Programming Languages and Techniques (CIS120)

Lecture 14

February 19, 2014

Modeling Recursive Functions
Mutable Queues

Announcements

- Midterm 1 will be in class on Friday
  - ROOMS:
    - Towne 100 (here) last names: A – L
    - DRLB A1 last names: M – Z
  - TIME: 11:00a.m. sharp, 50 mins
  - Can bring one single-sided, handwritten (by you!) sheet of notes (8.5 x 11) to the exam
  - Study: old exams, lecture notes, homeworks 1-4
- Labs today and tomorrow are review
- Review session 7-9PM tonight in Levine 101

Modeling Recursive Functions

Substitution Model

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
```

Substitution model

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

```
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
```

Modeling Recursive Functions

Abstract Stack Machine

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

```
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

Copy to the Heap, Replace w/Reference

Workspace

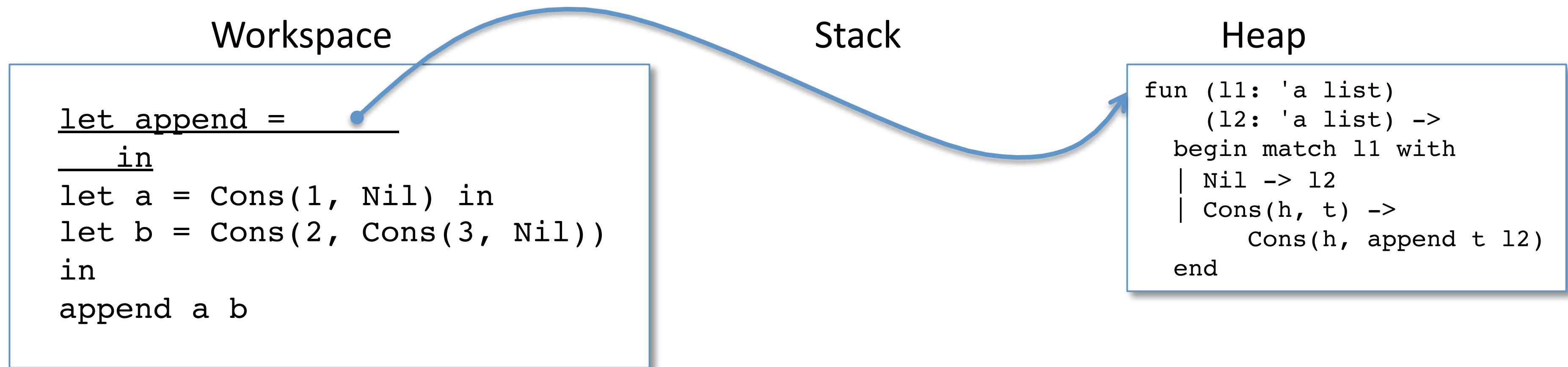
```
let append =
   in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

Heap

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

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

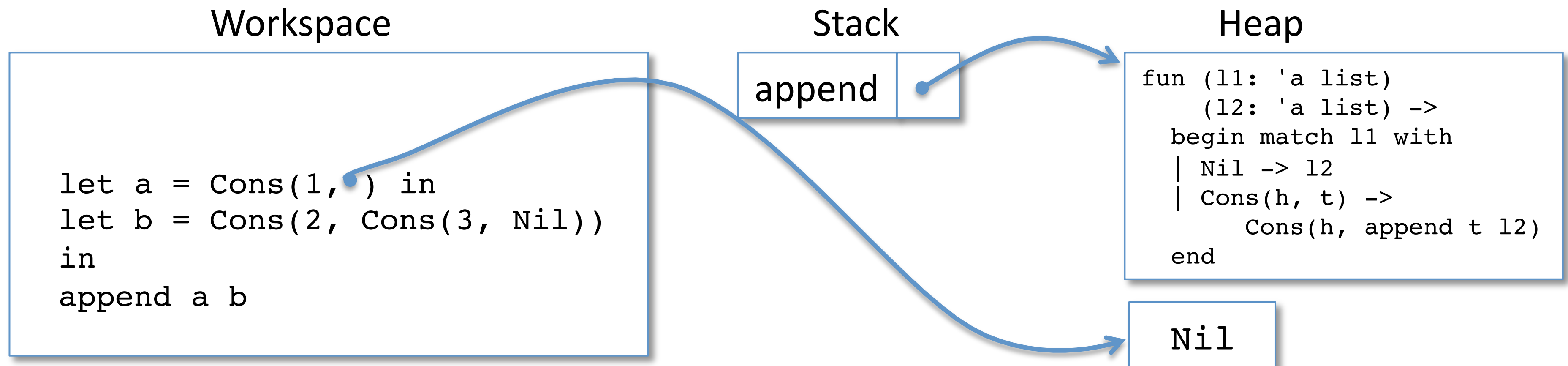
```
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

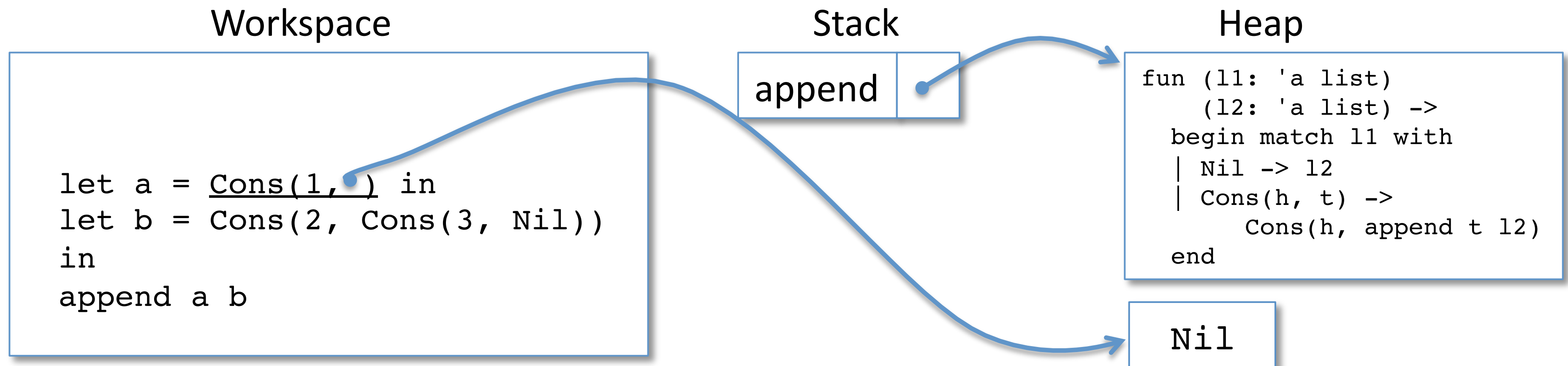
Heap

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

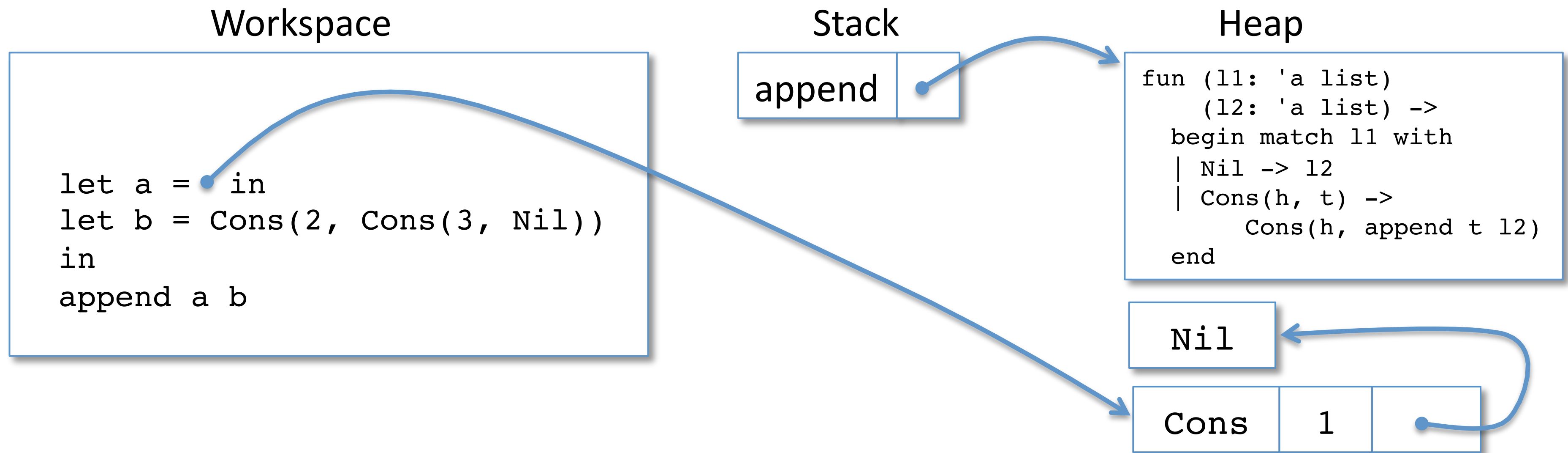
Allocate a Nil cell



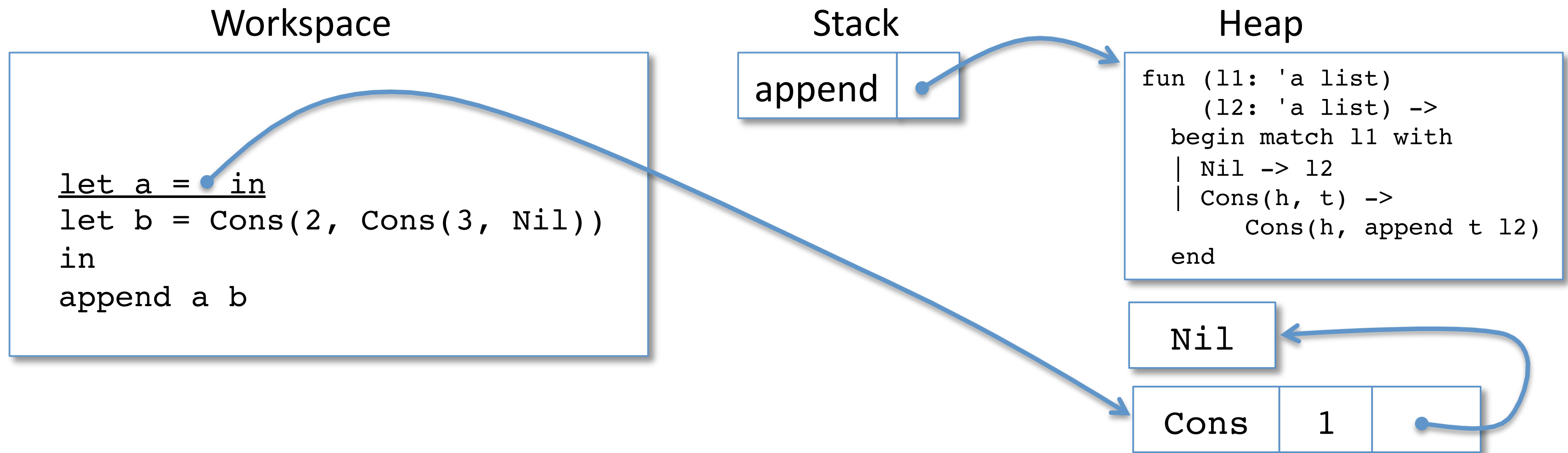
Allocate a Cons cell



Allocate a Cons cell



Let Expression

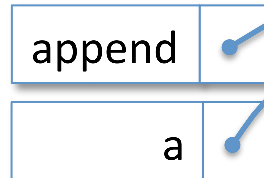


Create a Stack Binding

Workspace

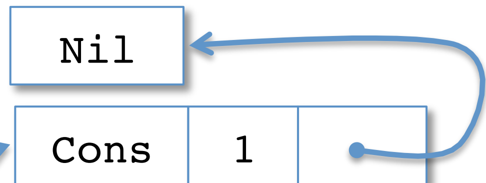
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack



Heap

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

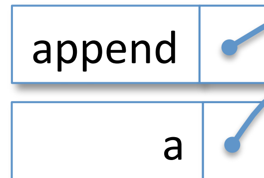


Allocate a Nil cell

Workspace

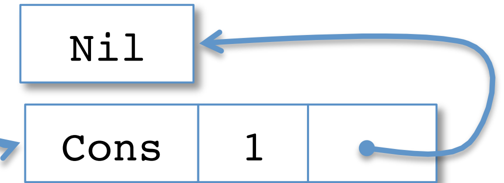
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

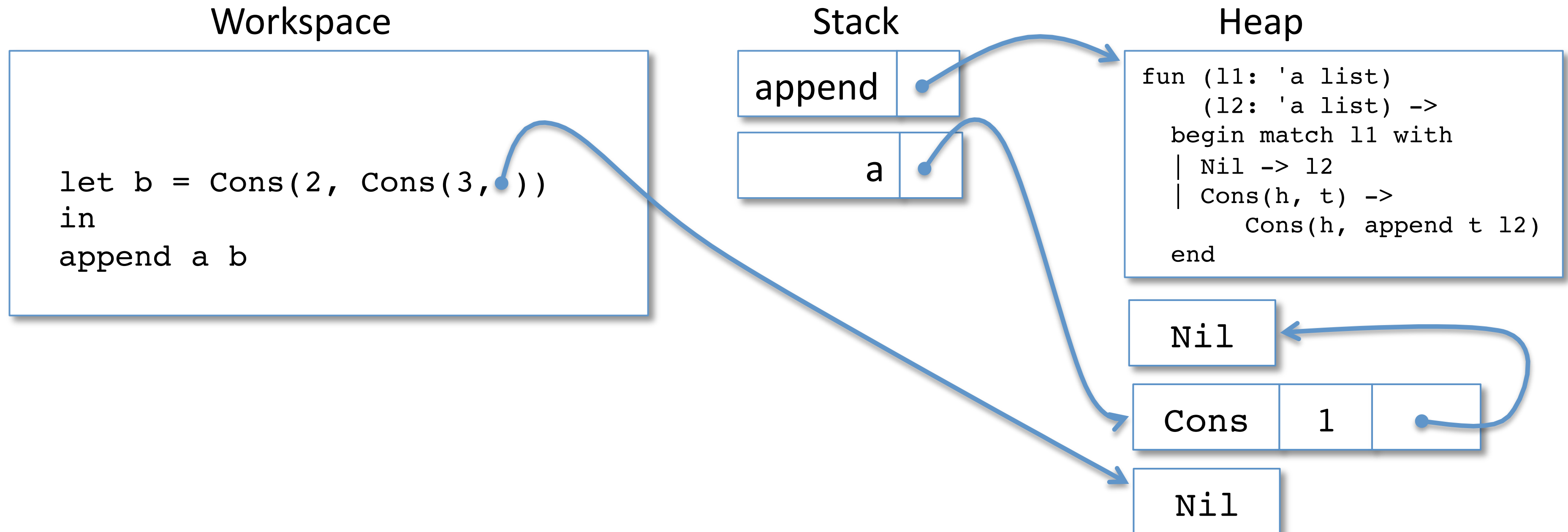


Heap

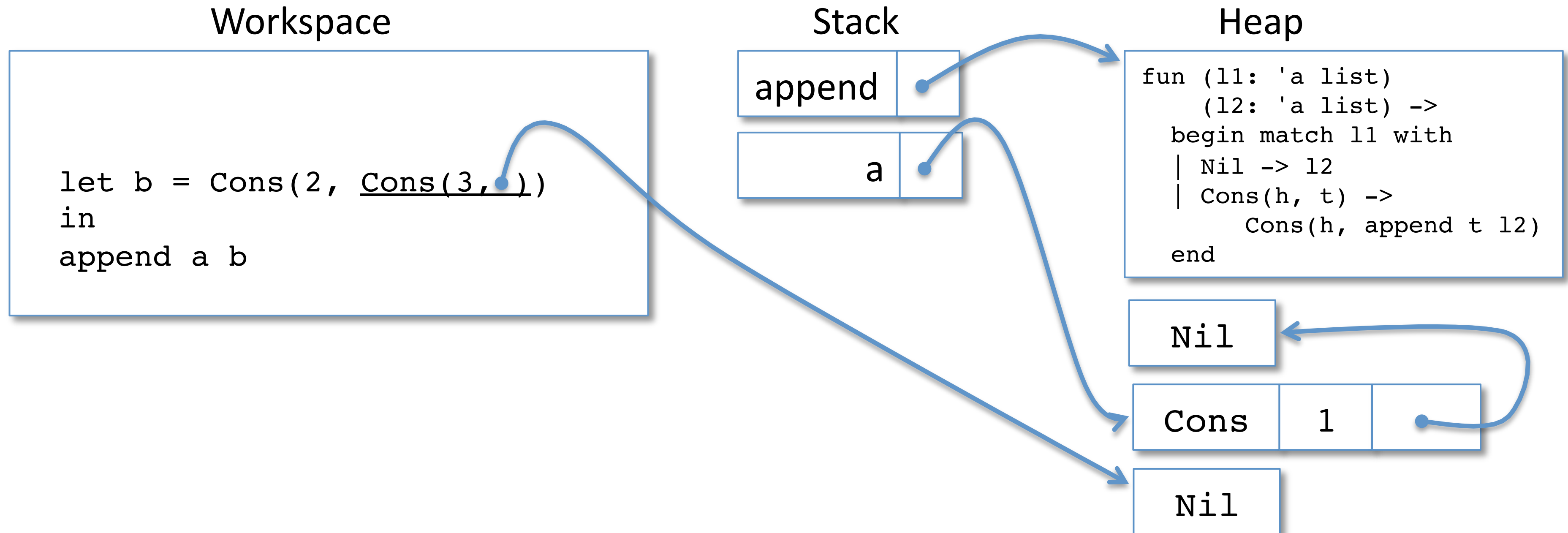
```
fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
        Cons(h, append t l2)
  end
```



