Programming Languages and Techniques (CIS1200)

Lecture 15

ASM (Functions), Queues

Lecture notes: Chapter 16

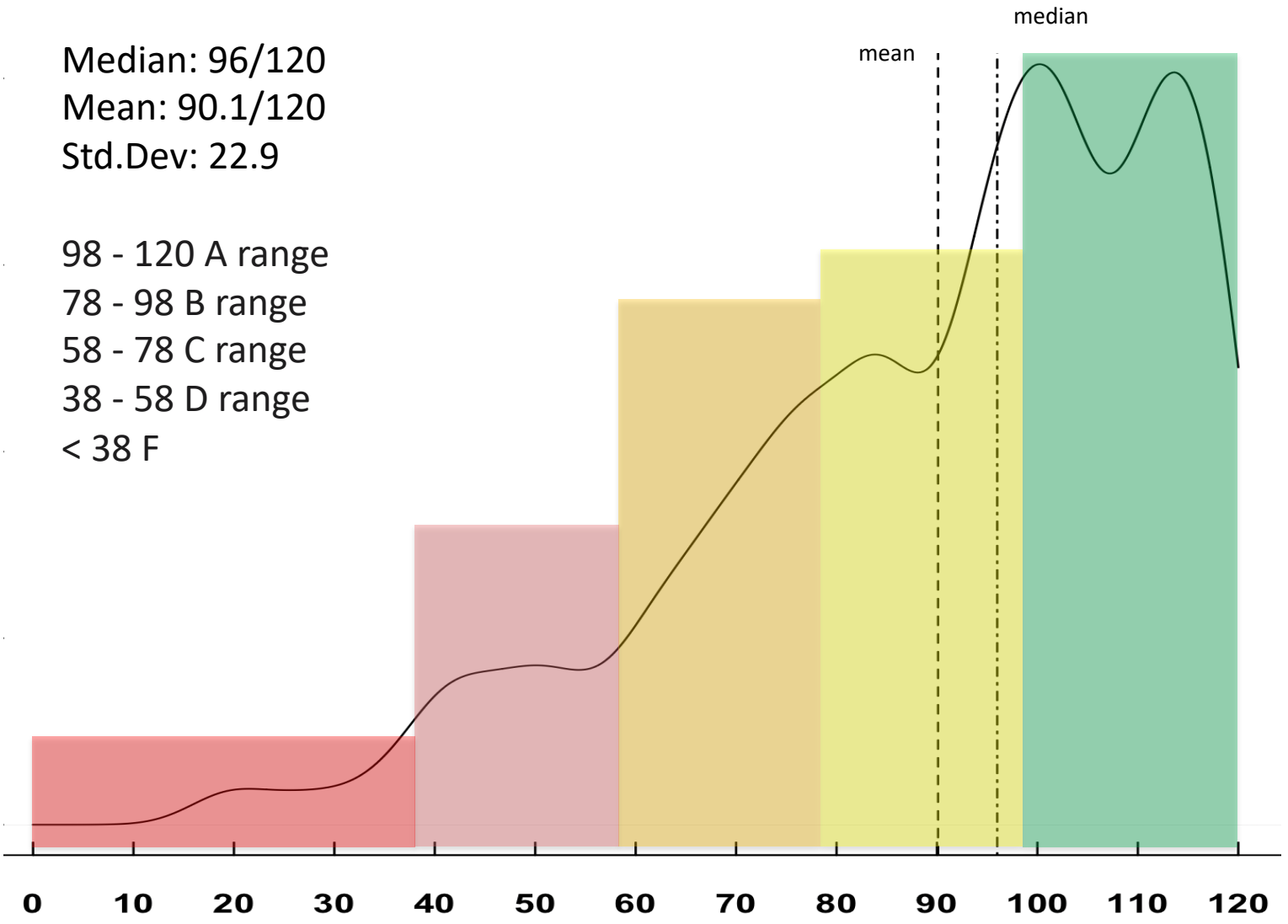
Announcements

- Midterm 1 Grades and Solutions available soon
  - Posted this afternoon
  - Regrade requests via Gradescope
    - Opens on Wednesday, October 9th
    - Due by Friday, October 18th
- HW04 available
  - due next Tuesday, October 15th
- No office hours for Benjamin today

