# Programming Languages and Techniques (CIS120)

Lecture 15

Feb 24, 2014

Linked Queues

## Announcements

- Today's clicker responses combined with 2/19 lecture
- Homework 5 (queues) available
  - due FRIDAY at 11:59:59pm
- Read Ch. 16 of lecture notes

- Homework 6 will be due FRIDAY, March 7th
  - will be available Friday
  - day before Spring Break
- After Spring Break we switch to Java

## Midterm 1 results

- Stats:
  - Median: 83
  - Mean: 81.99
  - Std dev: 9.89
  - Max: 100 (1)

Great Job!

- Grade ranges:
  - A 86–100 (= mastery of all course material)
  - B 76–85 (= good command of most course material)
  - C 66–75 (= substantial understanding of core material)

- Exams/Solutions available Wednesday after the make up.

Did you find the exam…

1. Much easier than you expected?
2. Easier than expected?
3. About what you expected?
4. Harder than expected?
5. Much harder?

Do you feel your performance on the exam…

1. was better than you expected?
2. was a pretty accurate reflection of your understanding of the material?
3. was disappointing?

## Mutable Queues

singly linked data structures

# (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
```

## 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).

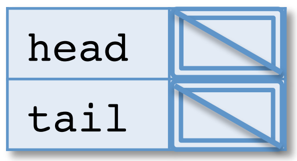
## Linked Queue Invariants

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

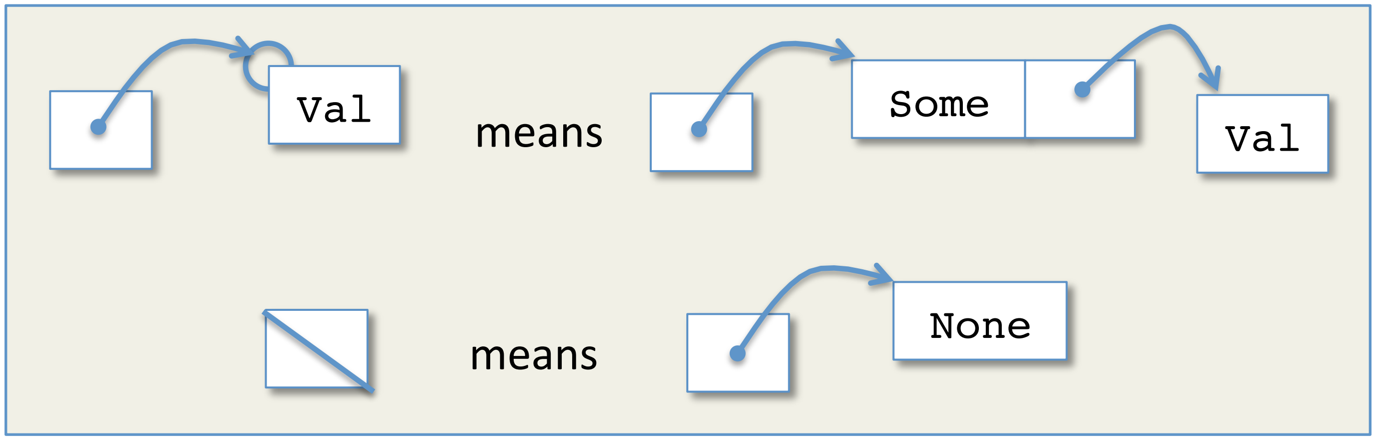
Either:
 (1) `head` and `tail` are both `None`    (i.e. the queue is empty)
or
 (2) `head` is `Some n1`, `tail` is `Some n2` and
   - `n2` is reachable from `n1` by following 'next' pointers
   - `n2.next` is `None`

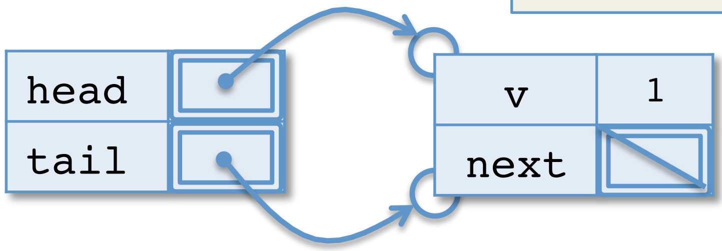
- We can check that these properties rule out all of the "bogus" examples.
- A queue operation may assume that these invariants hold of its inputs, and must ensure that the invariants hold when it's done.

# Visual Shorthand: Abbreviating Options

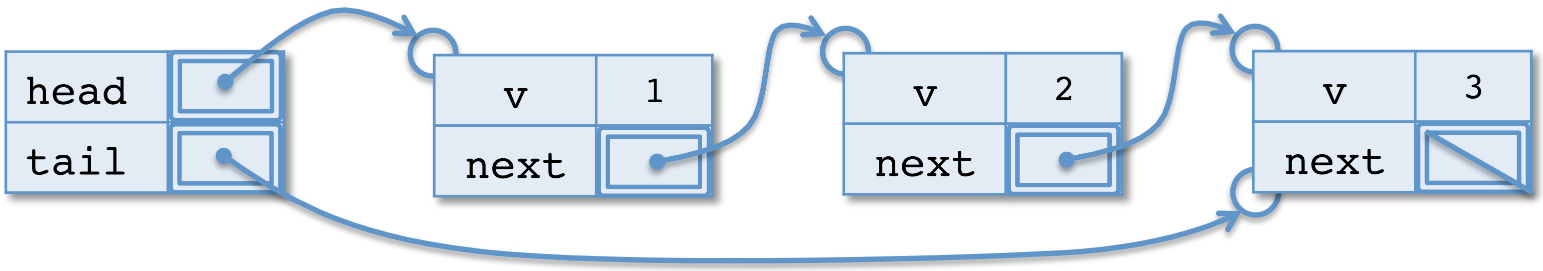


An empty queue





A queue with one element



A queue with three elements

Either:

(1) `head` and `tail` are both `None` (i.e. the queue is empty)

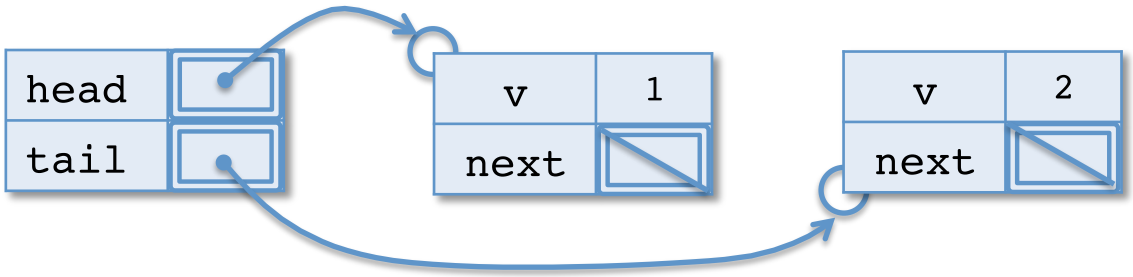
or

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

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

Is this a valid queue?

1. Yes
2. No



Either:

(1) `head` and `tail` are both `None` (i.e. the queue is empty)

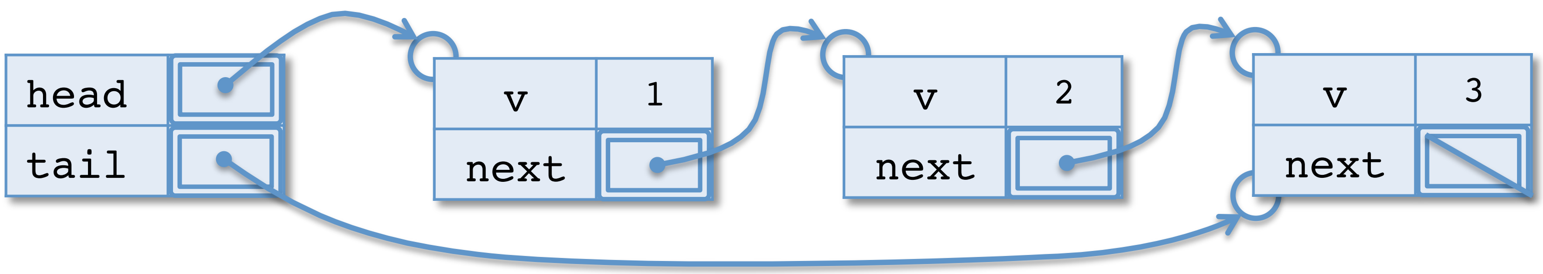
or

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

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

Is this a valid queue?

1. Yes
2. No



Either:

(1) `head` and `tail` are both `None` (i.e. the queue is empty)

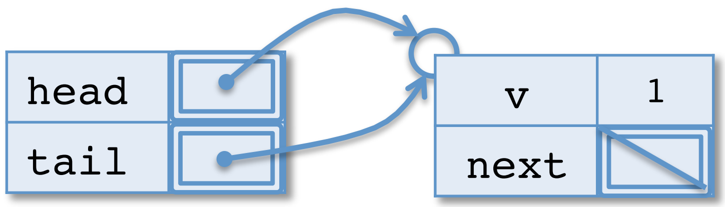
or

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

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

Is this a valid queue?

1. Yes
2. No



Either:

(1) `head` and `tail` are both `None` (i.e. the queue is empty)

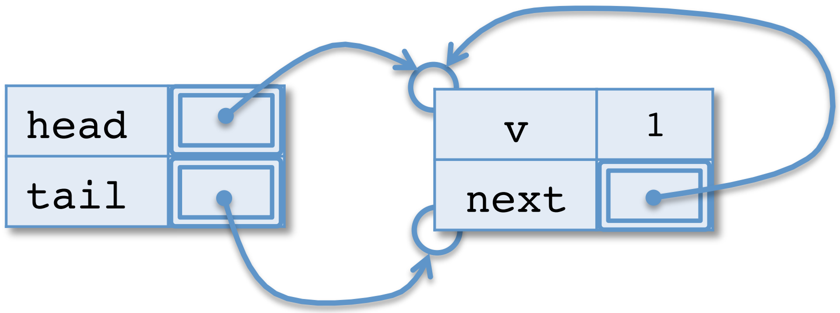
or

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

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

Is this a valid queue?

1. Yes
2. No



## 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
```

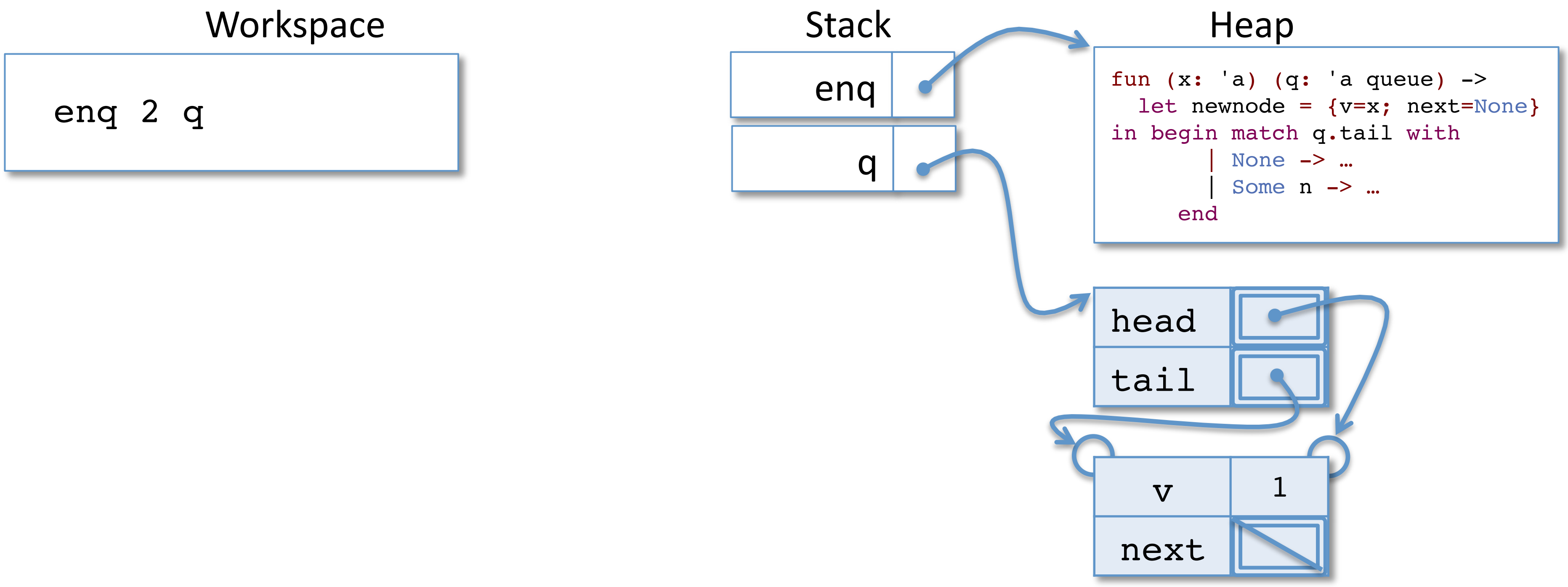
- `create` *establishes* the queue invariants
  - both head and tail are None
- `is_empty` *assumes* the queue invariants
  - it doesn’t have to check that q.tail is None

## 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 ->
        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



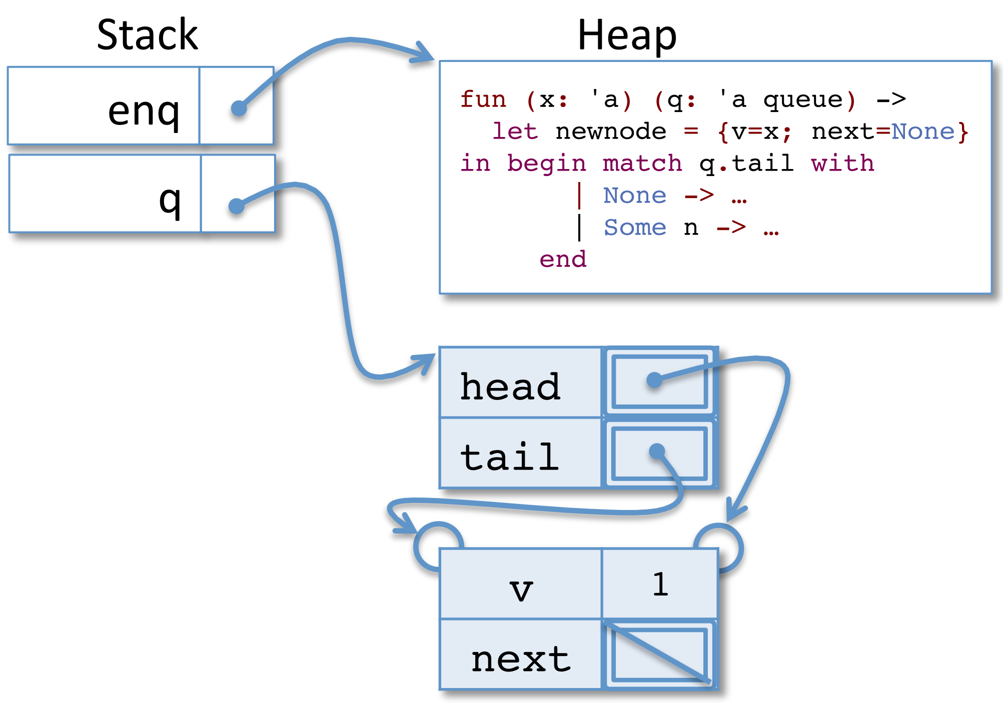
```
enq 2 q
```

```
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

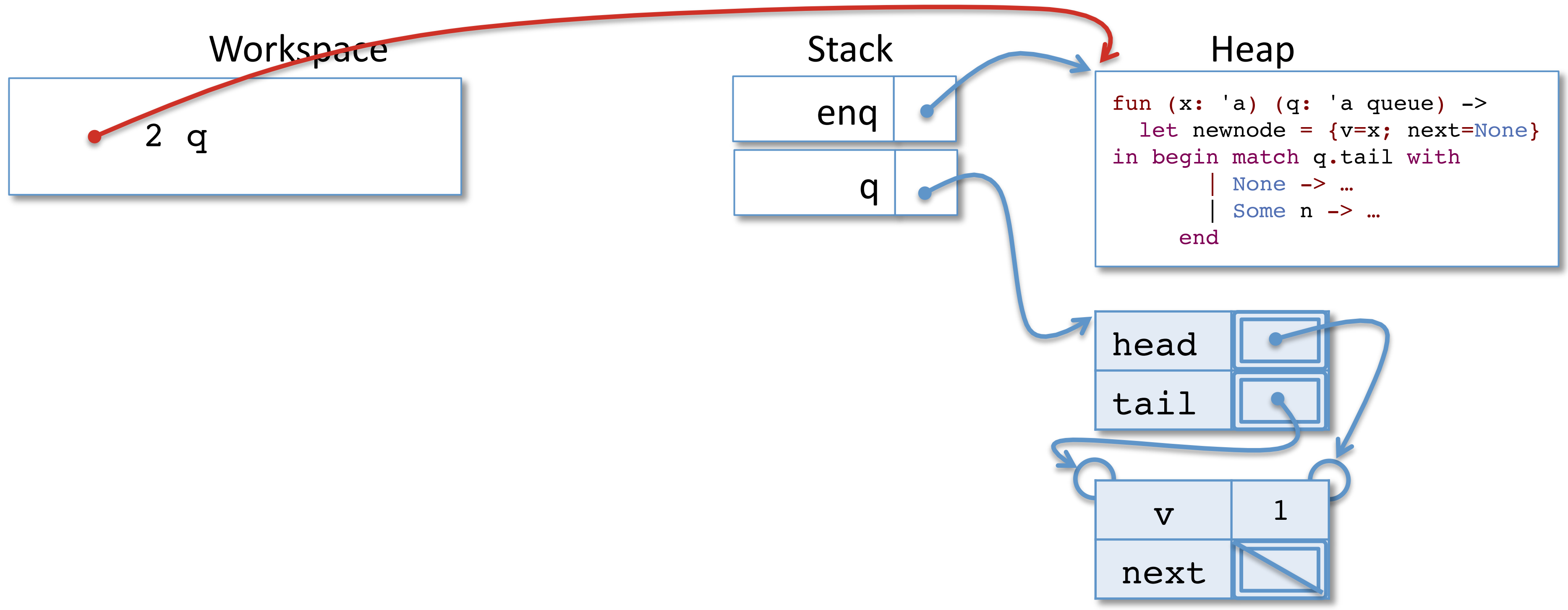
Workspace