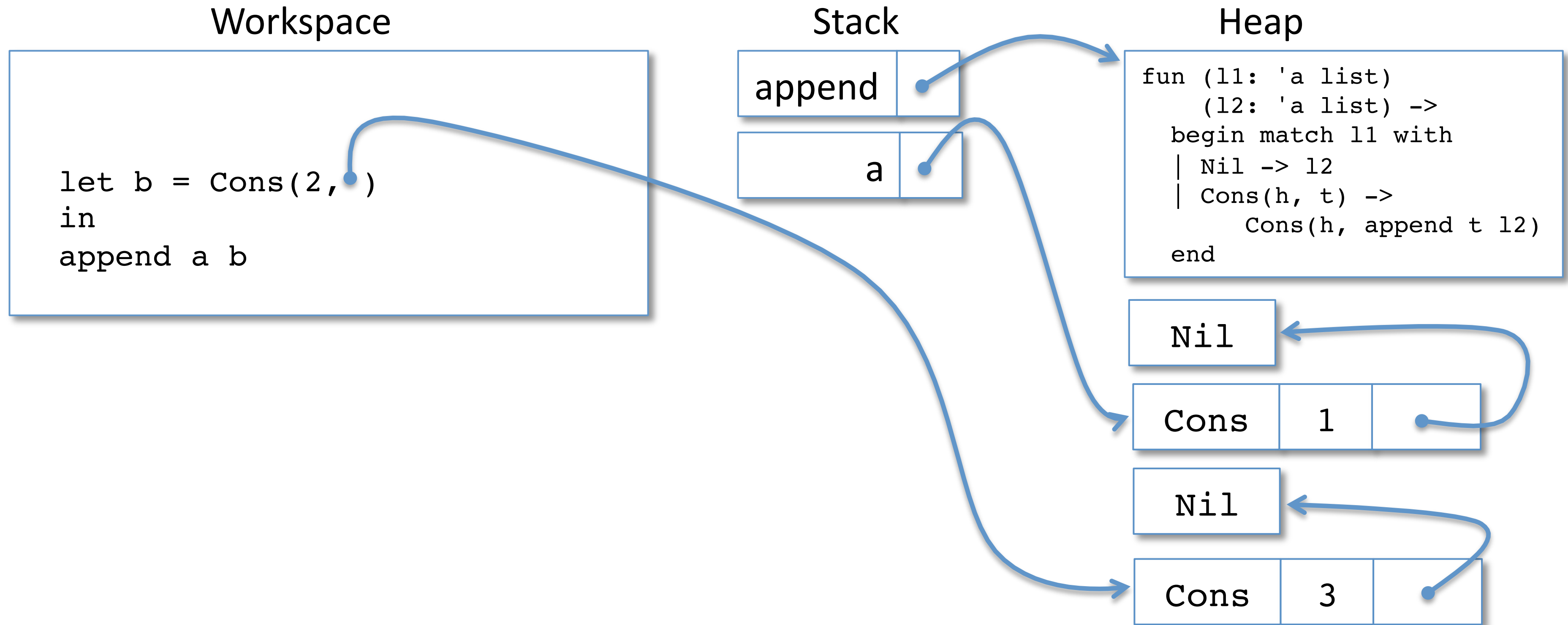
Allocate a Nil cell



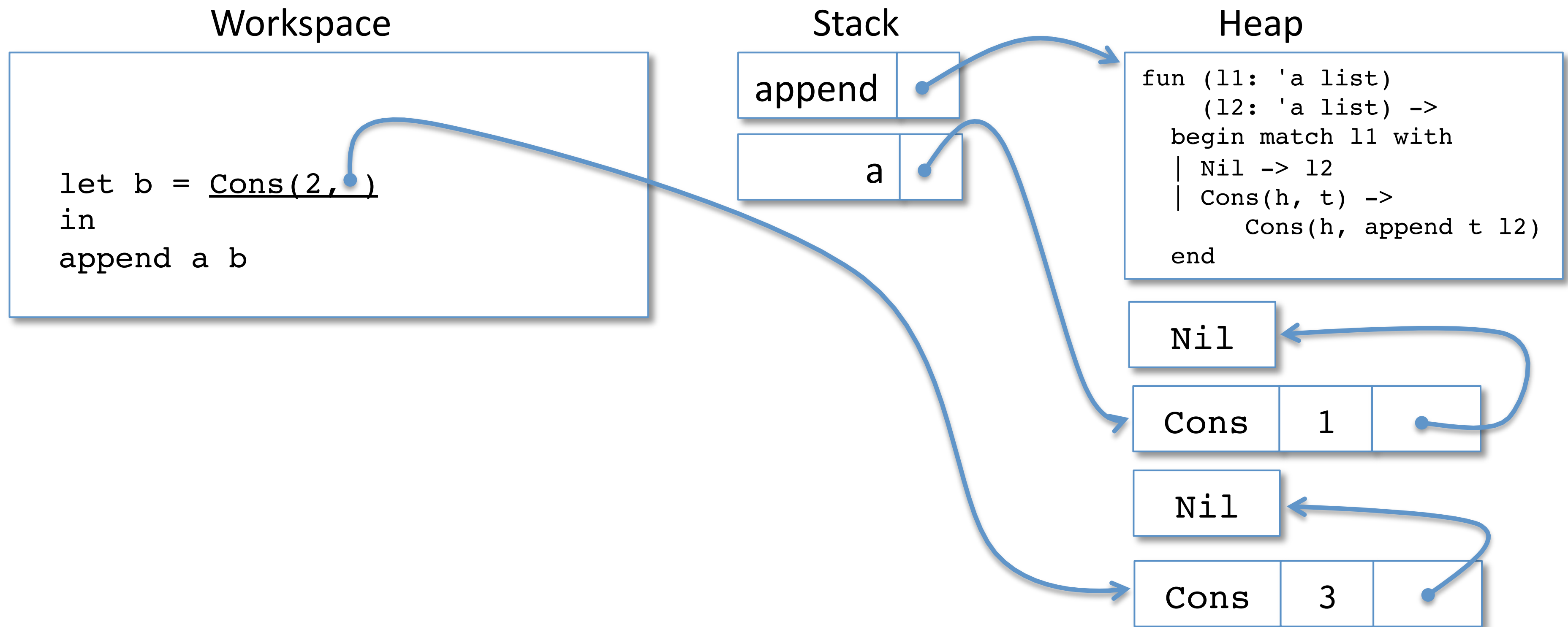
Allocate a Cons cell



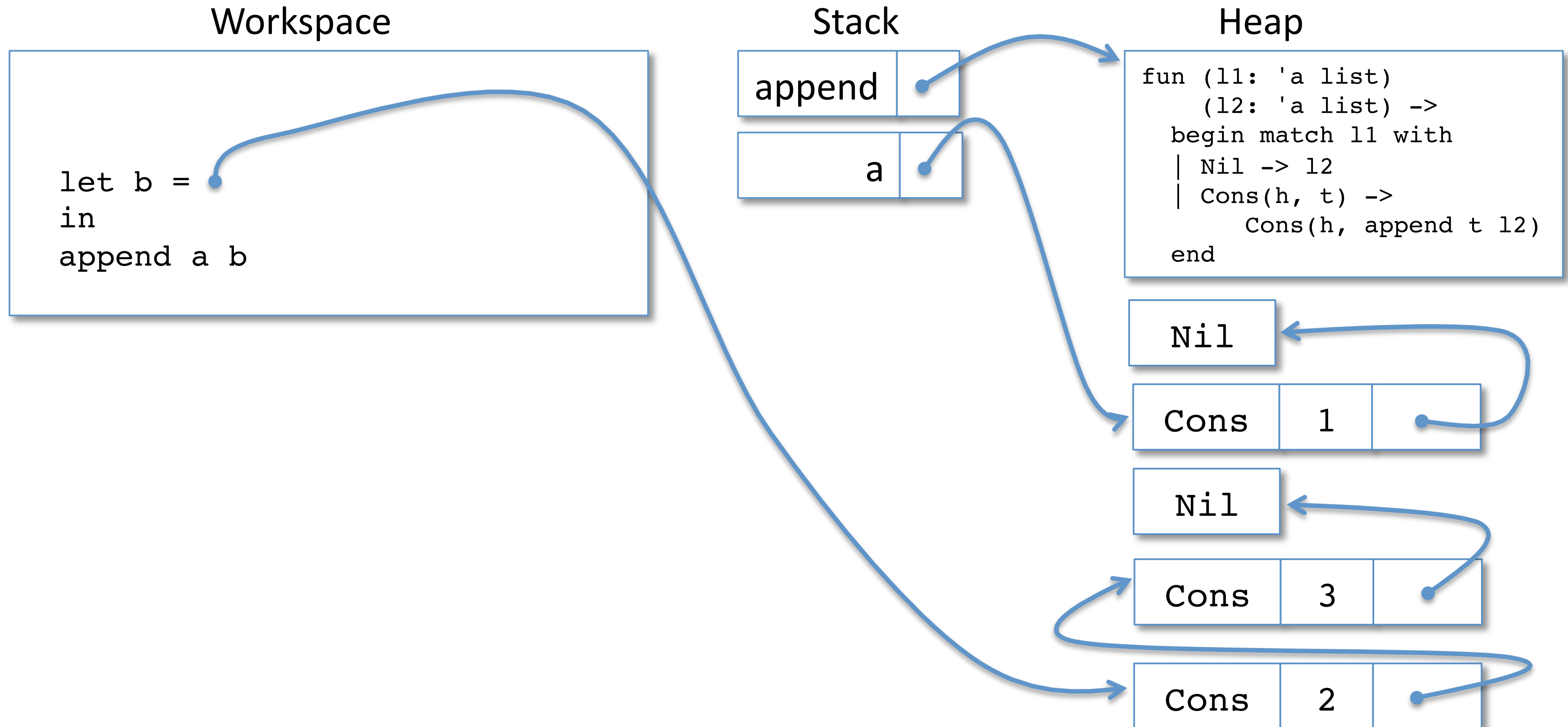
Allocate a Cons cell



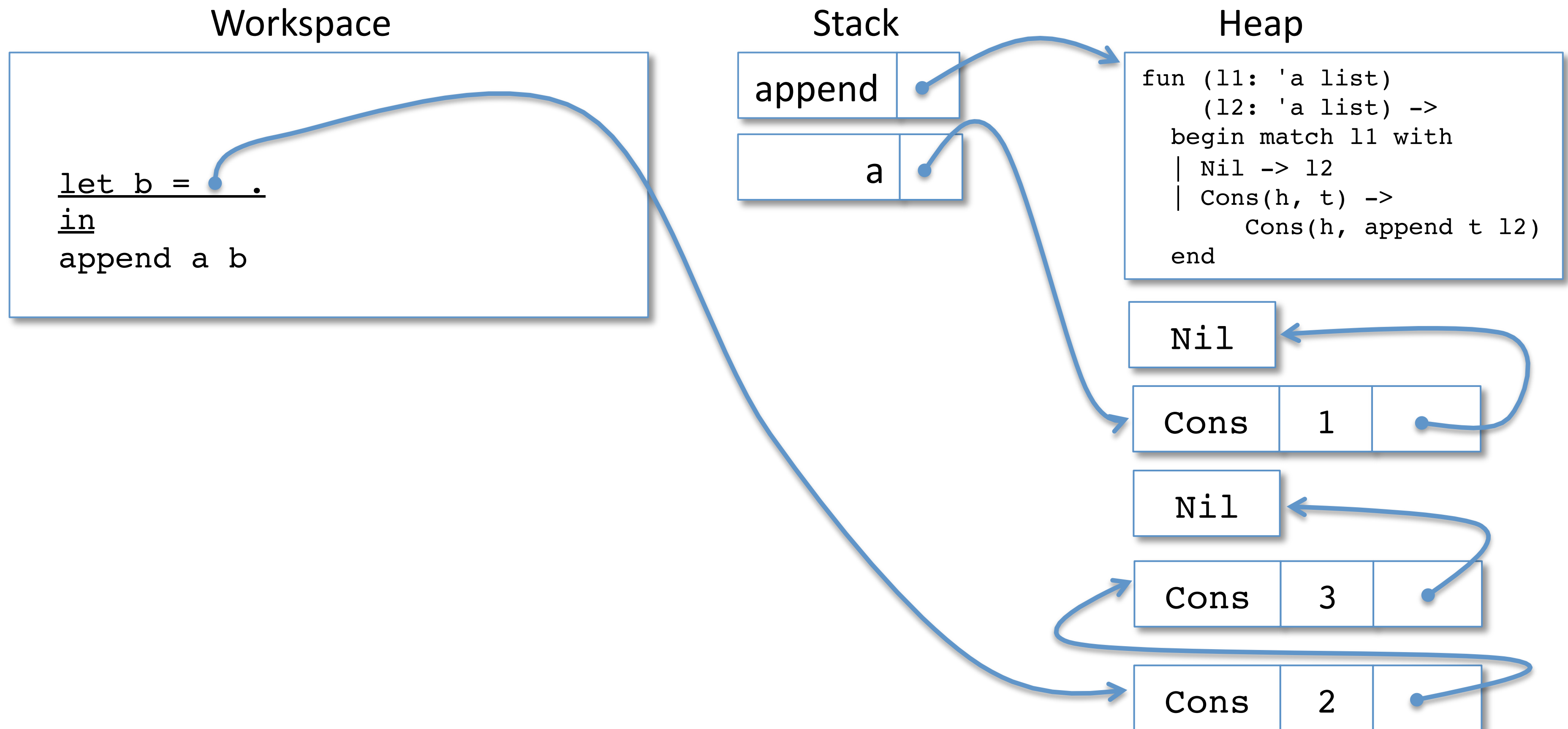
Allocate a Cons cell



Allocate a Cons cell



Let Expression

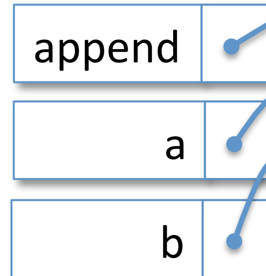


Create a Stack Binding

Workspace

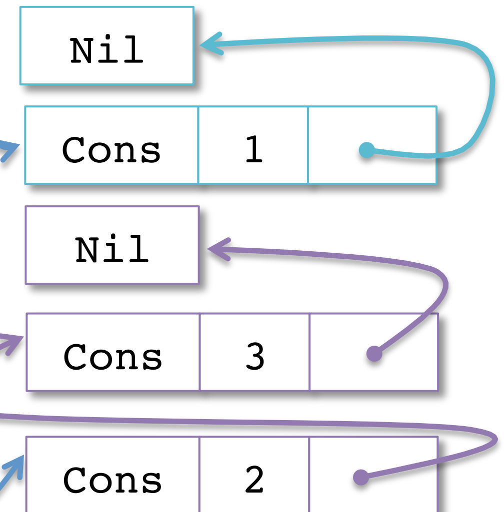
```
append a b
```

Stack



Heap

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

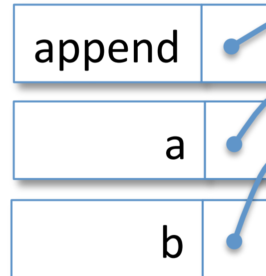


Lookup ‘append’

Workspace

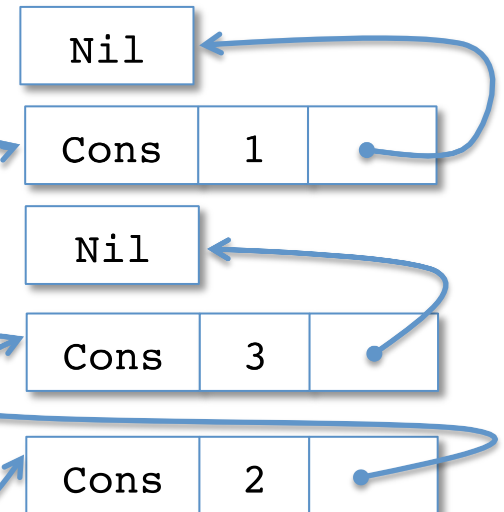
```
append a b
```

Stack

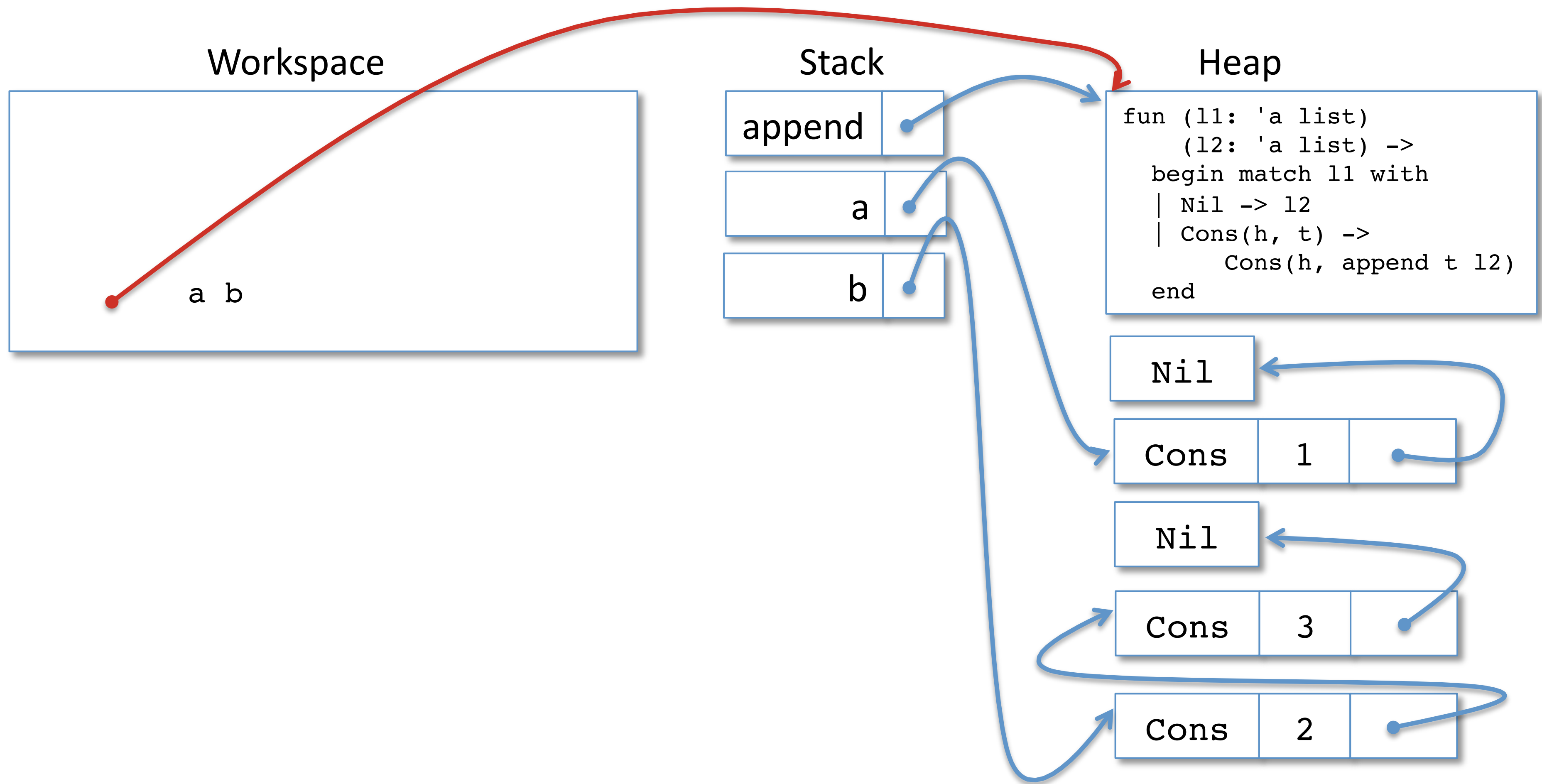


Heap

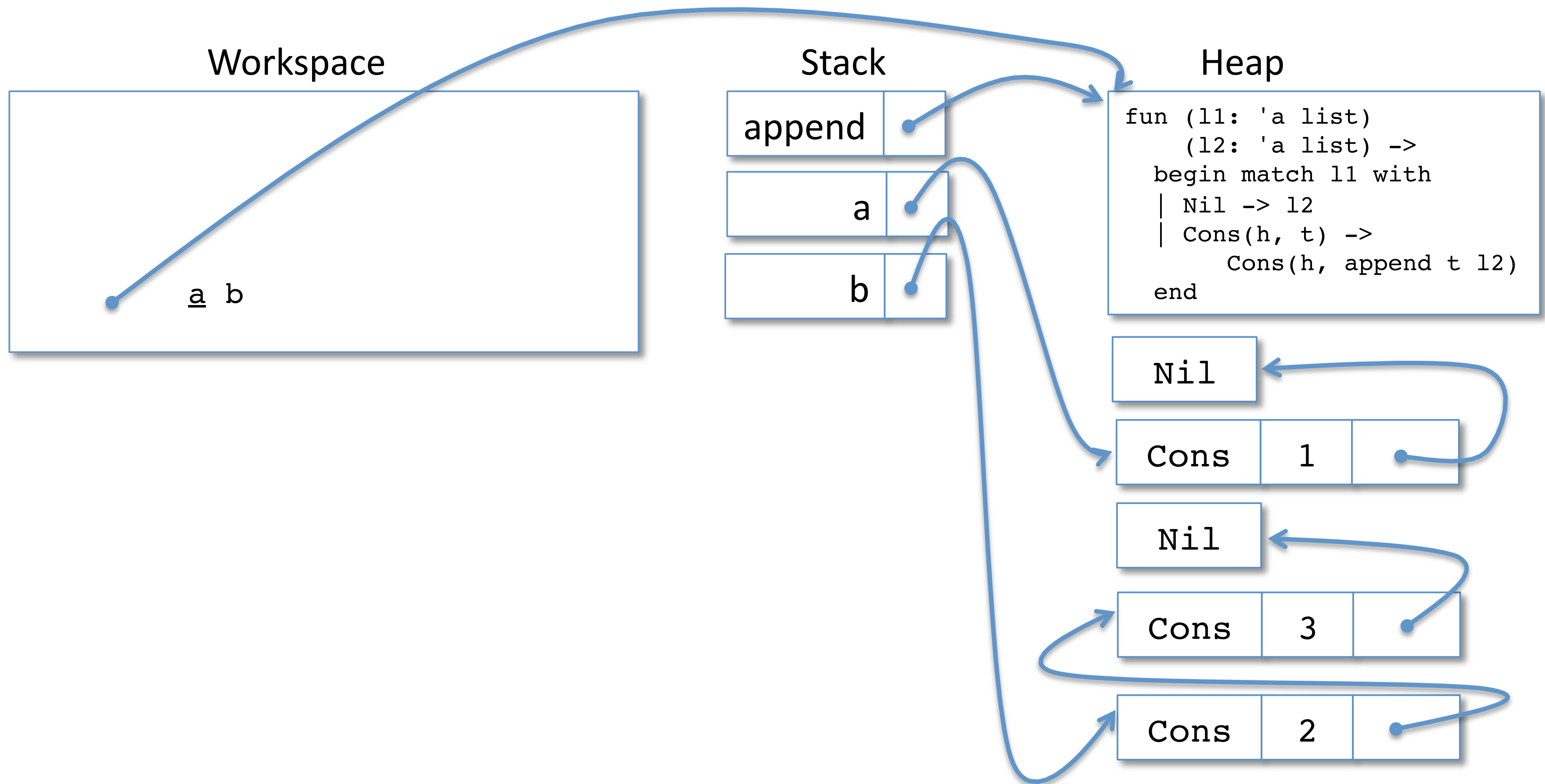
```
fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
   | Nil -> l2
   | Cons(h, t) ->
        Cons(h, append t l2)
  end
```



