Programming Languages and Techniques (CIS120)

Lecture 15

Feb 13, 2013

Linked Queues II

Announcements

- Homework 5 (queues) on the web
  - due Thursday, Feb 21st at midnight
- Midterm 1 will be in class on Friday, February 15th
  - ROOMS:

Towne 100 (here)	last names: A – K
Cohen G17	last names: L – Z

  - TIME: 11:00a.m. sharp, 50 mins
  - Covers up to Feb 6th
- Labs today and tomorrow are review
- Review session 6-8PM tonight Wu & Chen

Queues

First-in first-out mutable data structures

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 tail of the queue *)
  val enq : 'a -> 'a queue -> unit

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

end
```

Data Structure for Mutable Queues

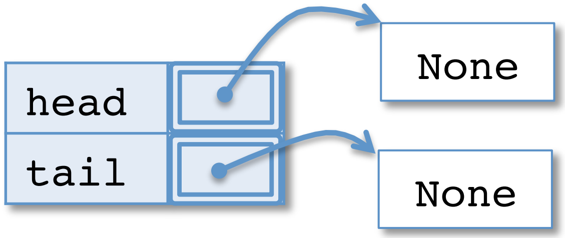
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

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

There are two parts to a mutable queue:

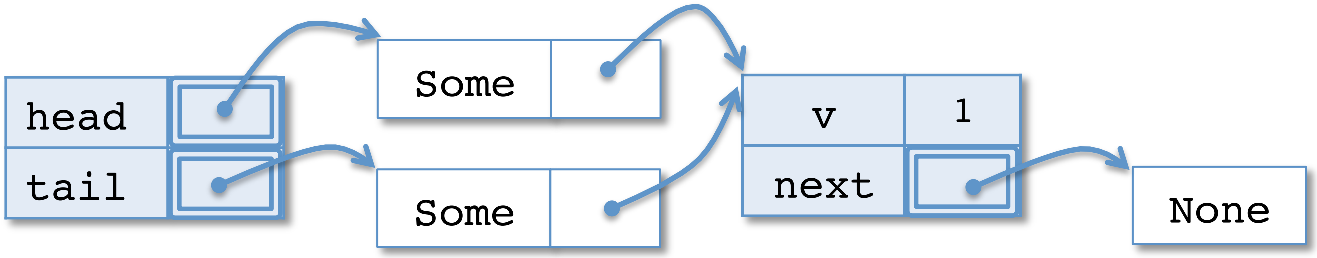
- the “internal nodes” of the queue with links from one to the next
- the head and tail references themselves

All of the references are options so that the queue can be empty (and so that the links can terminate).

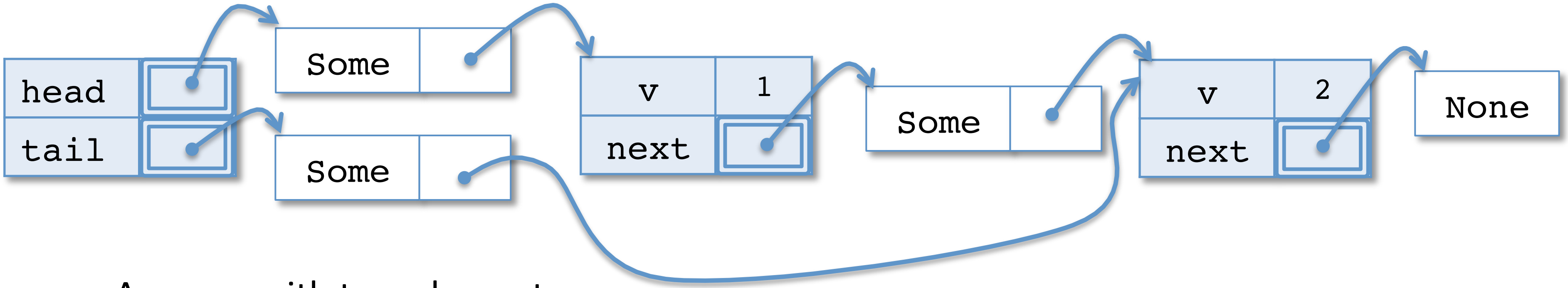
Queues in the Heap



An empty queue

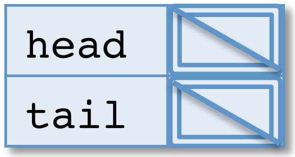


A queue with one element

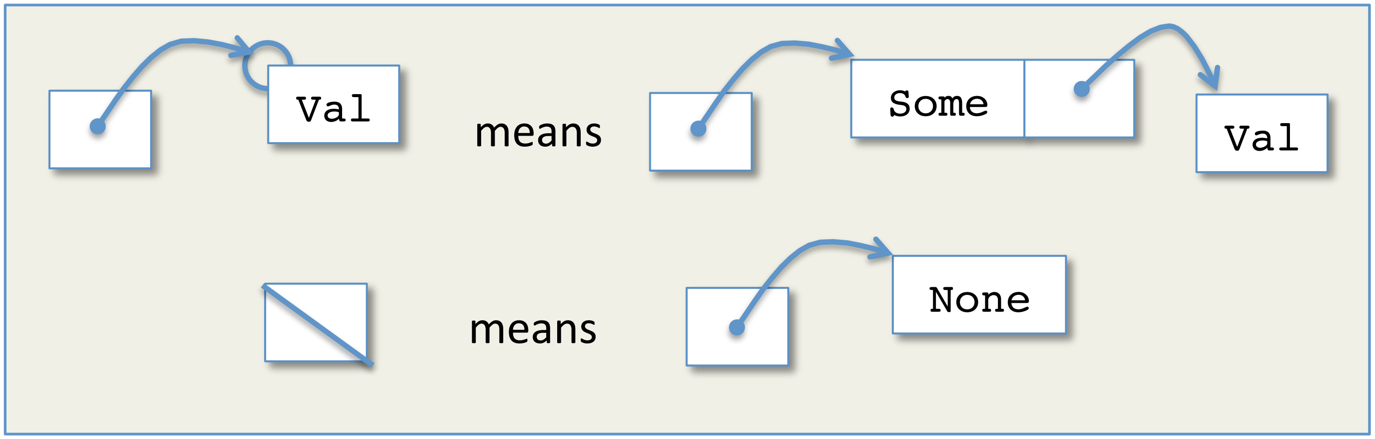


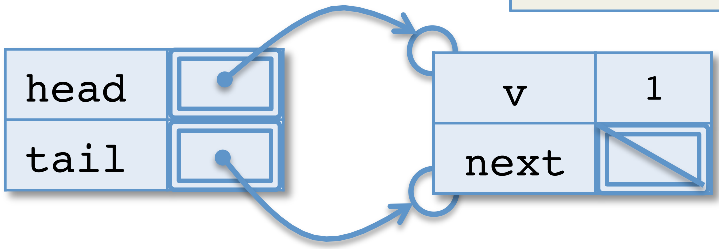
A queue with two elements

Visual Shorthand: Abbreviating Options

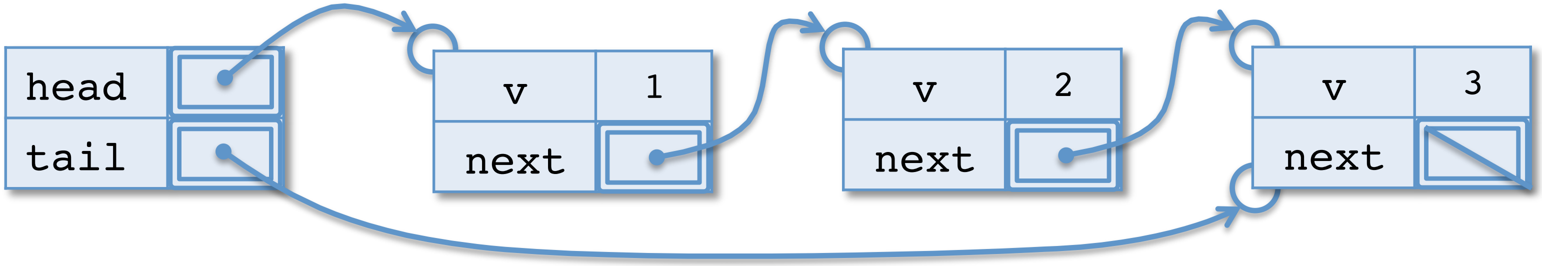


An empty queue





A queue with one element



A queue with three elements

Linked Queue Invariants

- Just as we imposed some restrictions on which trees are 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)

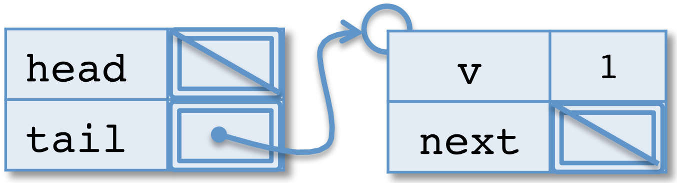
or

(2) `head` is `Some n1`, `tail` is `Some n2` and

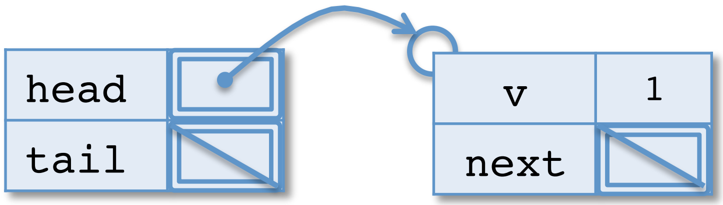
- `n2` is reachable from `n1` by following 'next' pointers
- `n2.next` is `None`

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

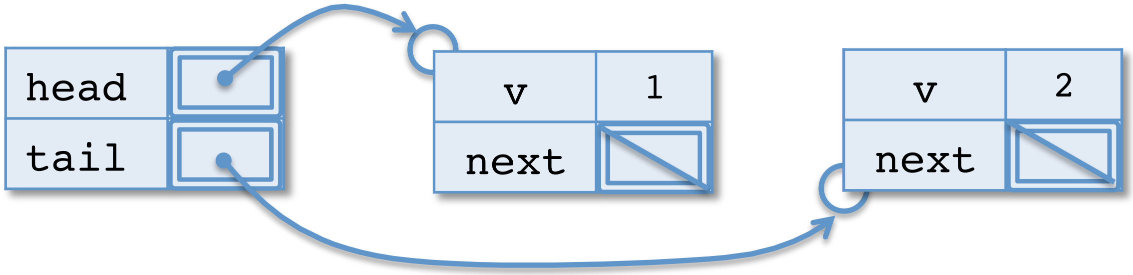
“Bogus” values of type `int queue`



head is None, tail is Some n

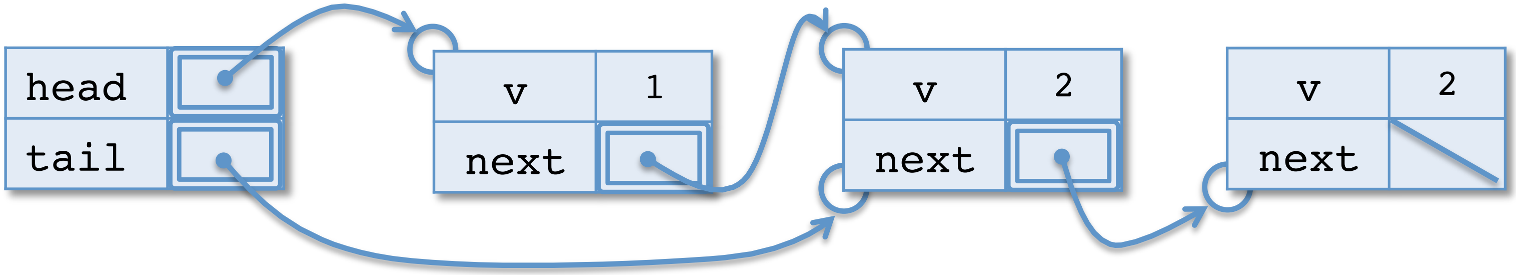


head is Some, tail is None

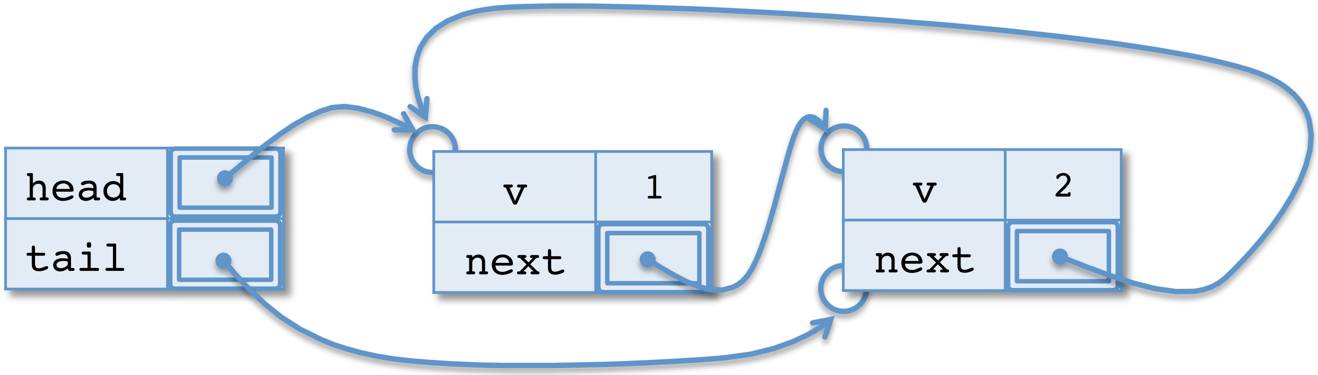


tail is not reachable from the head

More bogus `int` queues



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



the links contain a *cycle*!

Implementing Linked Queues

LinkedQ.ml

create and is_empty

```
(* create an empty queue *)
let create () : 'a queue =
    { head = None;
      tail = None }


(* determine whether a queue is empty *)
let is_empty (q:'a queue) : bool =
    q.head = None
```

- `create` *establishes* the queue invariants
  - both head and tail are None
- `is_empty` *assumes* the queue invariants
  - it doesn’t have to check that q.tail is None

enq

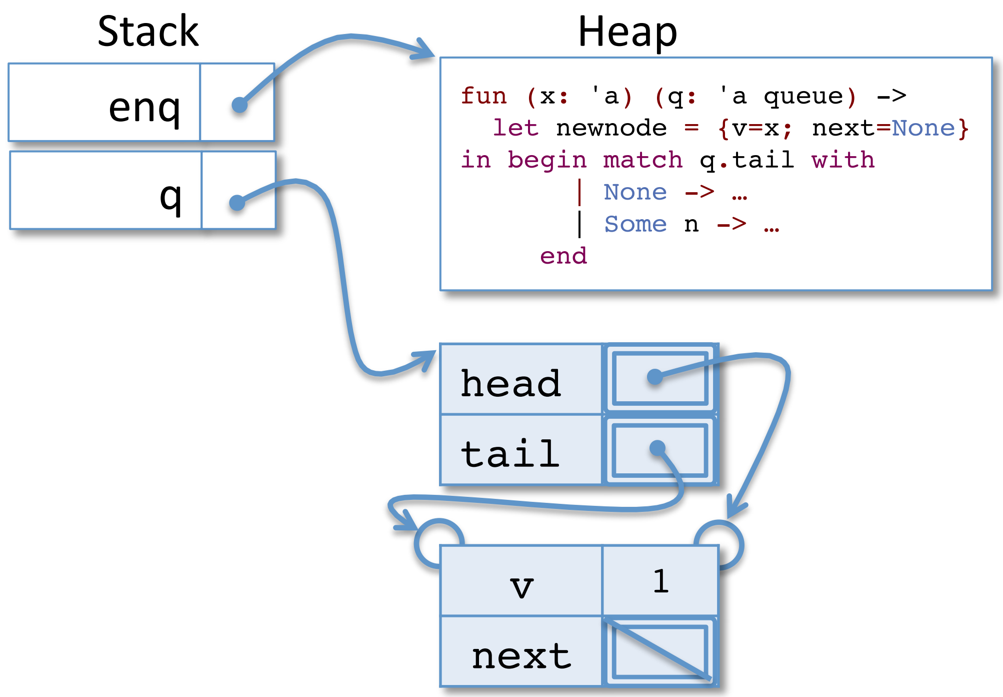
```
(* add an element to the tail of a queue *)
let enq (x: 'a) (q: 'a queue) : unit =
   let newnode = {v=x; next=None} in
   begin match q.tail with
     | None ->
         (* Note that the invariant tells us
           that q.head is also None *)
         q.head <- Some newnode;
         q.tail <- Some newnode
     | Some n ->
         n.next <- Some newnode;
         q.tail <- Some newnode
   end
```

- The code for enq is informed by the queue invariant:
  - either the queue is empty, and we just update head and tail, or
  - the queue is non-empty, in which case we have to "patch up" the "next" link of the old tail node to maintain the queue invariant.

Calling Enq on a non-empty queue

Workspace

```
enq 2 q
```

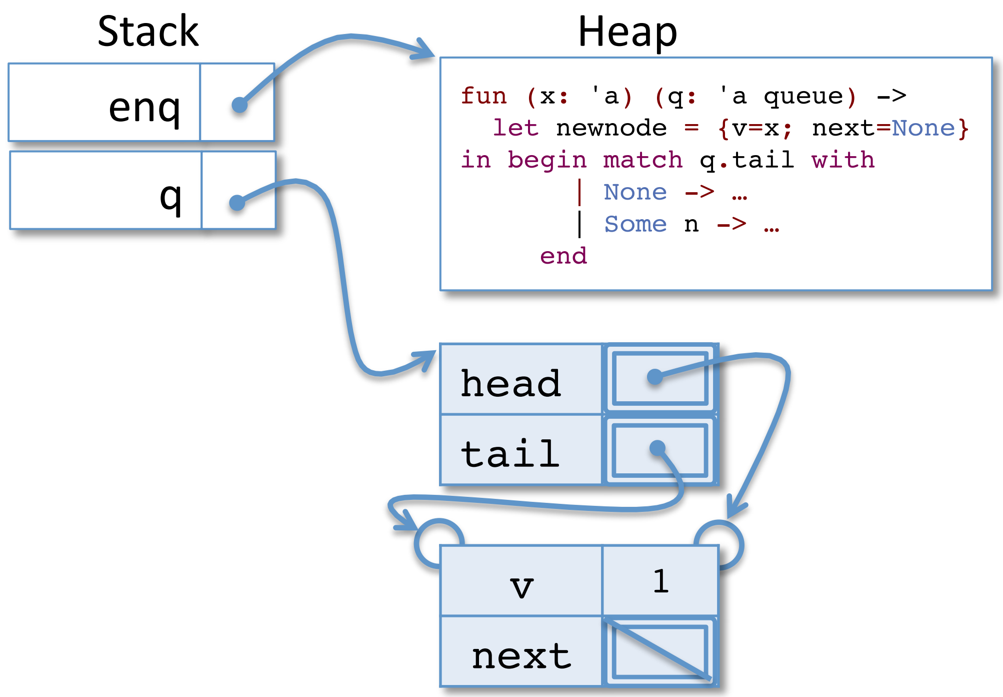


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

Calling Enq on a non-empty queue

Workspace