```
enq 2 q
```



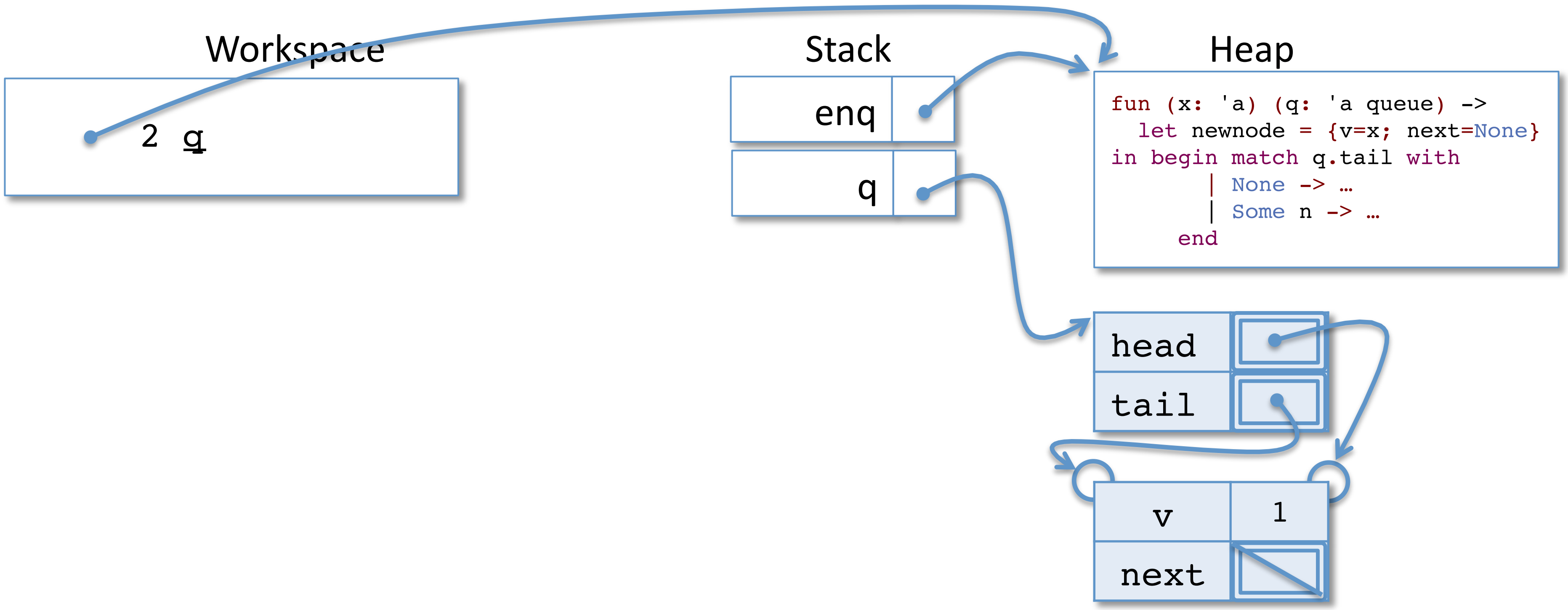
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

# Calling Enq on a non-empty queue



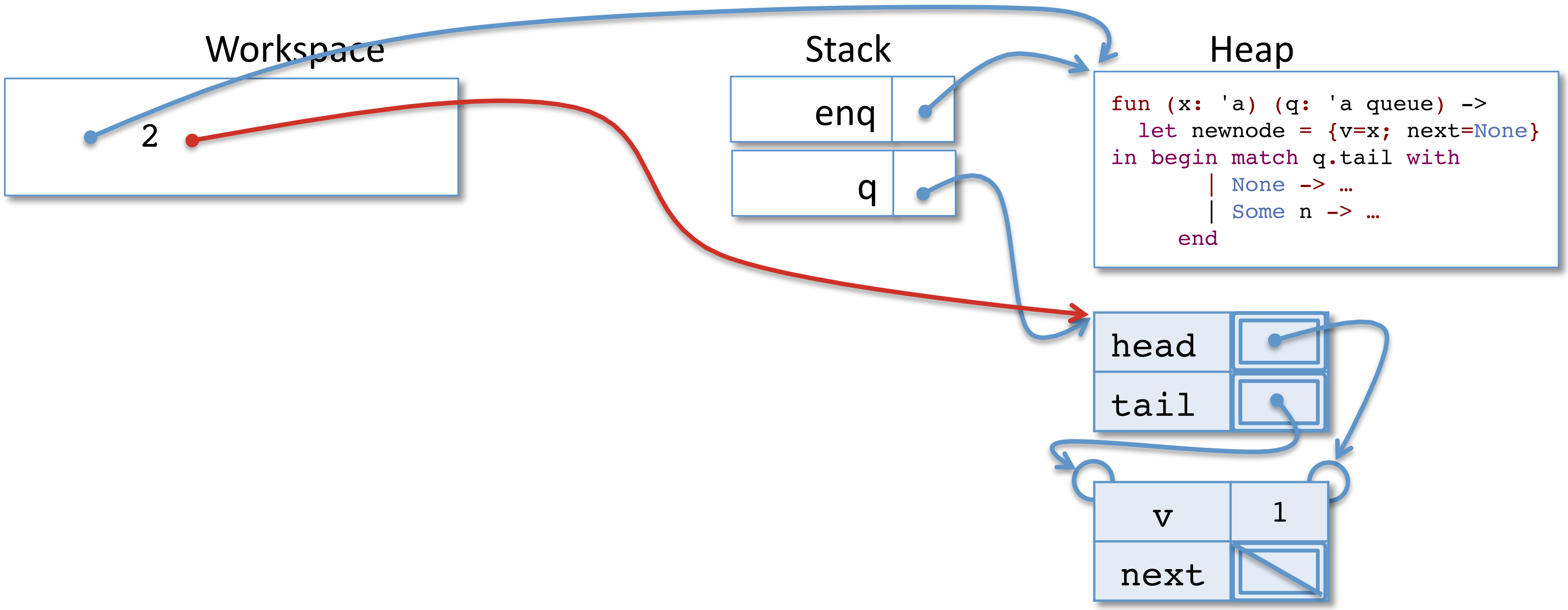
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

# 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

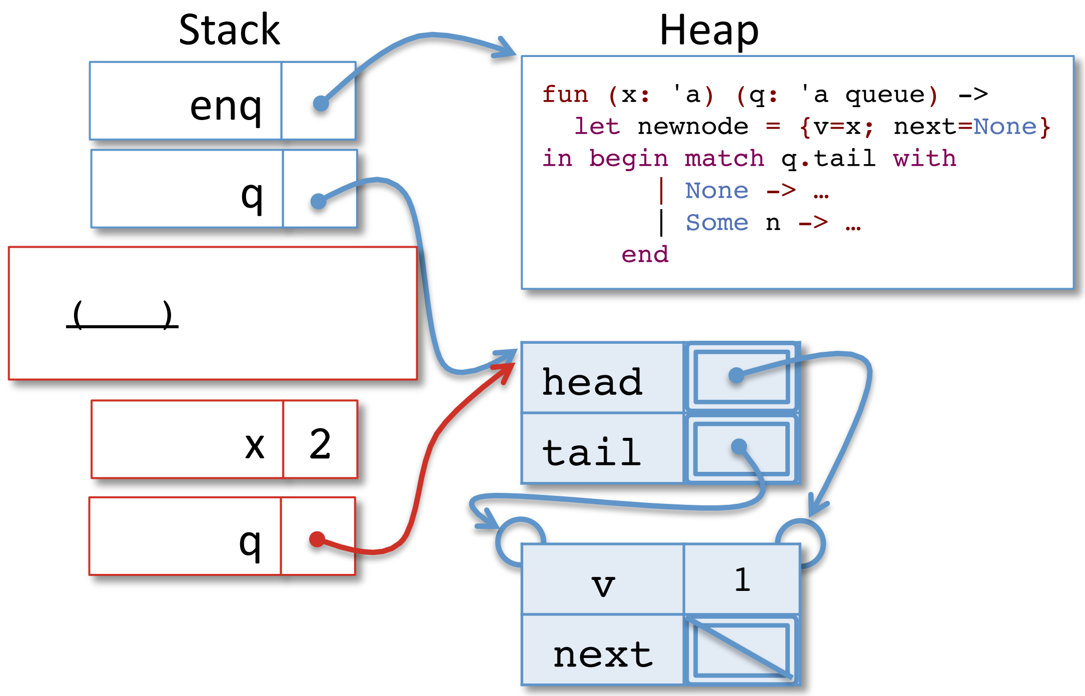


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

# 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
```

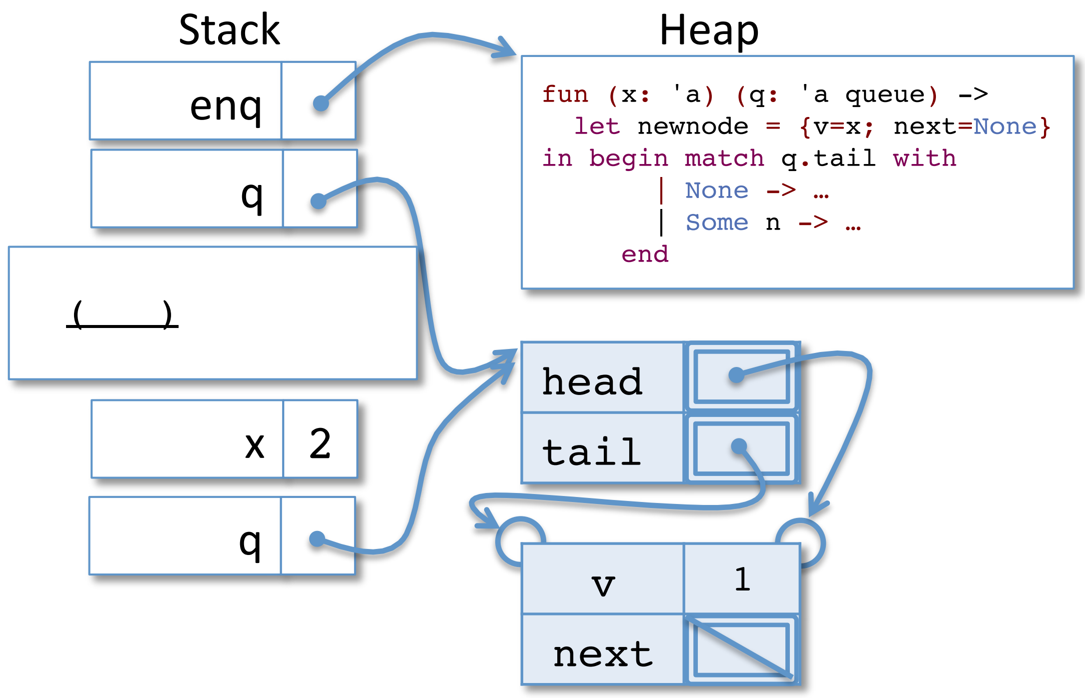


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```

# Calling Enq on a non-empty queue

Workspace

```
let newnode = {v=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
```

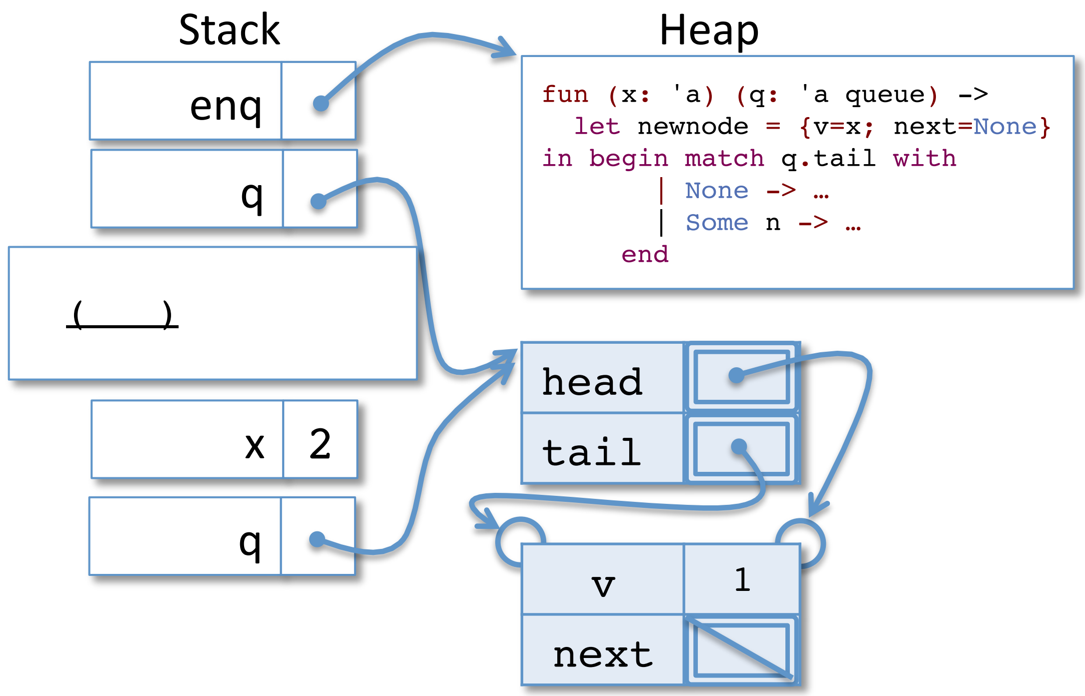


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```

# 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
```

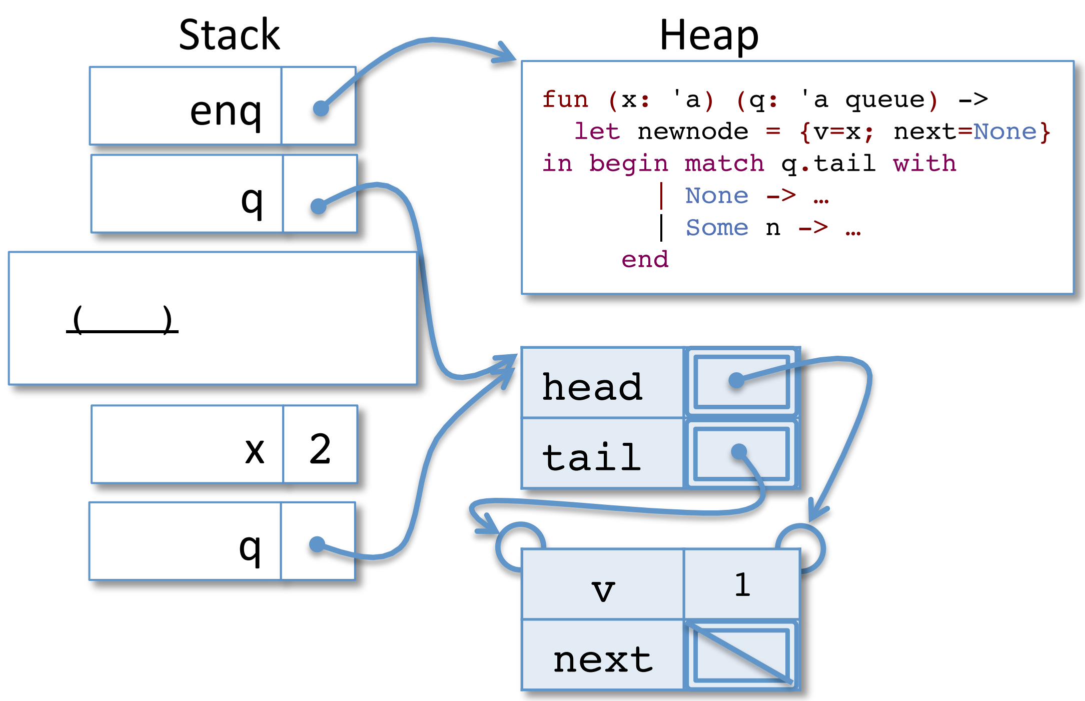


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```

# Calling Enq on a non-empty queue

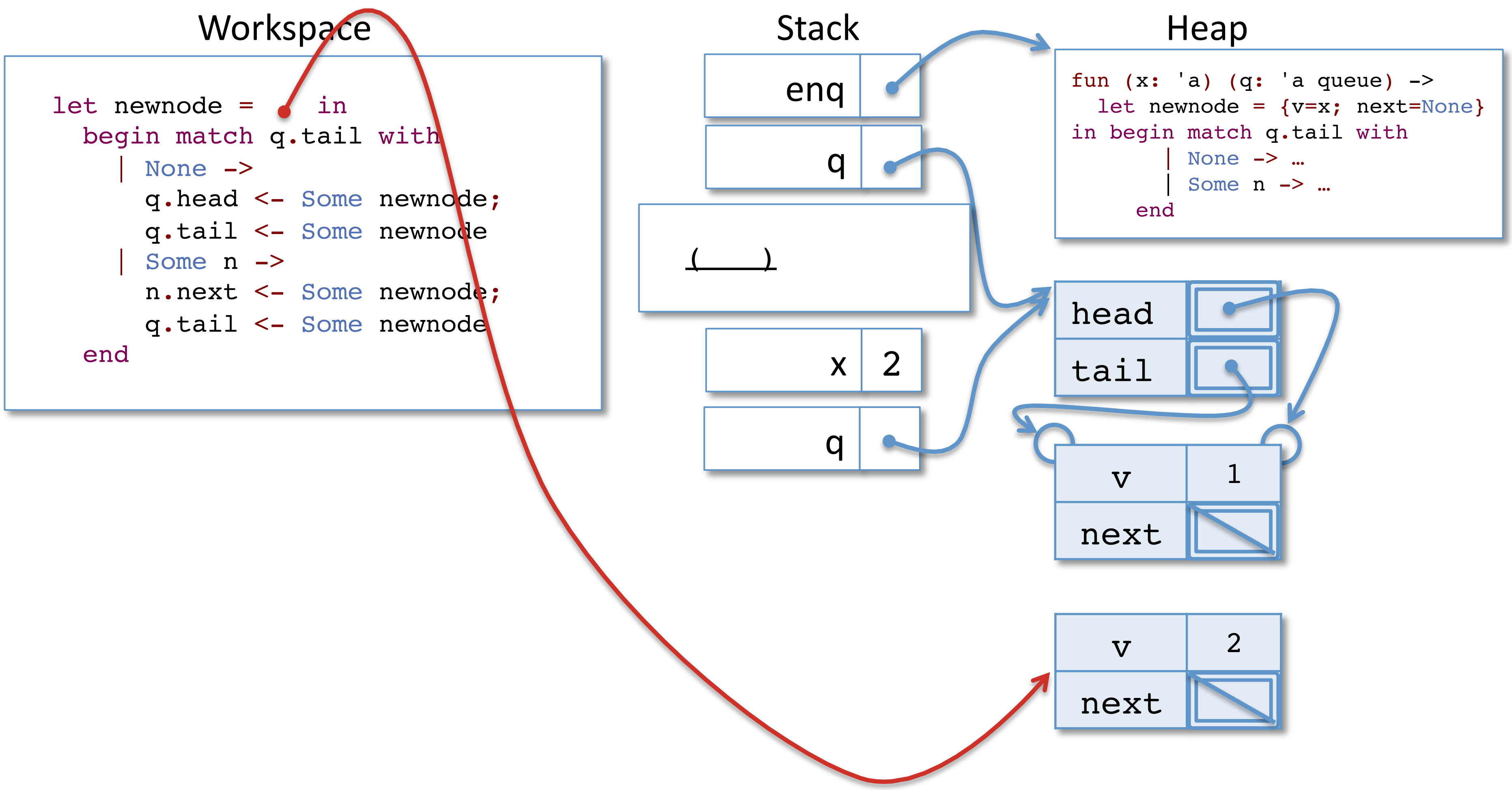
Workspace