Midterm 1 Analysis

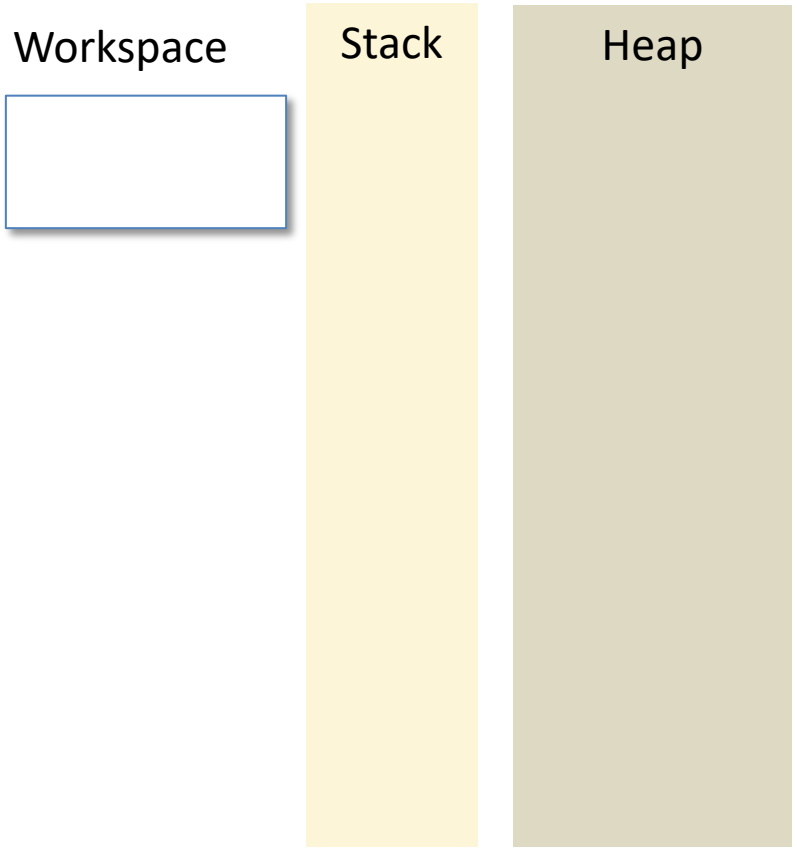
Median: 96/120
Mean: 90.1/120
Std.Dev: 22.9

