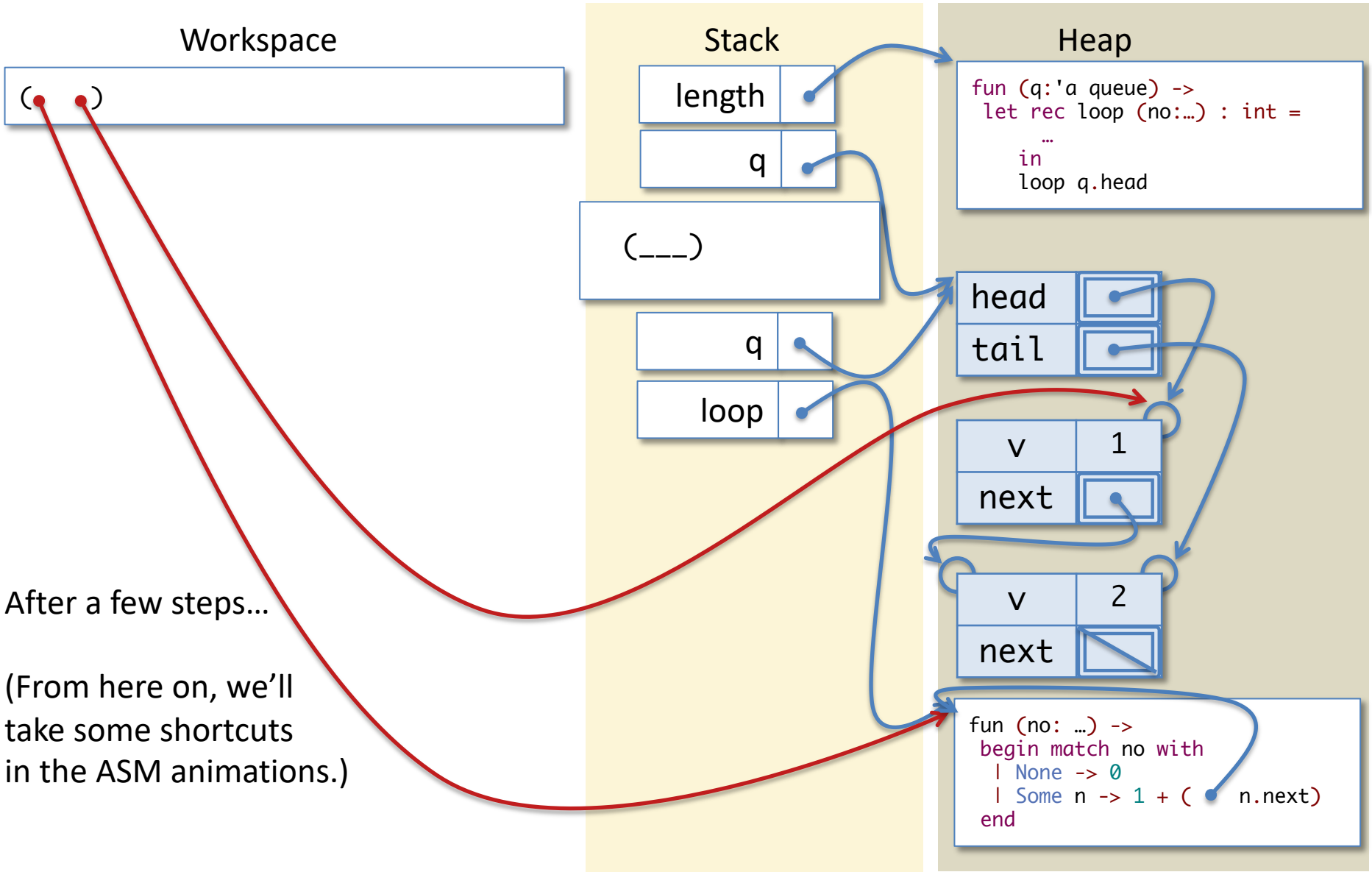
Workspace

```
loop q.head
```

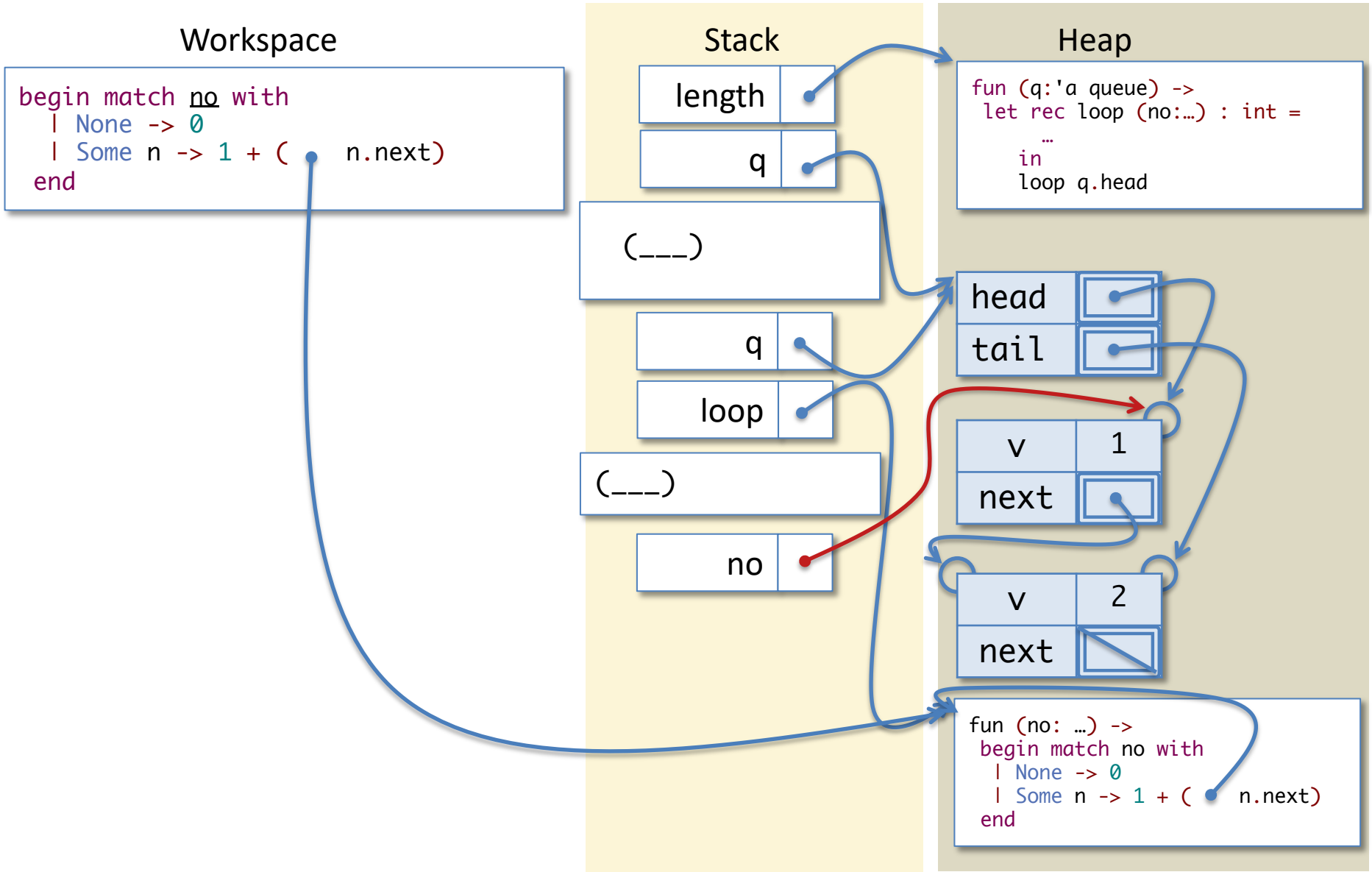


The reference to loop in its own body is *backpatched* when it moves into the heap…