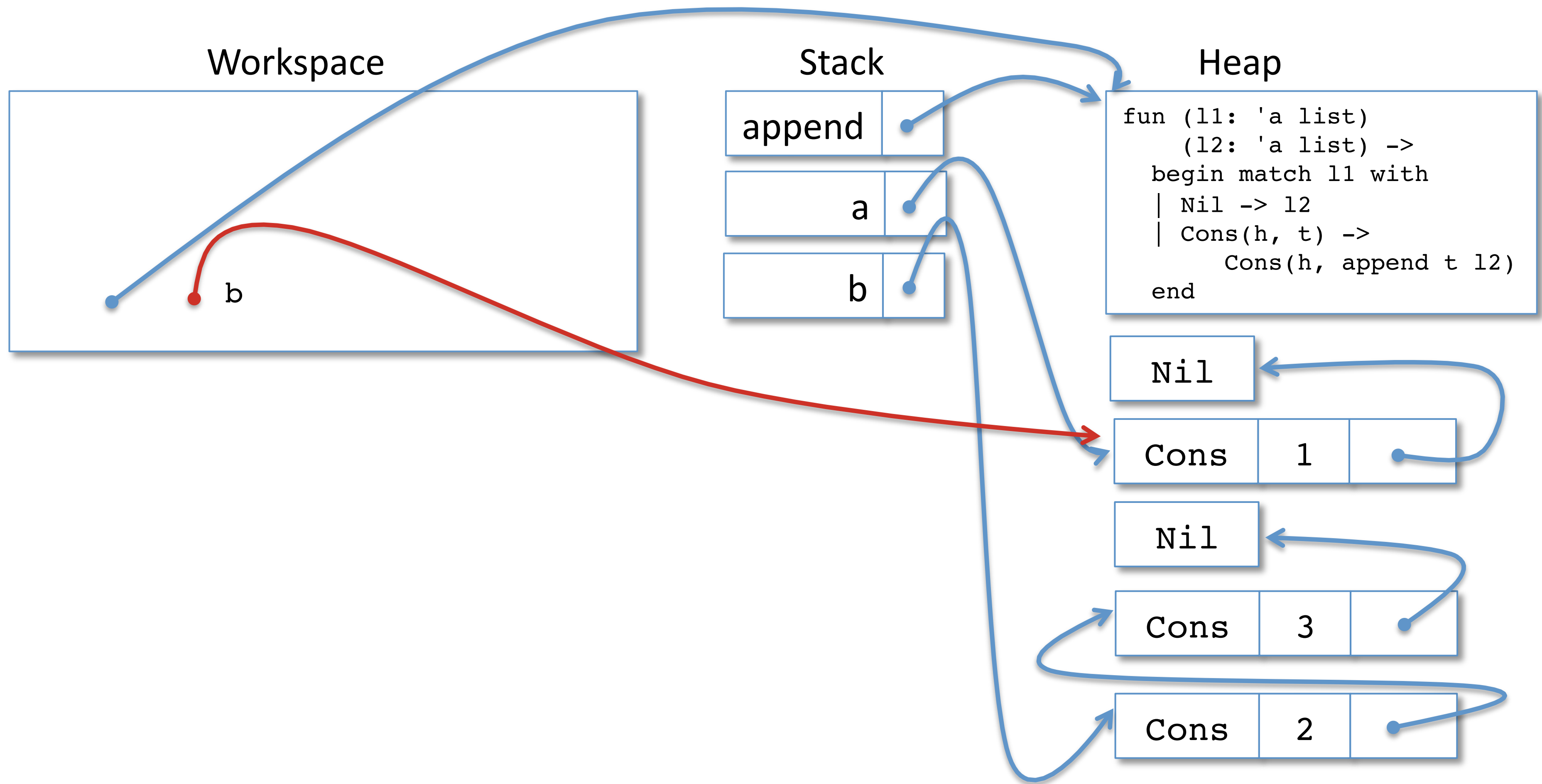
Lookup ‘append’



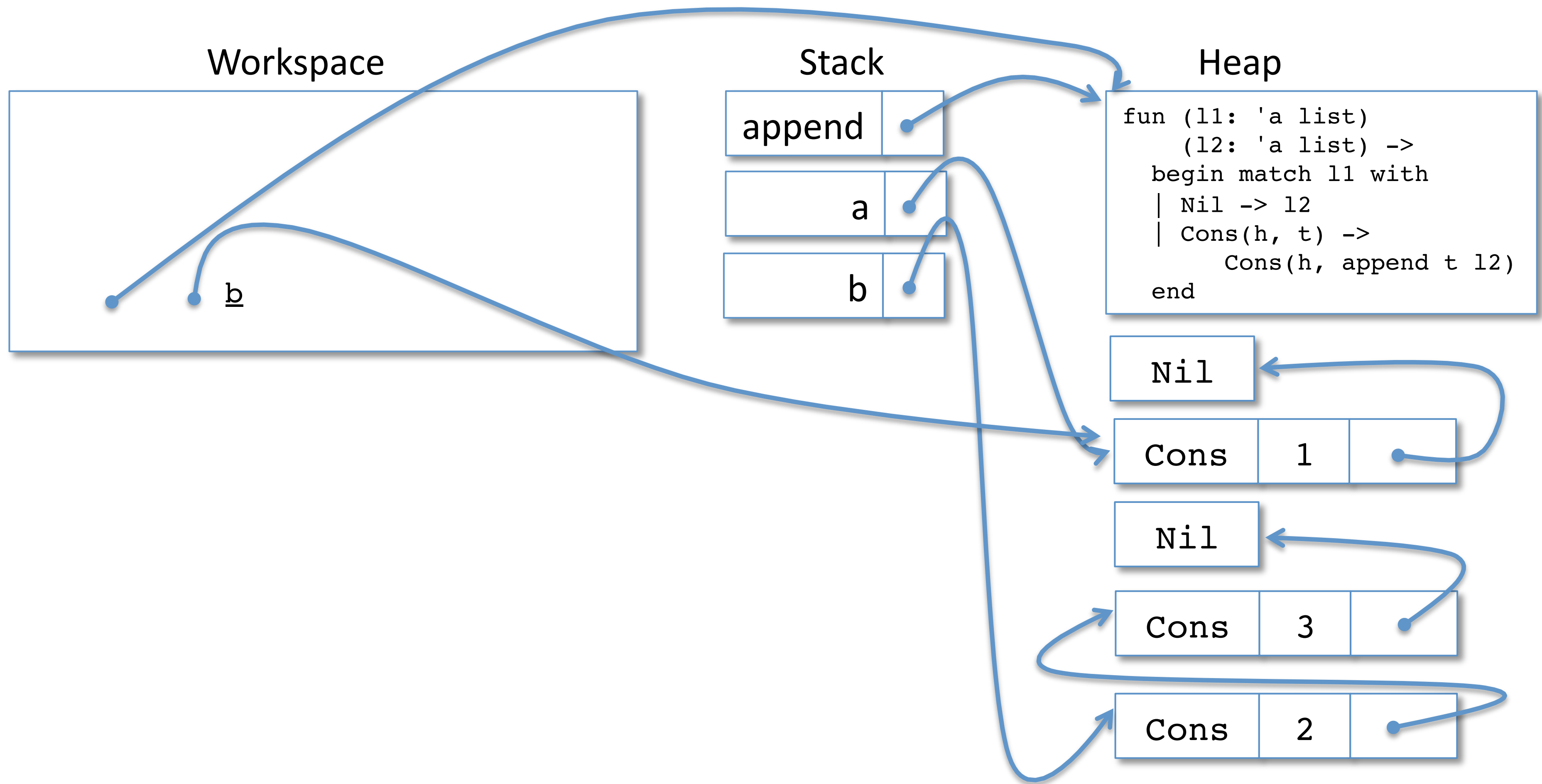
Lookup 'a'



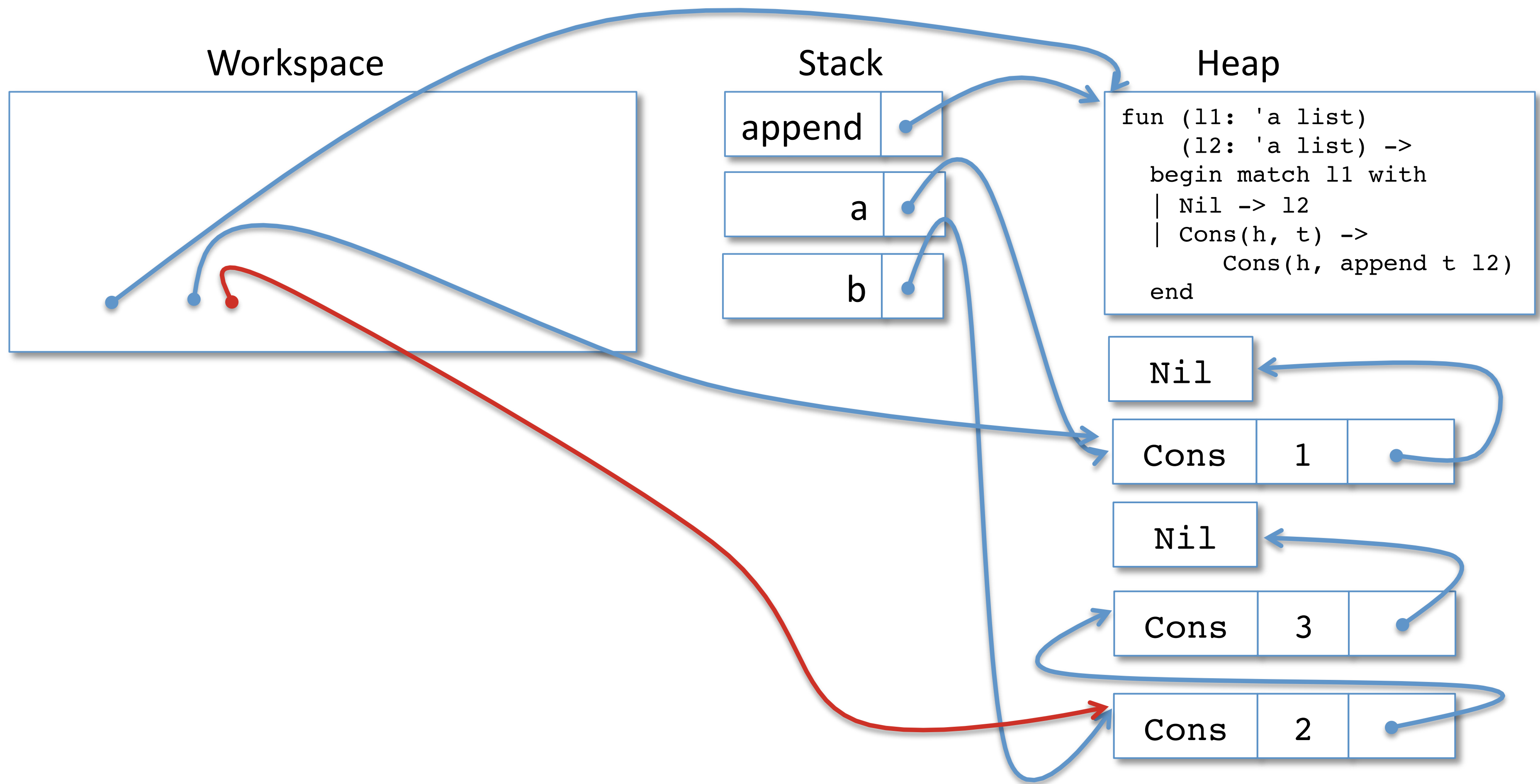
Lookup 'a'



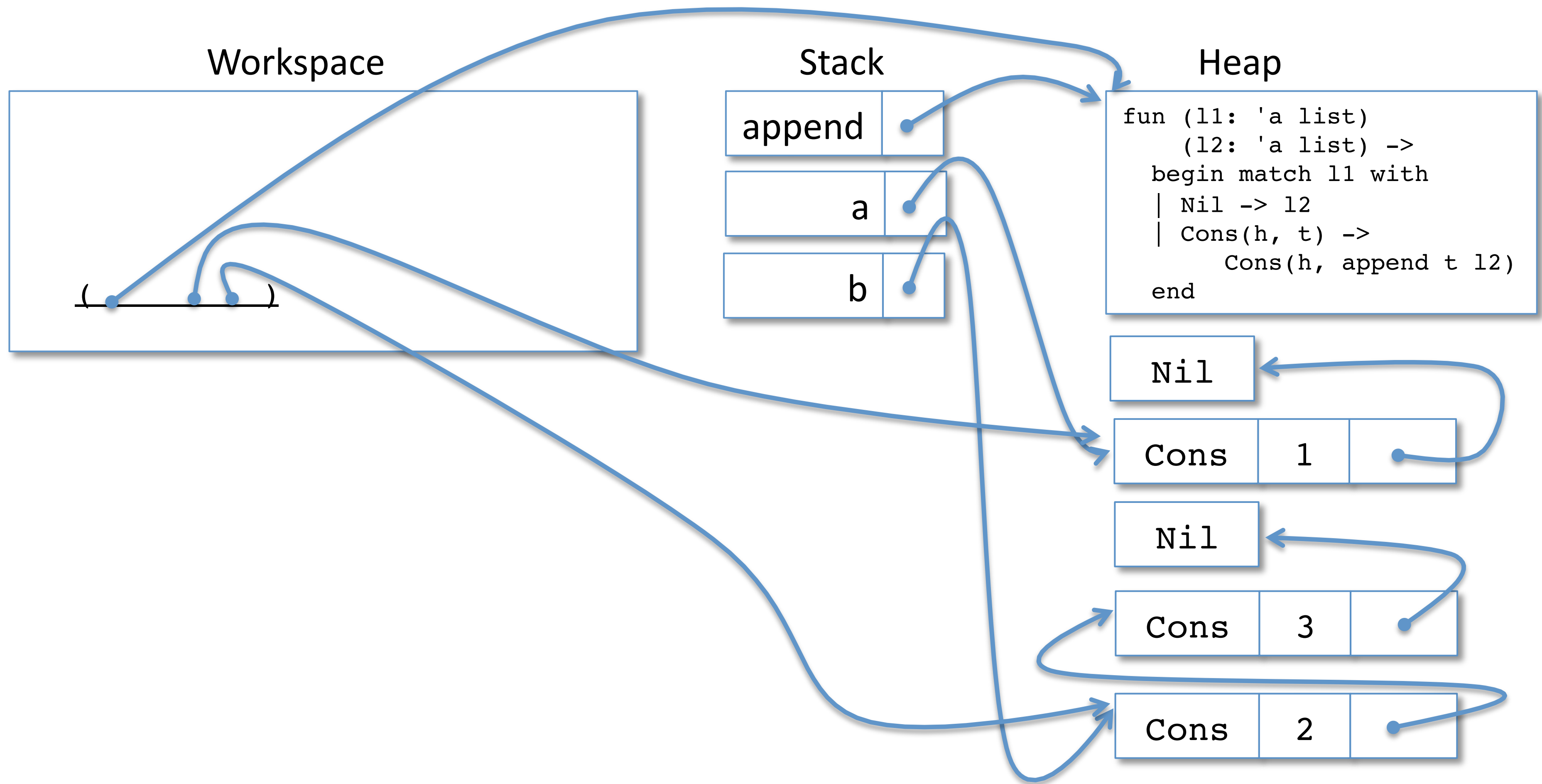
Lookup 'b'



Lookup 'b'