```
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

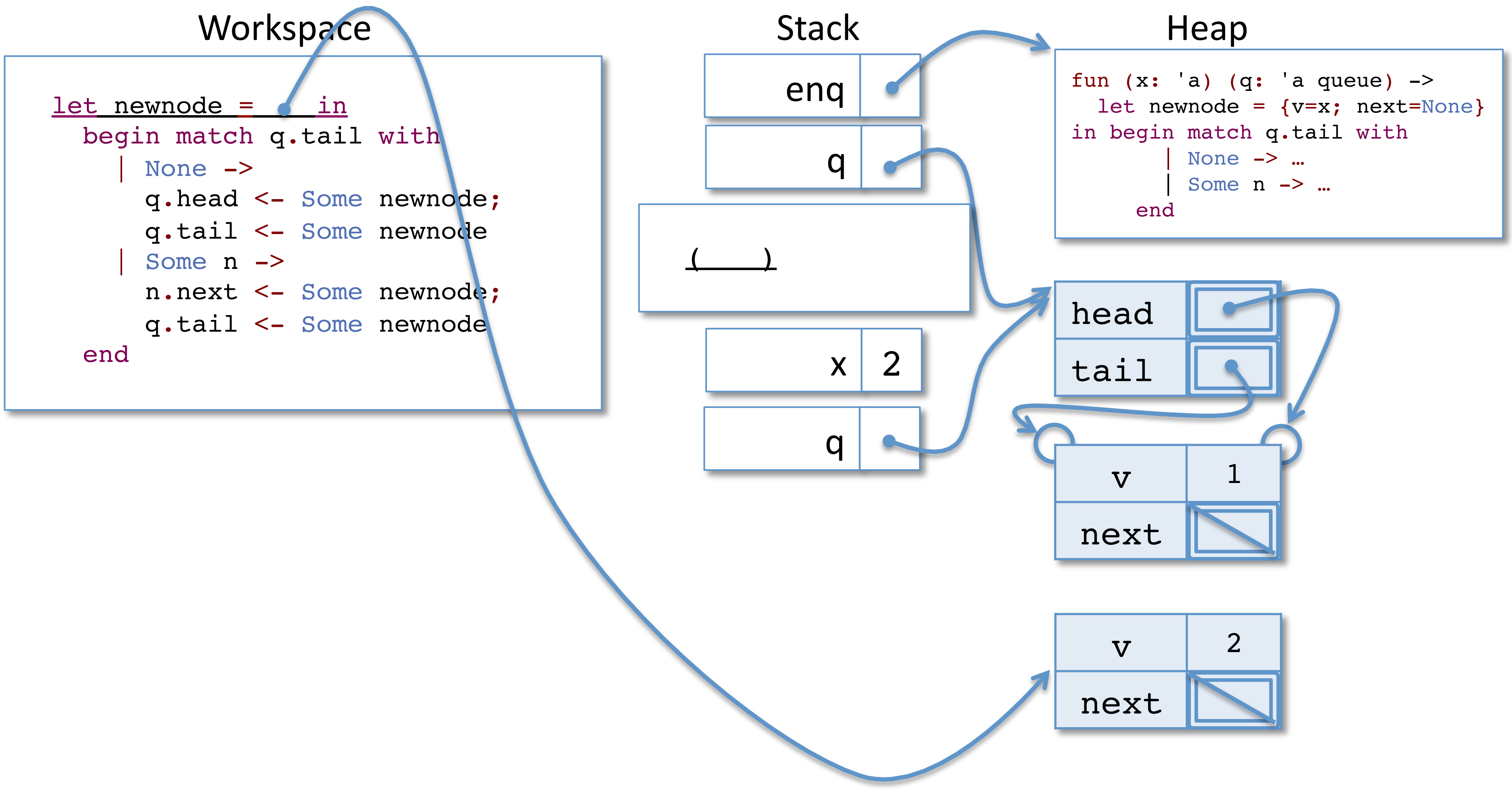


```
let newnode =     in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

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

Note: there is no “Some bubble”: this is a qnode, not a qnode option.

# Calling Enq on a non-empty queue



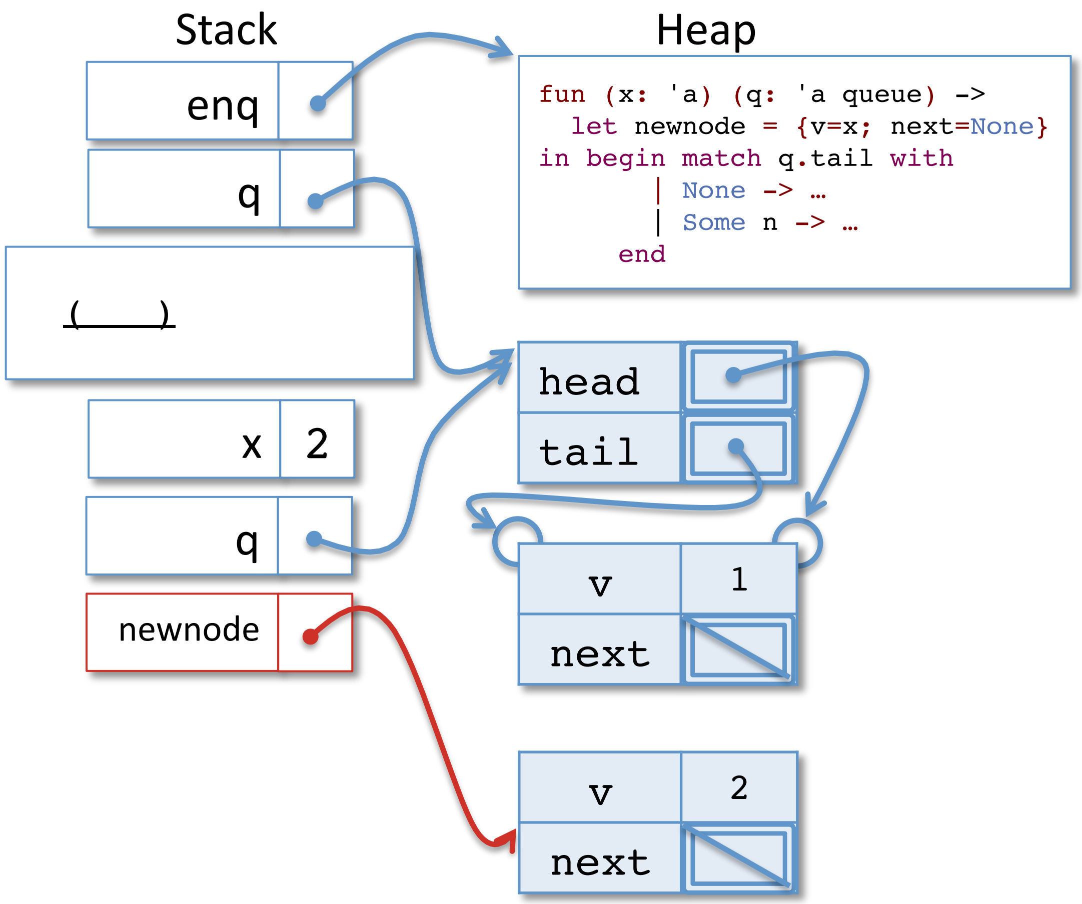
```
let newnode = ___ in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```

```
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
```

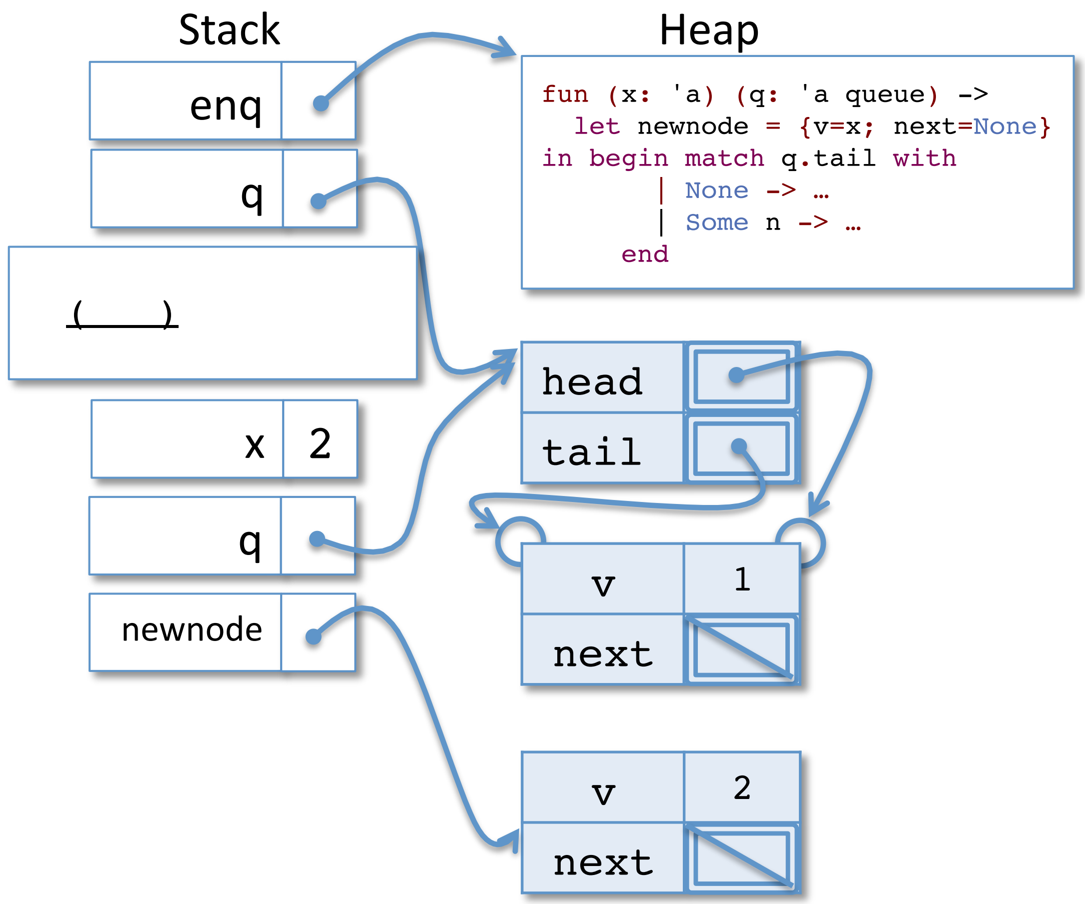


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```

# 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
```

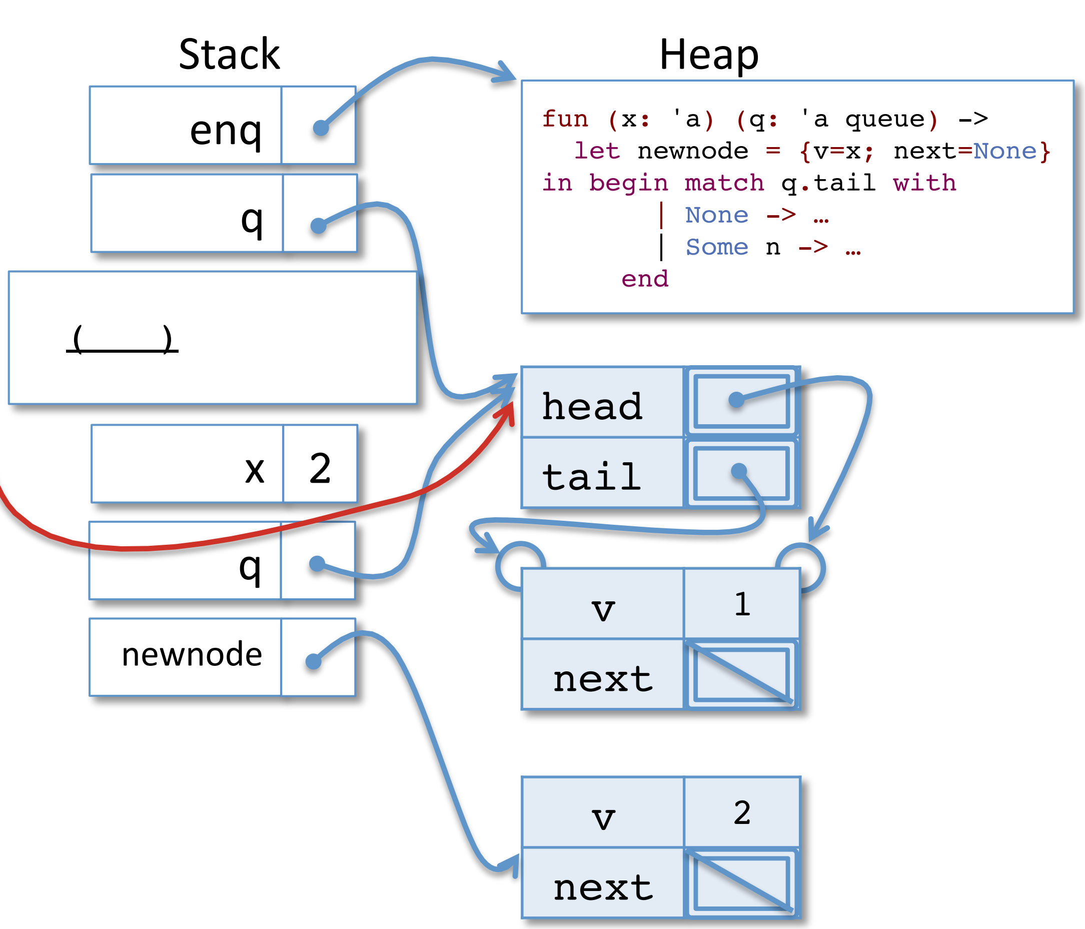


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```

# Calling Enq on a non-empty queue

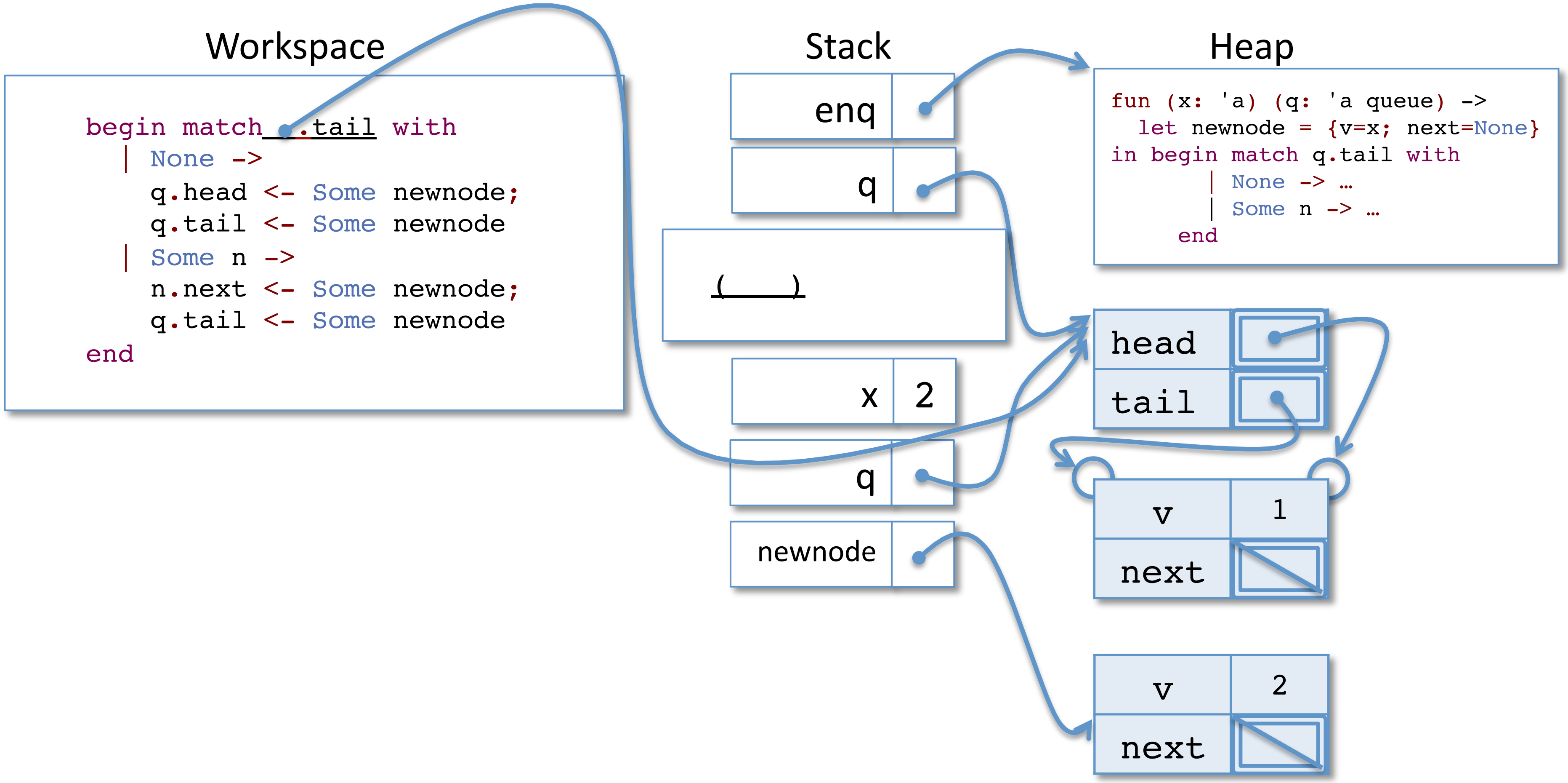
Workspace