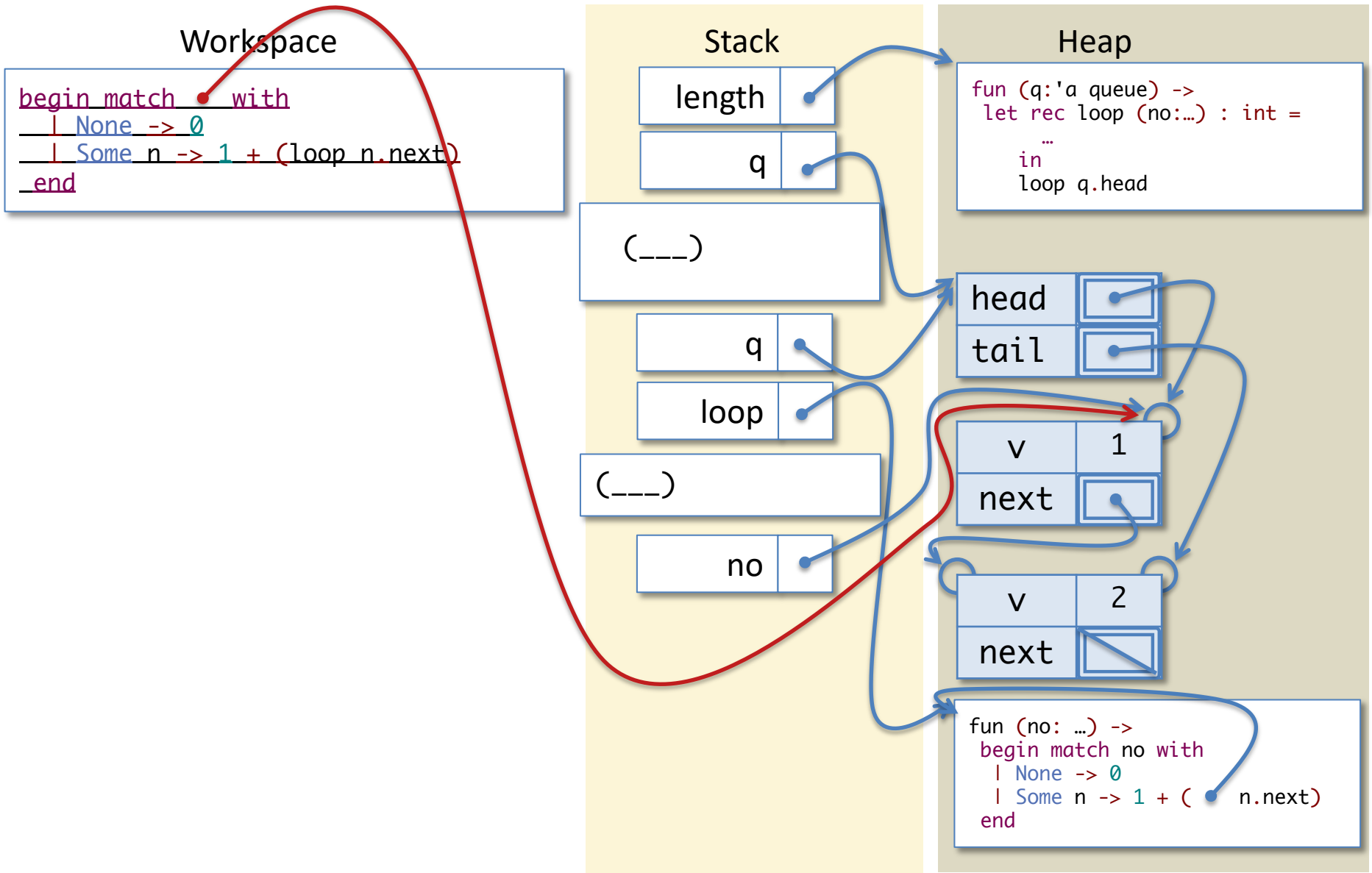
Evaluating length



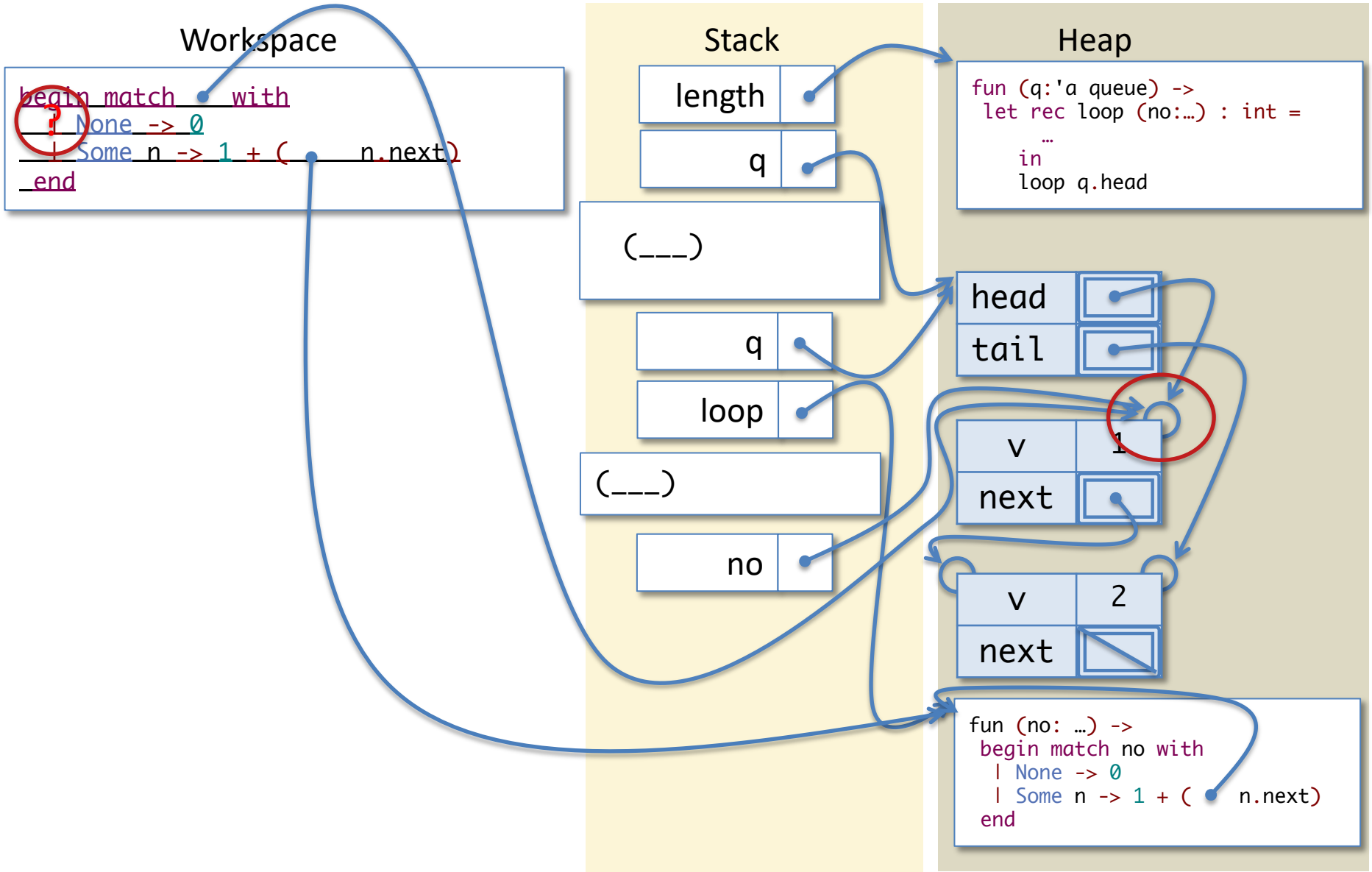
Evaluating length



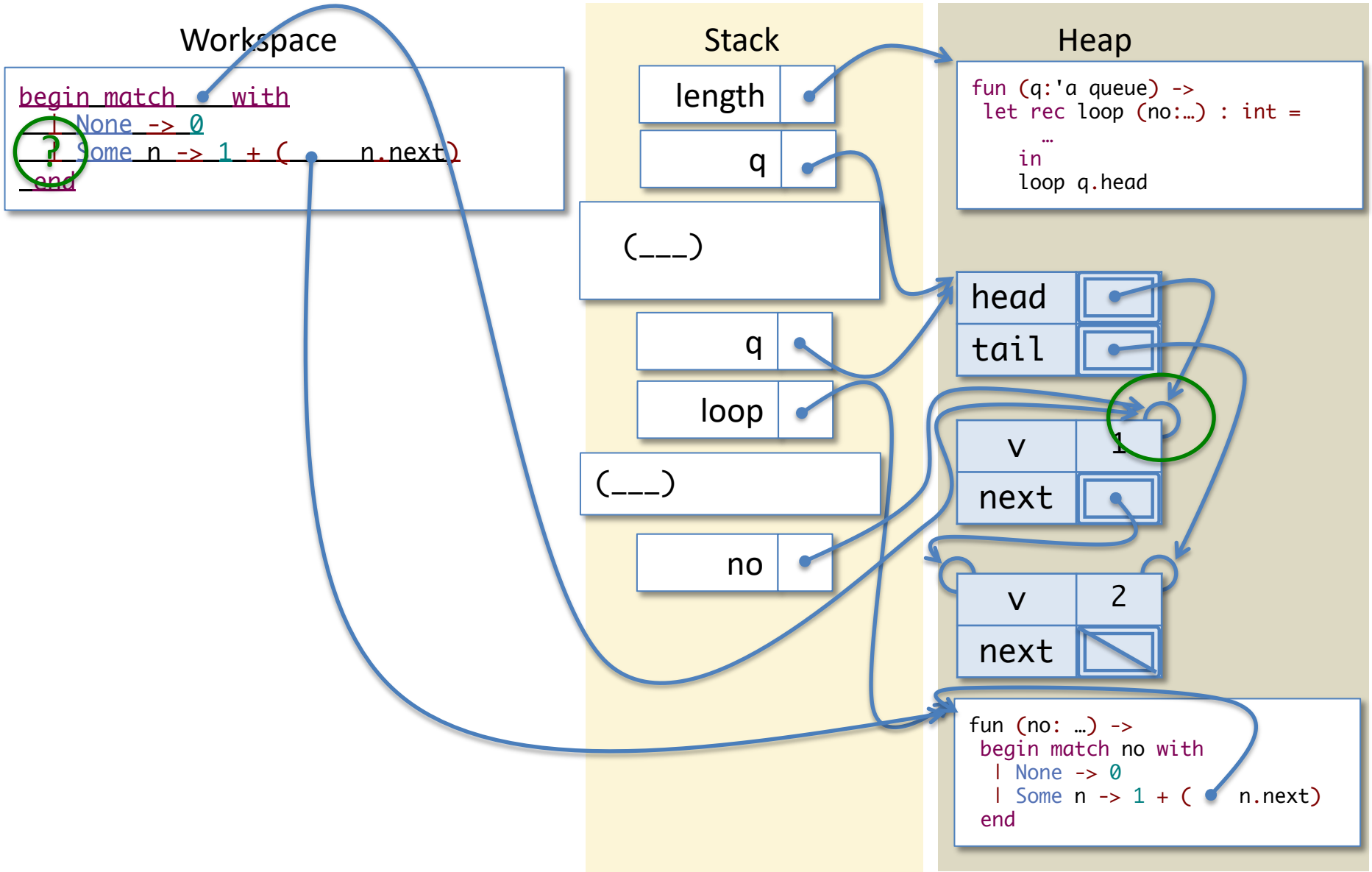
Evaluating length



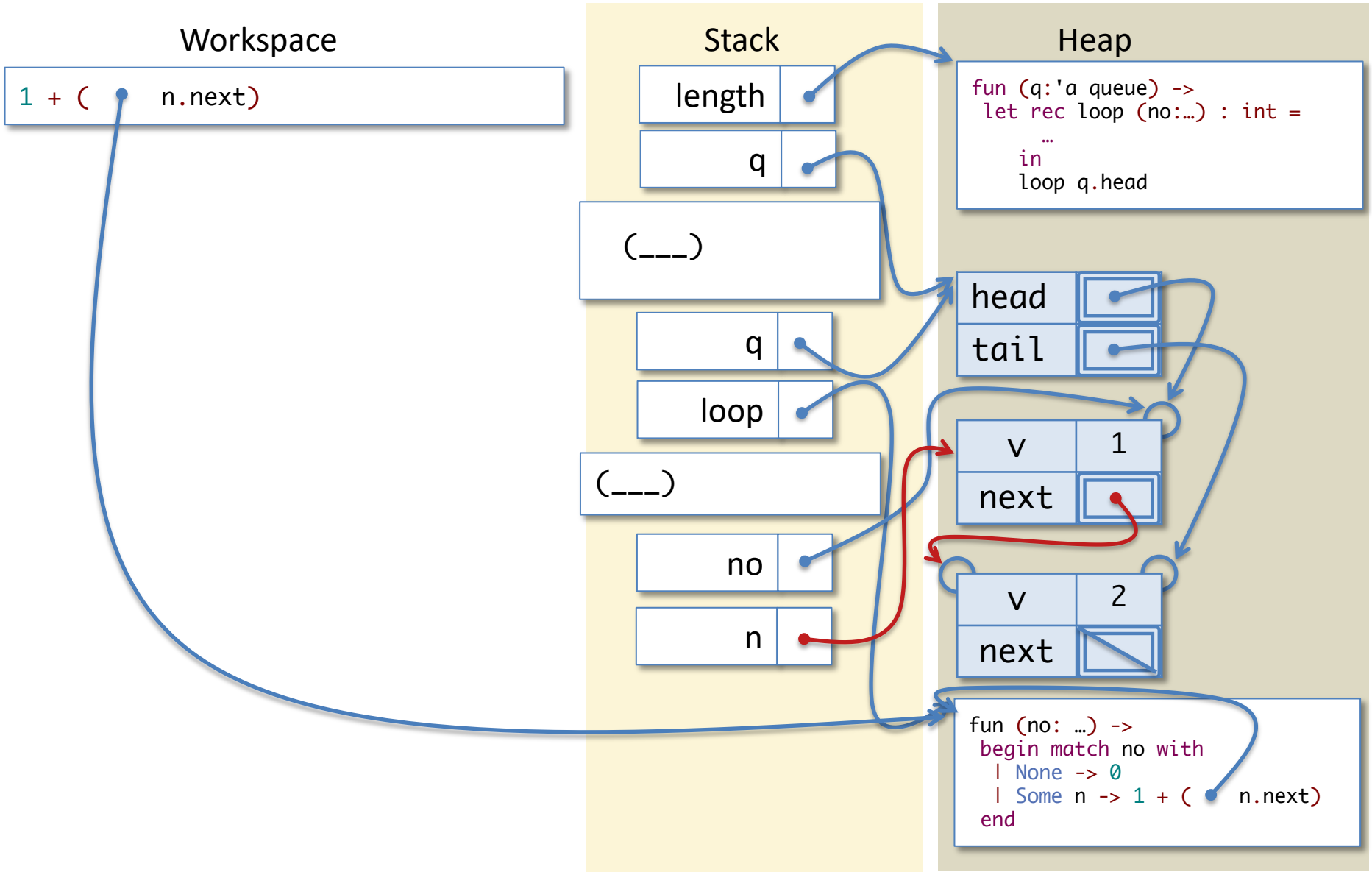
Evaluating length



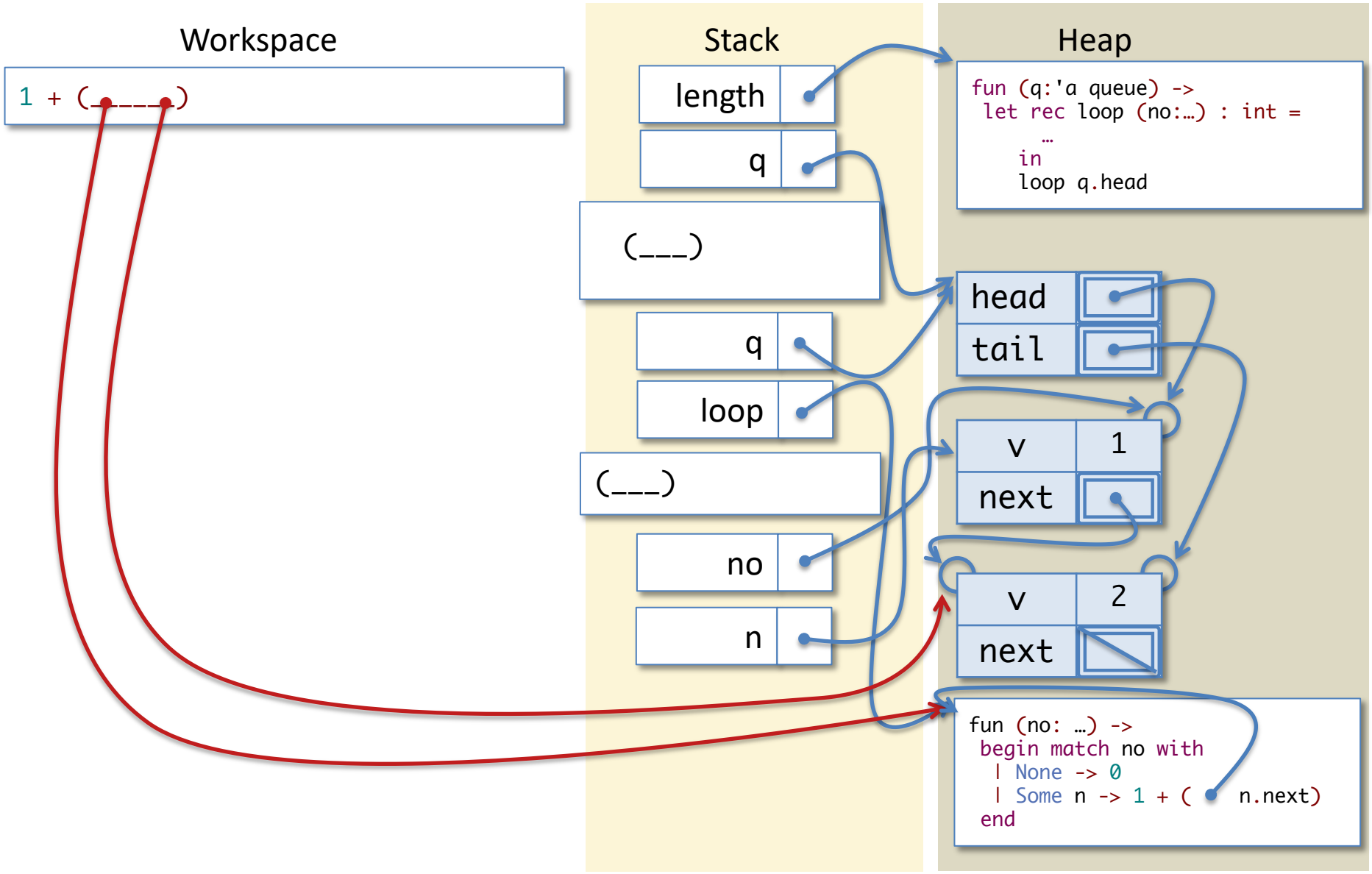
Evaluating length



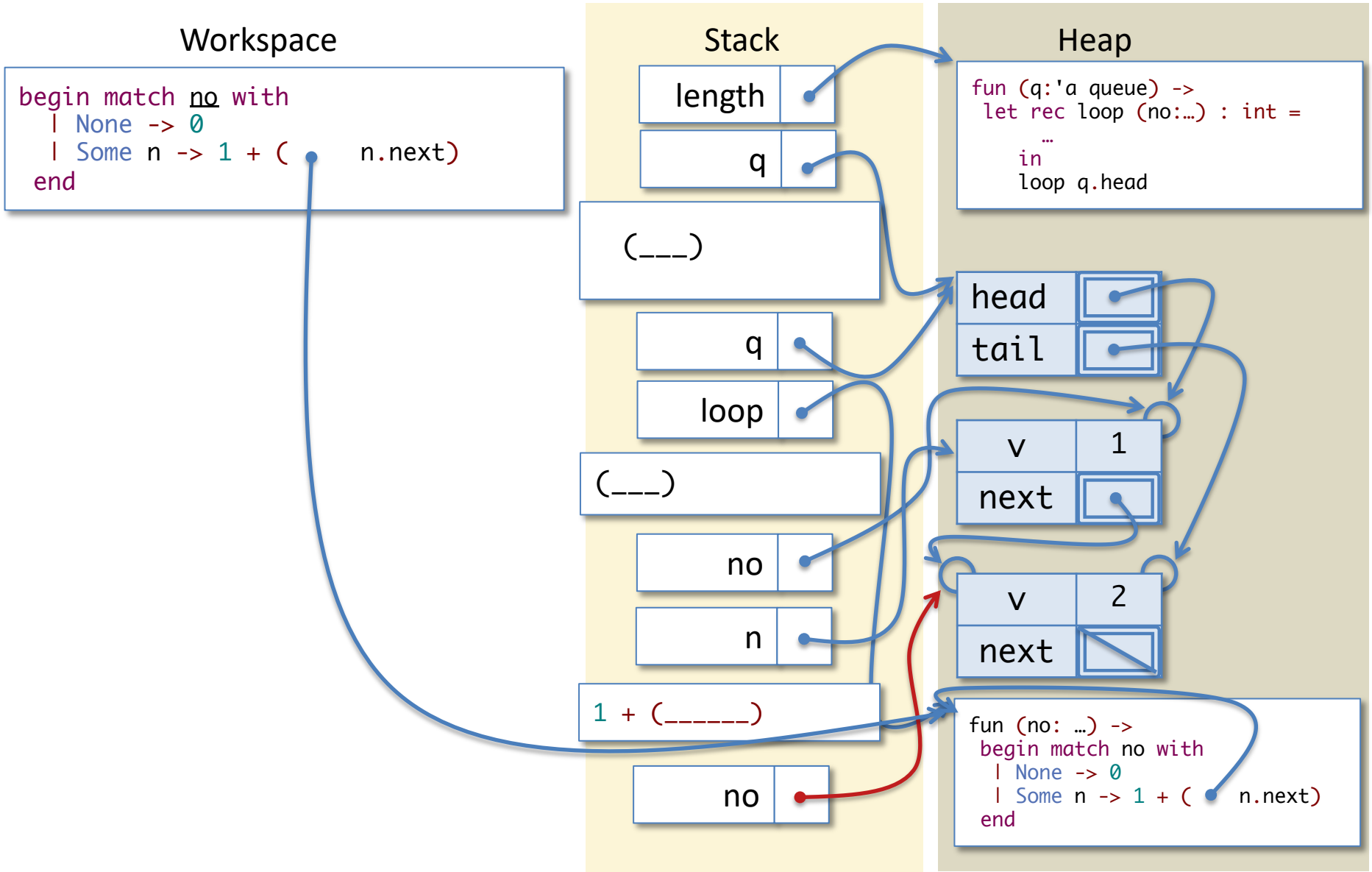
Evaluating length



Evaluating length

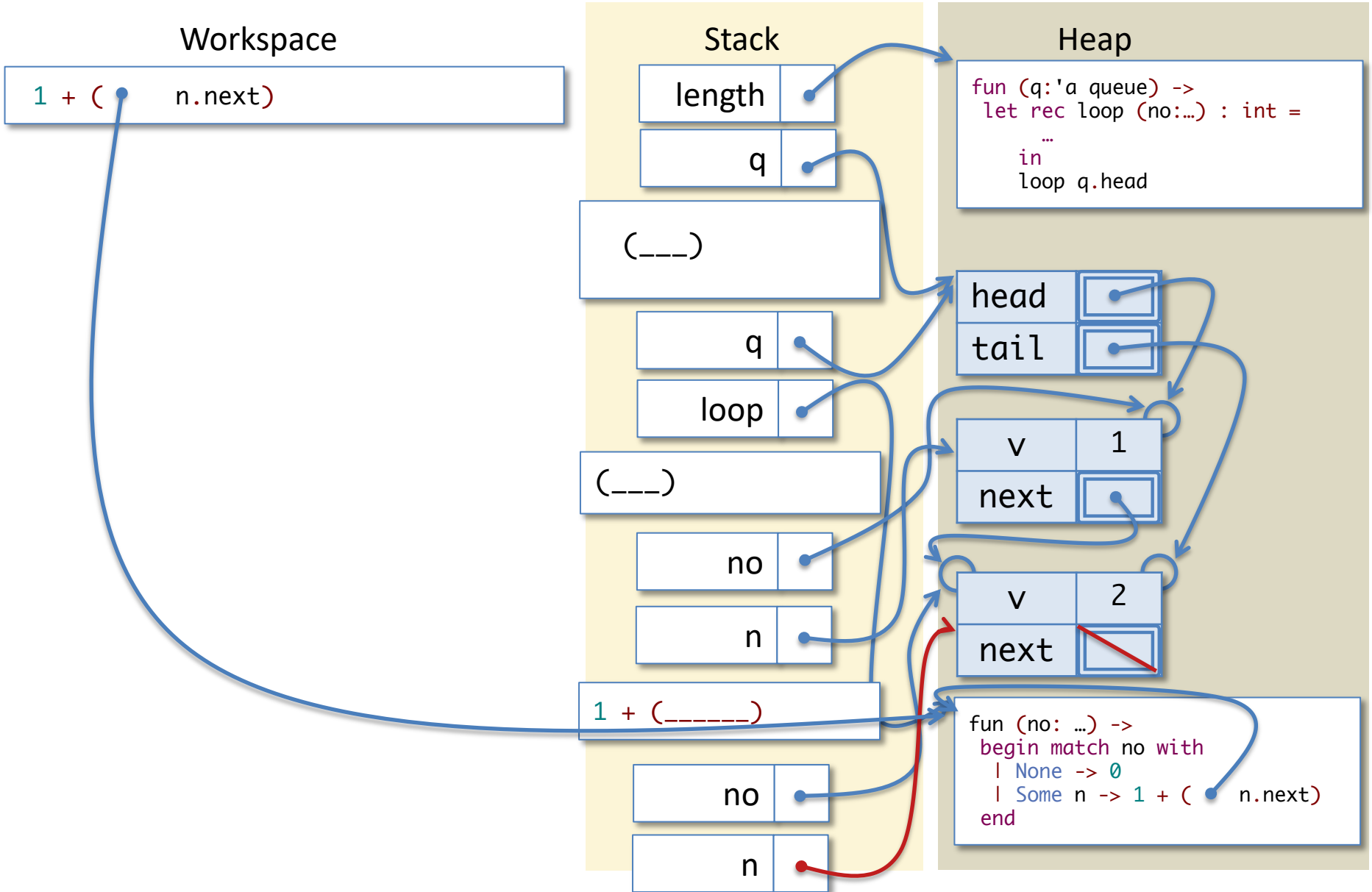


Evaluating length



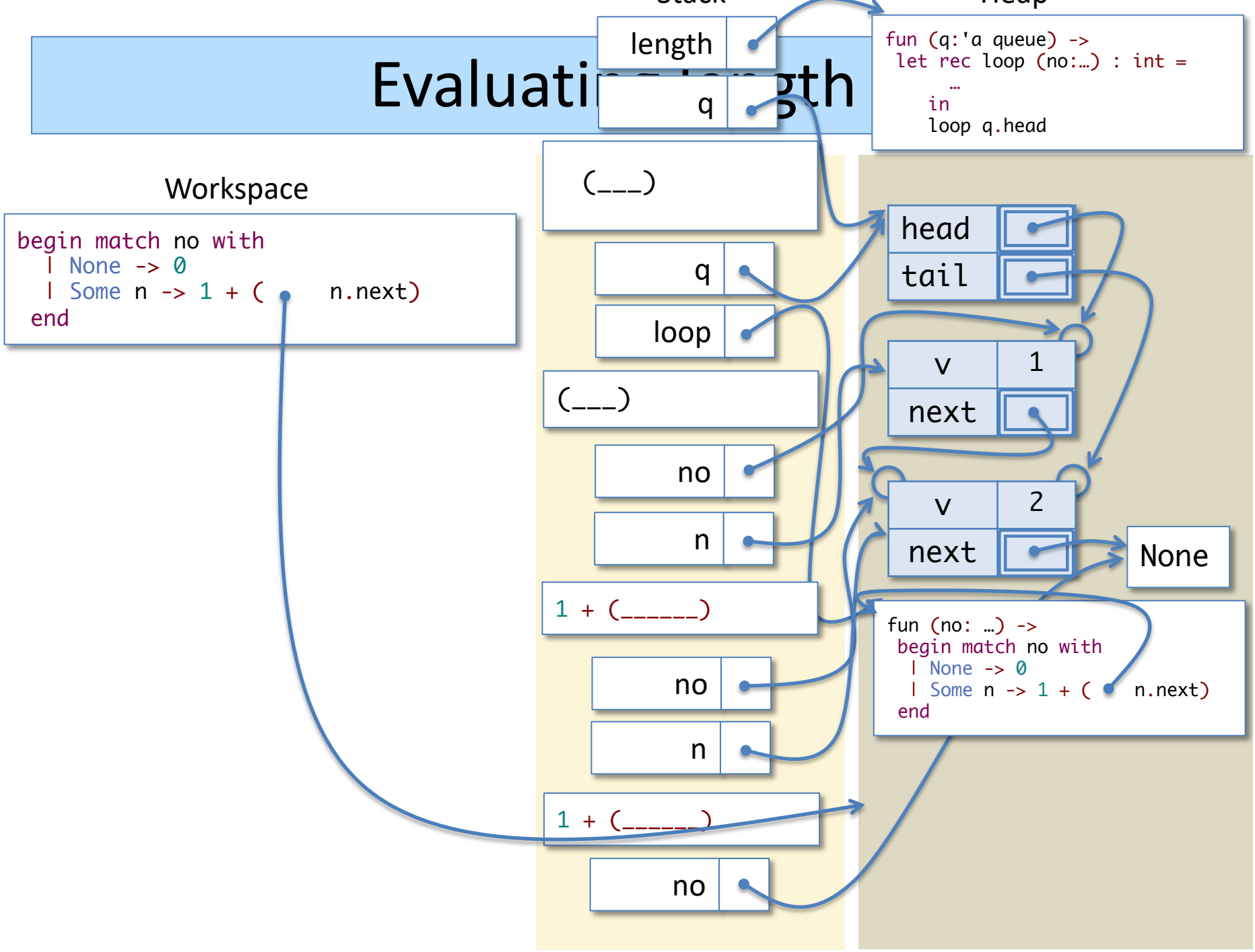
…after a few more steps…

Evaluating length

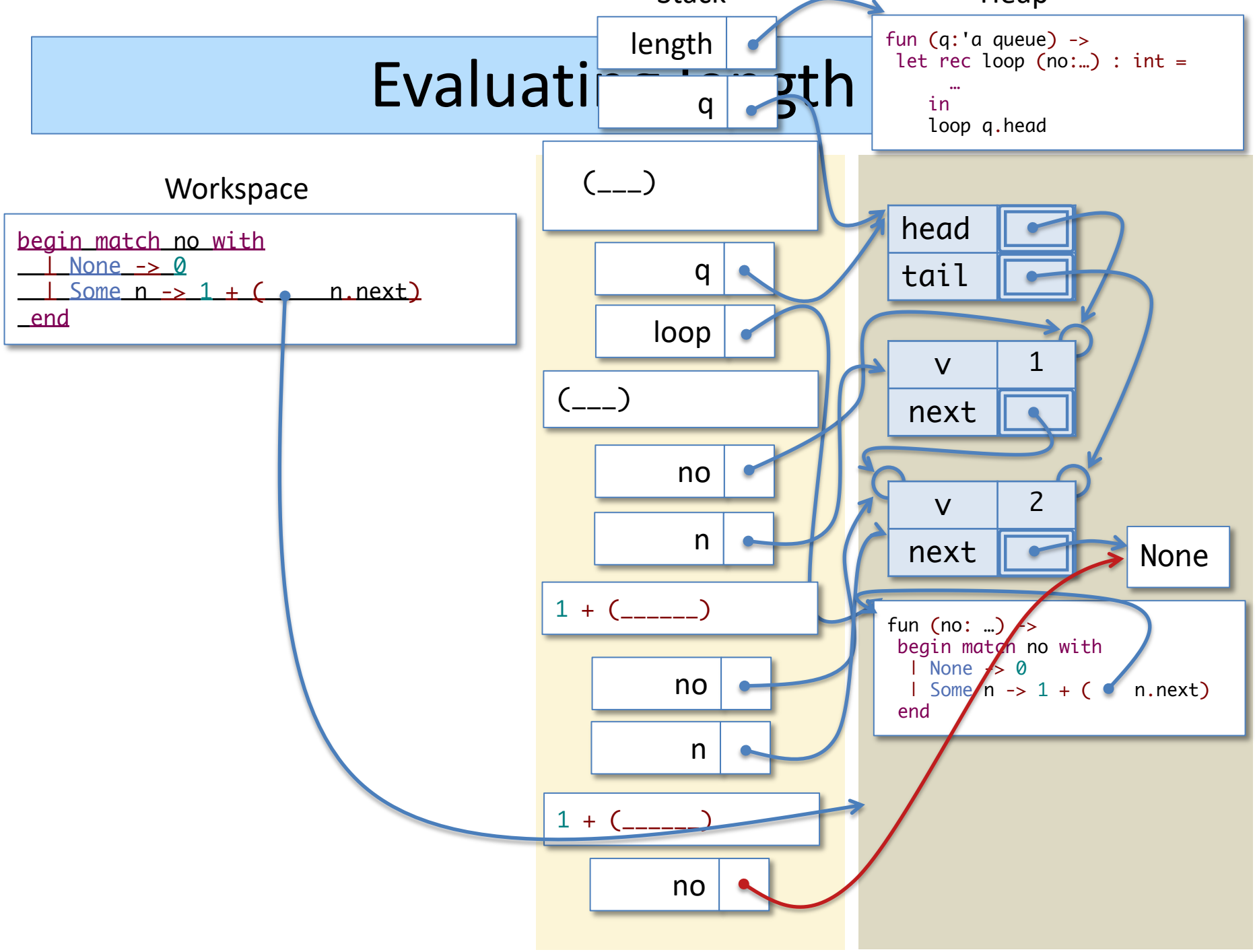


…after a few more steps…

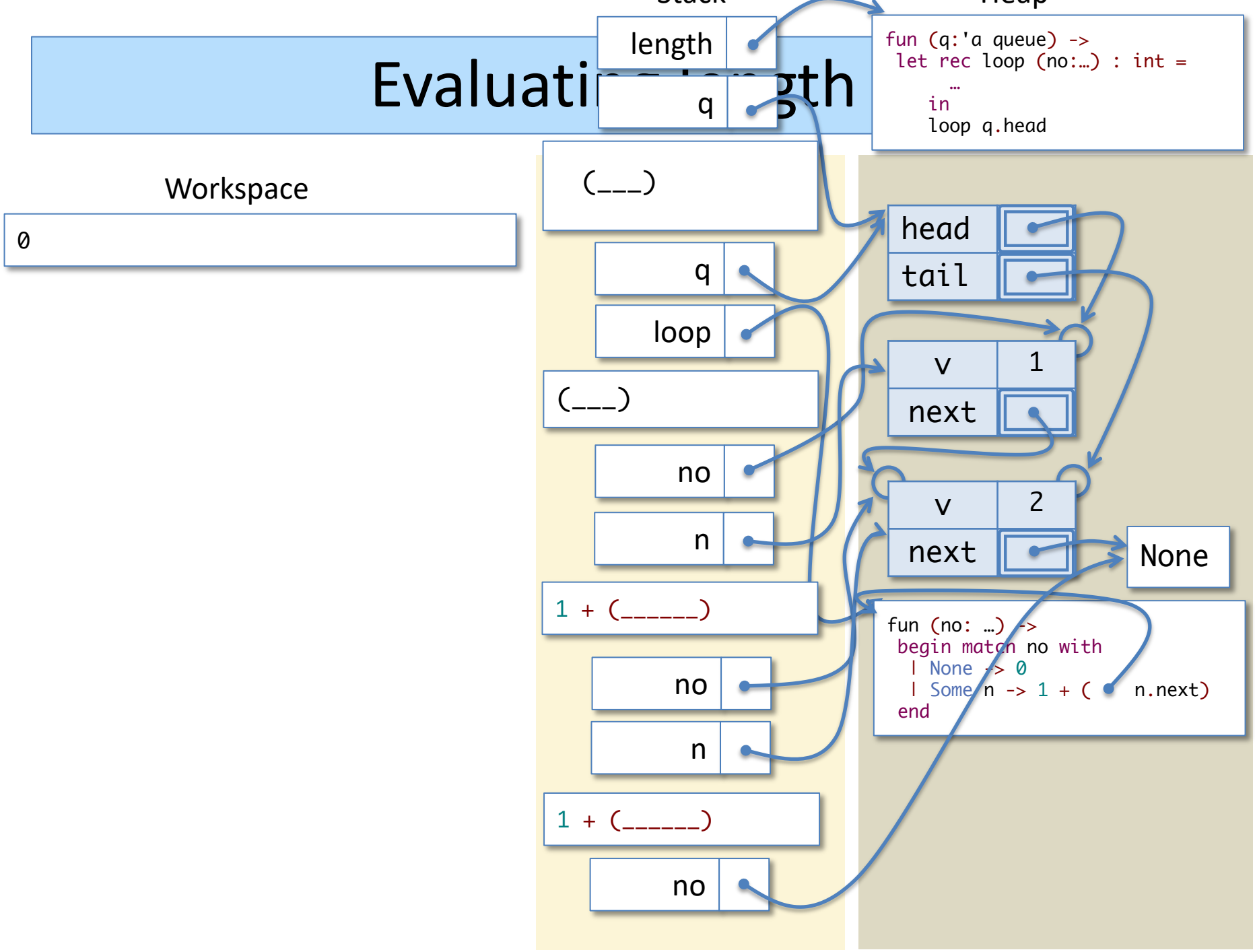
Evaluati gth



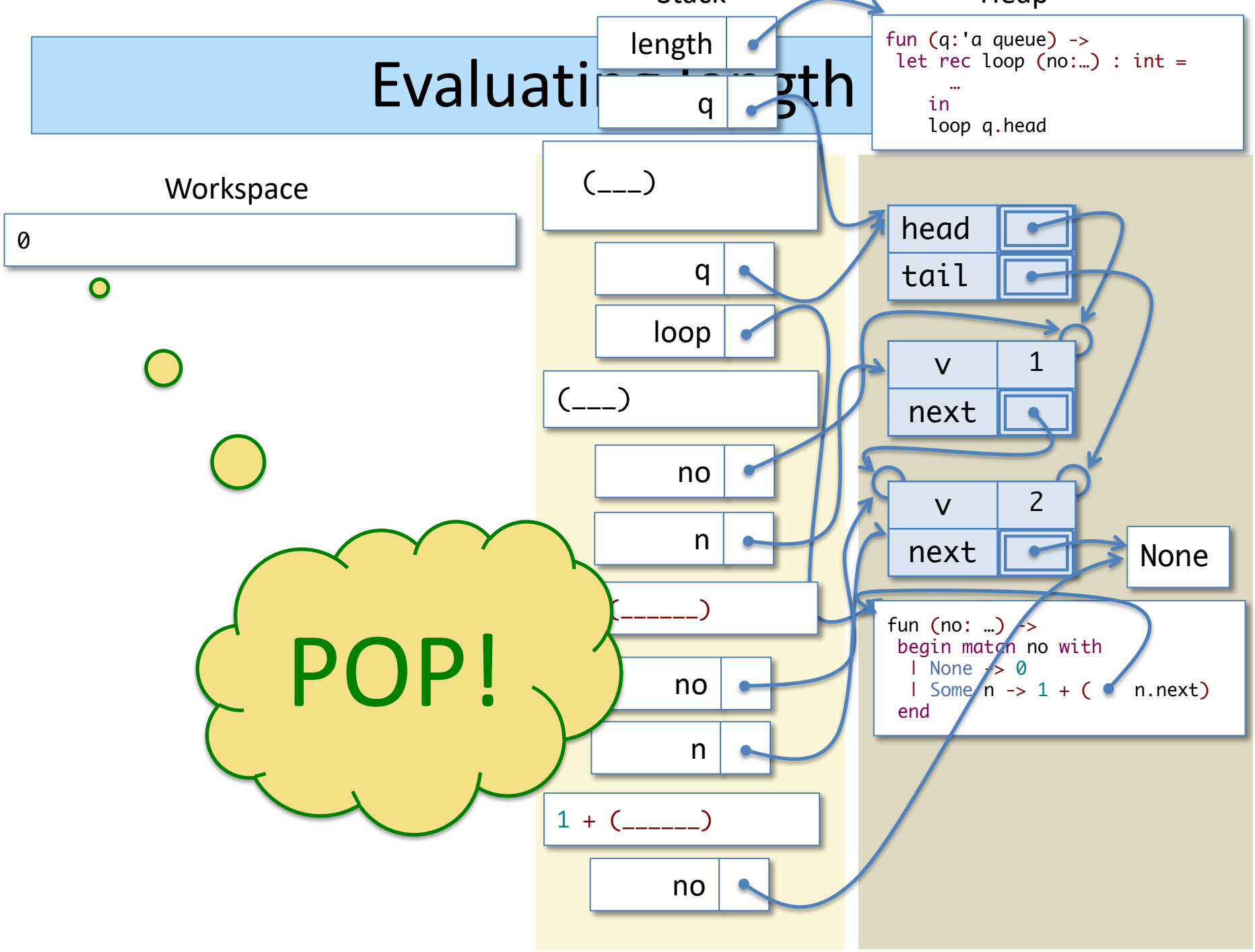
Evaluating length



Evaluating length



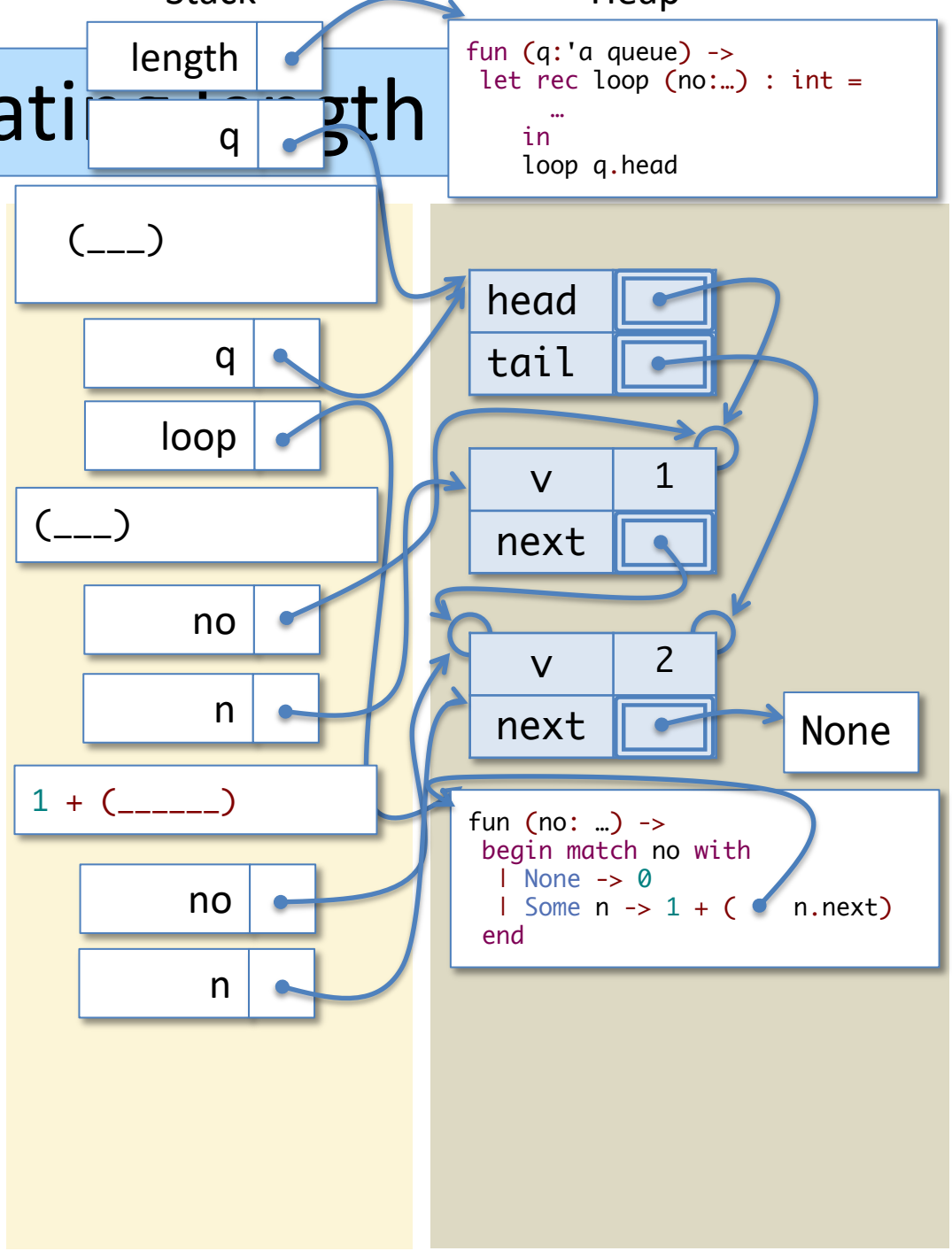
Evaluati gth



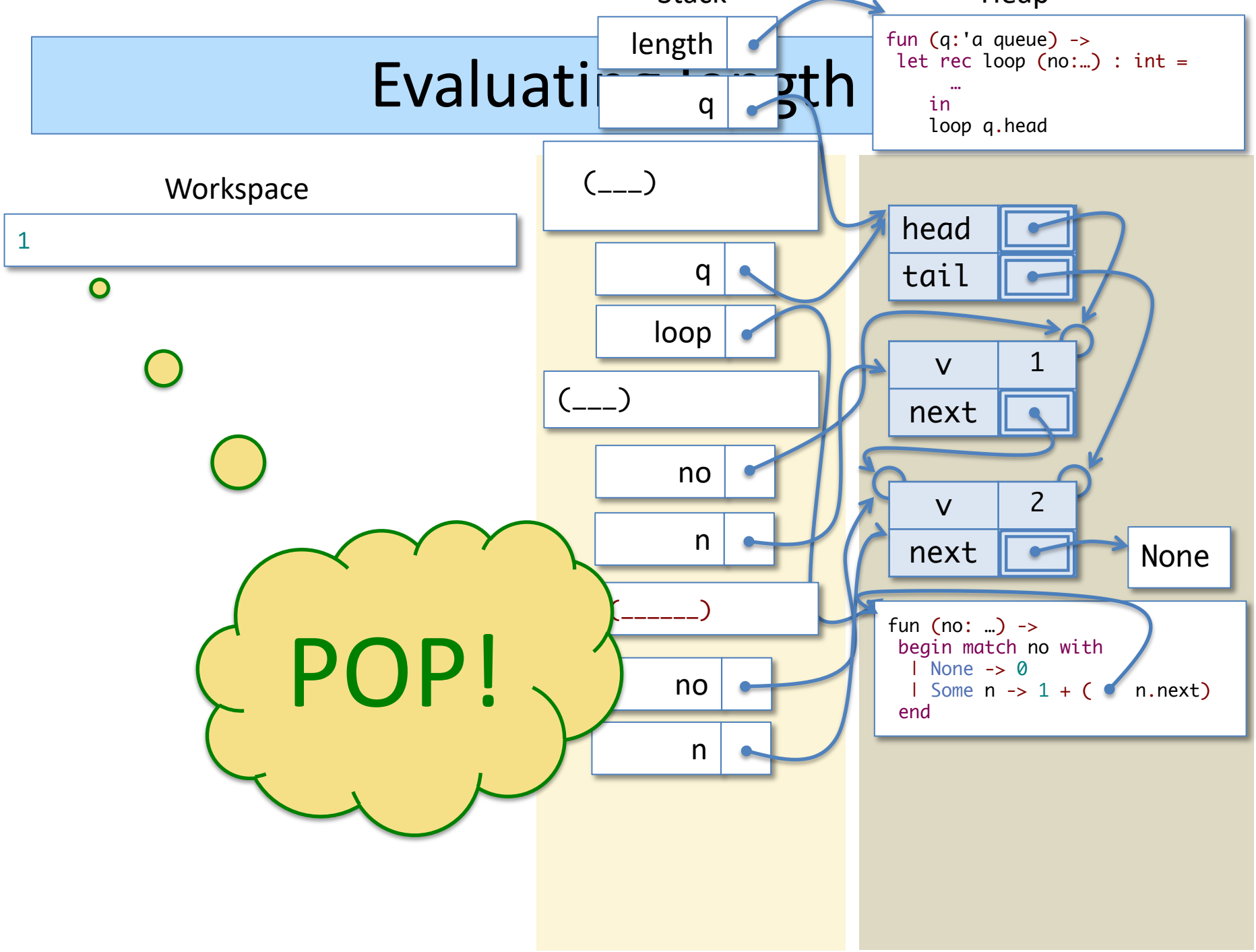
Evaluating length

Workspace

```
1 + 0
```



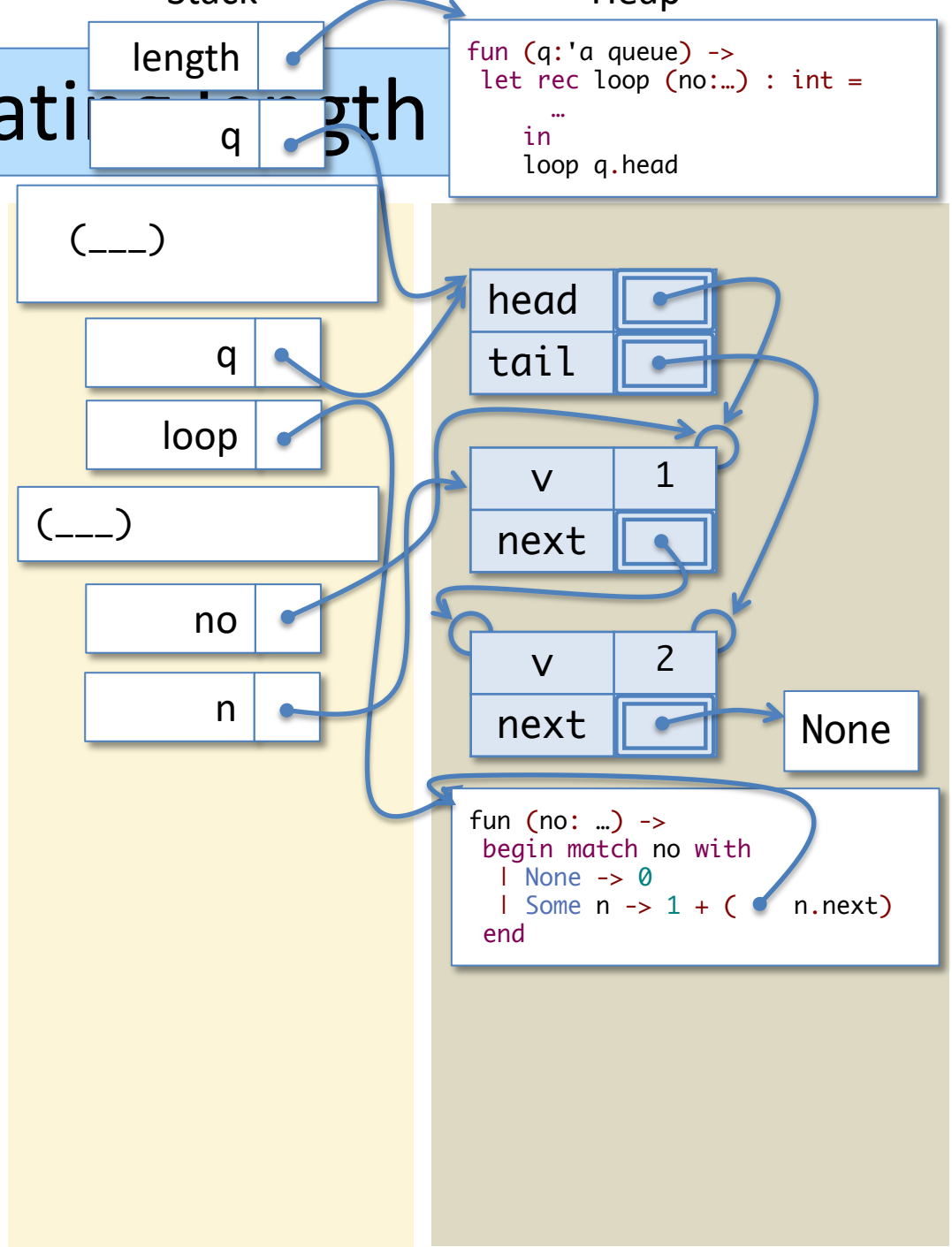
Evaluati gth



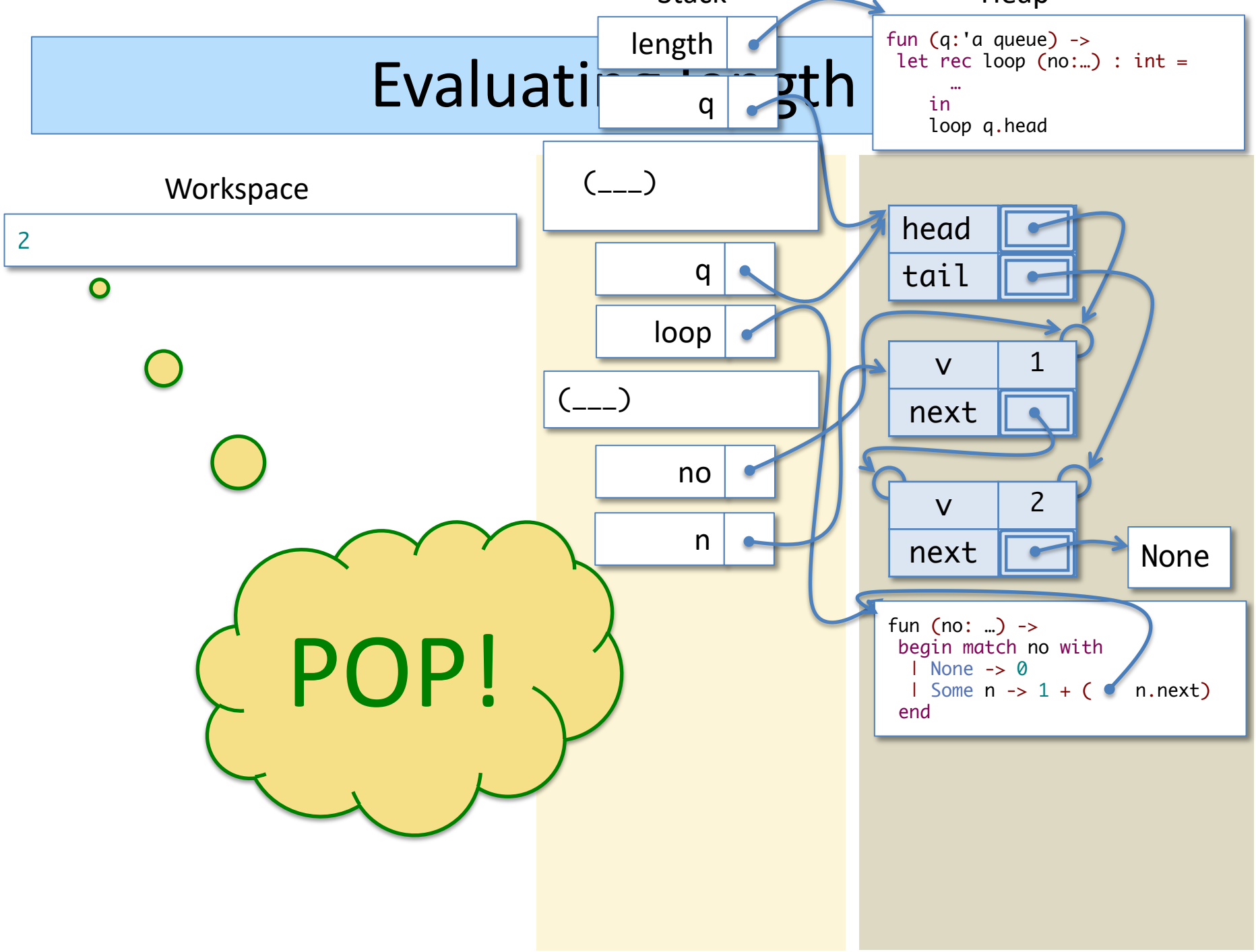
Evaluati gth

Workspace

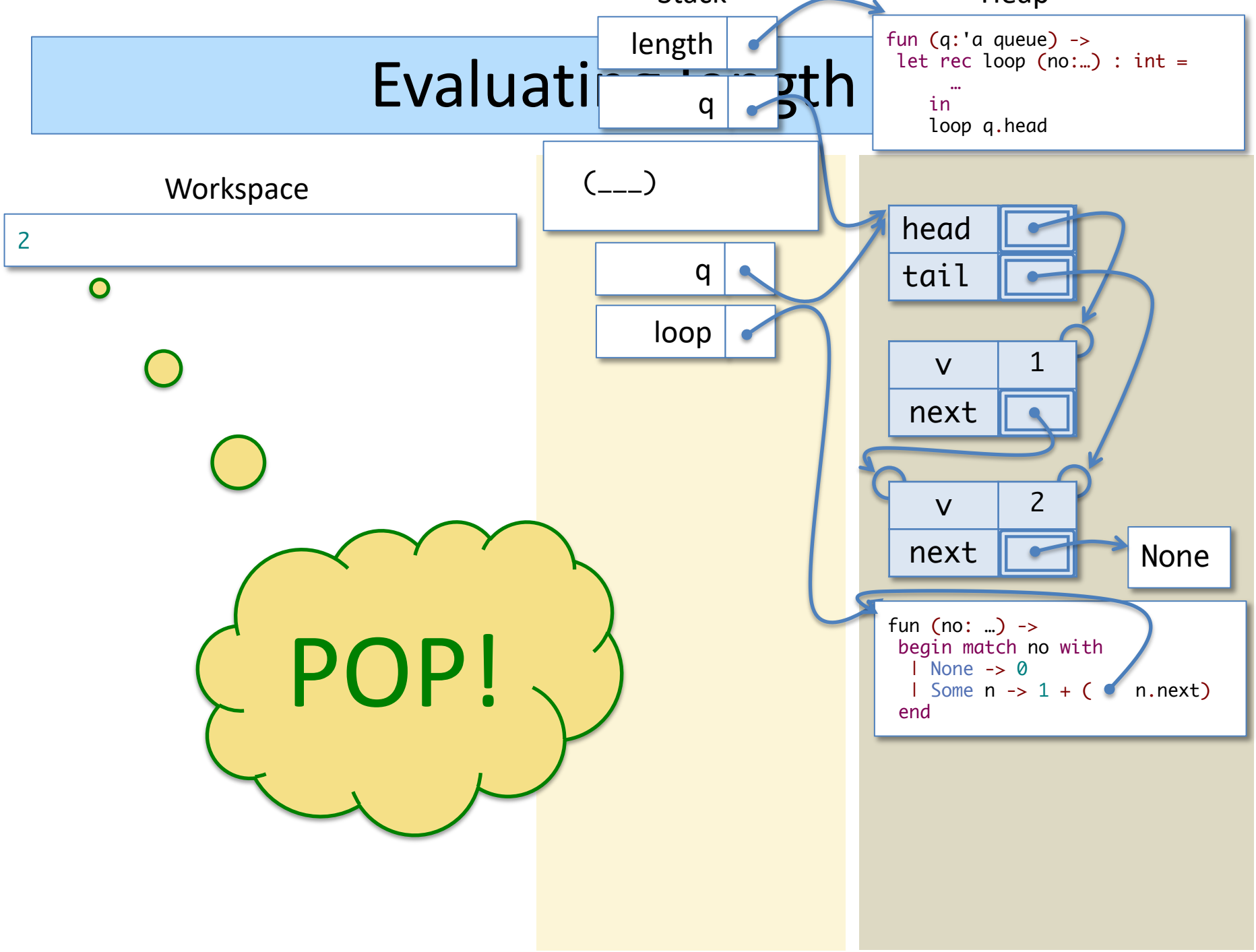
```
1 + 1
```



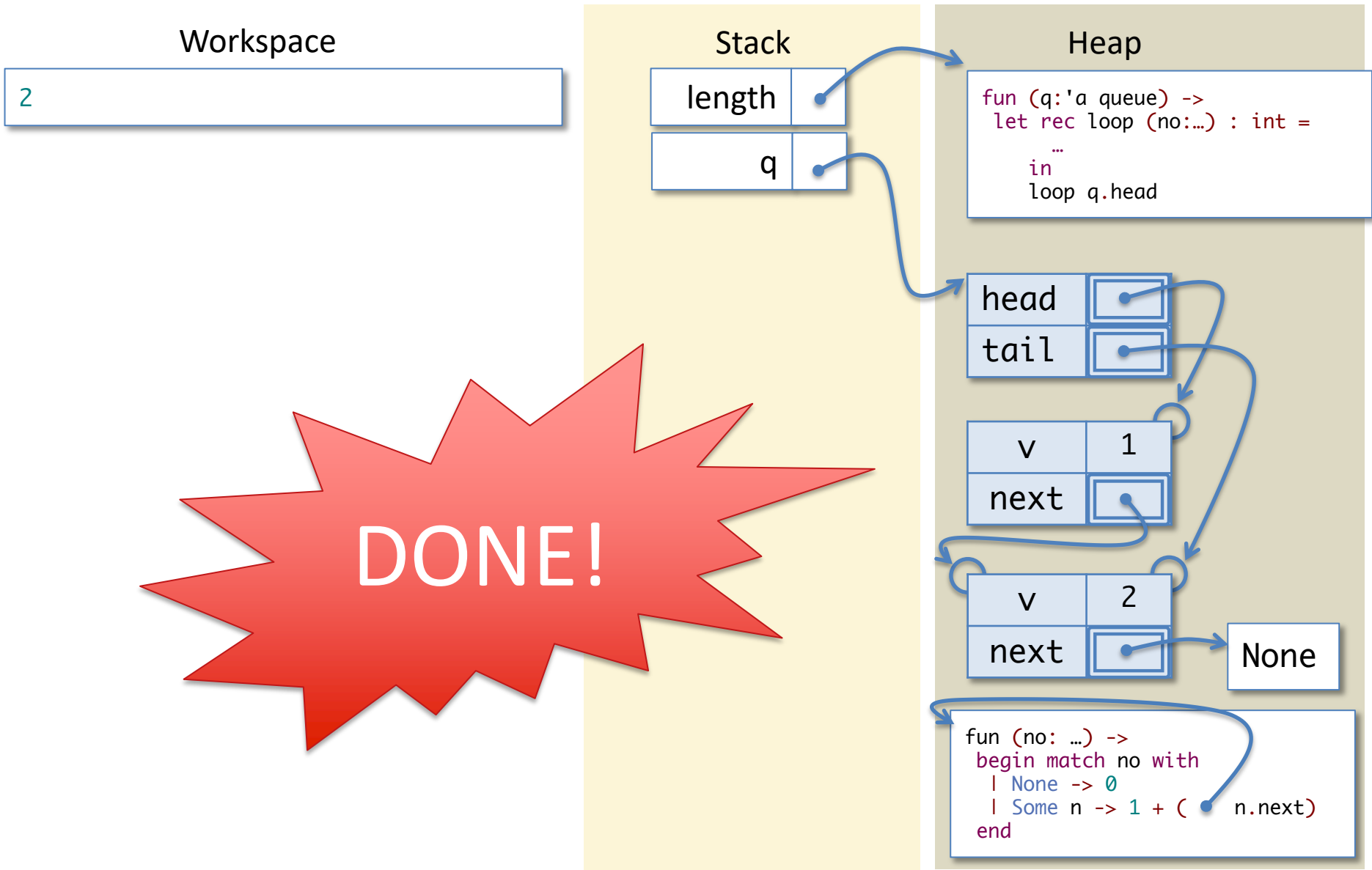
Evaluating length



Evaluati gth



Evaluating length



Iteration

Using tail calls for loops

length (recursively)

```
(* Calculate the length of the queue recursively *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

As we've just seen, this implementation uses a lot of stack space if q is large.

Can we do better?

`length` (using iteration)

```
(* Calculate the length of the list using iteration *)
let length (q:'a queue) : int =
  let rec loop (no:'a qnode option) (len:int) : int =
      begin match no with
        | None -> len
        | Some n -> loop n.next (1+len)
      end
  in
    loop q.head 0
```

This implementation of `length` also uses a helper function, `loop`:

- This loop takes an extra argument, `len`, called the *accumulator*
- Unlike the previous solution, the computation happens "on the way down" as opposed to "on the way back up"
- Note that `loop` will always be called in an otherwise-empty workspace—the results of the call to `loop` never need to be used to compute another expression. In contrast, we had (1 + (`loop` …)) in the recursive version.

“Tail Call Optimization”

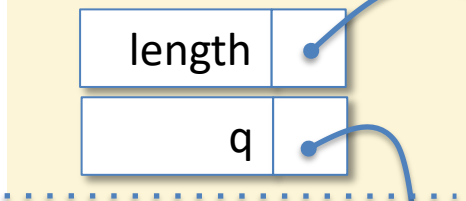
- Question: Why does it matter that ‘loop’ is only called in an empty workspace?
- Answer: We can *optimize* the behavior of the abstract stack machine in this case!
  - The workspace pushed onto the stack tells us “what to do” when the function call returns. If empty, "what to do" is pop immediately.
  - If there is nothing to do after a function call, we are done with the current set of local variables – so pop them early to save space.
  - Plus, no need to save the empty workspace either.
- The upshot is that we can execute a tail recursion just like a ‘for’ loop in Java or C, using a *constant* amount of stack space.

Tail Calls and Iterative length

Workspace