Do the Function call

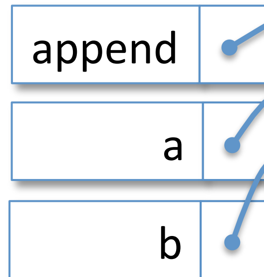


Save Workspace; push l1, l2

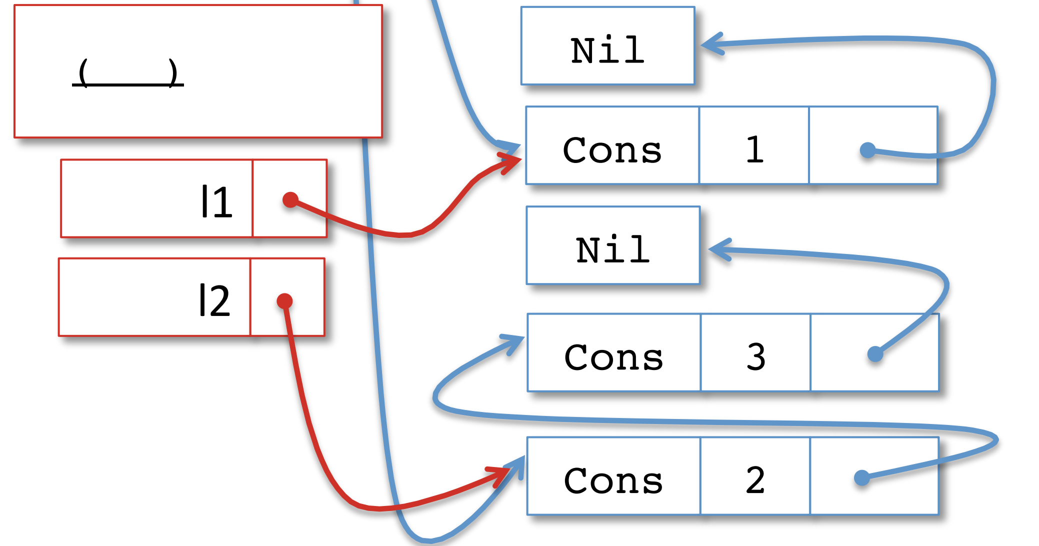
Workspace

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

Stack



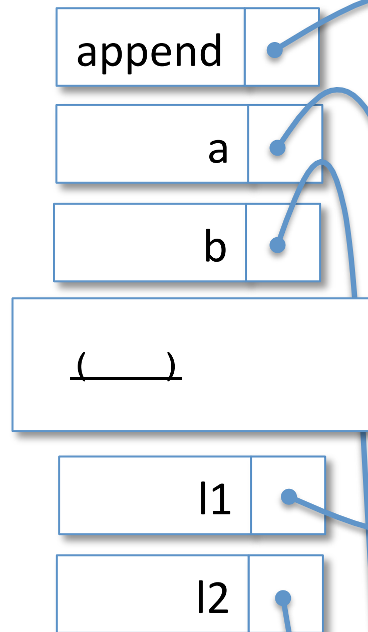
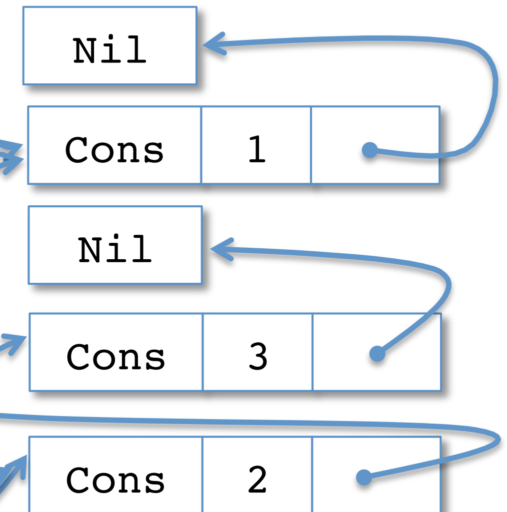
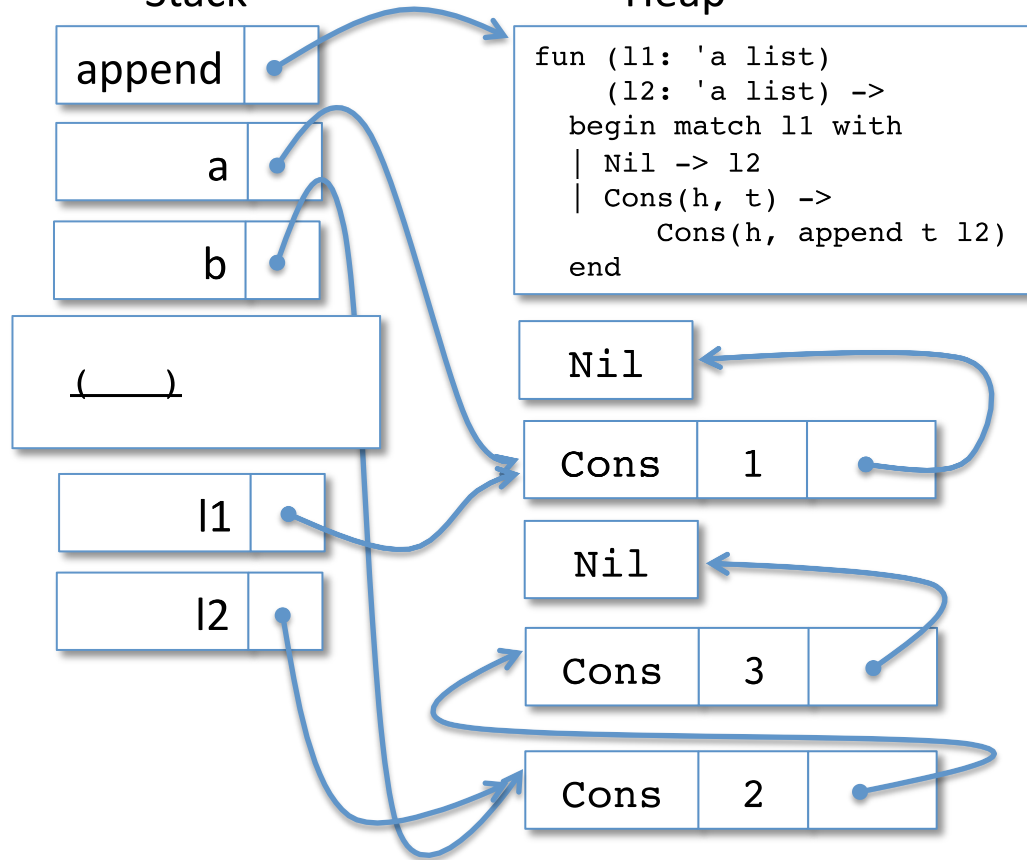
Heap



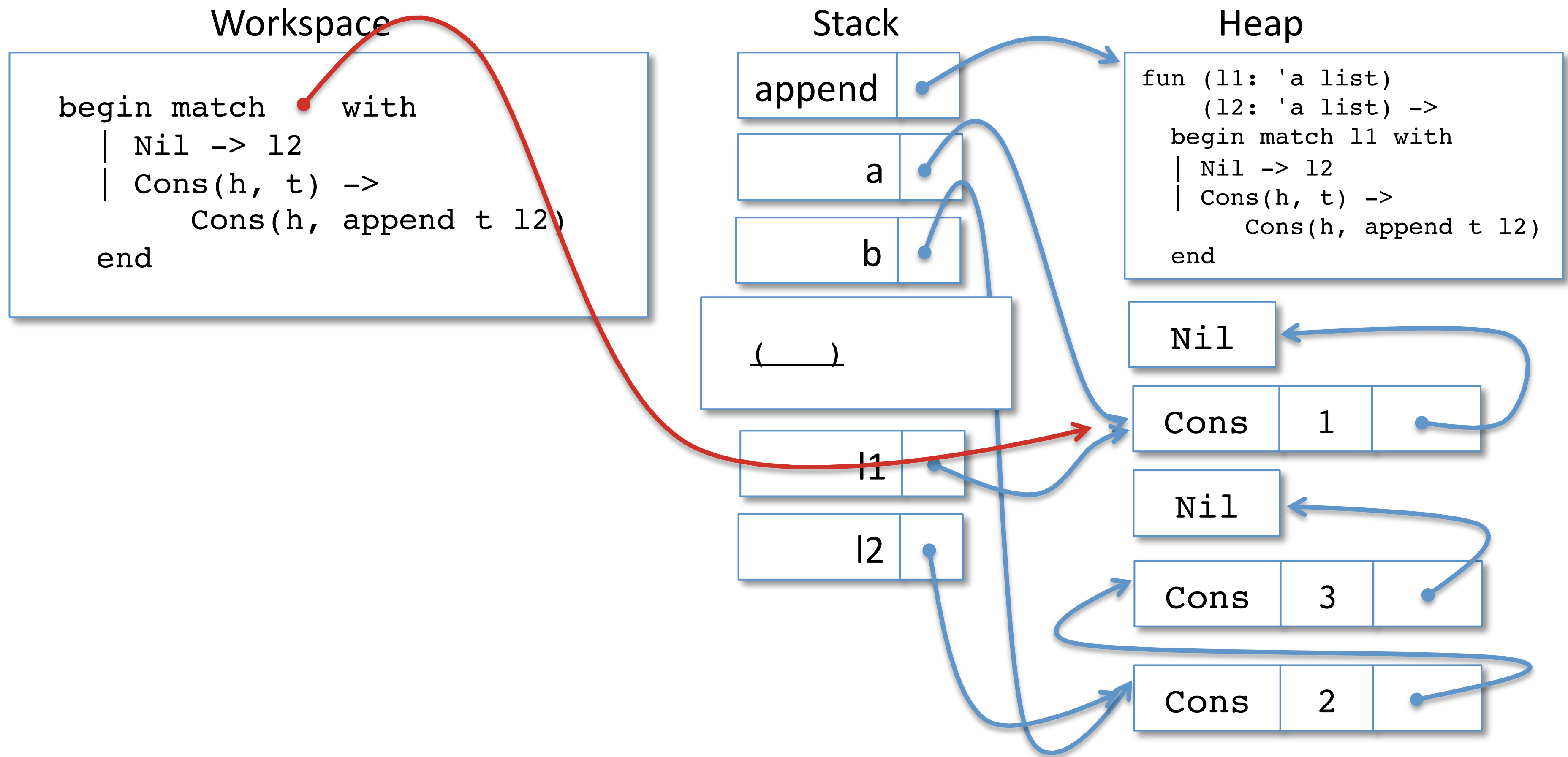
Lookup l1

Workspace

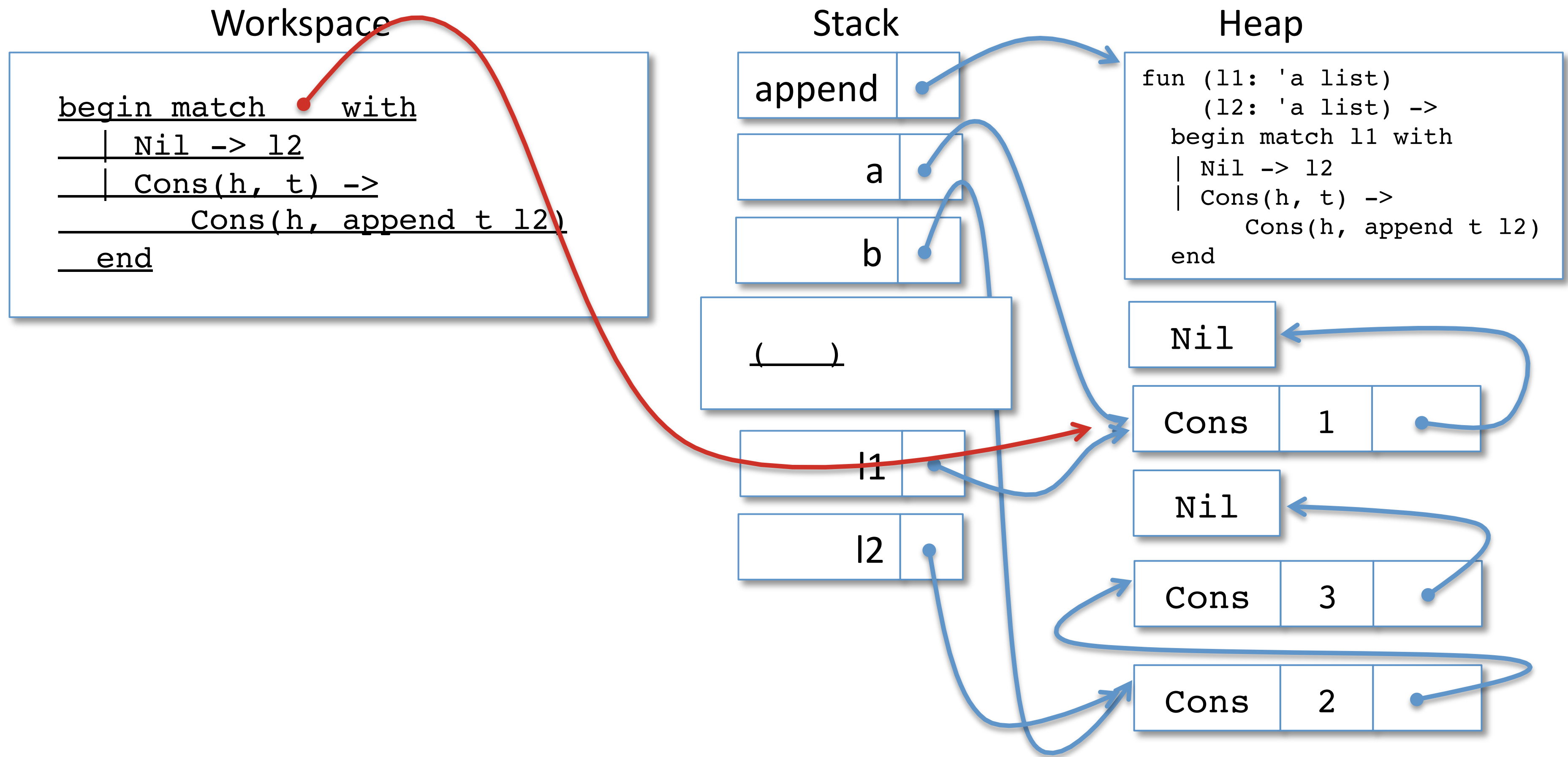
```
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end
```