```
enq 2 q
```



Stack

enq

q

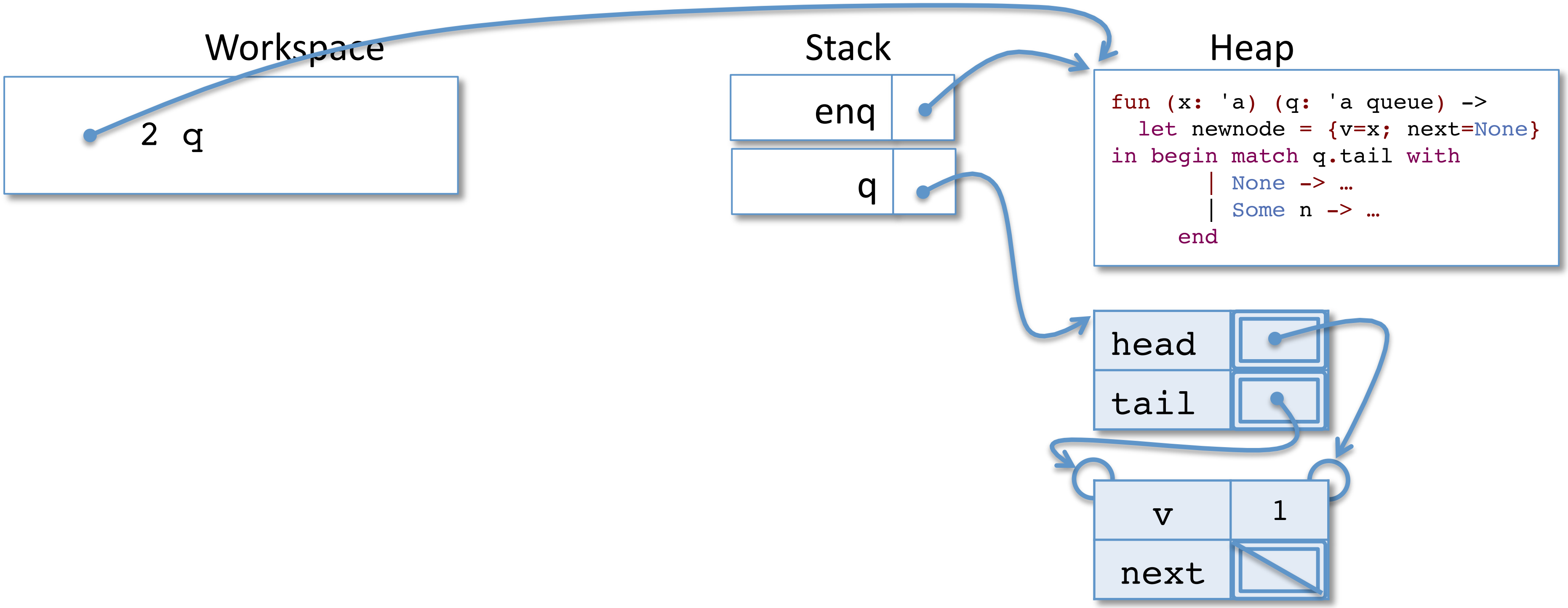
Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

head	
tail	

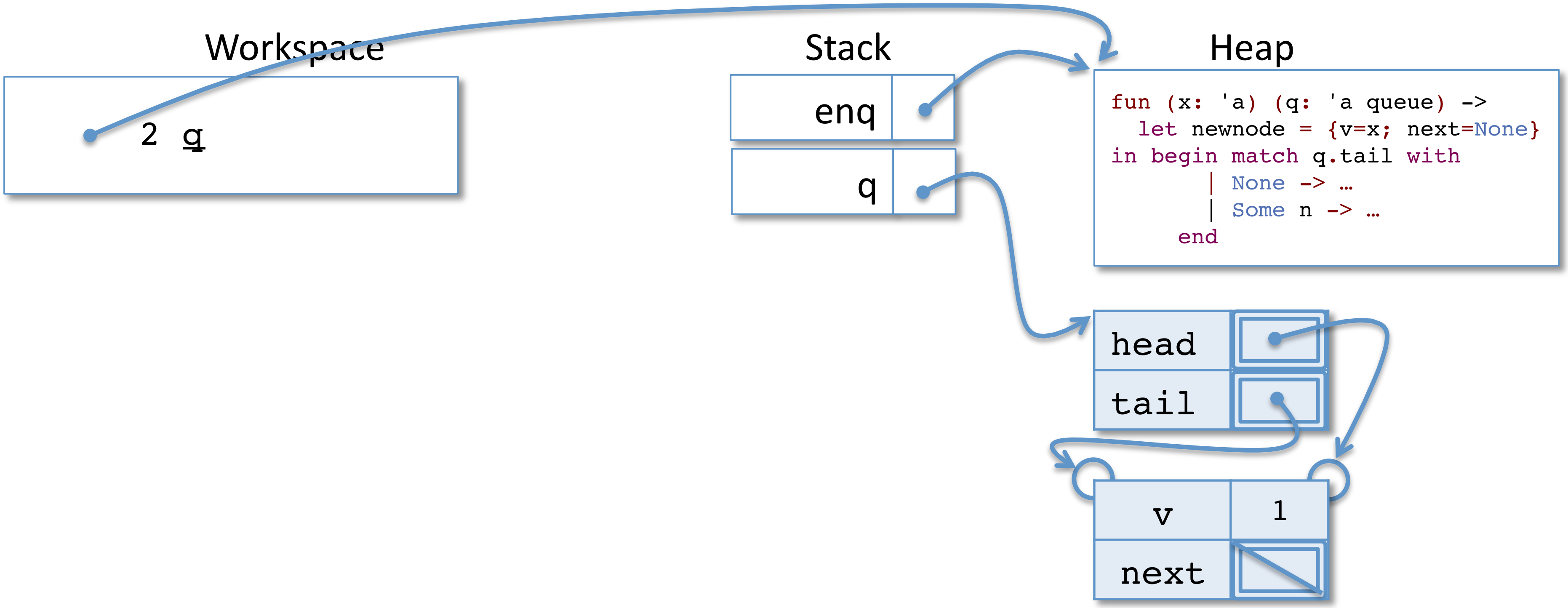
v	1
next	

Calling Enq on a non-empty queue



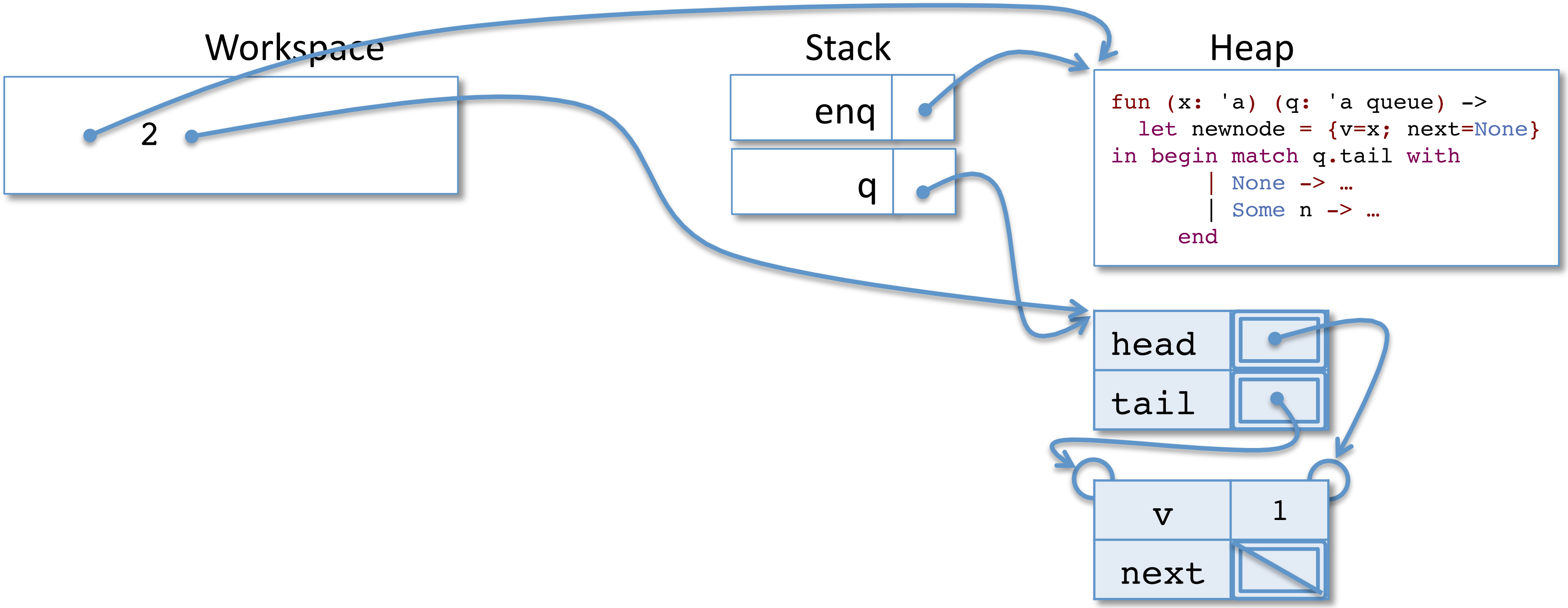
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

Calling Enq on a non-empty queue



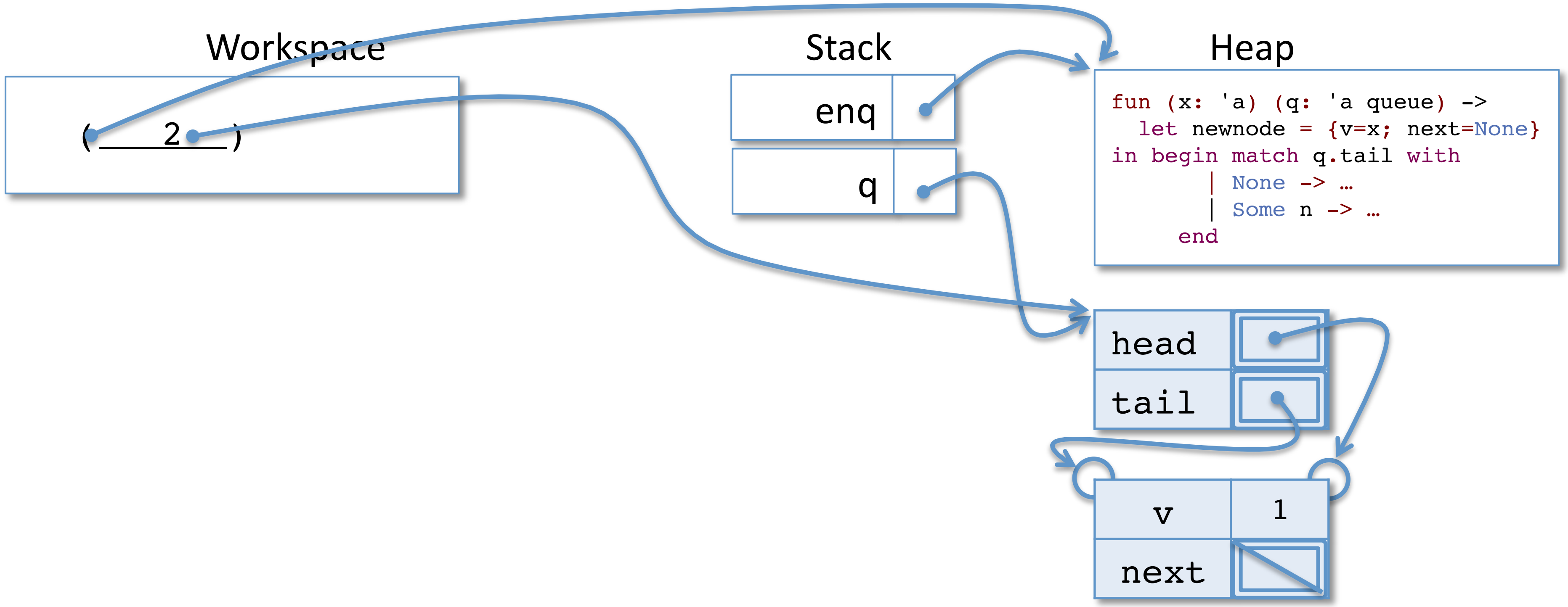
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

Calling Enq on a non-empty queue



```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

Calling Enq on a non-empty queue



Workspace

(2)

Stack

enq

q

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

head

tail

v

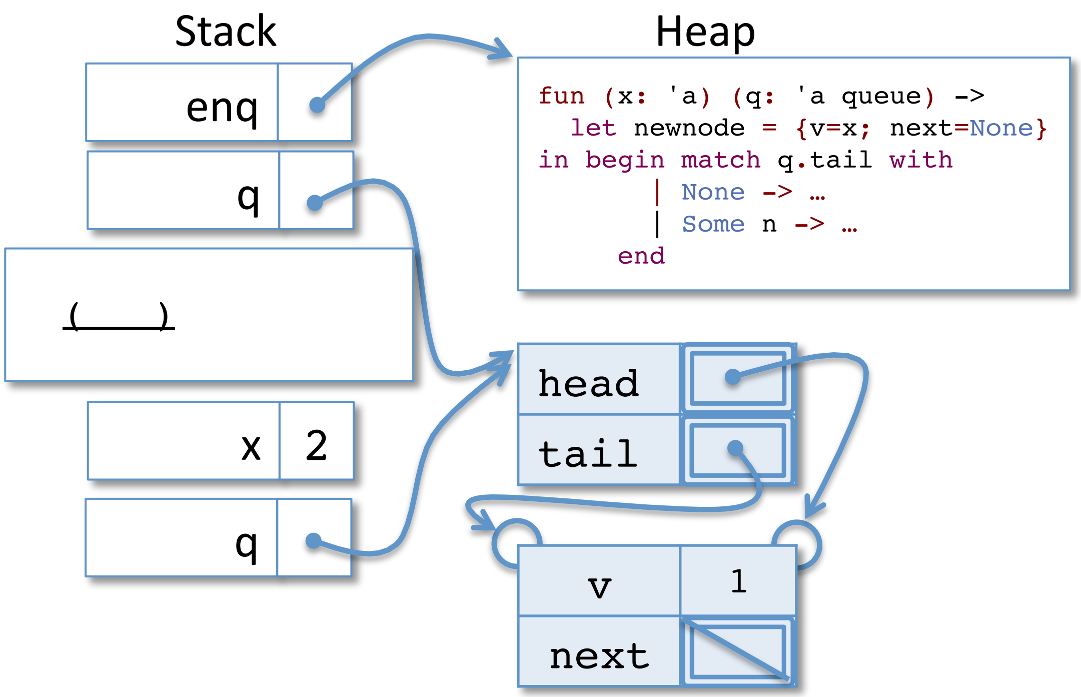
1

next

Calling Enq on a non-empty queue

Workspace

```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

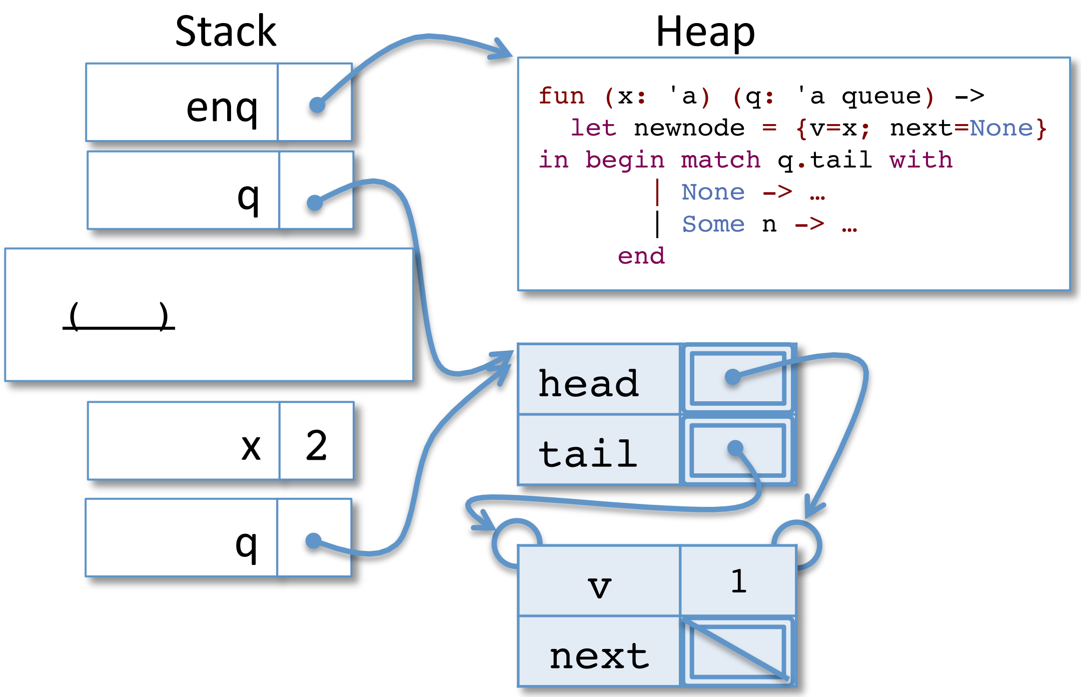


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

Calling Enq on a non-empty queue

Workspace

```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

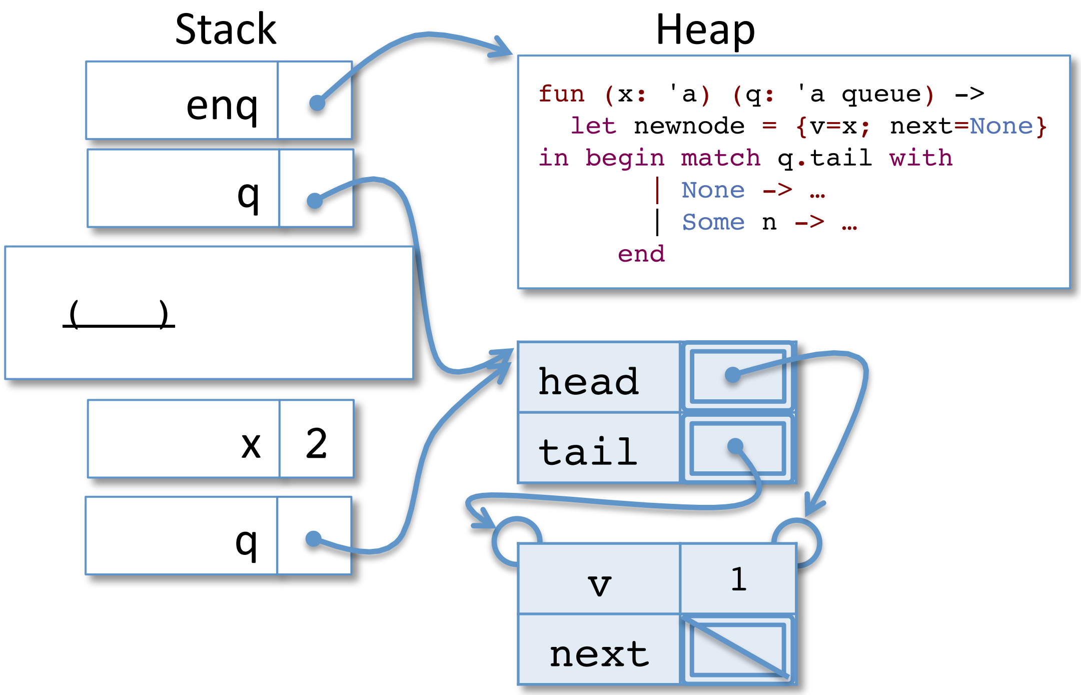


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
    end
```

Calling Enq on a non-empty queue

Workspace