```
length q
```

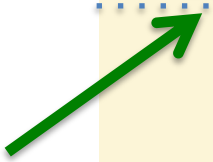
Stack

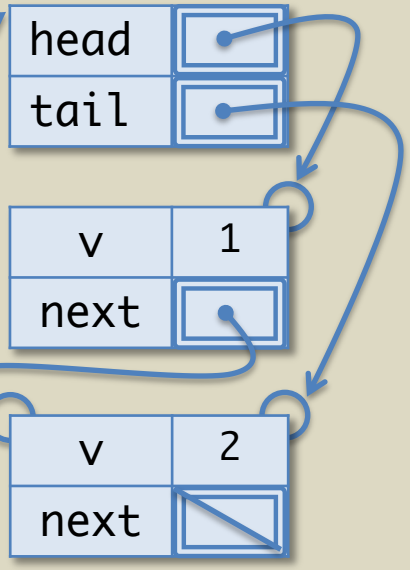


Heap

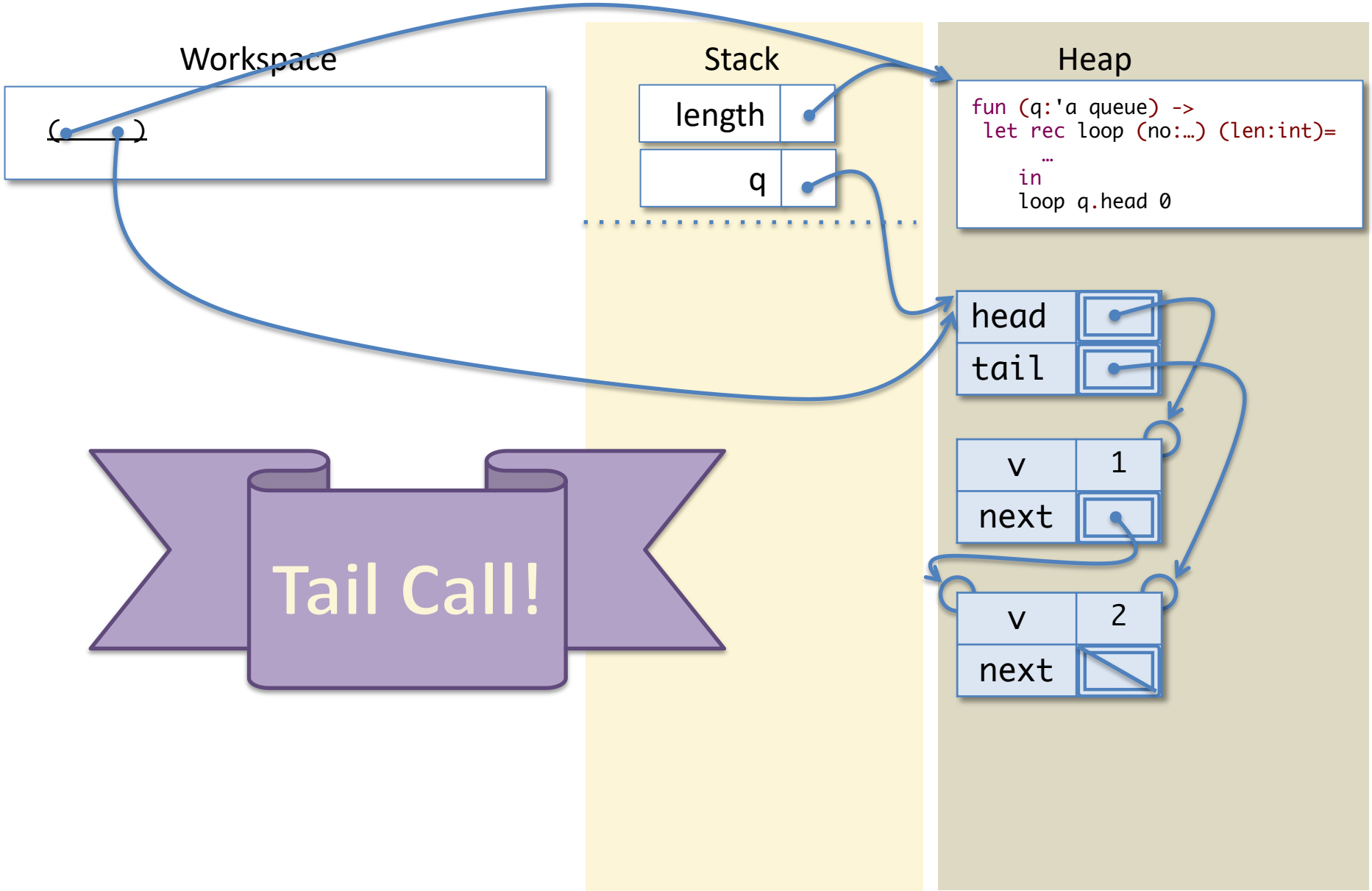
```
fun (q:'a queue) ->
 let rec loop (no:…) (len:int)=
       …
    in
    loop q.head 0
```

Bindings above this line
are top-level declarations.

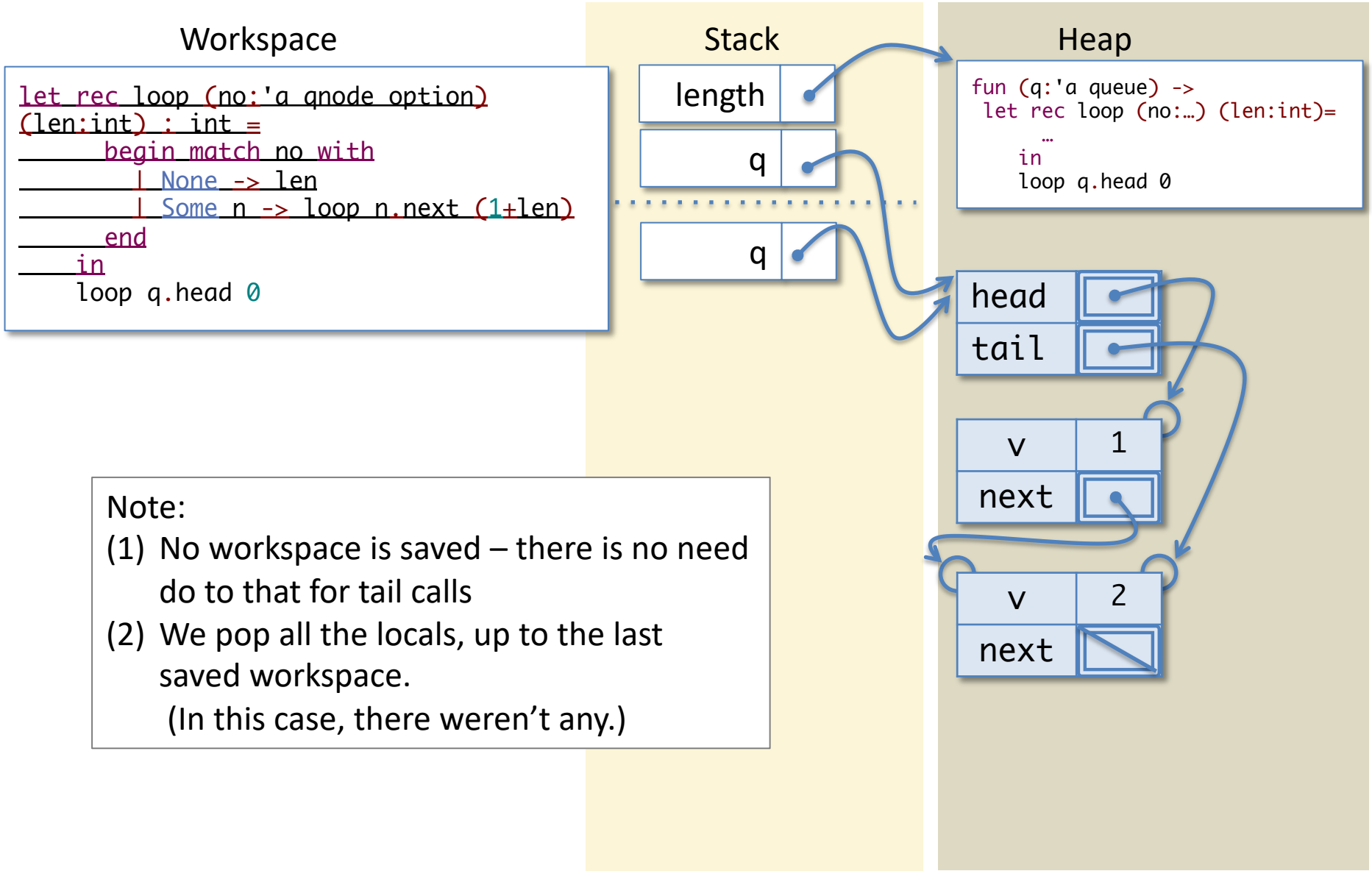




Tail Calls and Iterative length



Tail Calls and Iterative length

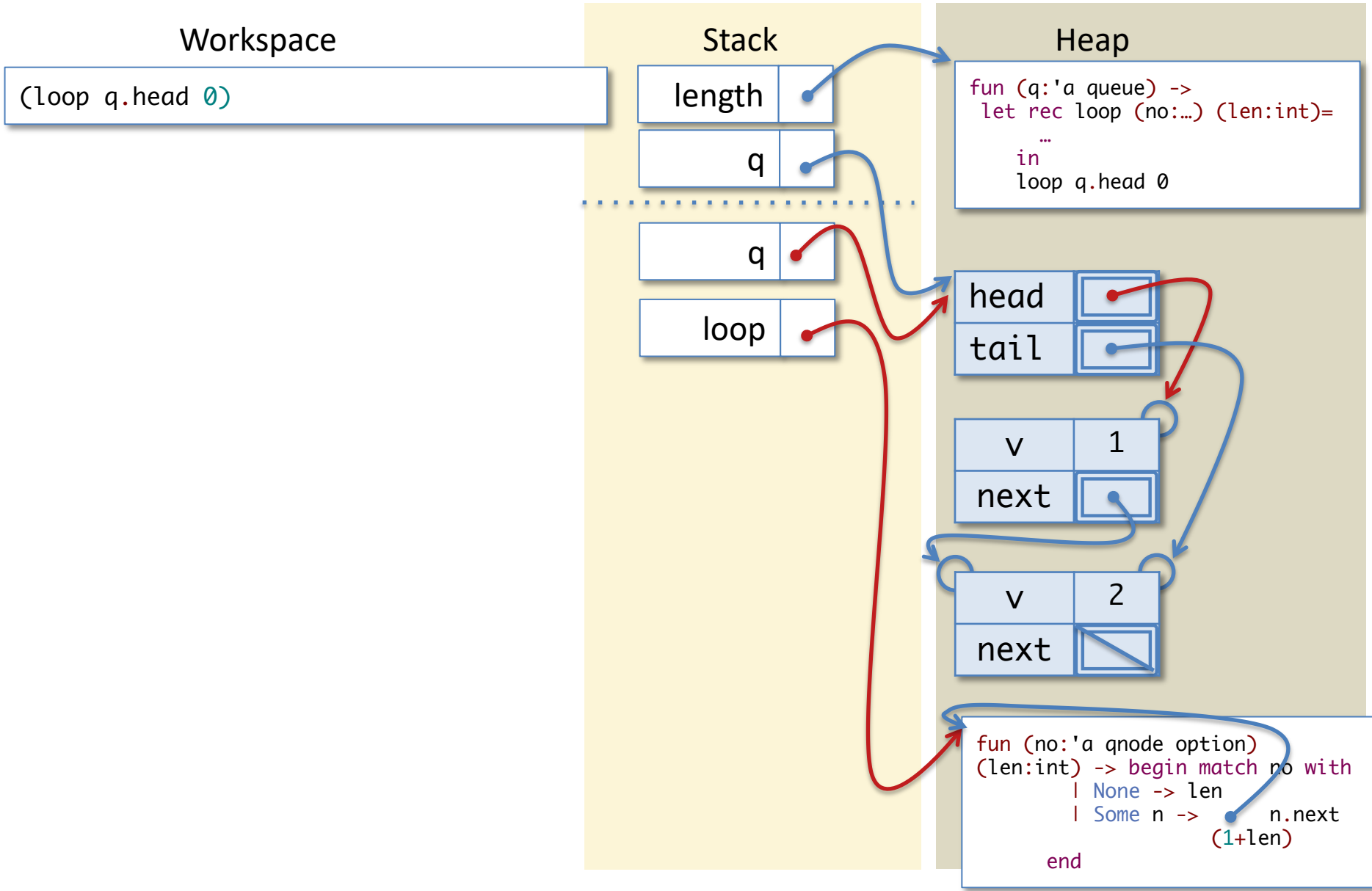


Note:

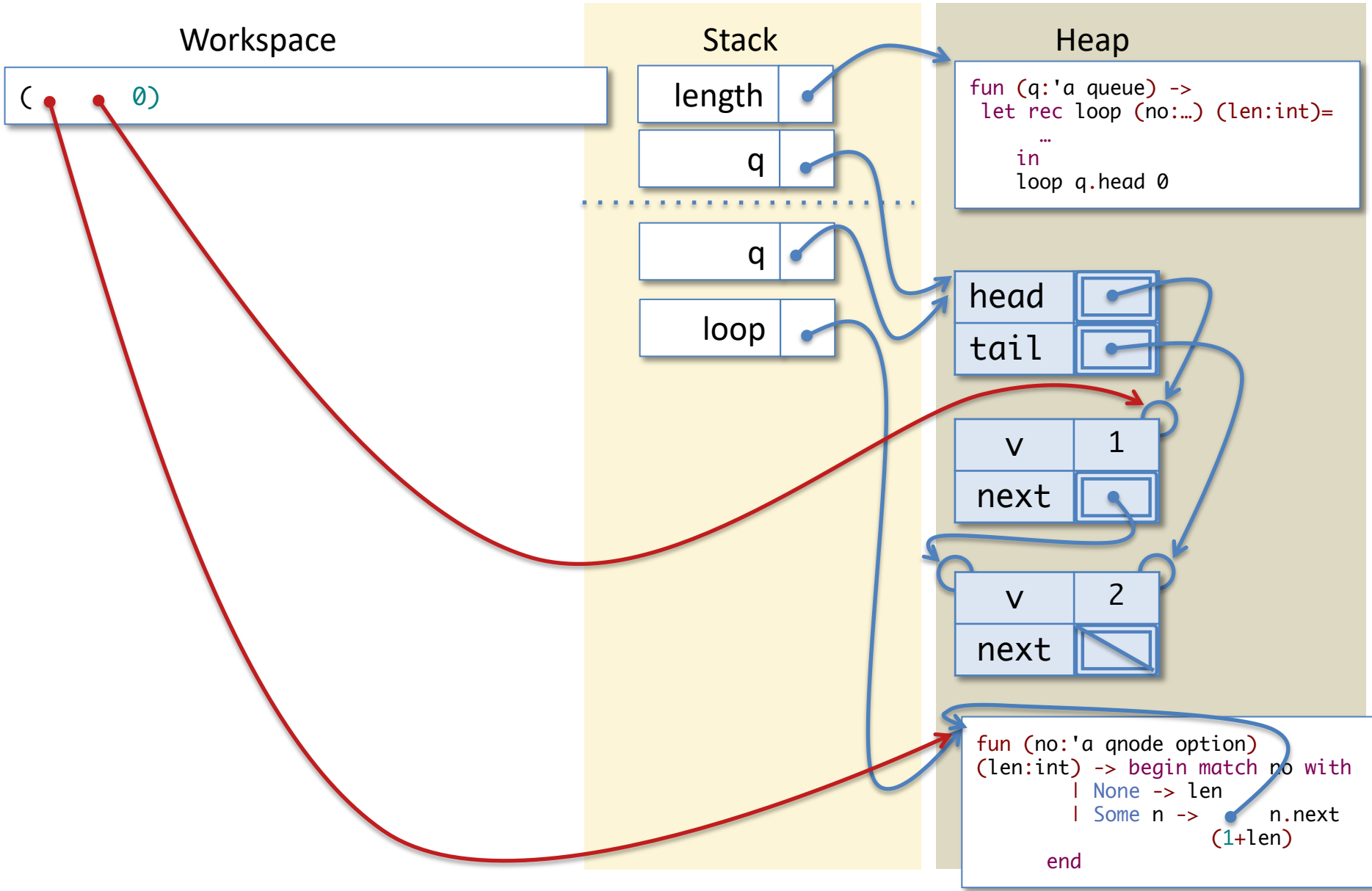
(1) No workspace is saved – there is no need do to that for tail calls

(2) We pop all the locals, up to the last saved workspace.
(In this case, there weren't any.)

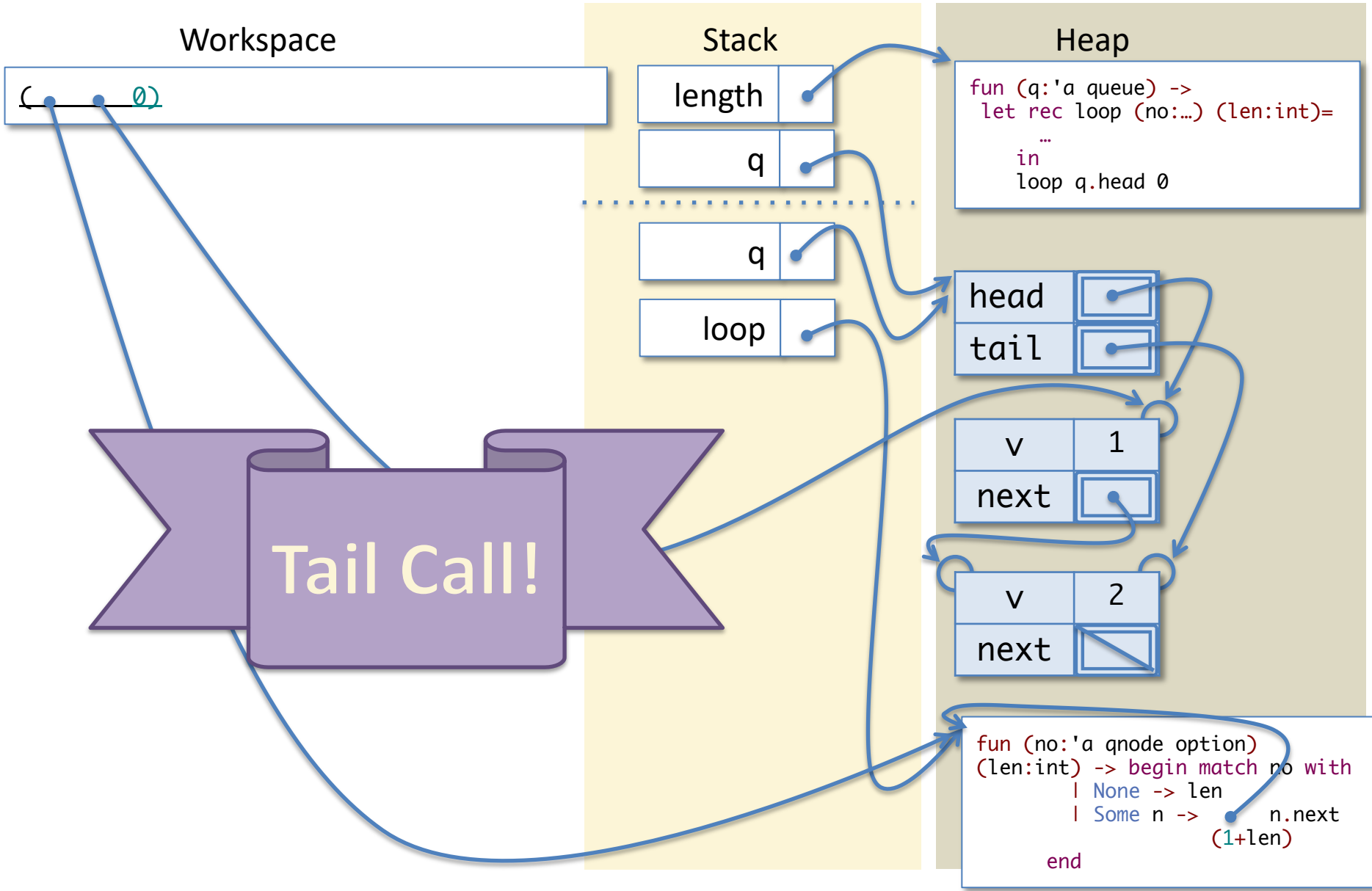
Tail Calls and Iterative length



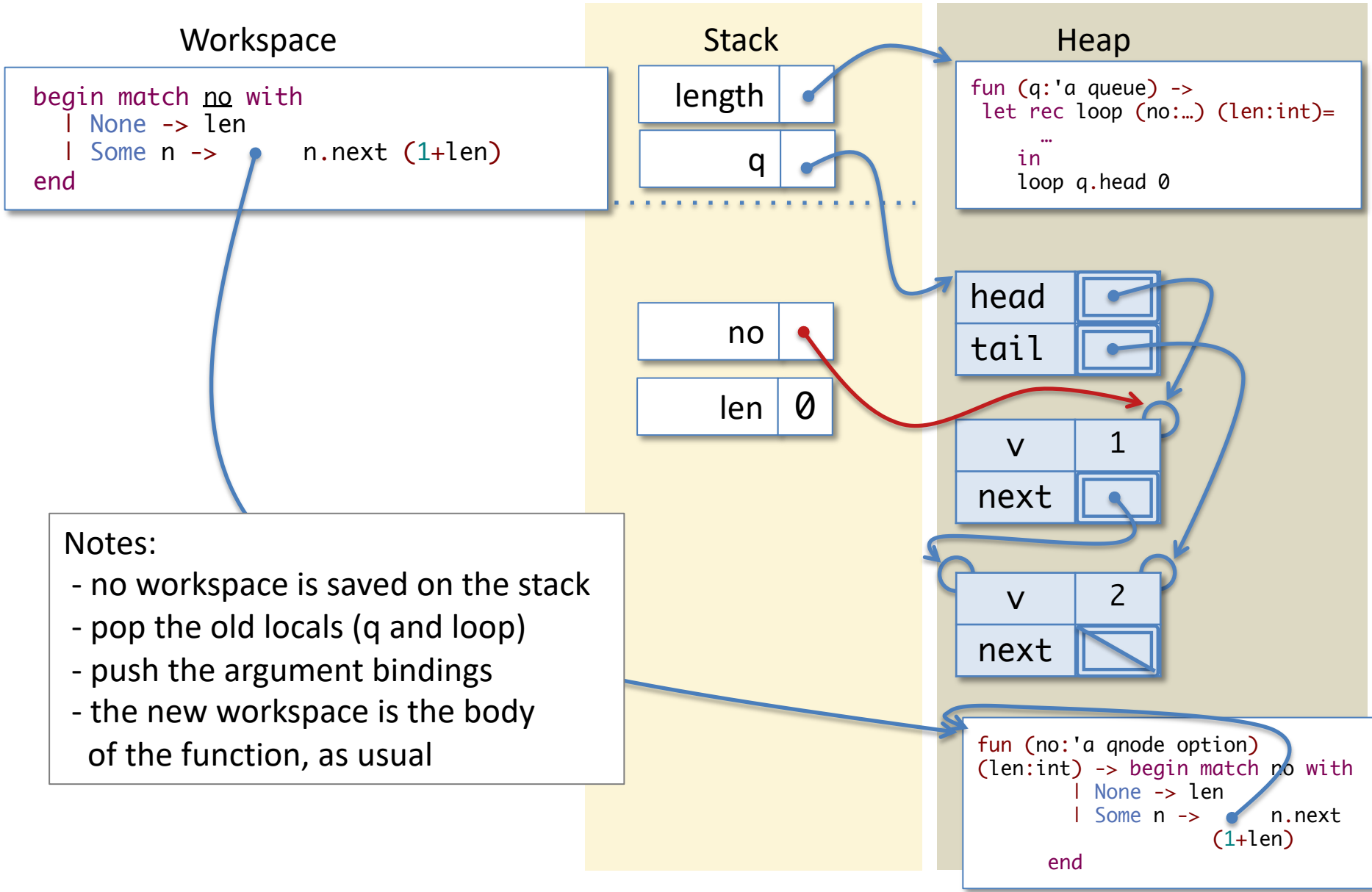
Tail Calls and Iterative length



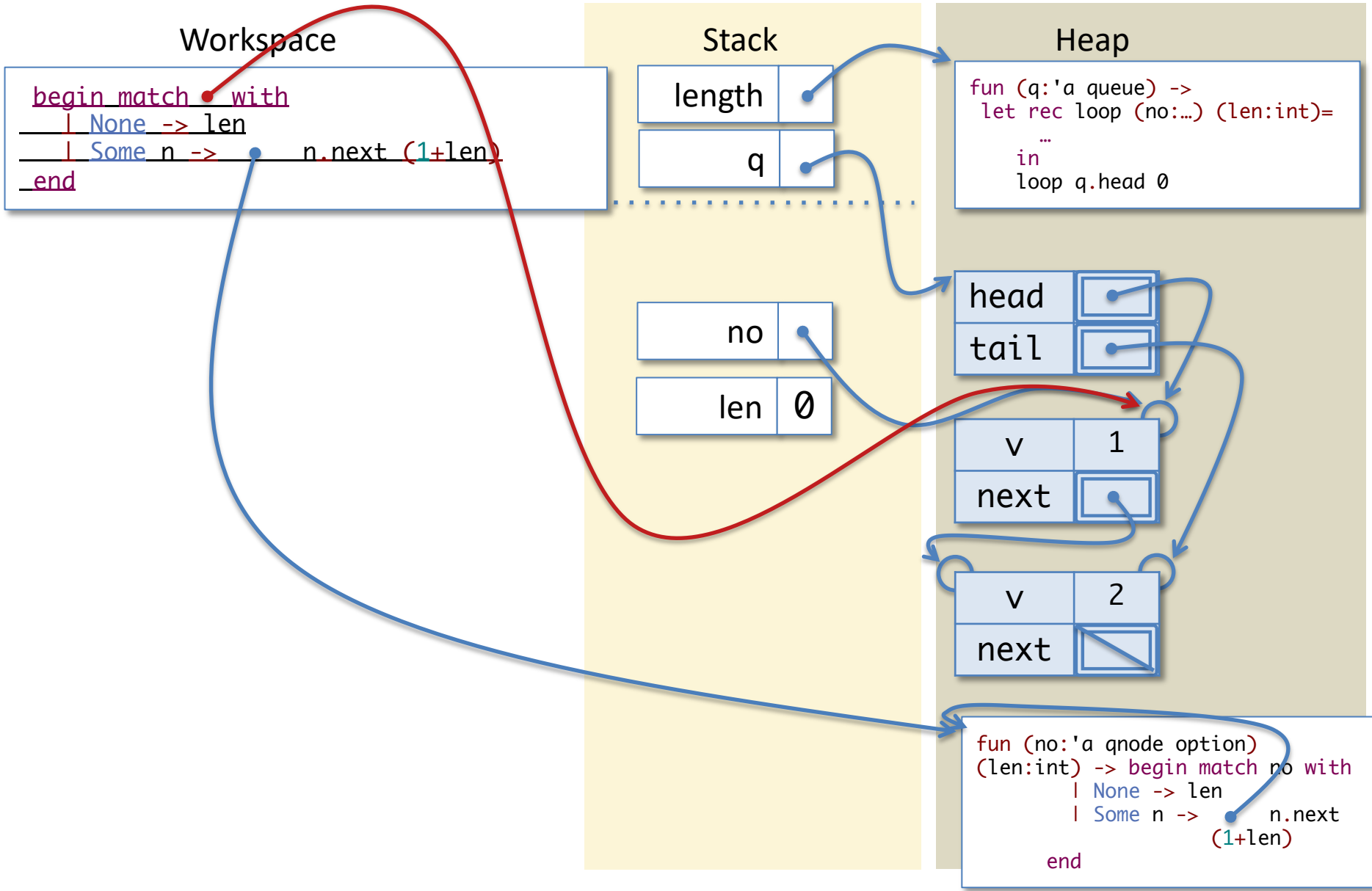
Tail Calls and Iterative length



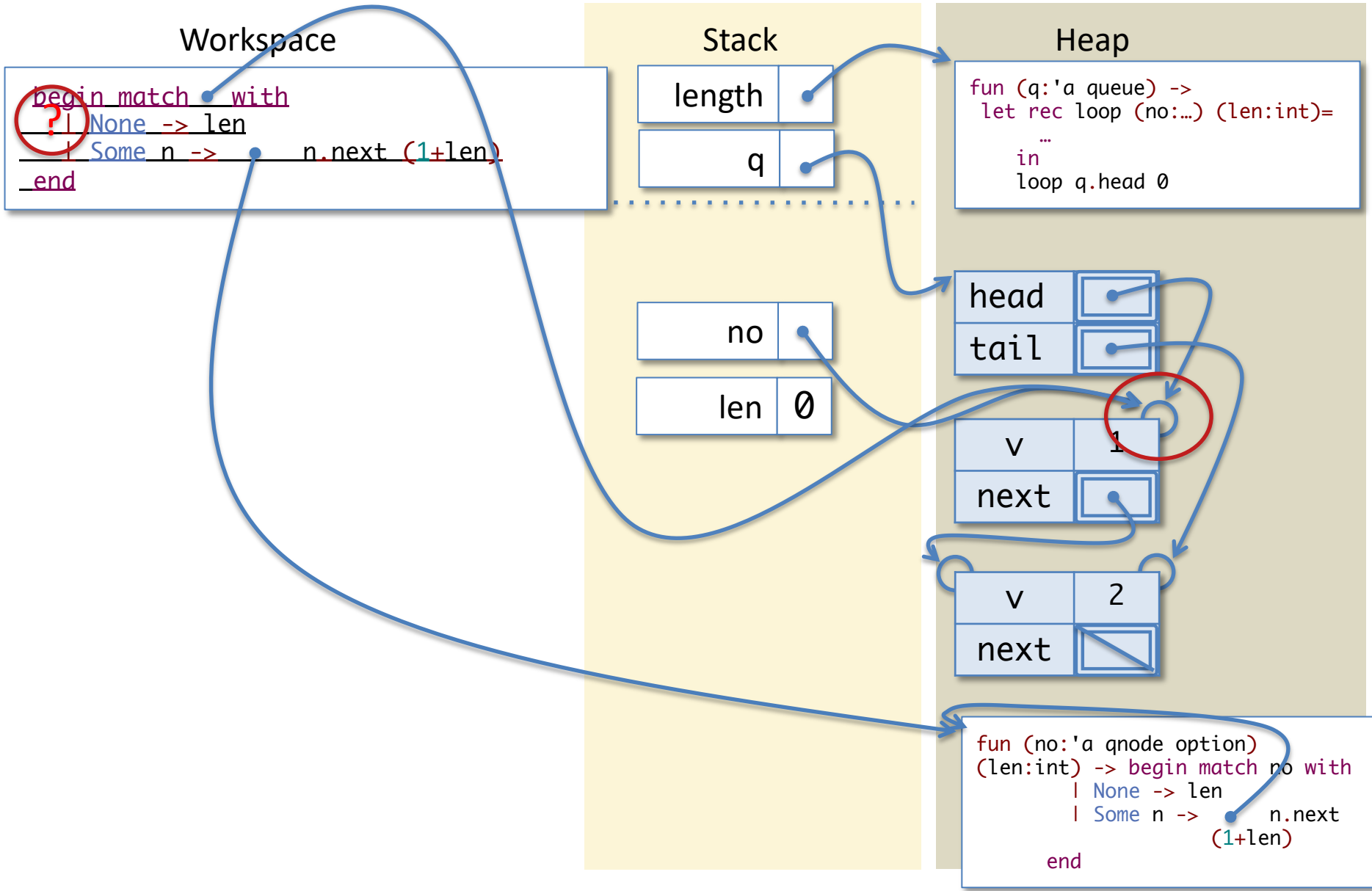
Tail Calls and Iterative length



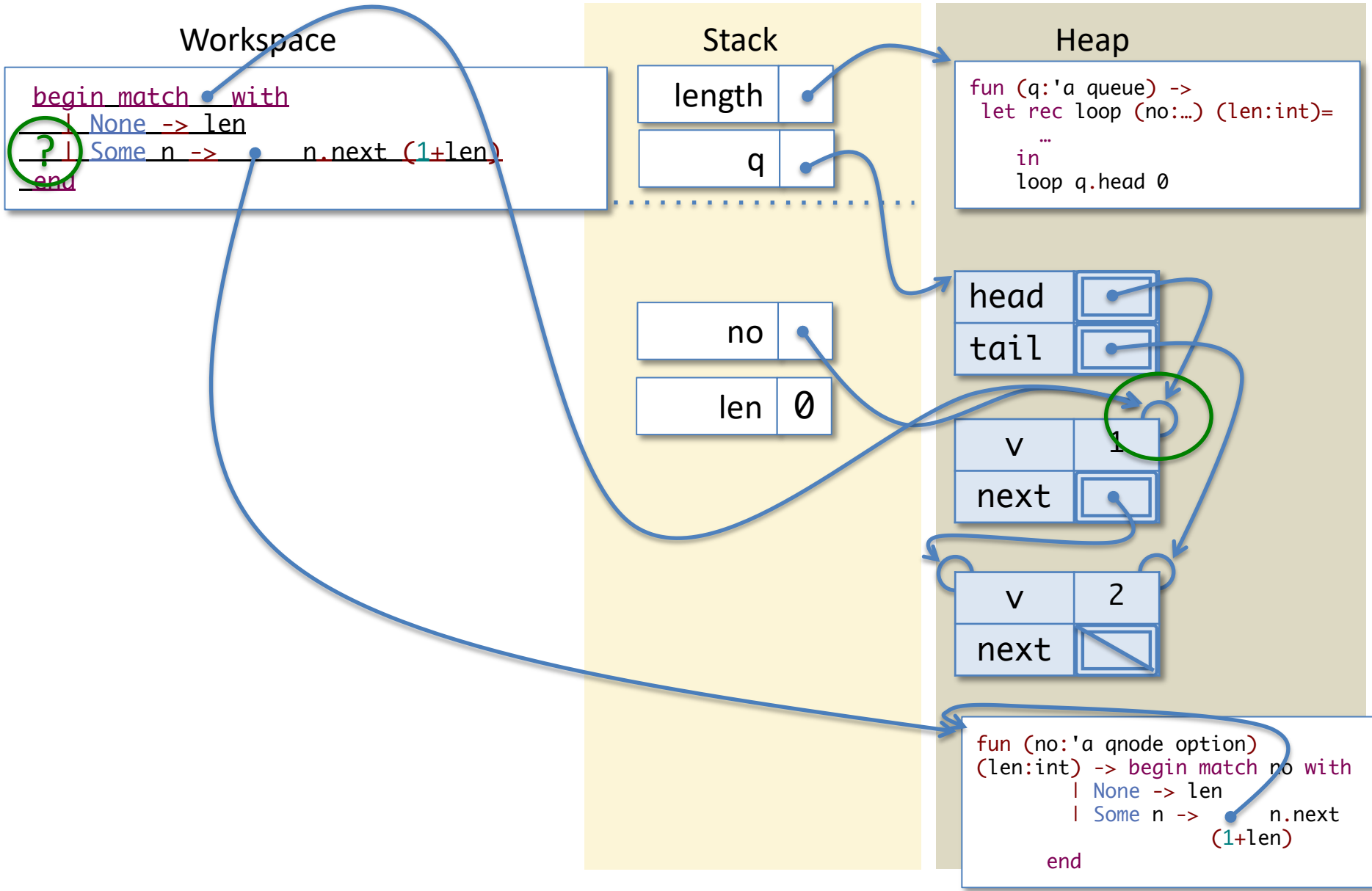
Tail Calls and Iterative length



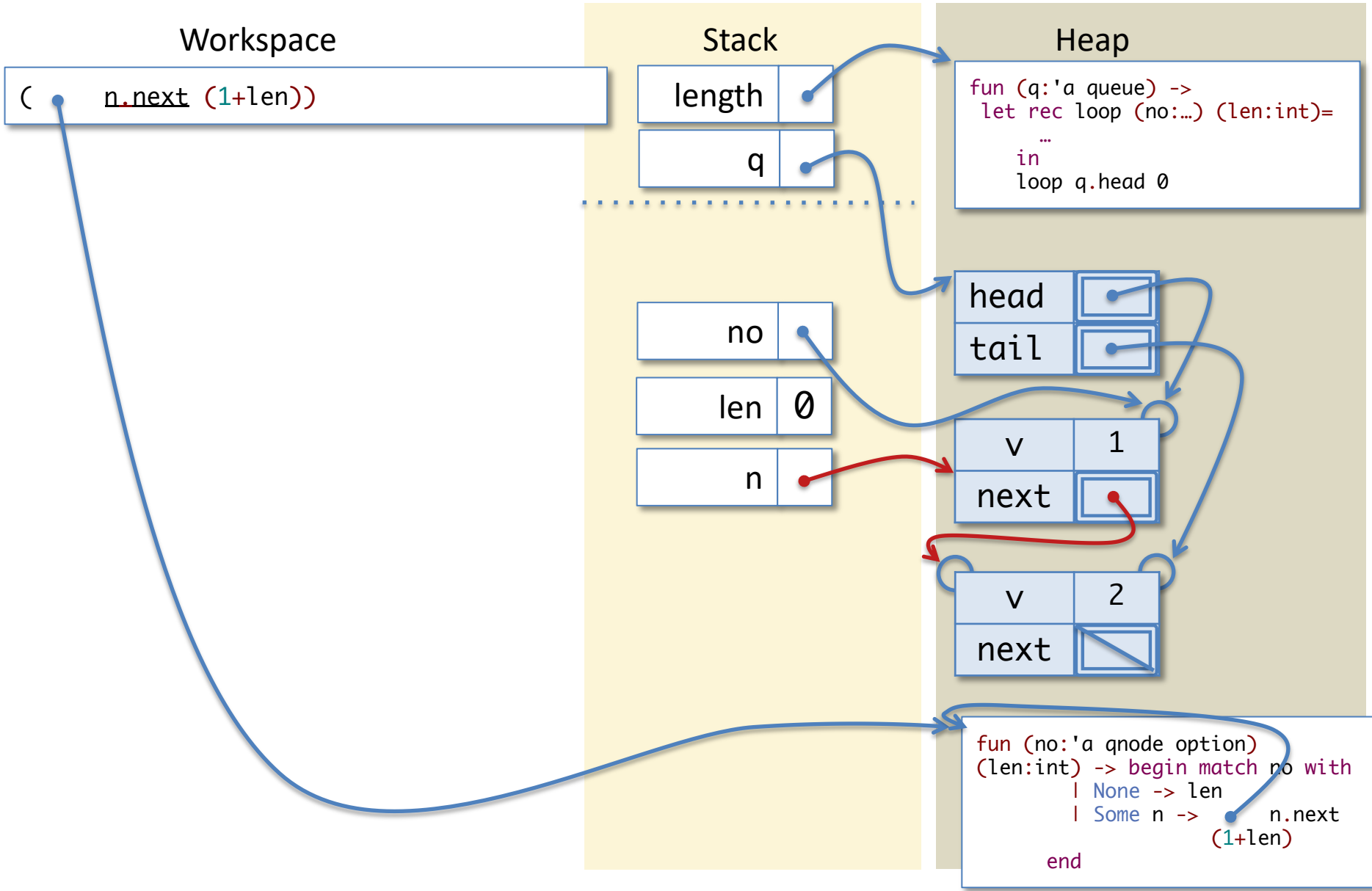
Tail Calls and Iterative length



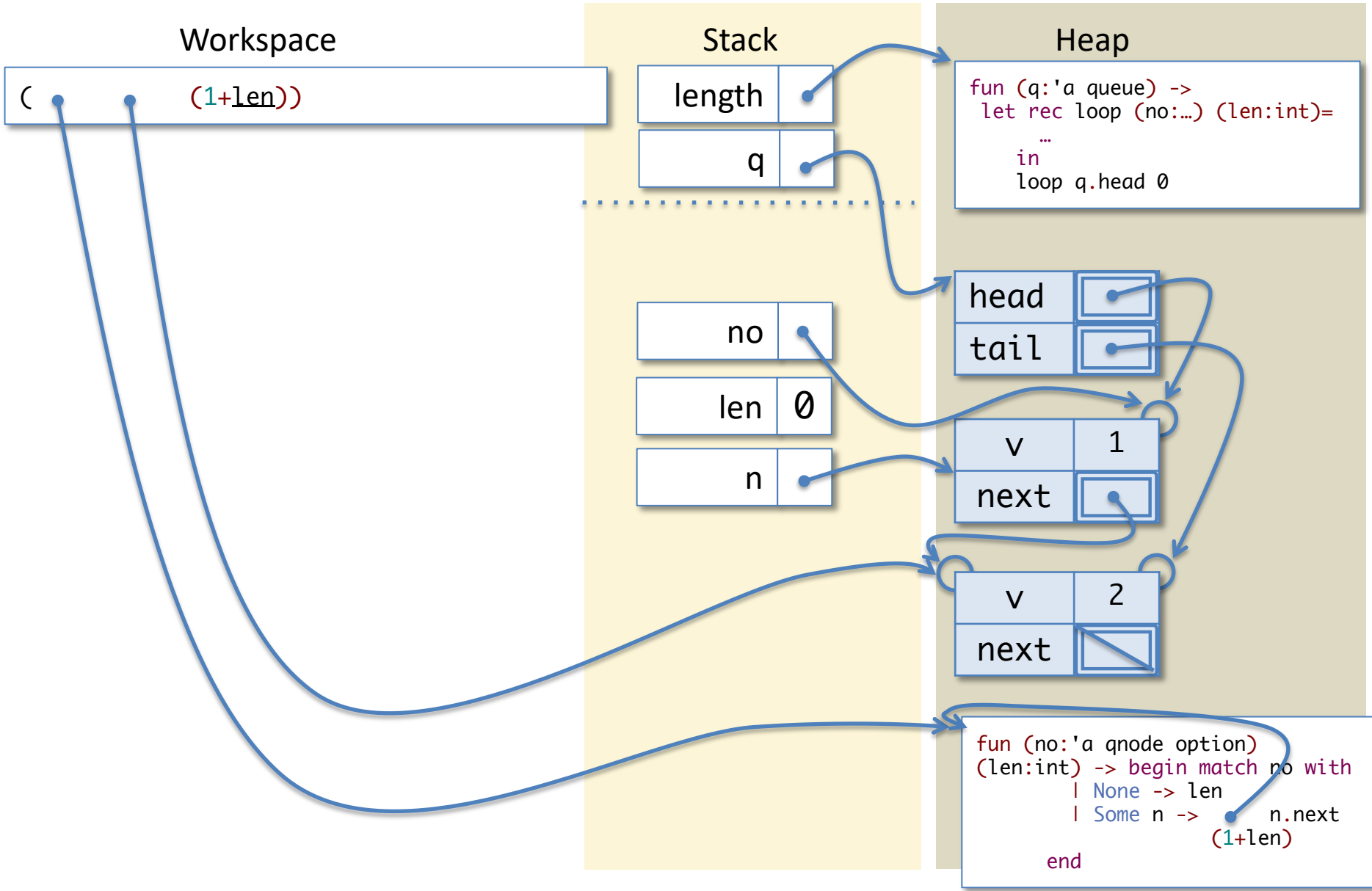
Tail Calls and Iterative length



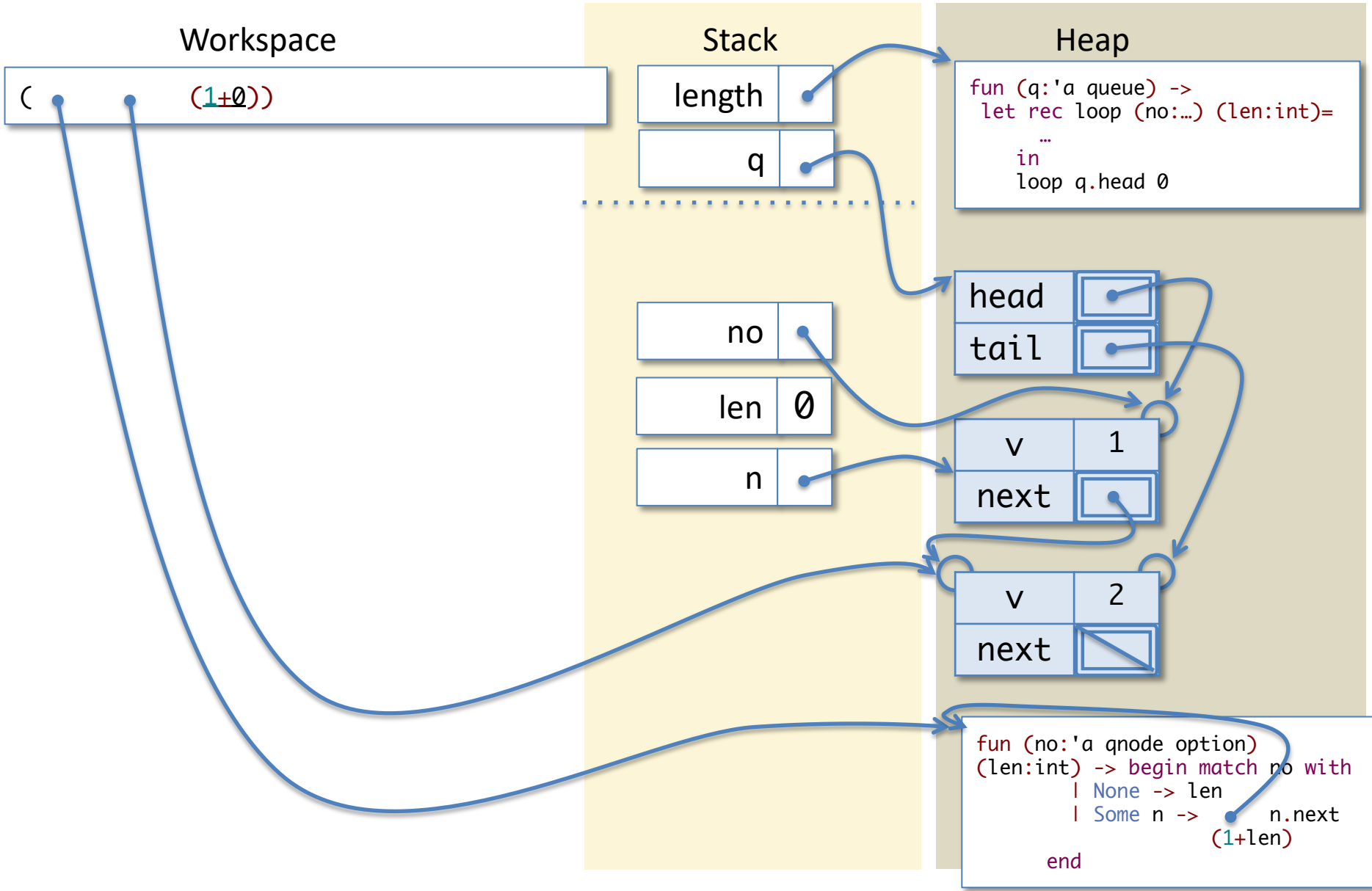
Tail Calls and Iterative length



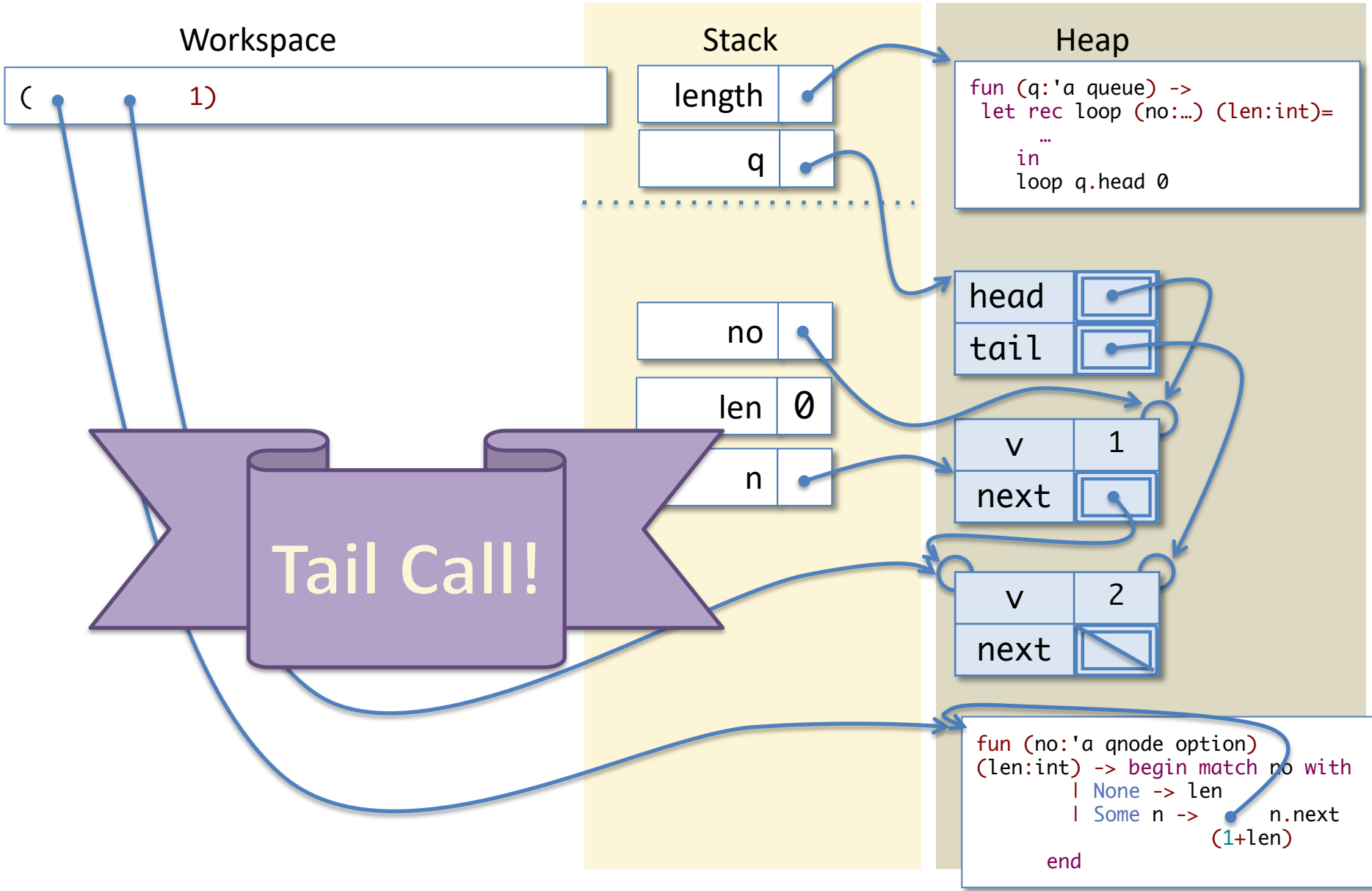
Tail Calls and Iterative length



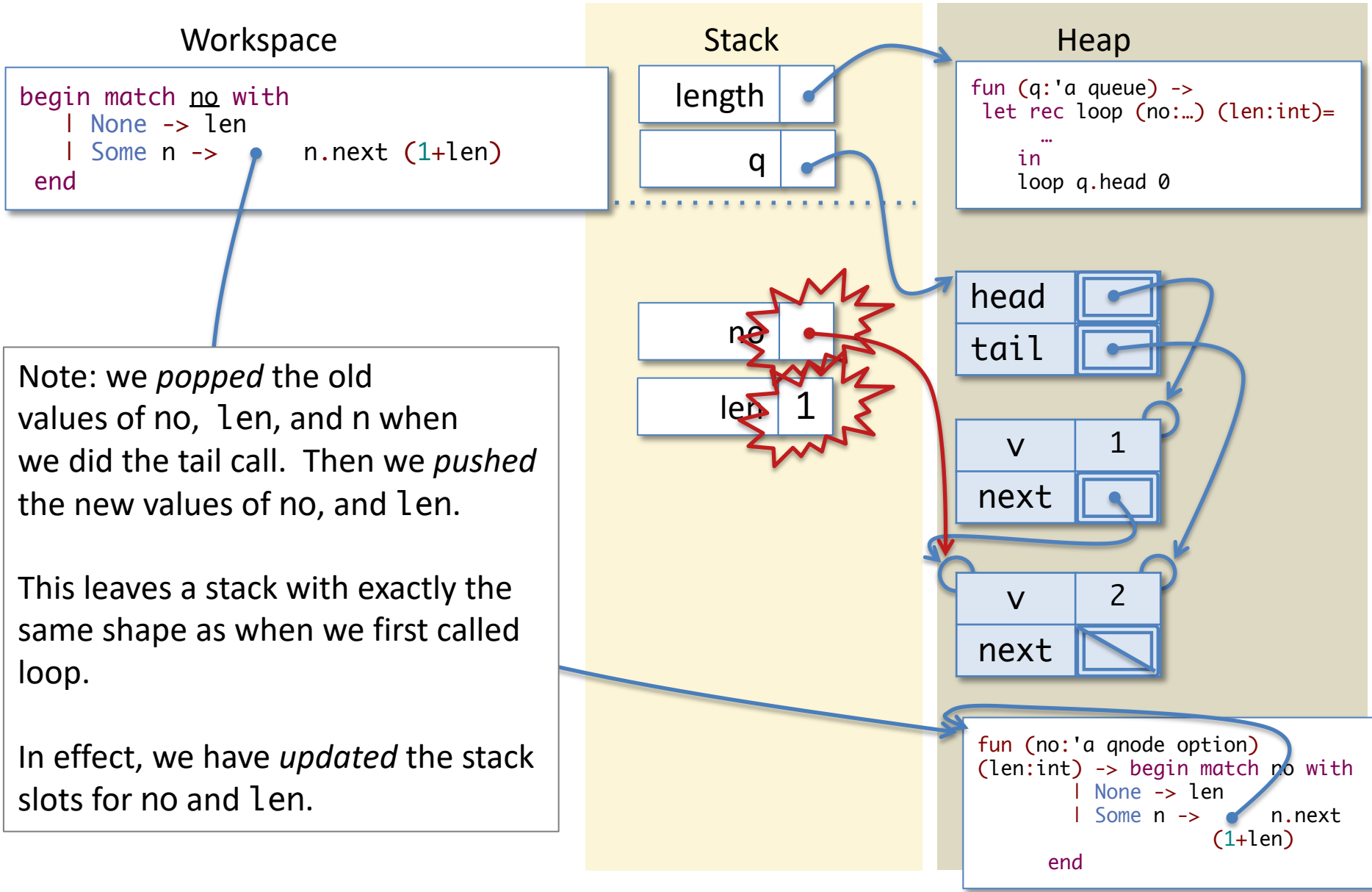
Tail Calls and Iterative length



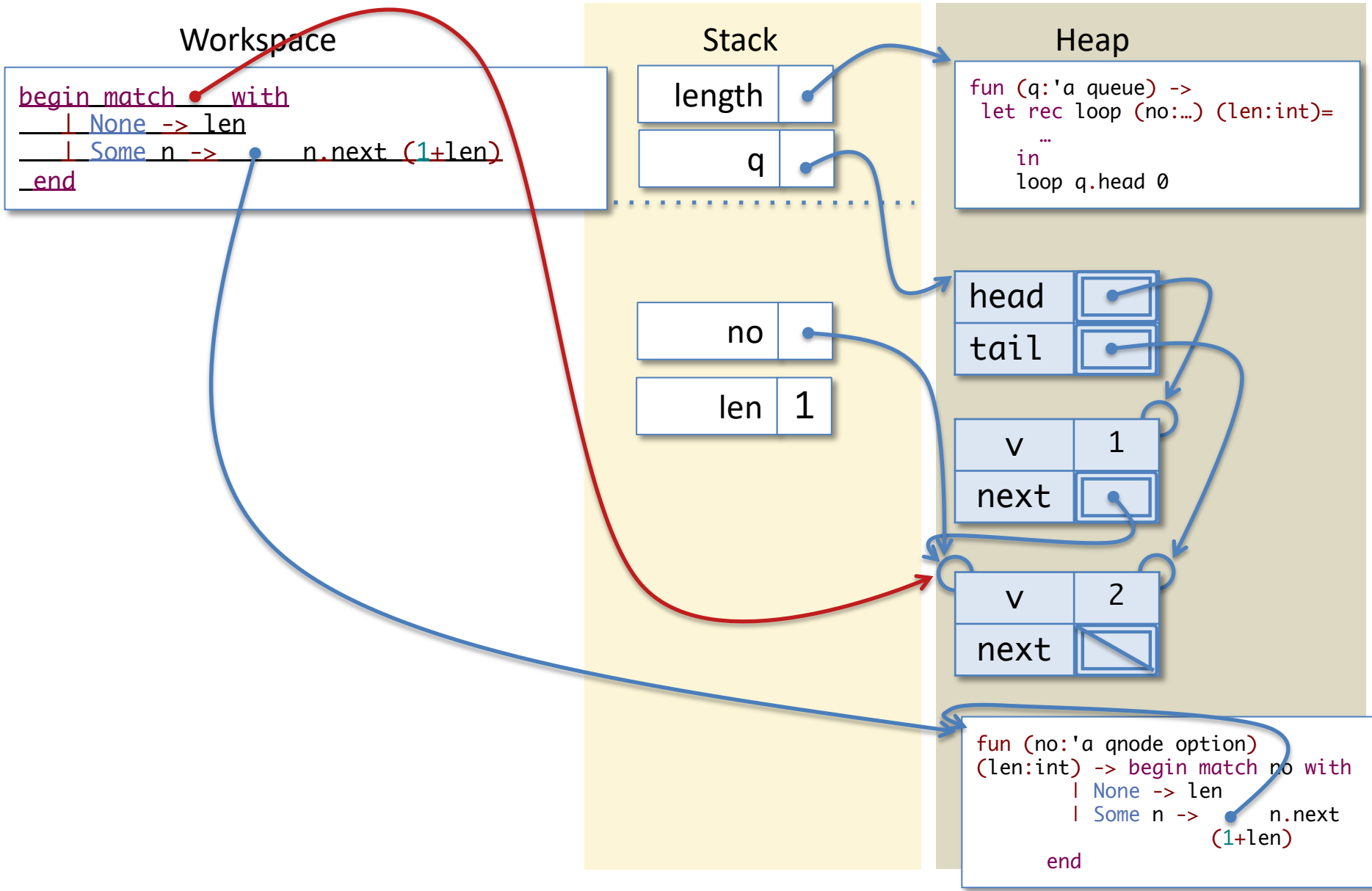
Tail Calls and Iterative length



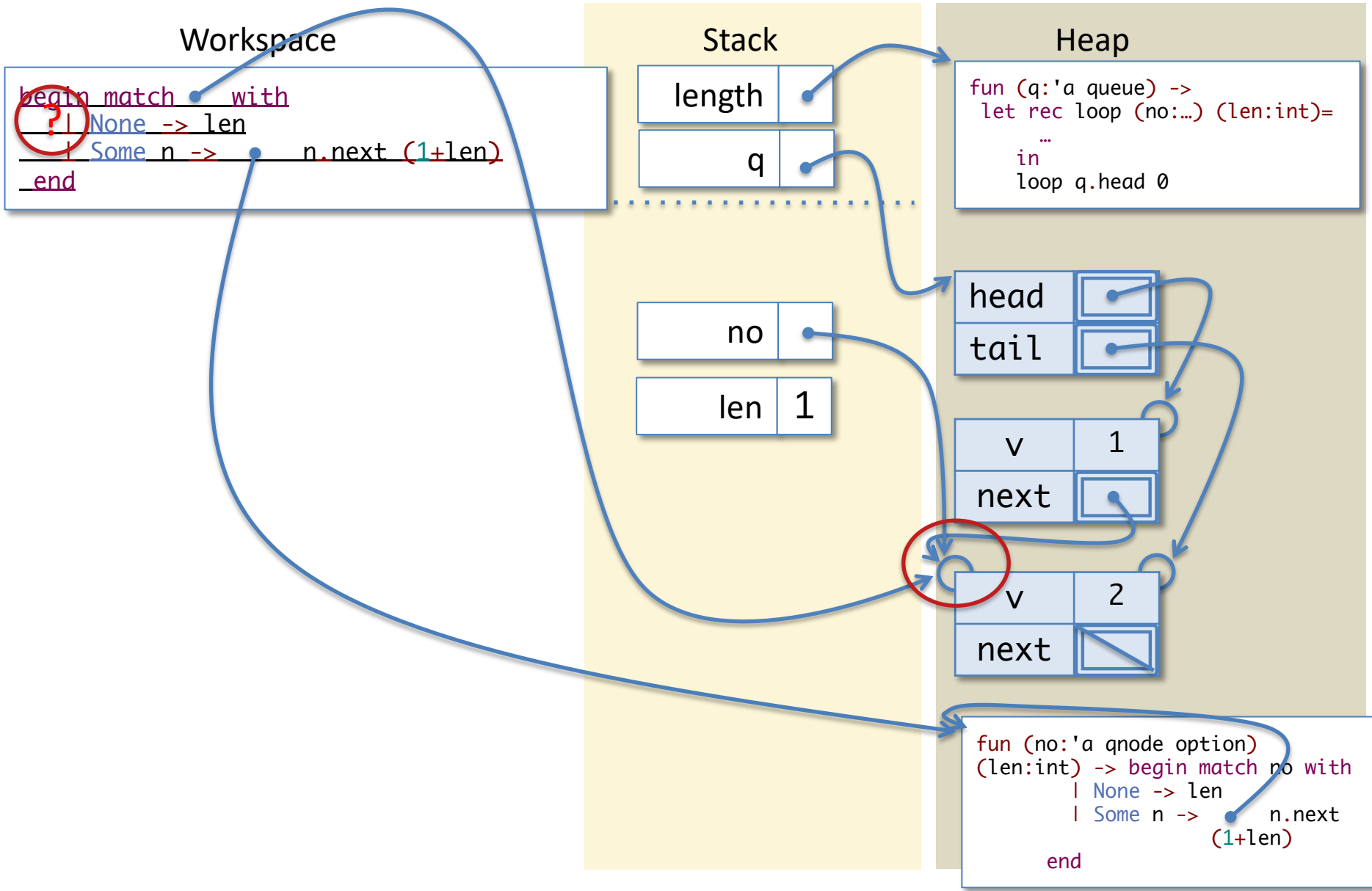
Tail Calls and Iterative length



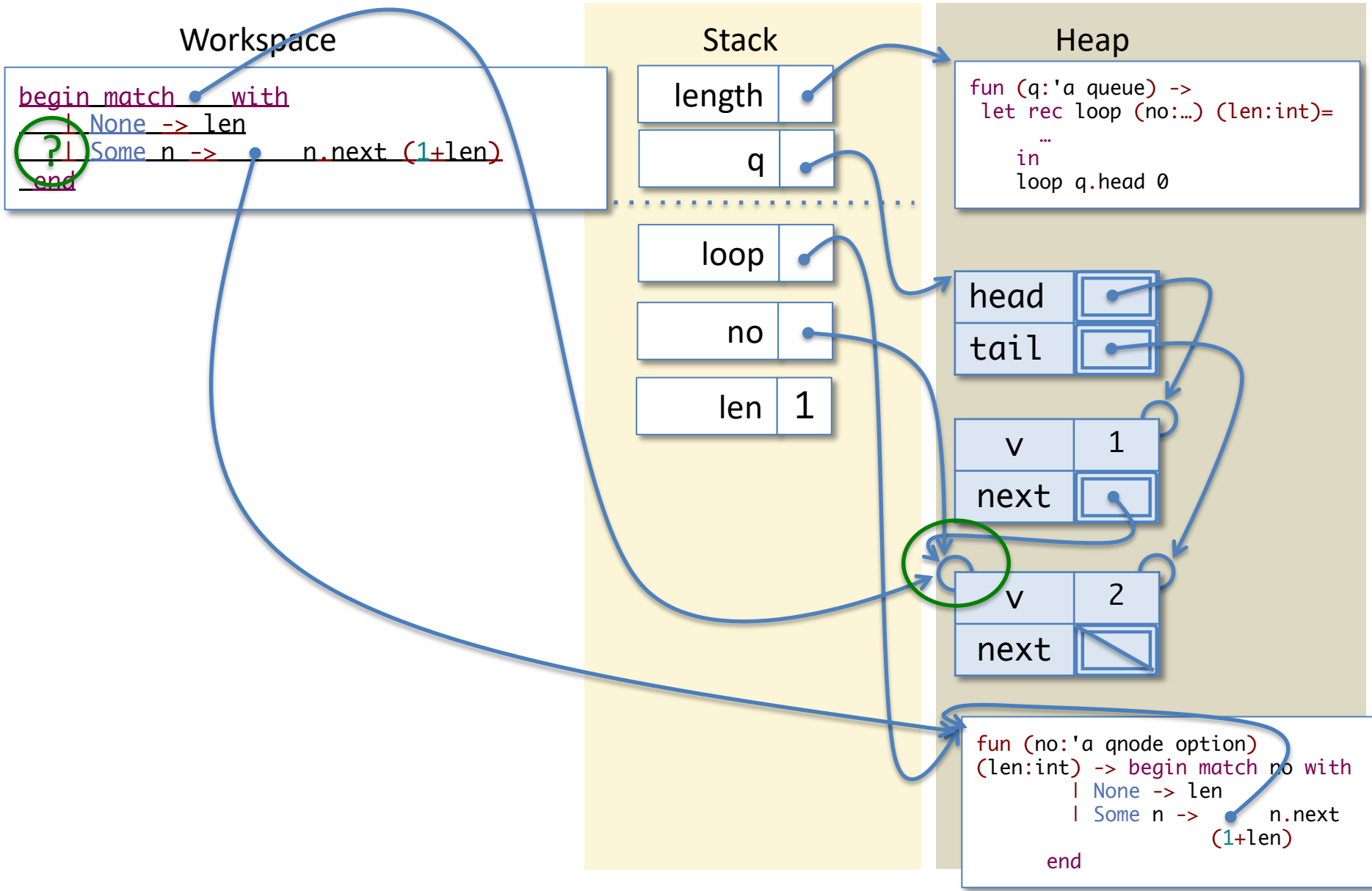
Tail Calls and Iterative length



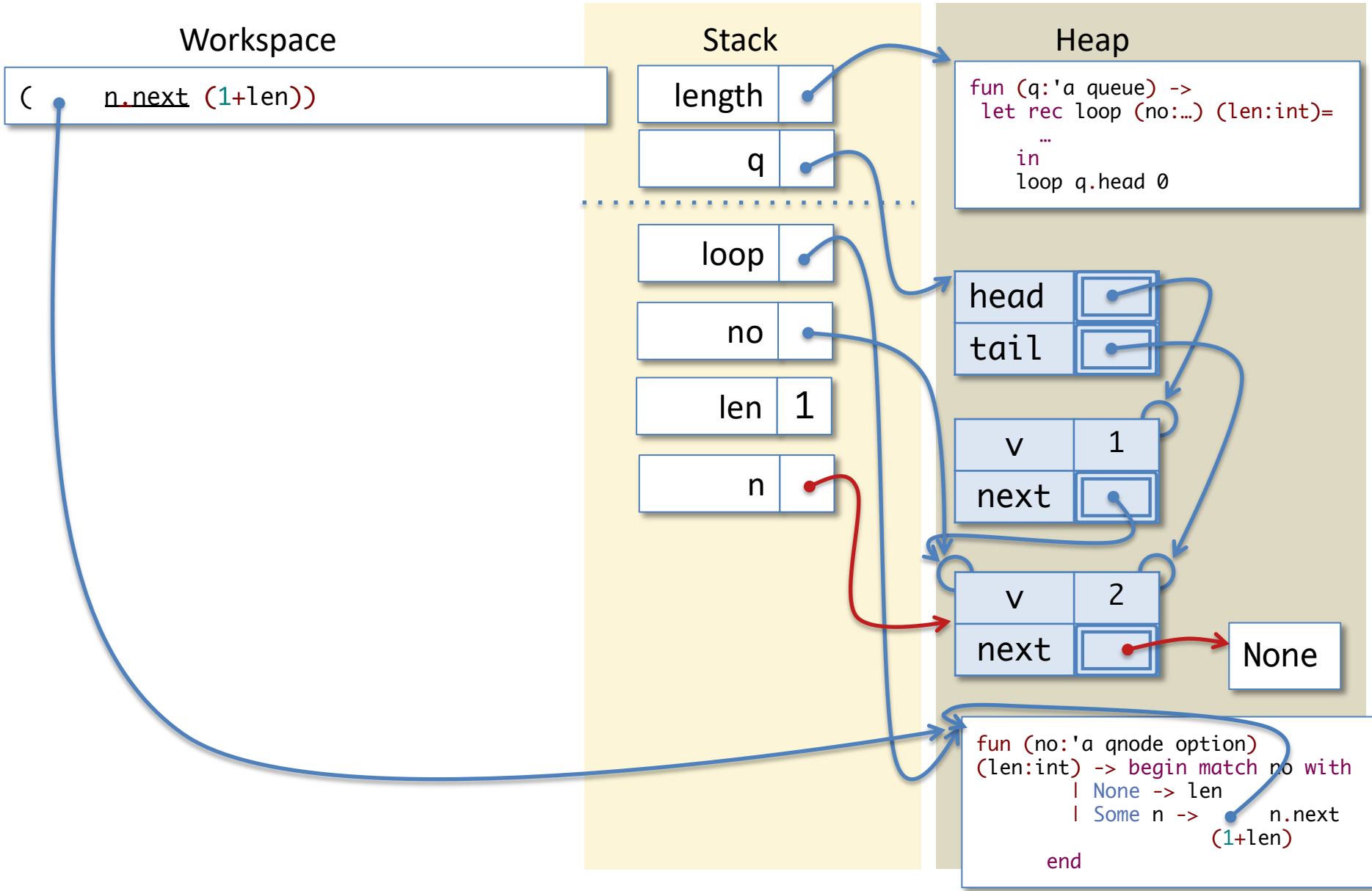
Tail Calls and Iterative length



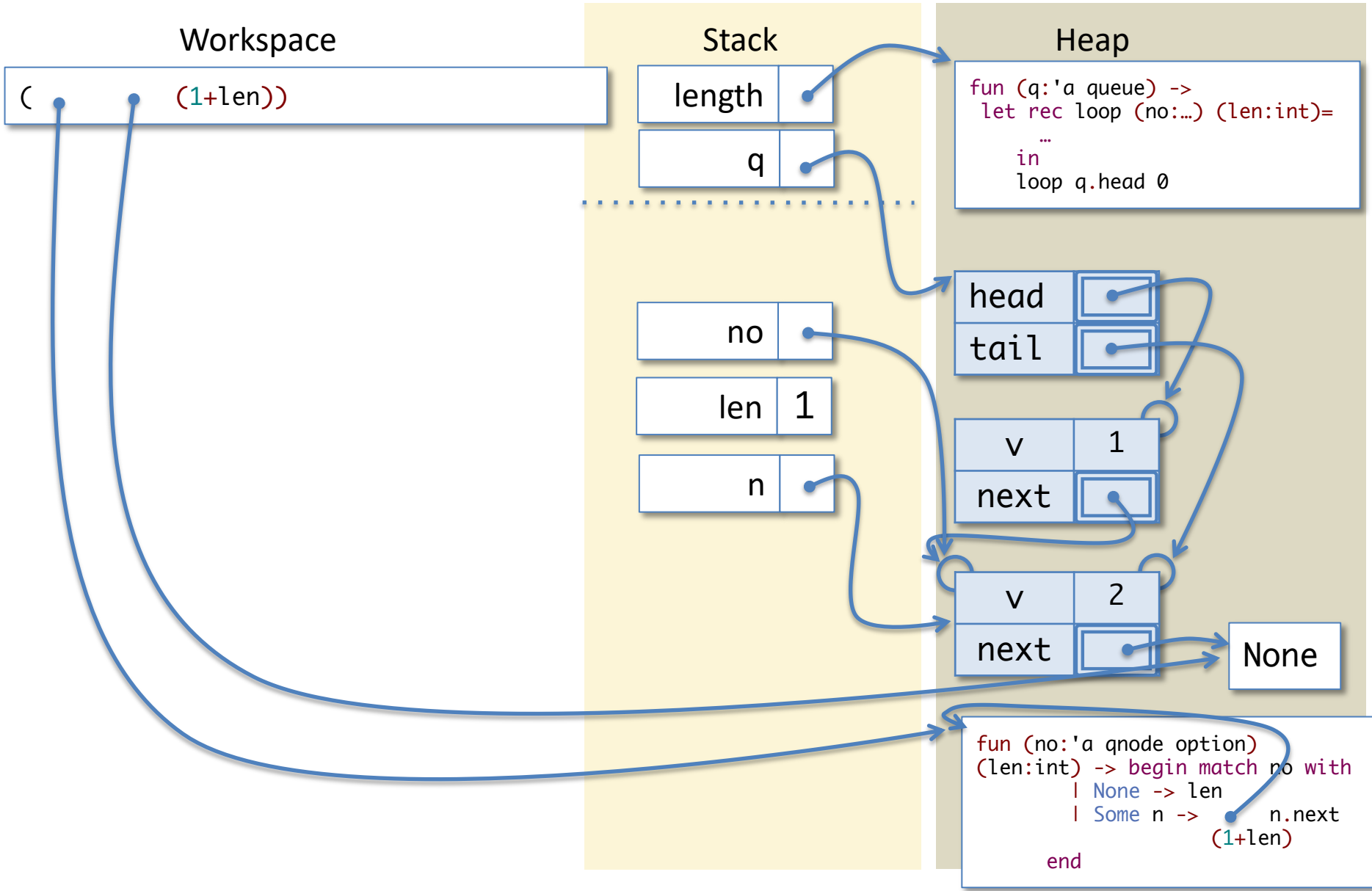
Tail Calls and Iterative length



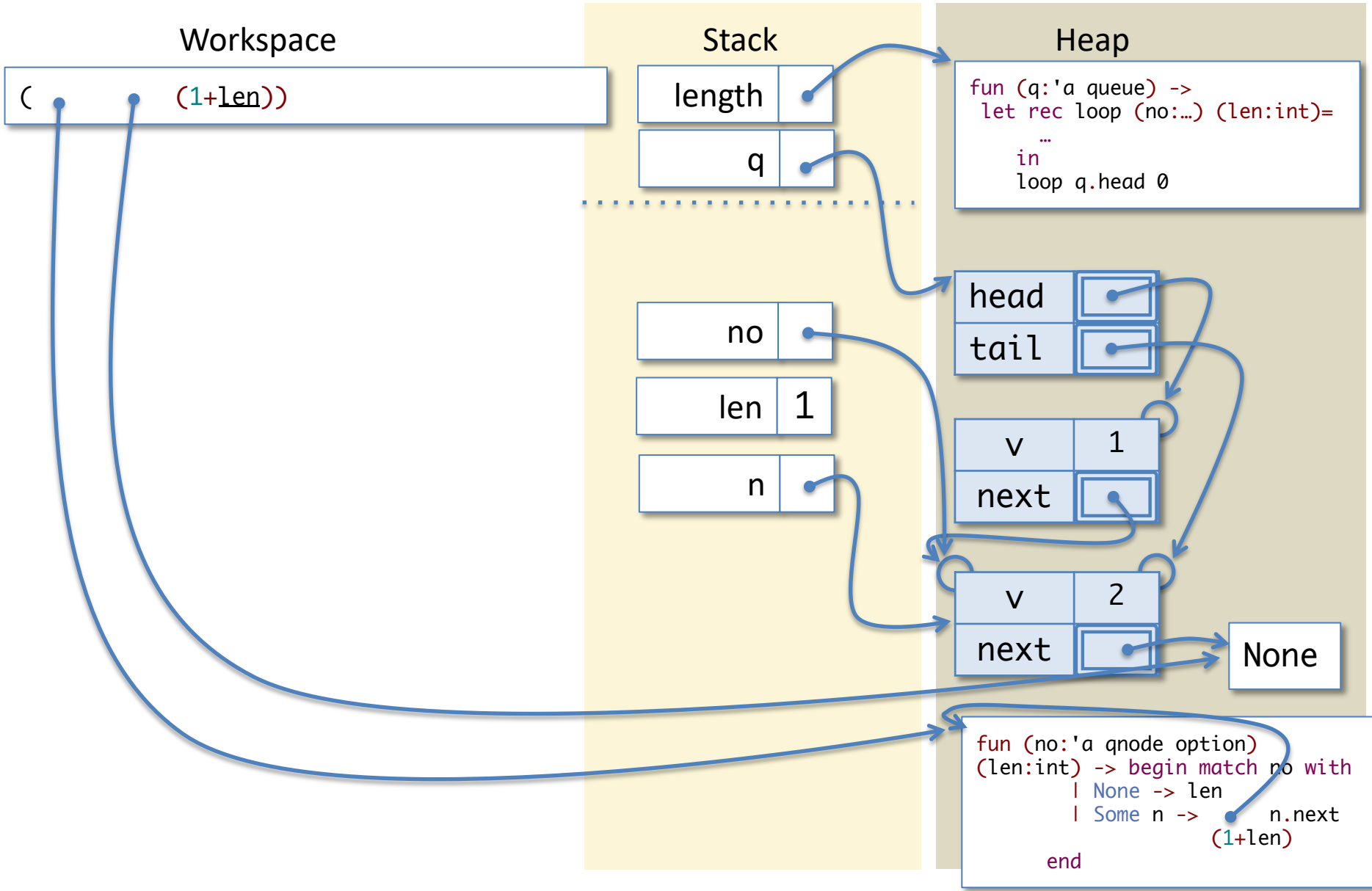
Tail Calls and Iterative length



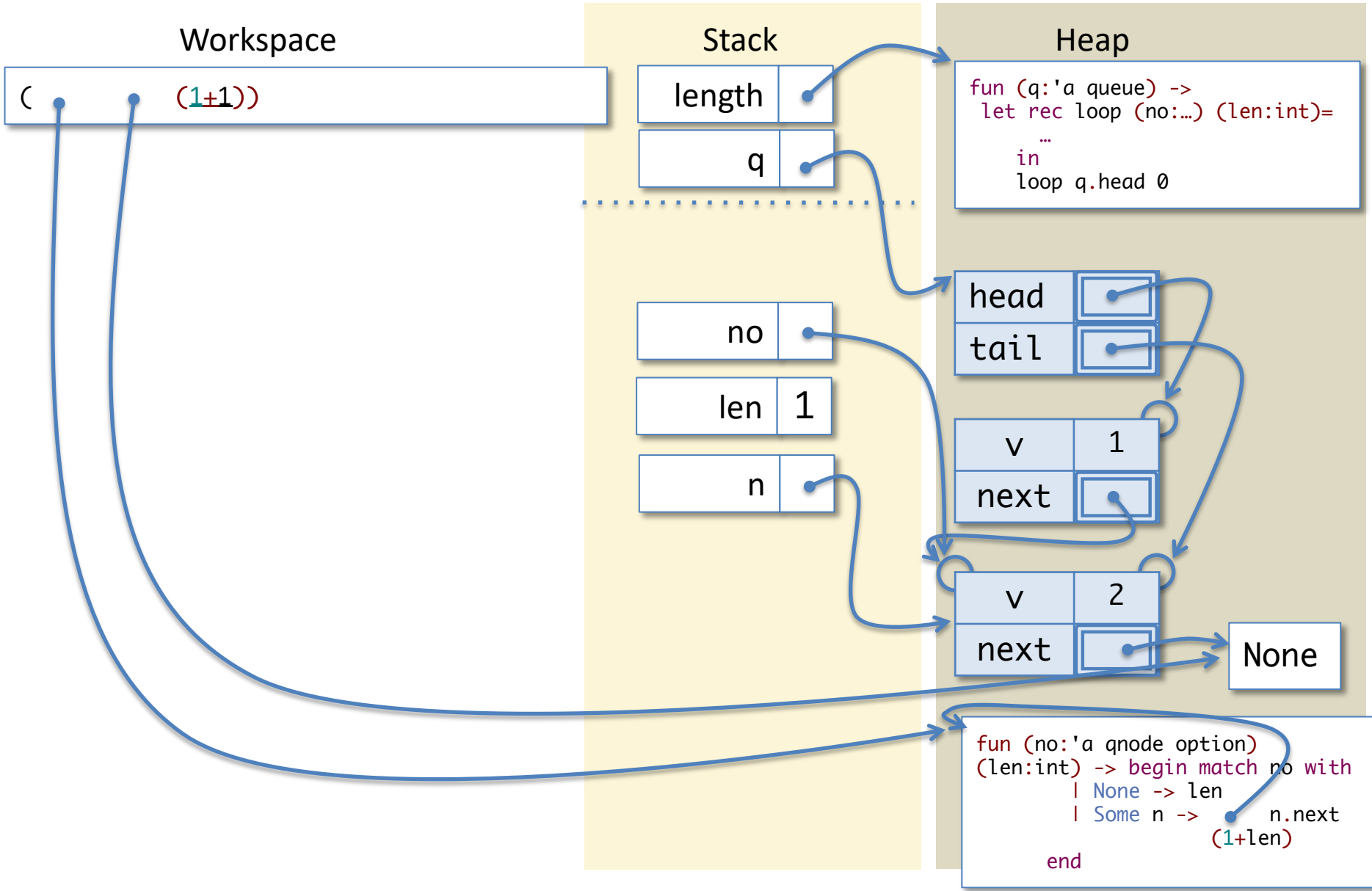
Tail Calls and Iterative length



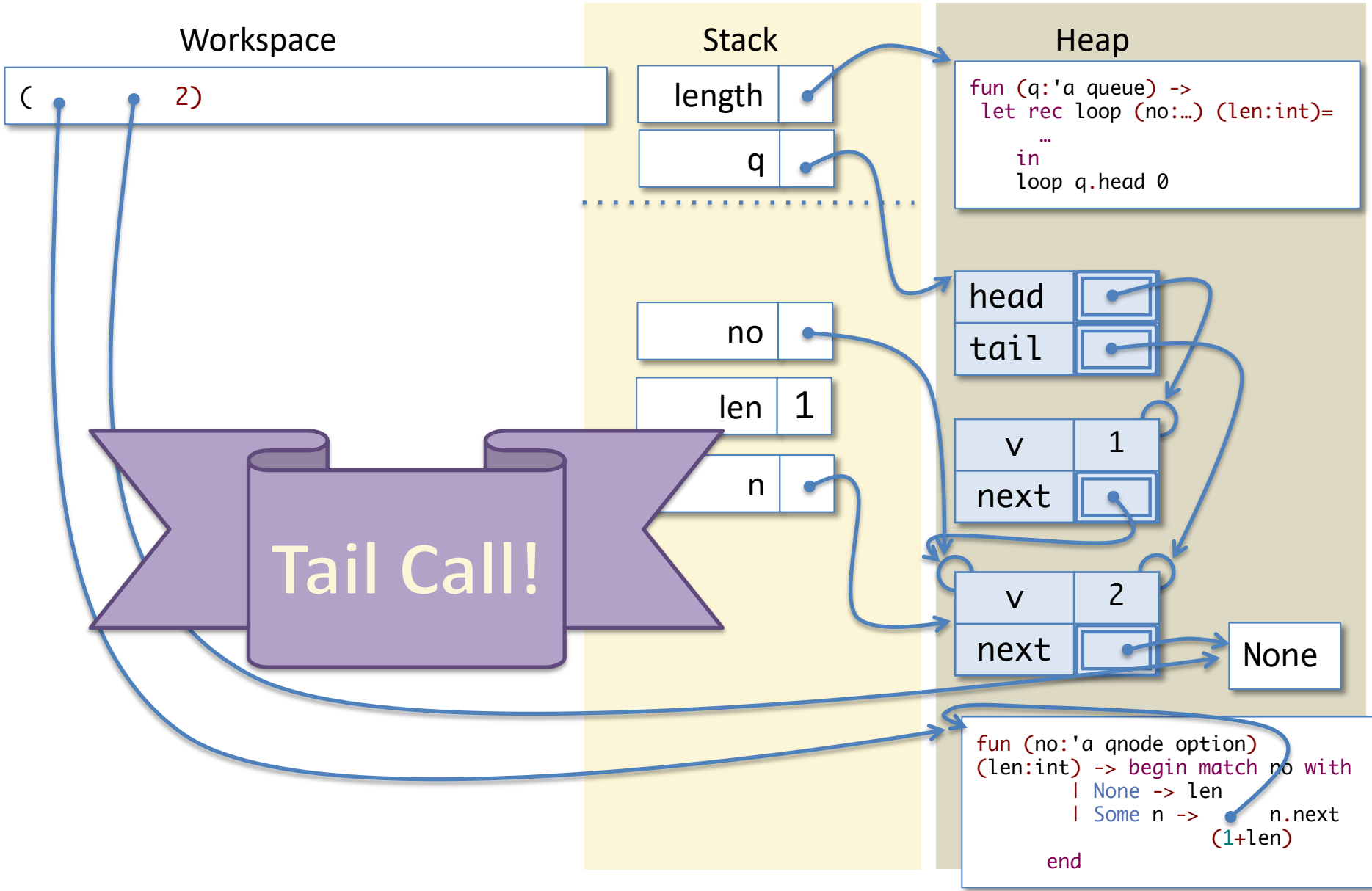
Tail Calls and Iterative length



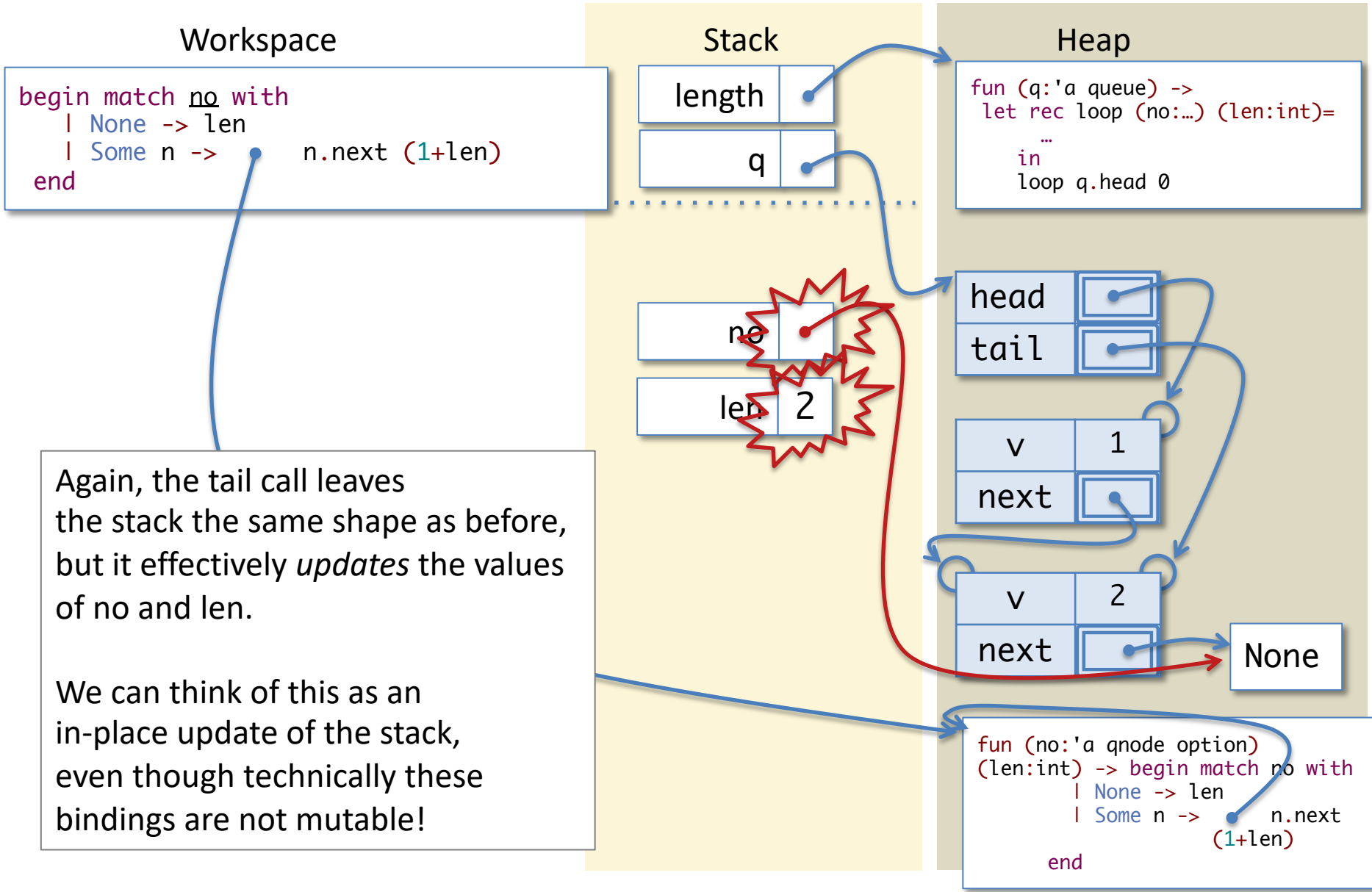
Tail Calls and Iterative length



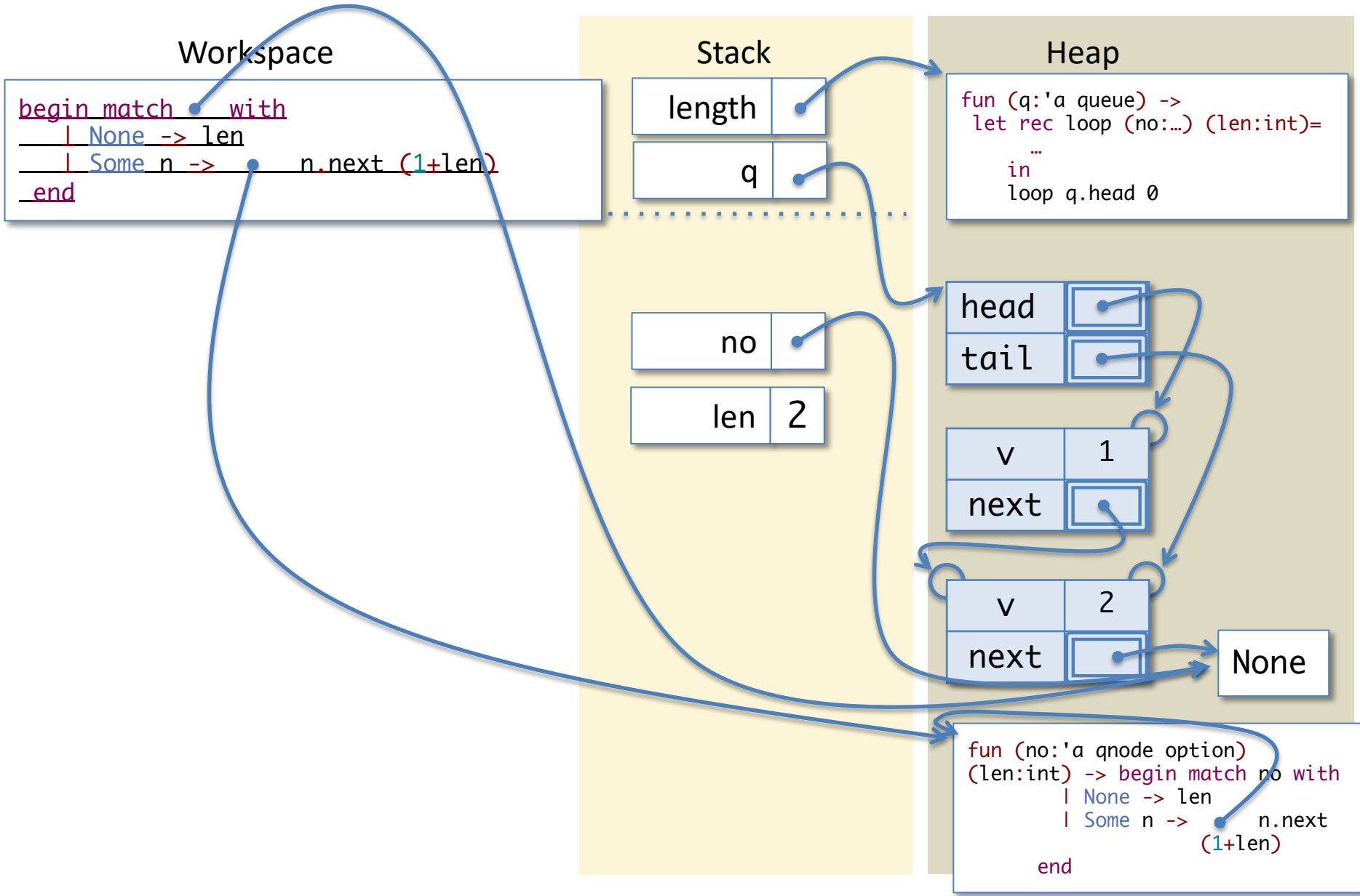
Tail Calls and Iterative length



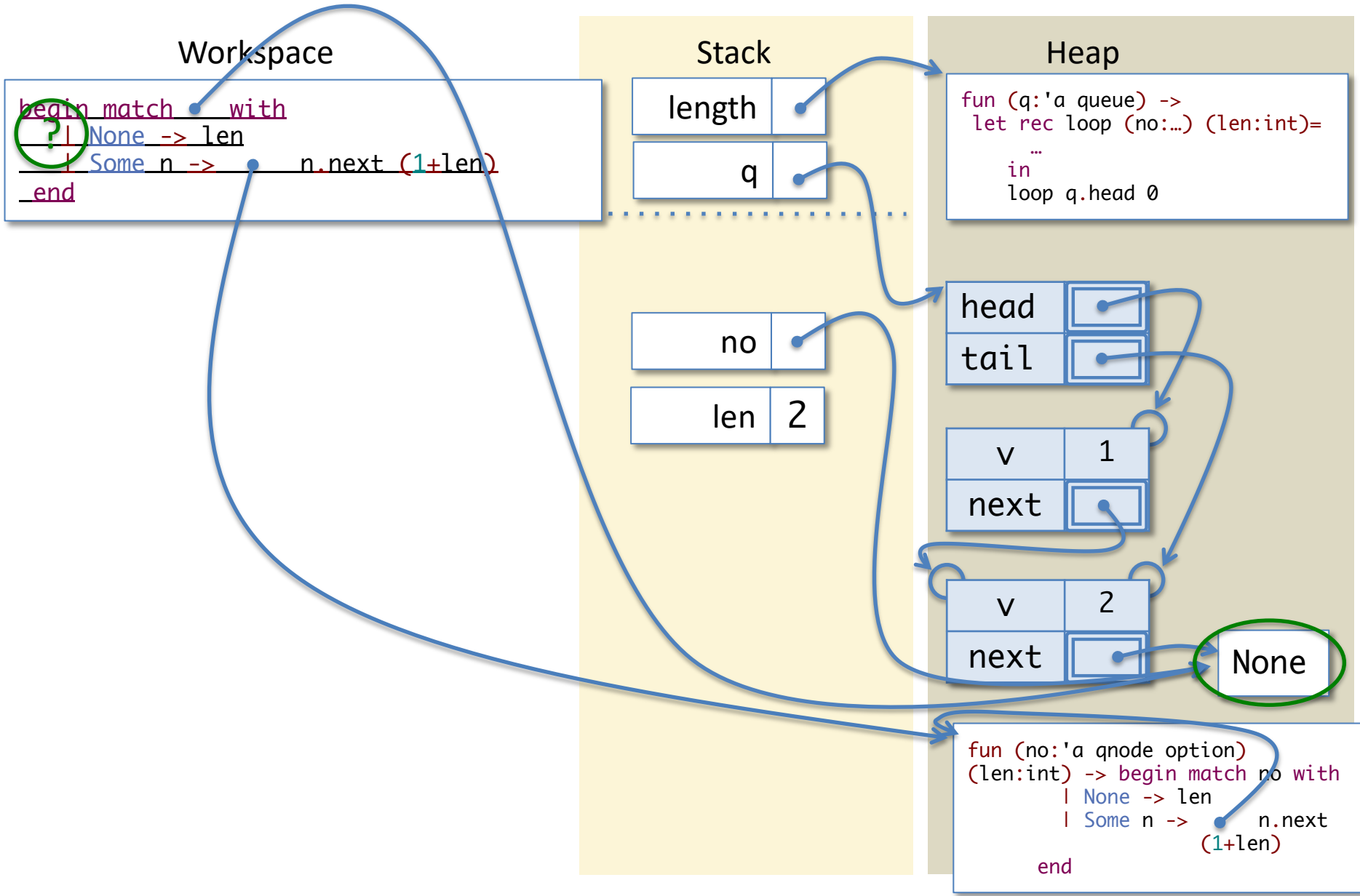
Tail Calls and Iterative length



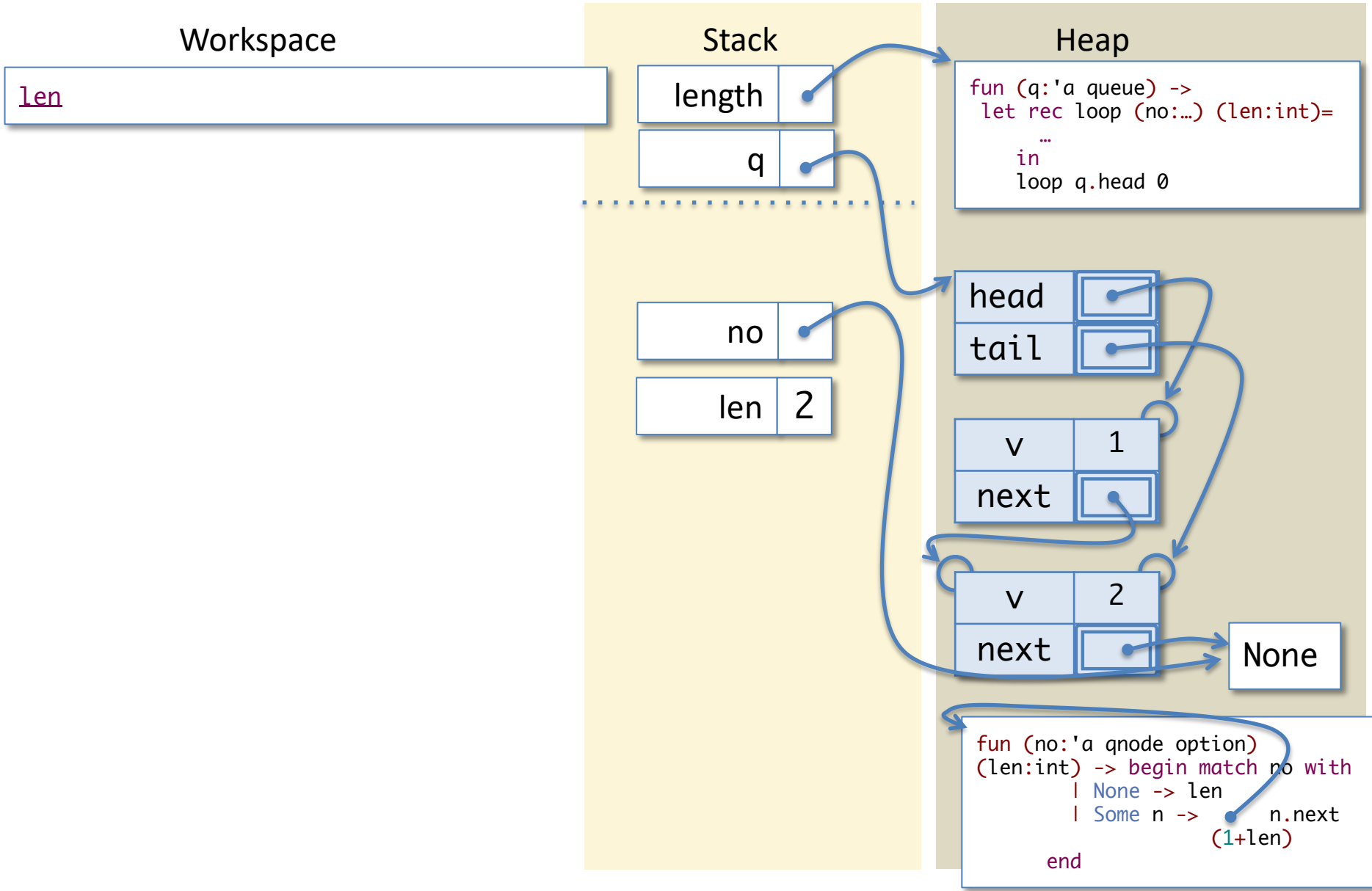
Tail Calls and Iterative length



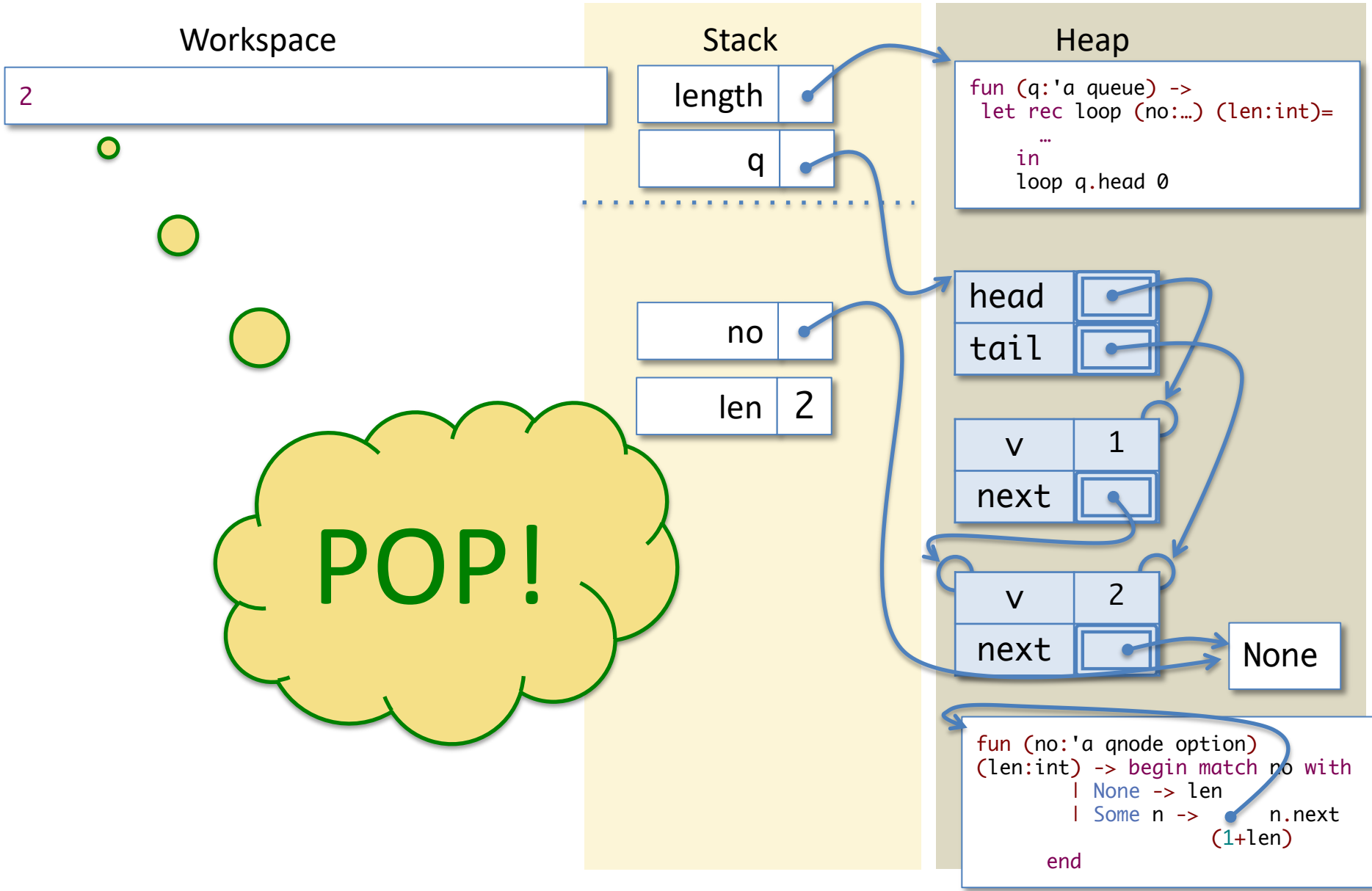
Tail Calls and Iterative length



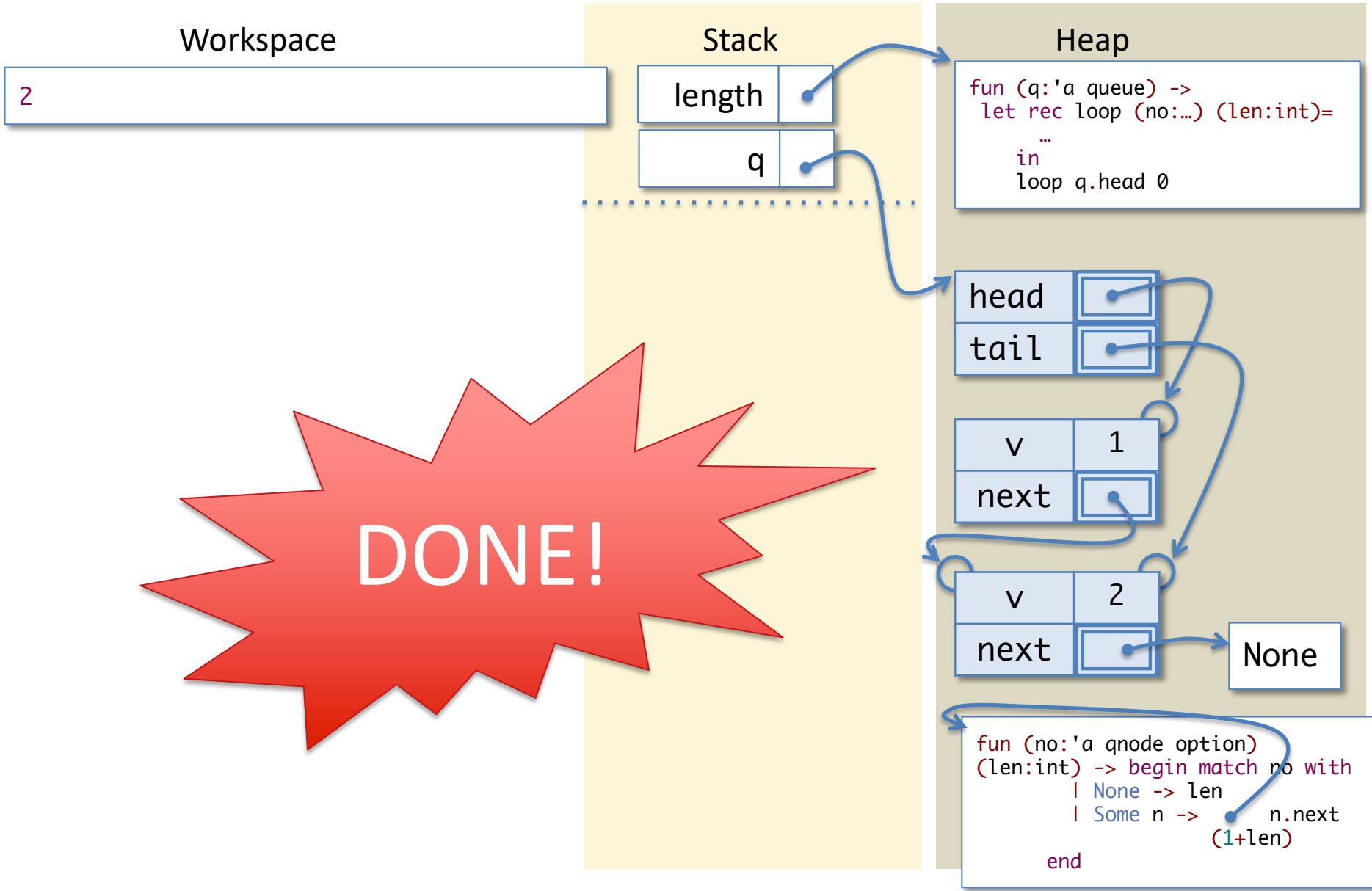
Tail Calls and Iterative length



Tail Calls and Iterative length



Tail Calls and Iterative length



Crucial Observations

- Tail call optimization lets the stack take only a *fixed amount of space*.

- The recursive call to `loop` effectively updates the stack bindings in place.
  - We can think of these bindings as the *state* being modified by each iteration of the loop.

- These two properties are the essence of iteration.
  - They are the difference between general recursion and iteration

16: What happens when you run this function on a (valid) queue containing 2 elements?

0

The value 2 is returned

0%

The value 0 is returned

0%

StackOverflow

0%

Your program hangs

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

What happens when you run this function on a (valid) queue containing 2 elements?