```
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



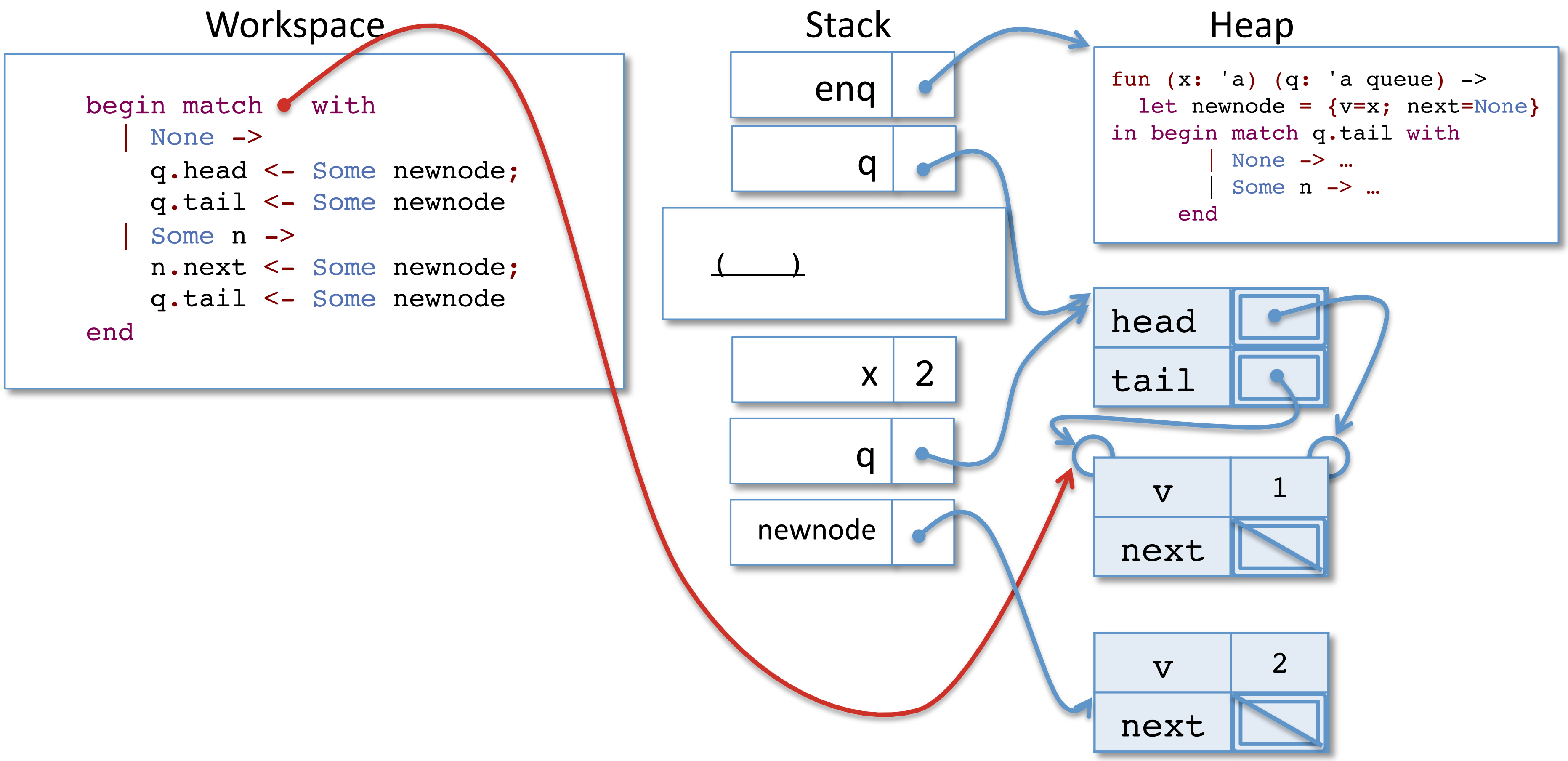
Workspace:

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

Heap:

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

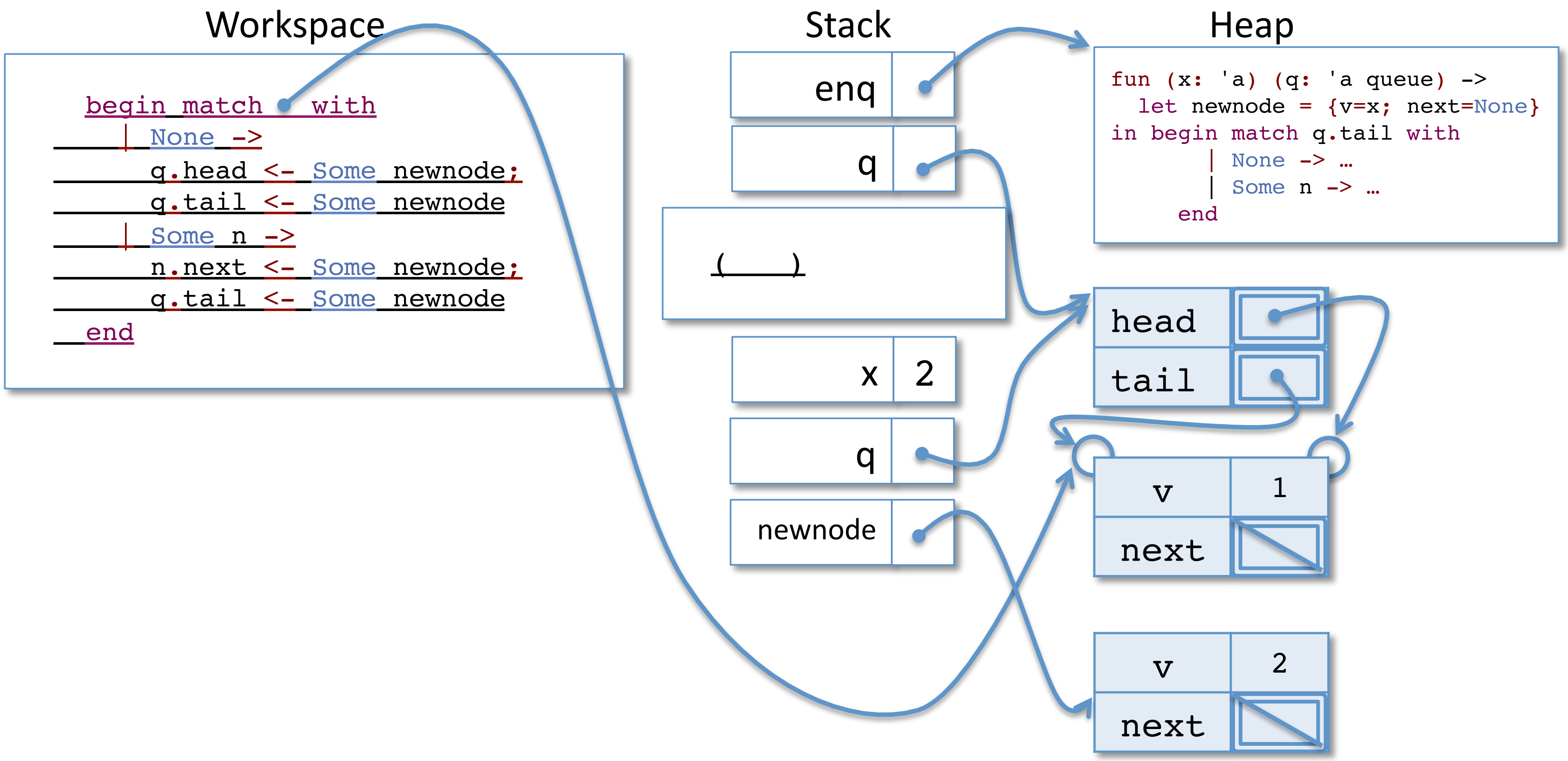
# Calling Enq on a non-empty queue



```
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



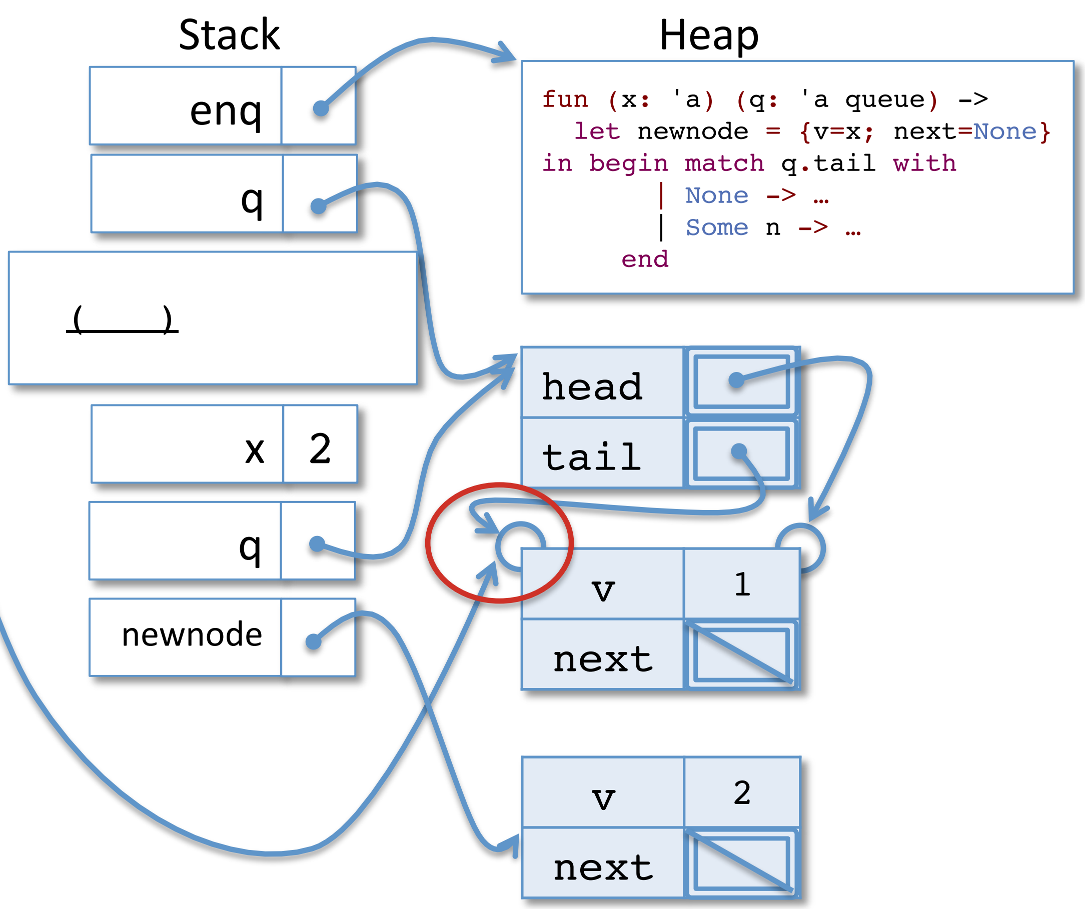
```
  begin match ● with
      | None ->
        q.head <- Some newnode;
        q.tail <- Some newnode
      | Some n ->
        n.next <- Some newnode;
        q.tail <- Some newnode
  end
```

```
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

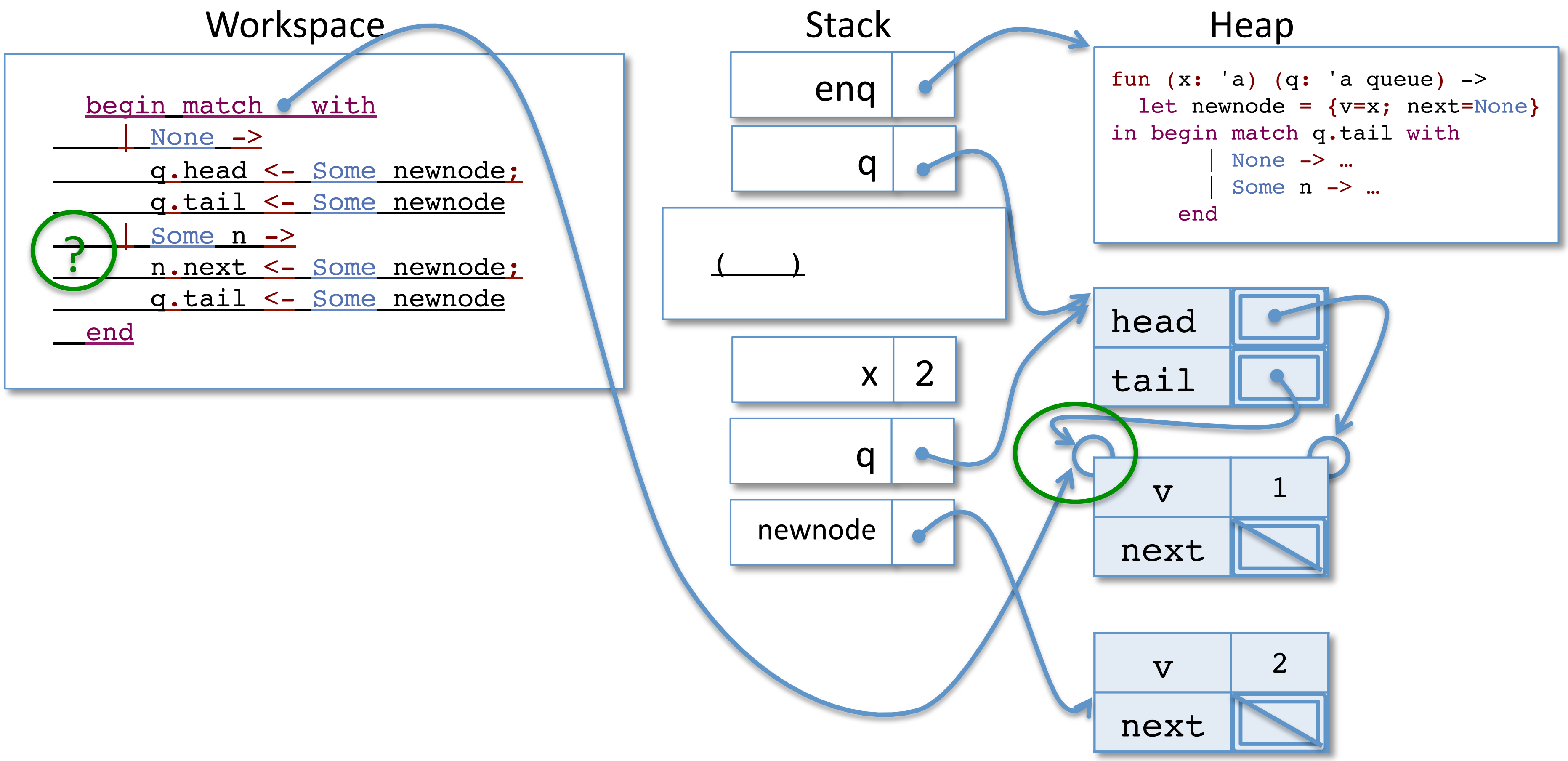
Workspace

```
  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:

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

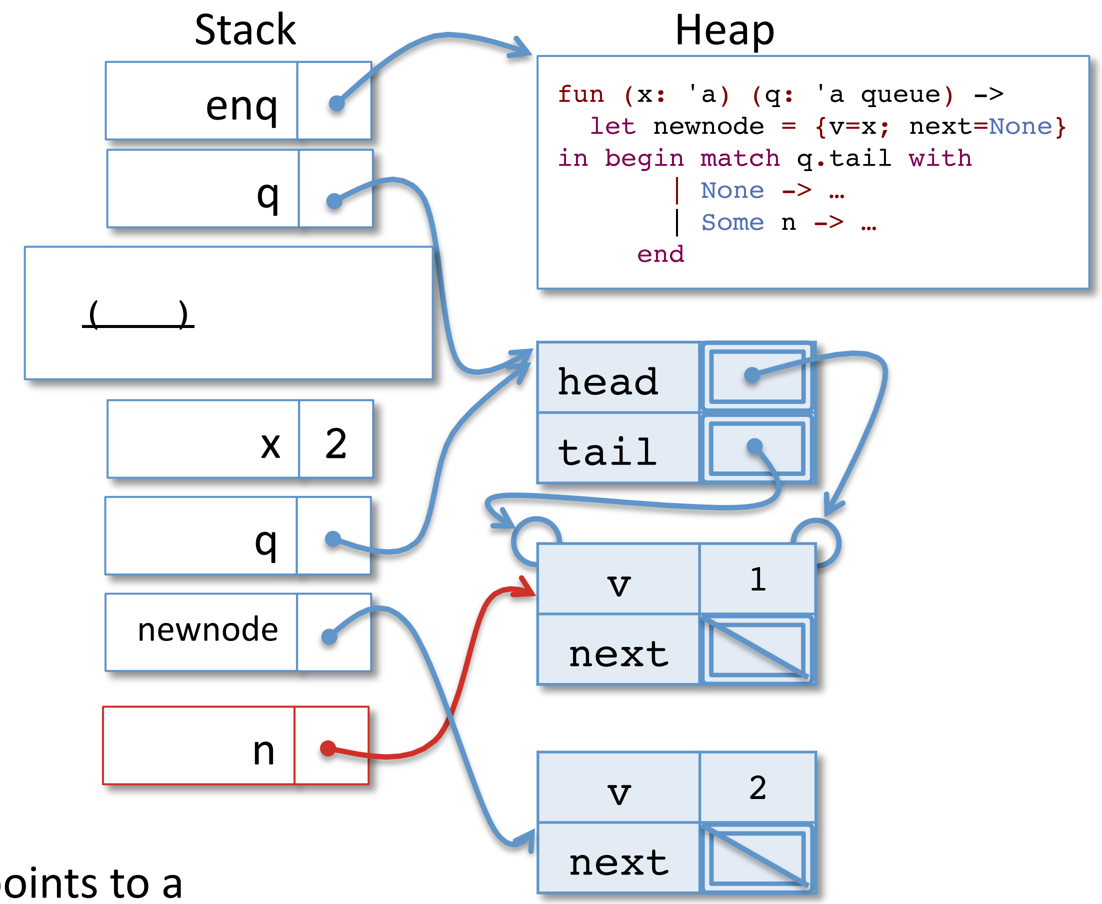
Heap:

```
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

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



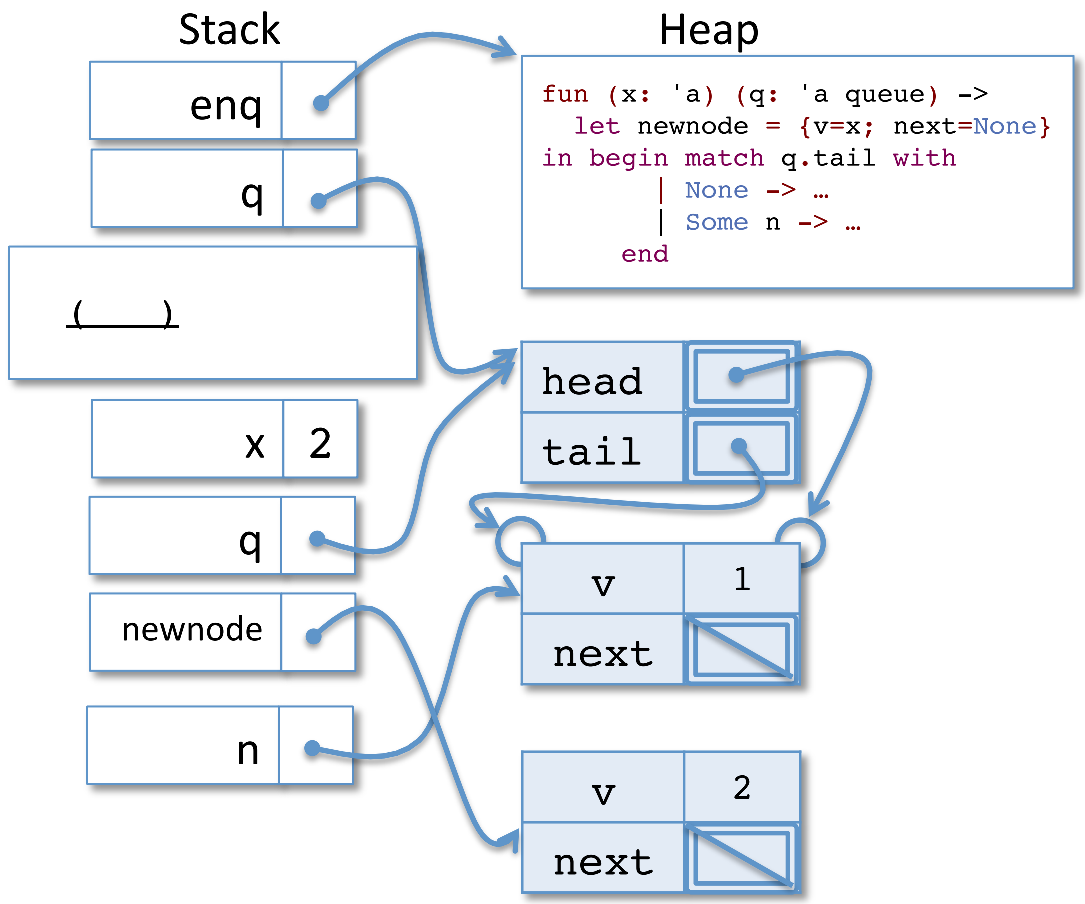
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```