```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

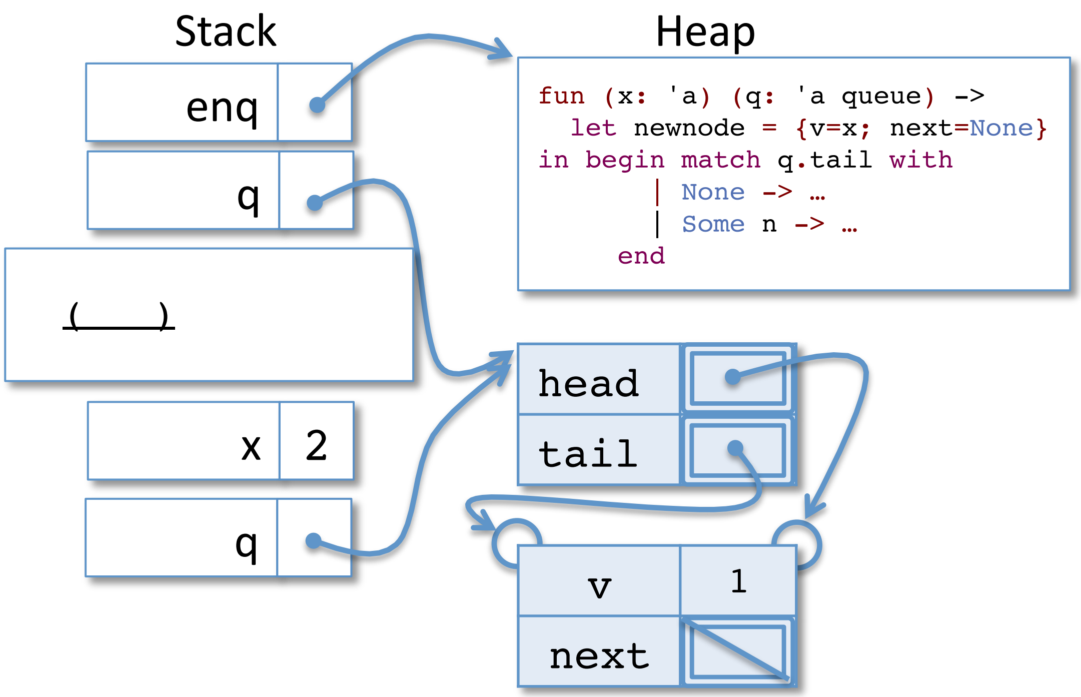


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
          | None -> …
          | Some n -> …
     end
```

Calling Enq on a non-empty queue

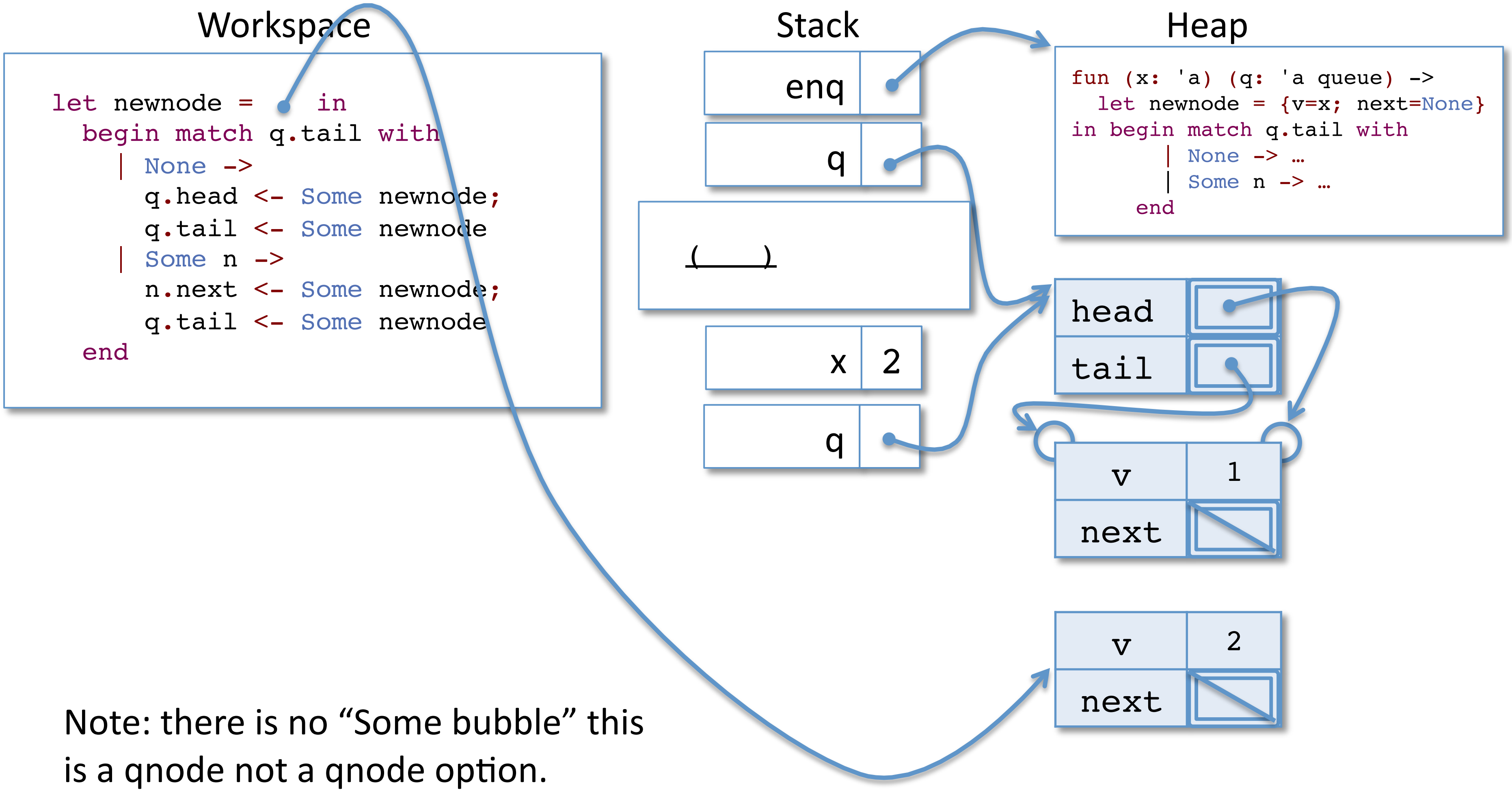
Workspace

```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
    end
```

Calling Enq on a non-empty queue



Workspace

```
let newnode =   in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

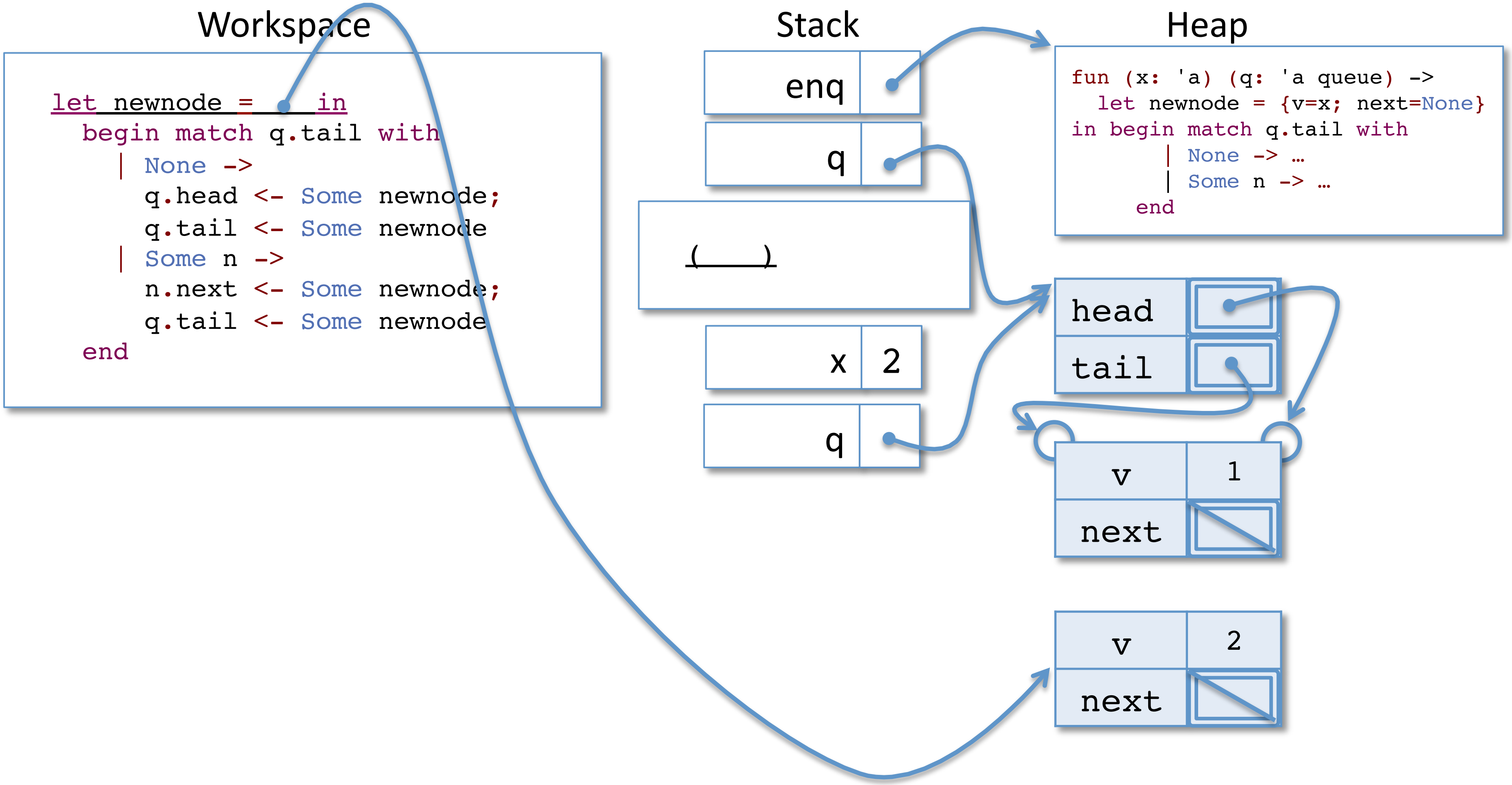
Stack

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```

Note: there is no "Some bubble" this is a qnode not a qnode option.

Calling Enq on a non-empty queue



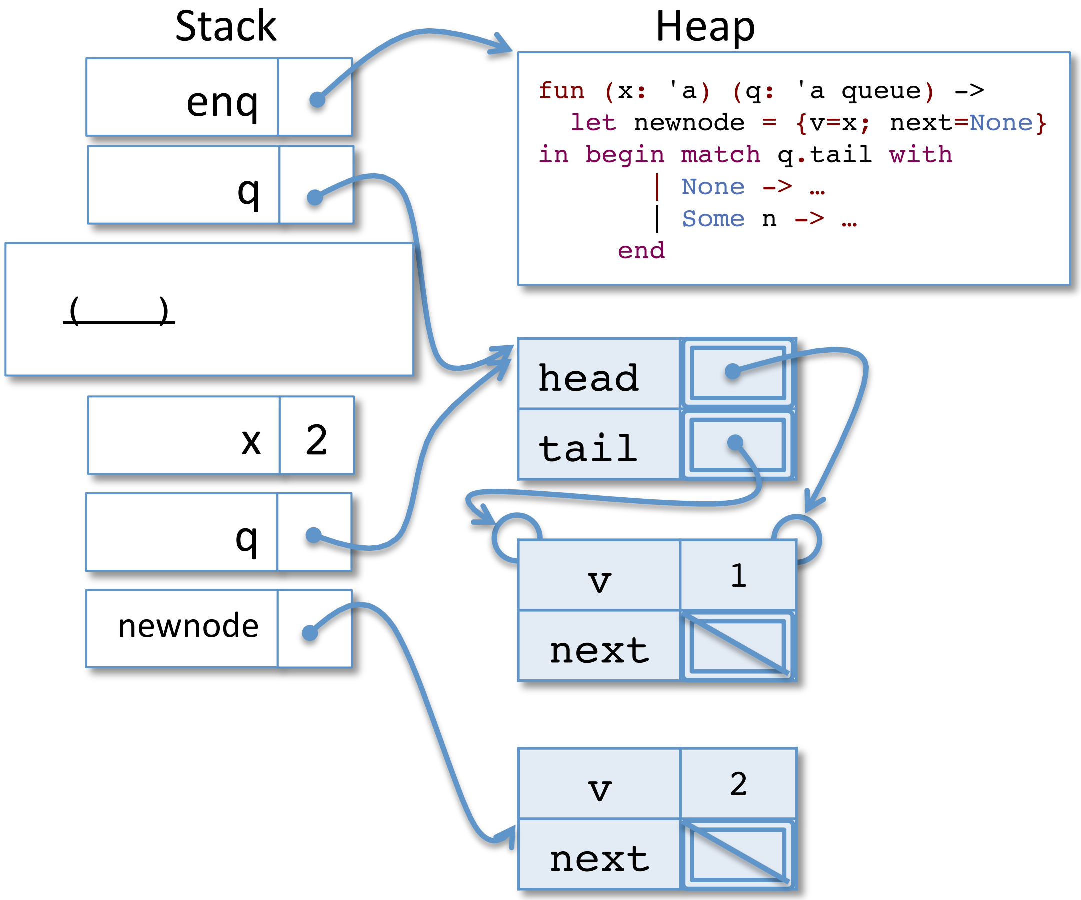
```
let newnode =    in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
          | None -> …
          | Some n -> …
      end
```

Calling Enq on a non-empty queue

Workspace

```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```

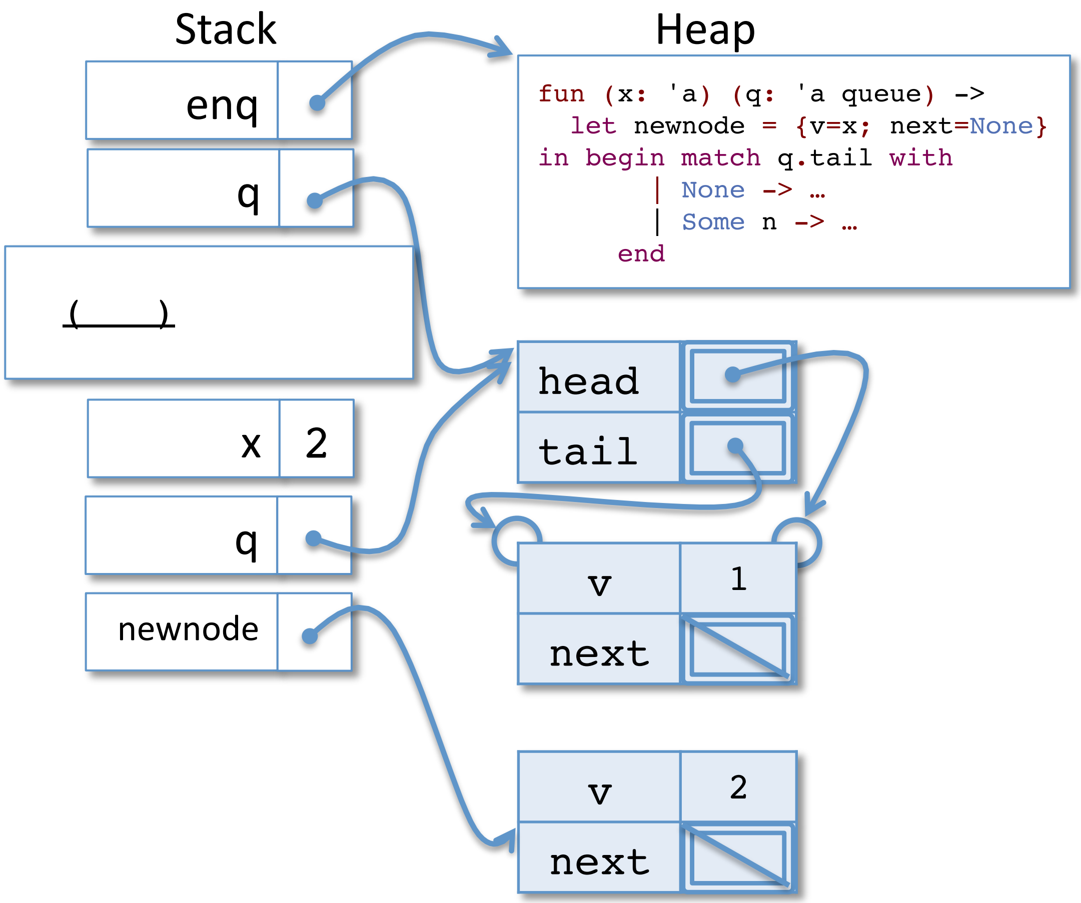


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

Calling Enq on a non-empty queue

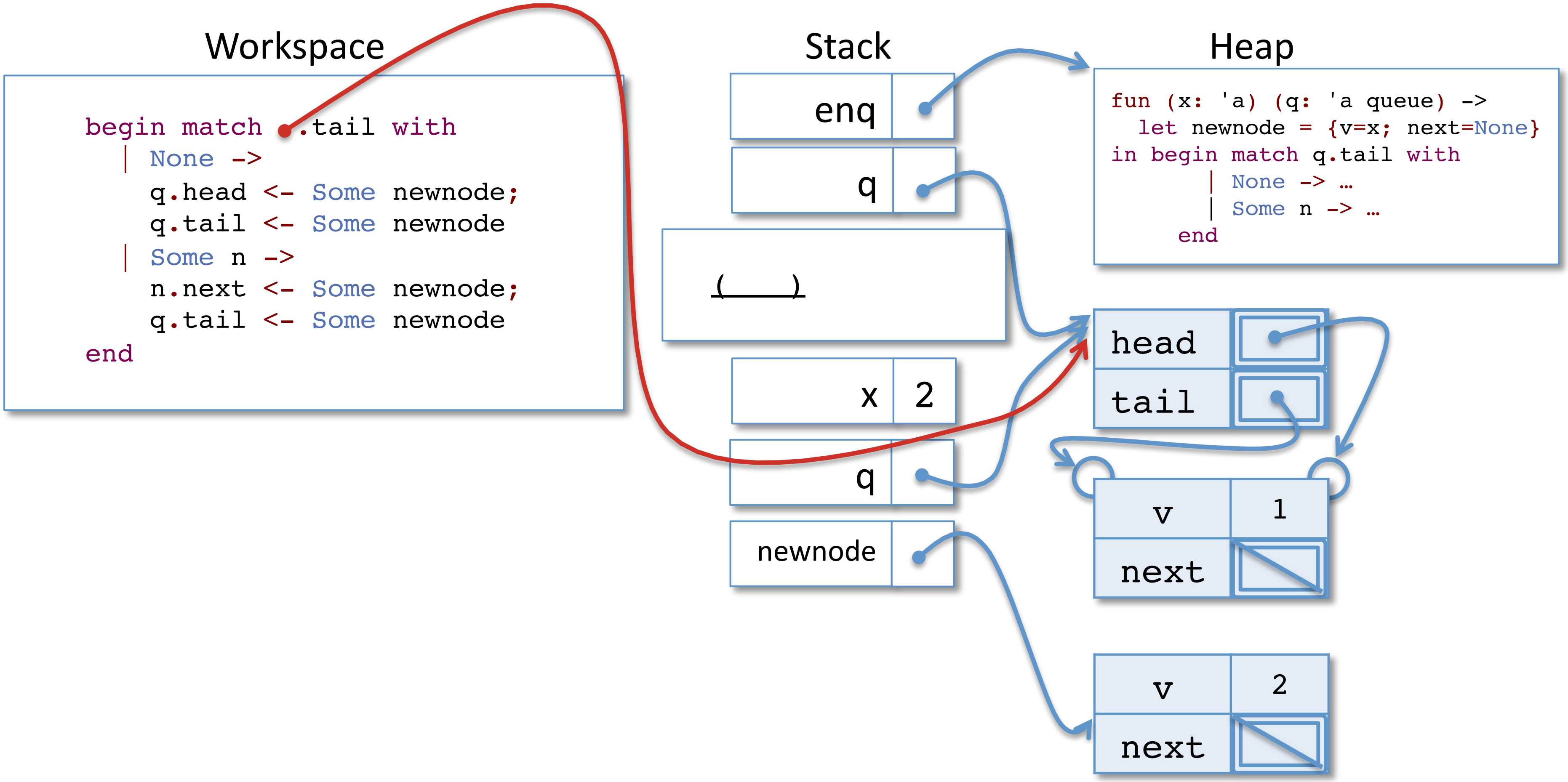
Workspace