```
let f (q:'a queue) : int =
    let rec loop (qn:'a qnode option) : int =
      begin match qn with
        | None -> 0
        | Some n -> 1 + loop qn
      end
    in loop q.head
```

1. The value 2 is returned
2. The value 0 is returned
3. StackOverflow
4. Your program hangs

ANSWER: 3

16: What happens when you run this function on a (valid) queue containing 2 elements?

0

The value 2 is returned

0%

The value 0 is returned

0%

StackOverflow

0%

Your program hangs

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

What happens when you run this function on a (valid) queue containing 2 elements?

```
let f (q:'a queue) : int =
    let rec loop (qn:'a qnode option) (len:int) : int =
      begin match qn with
        | None -> len
        | Some n -> loop qn (len + 1)
      end
    in loop q.head 0
```

1. The value 2 is returned
2. The value 0 is returned
3. StackOverflow
4. Your program hangs

ANSWER: 4

Infinite Loops

```
(* Accidentally go into an infinite loop… *)
let accidental_infinite_loop (q:'a queue) : int =
    let rec loop (qn:'a qnode option) (len:int) : int =
      begin match qn with
        | None -> len
        | Some n -> loop qn (len + 1)
      end
    in loop q.head 0
```

- This program will go into an infinite loop.
- Unlike a non-tail-recursive program, which uses some space on each recursive call, there is no resource being exhausted, so the program will "silently diverge" and simply never produce an answer…

More iteration examples

to_list

print

chop

to_list (using iteration)

```
(* Retrieve the list of values stored in the queue,
   ordered from head to tail. *)
let to_list (q: 'a queue) : 'a list =
  let rec loop (no: 'a qnode option) (l:'a list) : 'a list =
      begin match no with
      | None -> List.rev l
      | Some n -> loop n.next (n.v::l)
      end
  in loop q.head []
```

- Here, the state maintained across each iteration of the loop is the queue "index pointer" no and the (reversed) list of elements traversed.
- The "exit case" post processes the list by reversing it.

print (using iteration)

```
let print (q:'a queue) (string_of_element:'a -> string) : unit =
  let rec loop (no: 'a qnode option) : unit =
    begin match no with
      | None -> ()
      | Some n -> print_endline (string_of_element n.v);
                  loop n.next
    end
  in
    print_endline "--- queue contents ---";
    loop q.head;
    print_endline "--- end of queue -----"
```

- Here, the only state needed is the queue “index pointer”.

chop (remembering "previous")

```
(* Removes all elements from q starting with the first
   occurrence of elt.  *)
let chop (elt:'a) (q:'a queue) : unit =
  let rec loop (curr: 'a qnode option)
               (prev: 'a qnode option) : unit =
    begin match curr with
      | None -> ()
      | Some n ->
        if n.v == elt
        then begin
          q.tail <- prev;    (* NOTE: update the tail with prev *)
          begin match prev with
          | None   -> q.head <- None (* We deleted everything! *)
          | Some n -> n.next <- None (* prev is the new tail *)
          end
        end else loop n.next curr
    end
  in loop q.head None
```

- Here, the state needed is the current "pointer" and (optionally) the previous node.
  - We need to keep track of the previous node to update the tail

Singly-linked Queue Processing

- General structure (schematically) :

```
(* Process a singly-linked queue. *)
let queue_operation (q: 'a queue) : 'b =
  let rec loop (current: 'a qnode option) (s:'a state) : 'b =
      begin match current with
      | None -> …  (* iteration complete, produce result *)

      | Some n ->  … (* do something with n,
                        create new loop state *)
                   loop n.next new_s

      end
  in loop q.head init
```

- What is useful to put in the state?
  - Accumulated information about the queue (e.g., length so far)
  - Link to previous node (so that it could be updated, for example)

General Guidelines

- Processing *must* maintain the queue invariants
- Update the head and tail references (if necessary)
- If changing the link structure:
  - Sometimes useful to keep reference to the previous node (allows removal of the current node)
- Drawing pictures of the queue heap structure is helpful

- If iterating over the whole queue (e.g. to find an element)
  - It is usually *not useful* to use helpers like “is_empty” or “contains” because you will have to account for those cases during the traversal anyway!

Tail Recursion & Iteration

- A function call is in *tail position* if, when the call is evaluated in the workspace, there is no more work to be done before returning.
  - i.e., the result of the workspace is the result of the call

```
f x                    (* ← this is in tail position *)
```

```
if … then
    f x              (* ← this f is in tail position *)
else
   (f x) + 1         (* ← this f is not in tail position *)
```

```
begin match … with
 | … -> f x           (* ← this f is in tail position *)
 | … -> g (f x)       (* ← this f is not in tail position,*)
end                   (*    but g is in tail position *)
```

```
(g x) || (f x)  (* ← this f is in tail position, g isn't *)
```

```
cmd ; (f x)         (* ← this f is in tail position *)
```

(Bonus Material)
ASM: Sharing and Space

another example of
pattern matching and recursion

Sharing / Memory Usage Example

Even for "pure" (non-mutating / imperative) programs, the Abstract Stack Machine provides a useful, refined model of computation that can help when reasoning about memory usage…

```
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 rec 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

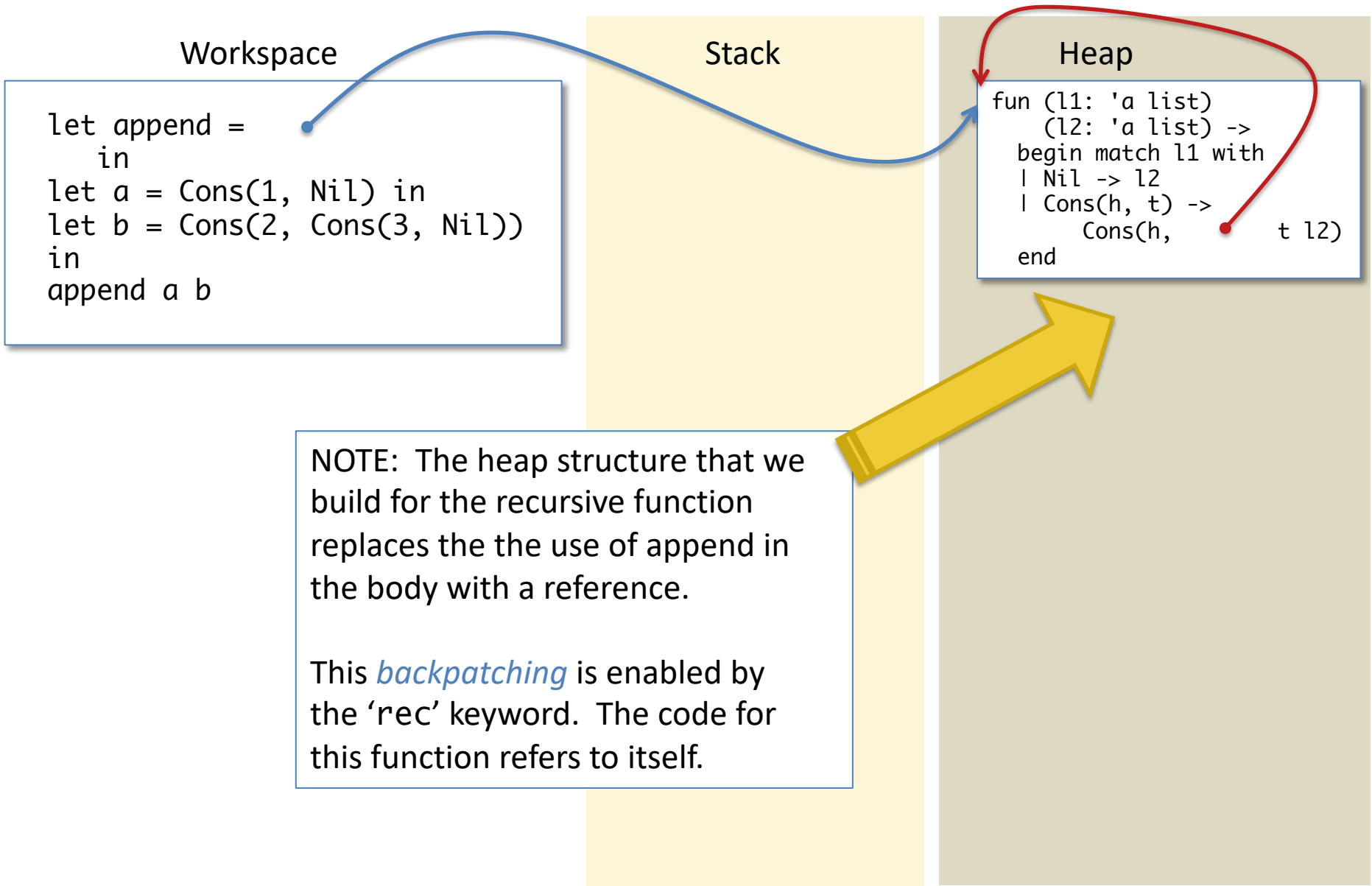
Workspace

```
let rec 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
```

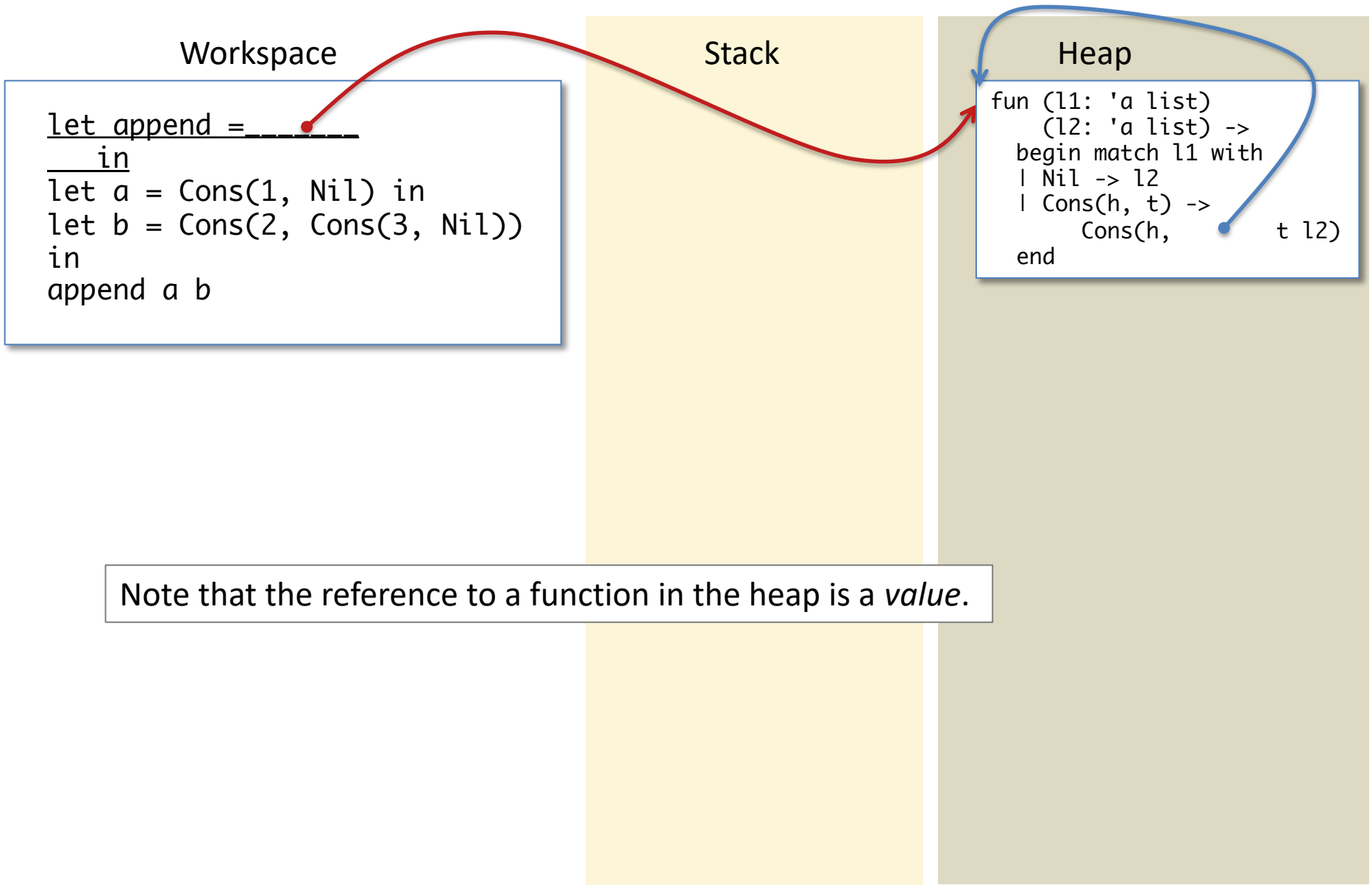
Stack

Heap

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,        t l2)
  end
```

Allocate a Nil cell

Workspace

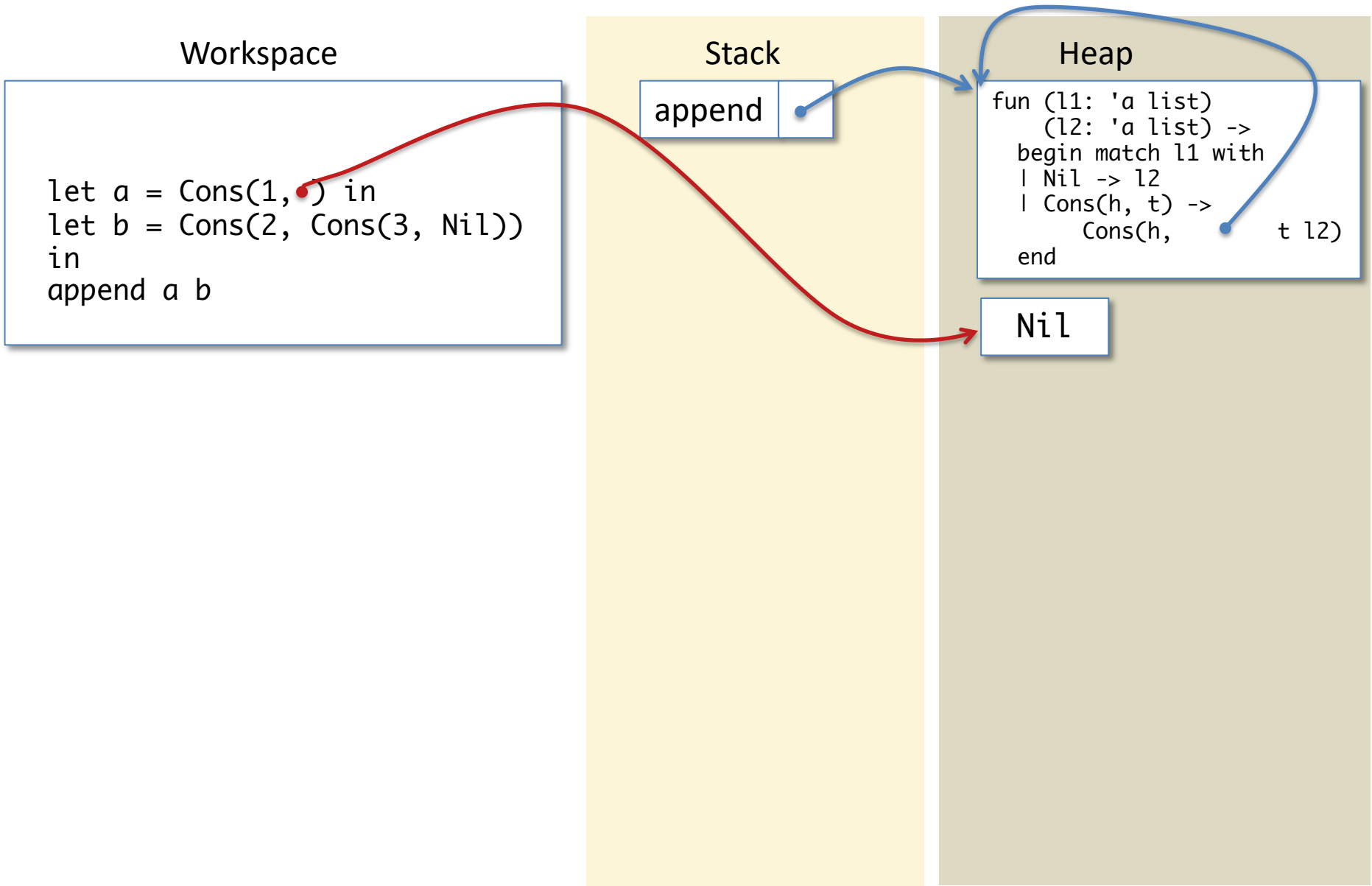
```
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