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

# Calling Enq on a non-empty queue

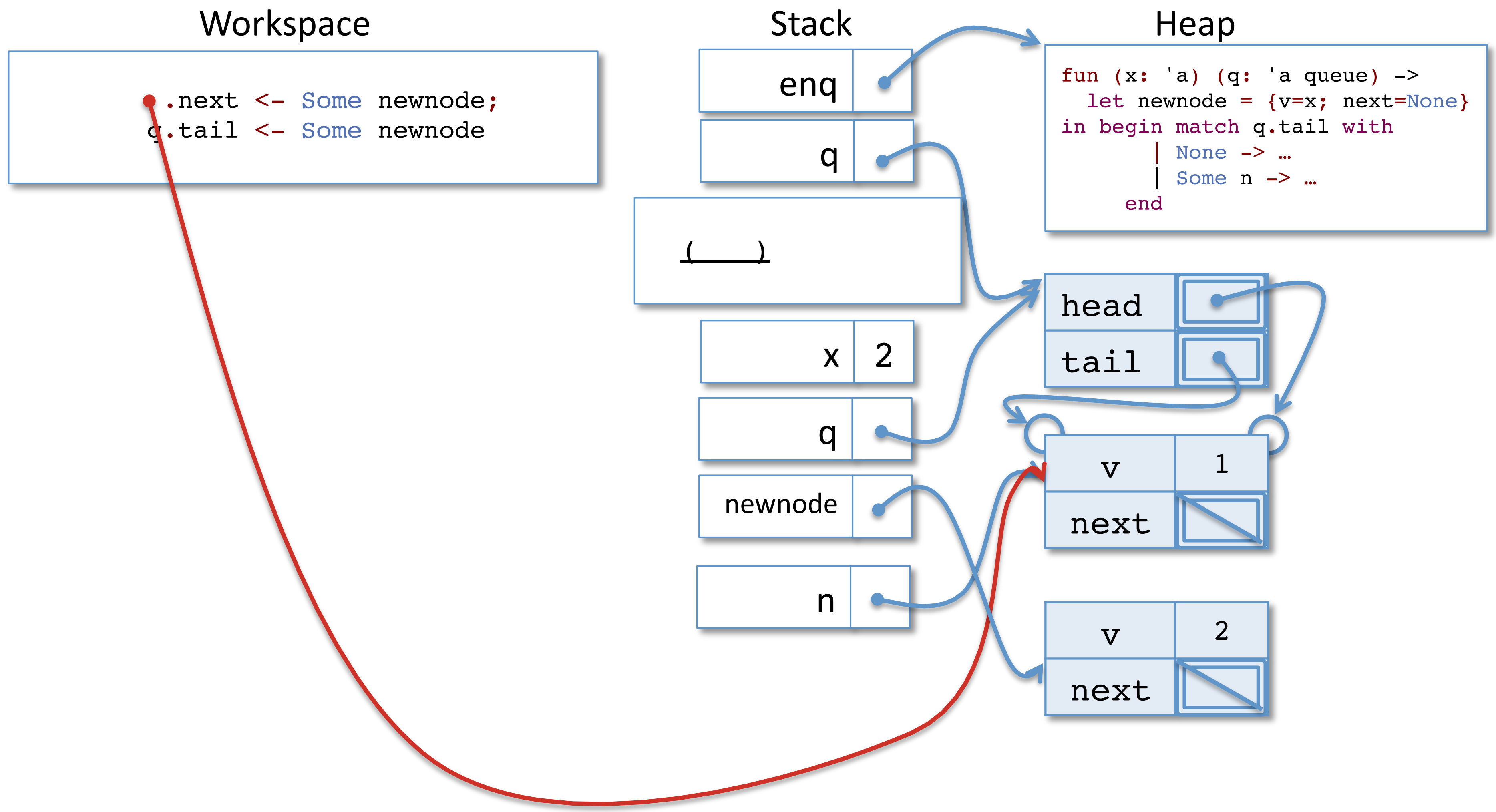
Workspace

```
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
```

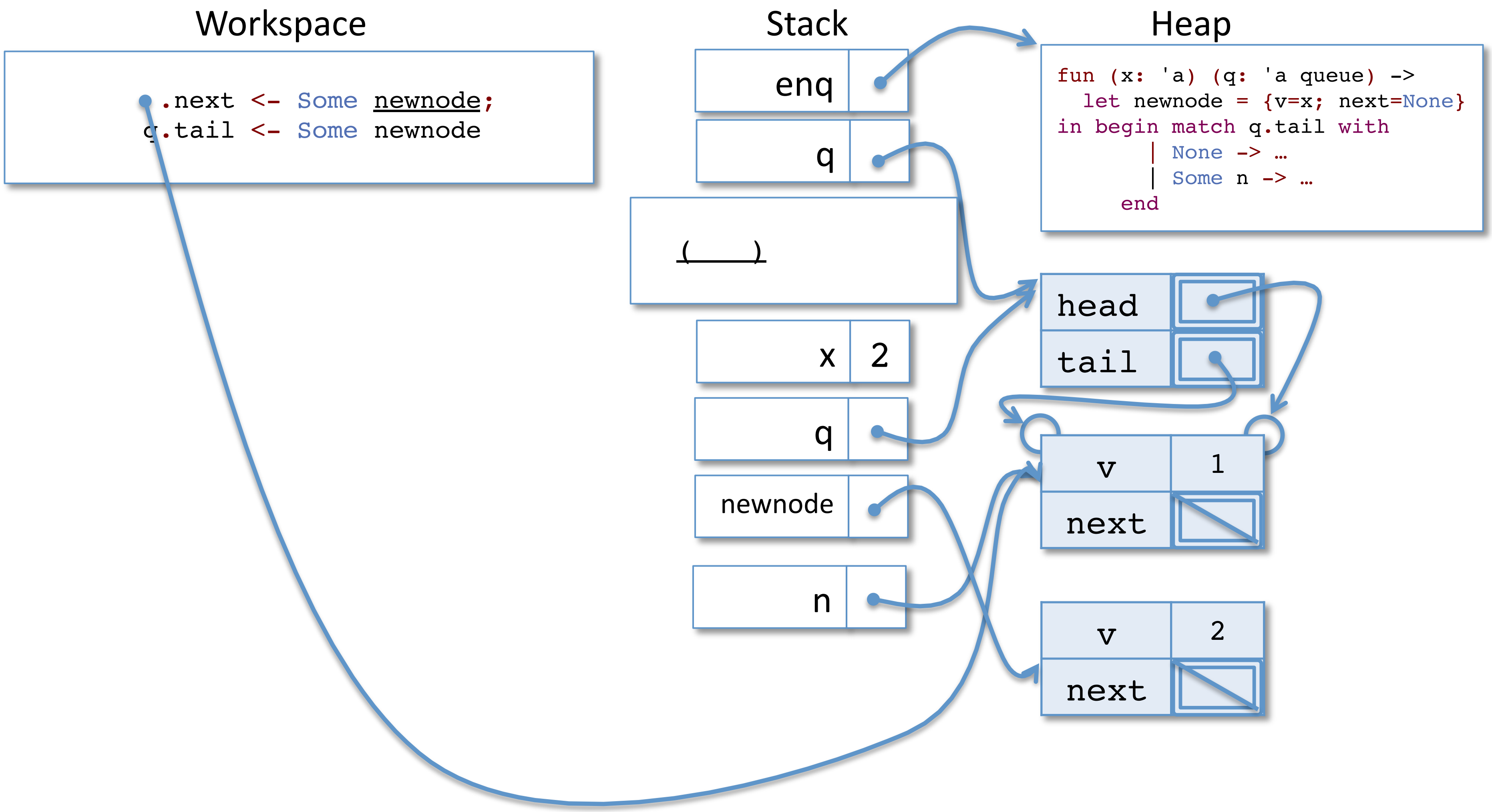
# Calling Enq on a non-empty queue



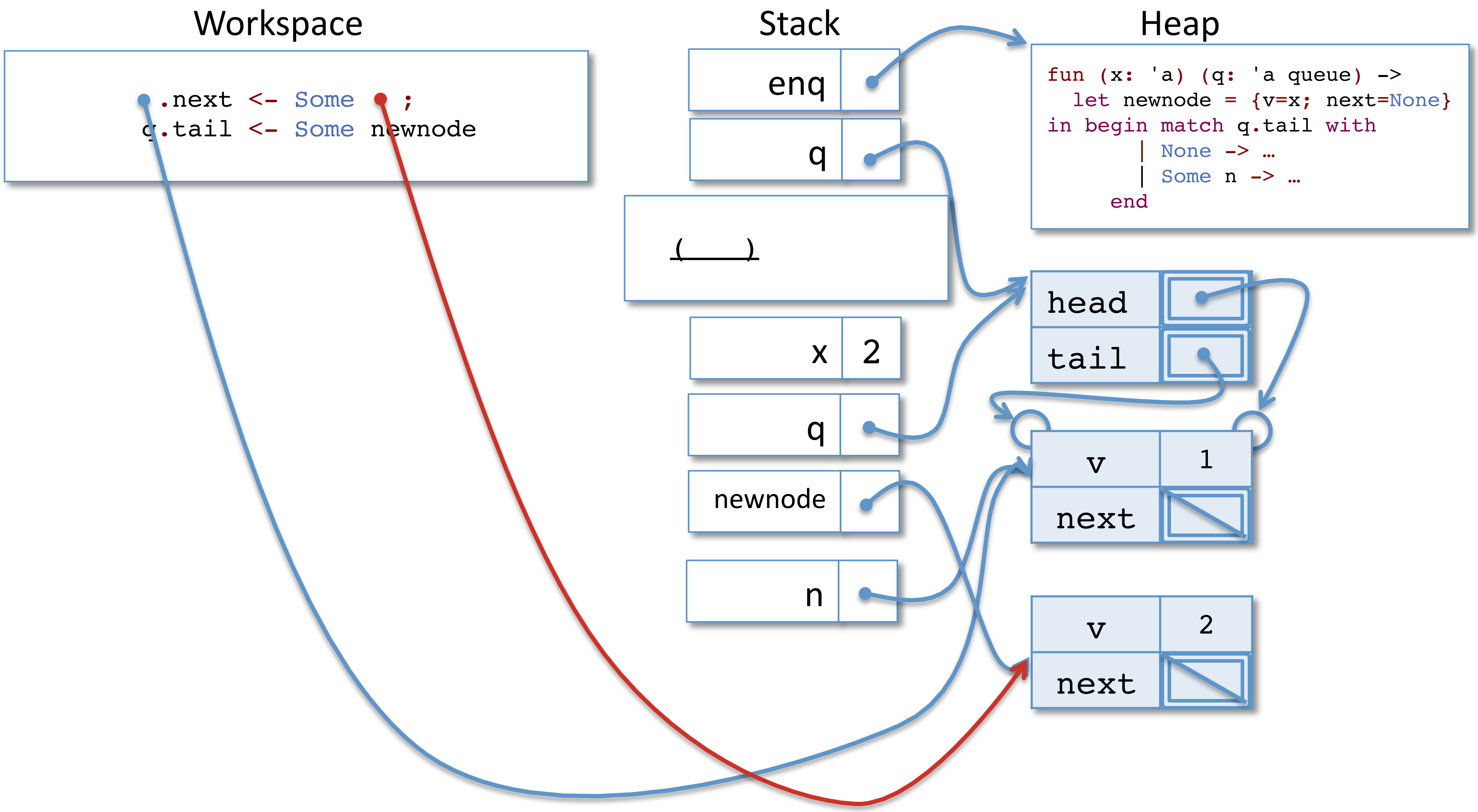
```
.next <- Some newnode;
q.tail <- Some newnode
```

```
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

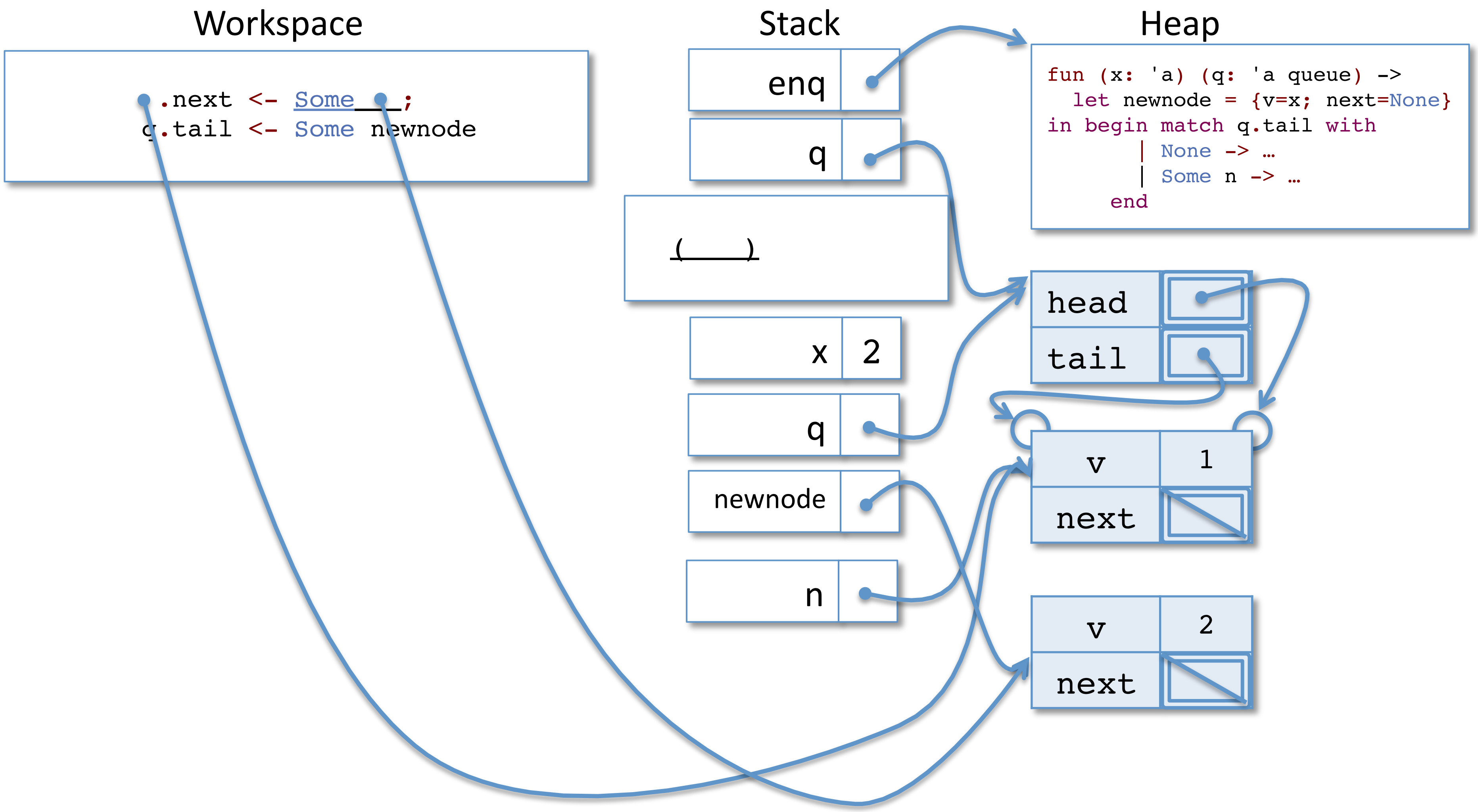


# Calling Enq on a non-empty queue



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

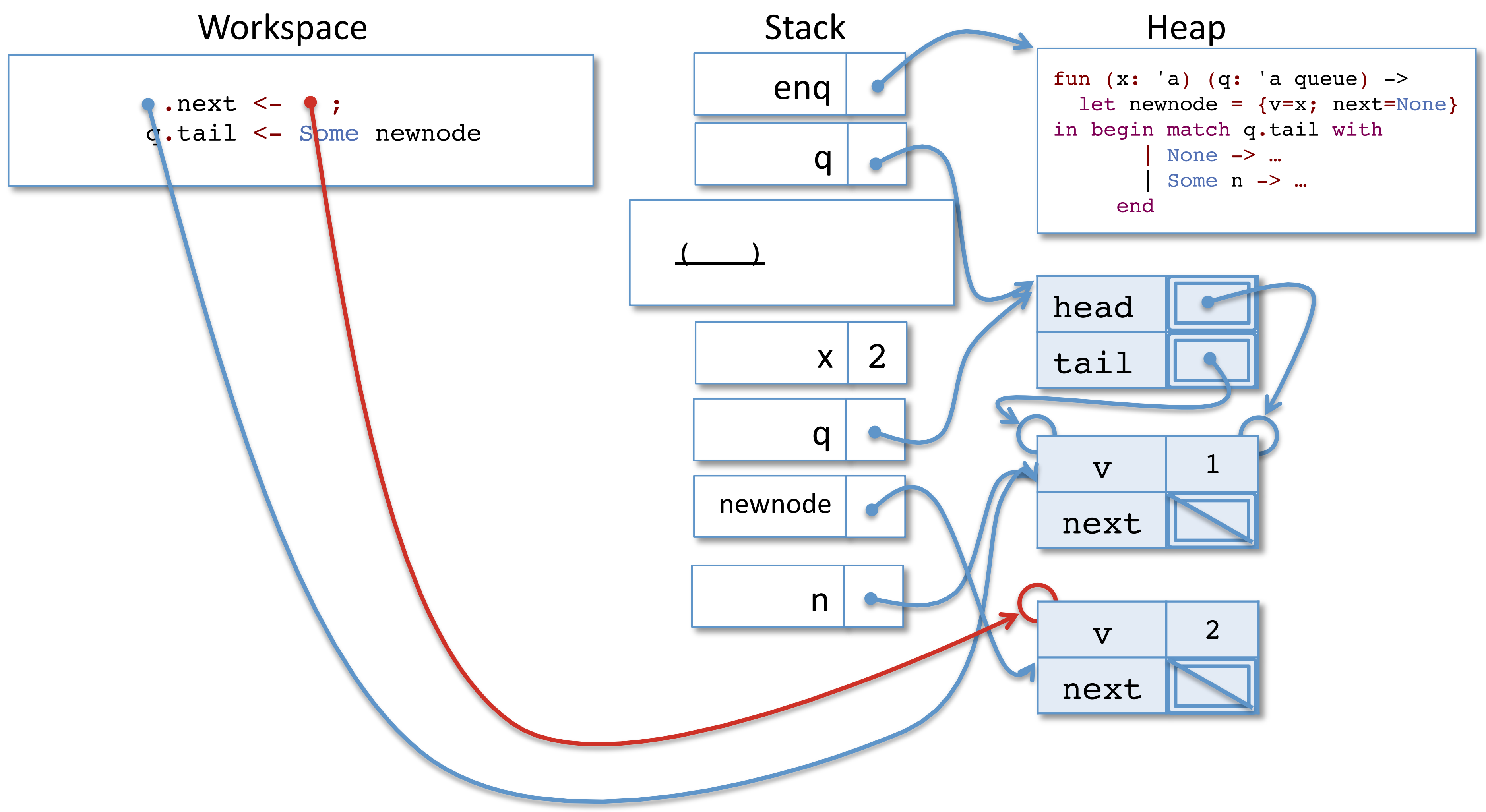
# Calling Enq on a non-empty queue



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

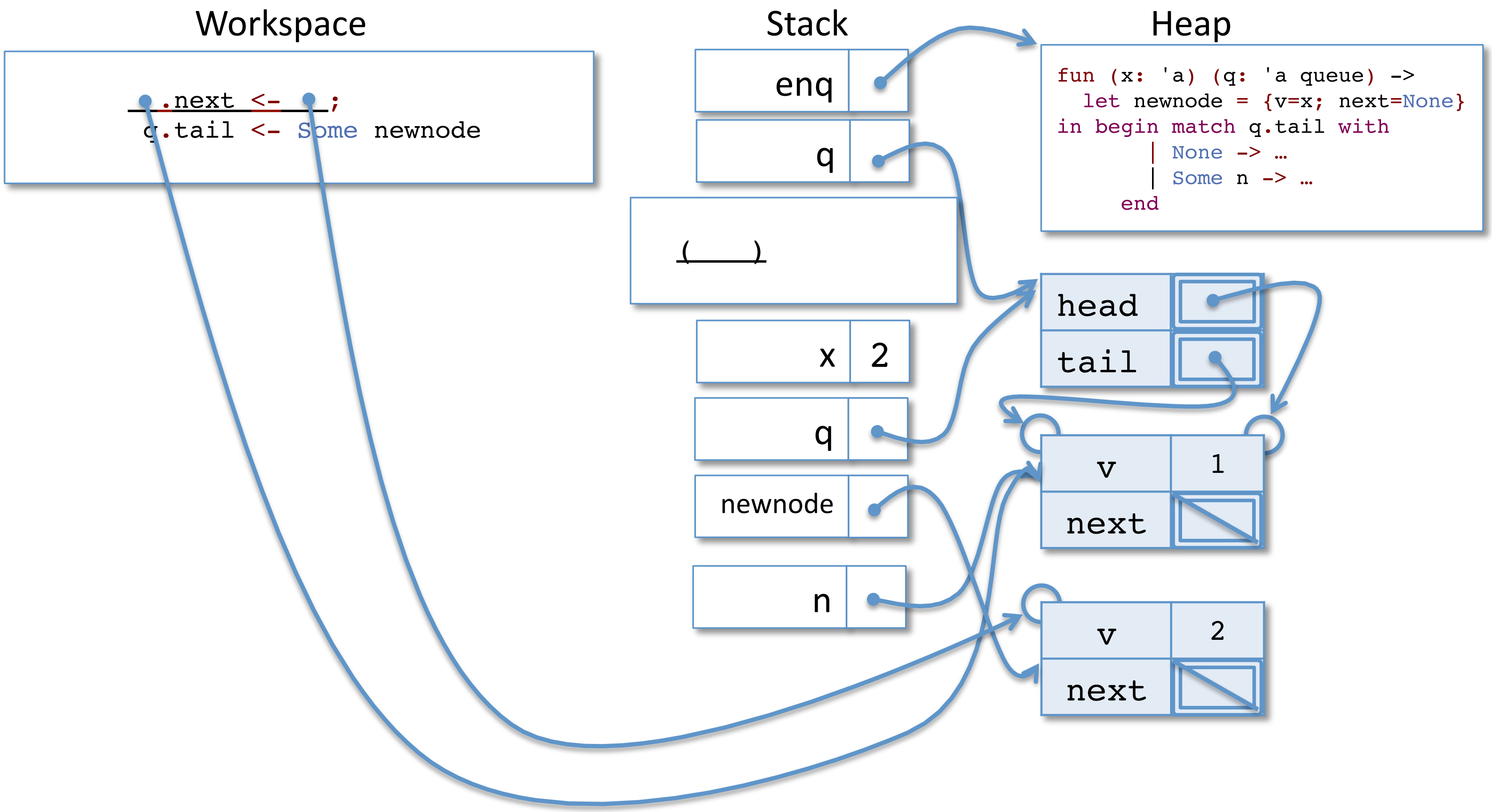
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

# Calling Enq on a non-empty queue