```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
          | None -> …
          | Some n -> …
     end
```

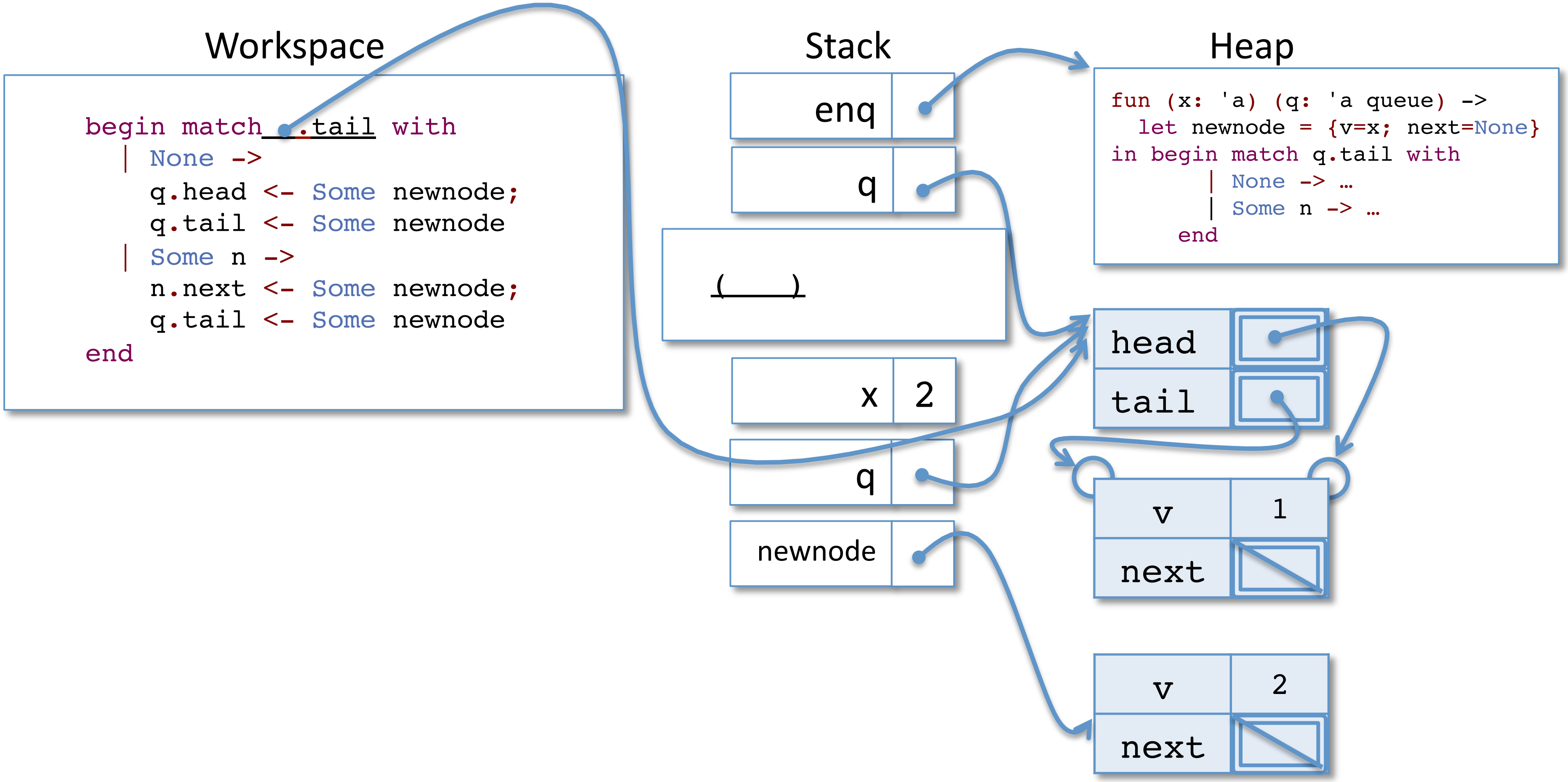
Calling Enq on a non-empty queue



```
begin match ●.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
     | None -> …
     | Some n -> …
   end
```

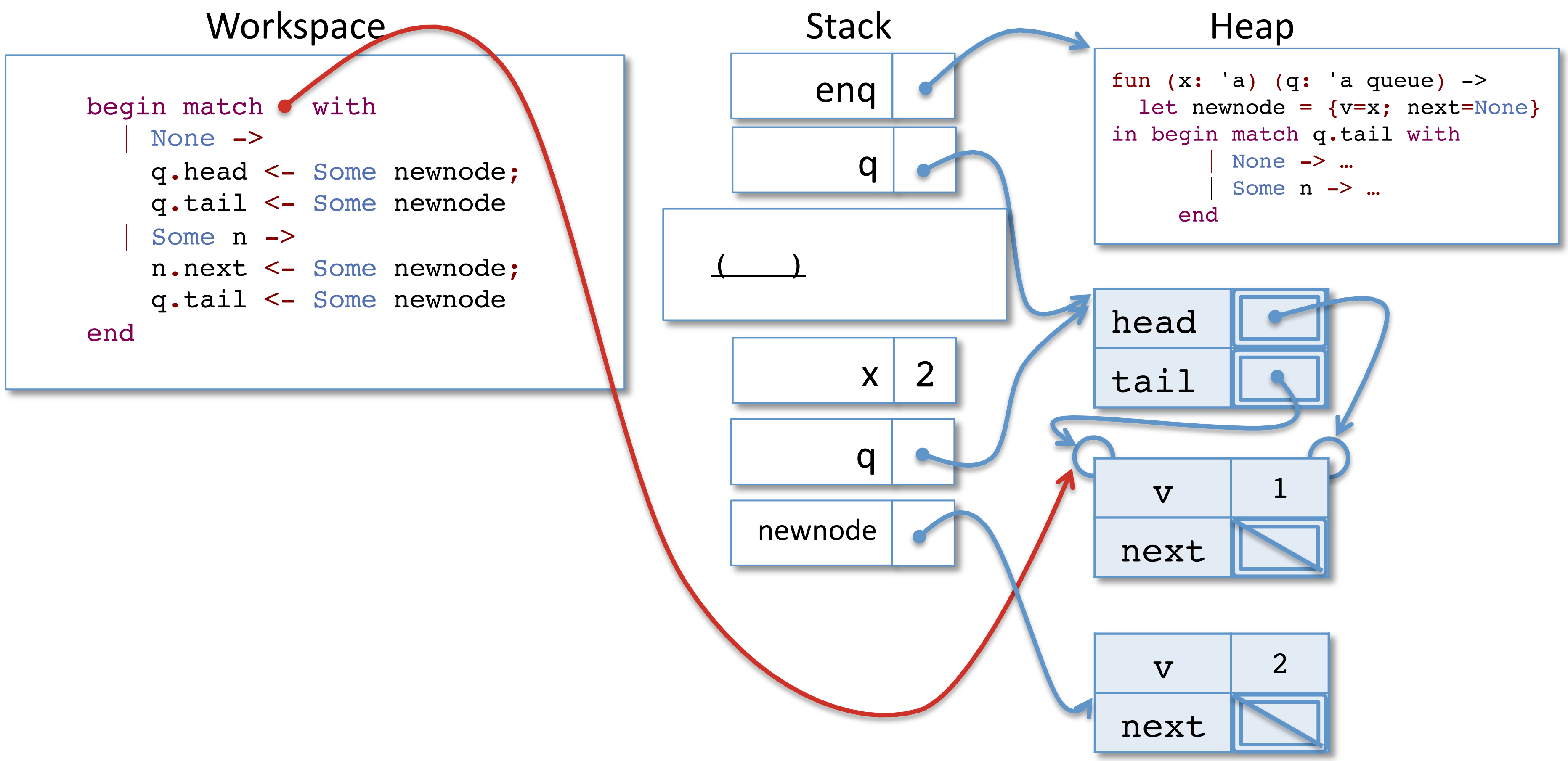
Calling Enq on a non-empty queue



```
begin match __.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

Calling Enq on a non-empty queue



Workspace

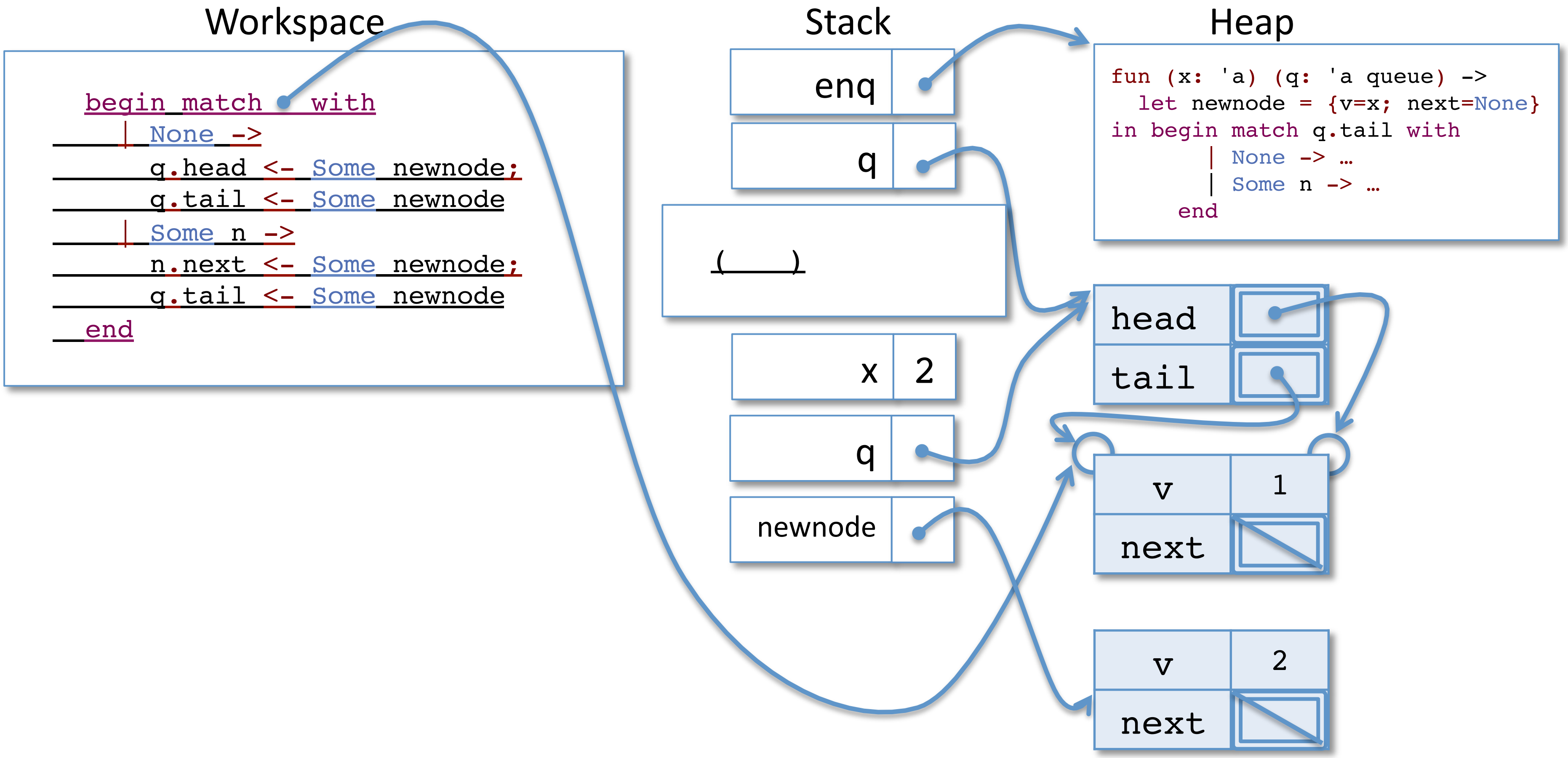
```
begin match ● with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```

Stack

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```

Calling Enq on a non-empty queue



Workspace

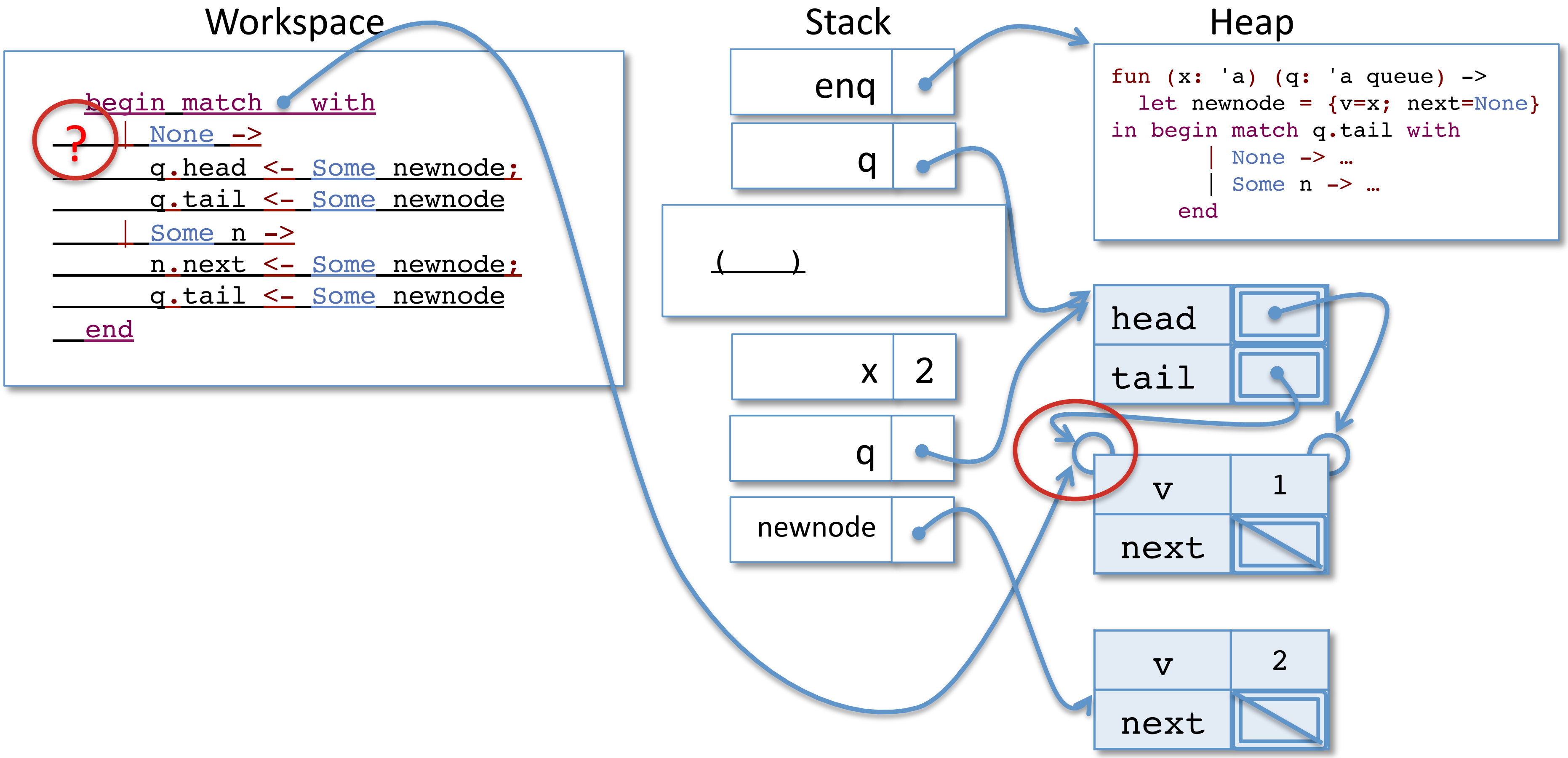
```
begin match • with
    | None ->
        q.head <- Some newnode;
        q.tail <- Some newnode
    | Some n ->
        n.next <- Some newnode;
        q.tail <- Some newnode
end
```

Stack

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

Calling Enq on a non-empty queue



Workspace

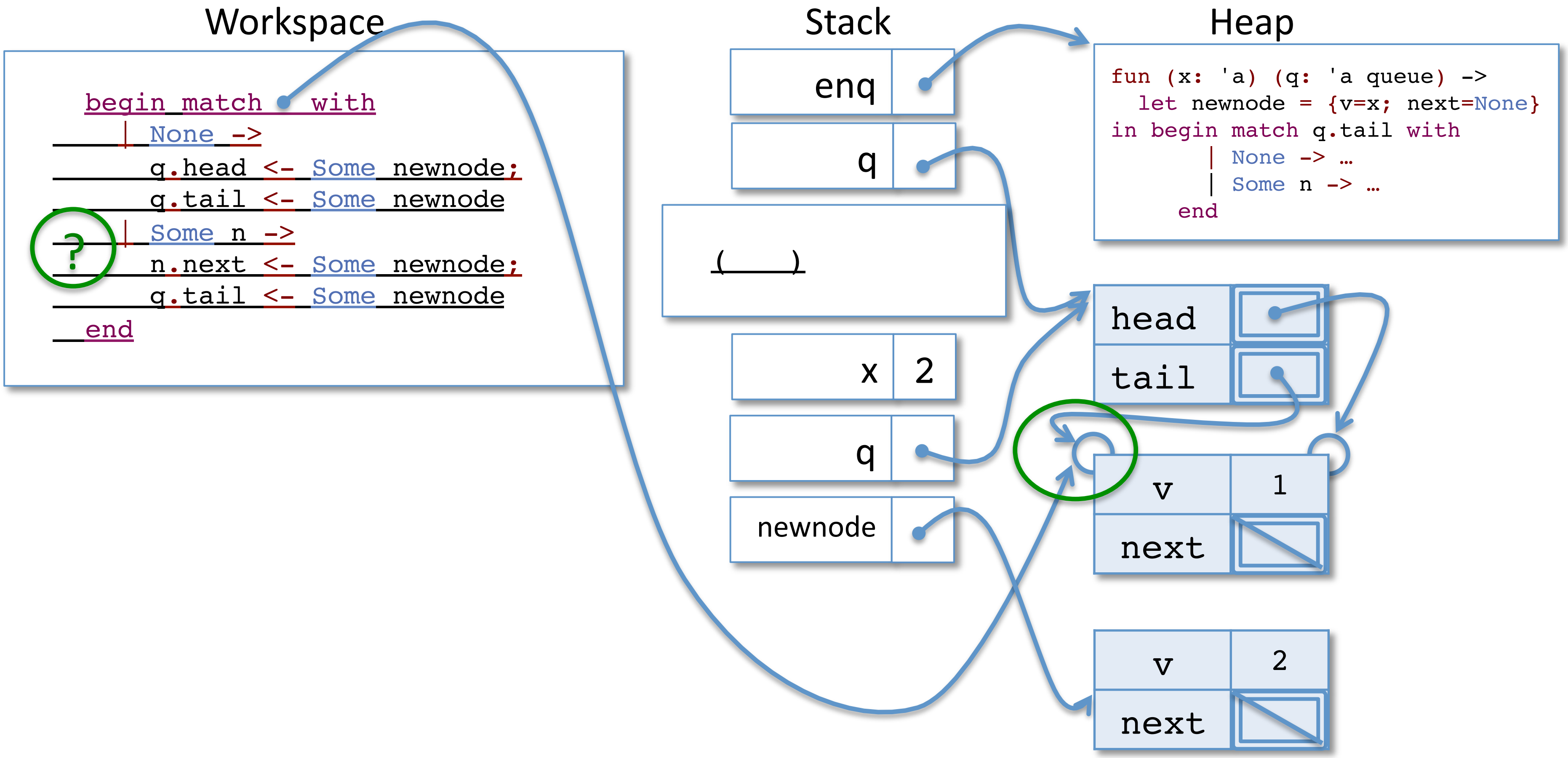
```
begin match   with
?  | None ->
       q.head <- Some newnode;
       q.tail <- Some newnode
   | Some n ->
       n.next <- Some newnode;
       q.tail <- Some newnode
 end
```

Stack

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
    end
```

Calling Enq on a non-empty queue



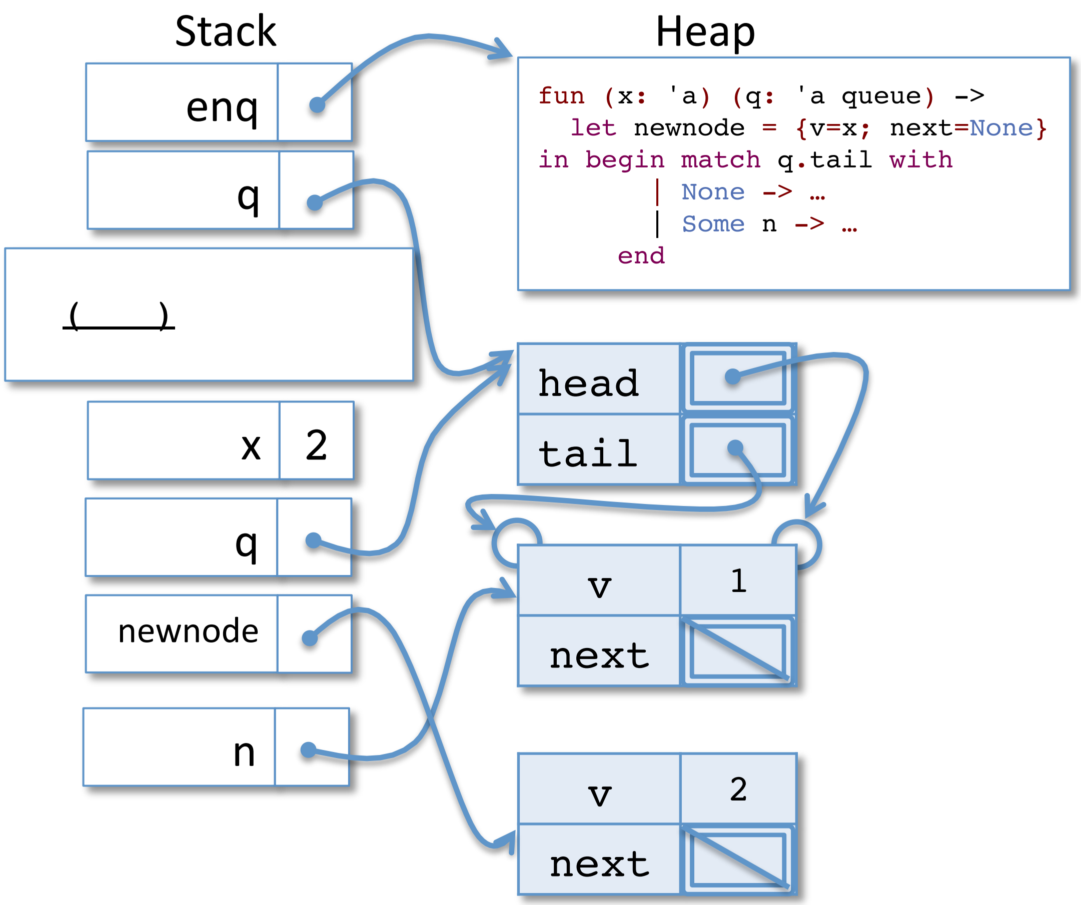
```
begin match  with
   | None ->
       q.head <- Some newnode;
       q.tail <- Some newnode
   | Some n ->
       n.next <- Some newnode;
       q.tail <- Some newnode
end
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
           | None -> …
           | Some n -> …
     end
```