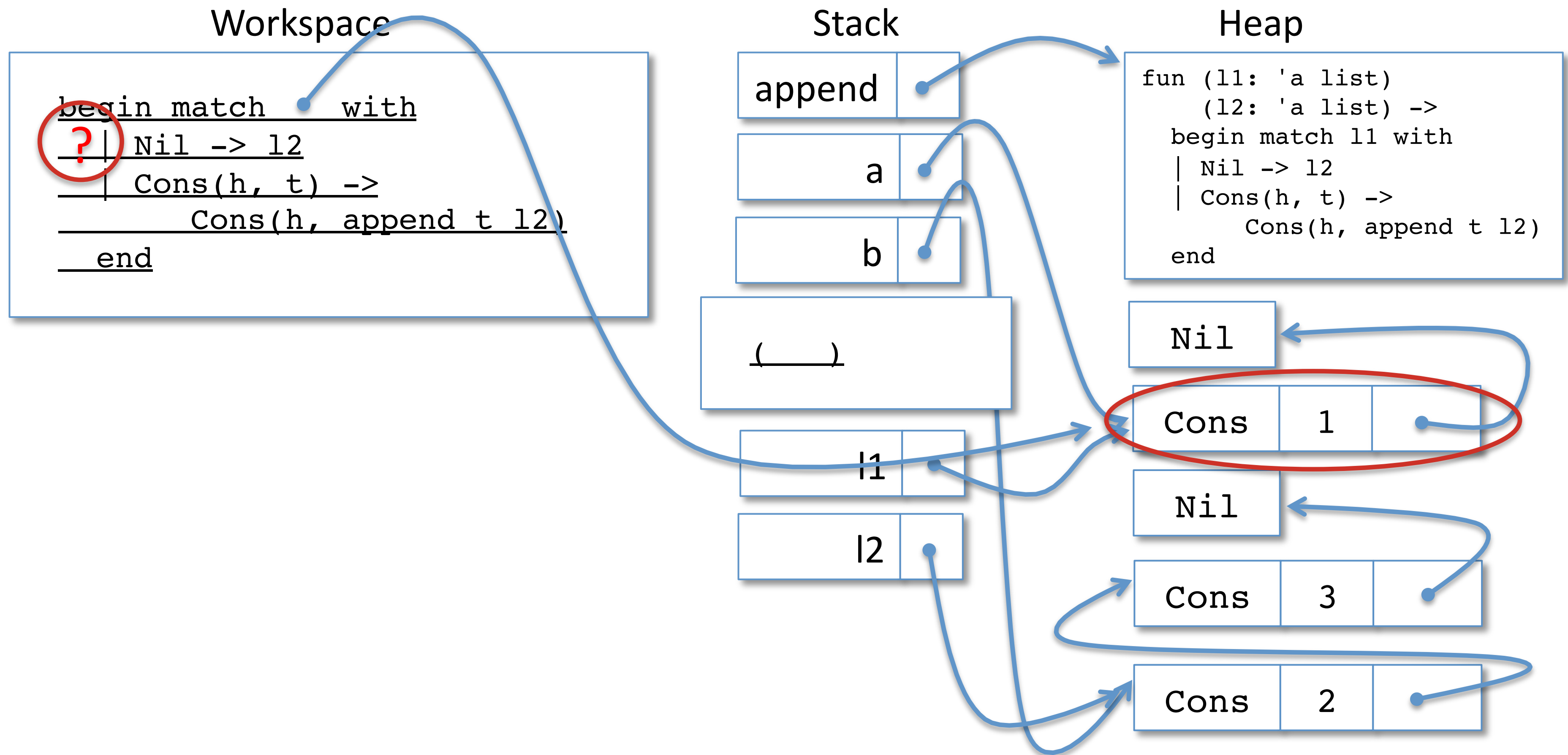
Lookup l1



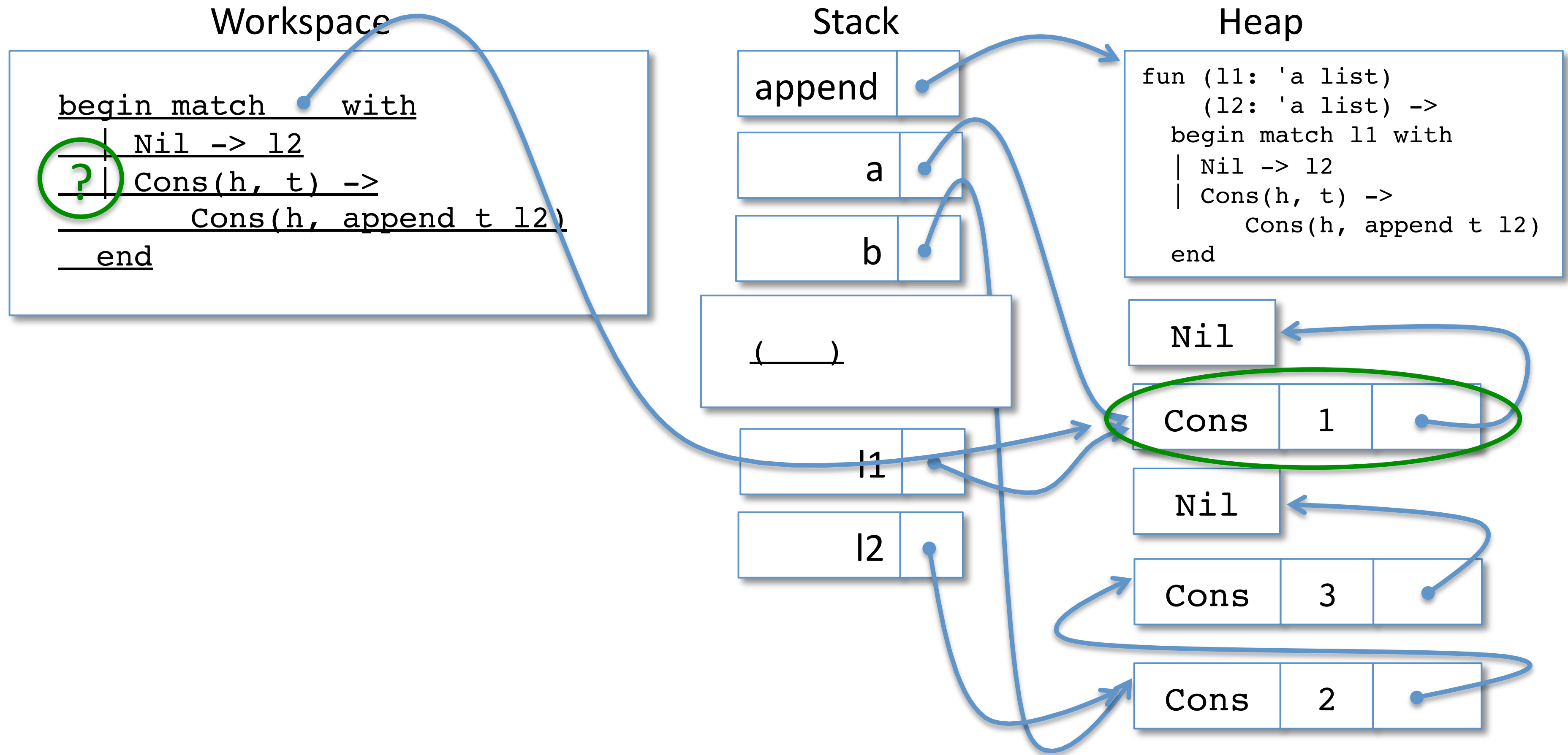
Match Expression



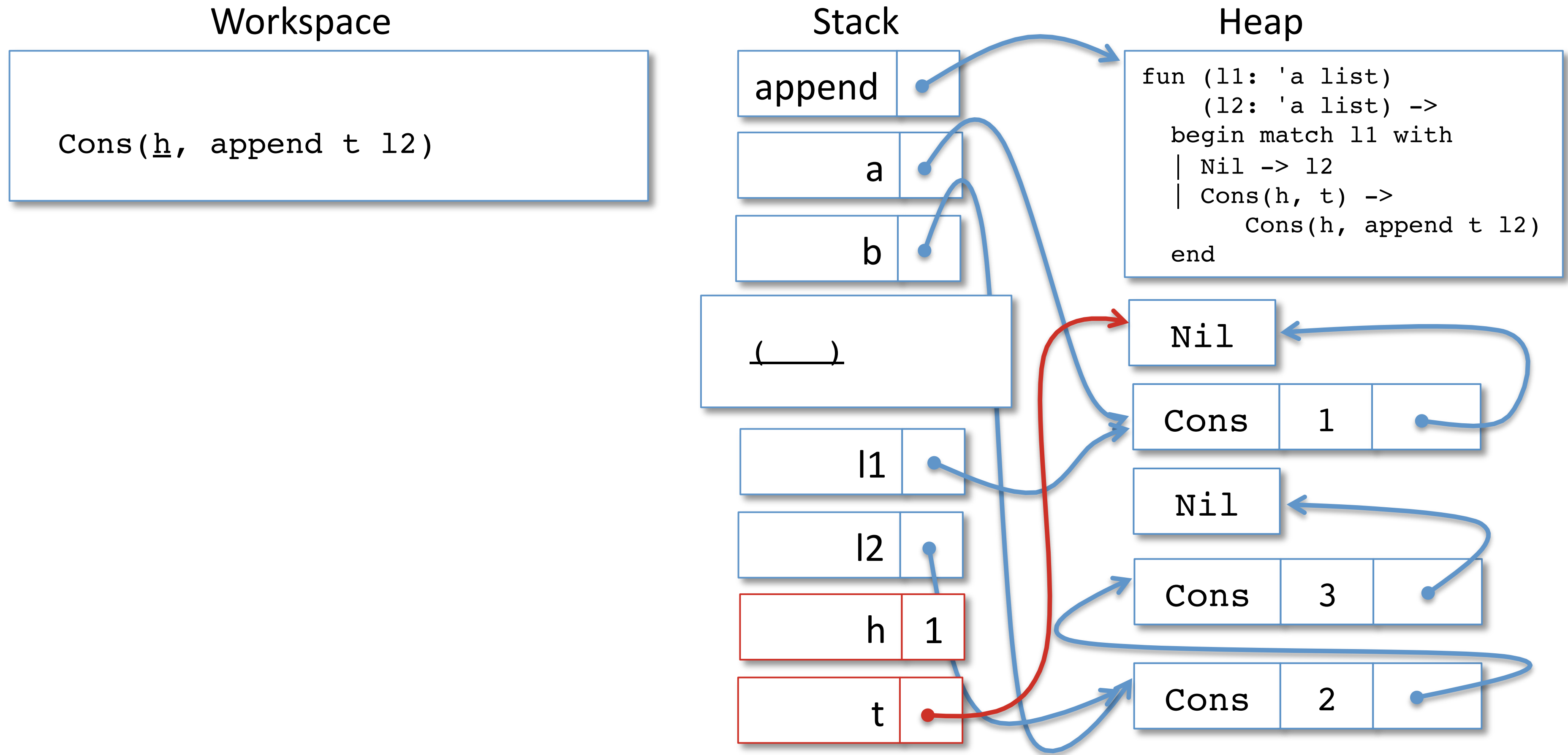
Nil case Doesn't Match



Cons case *Does* Match



Push h and t; lookup h

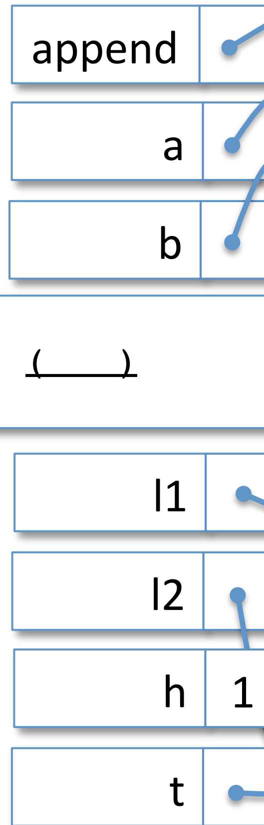


Lookup h

Workspace

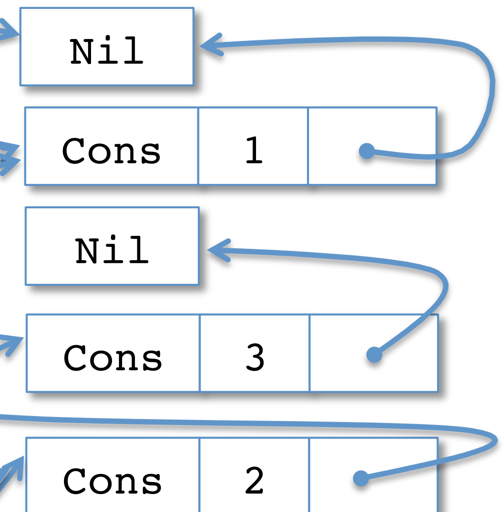
```
Cons(1, append t l2)
```

Stack



Heap

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

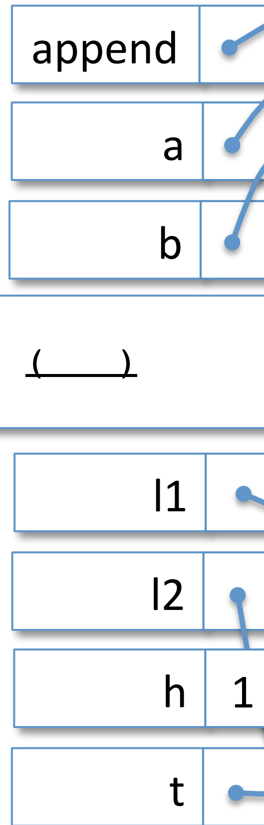


Lookup ‘append’

Workspace

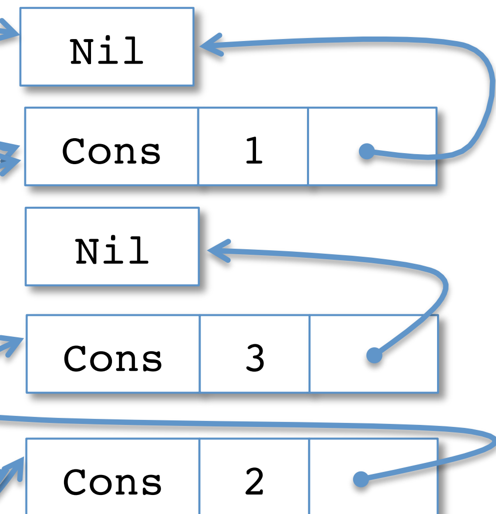
```
Cons(1, append t l2)
```

Stack

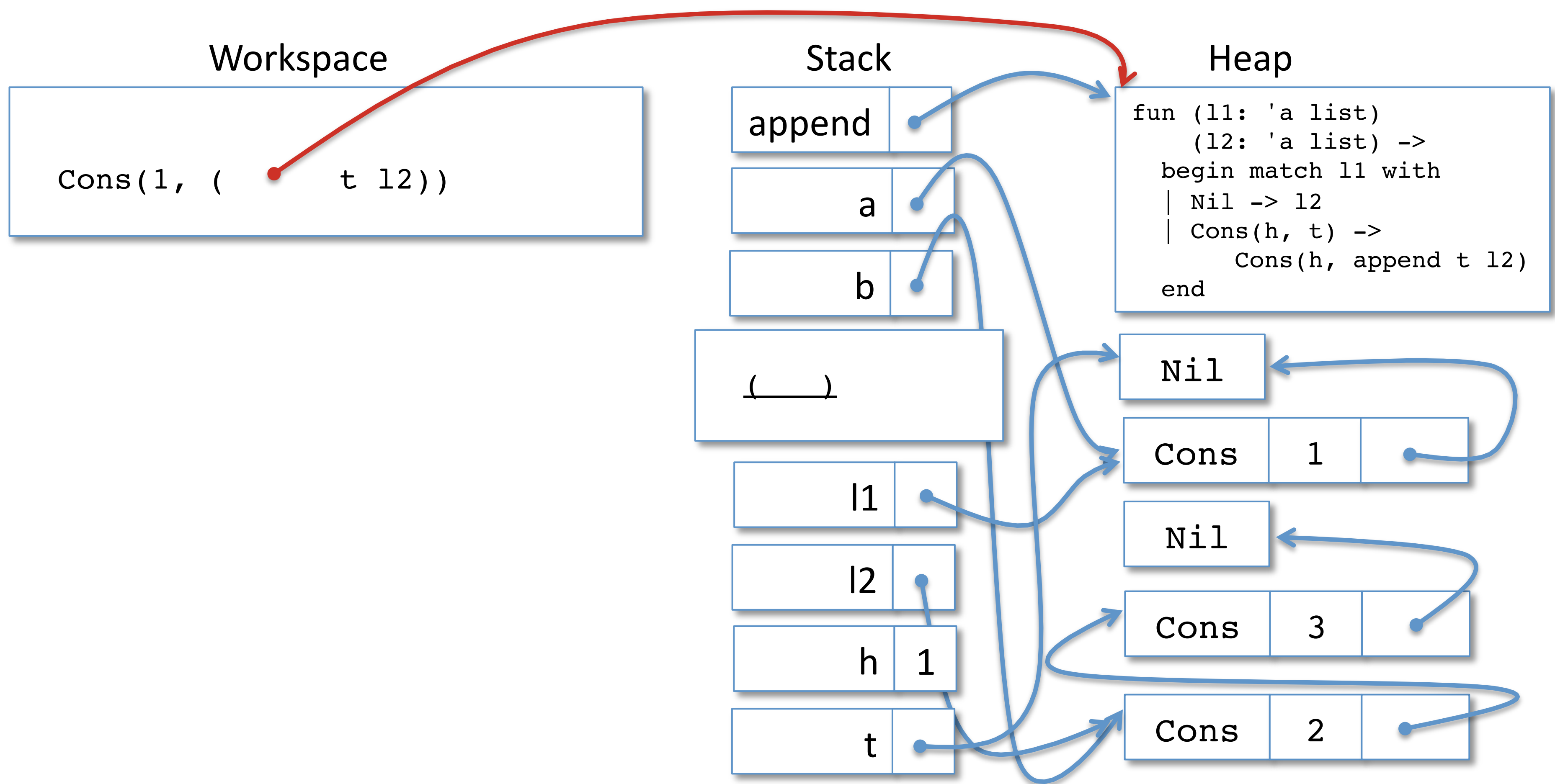


Heap

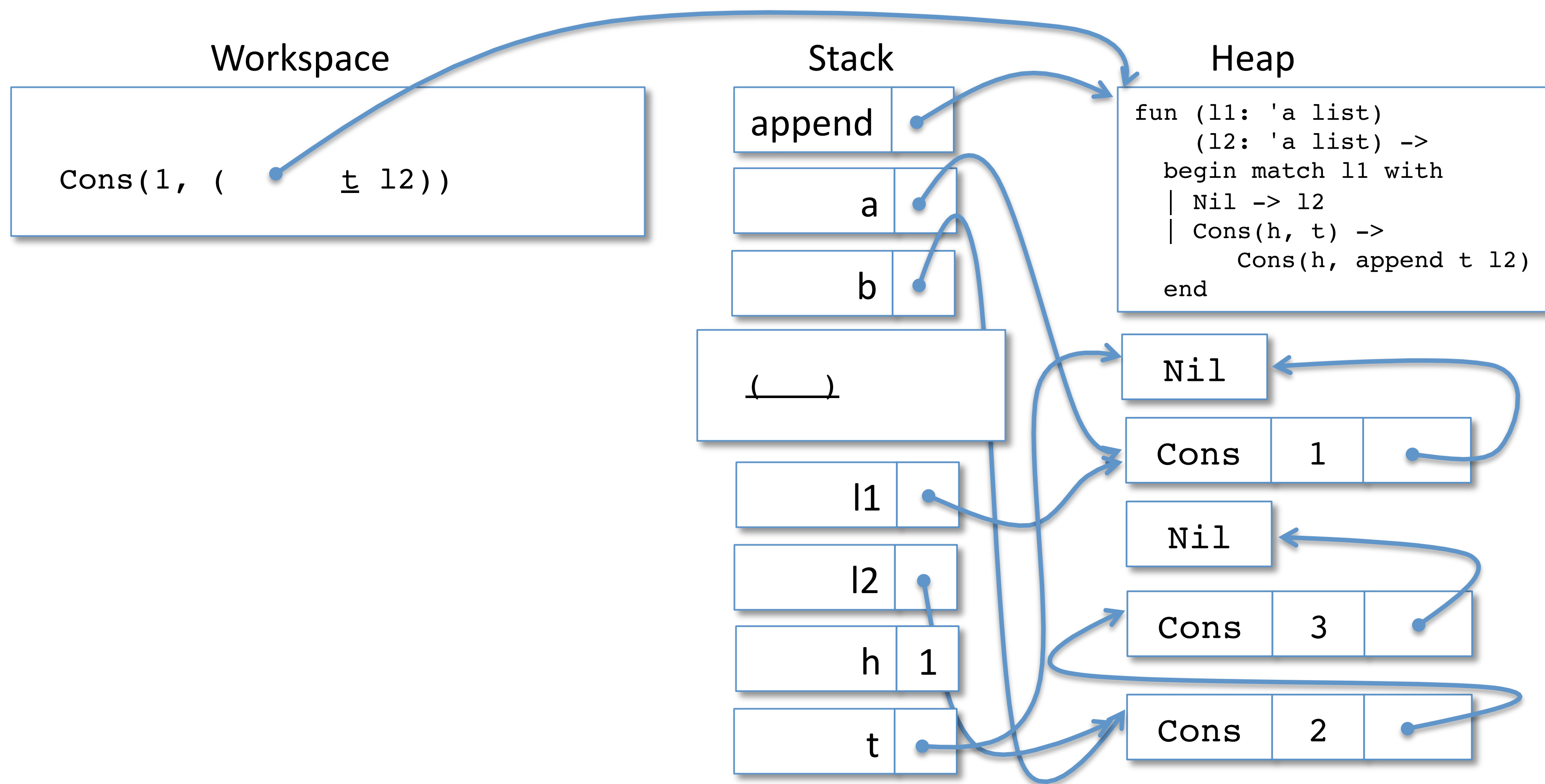
```
fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
   | Nil -> l2
   | Cons(h, t) ->
        Cons(h, append t l2)
  end
```



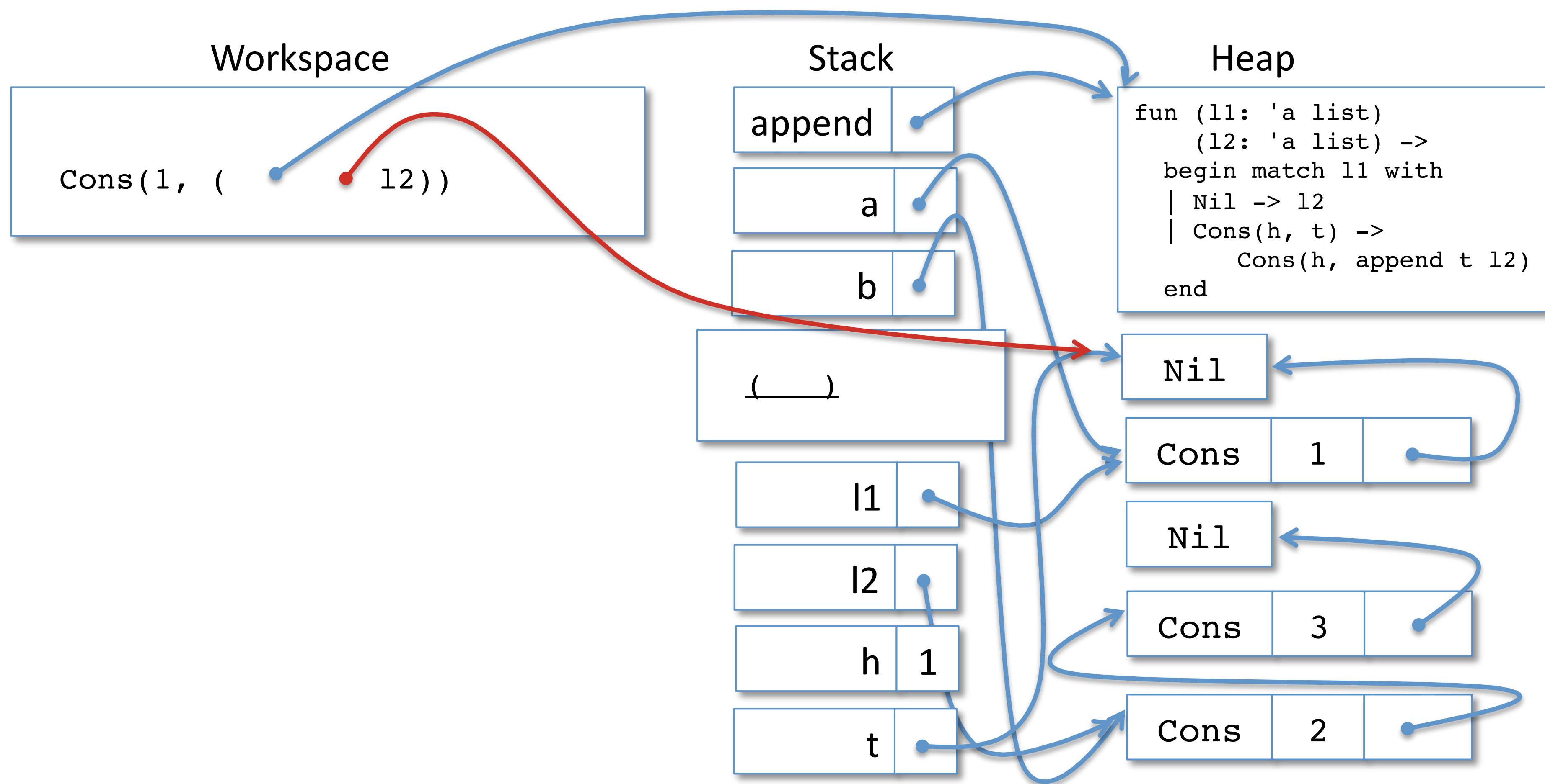
Lookup 'append'



Lookup ‘t’



Lookup 't'



Lookup 'l2'