98 - 120 A range
78 - 98 B range
58 - 78 C range
38 - 58 D range
< 38 F



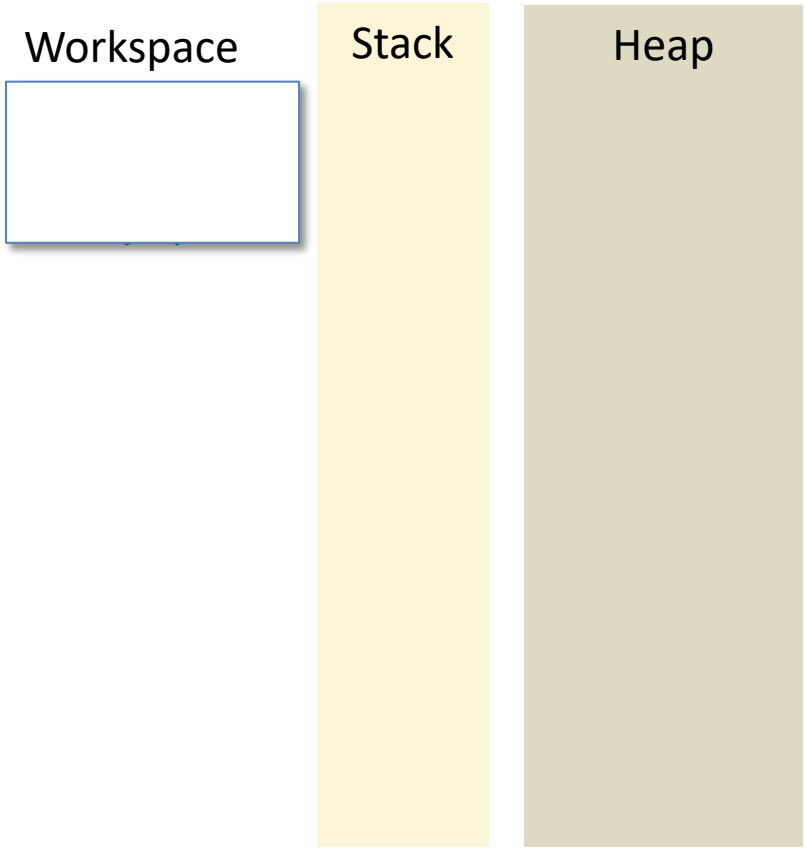
Review: Abstract Stack Machine

- Three "spaces"
  - workspace
    - the expression the computer is currently working on simplifying
  - stack
    - temporary storage for `let` bindings and partially simplified expressions
  - heap
    - storage area for large data structures
- Initial state:
  - workspace contains whole program
  - stack and heap are empty
- Machine operation:
  - In each step, choose next part of the workspace expression and simplify it
  - (Sometimes this will also involve changes to the stack and/or heap)
  - Stop when there are no more simplifications to be done



Abstract Stack Machine

- Three "spaces"
  - workspace
    - the expression the computer is currently working on simplifying
  - stack
    - temporary storage for `let` bindings and partially simplified expressions
  - heap
    - storage area for large data structures
- Initial state:
  - workspace contains whole program
  - stack and heap are empty
- Machine operation:
  - In each step, choose next part of the workspace expression and simplify it
  - (Sometimes this will also involve changes to the stack and/or heap)
  - Stop when there are no more simplifications to be done



Abstract stack machine

Review: Values and References

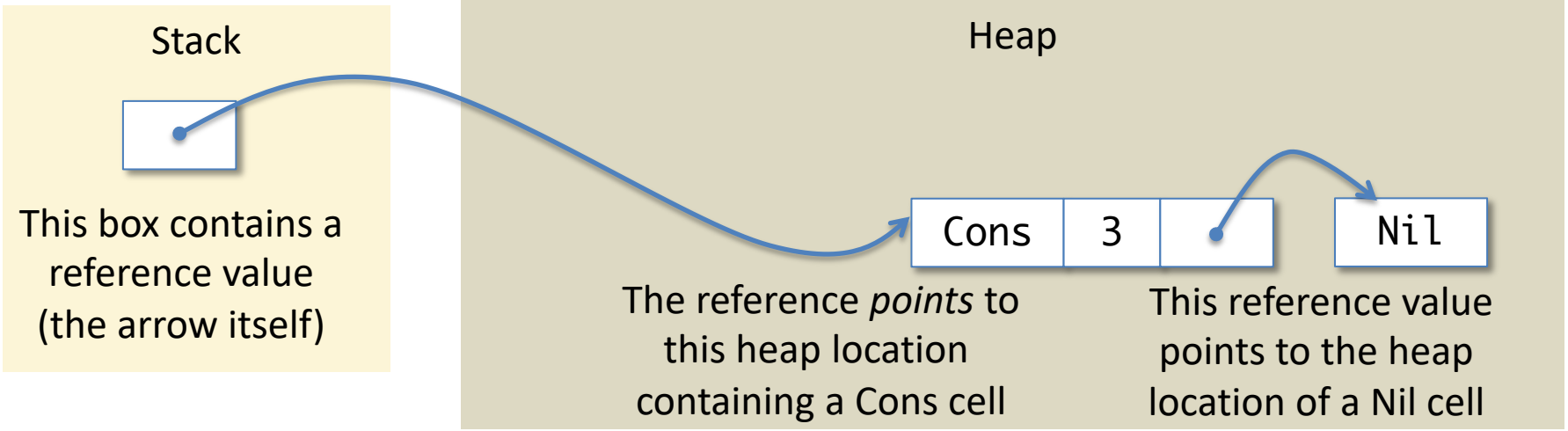
A *value* is either:

- a *primitive value* like an integer, or,
- a *reference* to a location in the heap

A reference value is the *address (location)* of data in the heap.