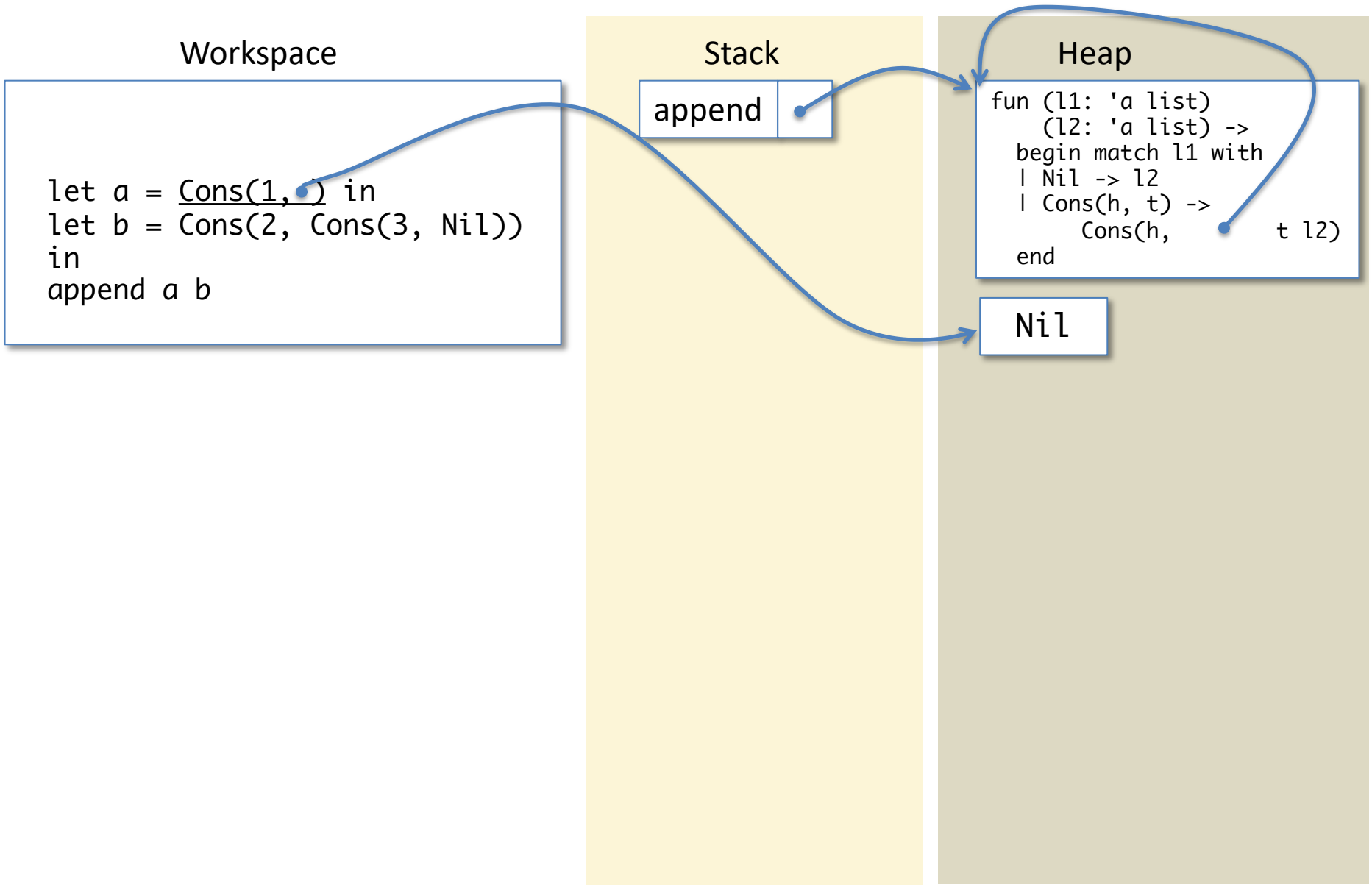
Heap

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

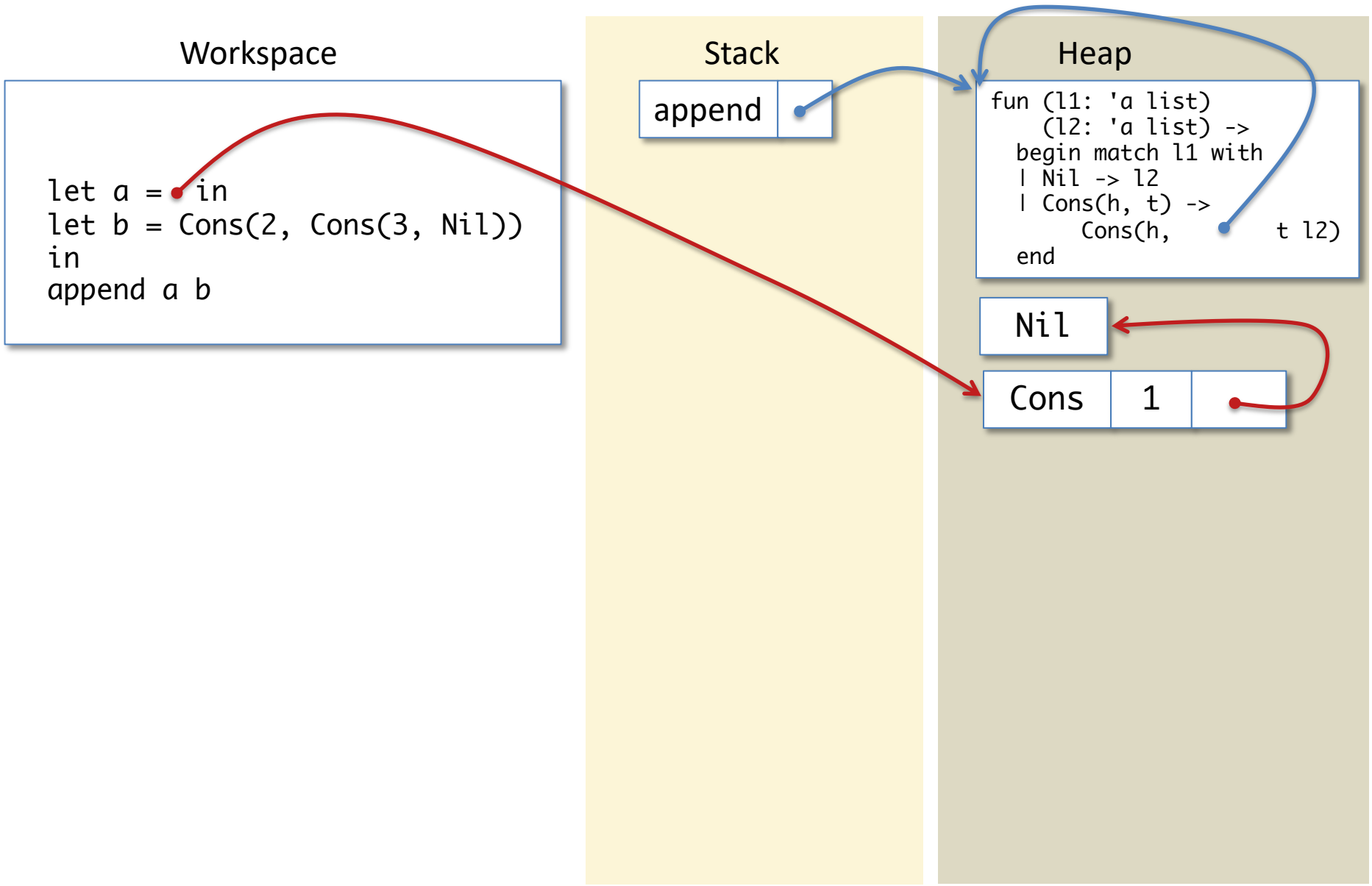
Allocate a Nil cell



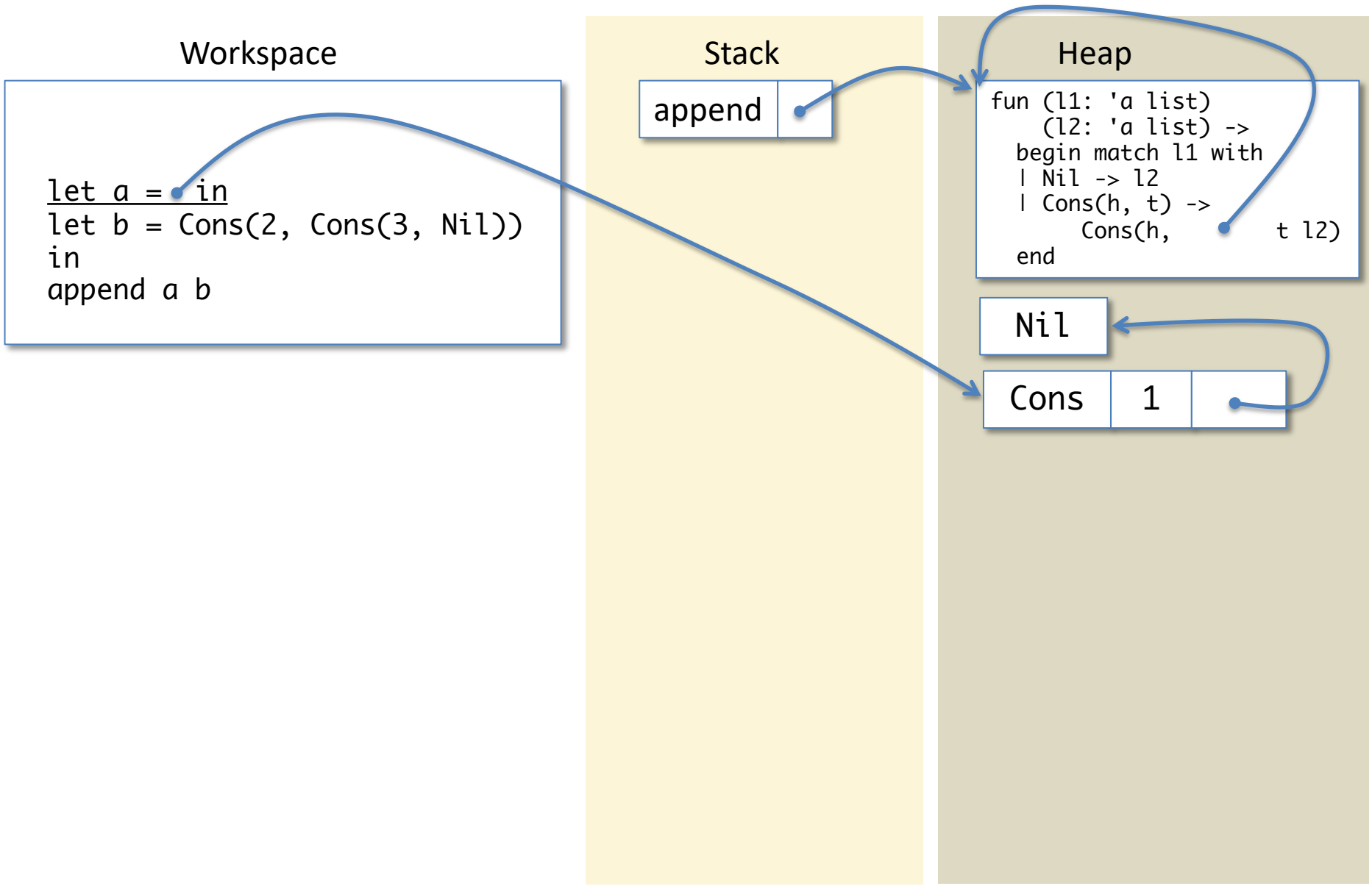
Allocate a Cons cell



Allocate a Cons cell



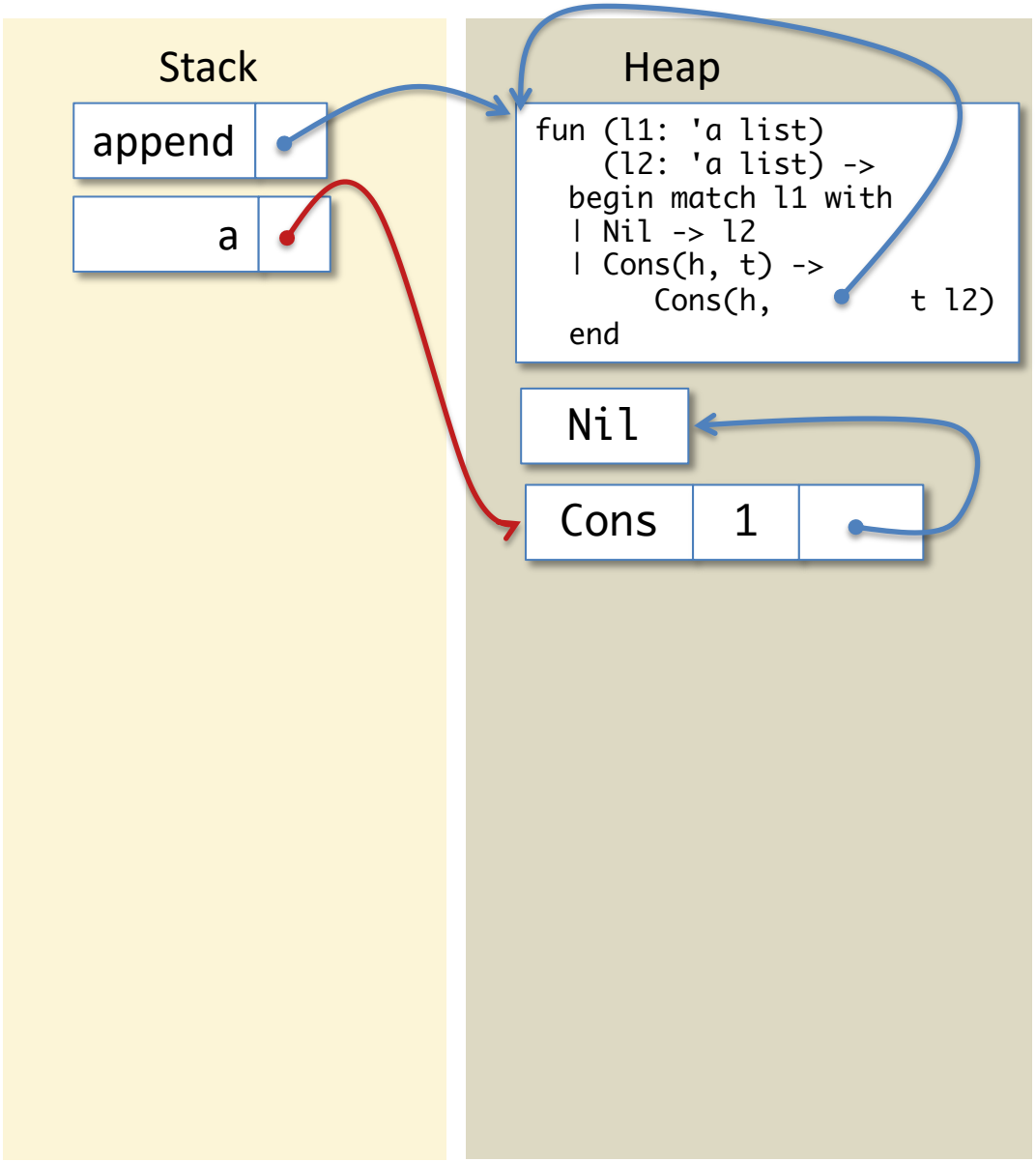
Let Expression



Create a Stack Binding

Workspace

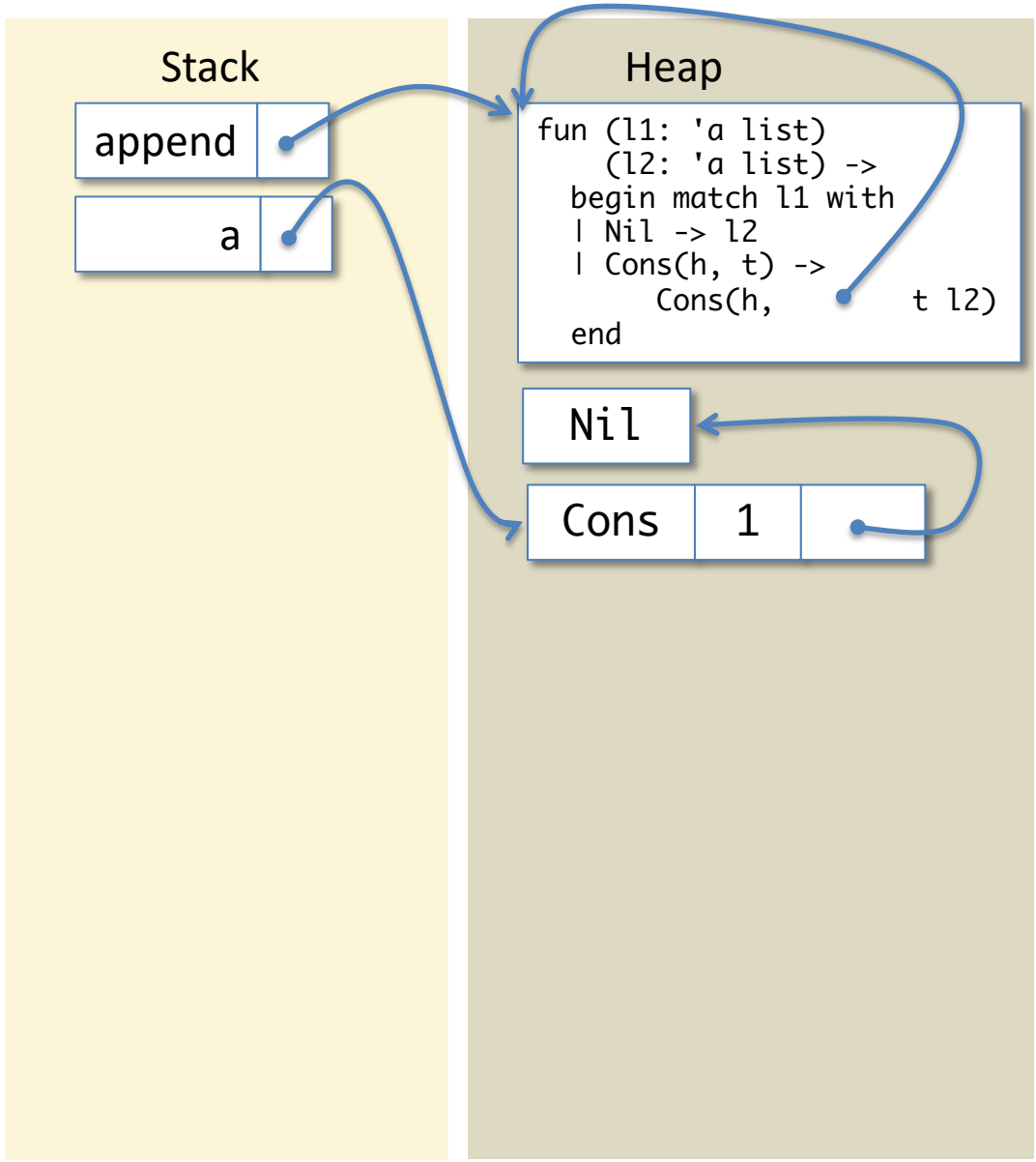
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```



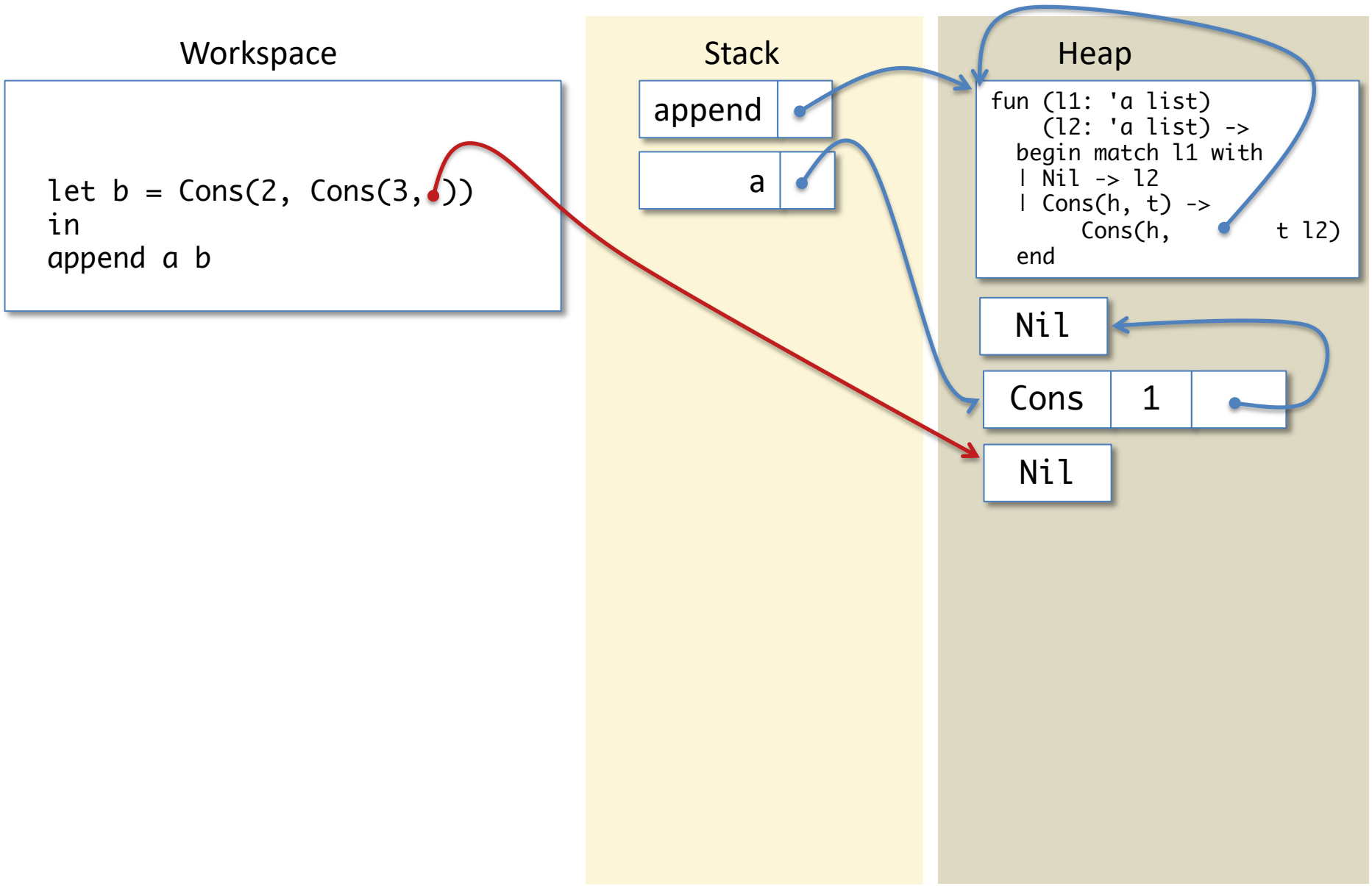
Allocate a Nil cell

Workspace

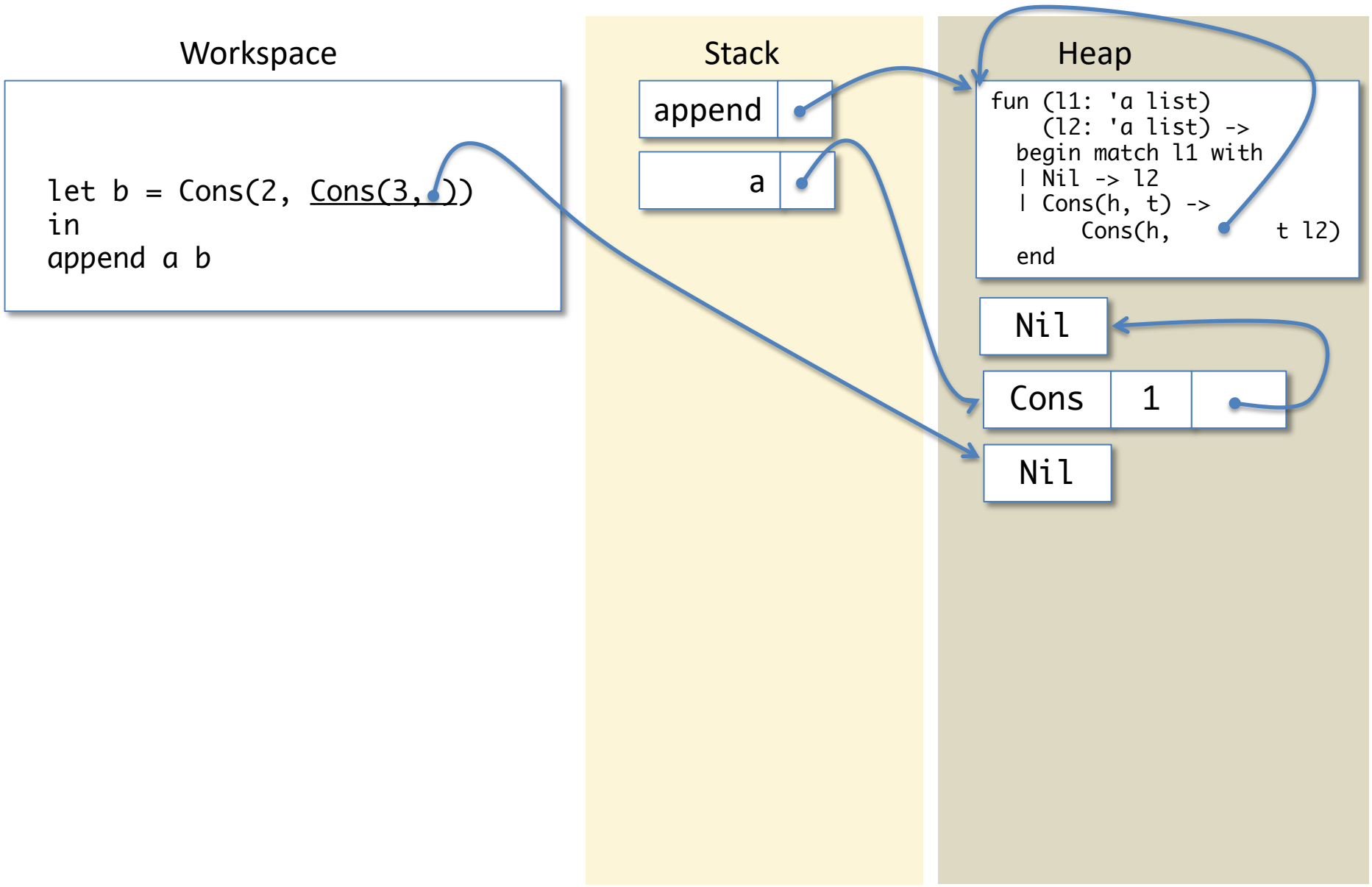
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```



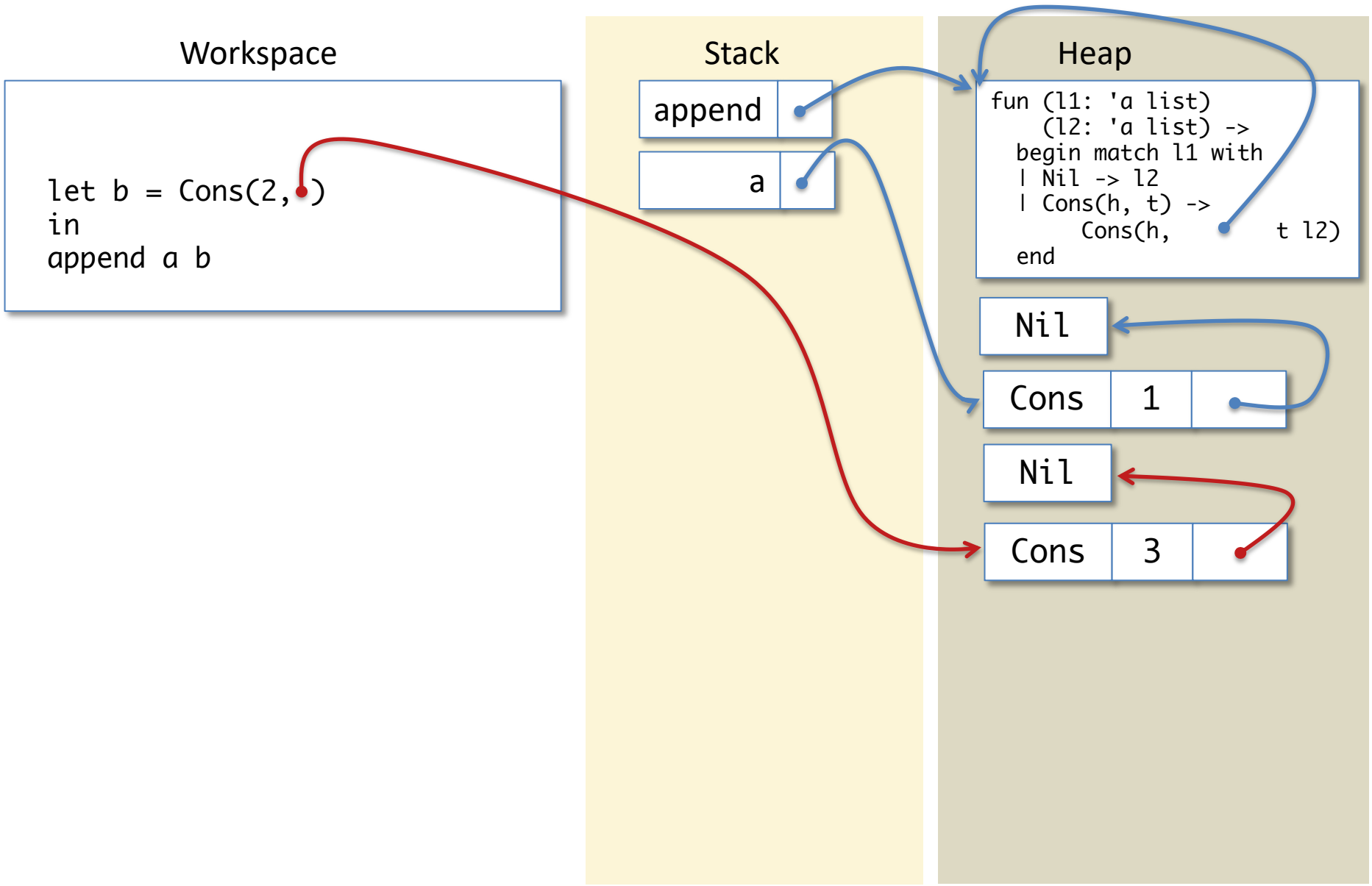
Allocate a Nil cell



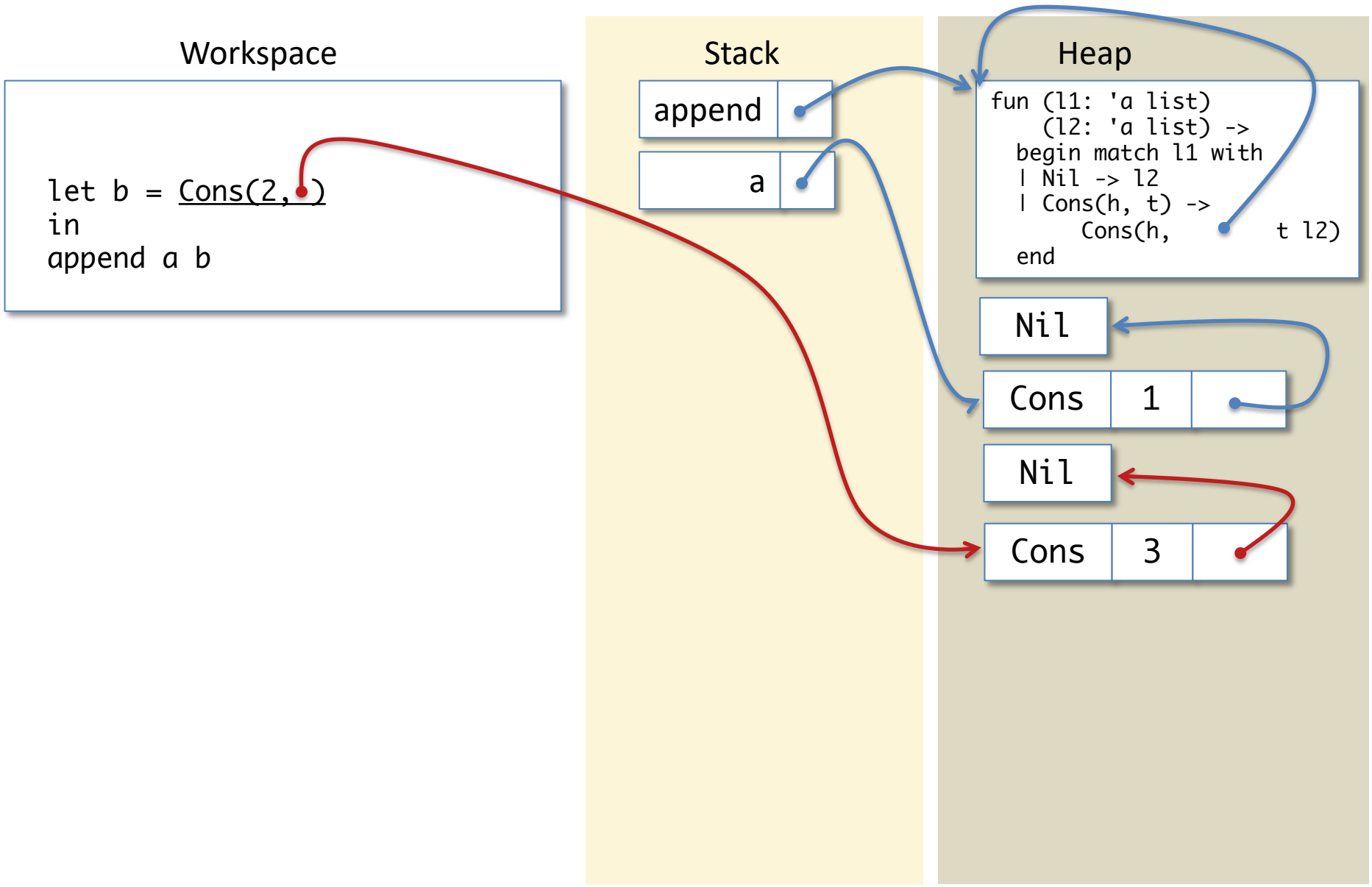
Allocate a Cons cell



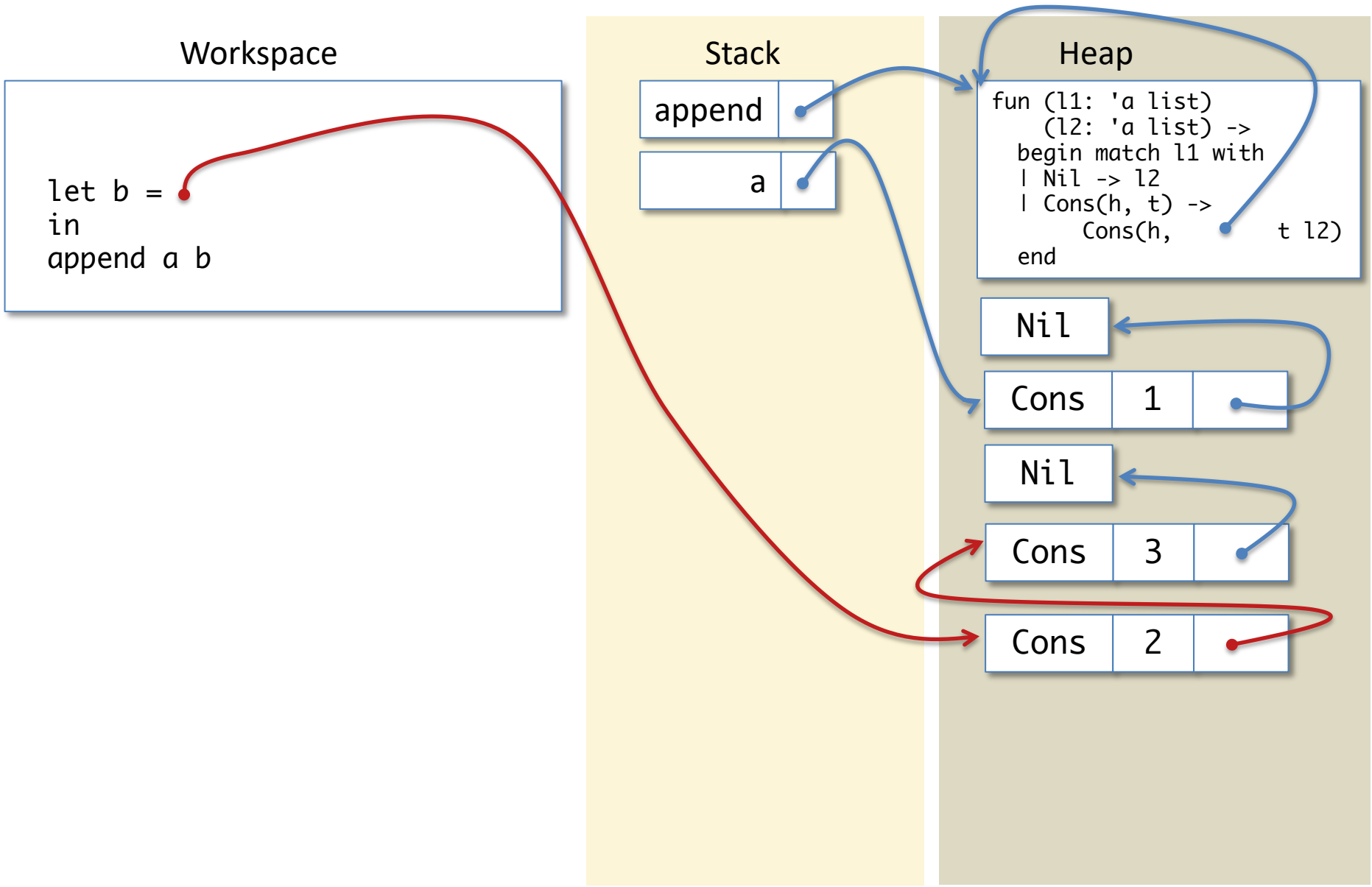
Allocate a Cons cell



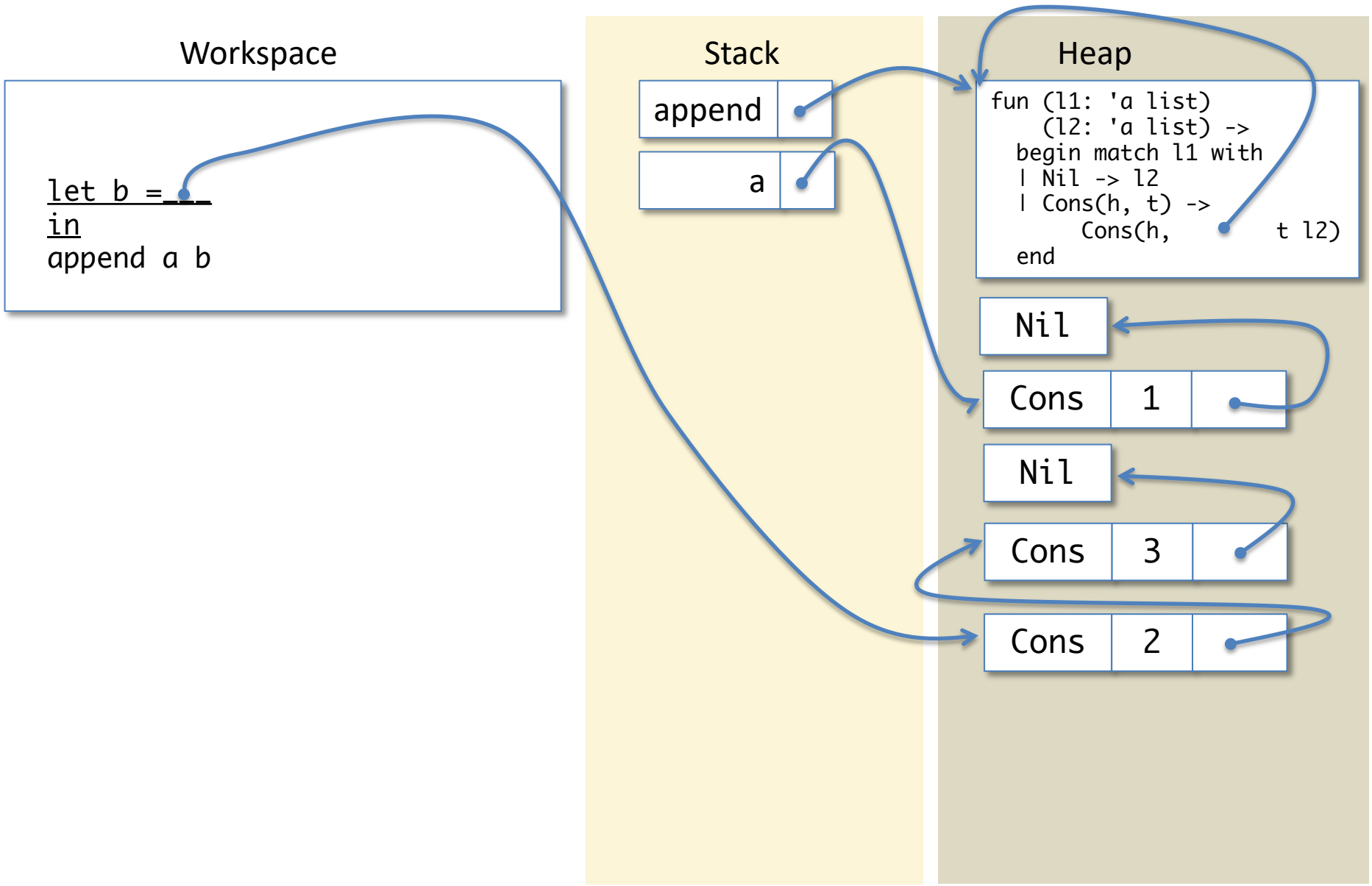
Allocate a Cons cell



Allocate a Cons cell



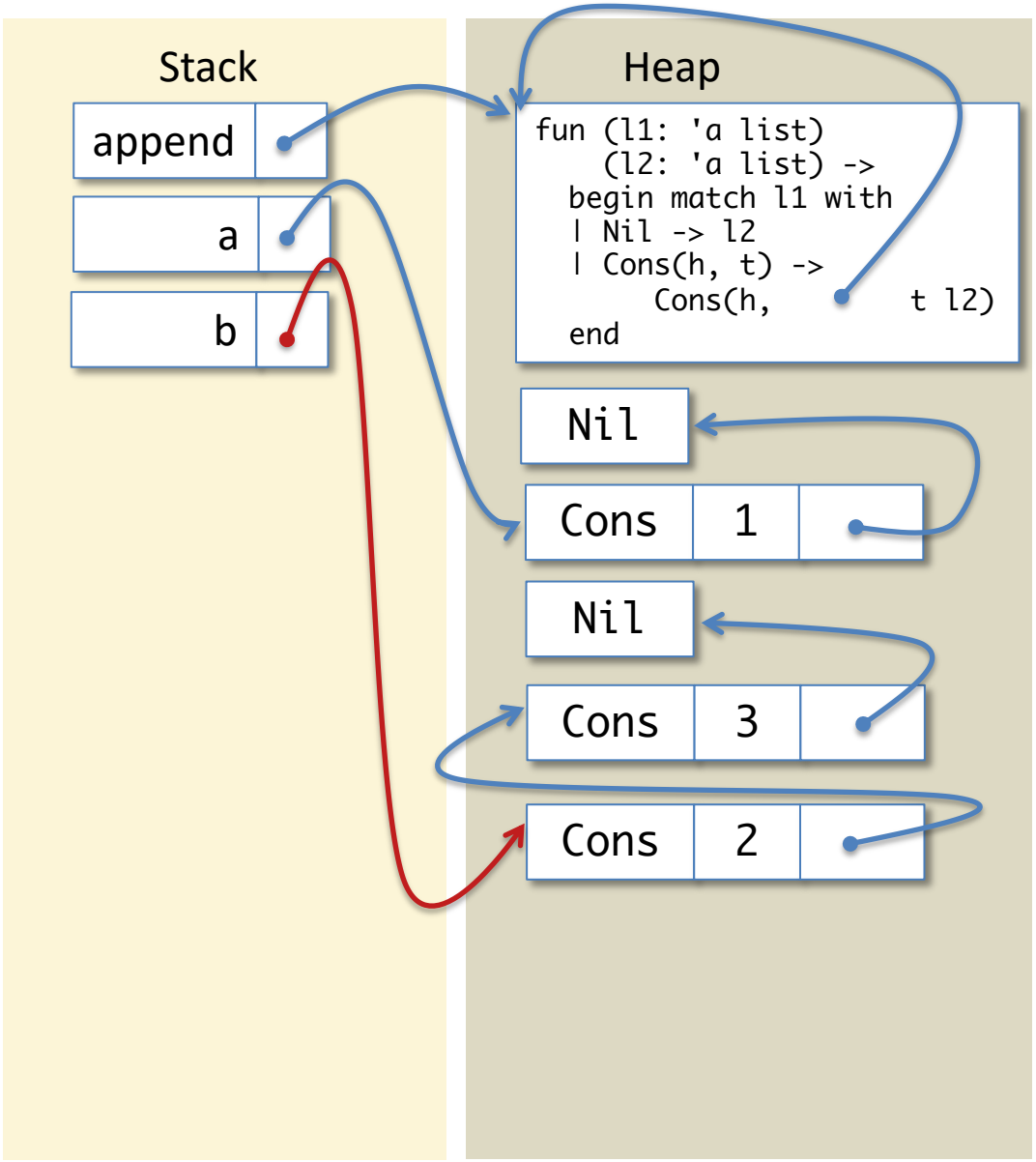
Let Expression



Create a Stack Binding

Workspace

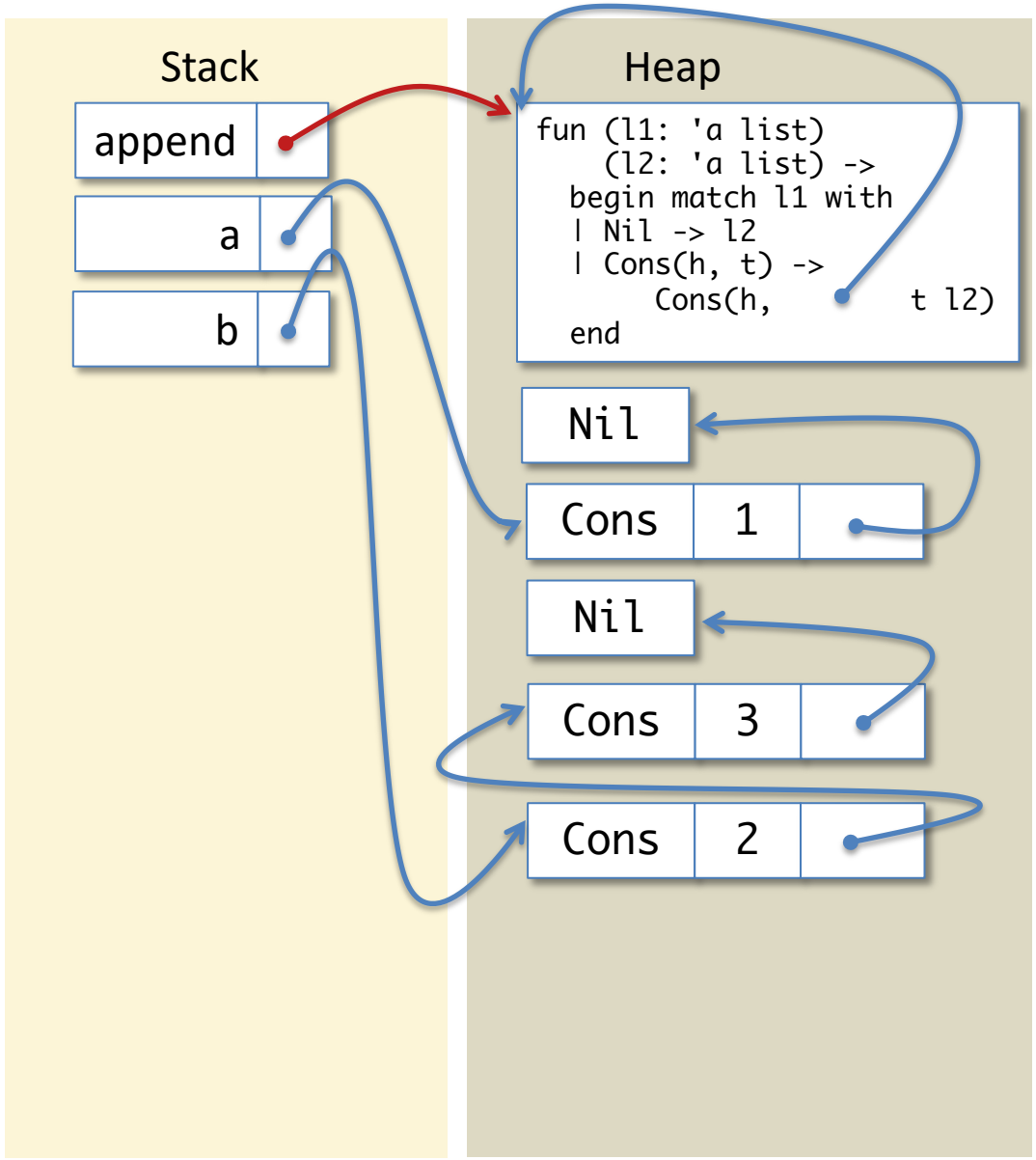
```
append a b
```



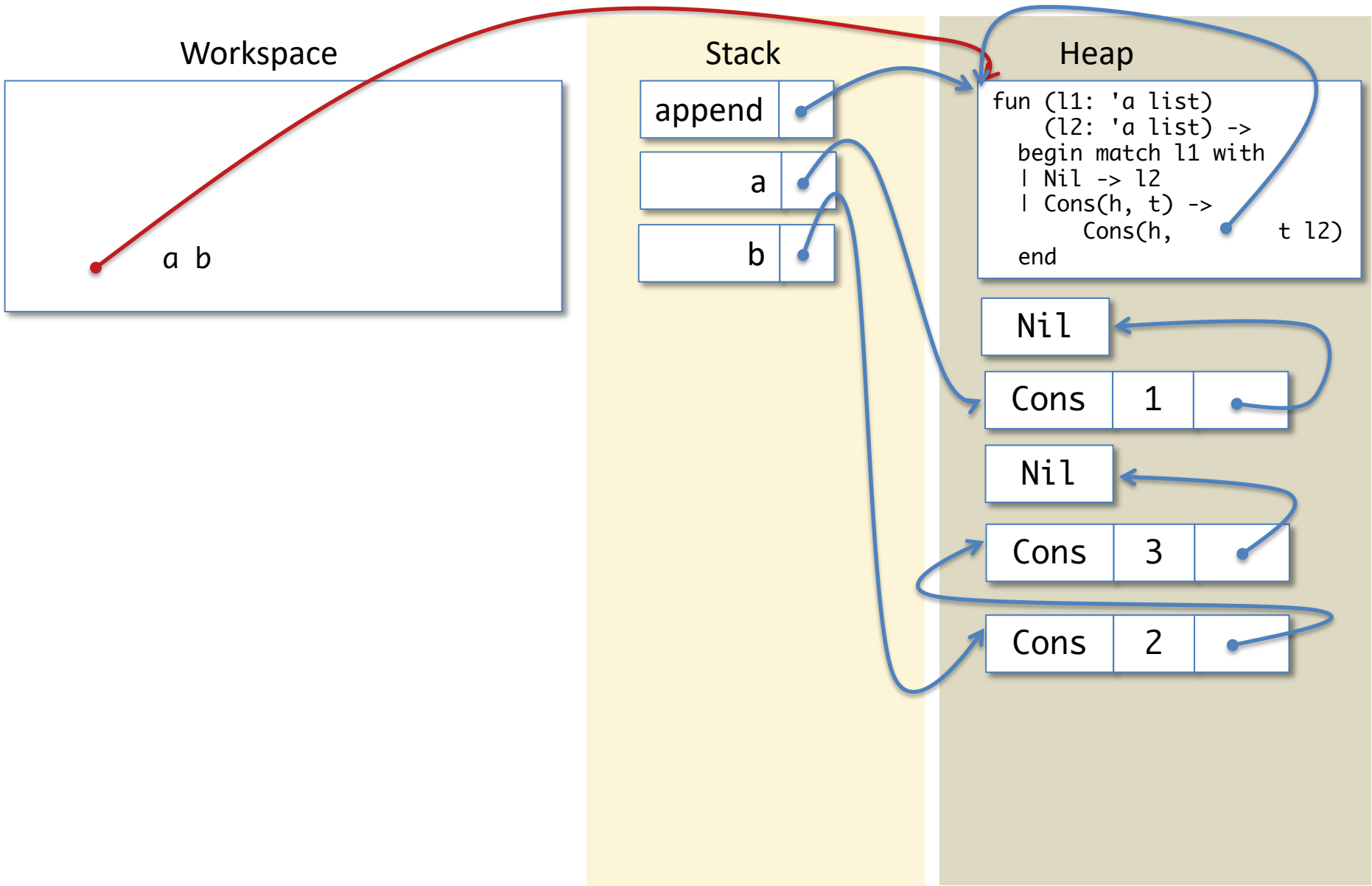
Lookup ‘append’