Workspace

```
Cons(1, (        l2))
```

Stack

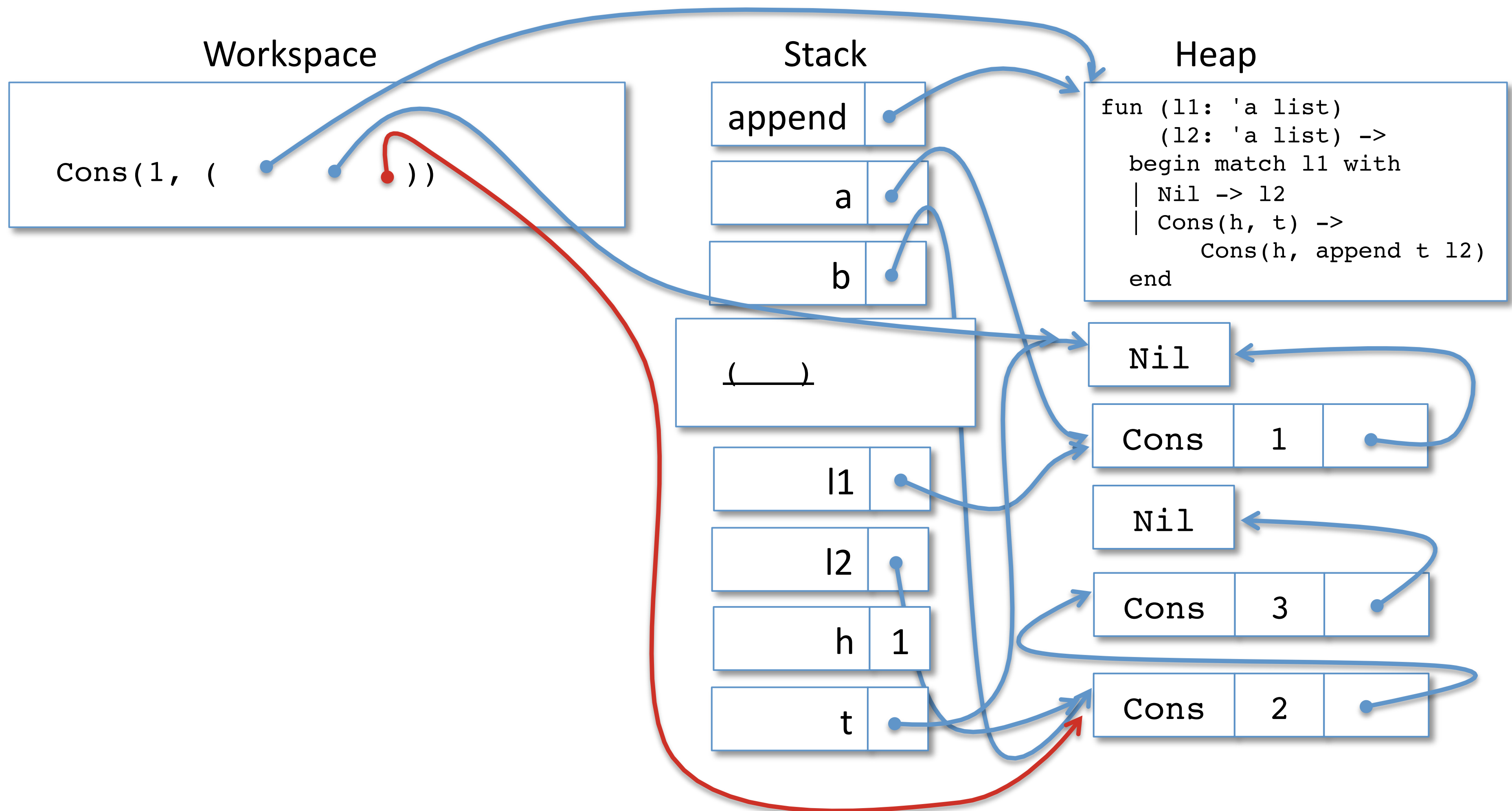
append	•
a	•
b	•
(___)	
l1	•
l2	•
h	1
t	•

Heap

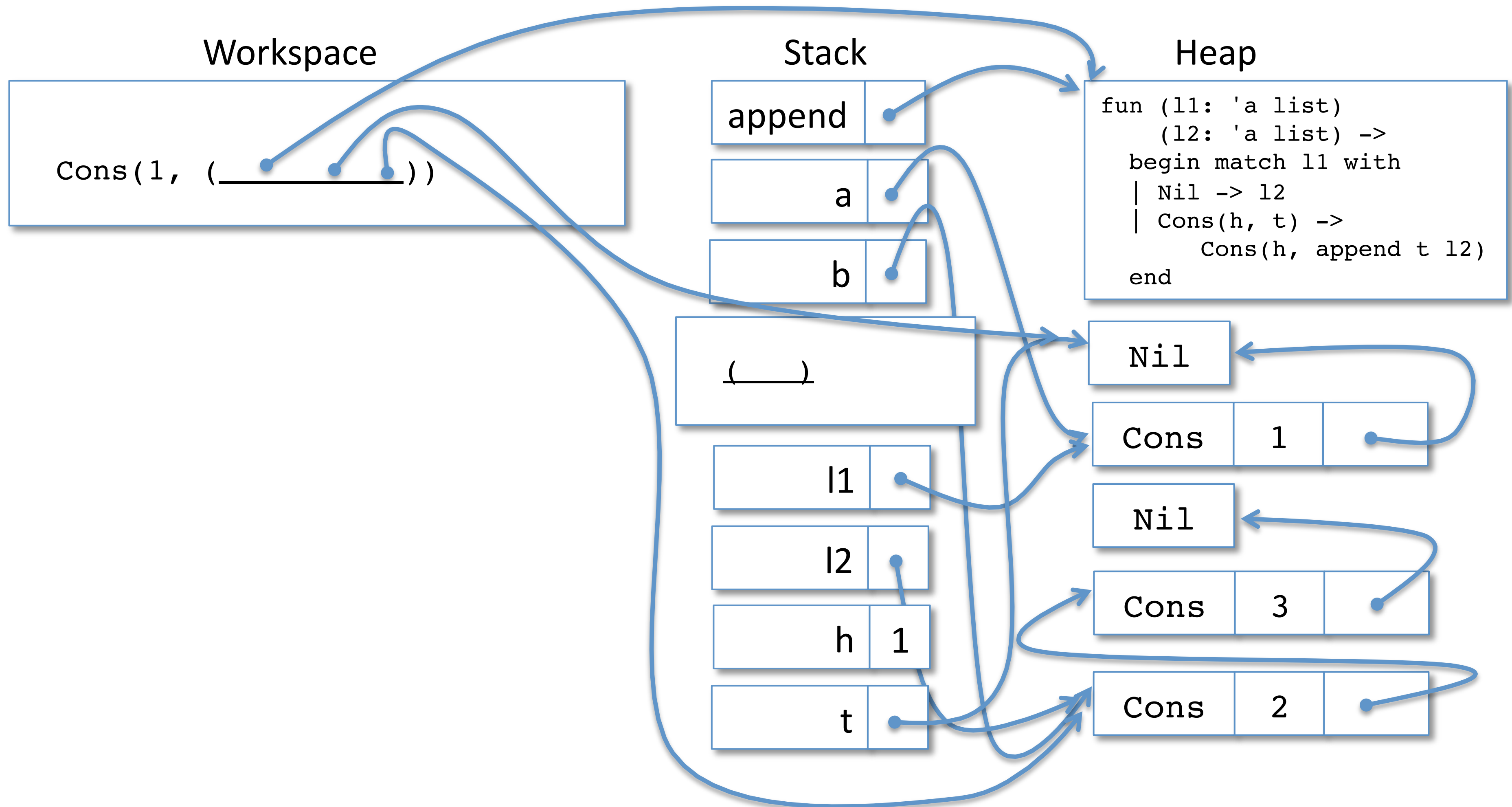
```
fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end
```

Nil		
Cons	1	•
Nil		
Cons	3	•
Cons	2	•

Lookup 'l2'



Do the Function Call

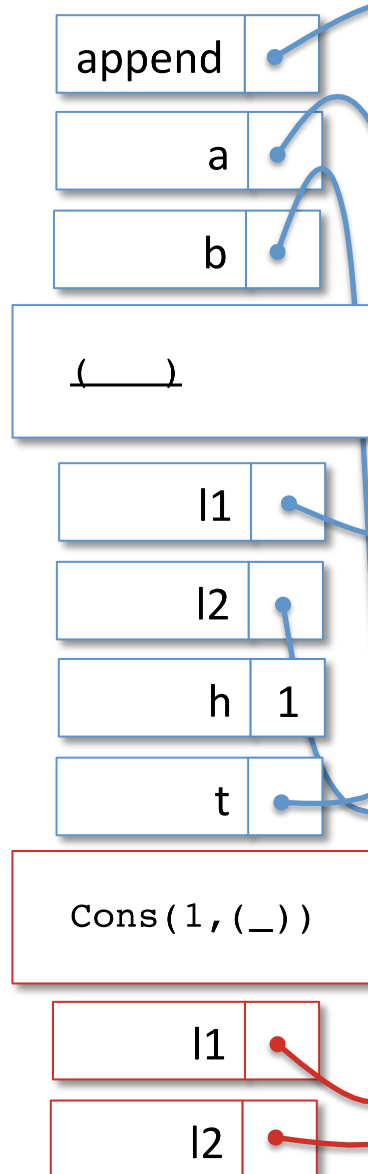


Save the Workspace; push l1, l2

Workspace

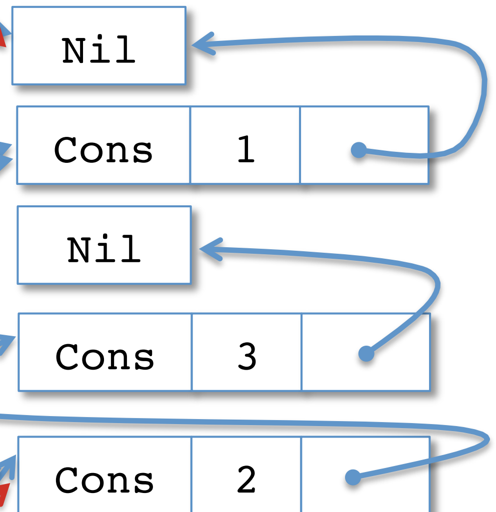
```
begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
 end
```

Stack



Heap

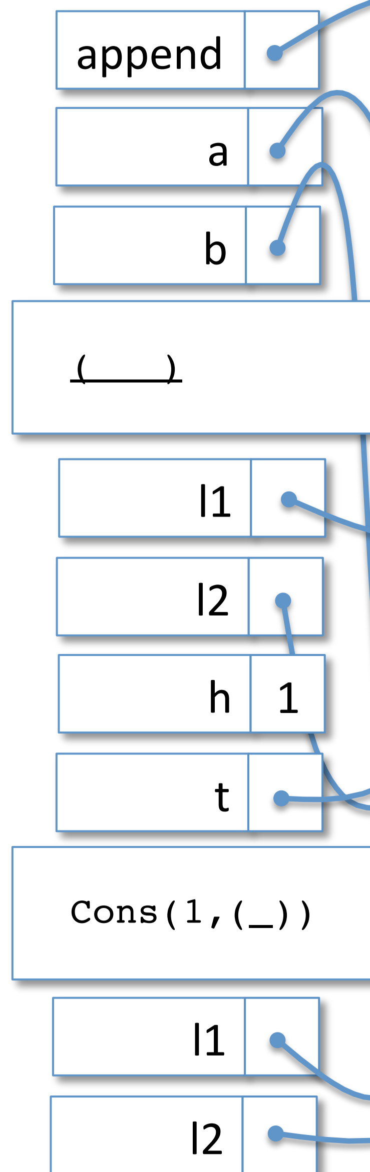
```
fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
   | Nil -> l2
   | Cons(h, t) ->
        Cons(h, append t l2)
  end
```

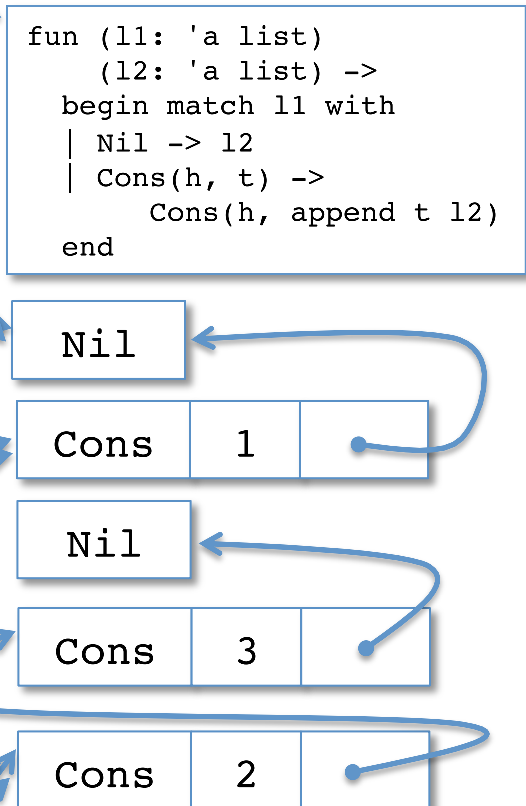


Lookup 'l1'

Workspace

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





Lookup ‘l1’

Workspace

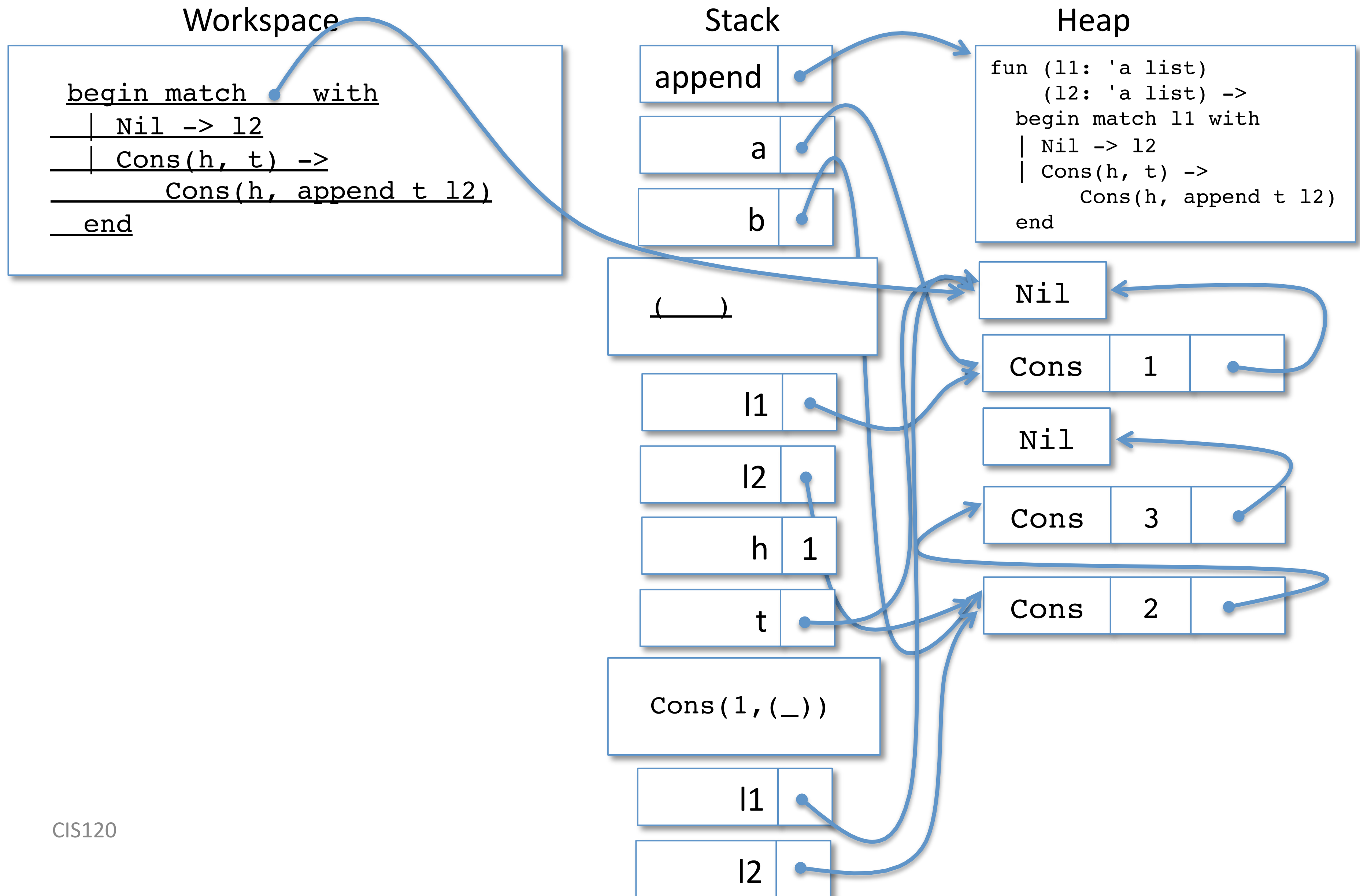
```
begin match      with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end
```

Stack

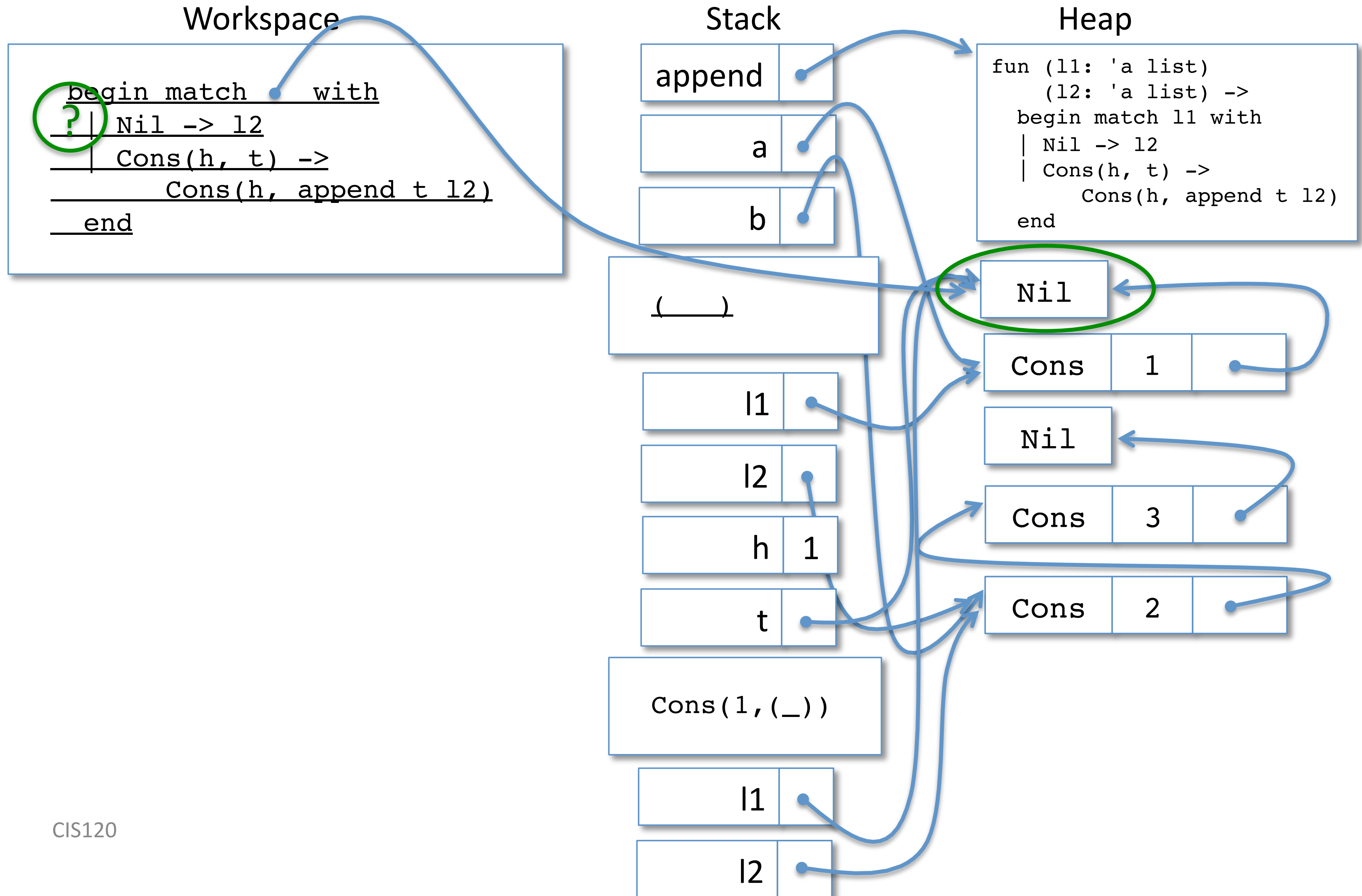
Heap

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

Match Expression



The Nil case Matches



Simplify the Branch (nothing to push)