We draw a reference value as an "arrow"

- The arrow "points" to a box or cell located at this address
- Where we are storing this value also matters:



References and Equality

= vs. ==

Reference Equality

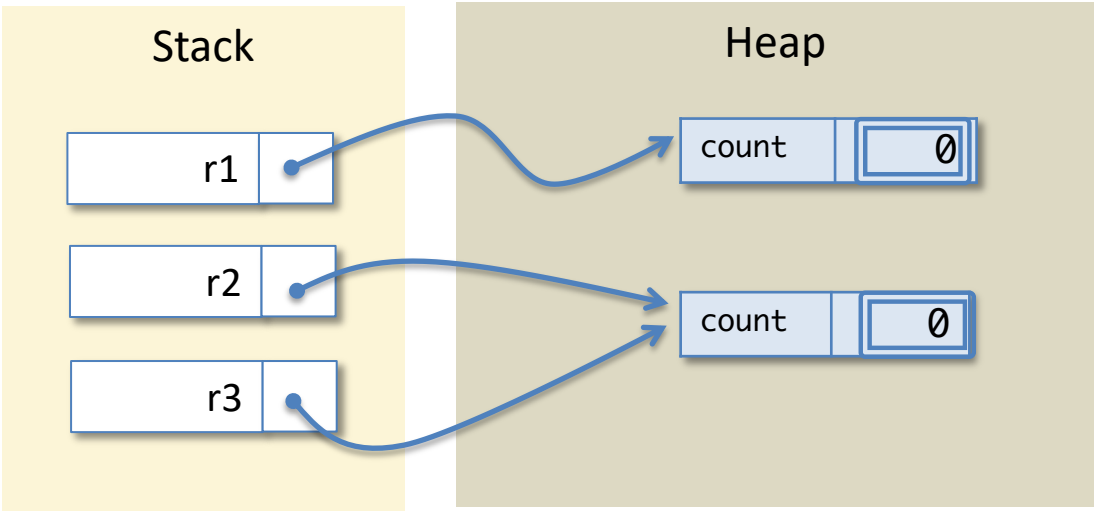
- Suppose we have two counters. Are they at the same location?

```
type counter = { mutable count : int }
let c1 : counter = …
let c2 : counter = …
```

  - We could increment one and see whether the other's value changes.
  - But we could also just test whether the references are **aliases**.

- OCaml uses '==' to mean *reference* equality:
  - two reference values are '==' if they point to the same location in the heap; so:

```
r2 == r3

not (r1 == r2)

r1 = r2
```



Structural vs. Reference Equality

- *Structural (in)equality*: `v1 = v2` `v1 <> v2`
  - recursively traverses over the *structure* of the data, comparing the two values' components for structural equality
  - function values cannot be compared structurally
  - structural equality can go into an infinite loop on cyclic structures
  - appropriate for comparing *immutable* datatypes

- *Reference (in)equality*: `v1 == v2` `v1 != v2`
  - Only looks at where the two references point in the heap
  - function values are only equal to themselves
  - even if v1 = v2, we may not have v1 == v2
  - appropriate for comparing *mutable* datatypes

14: What is the result of evaluating the following expression?

0

true

0%

false

0%

runtime error

0%

compile-time error

0%

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = p1 in

p1 = p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

14: What is the result of evaluating the following expression?

0

true

0%

false

0%

runtime error

0%

compile-time error

0%

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = p1 in

p1 == p2
```

1. `true`
2. `false`
3. runtime error
4. compile-time error

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = { x = 0; y = 0 } in

p1 == p2
```

1. true
2. false

Answer: false

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = { x = 0; y = 0 } in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 = l2
```

1. true
2. false

Answer: true

What is the result of evaluating the following expression?