Calling Enq on a non-empty queue

Workspace

```
n.next <- Some newnode;
q.tail <- Some newnode
```



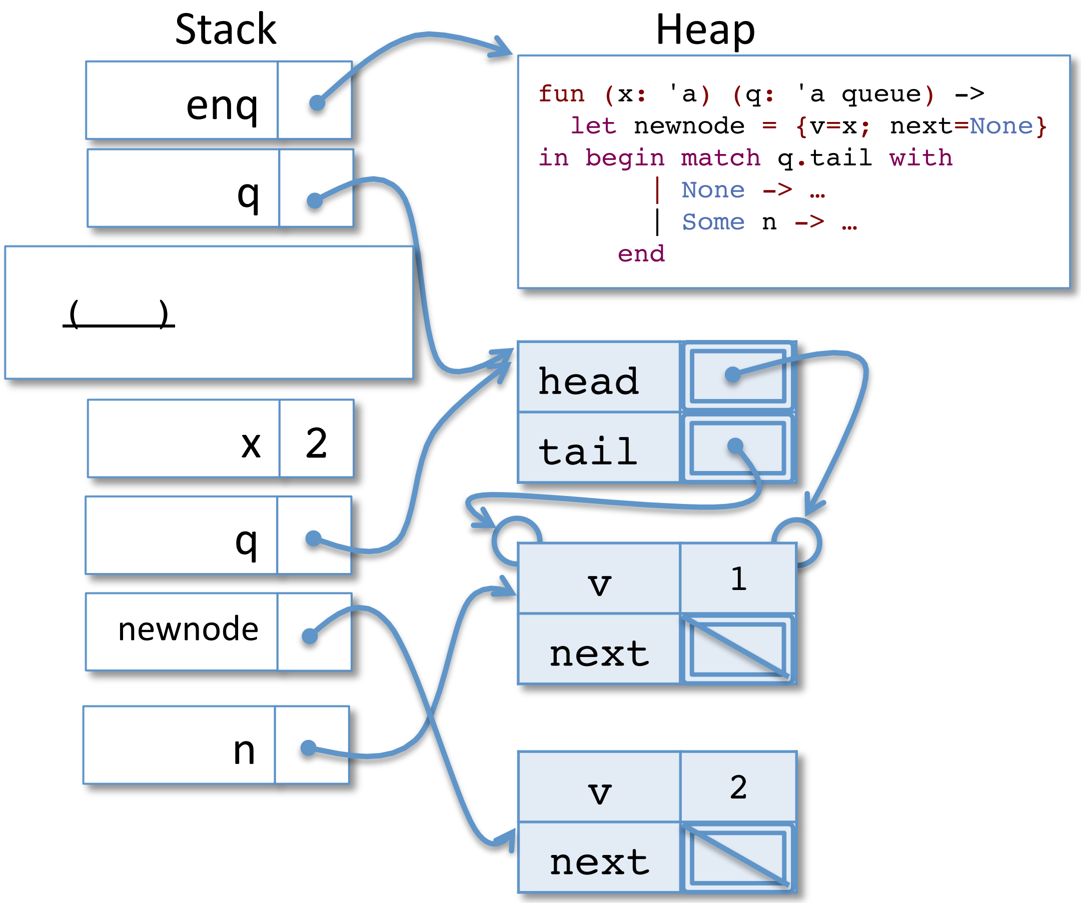
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```

Note: n points to a qnode, not a qnode option.

Calling Enq on a non-empty queue

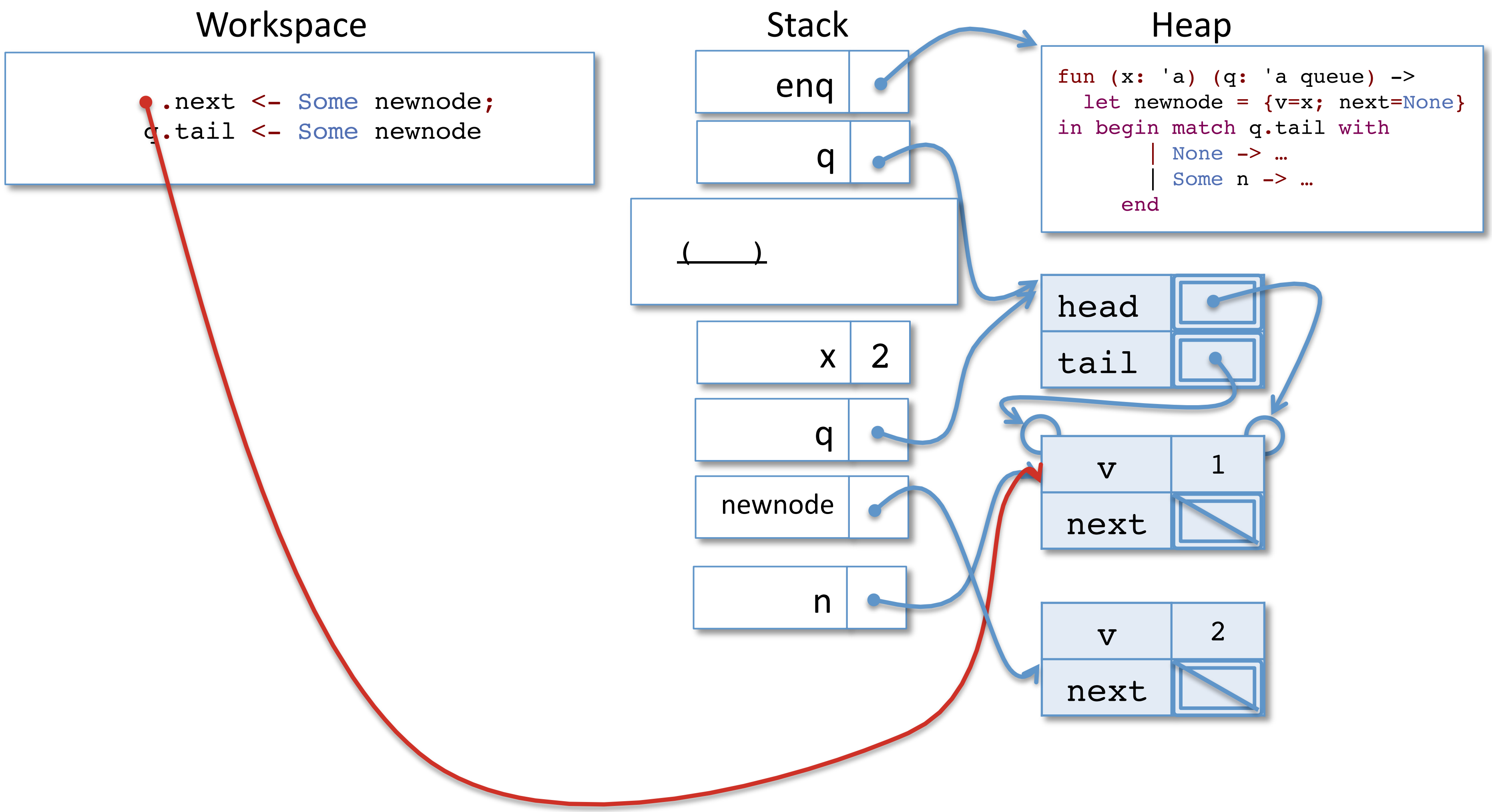
Workspace

```
n.next <- Some newnode;
q.tail <- Some newnode
```



```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
    end
```

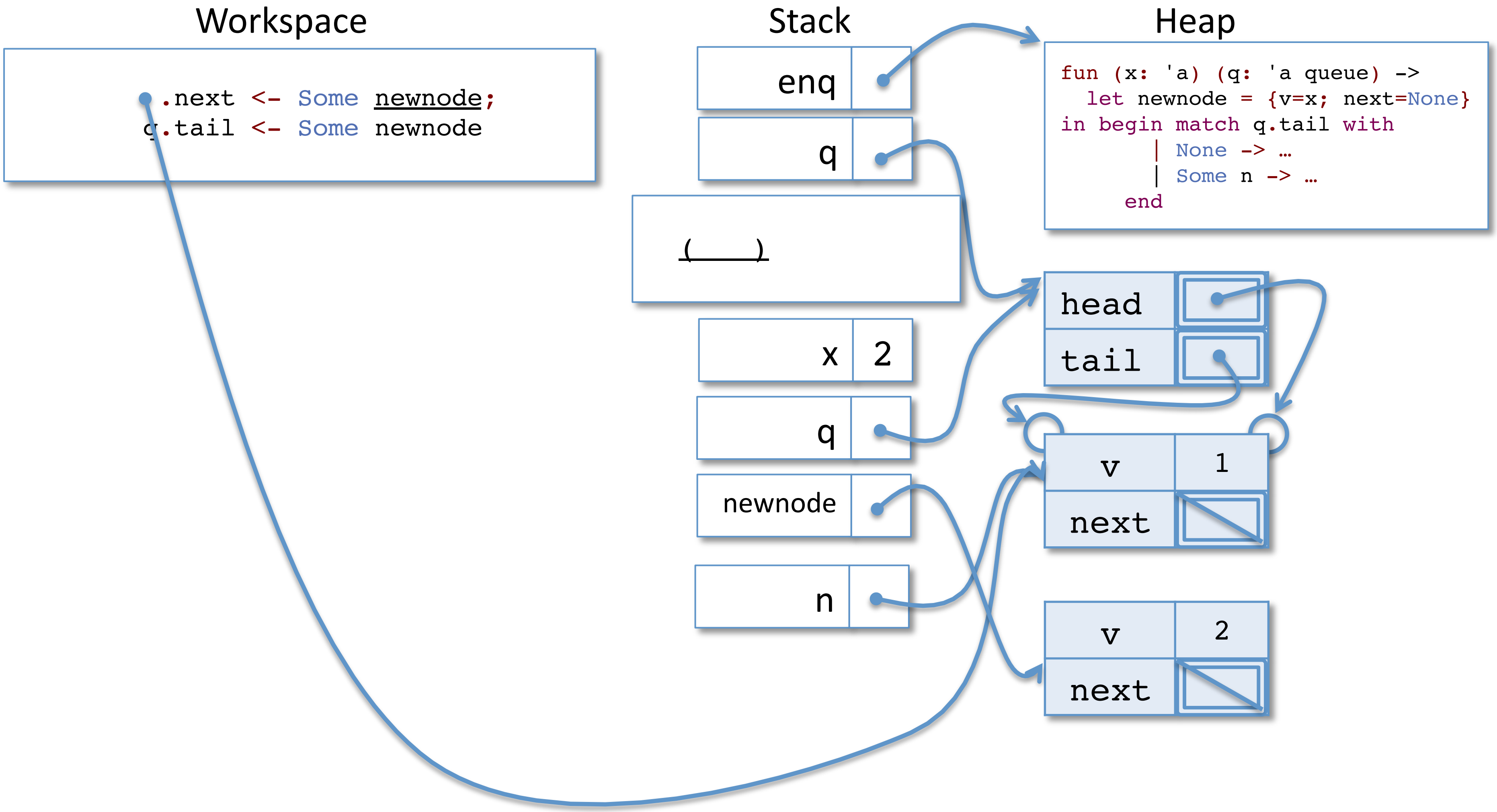
Calling Enq on a non-empty queue



```
.next <- Some newnode;
q.tail <- Some newnode
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

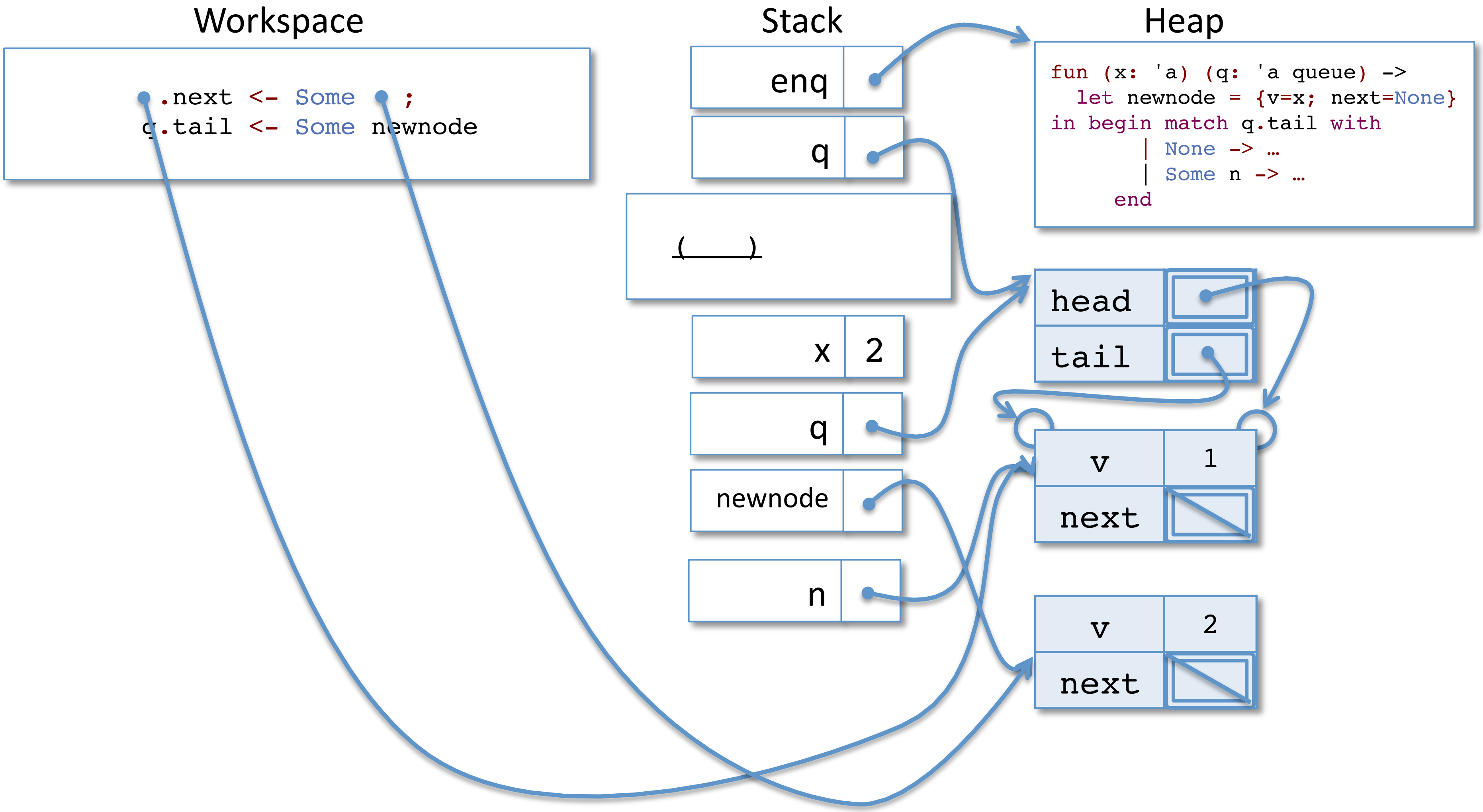
Calling Enq on a non-empty queue



```
 .next <- Some newnode;
q.tail <- Some newnode
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

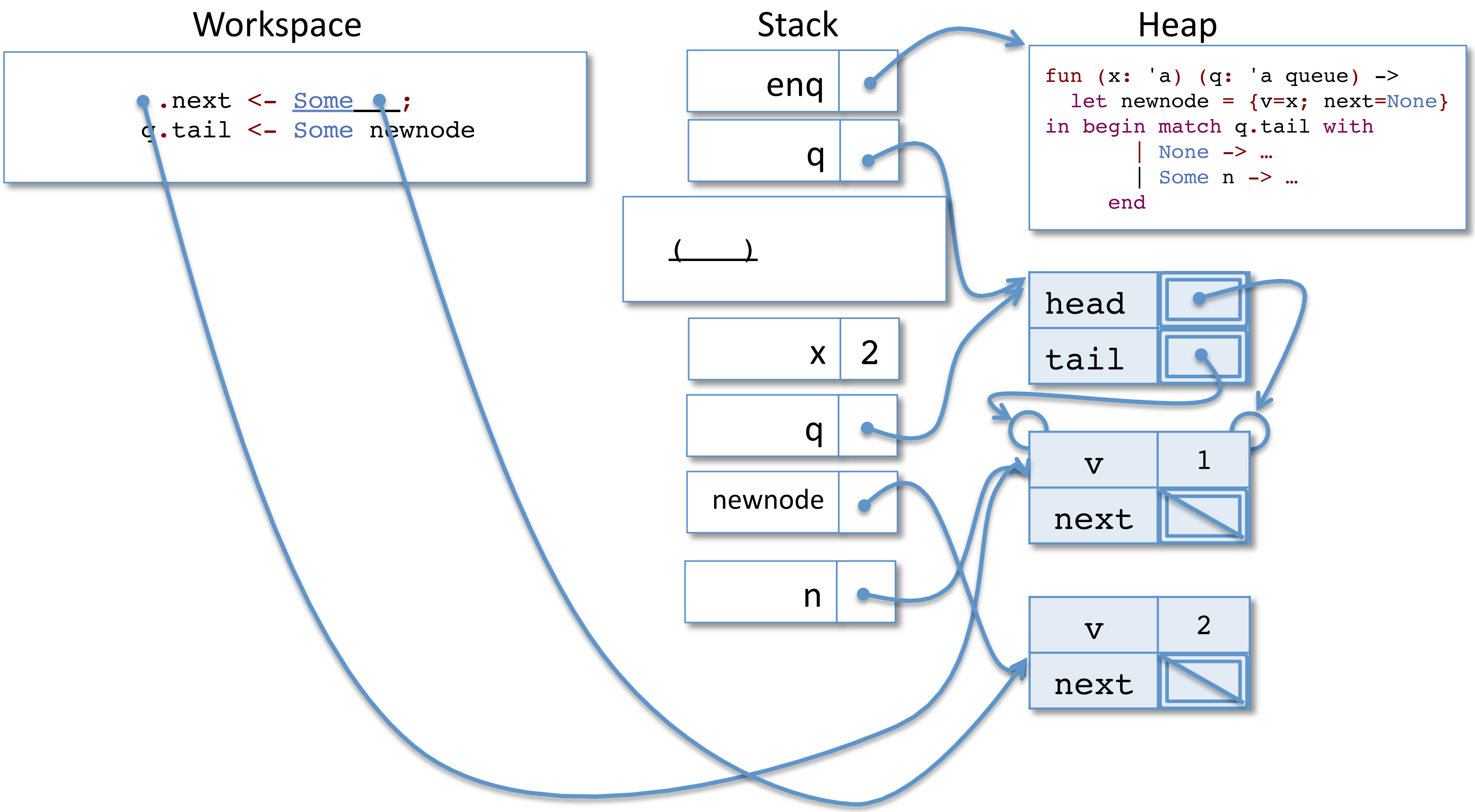
Calling Enq on a non-empty queue



```
 .next <- Some  ;
q.tail <- Some newnode
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

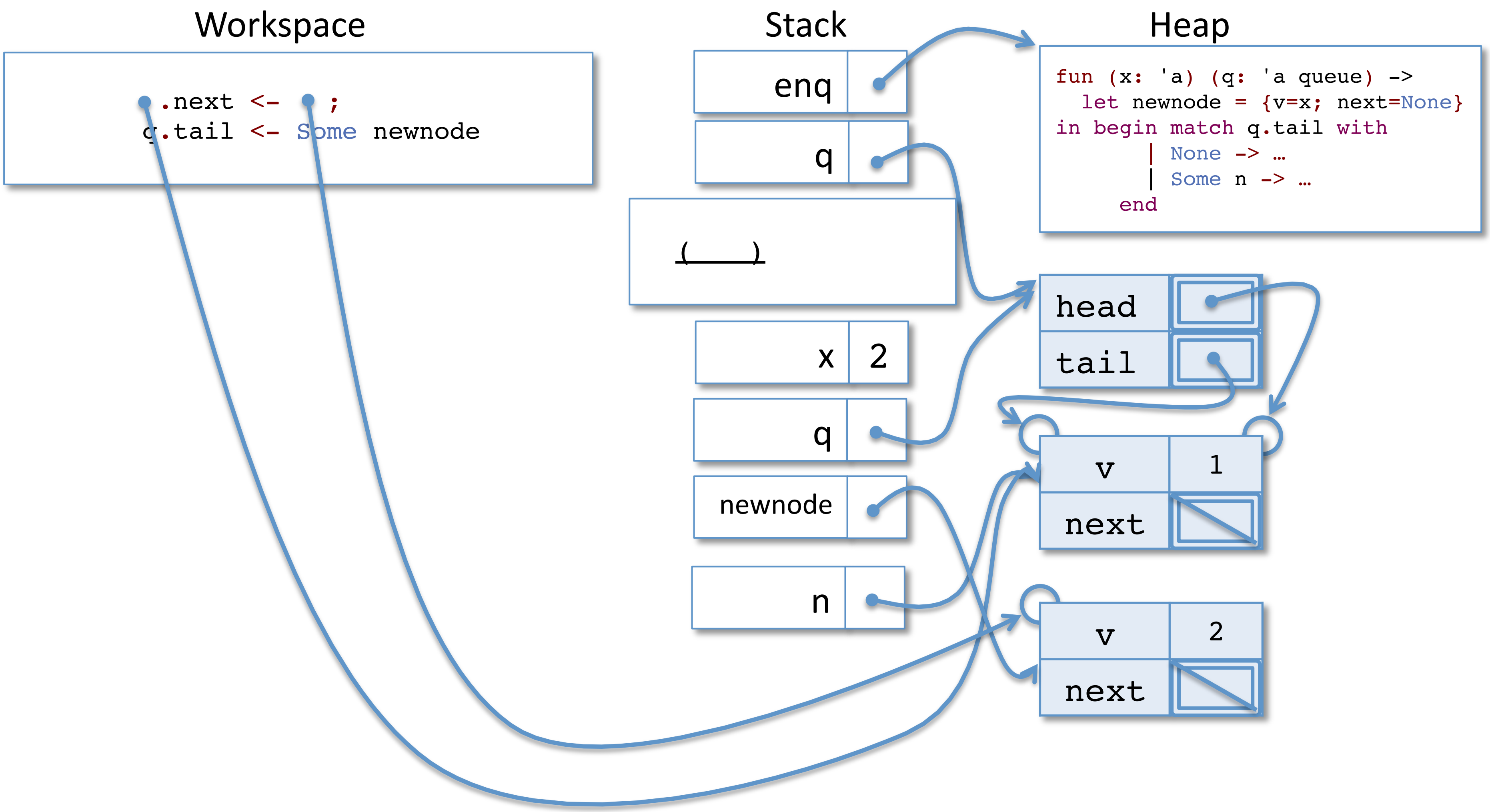
Calling Enq on a non-empty queue