```
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

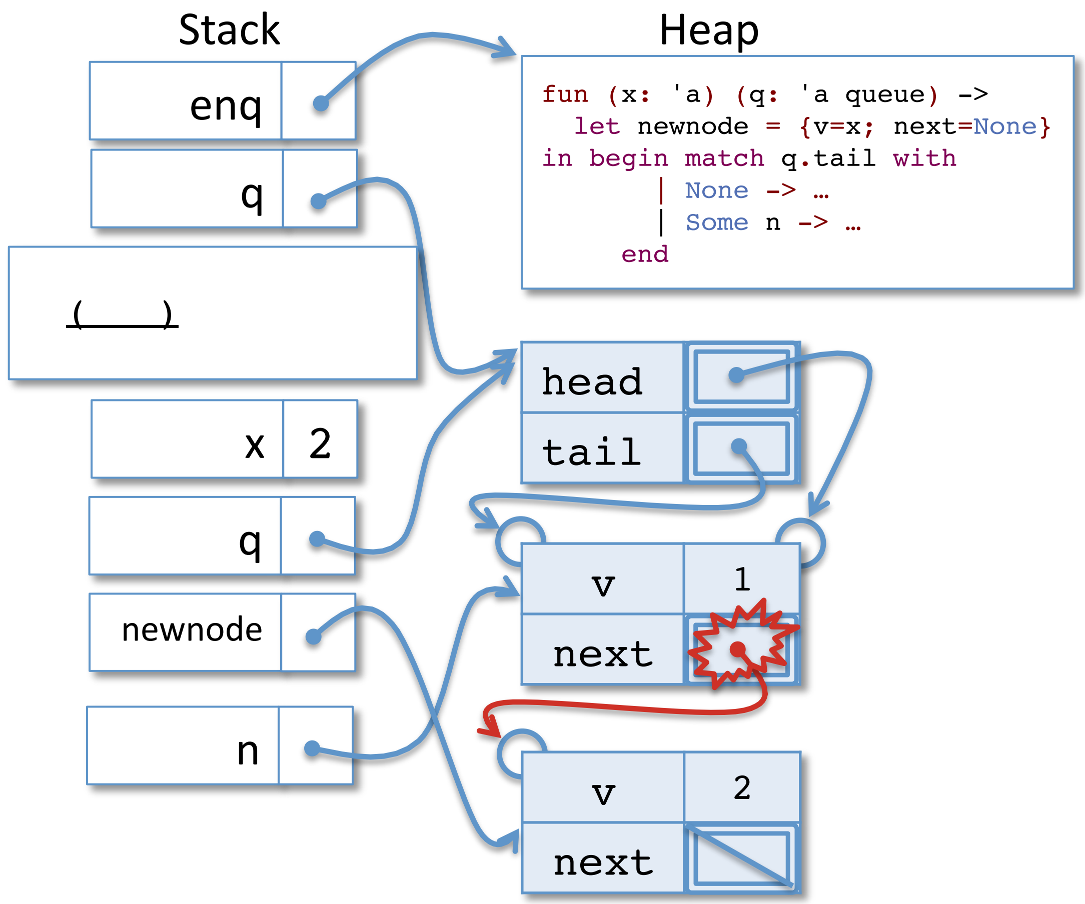


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

# Calling Enq on a non-empty queue

Workspace

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

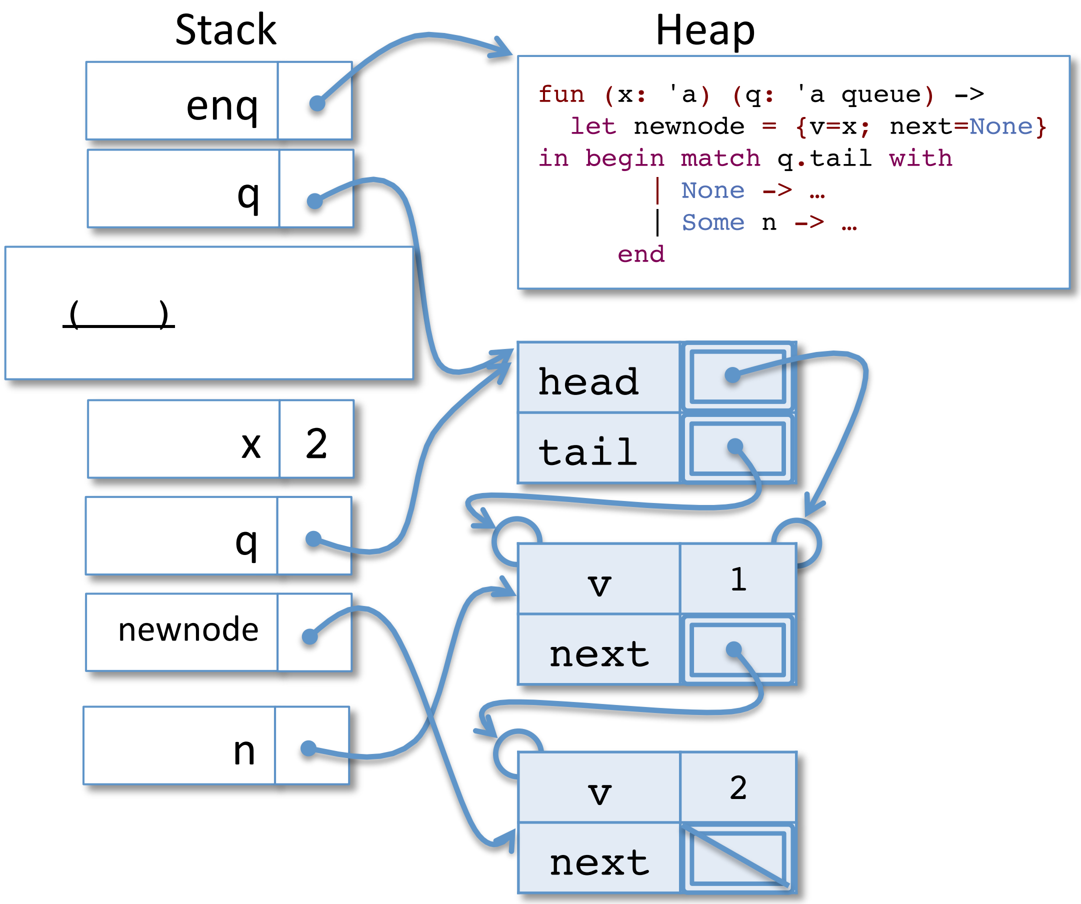


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

# Calling Enq on a non-empty queue

Workspace

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

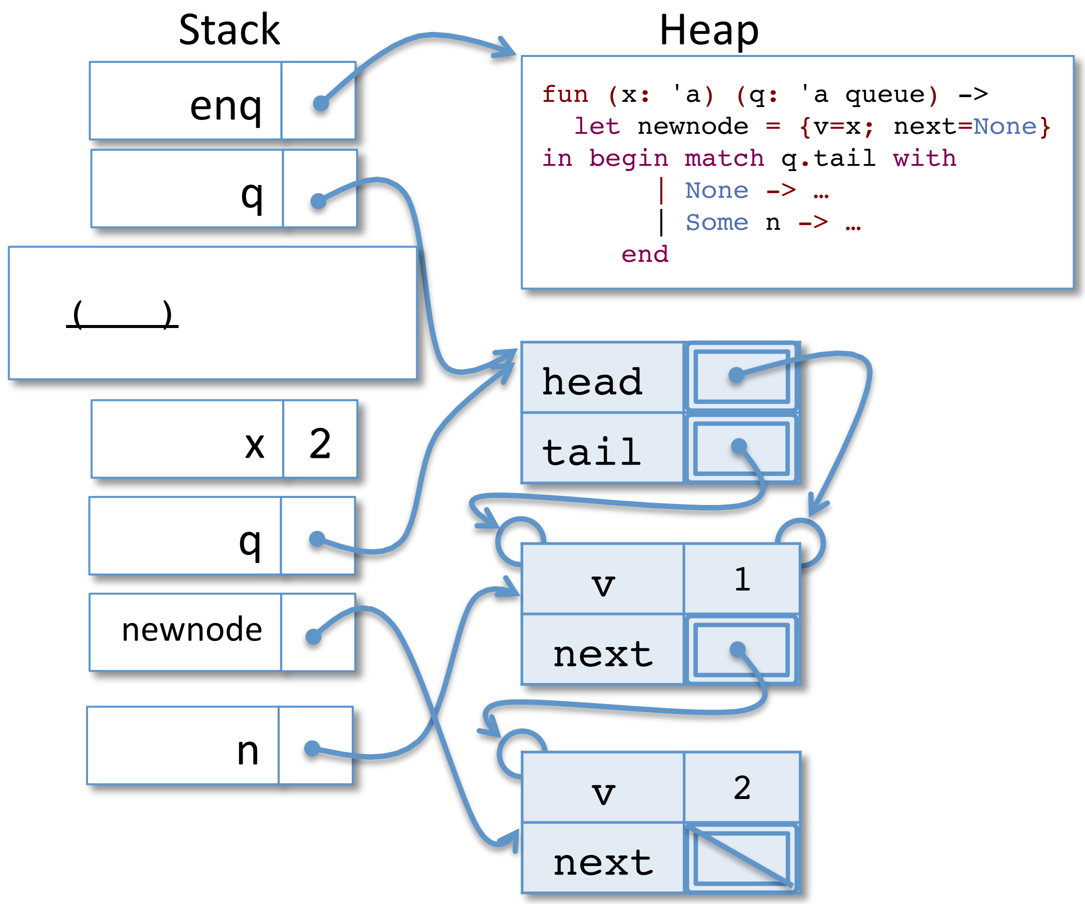


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
          | None -> …
          | Some n -> …
     end
```

# Calling Enq on a non-empty queue

Workspace

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

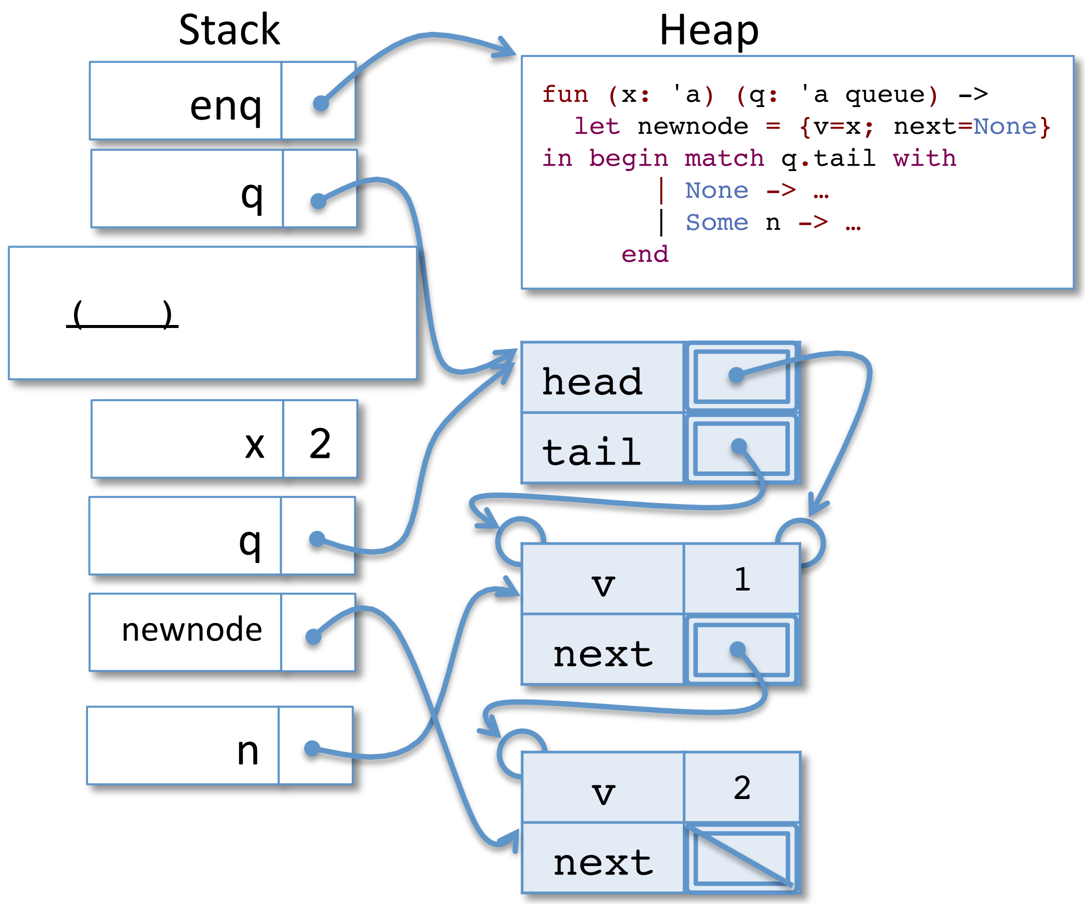