Workspace

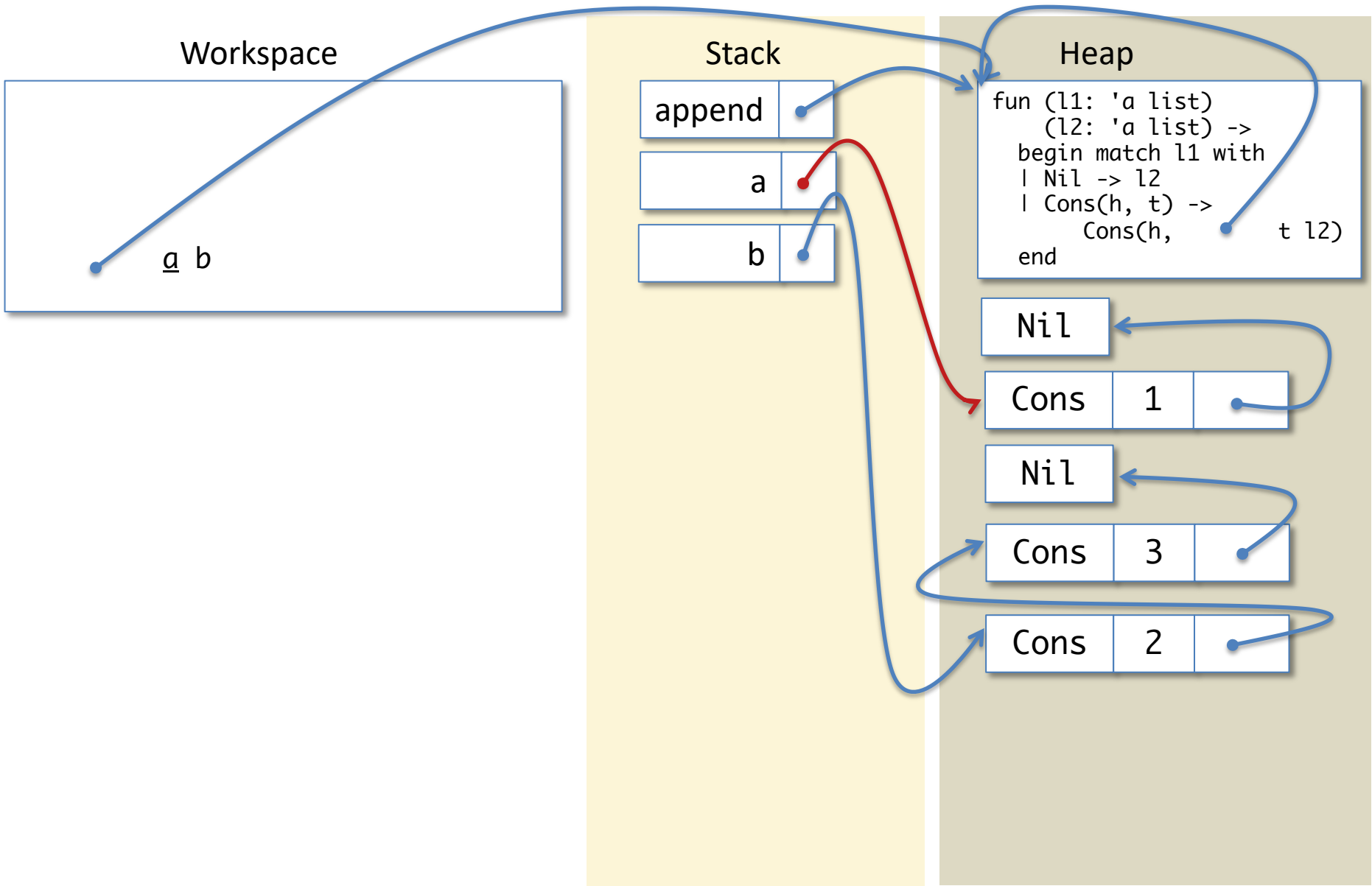
append a b



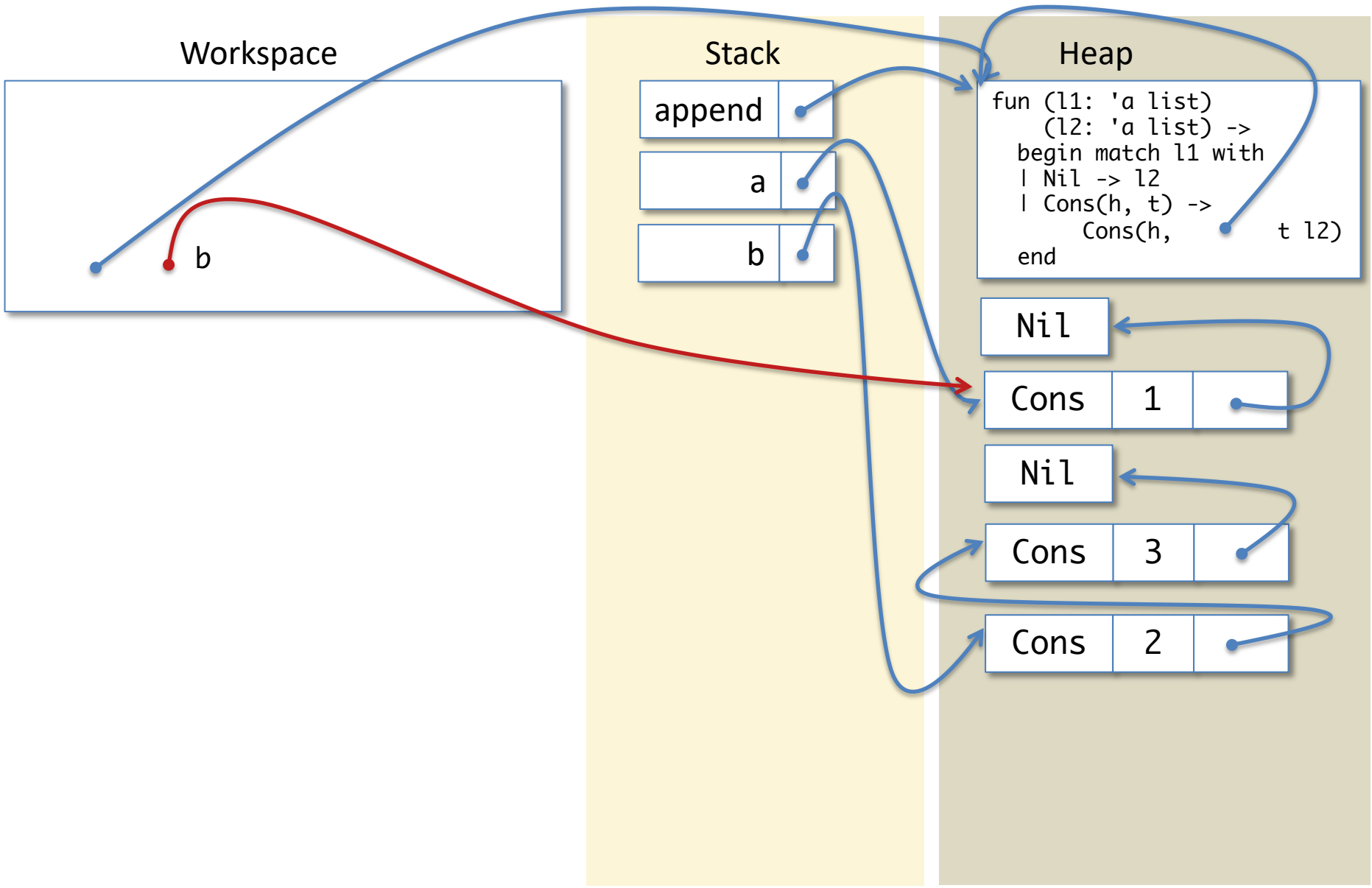
Lookup ‘append’



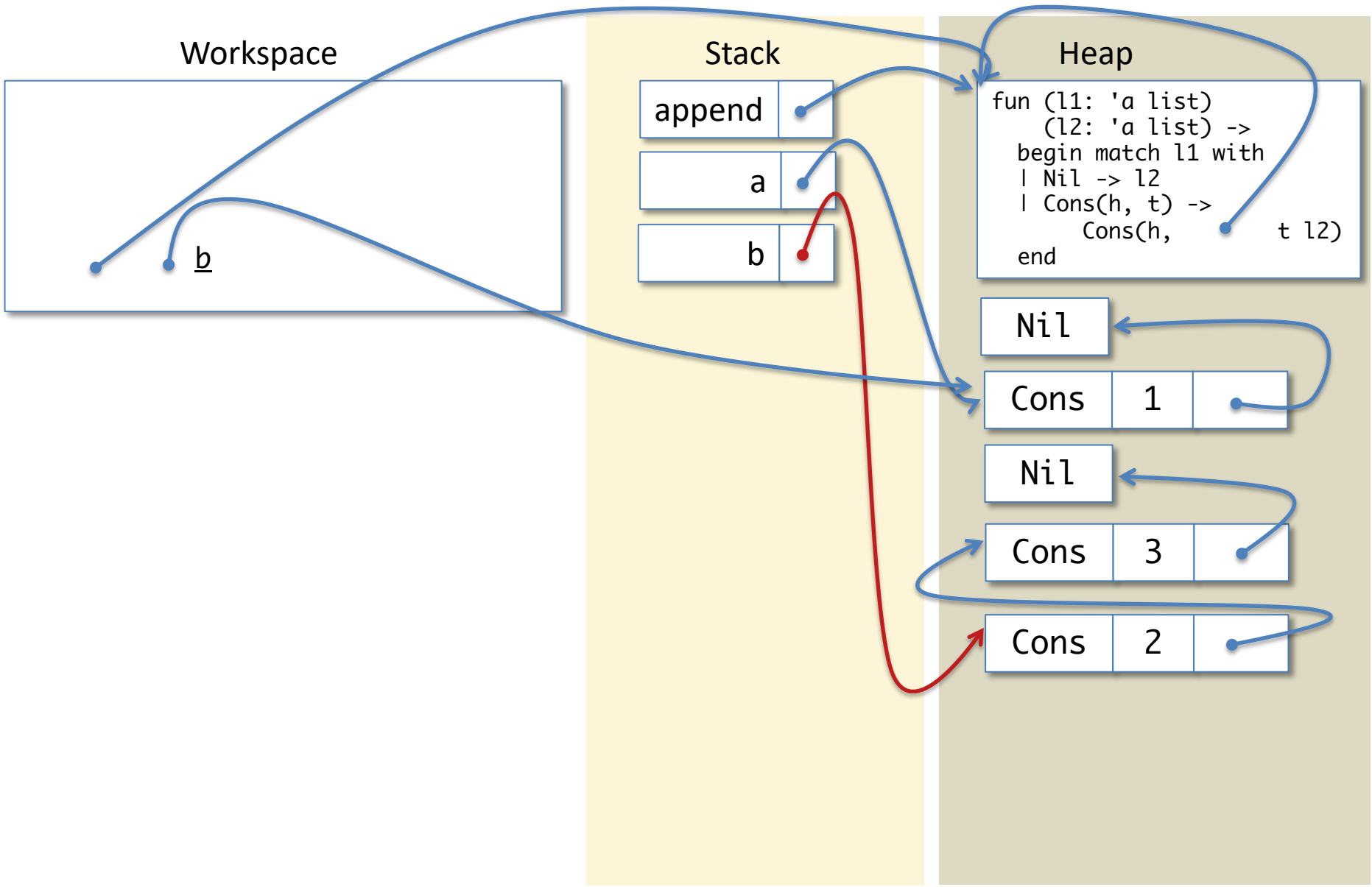
Lookup ‘a’


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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

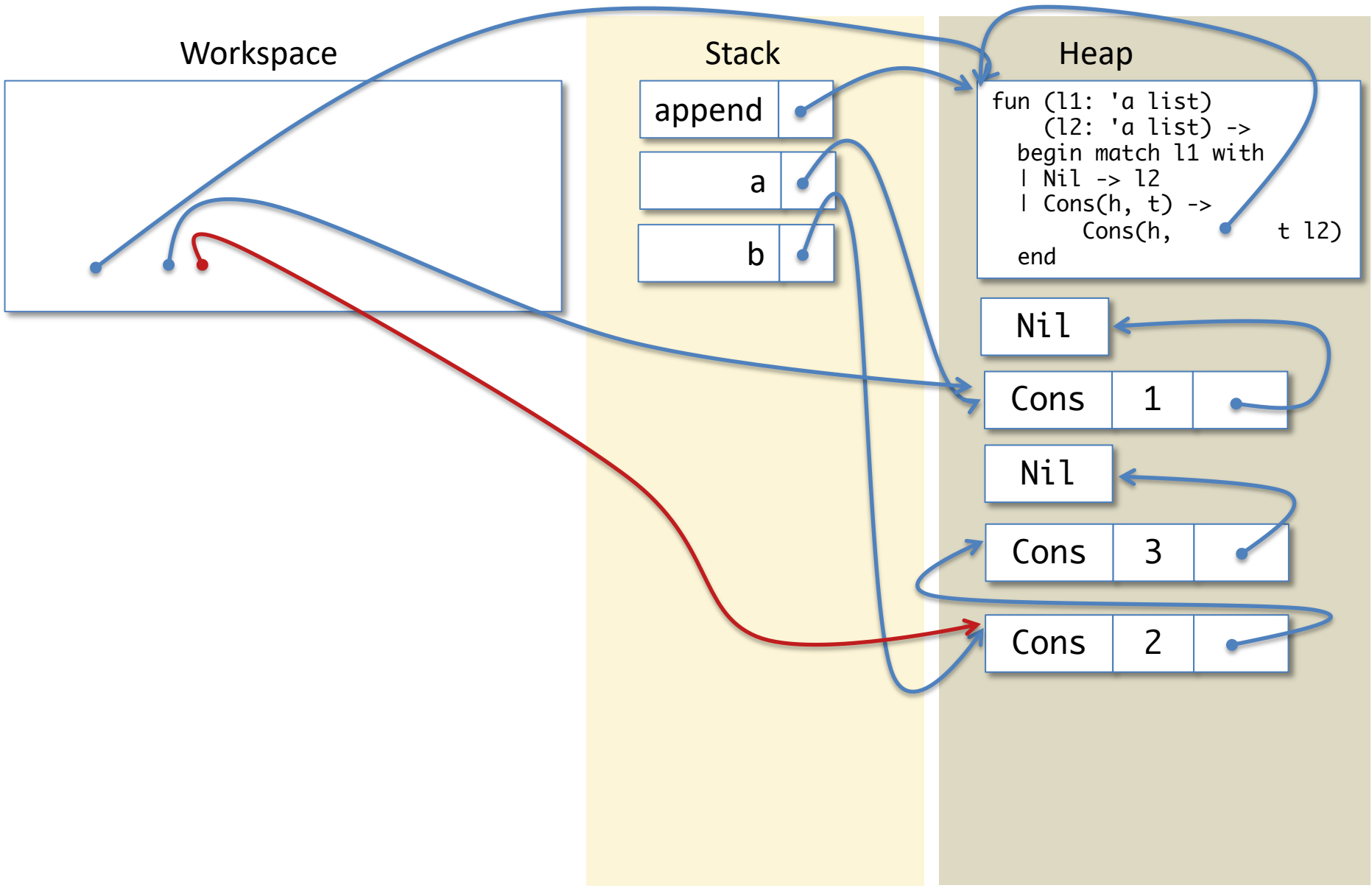
Lookup ‘a’


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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

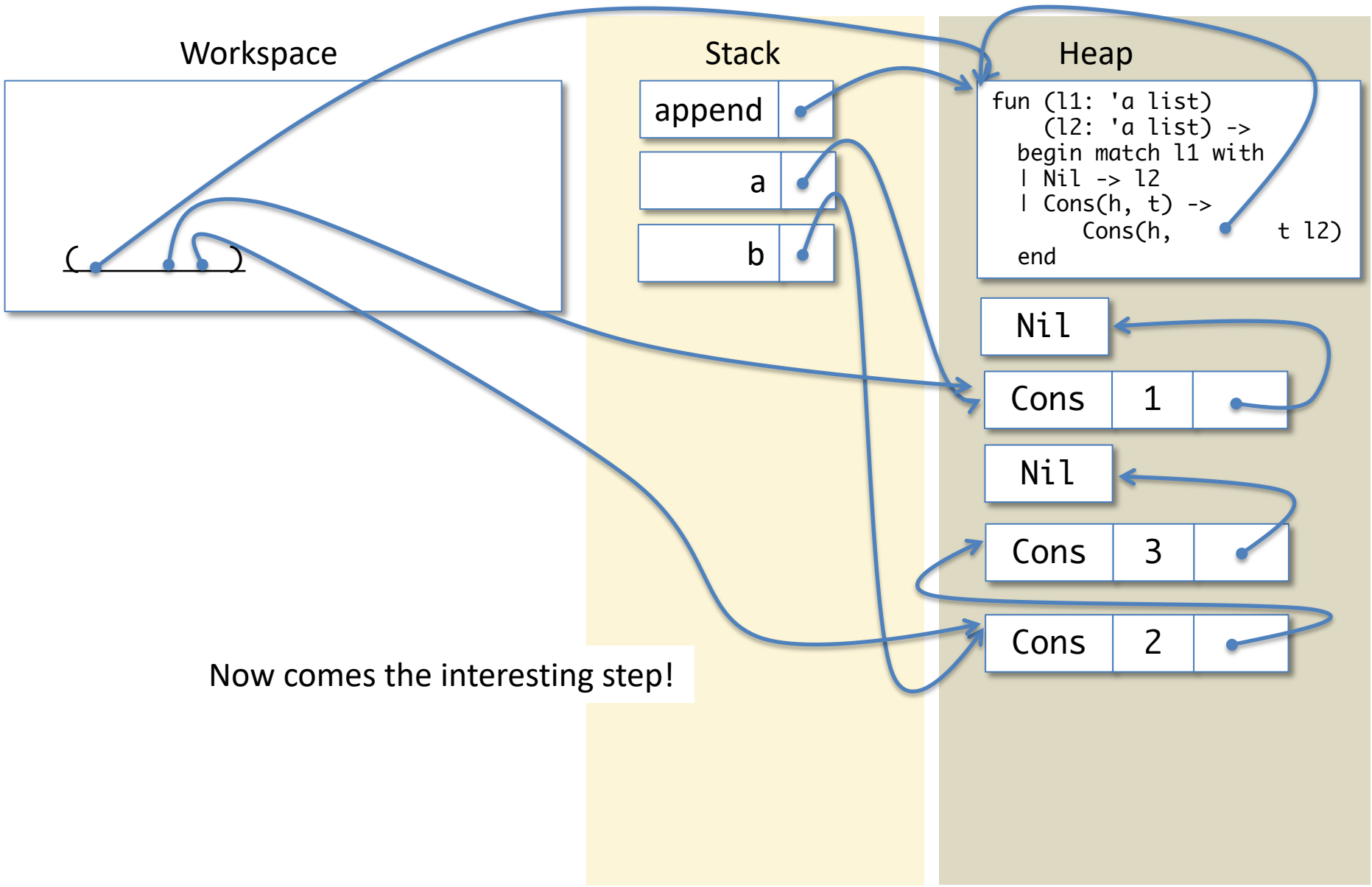
Lookup ‘b’


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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

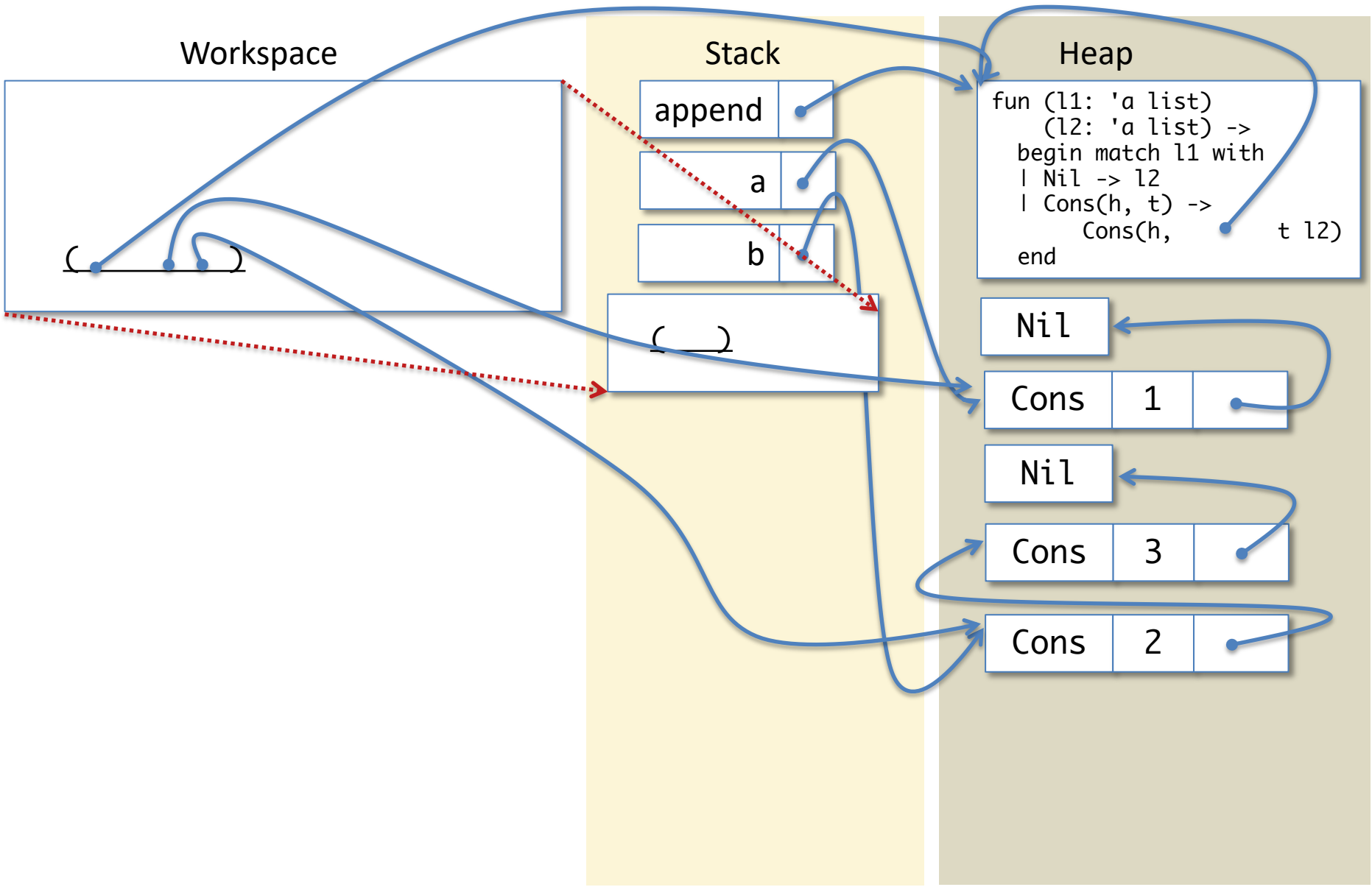
Lookup ‘b’


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

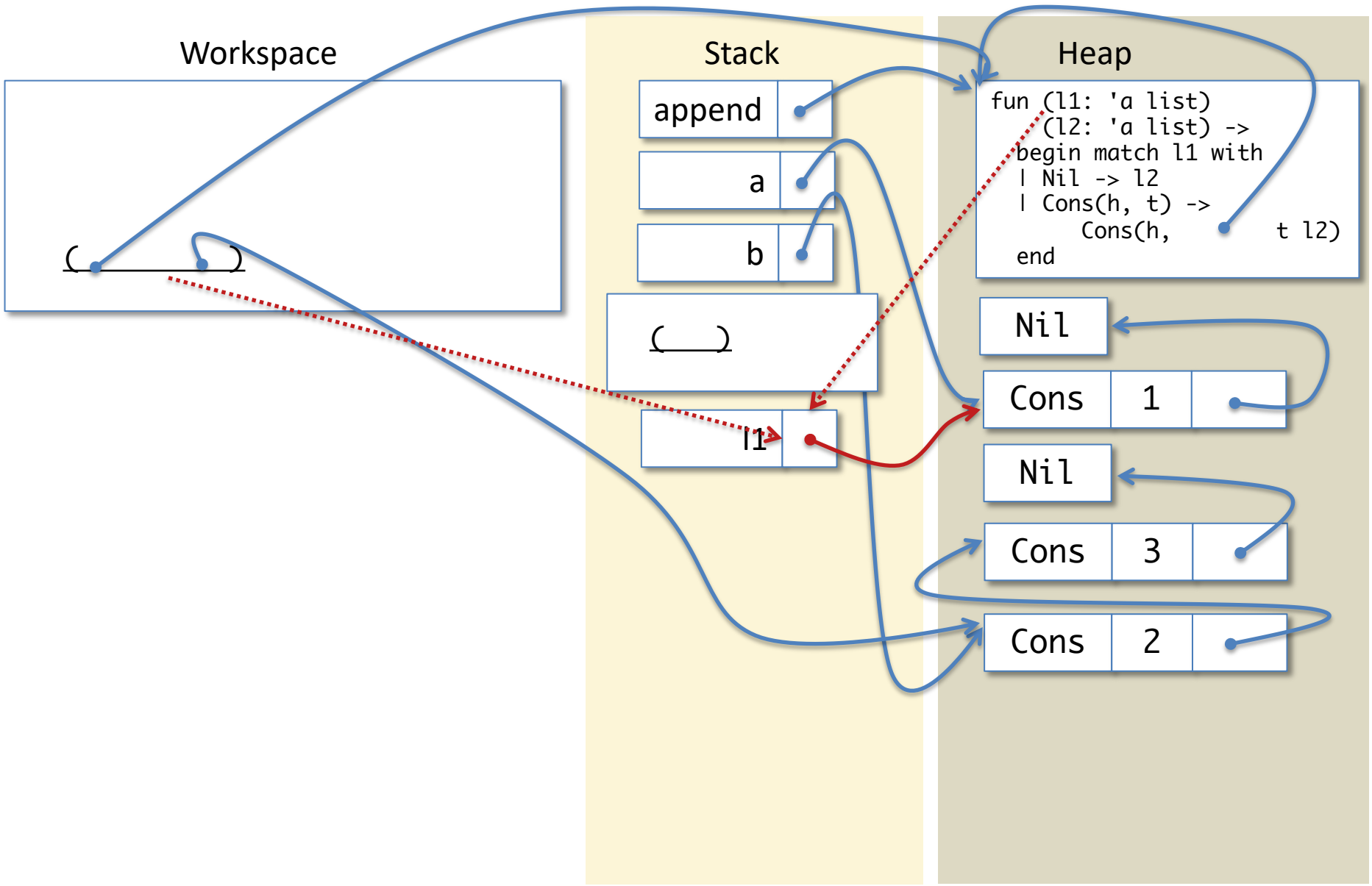
Do the Function call



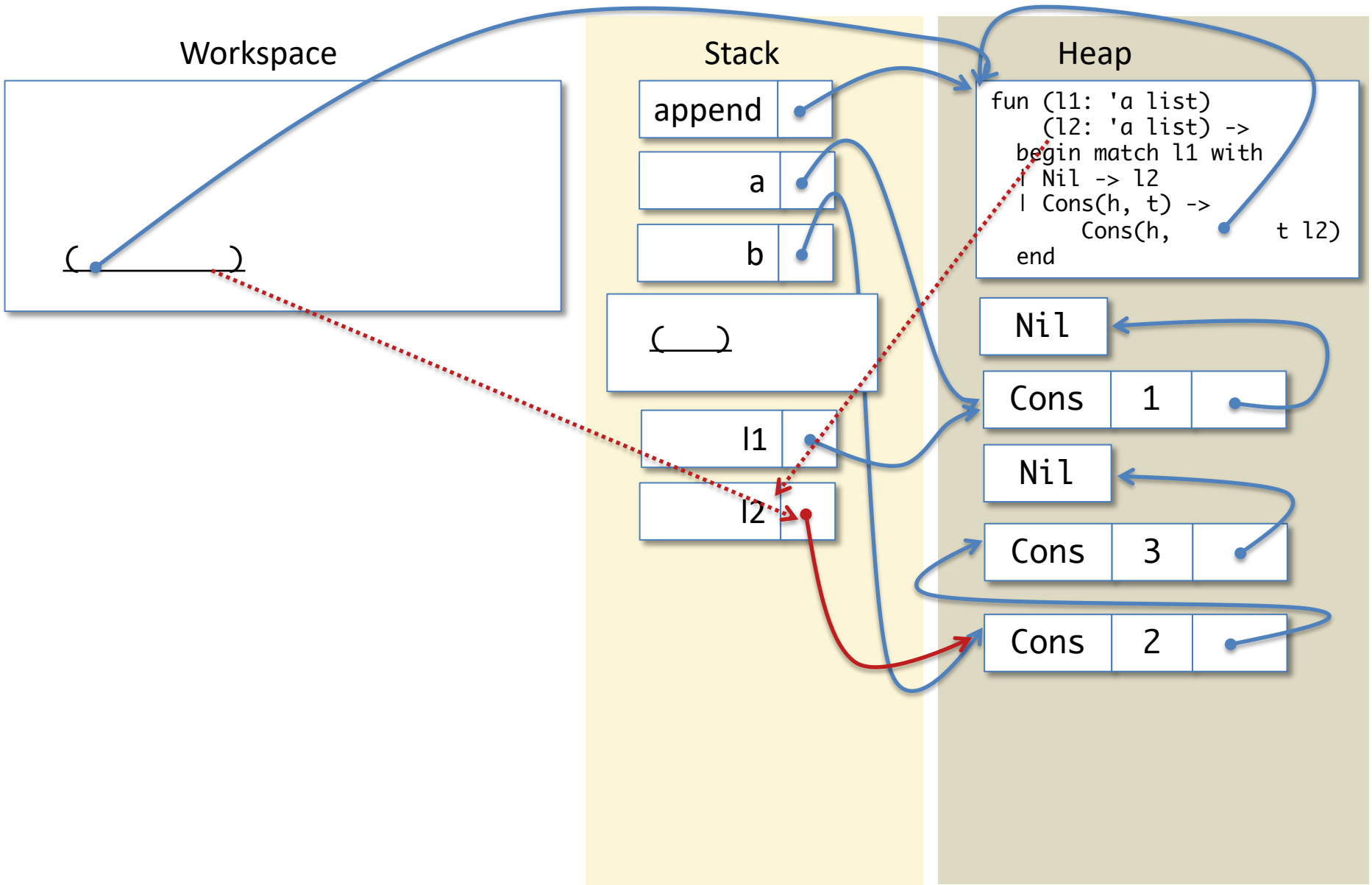
Call (1): Save Workspace



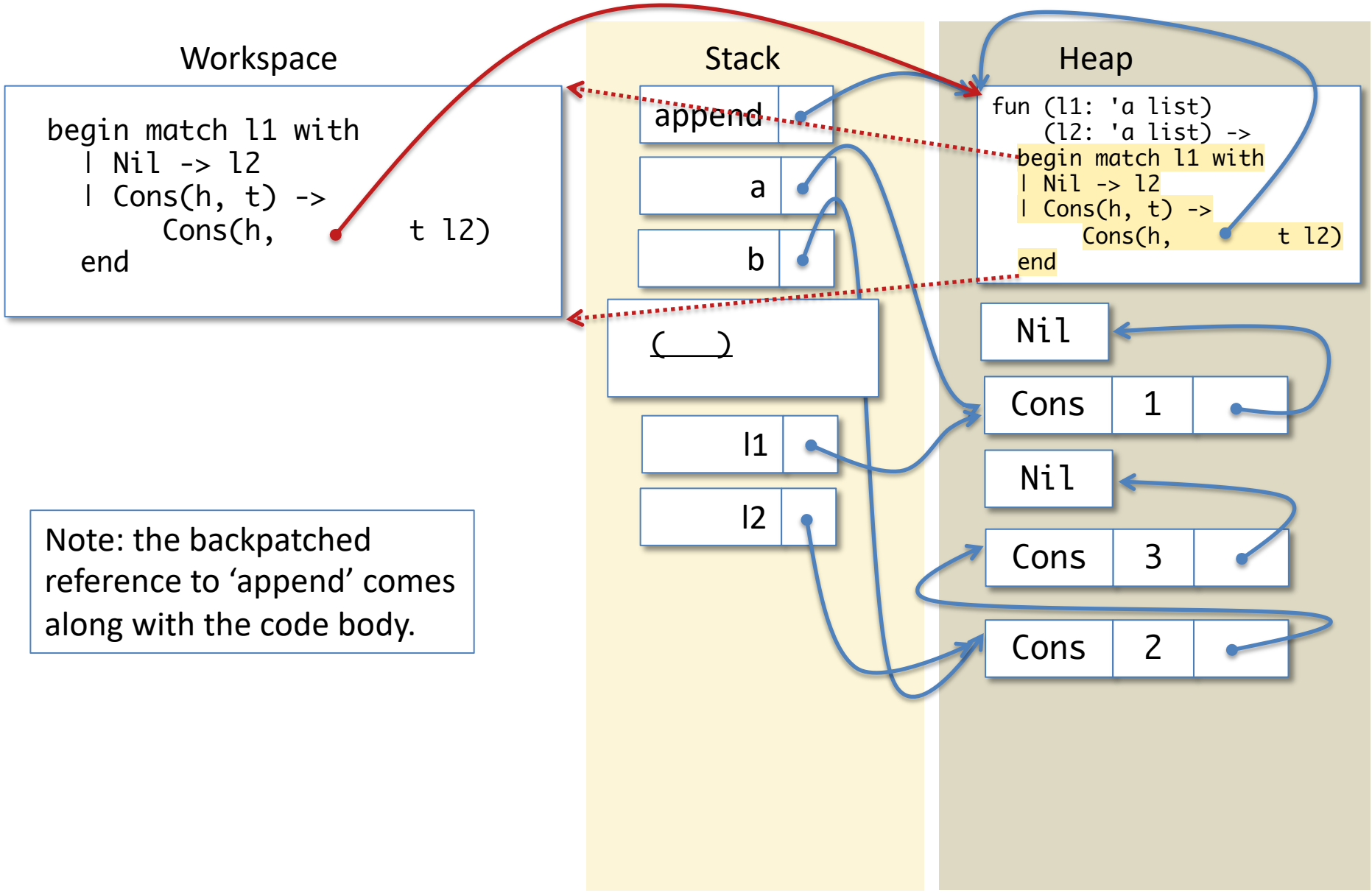
push l1


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

push l2

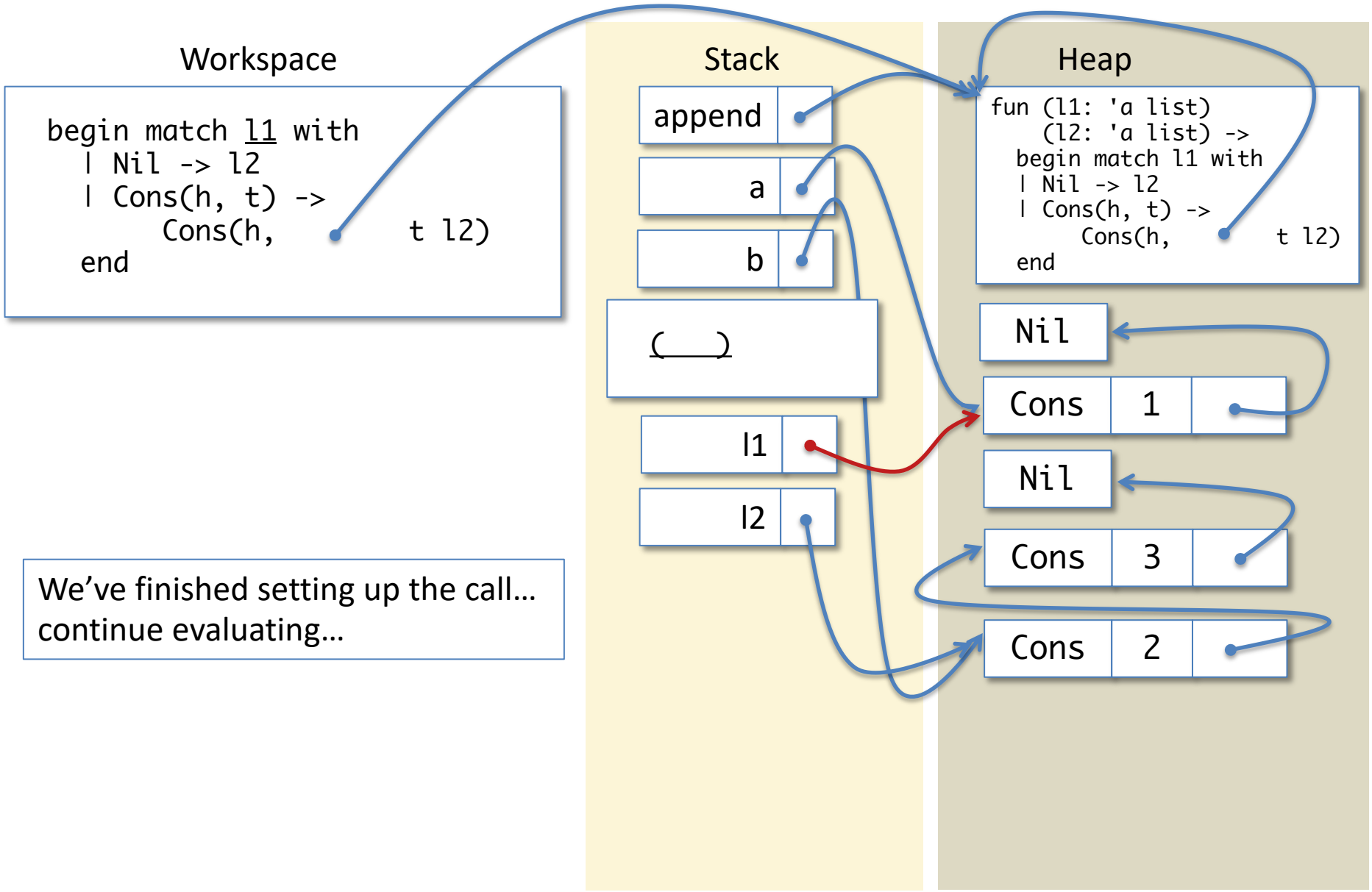

Workspace
Stack
Heap
append
a
b
l1
l2
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

Install Function Body in Workspace



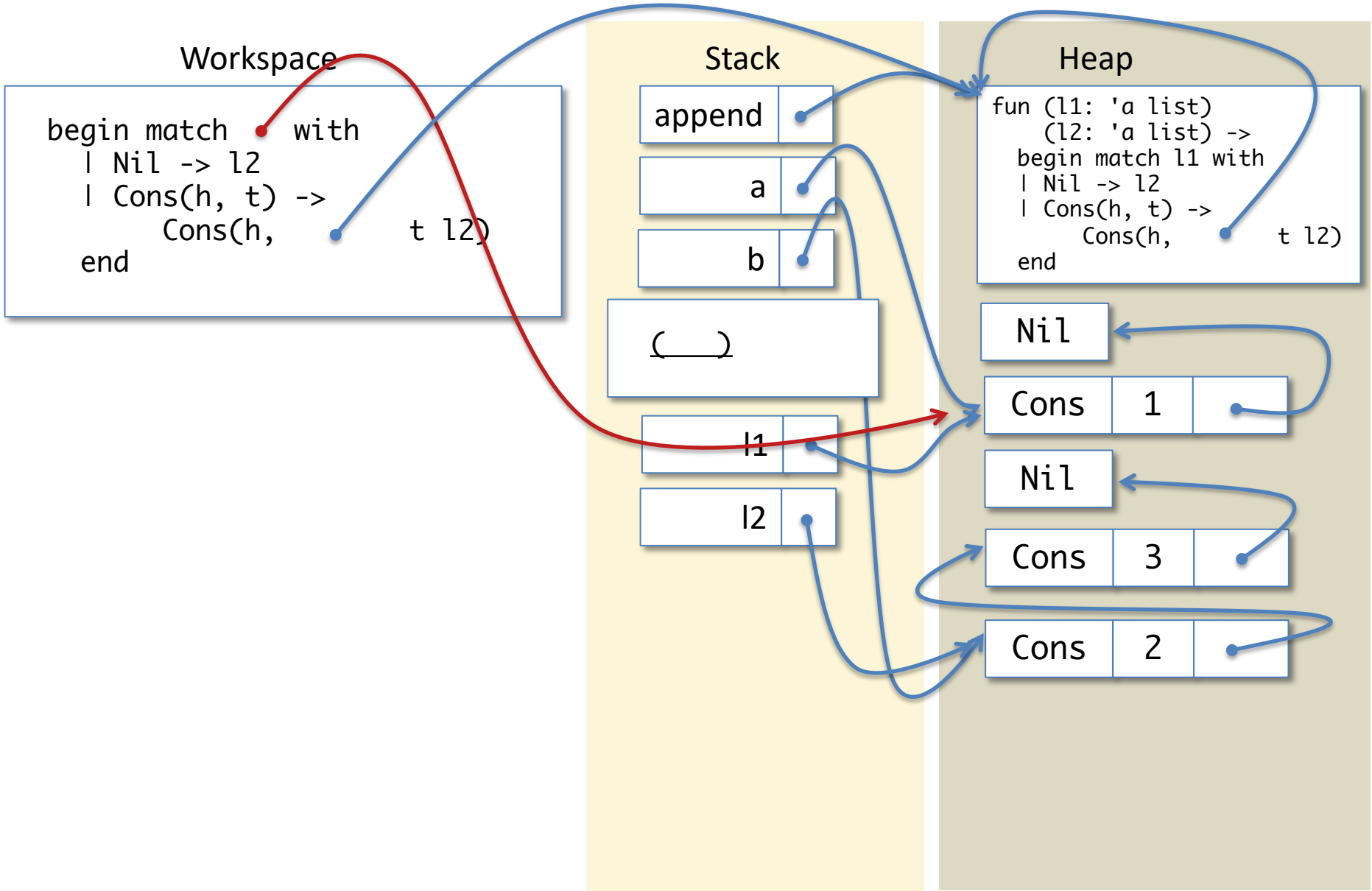
Note: the backpatched reference to 'append' comes along with the code body.