```
.next <- Some ___;
q.tail <- Some newnode
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

Calling Enq on a non-empty queue



Workspace

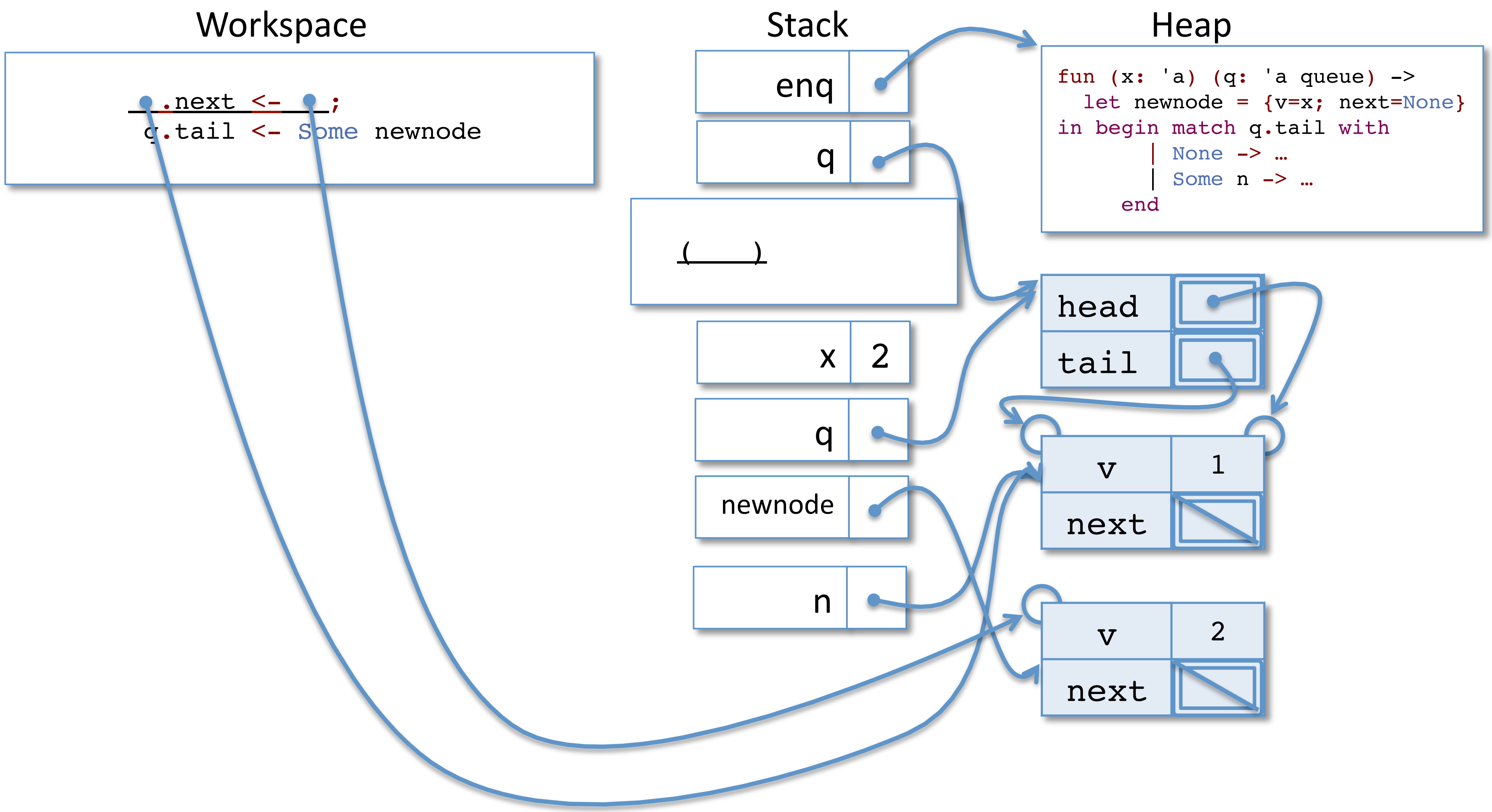
```
 .next <-   ;
q.tail <- Some newnode
```

Stack

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```

Calling Enq on a non-empty queue



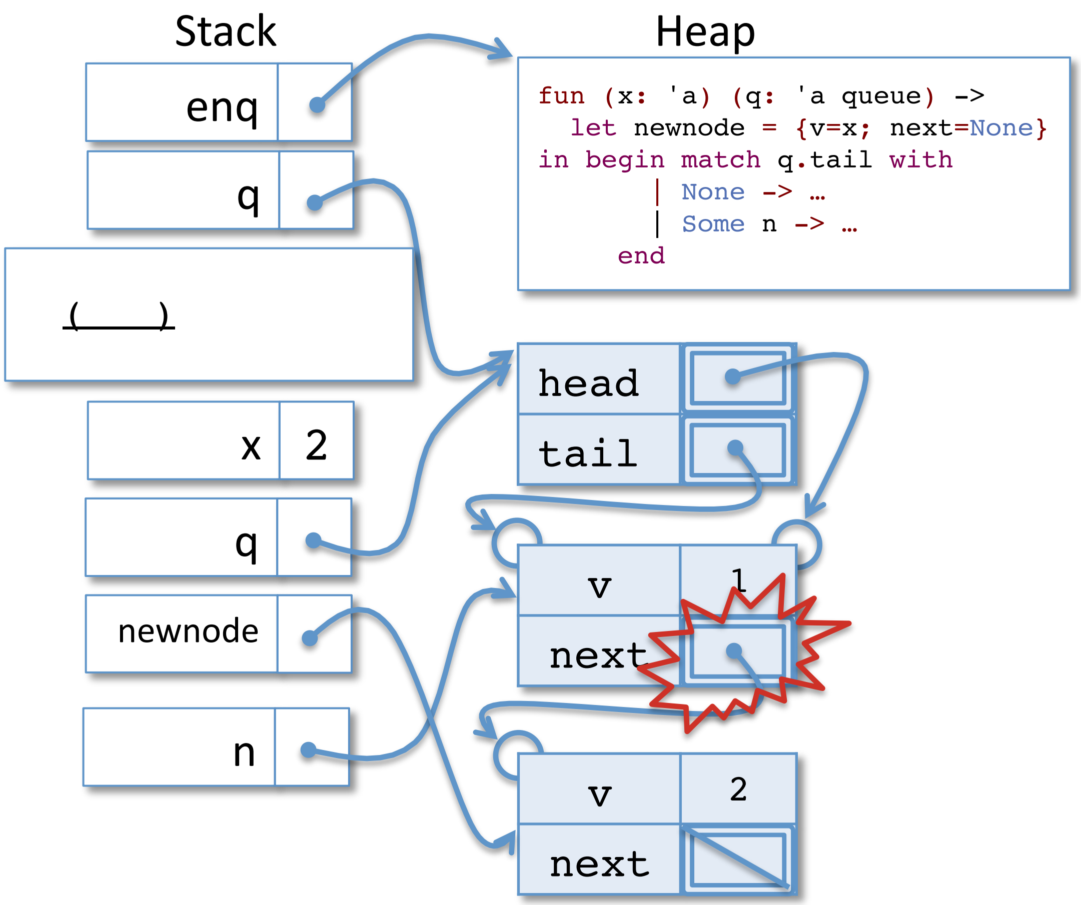
```
 .next <-  ;
q.tail <- Some newnode
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```

Calling Enq on a non-empty queue

Workspace

```
();
 q.tail <- Some newnode
```

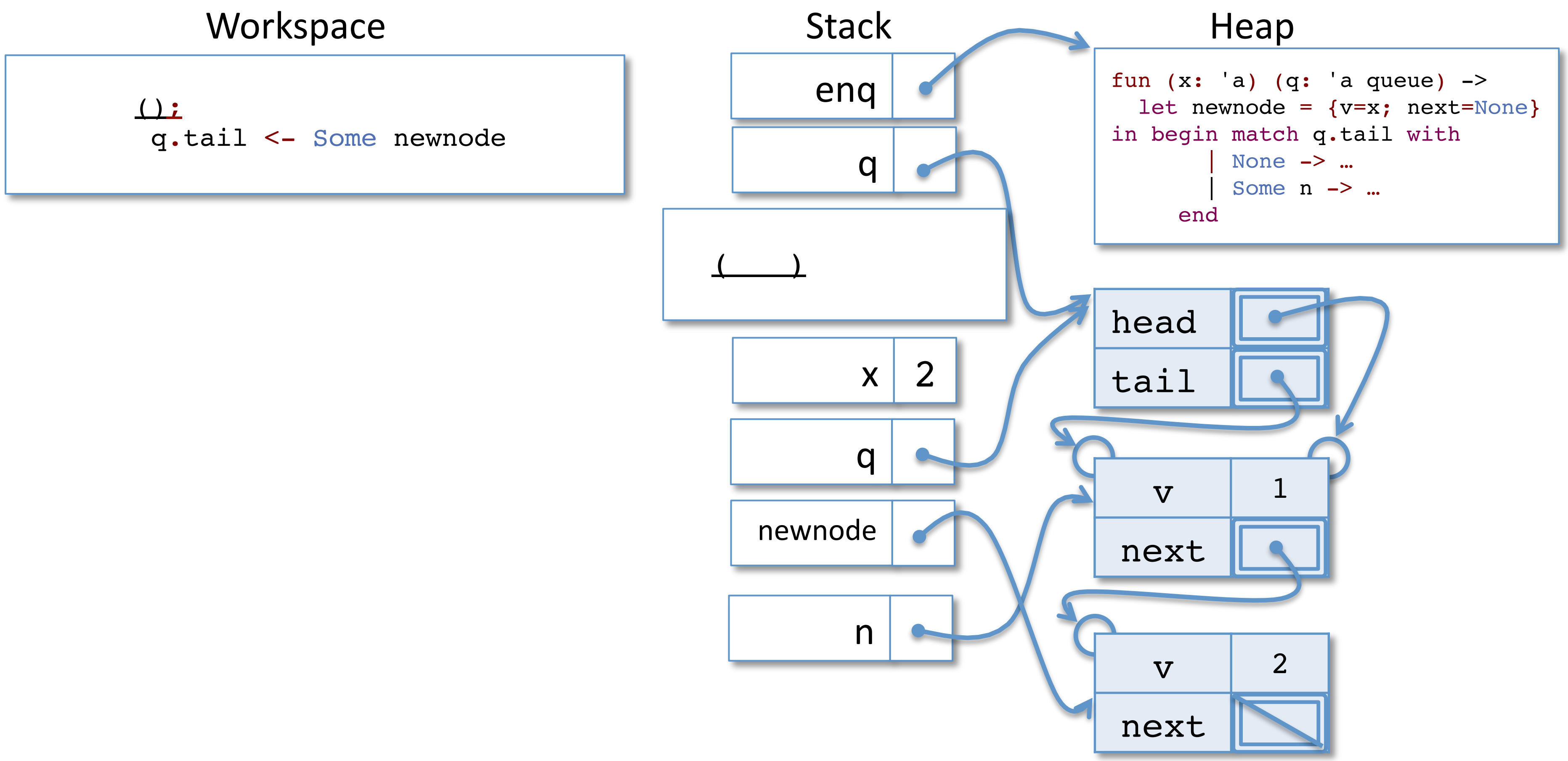


Stack

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

Calling Enq on a non-empty queue



Workspace

```
();
 q.tail <- Some newnode
```

Stack

enq

q

(___)

x 2

q

newnode

n

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

head

tail

v 1

next

v 2

next

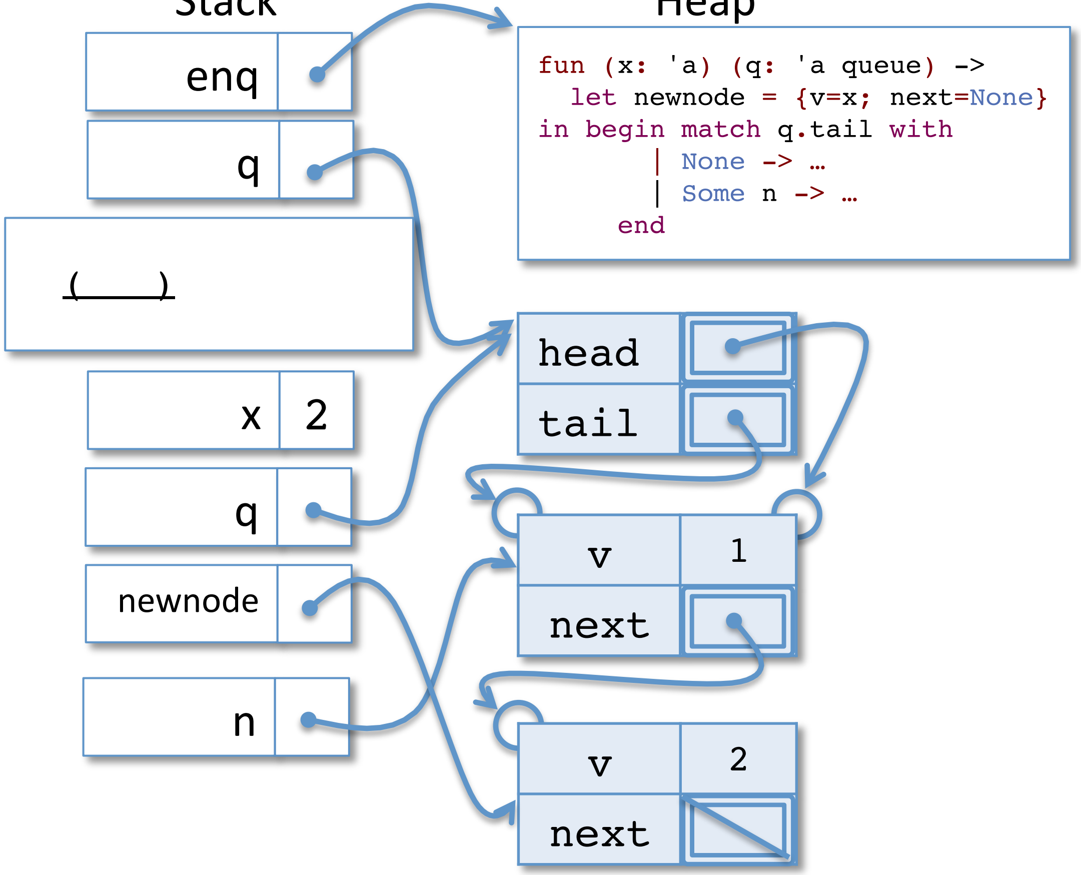
Calling Enq on a non-empty queue

Workspace

```
q.tail <- Some newnode
```

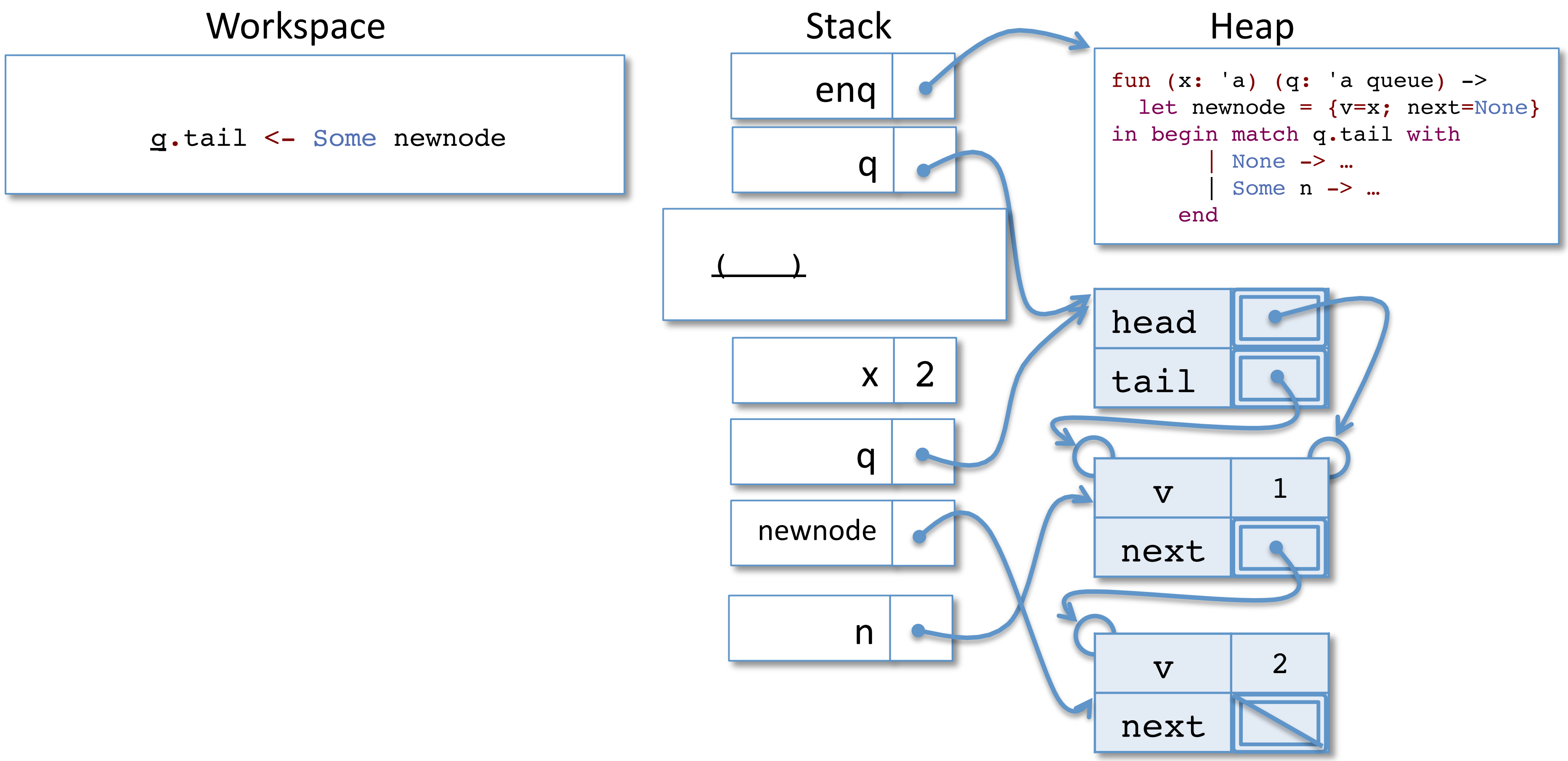
Stack

Heap