```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

# Calling Enq on a non-empty queue

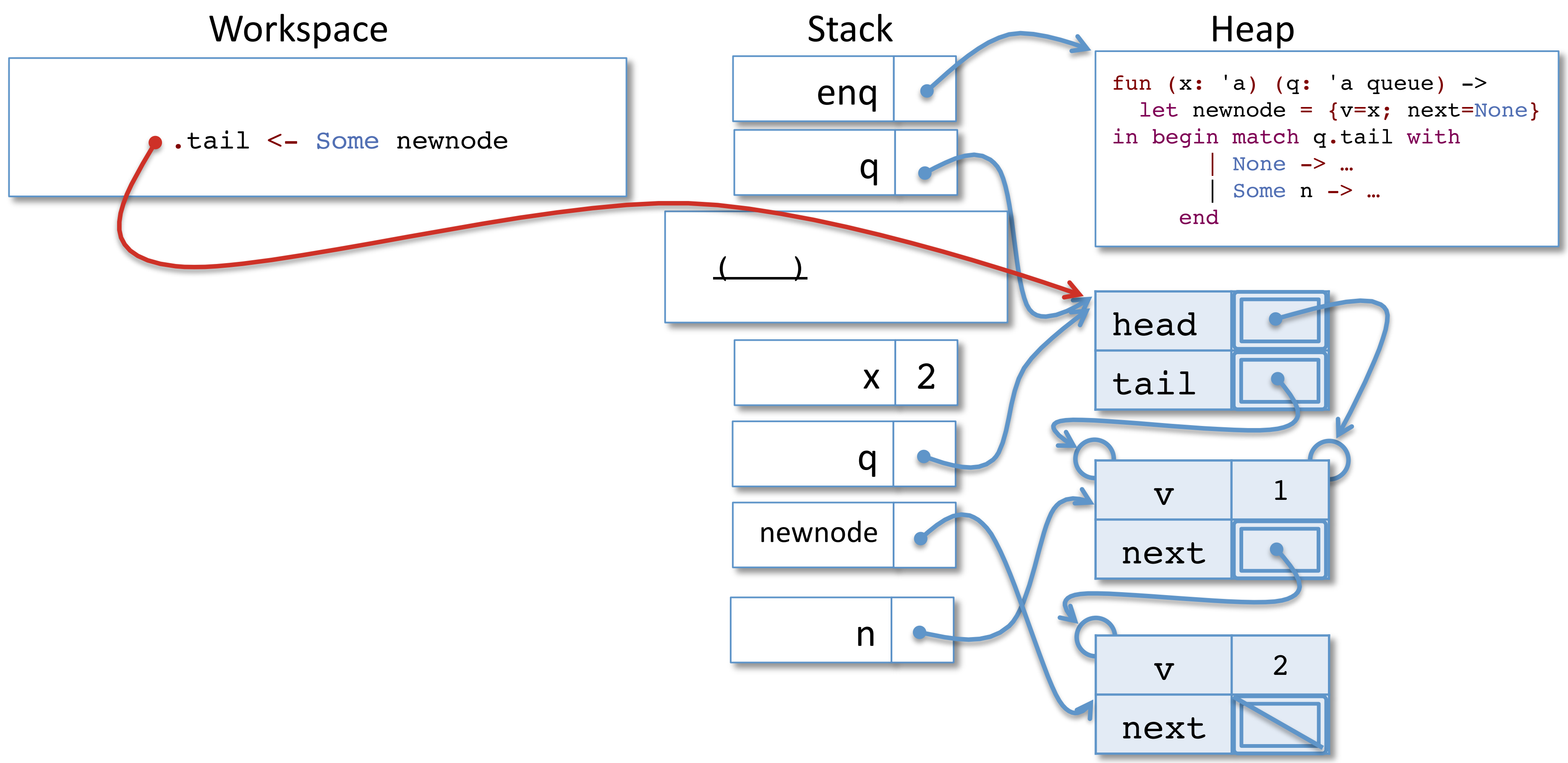
Workspace

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



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

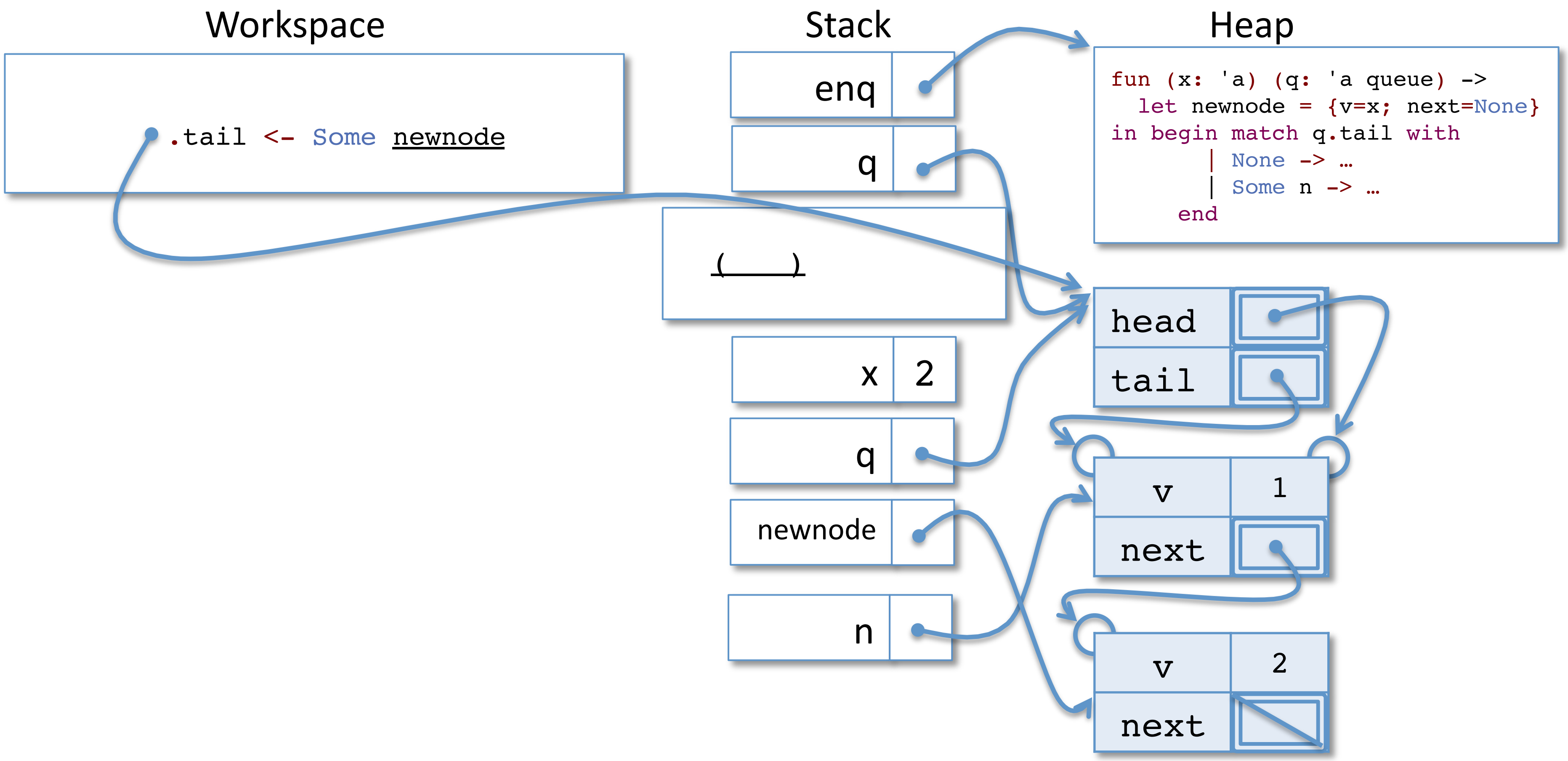
# Calling Enq on a non-empty queue



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

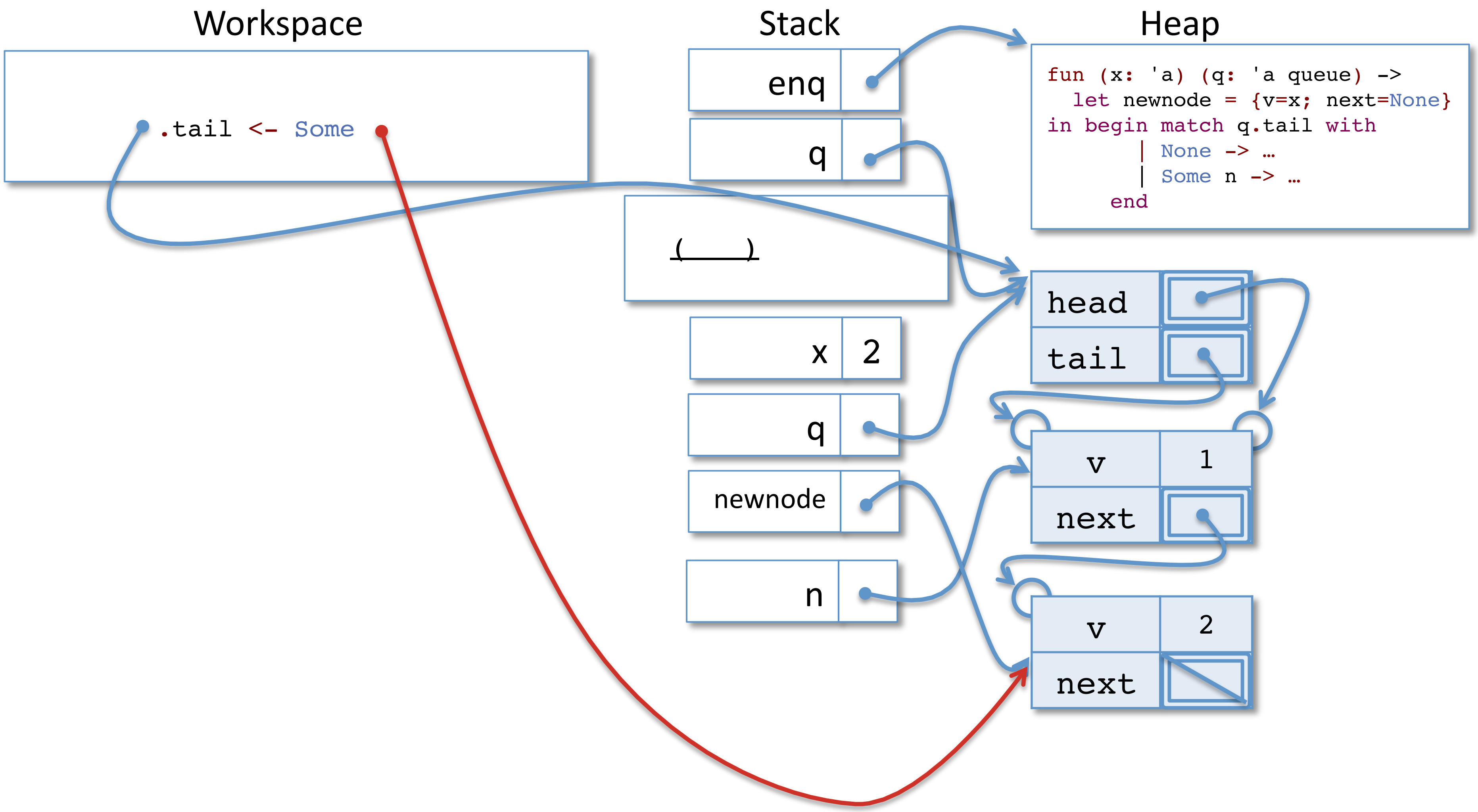
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```

# Calling Enq on a non-empty queue



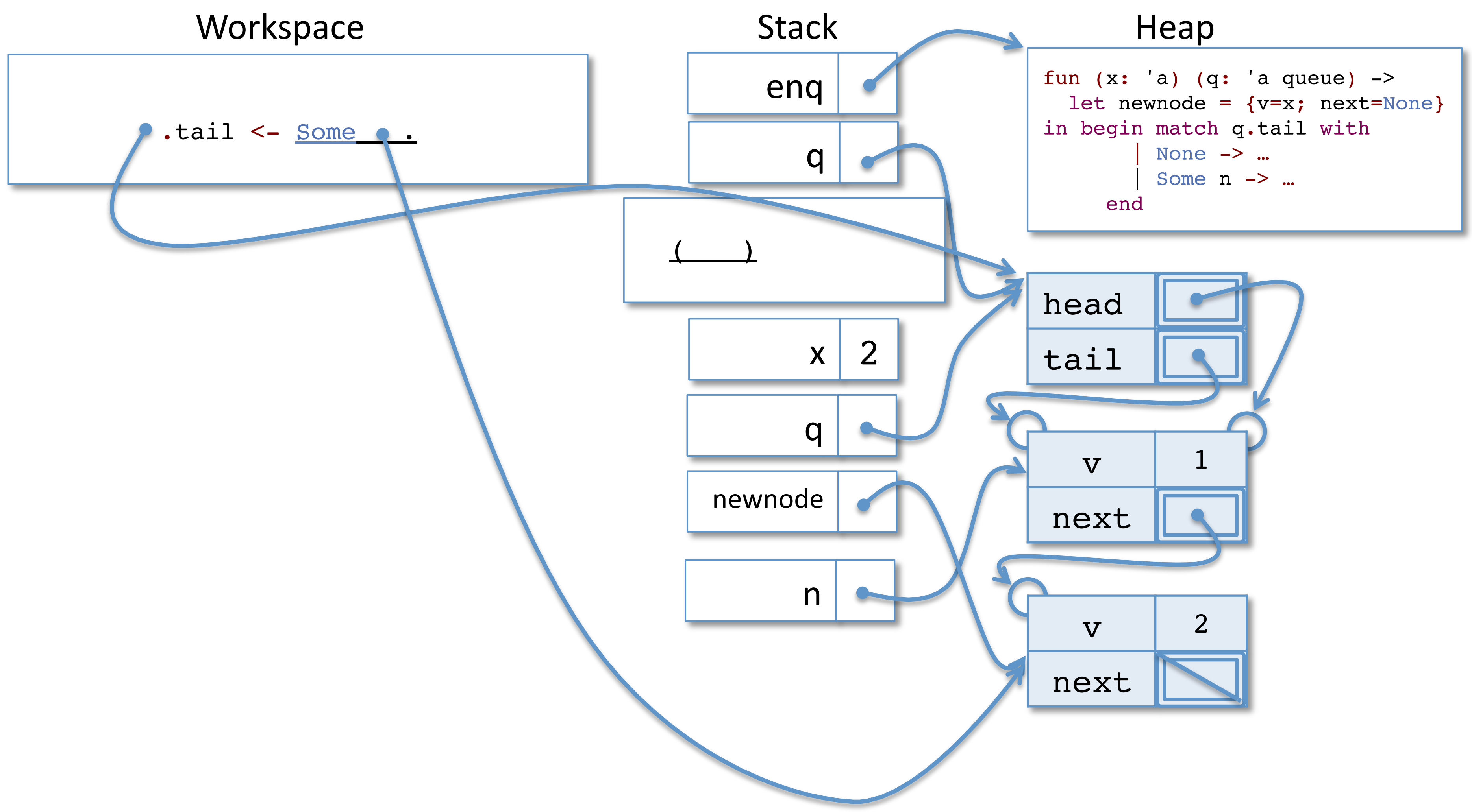
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

# Calling Enq on a non-empty queue



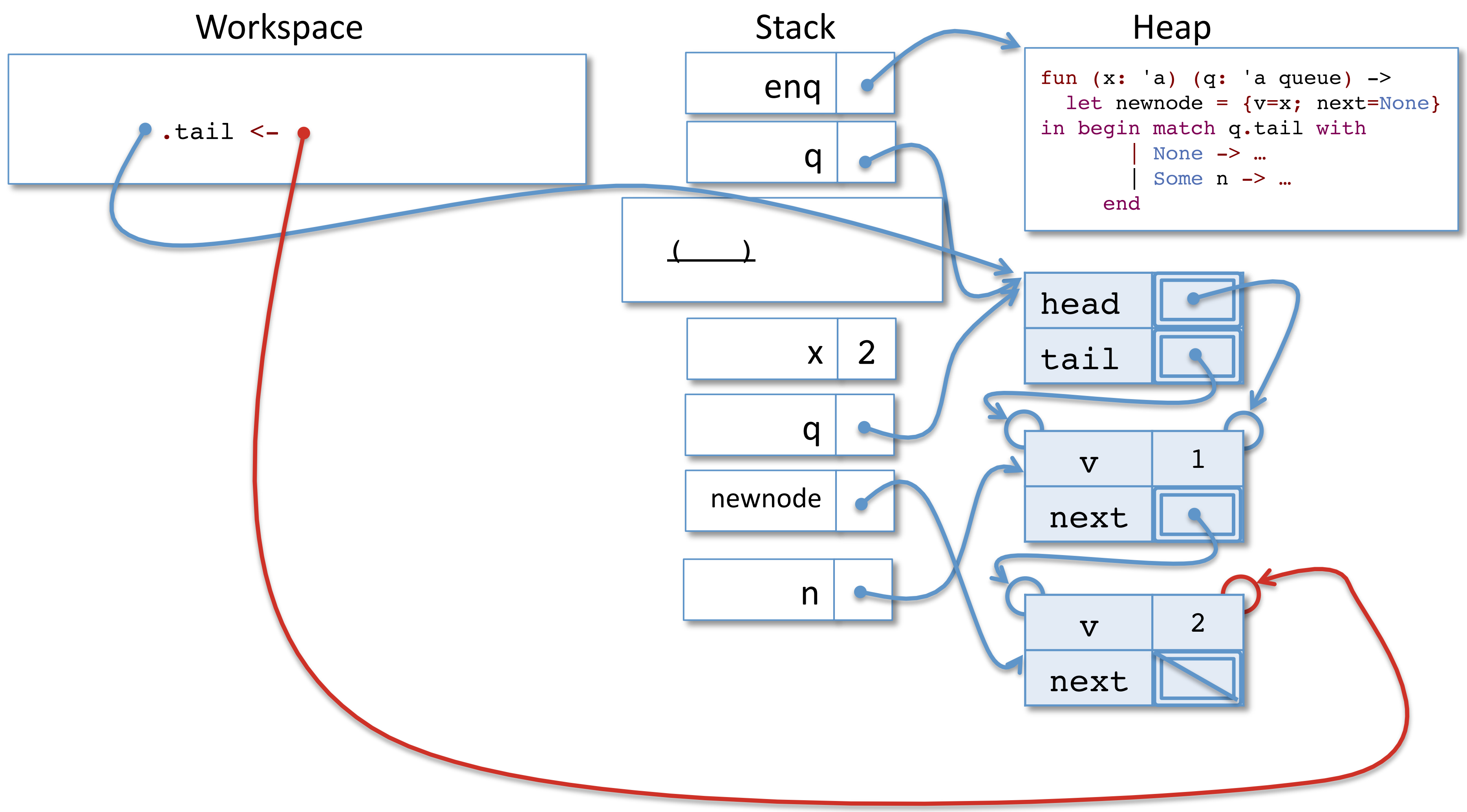
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
     end
```

# Calling Enq on a non-empty queue



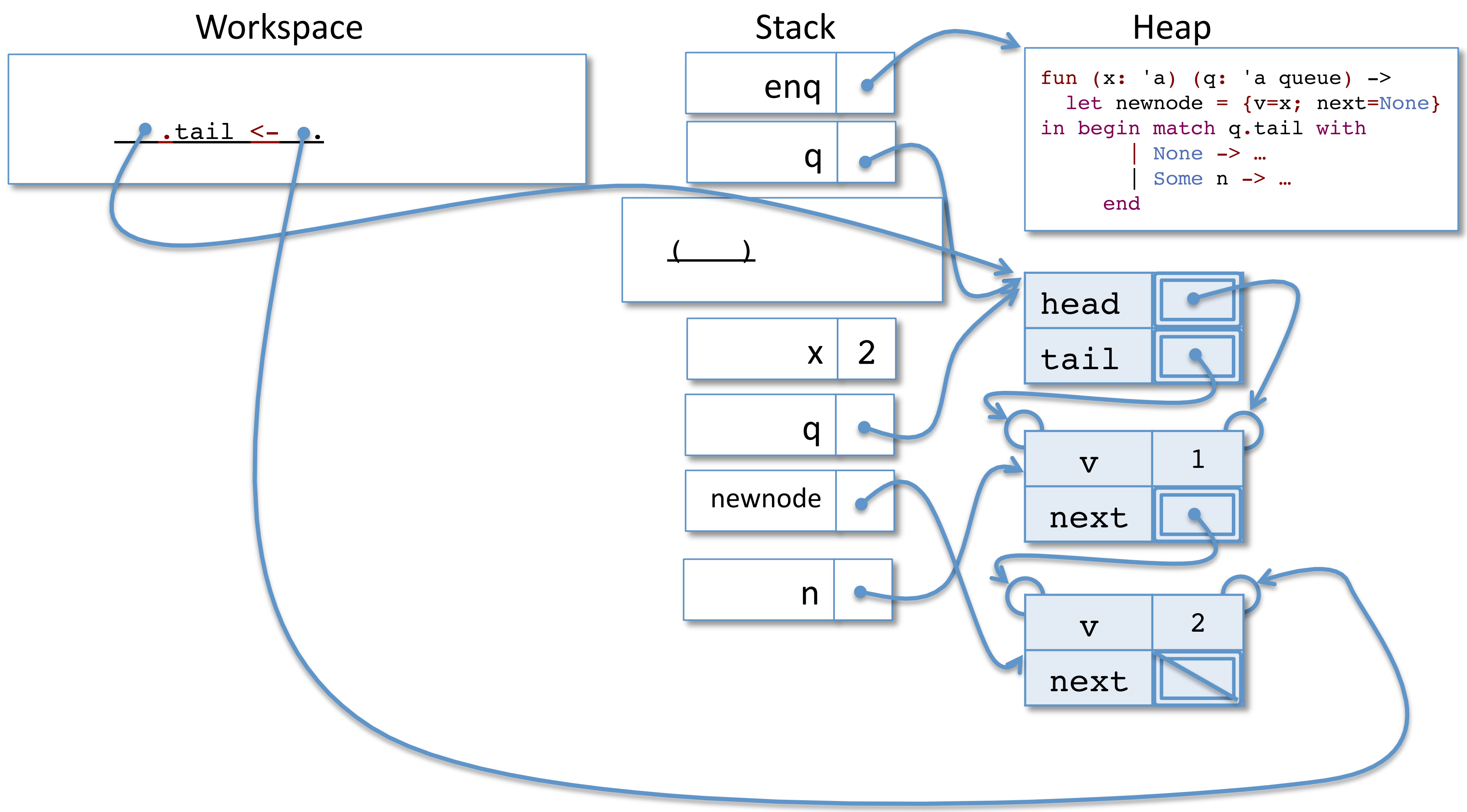
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
       | None -> …
       | Some n -> …
    end
```