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

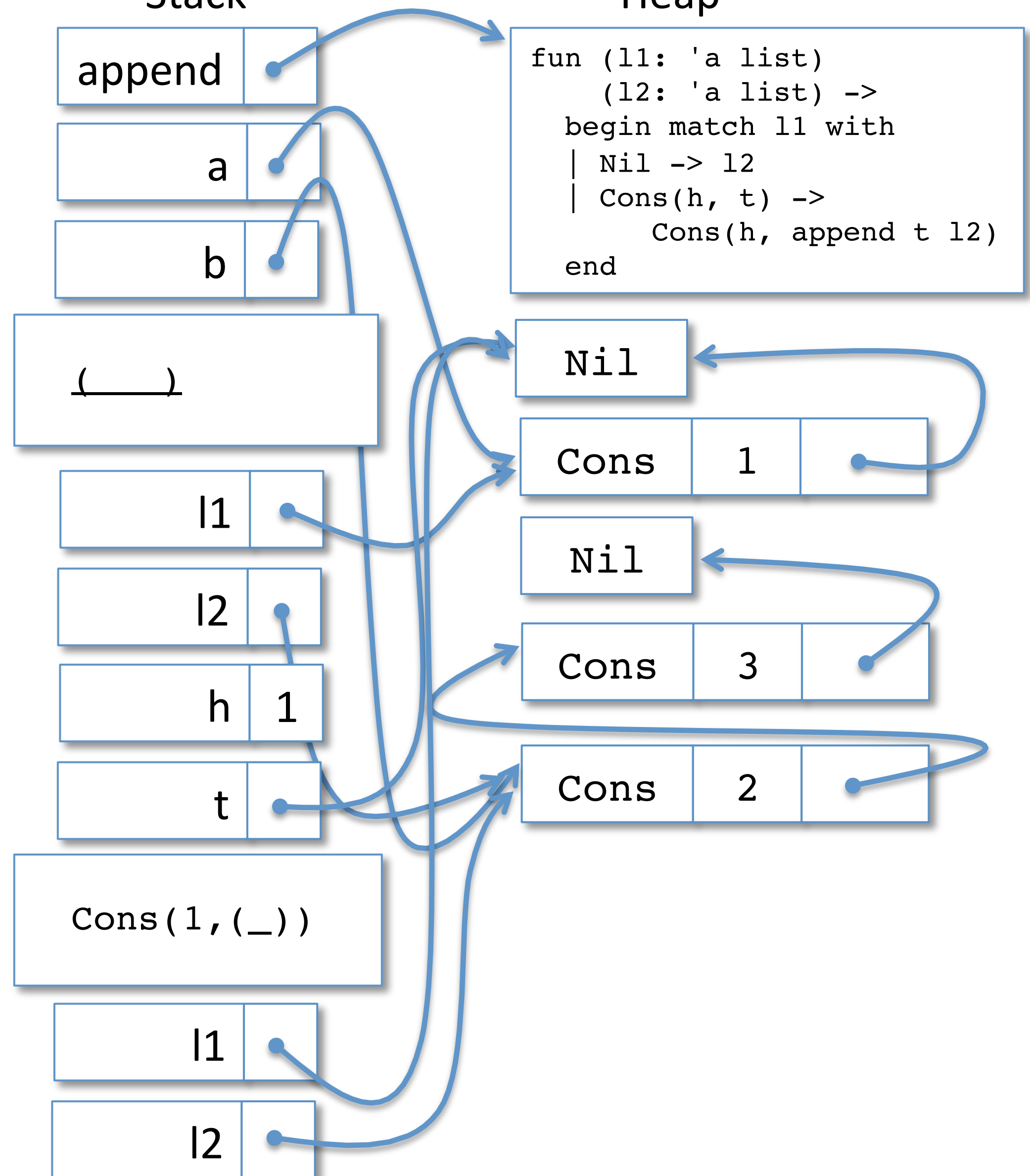
Cons | 3 | •

Cons | 2 | •

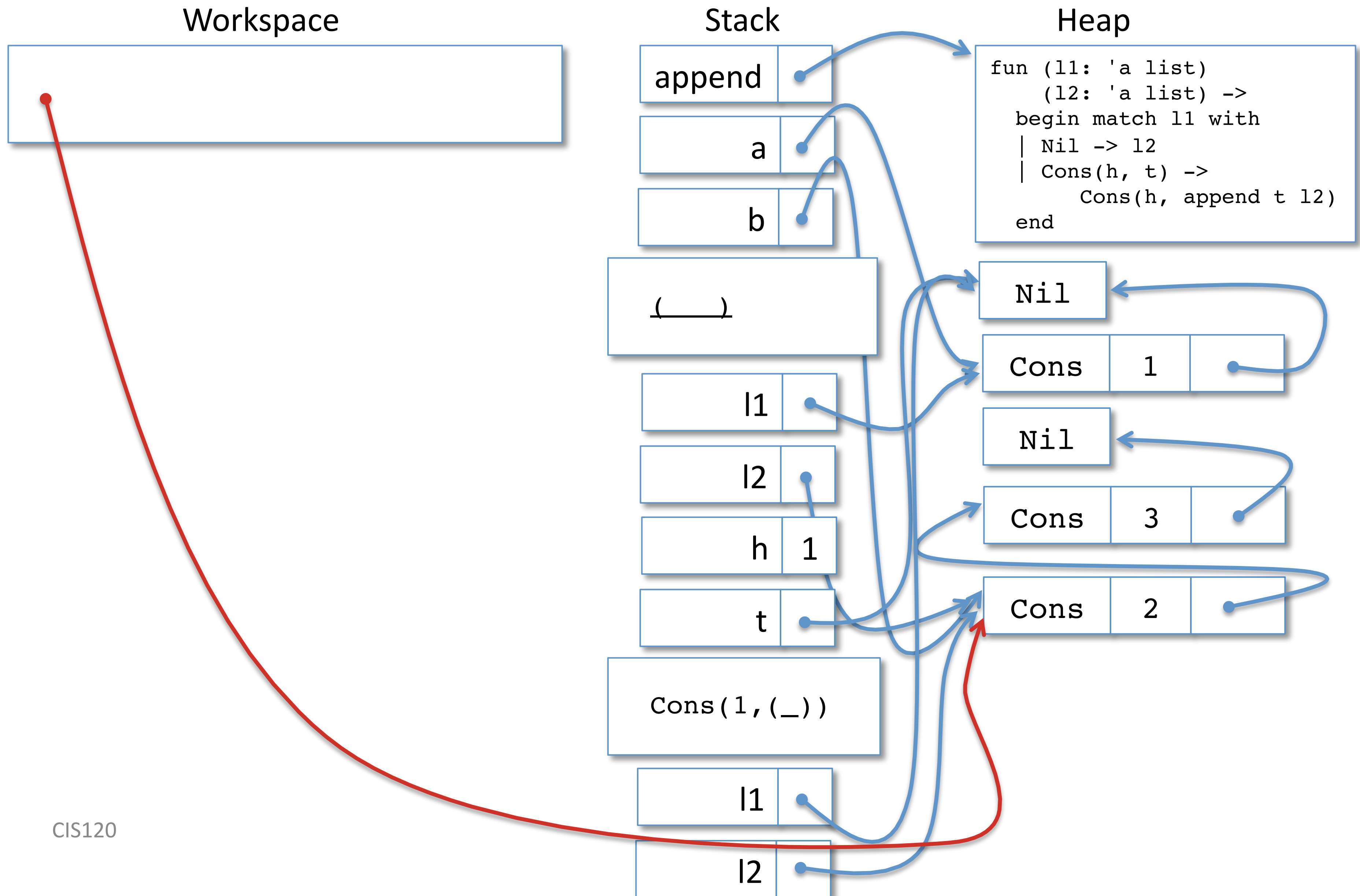
Lookup 'l2'

Workspace

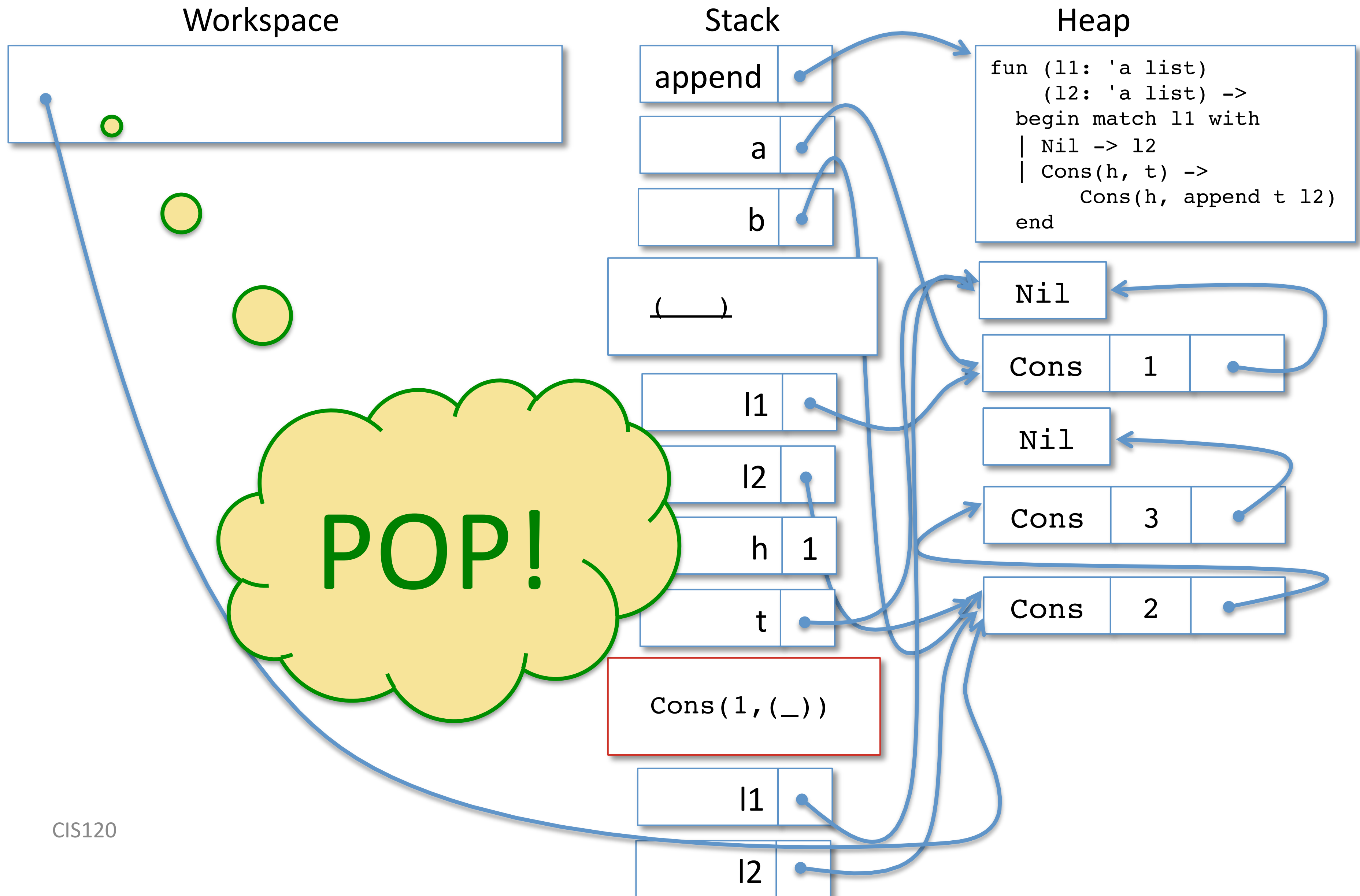
l2



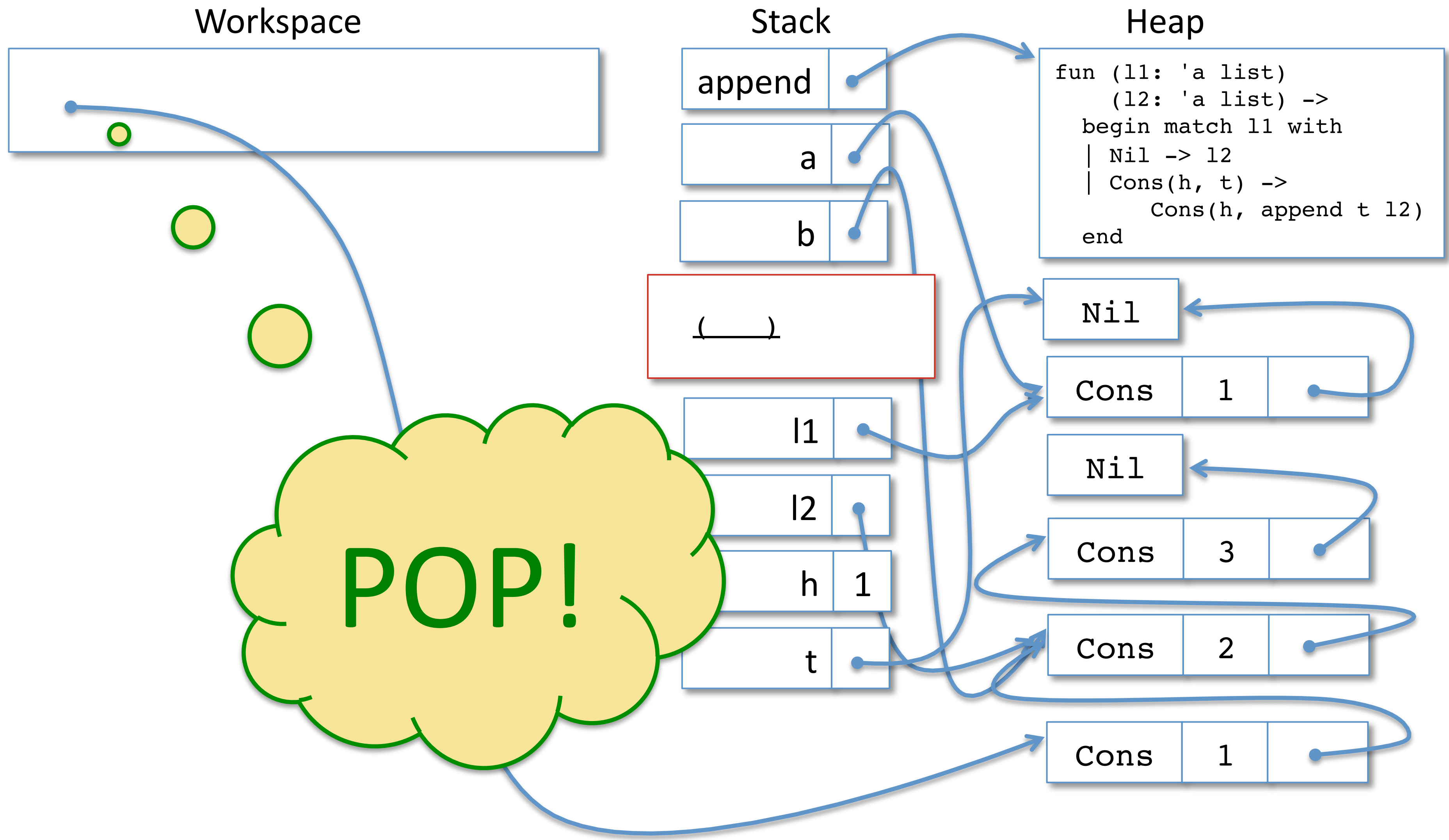
Lookup ‘l2’