```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

Calling Enq on a non-empty queue



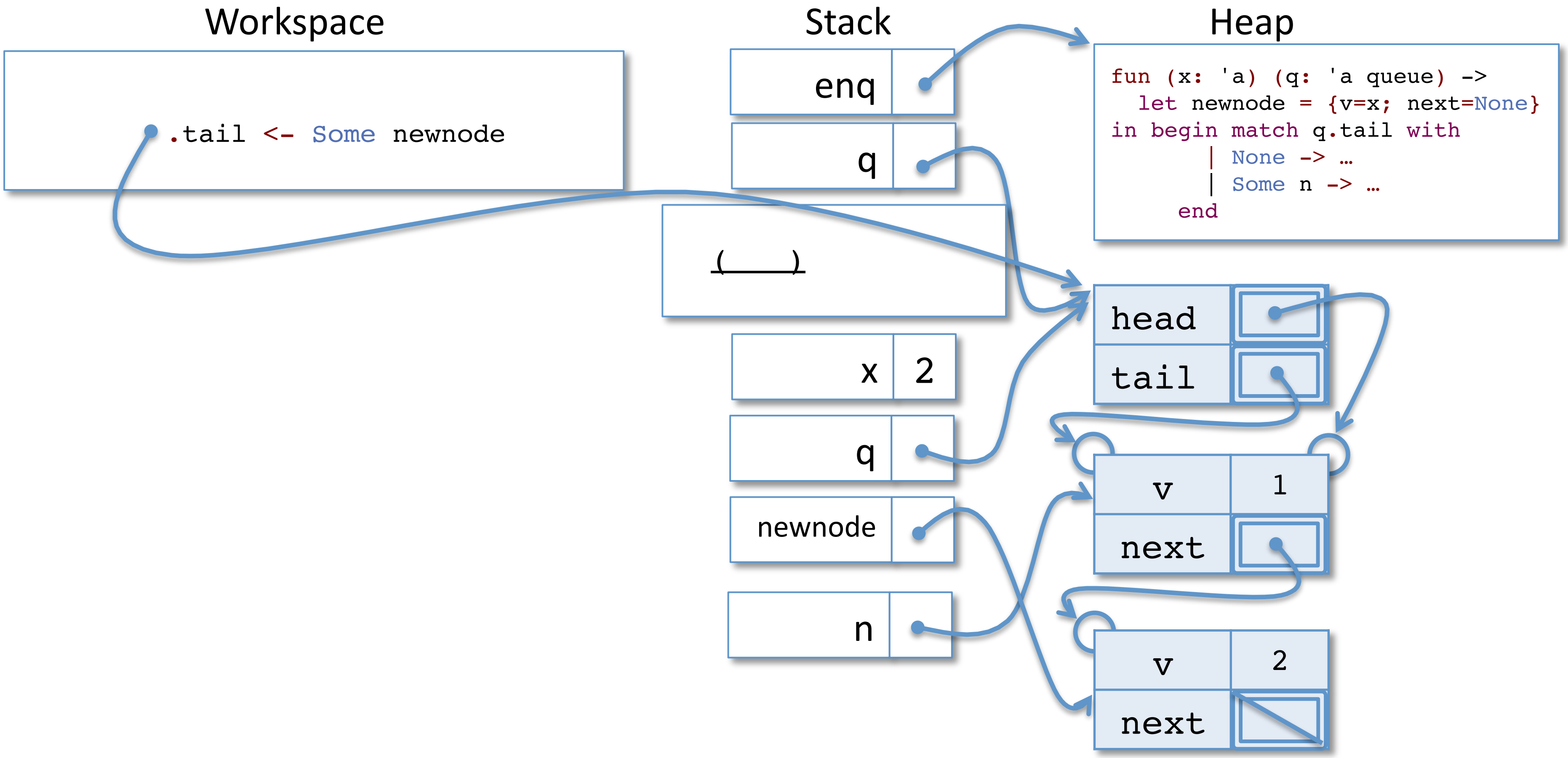
Workspace

```
q.tail <- Some newnode
```

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

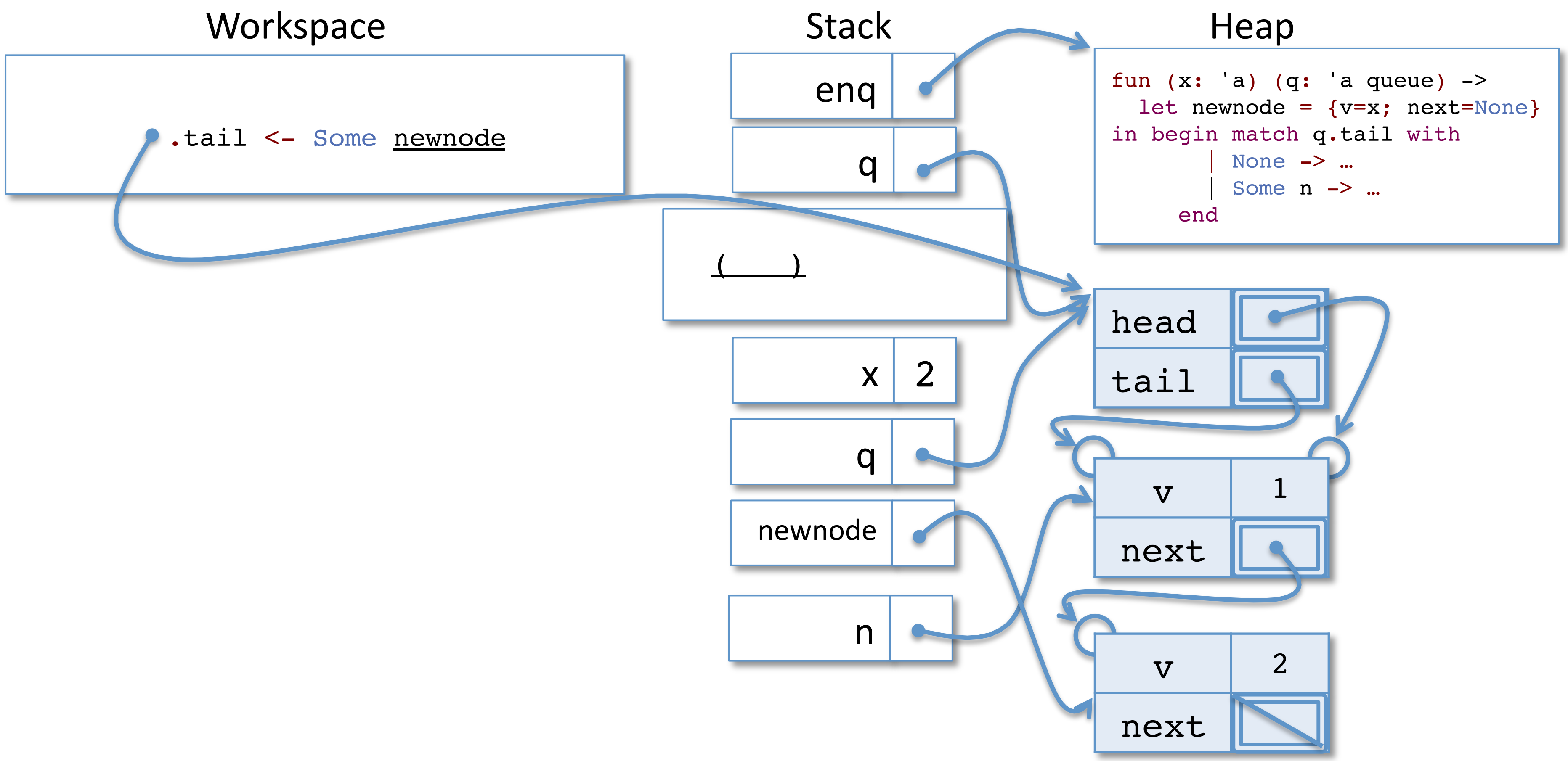
Calling Enq on a non-empty queue



```
.tail <- Some newnode
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
      | None -> …
      | Some n -> …
    end
```

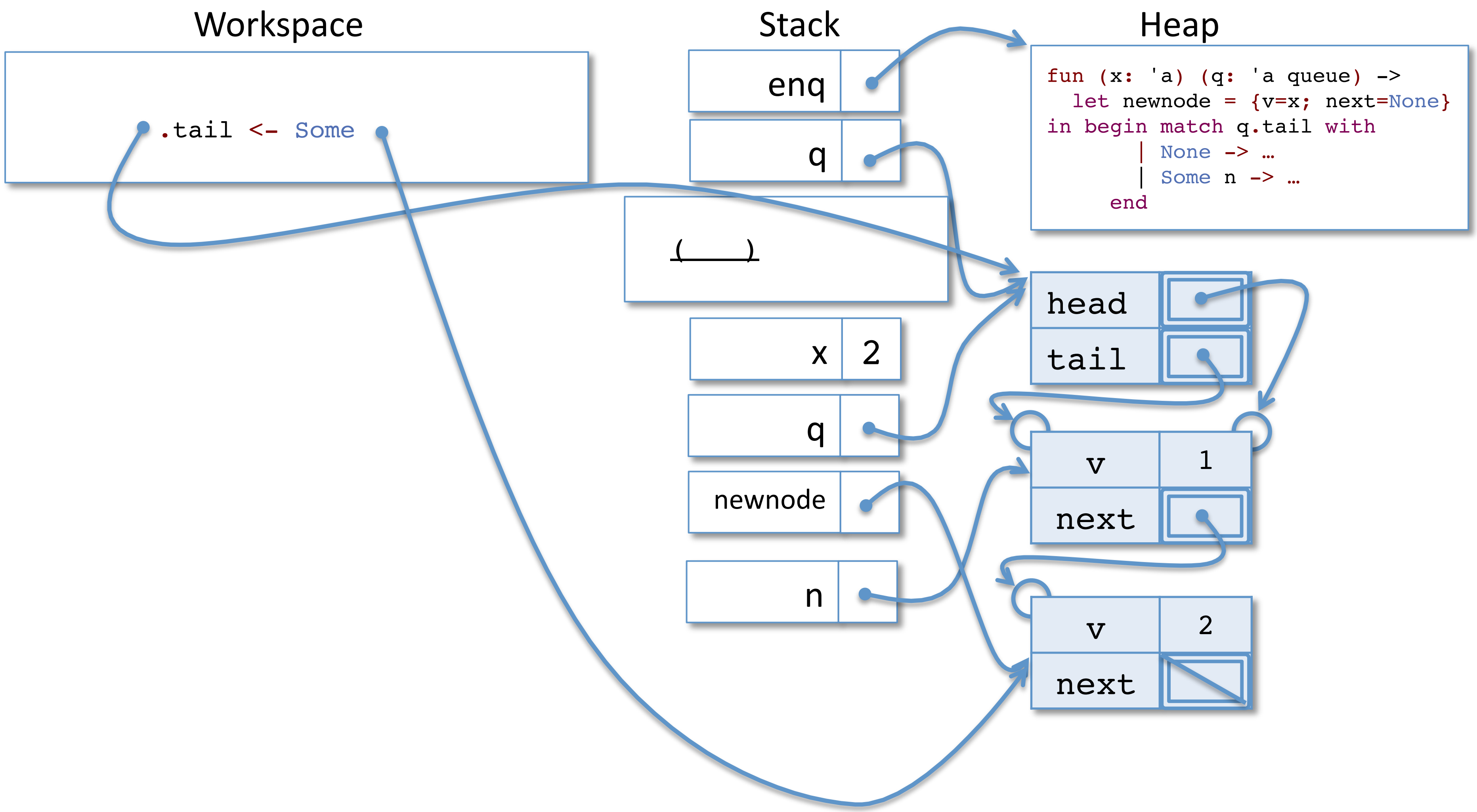
Calling Enq on a non-empty queue



```
.tail <- Some newnode
```

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```

Calling Enq on a non-empty queue



Workspace

```
.tail <- Some
```

Stack

enq	
q	
(___)	
x	2
q	
newnode	
n	

Heap

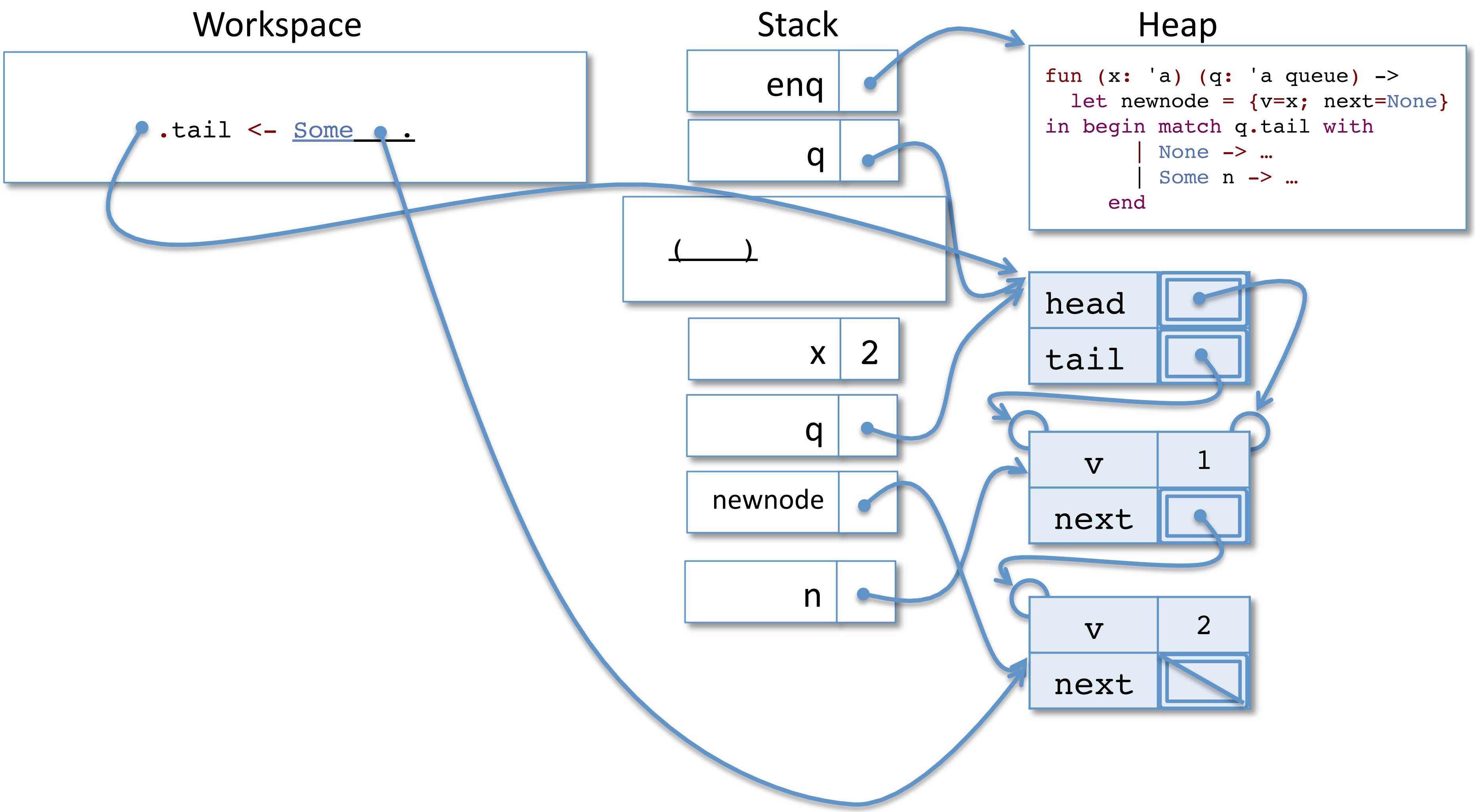
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