# Calling Enq on a non-empty queue



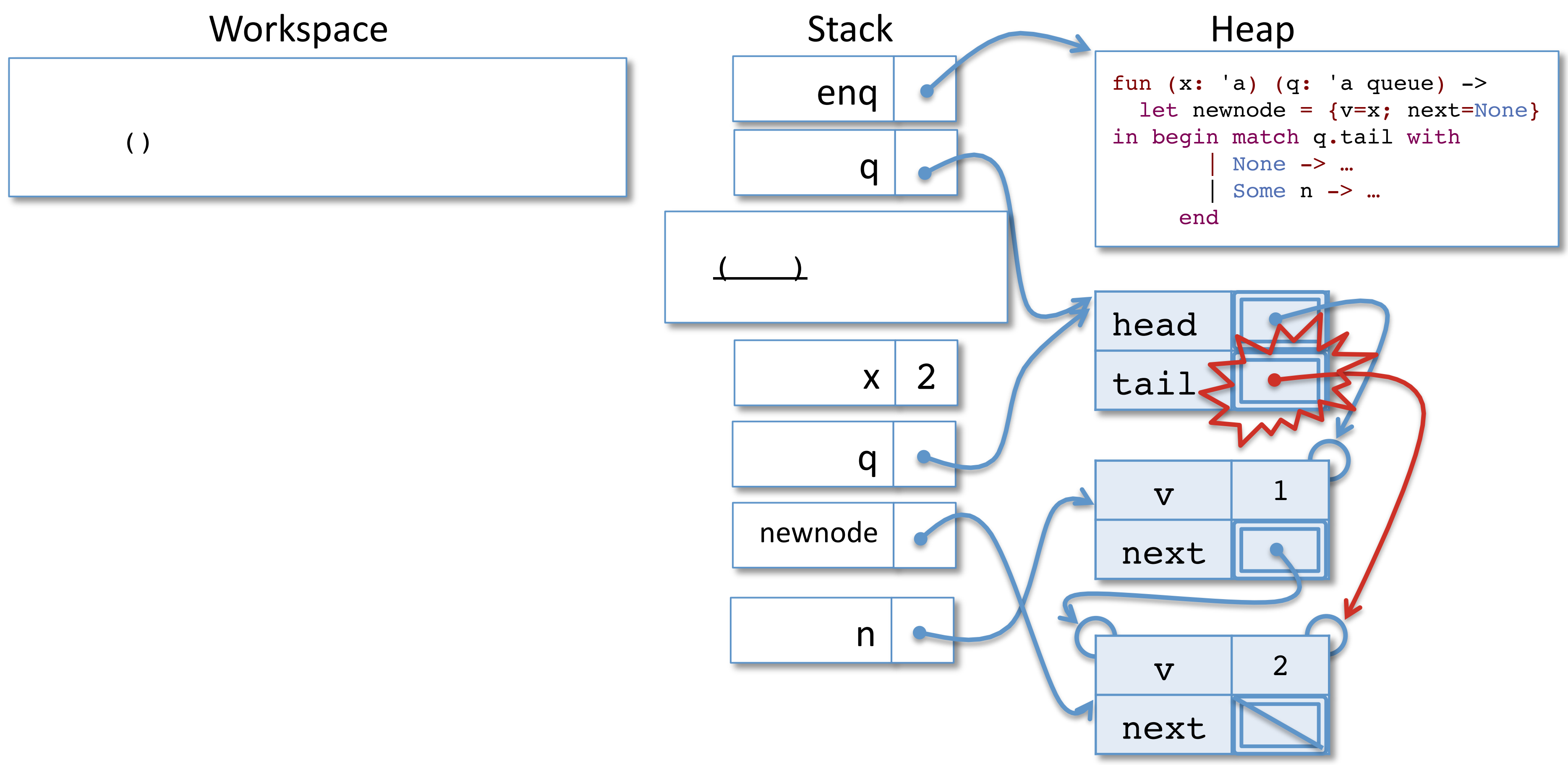
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

# Calling Enq on a non-empty queue



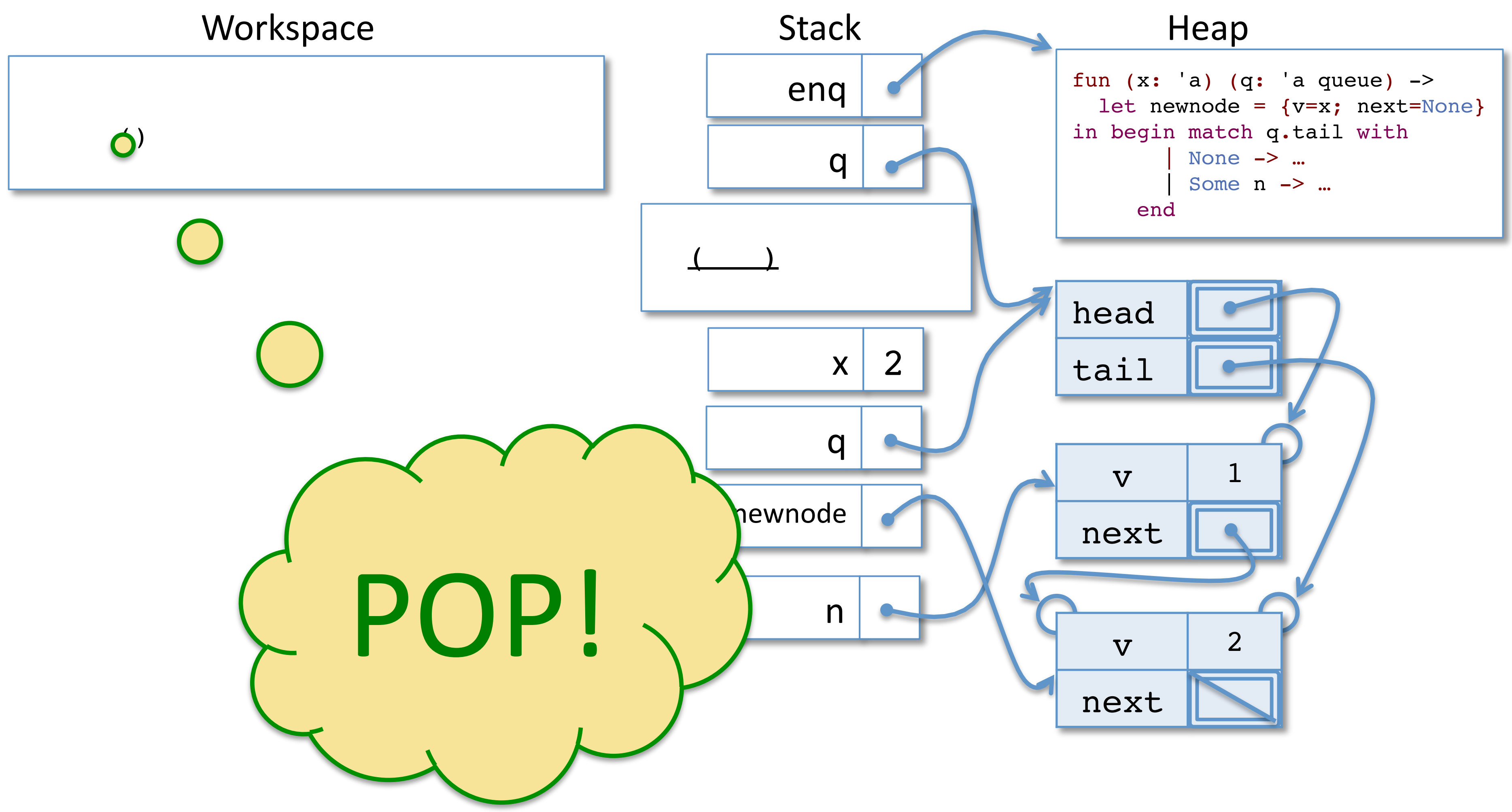
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
     end
```

# Calling Enq on a non-empty queue



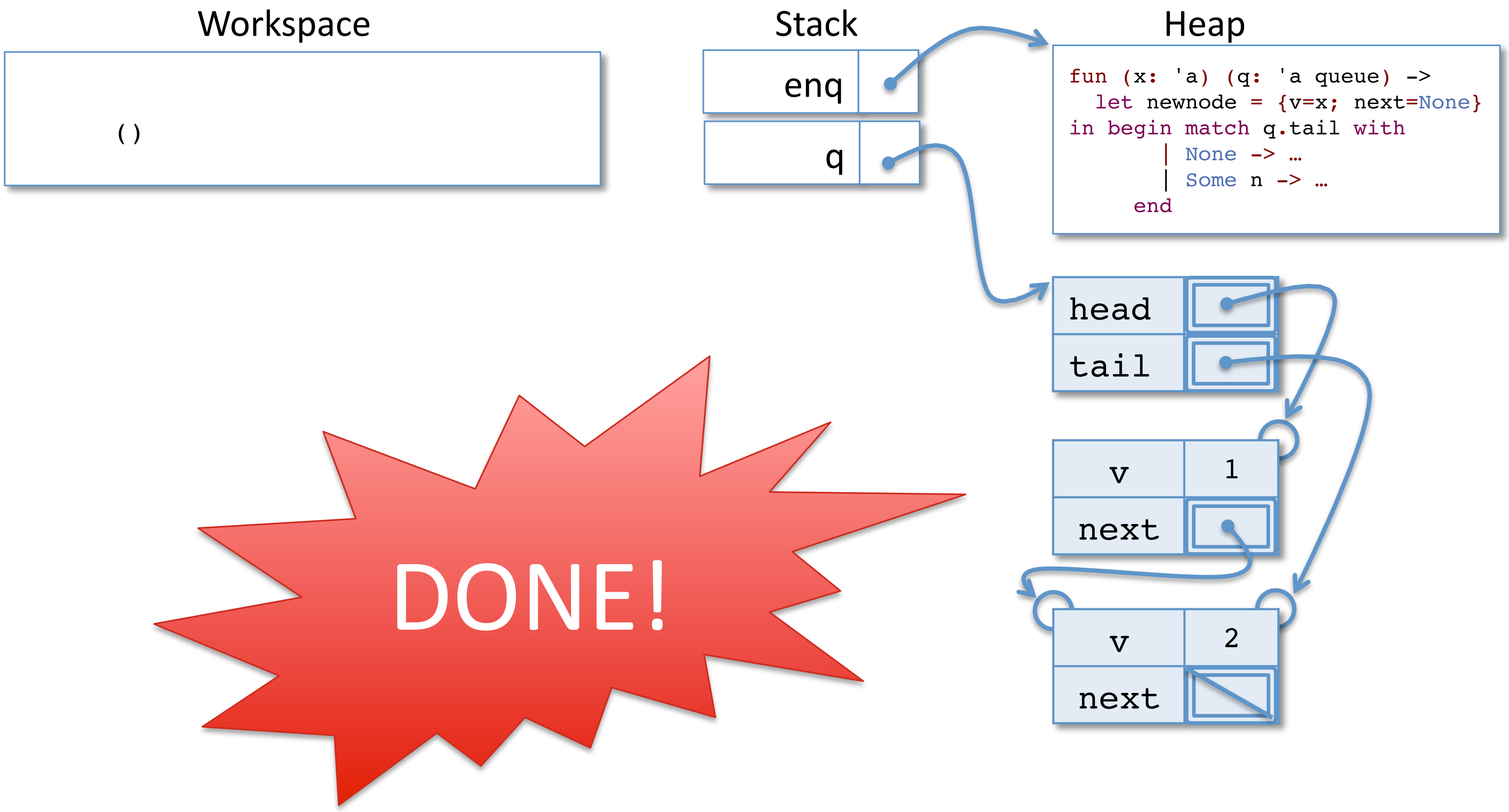
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
         | None -> …
         | Some n -> …
     end
```

# Calling Enq on a non-empty queue



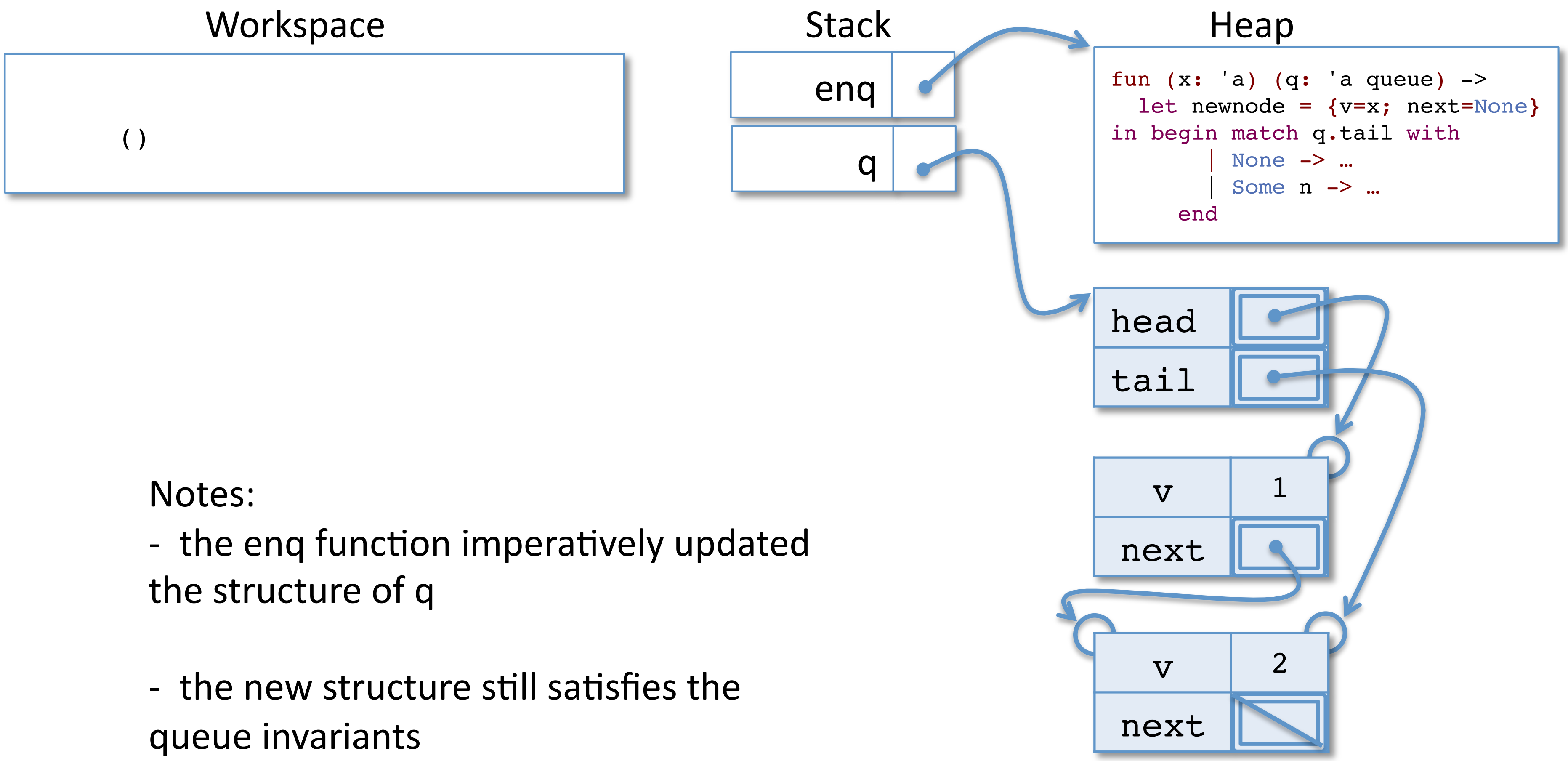
```
fun (x: 'a) (q: 'a queue) ->
  let newnode = {v=x; next=None}
in begin match q.tail with
        | None -> …
        | Some n -> …
    end
```

# 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
```

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