Lookup l1

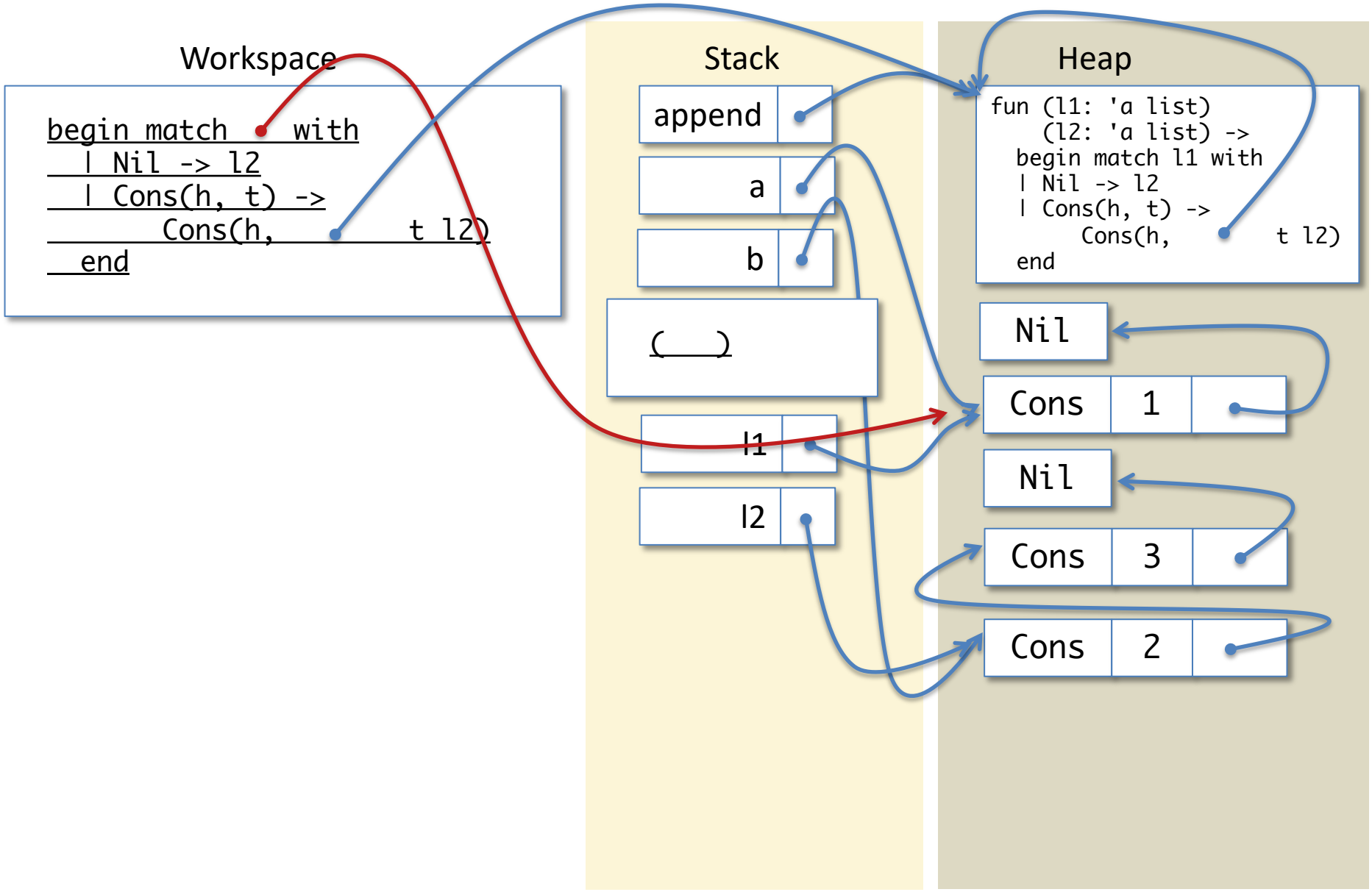


We've finished setting up the call…
continue evaluating…

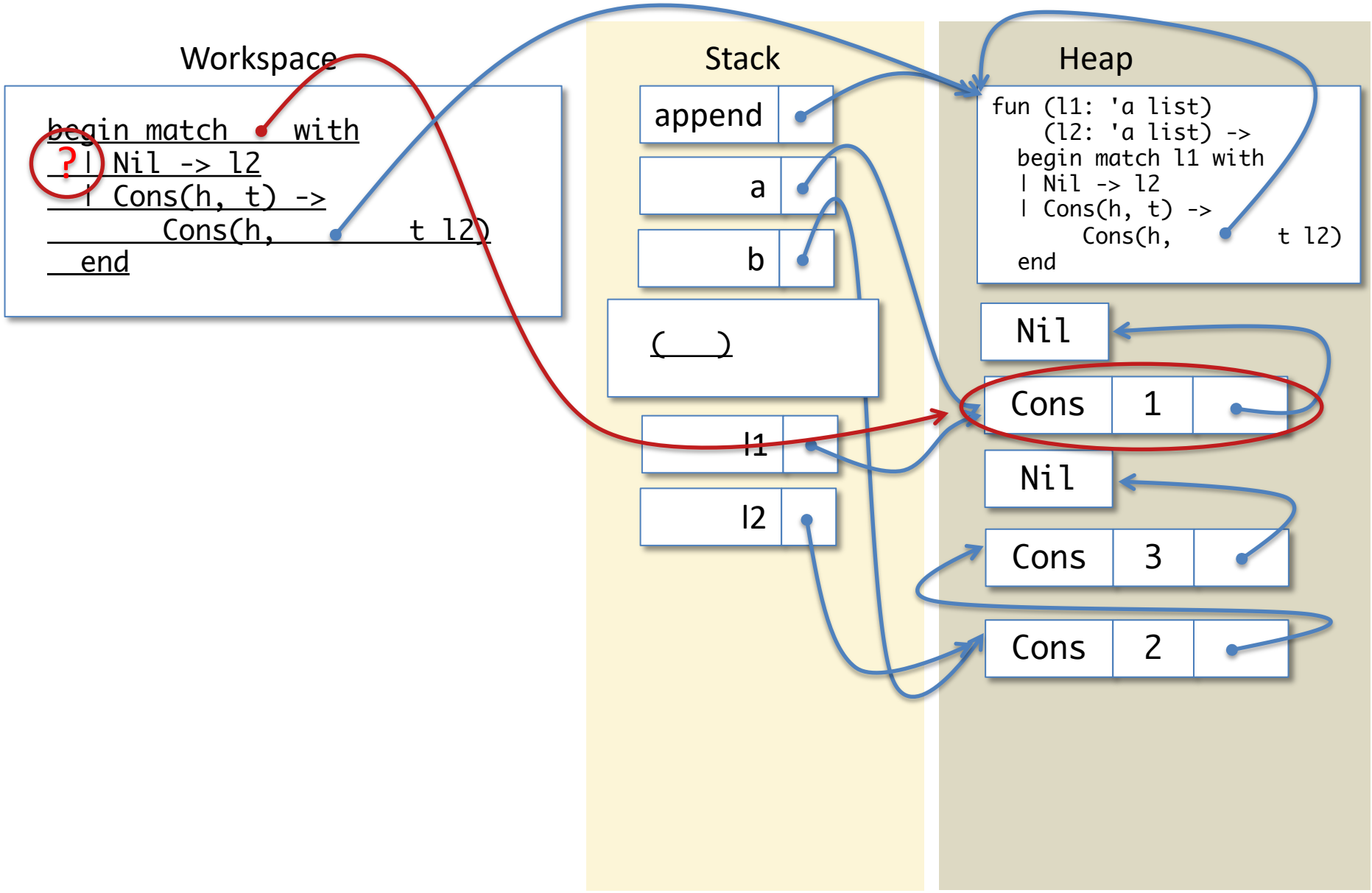
Lookup l1


Workspace
begin match with
| Nil -> l2
| Cons(h, t) ->
Cons(h, 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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

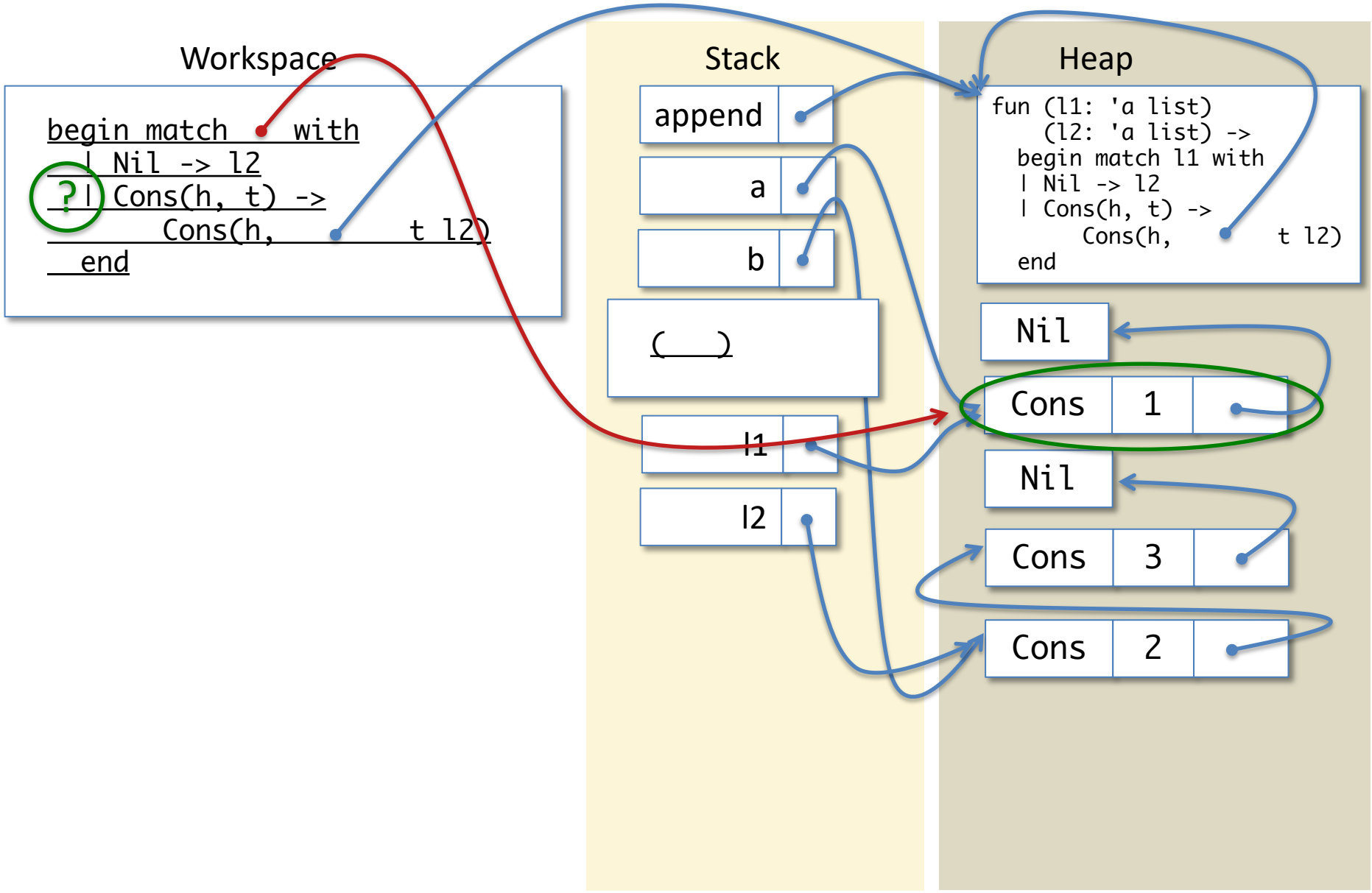
Match Expression



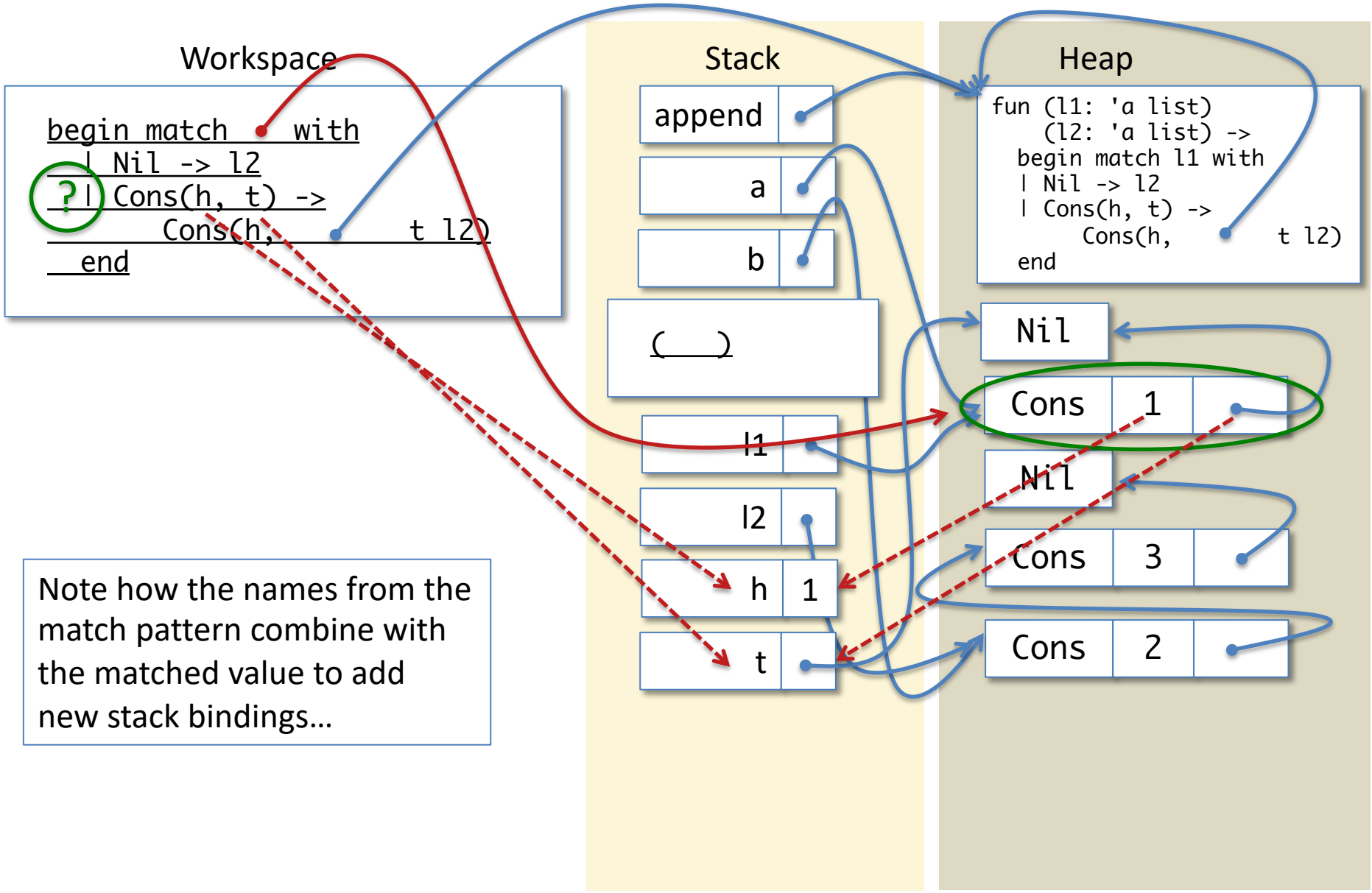
Nil Case Doesn't Match



Cons Case *Does* Match

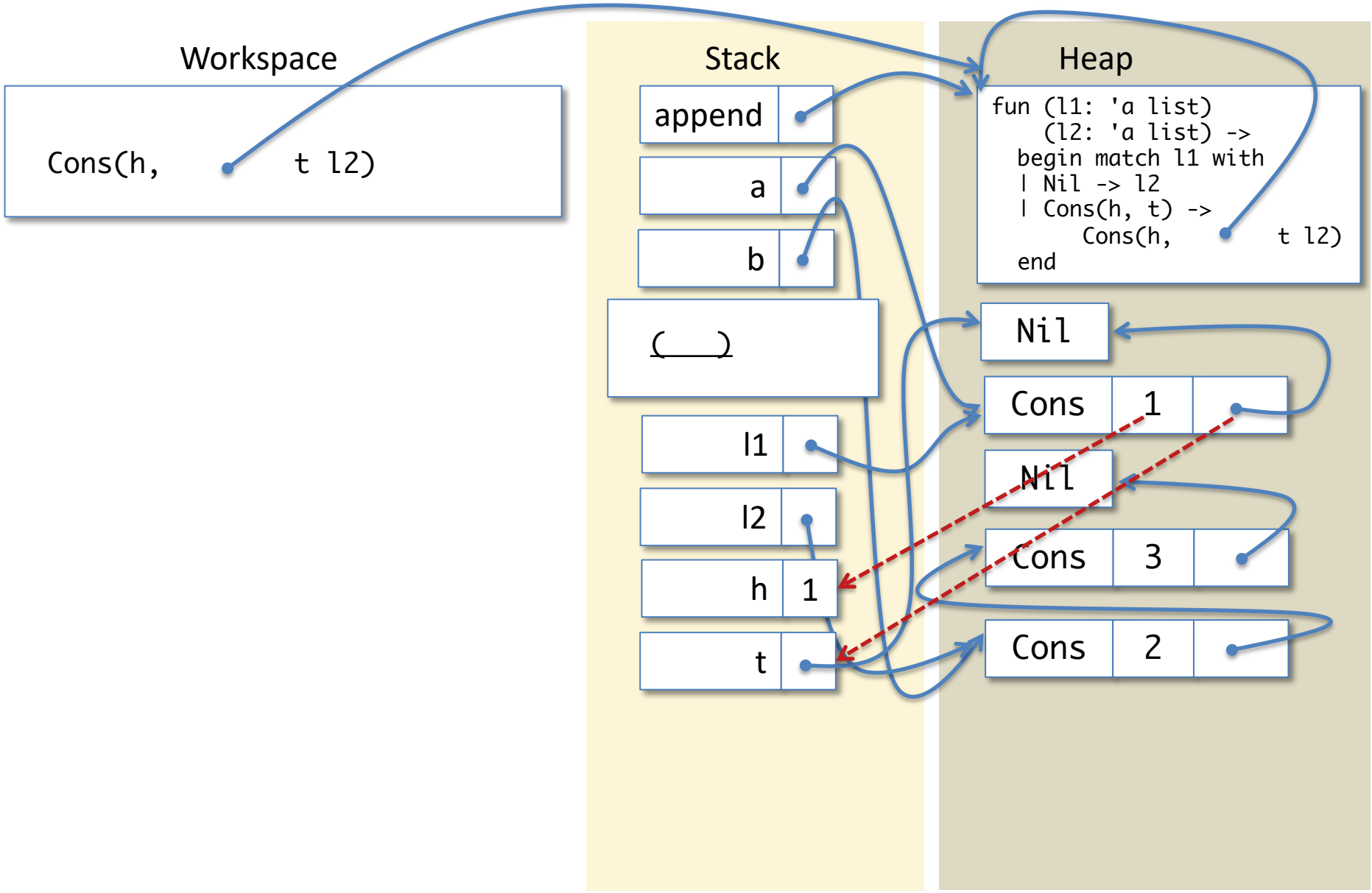


Cons Case *Does* Match

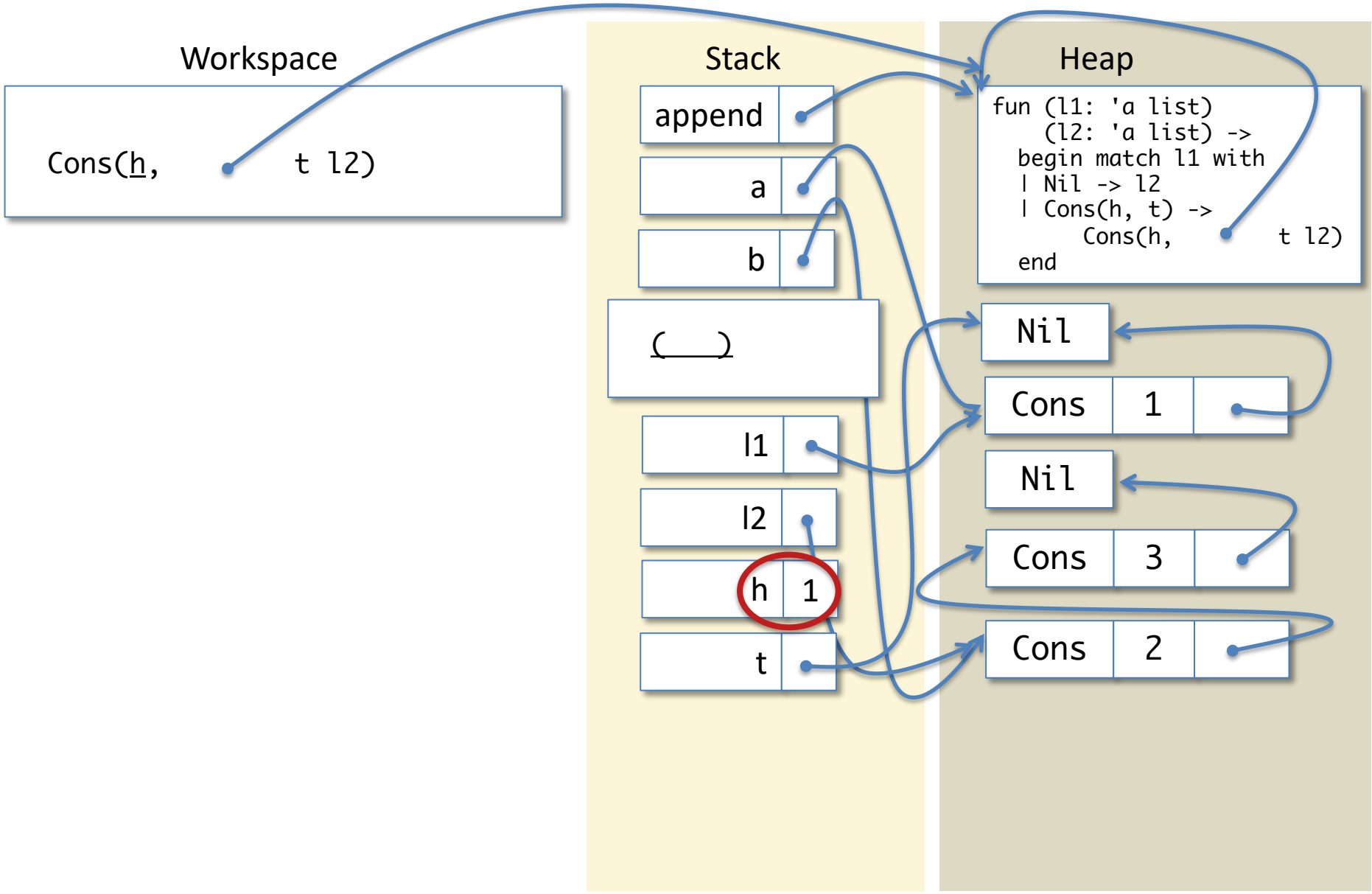


Note how the names from the match pattern combine with the matched value to add new stack bindings…

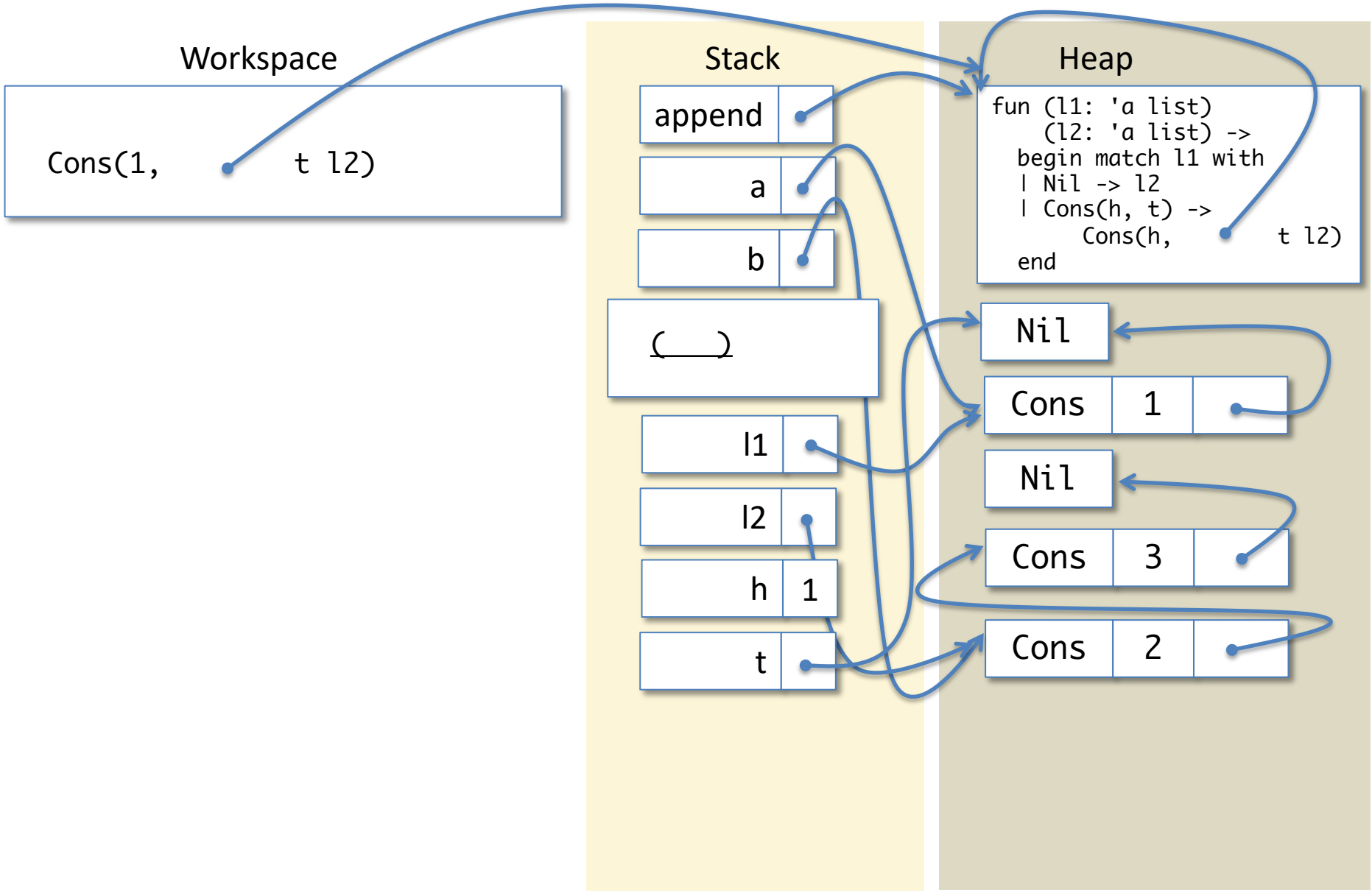
Simplify the Branch: Push h, t



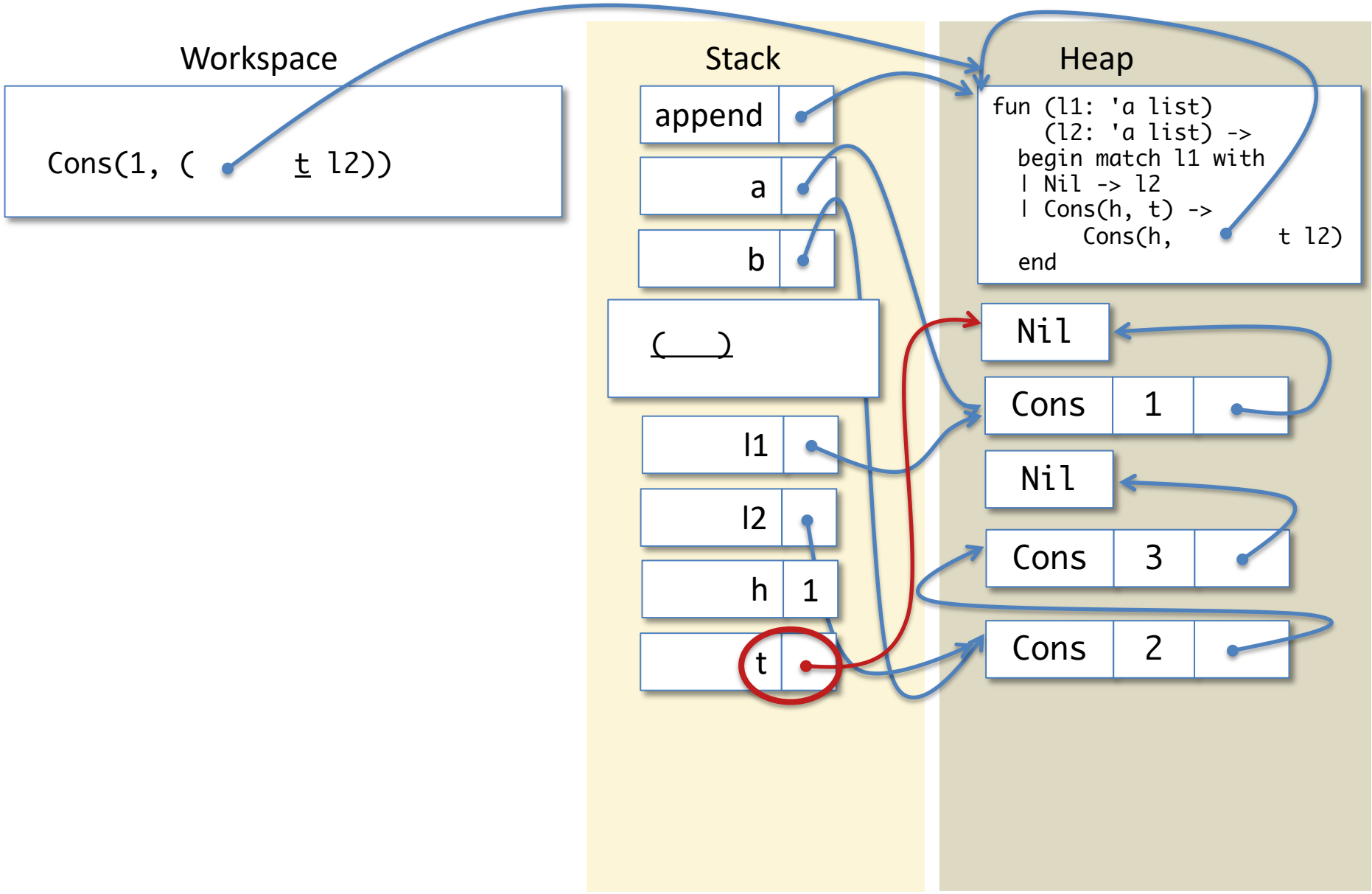
Lookup ‘h’


Workspace
Cons(h, 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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

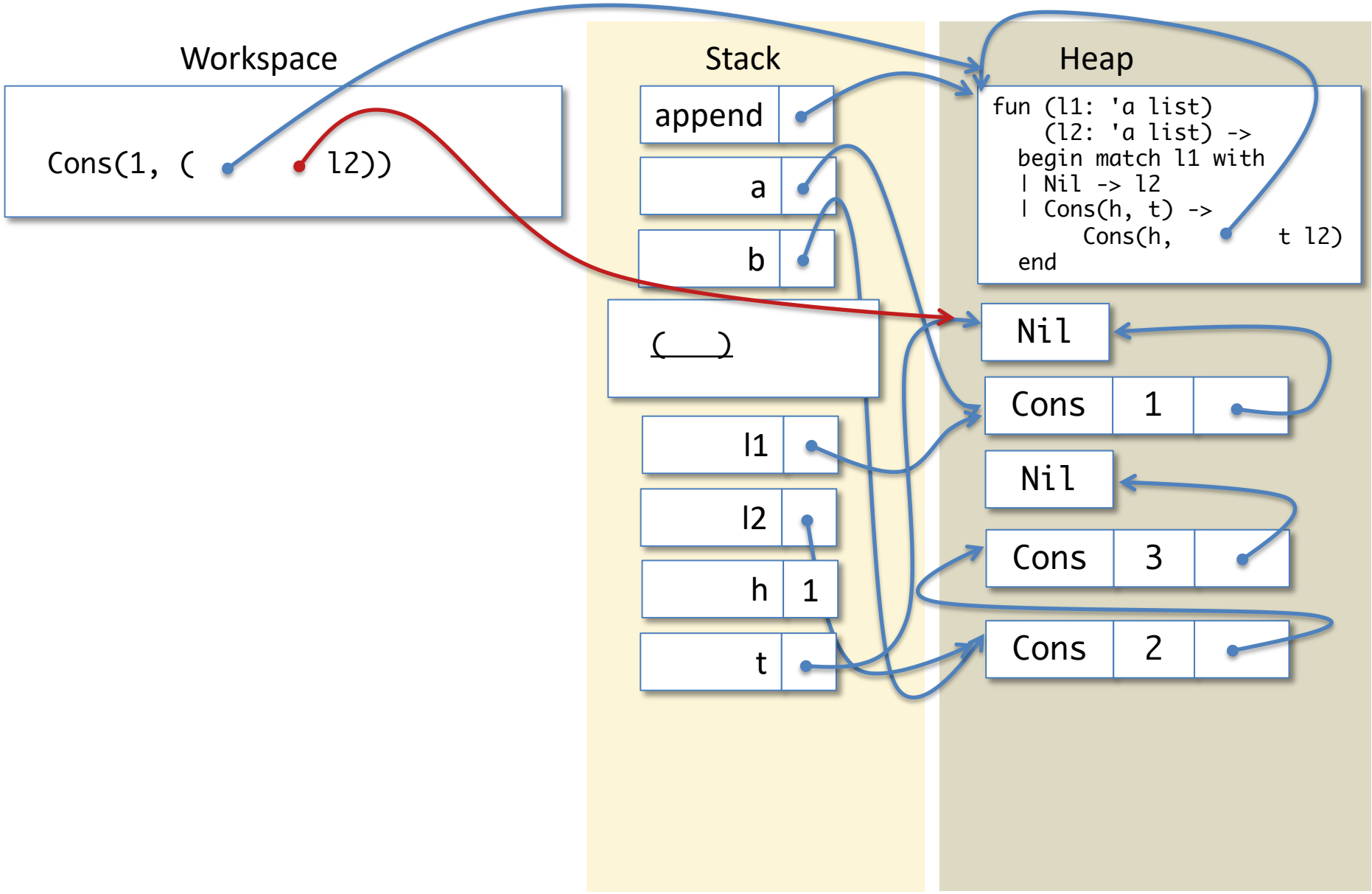
Lookup ‘h’


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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

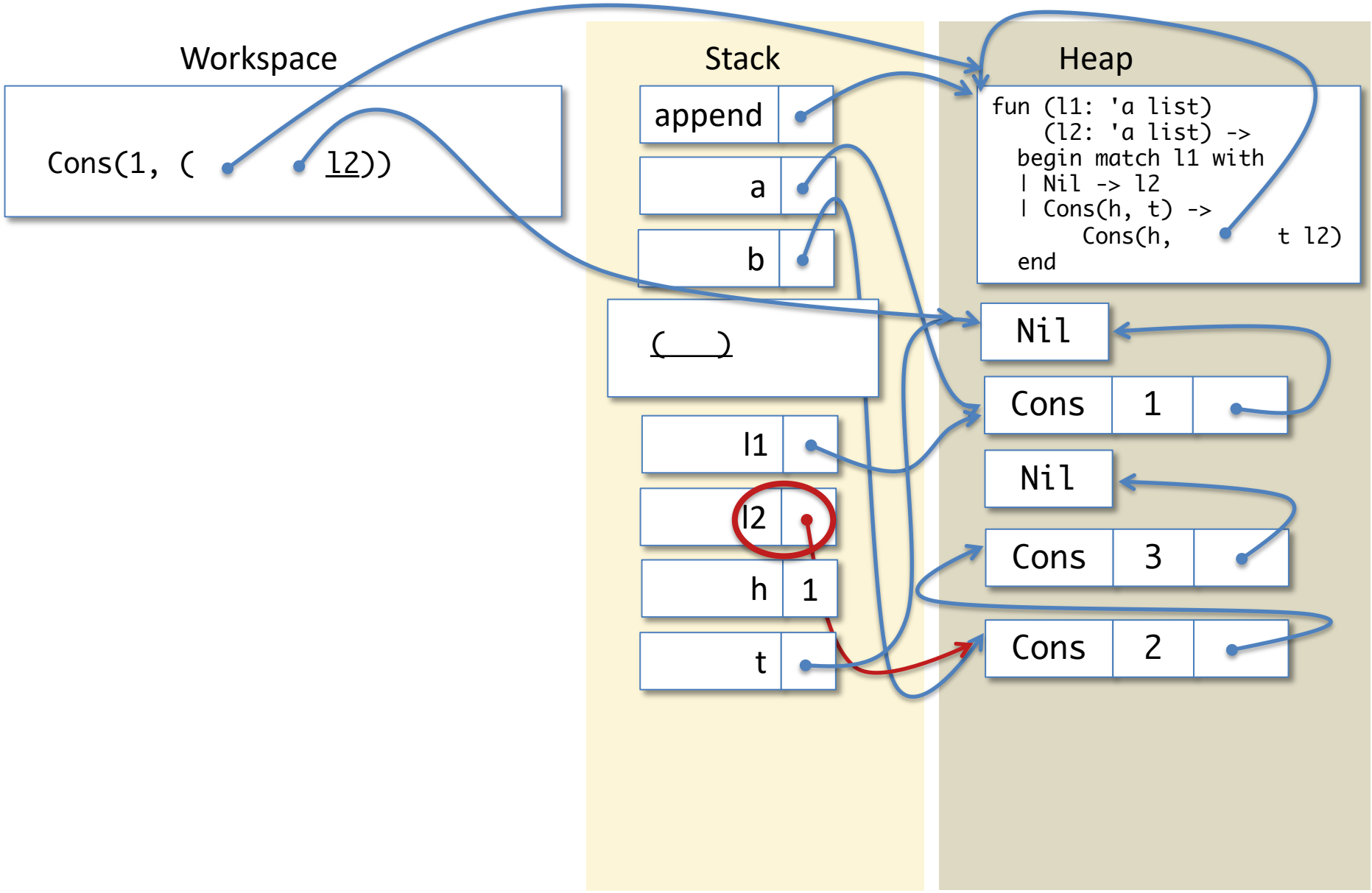
Lookup ‘t’


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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

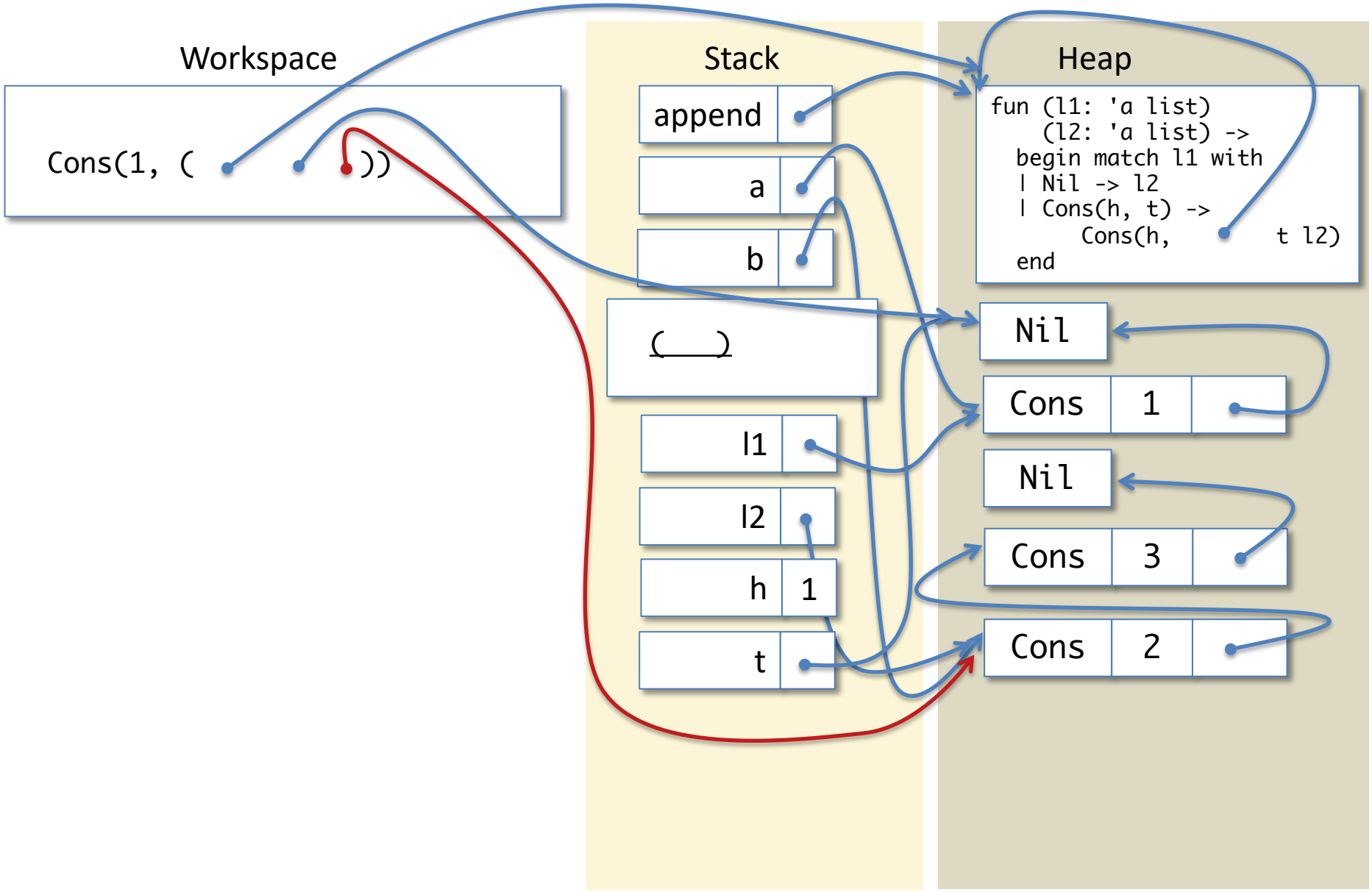
Lookup ‘t’


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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

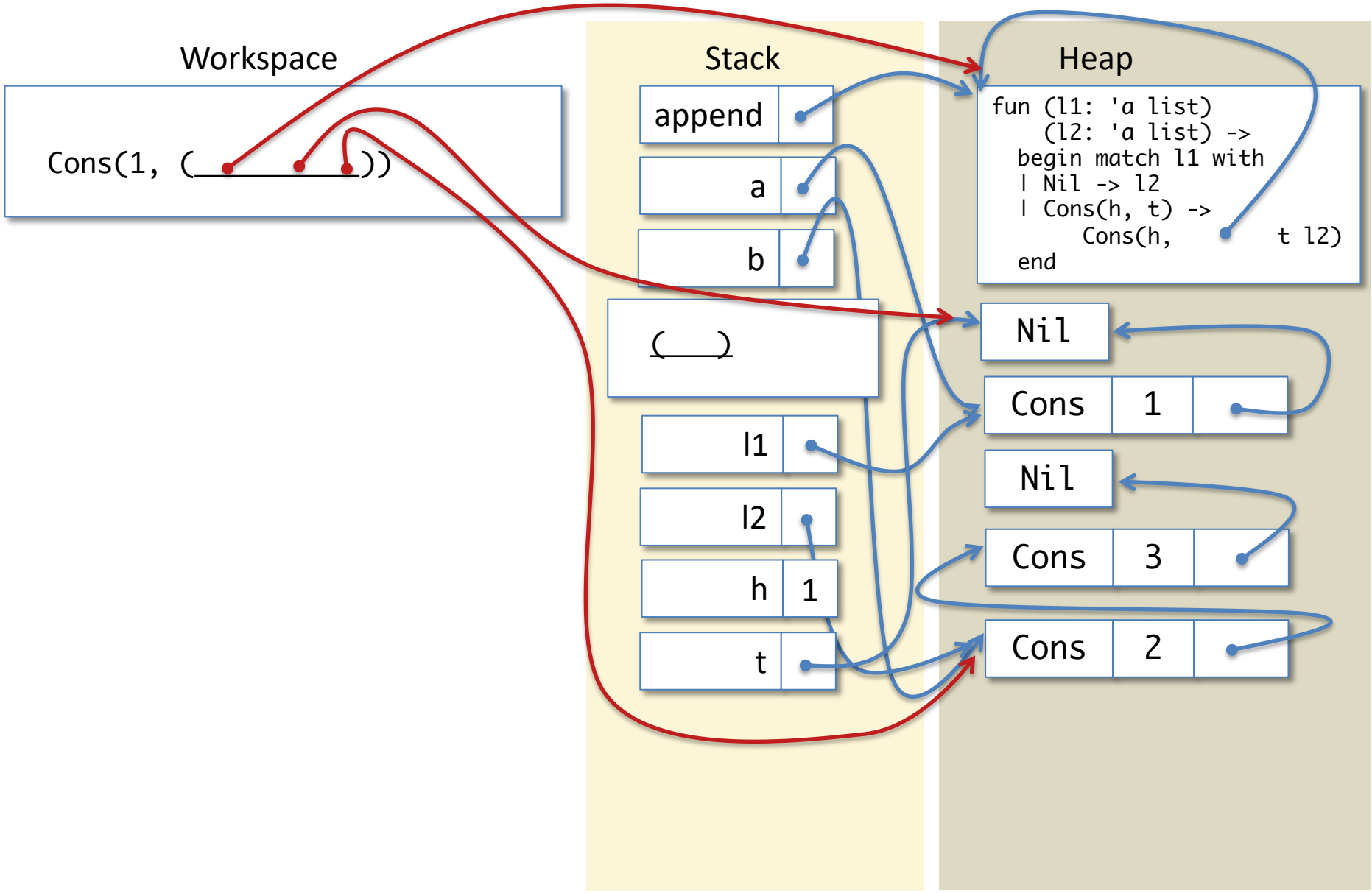
Lookup ‘l2’


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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

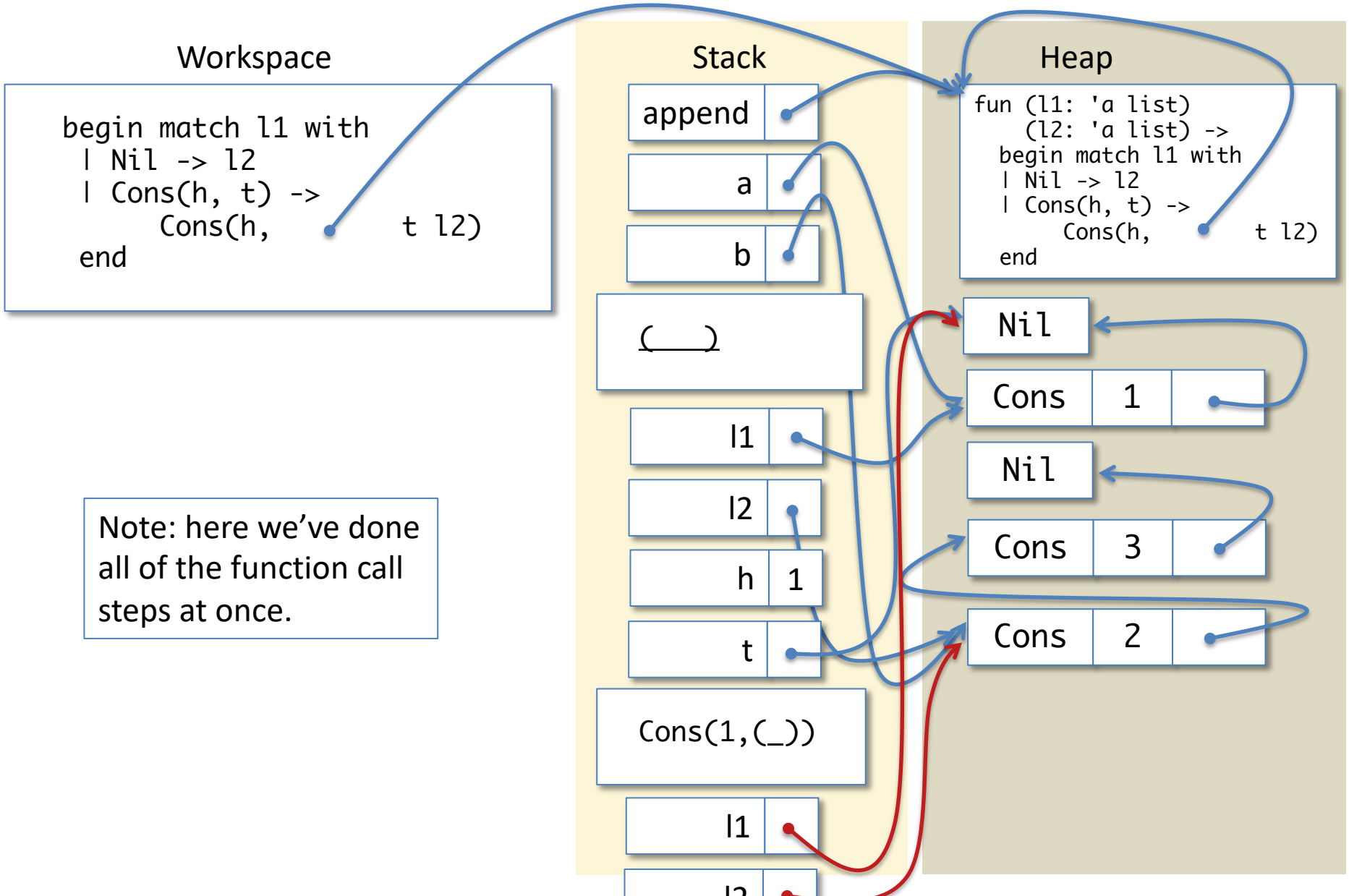
Lookup ‘l2’


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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

Do the Function Call

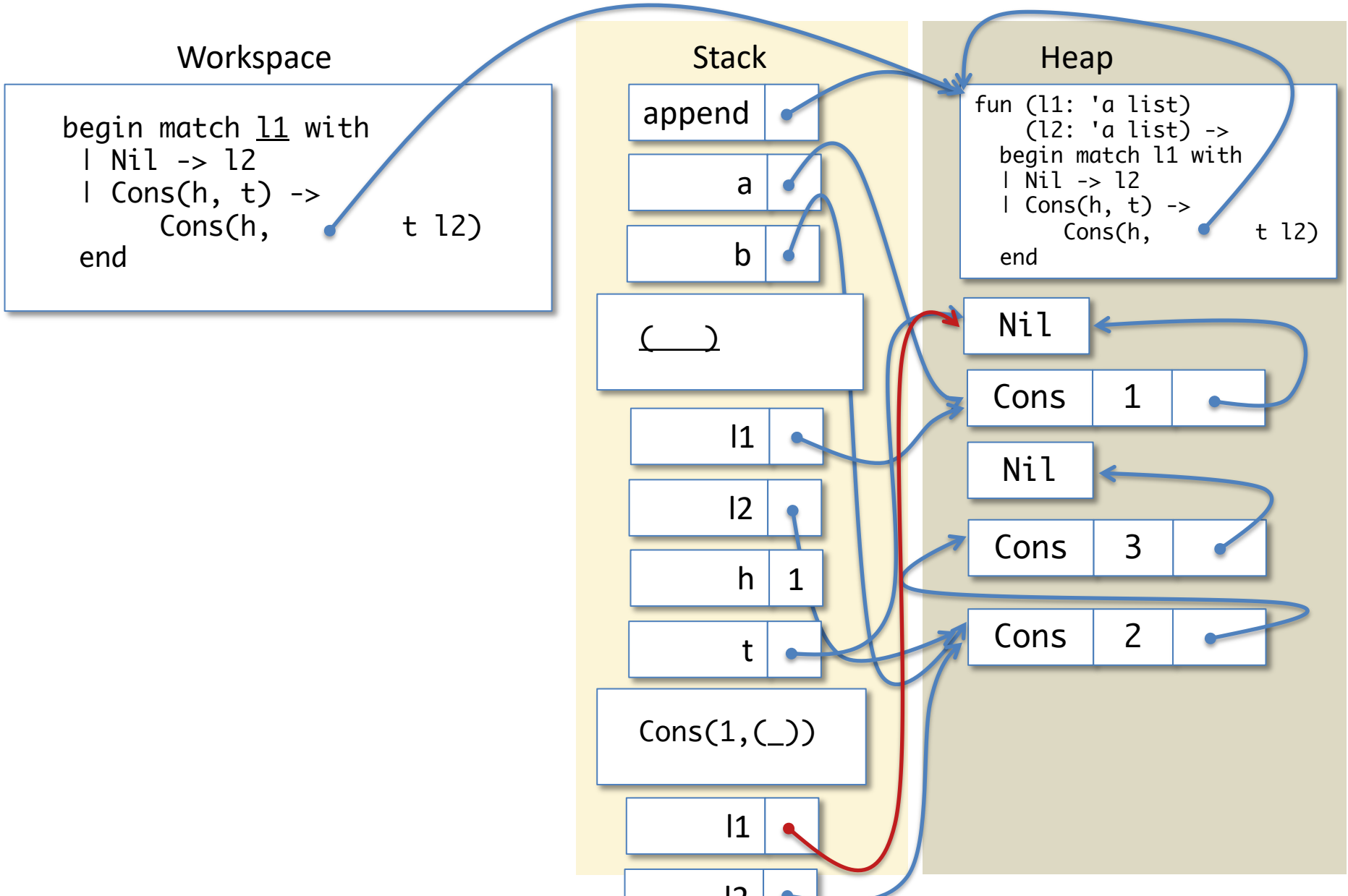


Save the Workspace; Push l1, l2

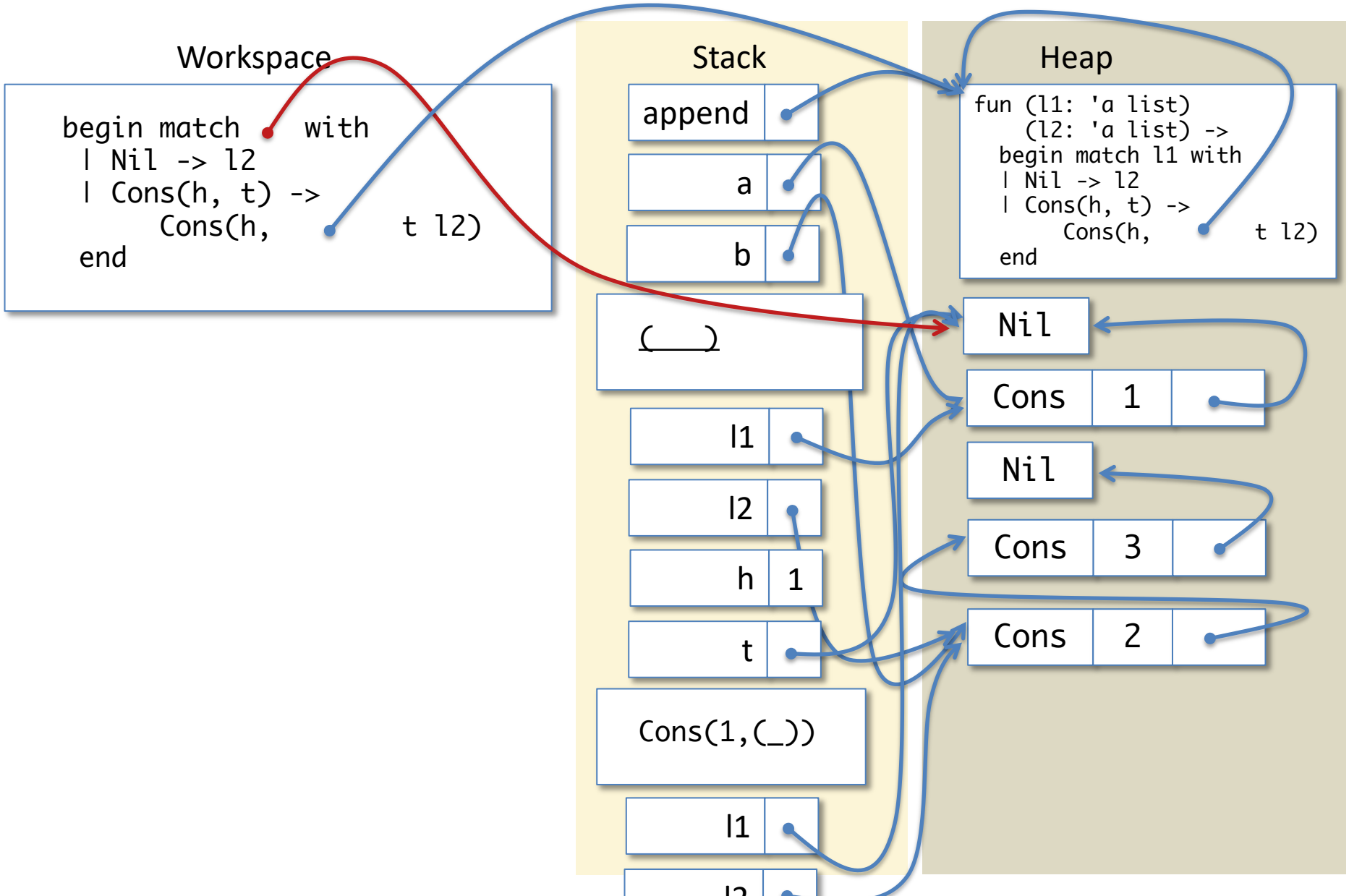


Note: here we've done all of the function call steps at once.