head	
tail	

v	1
next	

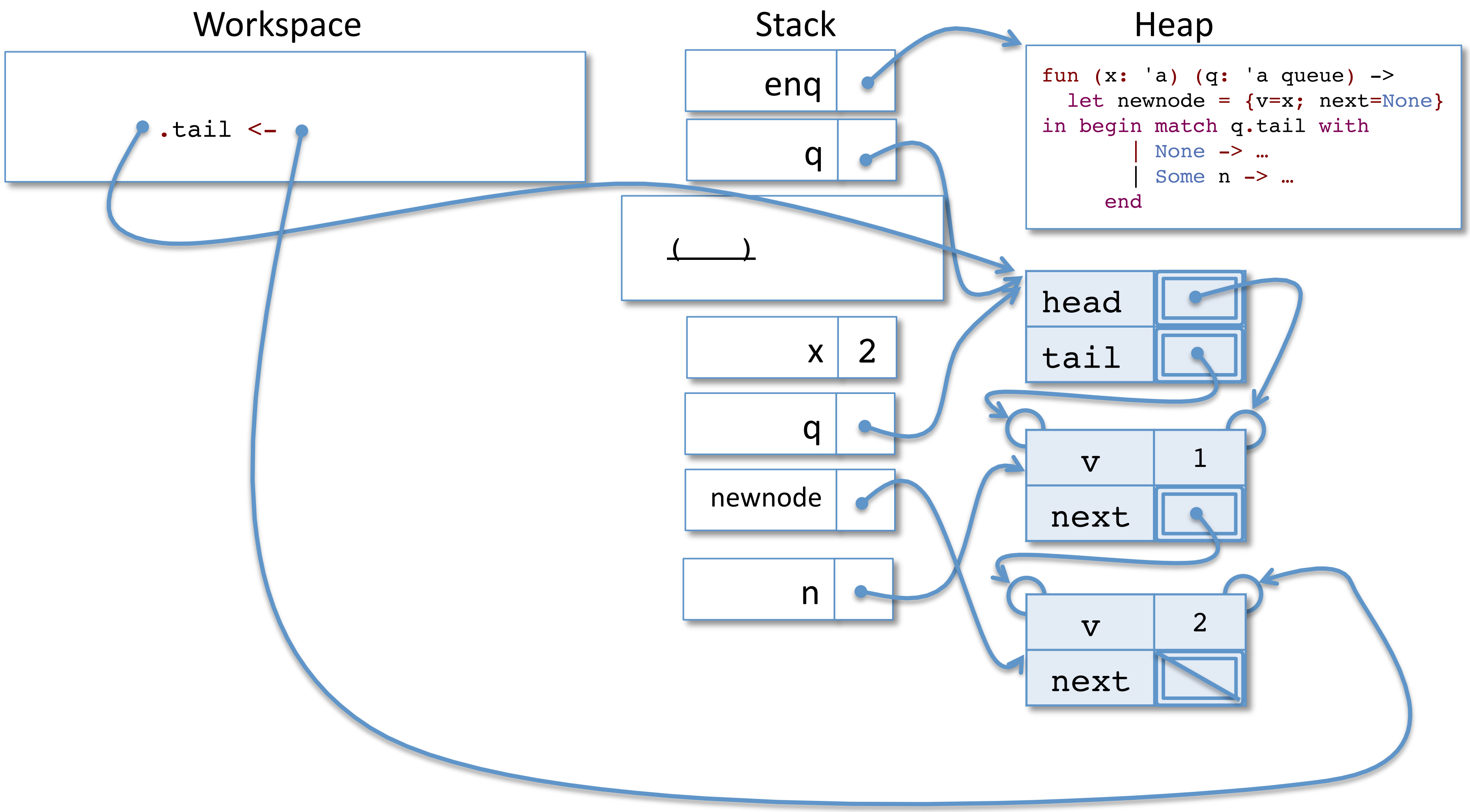
v	2
next	

Calling Enq on a non-empty queue



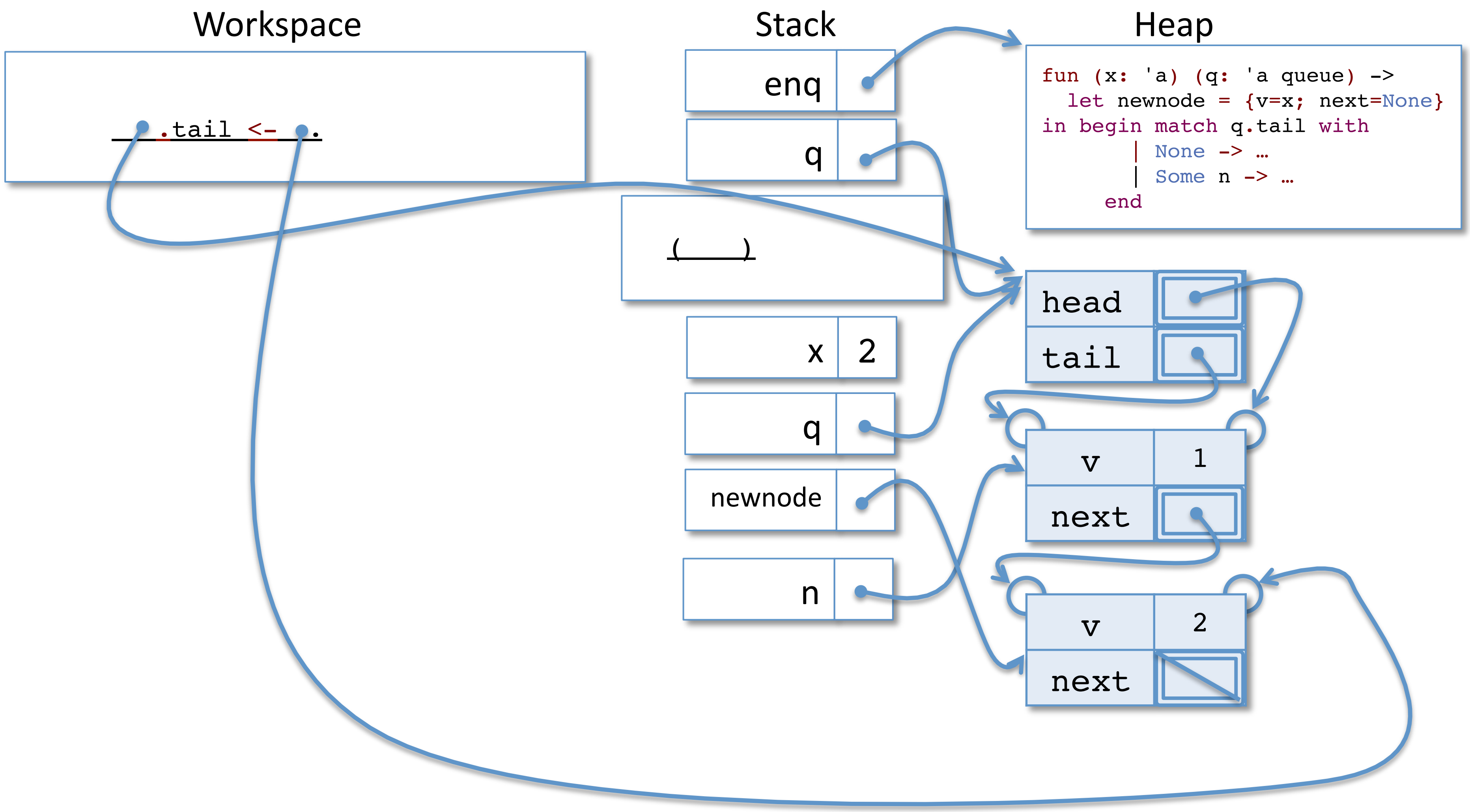
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
     | None -> …
     | Some n -> …
   end
```

Calling Enq on a non-empty queue



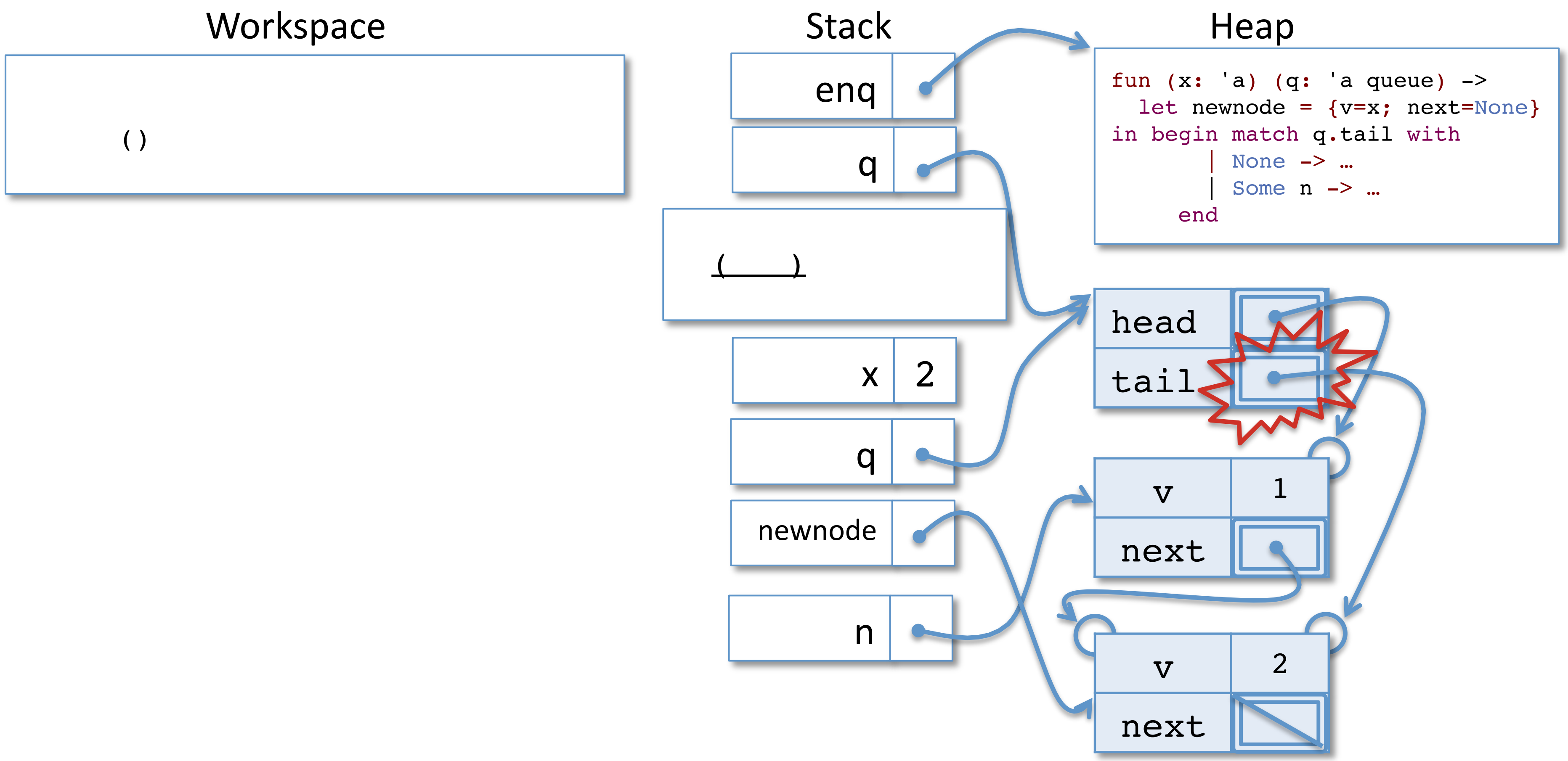
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

Calling Enq on a non-empty queue



```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```

Calling Enq on a non-empty queue



Workspace

()

Stack

enq

q

(___)

x 2

q

newnode

n

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

head

tail

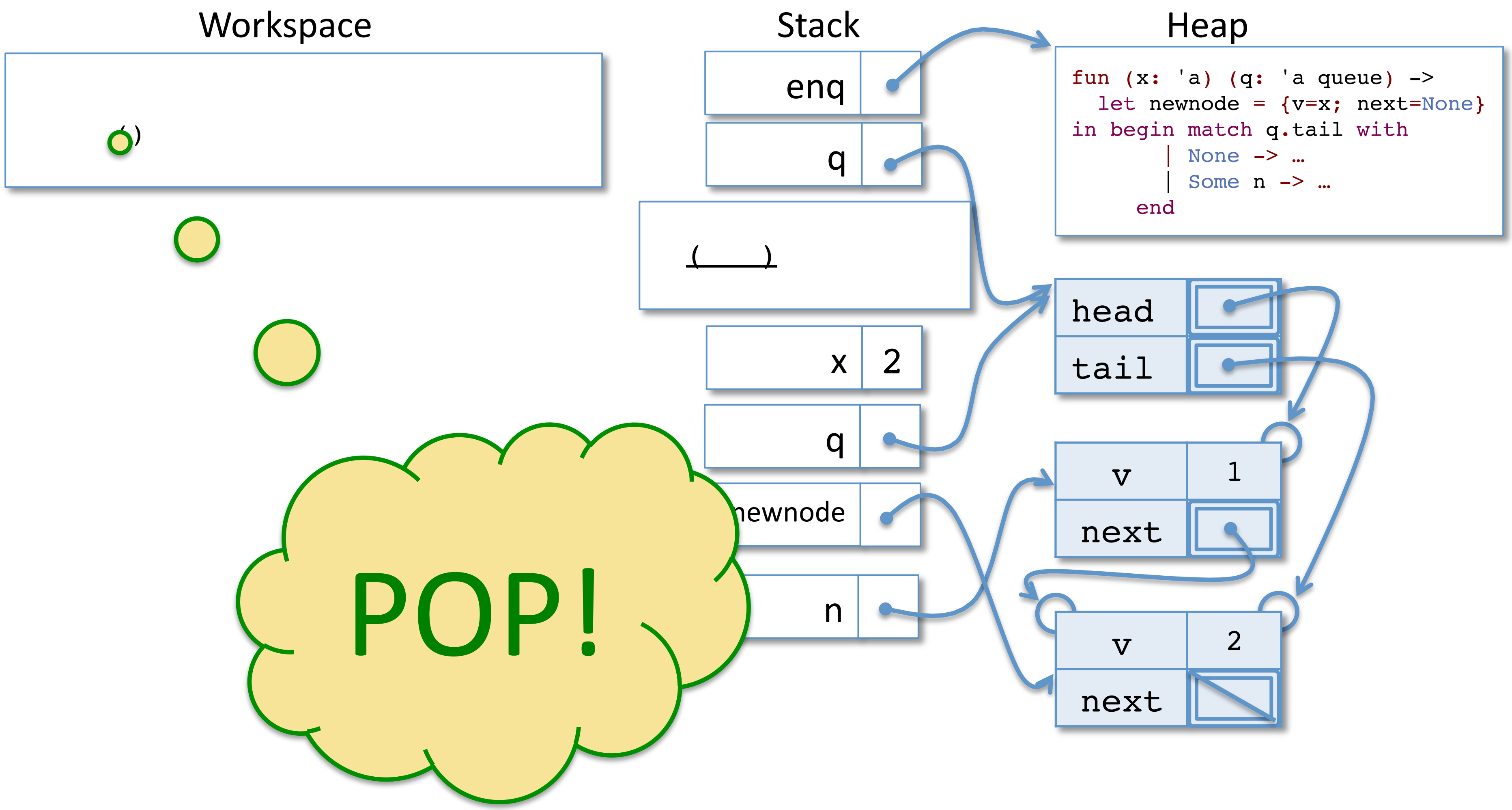
v 1

next

v 2

next

Calling Enq on a non-empty queue



Workspace

()

POP!

Stack

enq

q

(___)

x 2

q

newnode

n

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```

head

tail

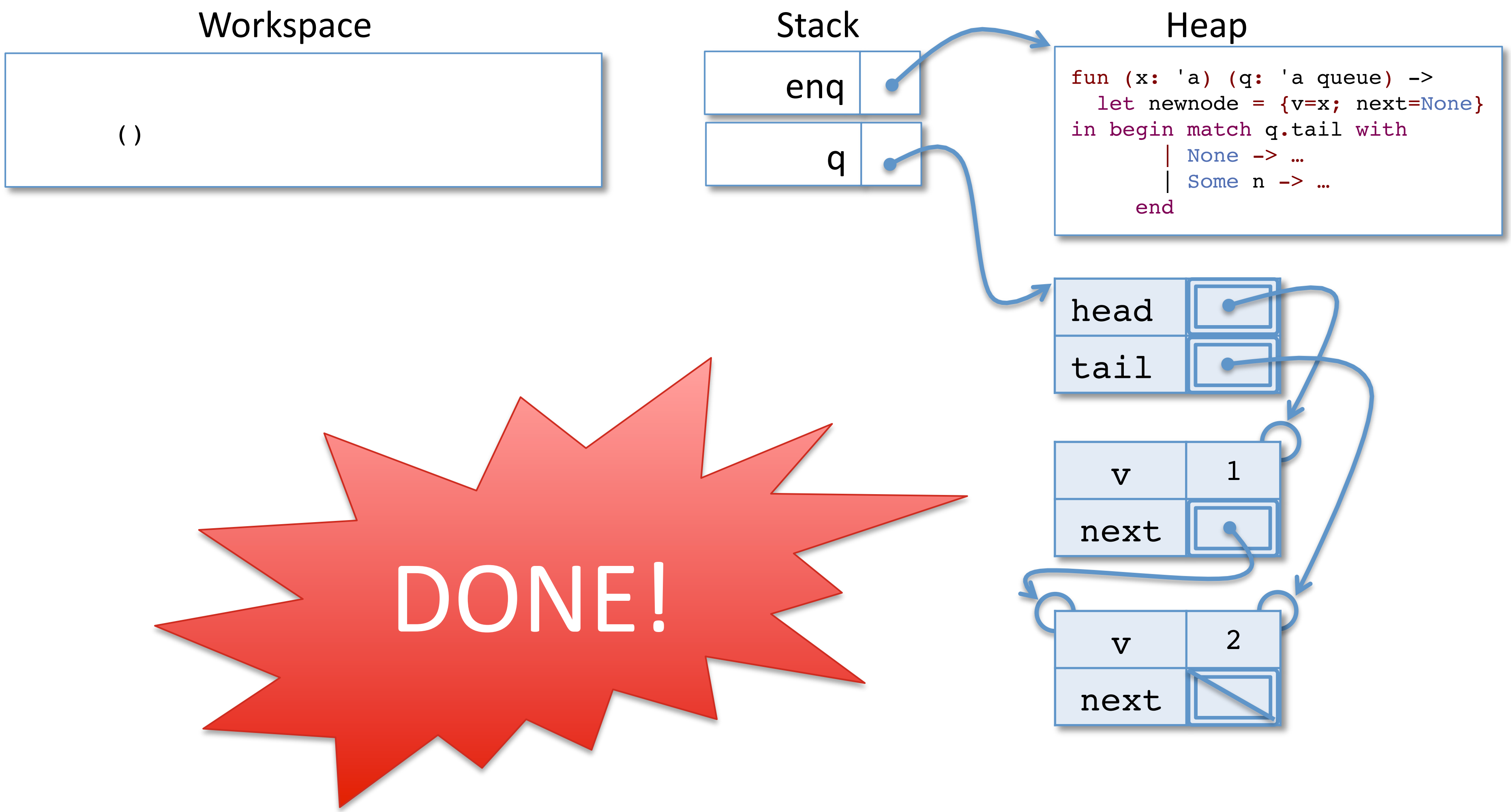
v 1

next

v 2

next

Calling Enq on a non-empty queue



Workspace

()

Stack

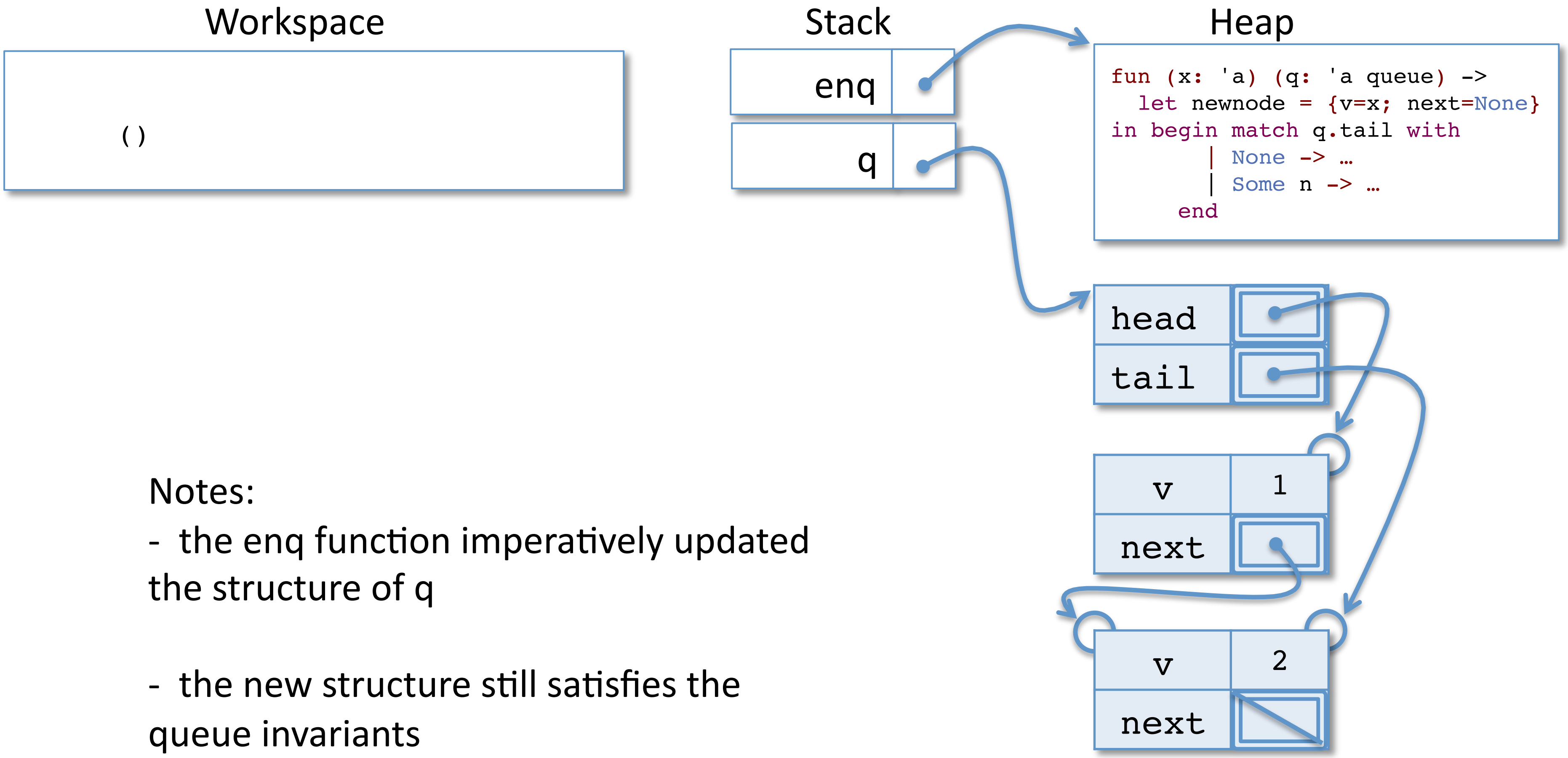
enq

q

Heap

```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```

Calling Enq on a non-empty queue



```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
           | None -> …
           | Some n -> …
     end
```

Notes:

- the enq function imperatively updated the structure of q

- the new structure still satisfies the queue invariants

deq

```
(* remove an element from the head of the queue *)
 let deq (q: 'a queue) : 'a =
   begin match q.head with
     | None ->
         failwith "deq called on empty queue"
     | Some n ->
         q.head <- n.next;
         if n.next = None then q.tail <- None;
         n.v
   end
```

- The code for deq must also “patch pointers” to maintain the queue invariant:
  - The head pointer is always updated to the next element in the queue.
  - If the removed node was the last one in the queue, the tail pointer must be updated to None