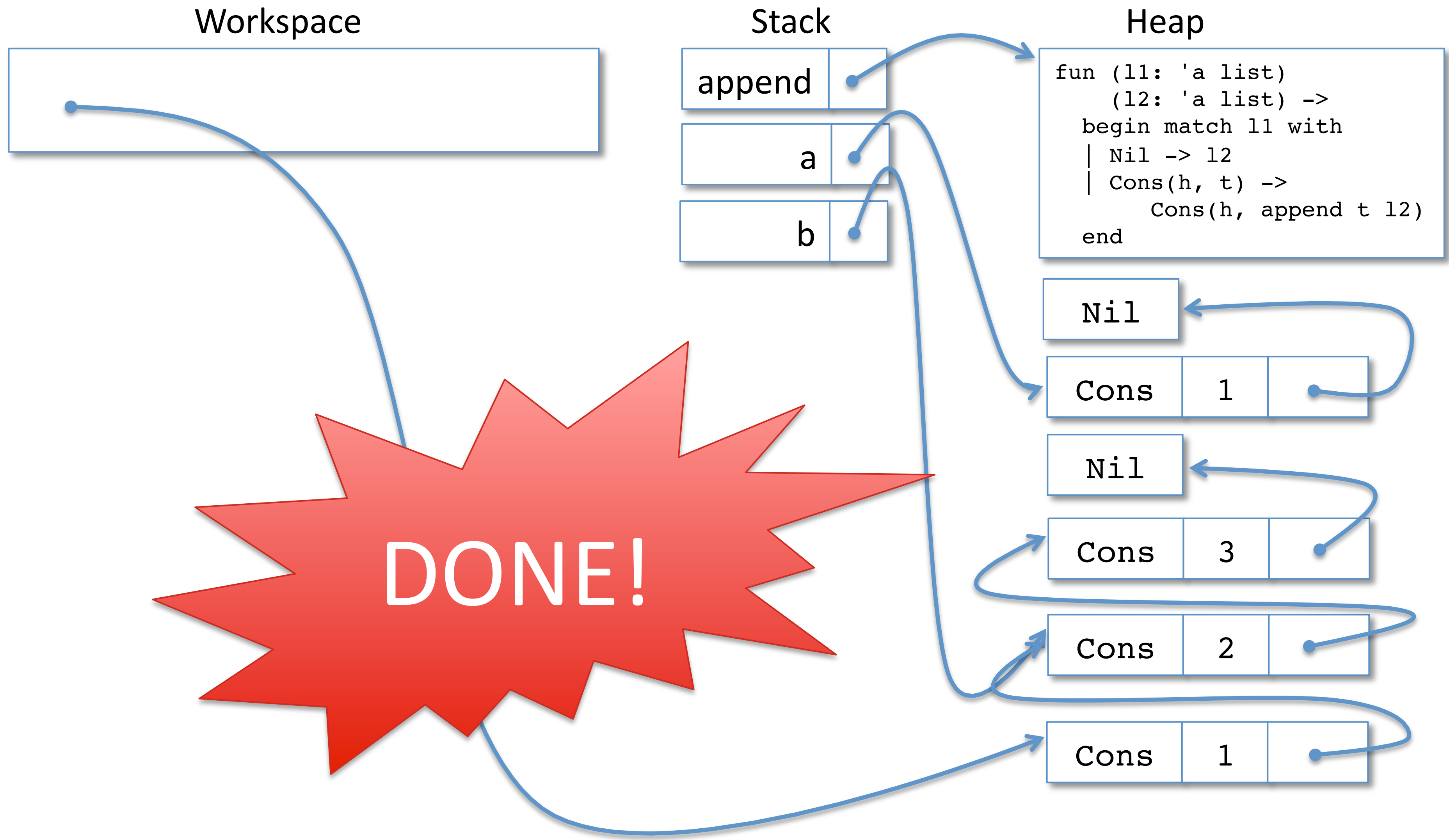
Done! Pop stack to last Workspace



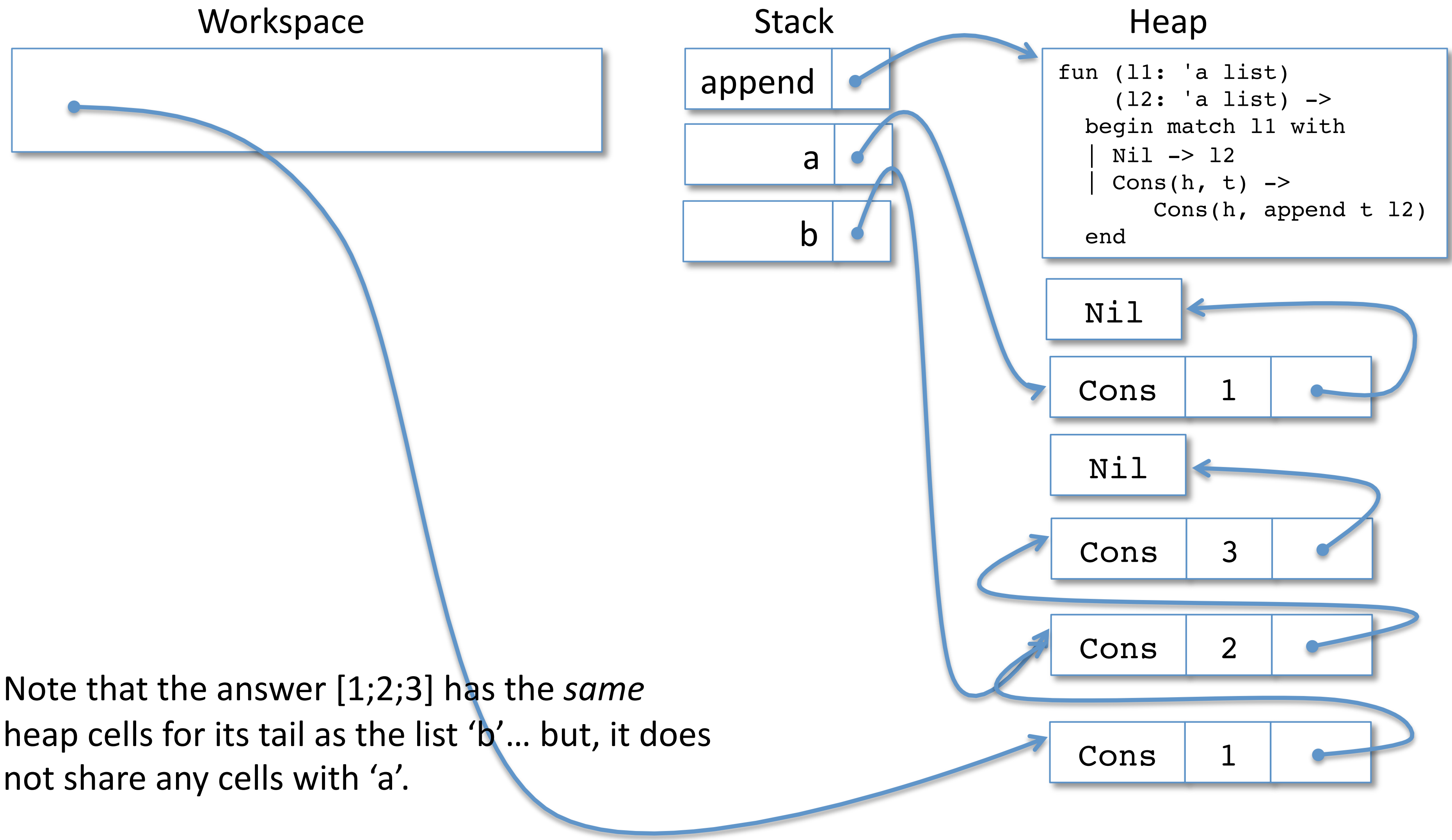
Done! Pop stack to last Workspace



Done! (PHEW!)



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'.

Putting State to Work

Mutable Queues

A design problem

Suppose you are implementing a website to sell tickets to a very popular music event. To be fair, you would like to allow people to select seats first come, first served. How would you do it?

- Understand the problem
  - Some people may visit the website to buy tickets while others are still selecting their seats
  - Need to remember the order in which people purchase tickets
- Define the interface
  - Need a datastructure to store ticket purchasers
  - Need to add purchasers to the end of the line
  - Need to allow purchasers at the beginning of the line to select seats
  - Needs to be *mutable* so the state can be shared across web sessions

(Mutable) Queue Interface

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

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Determine if the queue is empty *)
  val is_empty : 'a queue -> bool


  (* Add a value to the end of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the first value (if any) and return it *)
  val deq : 'a queue -> 'a

end
```

We can tell, just looking at this interface, that it is for a MUTABLE data structure. How?

Because queues are mutable, we must allocate a new one every time we need one.

Adding an element to the queue returns unit because it modifies the given queue.

Define test cases

```
let test () : bool =
  let q : int queue = create () in
  enq 1 q;
  enq 2 q;
  1 = deq q
;; run_test "queue test 1" test

let test () : bool =
  let q : int queue = create () in
  enq 1 q;
  enq 2 q;
  let _ = deq q in
  2 = deq q
;; run_test "queue test 2" test
```

Implement the behavior

```
module ListQueue : QUEUE  = struct

  type 'a queue = { mutable contents : ‘a list }

  let create () : 'a queue =
    { contents = [] }

  let is_empty (q:'a queue) : bool =
    q.contents = []

  let enq (x:'a) (q:'a queue) : unit =
    q.contents <- (q.contents @ [x])

  let deq (q:'a queue) : 'a =
    begin match q.contents with
      | [] -> failwith "deq called on empty queue"
      | x::tl -> q.contents <- tl; x
    end
 end
```

Here we are using type abstraction to protect the state. Outside of the module, no one knows that queues are implemented with a mutable structure. So, only these functions can modify this structure.

A Better Implementation

- Implementation is slow because of append:
  - `q.contents @ [x]` copies the entire list each time
  - As the queue gets longer, it takes longer to add data
  - Only has a *single* reference to the beginning of the list
- Let's do it again with TWO references, one to the beginning (head) and one to the end (tail).
  - Dequeue by updating the head reference (as before)
  - Enqueue by updating the tail of the list
- Challenge: The list itself must be mutable
  - because we add to one end and remove from the other

Data Structure for Mutable Queues

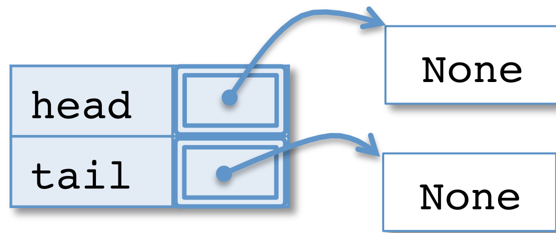
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

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

There are two parts to a mutable queue:

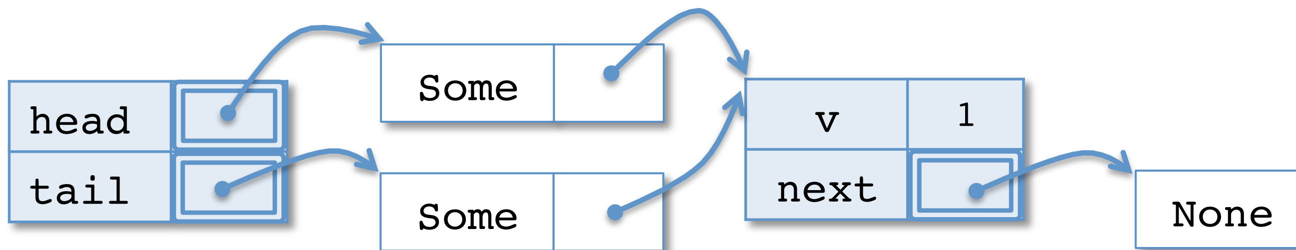
1. the "internal nodes" of the queue, with links from one to the next
1. a record with links to the head and tail nodes

All of the links are *options* so that the queue can be empty.

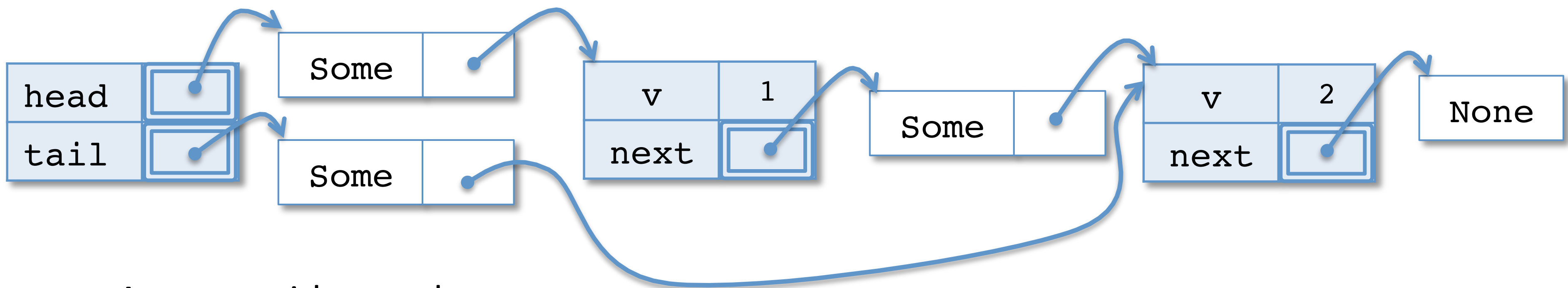
Queues in the Heap



An empty queue

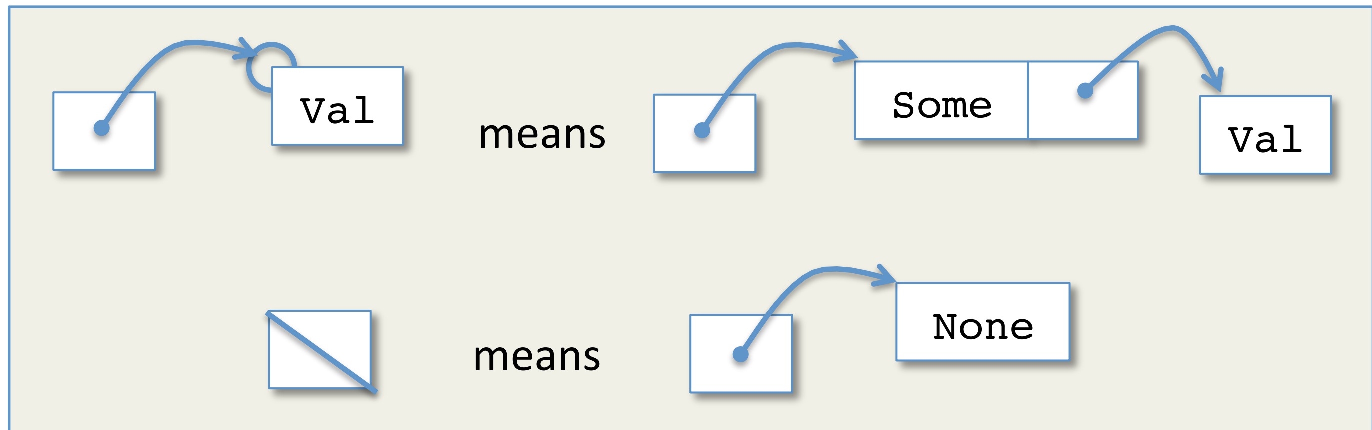


A queue with one element

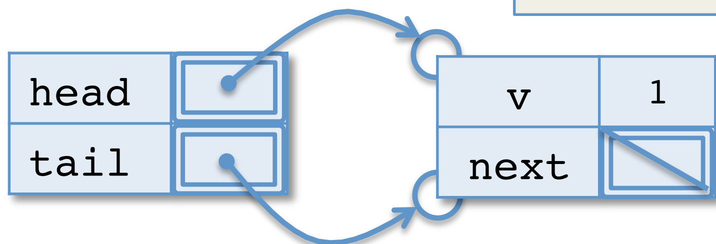


A queue with two elements

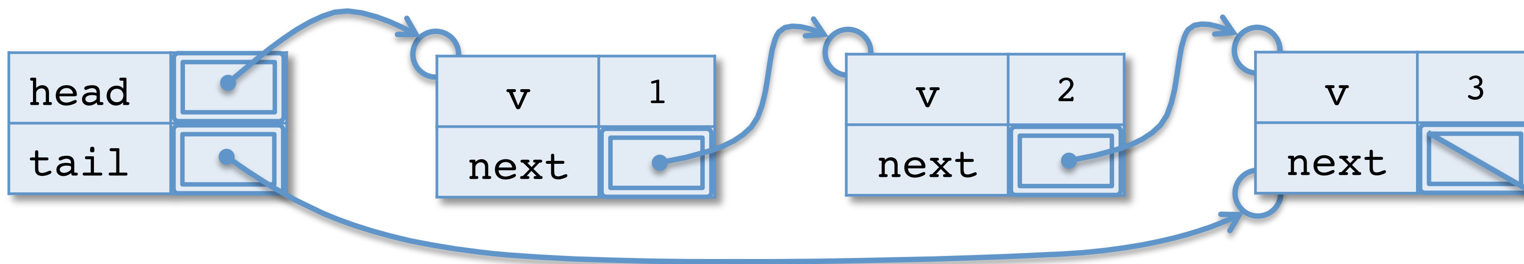
Visual Shorthand: Abbreviating Options



An empty queue

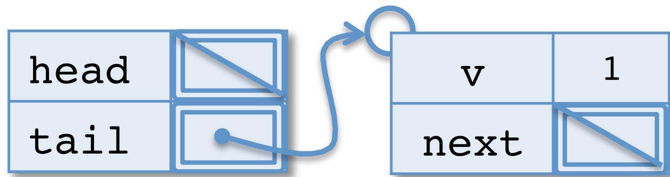


A queue with one element

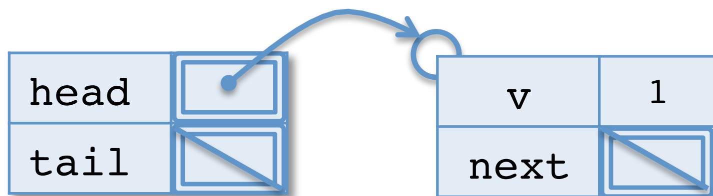


A queue with three elements

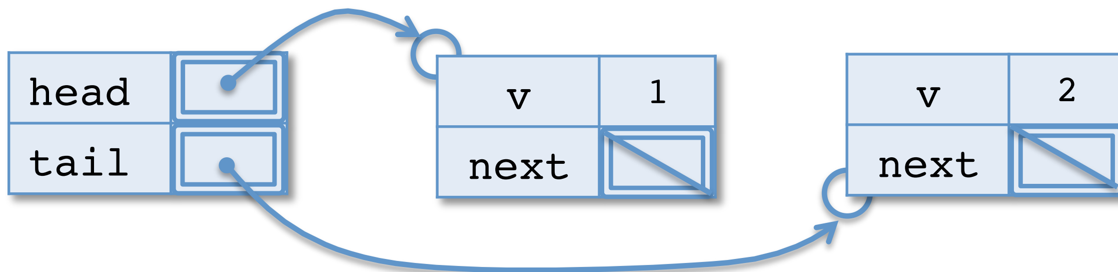
"Bogus" values of type `int queue`



head is None, tail is Some n

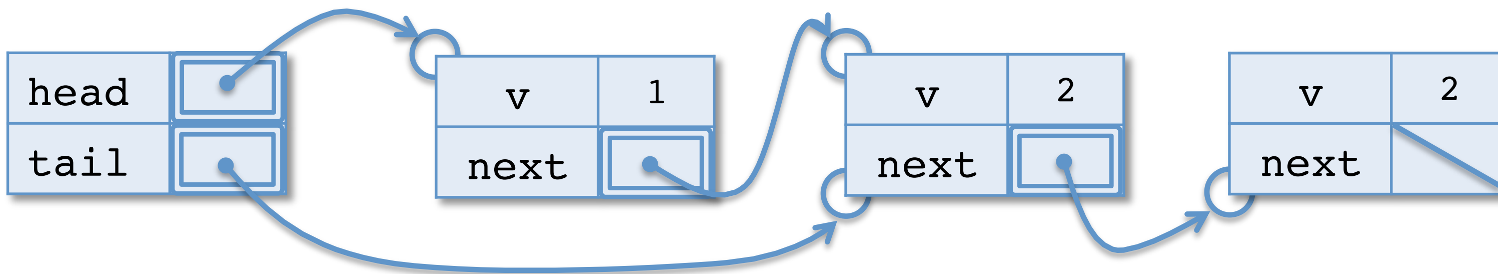


head is Some, tail is None

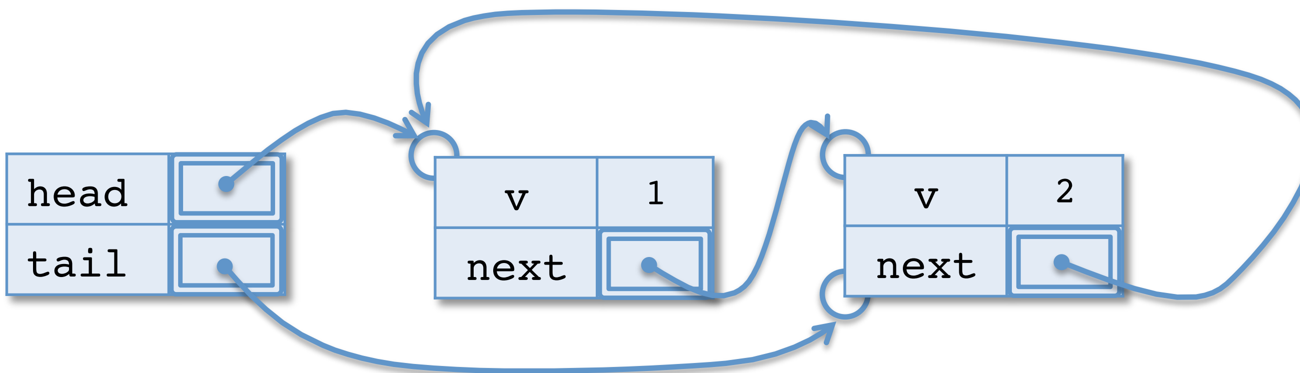


tail is not reachable from the head

More bogus `int` queues



tail doesn't point to the last element of the queue



the links contain a *cycle*!

Linked Queue Invariants

- Just as we imposed some restrictions on which trees count as legitimate Binary Search Trees, Linked Queues must also satisfy some *invariants*:

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`

- We can check that these properties rule out all of the "bogus" examples.
- Each queue operation may assume that these invariants hold of its inputs, and must ensure that the invariants hold when it's done.