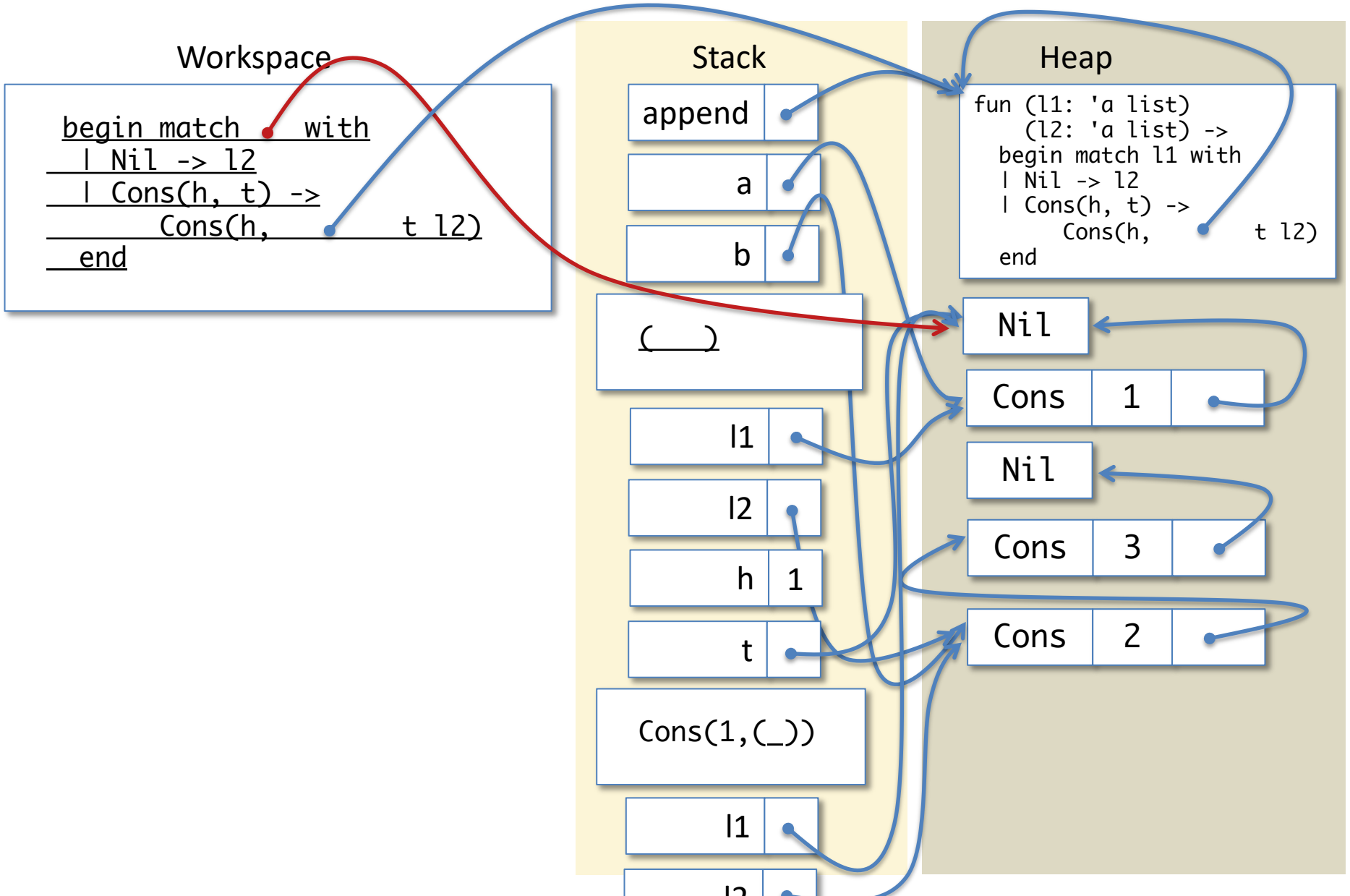
Lookup ‘l1’


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

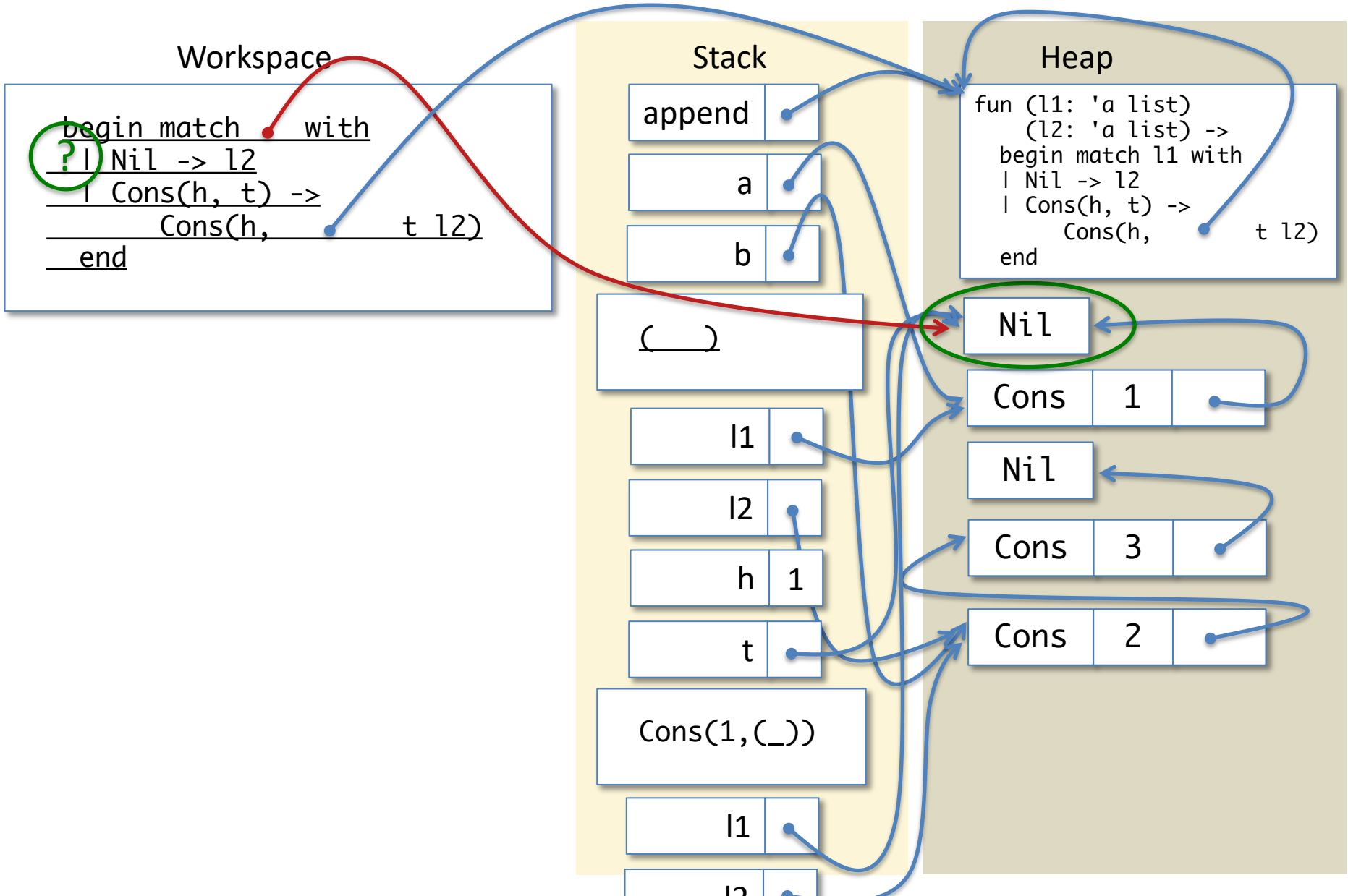
Lookup ‘l1’


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

Match Expression



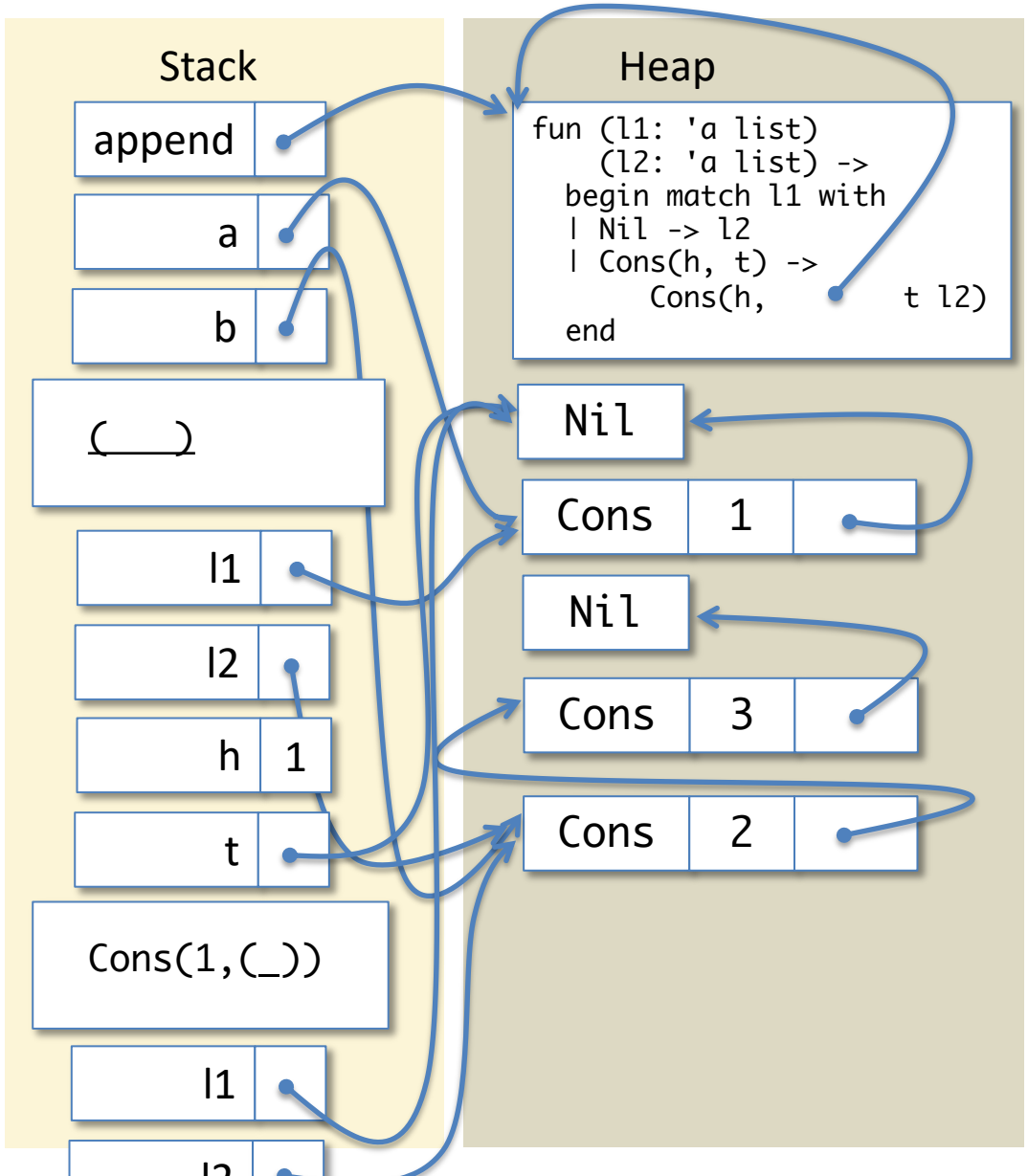
The Nil Case Matches



Simplify the Branch (nothing to push)

Workspace

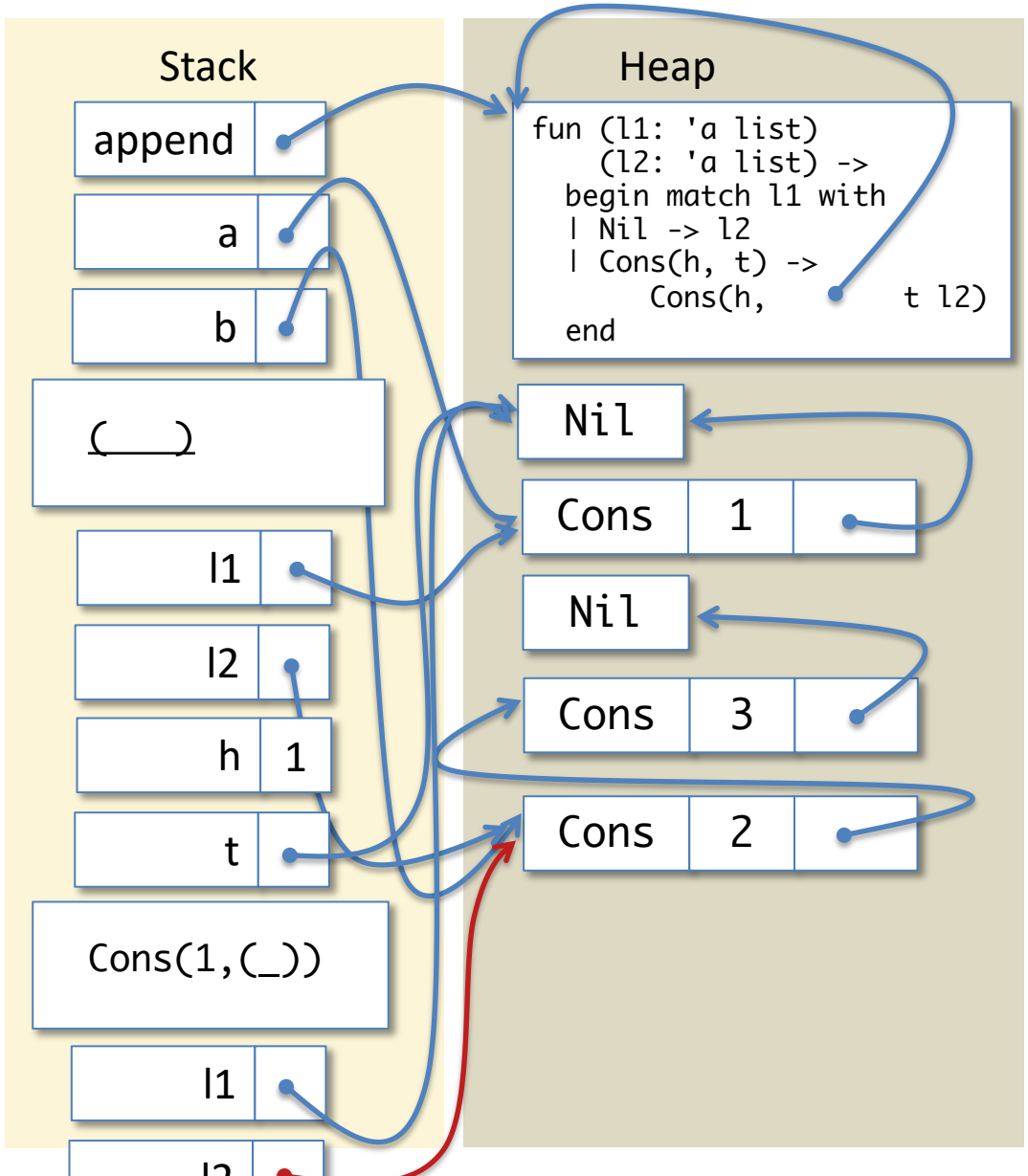
```
l2
```



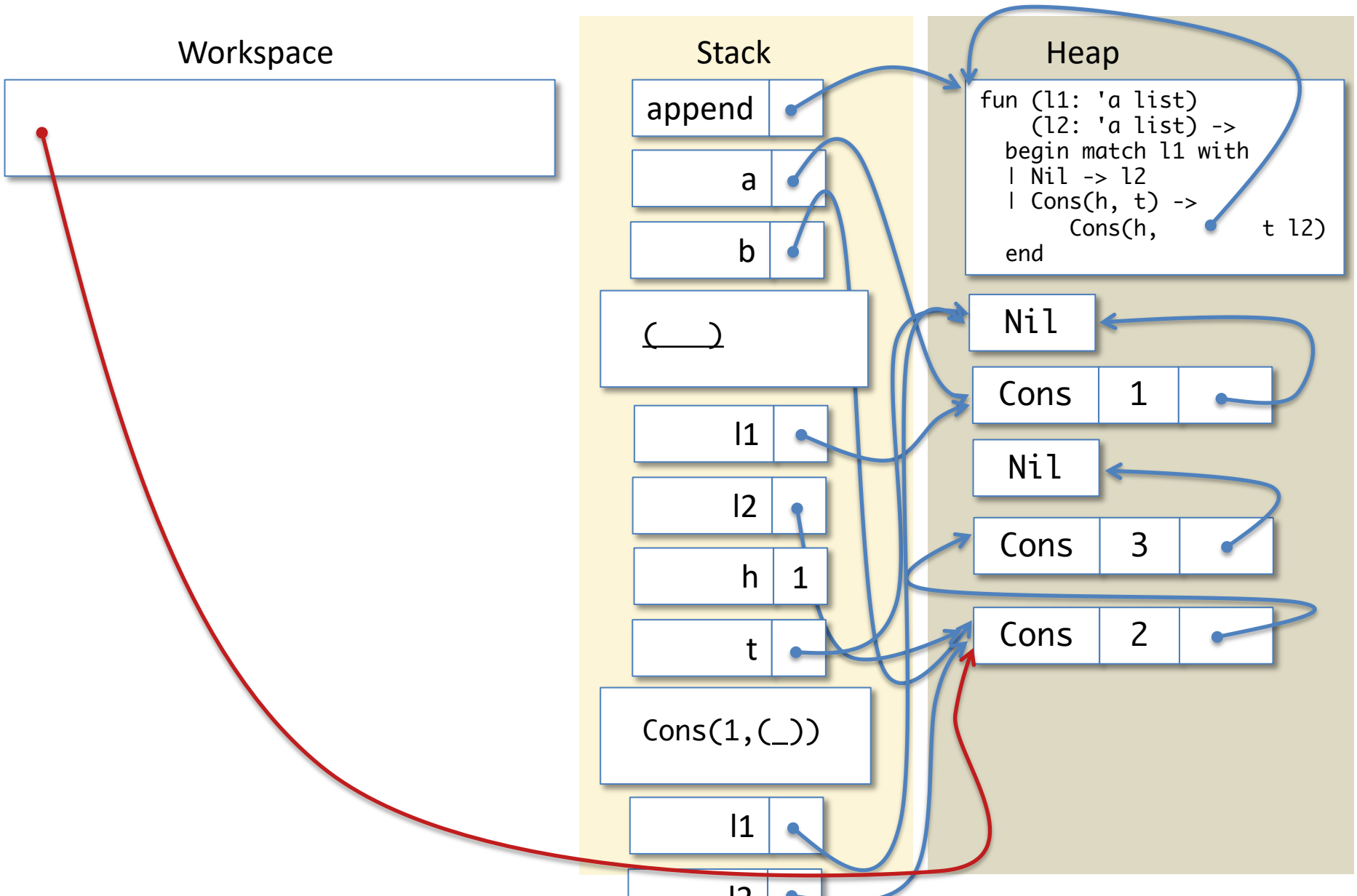
Lookup ‘l2’

Workspace

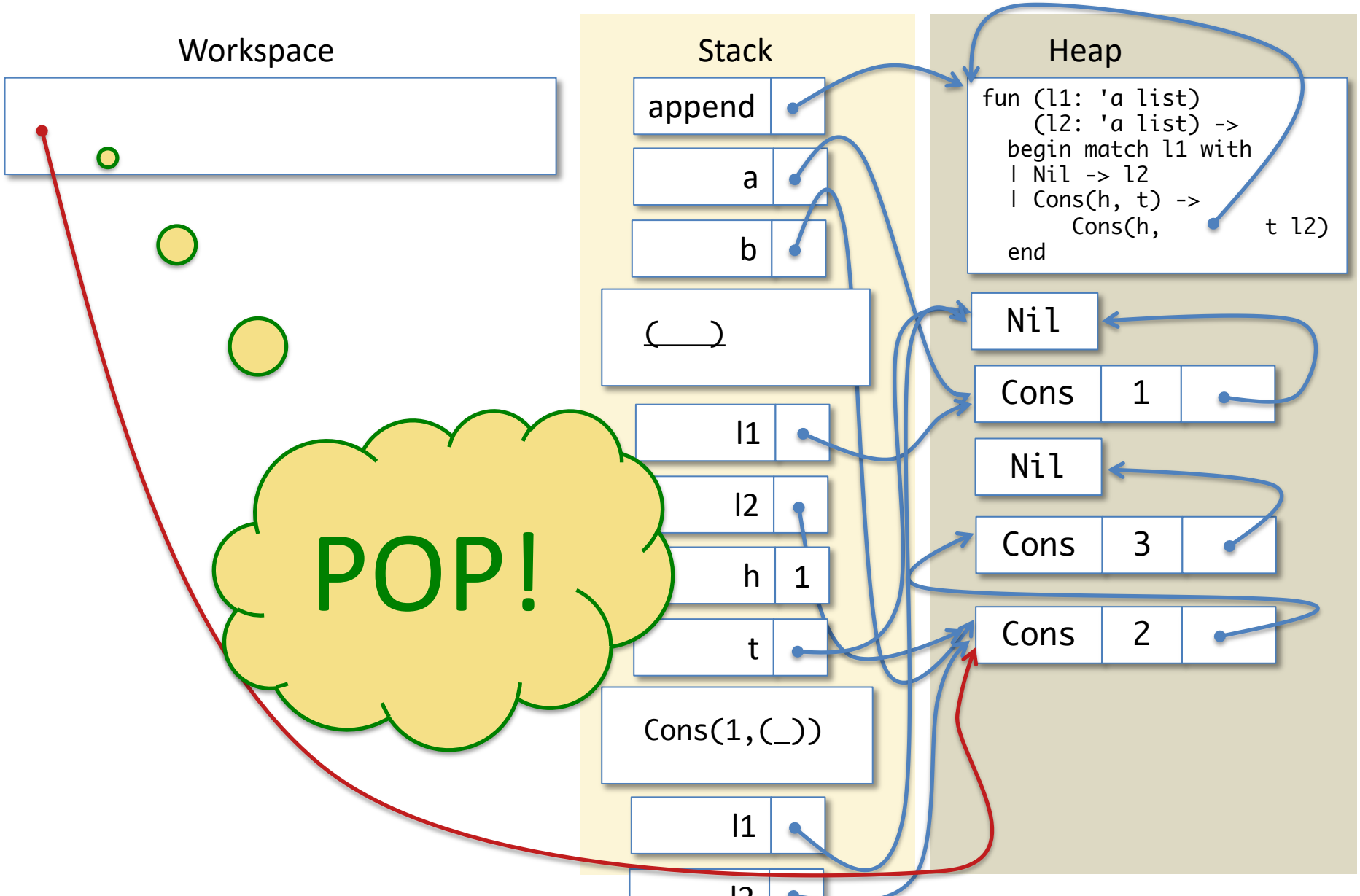
l2



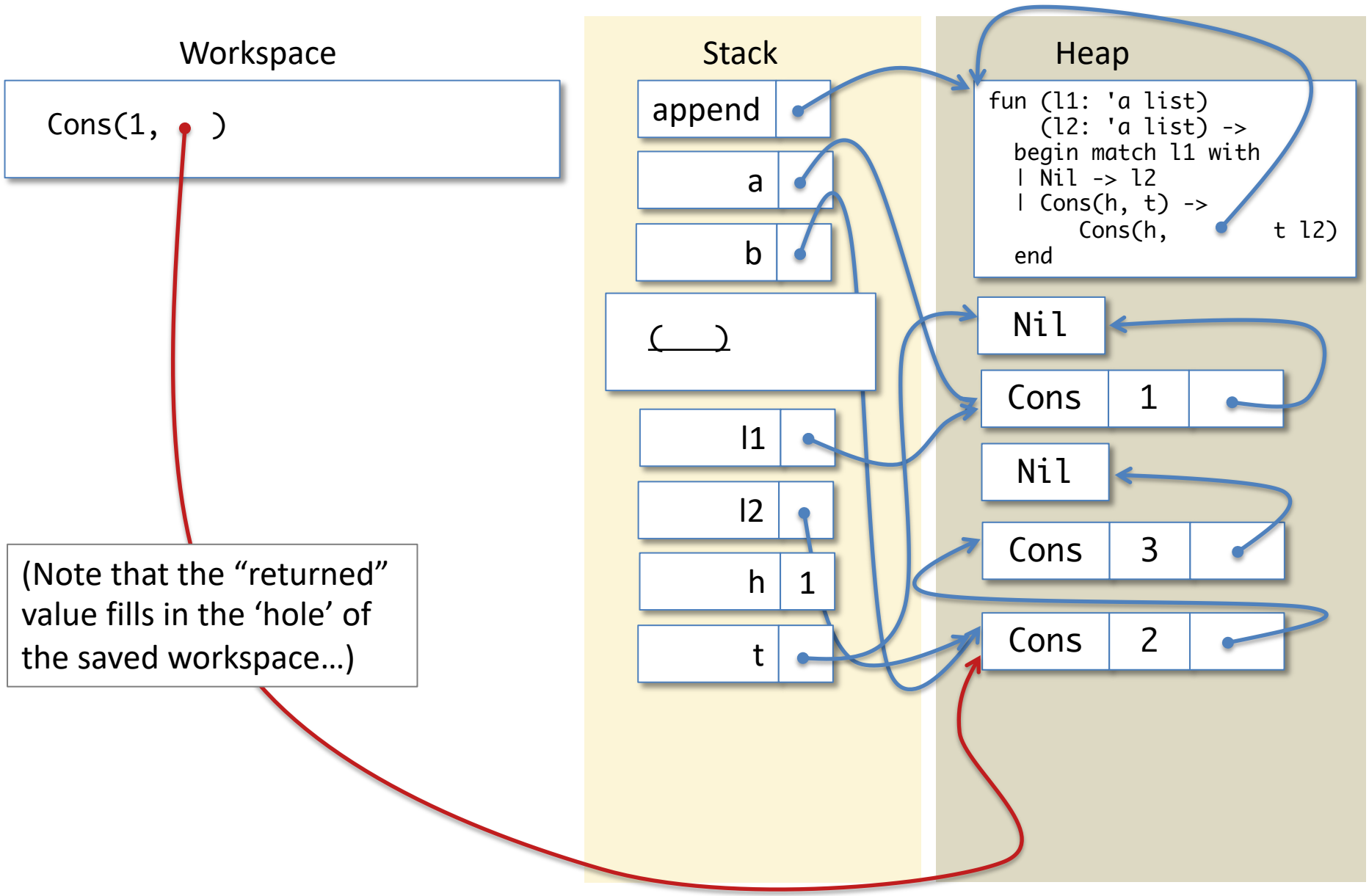
Lookup ‘l2’


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

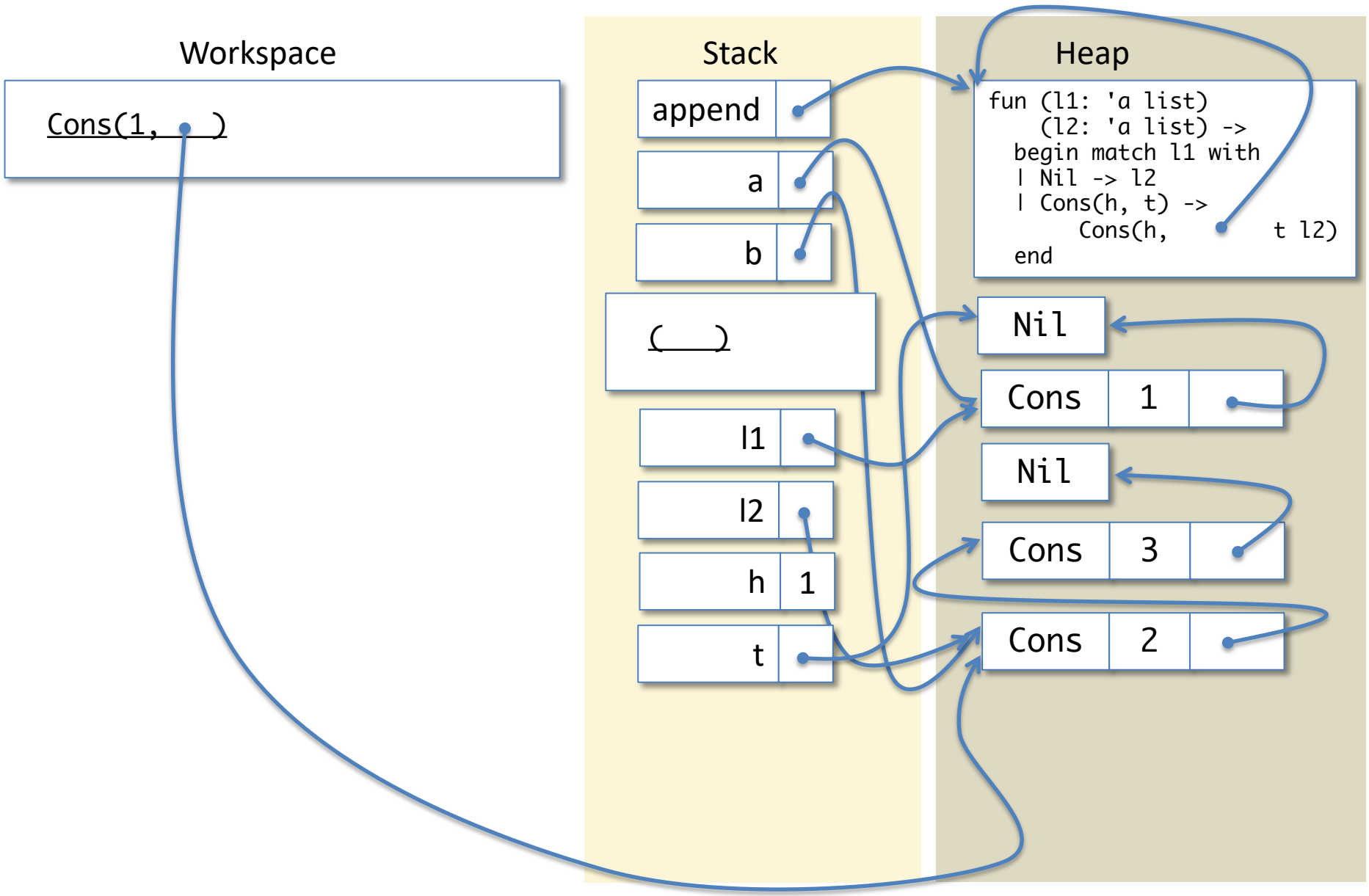
Done! Pop stack to last Workspace



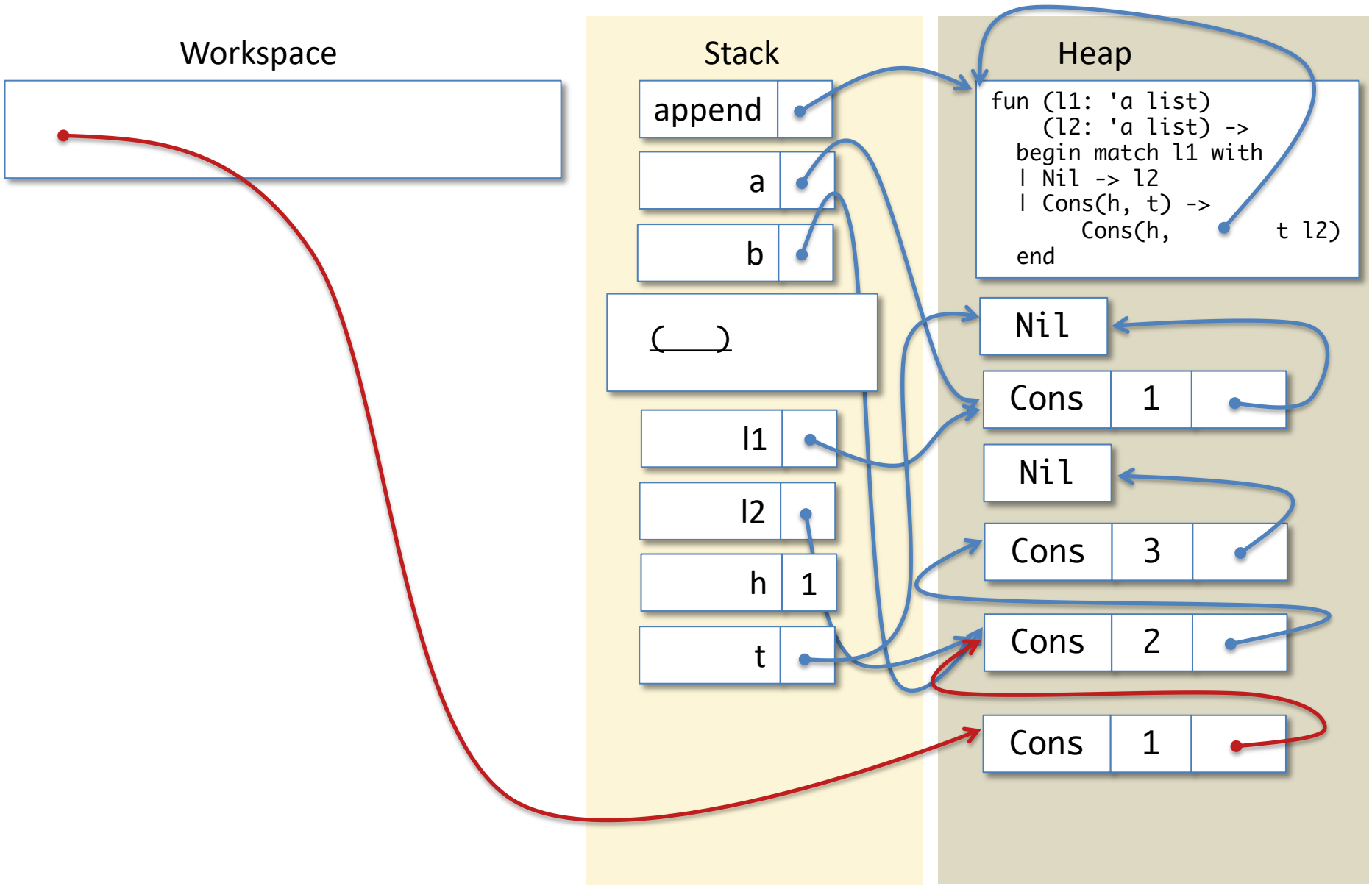
Done! Pop stack to last Workspace



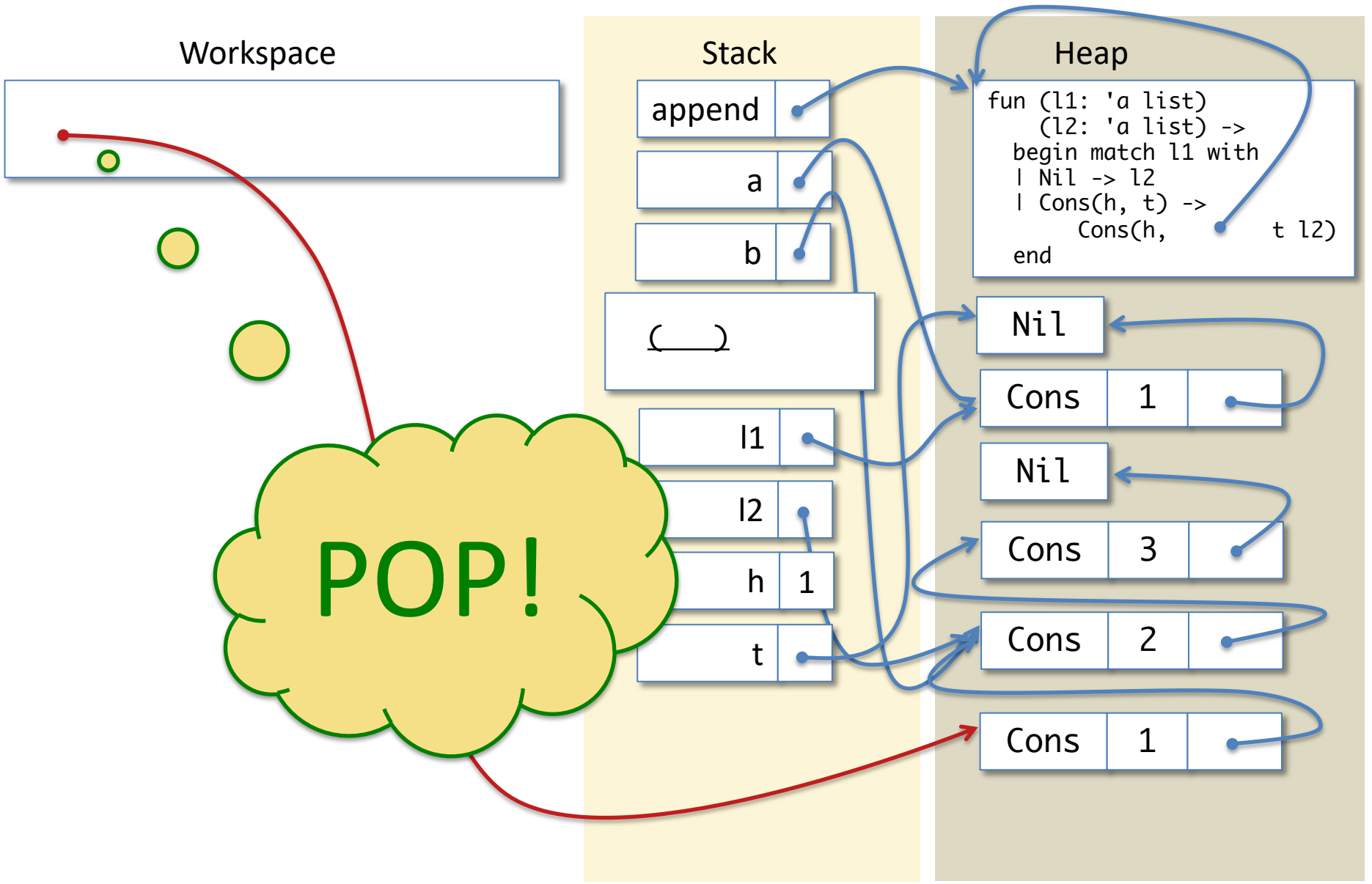
Allocate a Cons cell



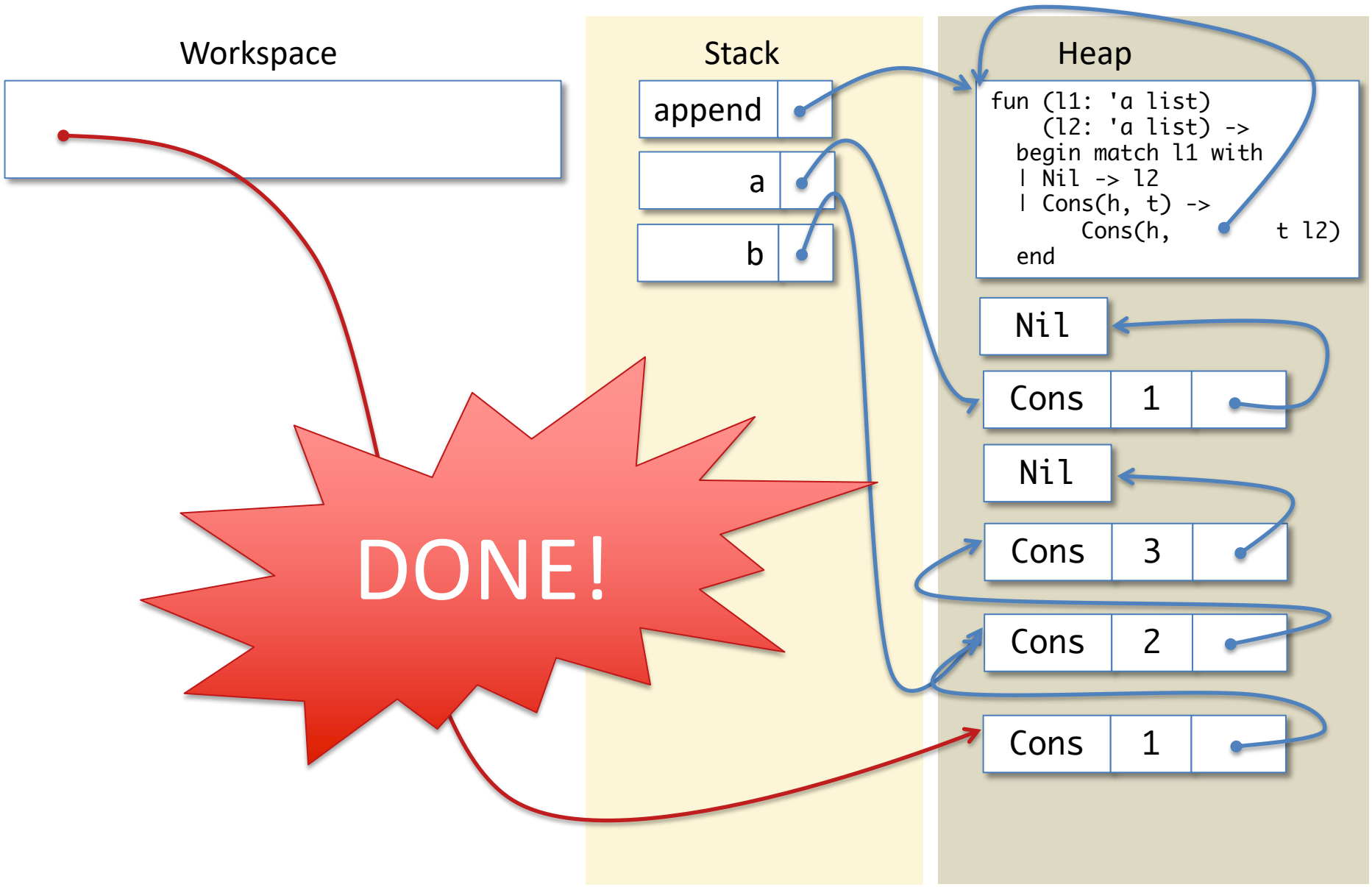
Allocate a Cons cell



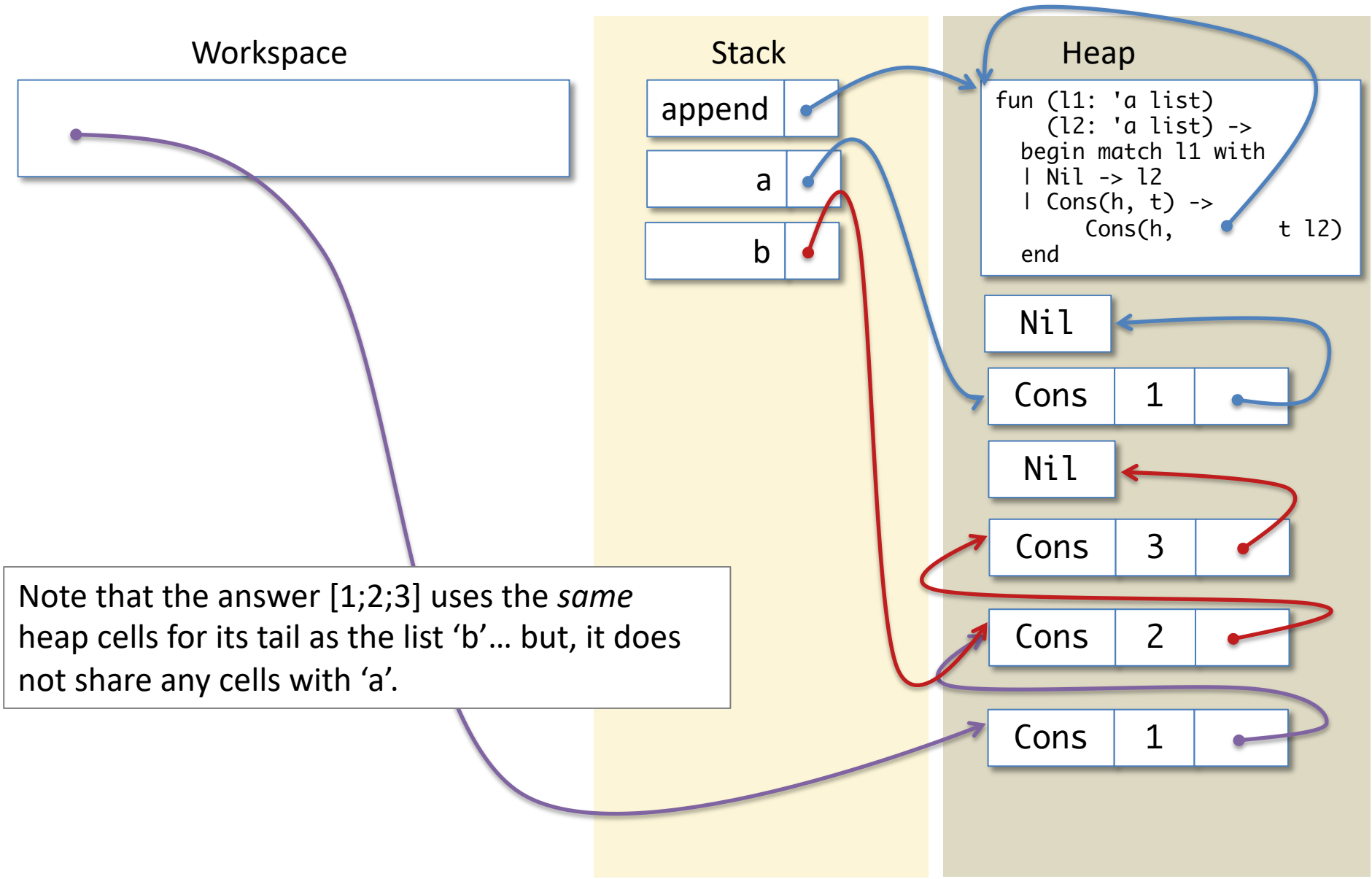
Done! Pop stack to last Workspace



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, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2
Cons 1
DONE!

Done! (PHEW!)


Workspace
Stack
append
a
b
Heap
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2
Cons 1
Note that the answer [1;2;3] uses the *same* heap cells for its tail as the list 'b'… but, it does not share any cells with 'a'.