```
let p1 : point = { x = 0; y = 0 } in
let p2 : point = p1 in
let l1 : point list = [p1] in
let l2 : point list = [p2] in

l1 == l2
```

1. true
2. false

Answer: false

ASM: Lists and datatypes

Tracking the space usage of *immutable* data structures

Simplification

Workspace

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

Stack

Heap

For uniformity, we'll pretend lists are declared like this:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

Simplification

Workspace

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

Stack

Heap

For uniformity, we'll pretend lists are declared like this:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

Simplification

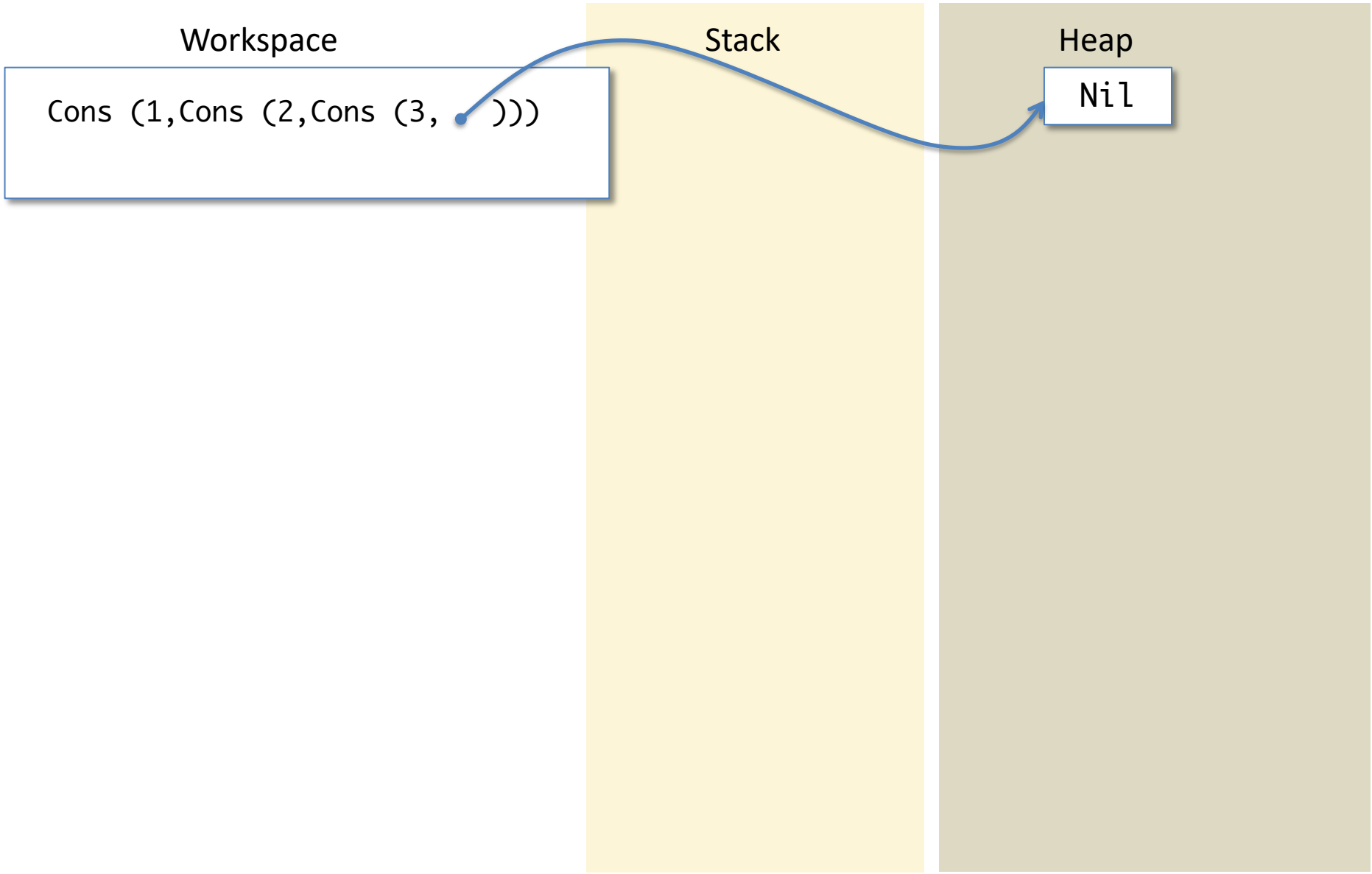
Workspace

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

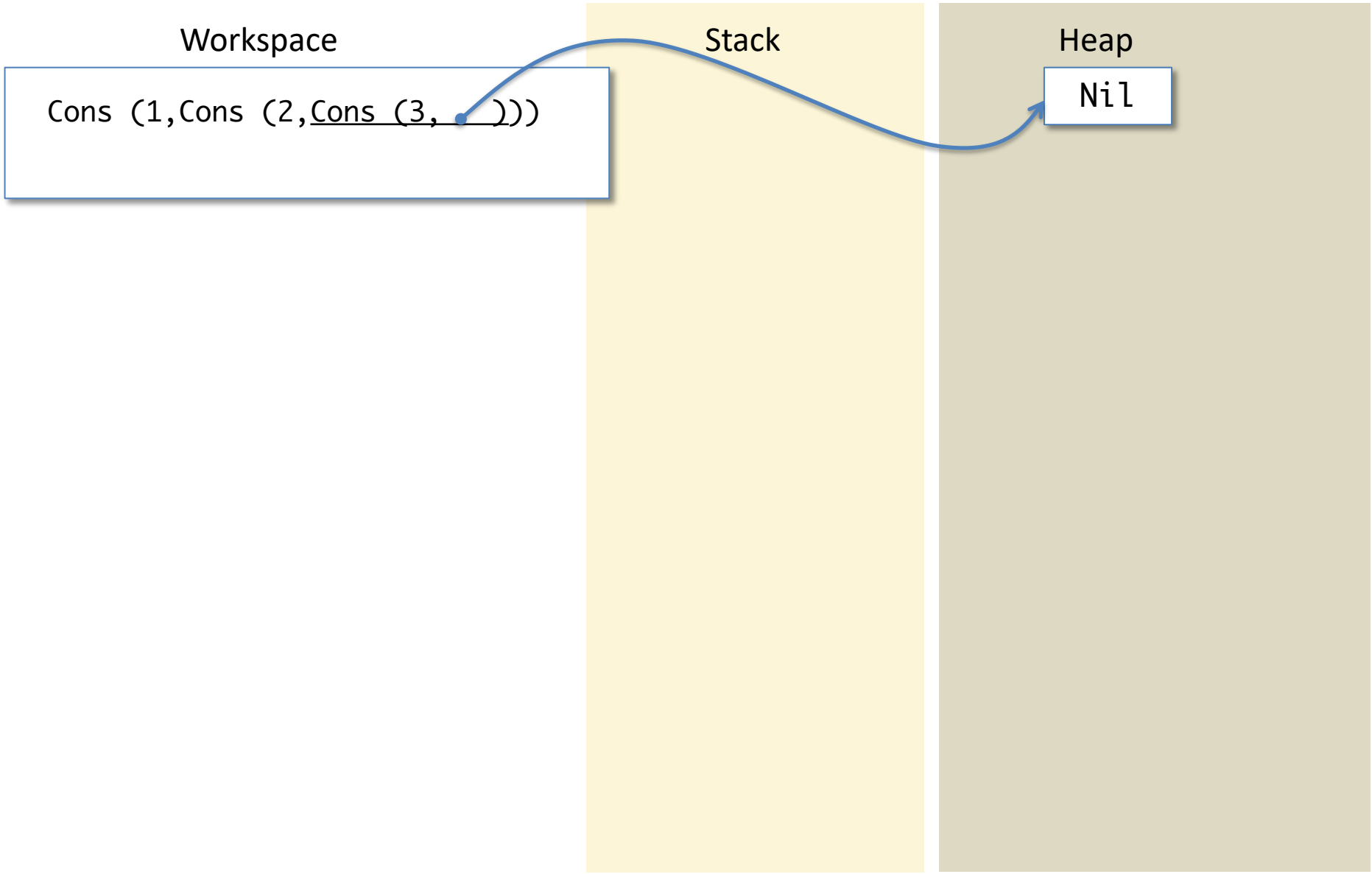
Stack

Heap

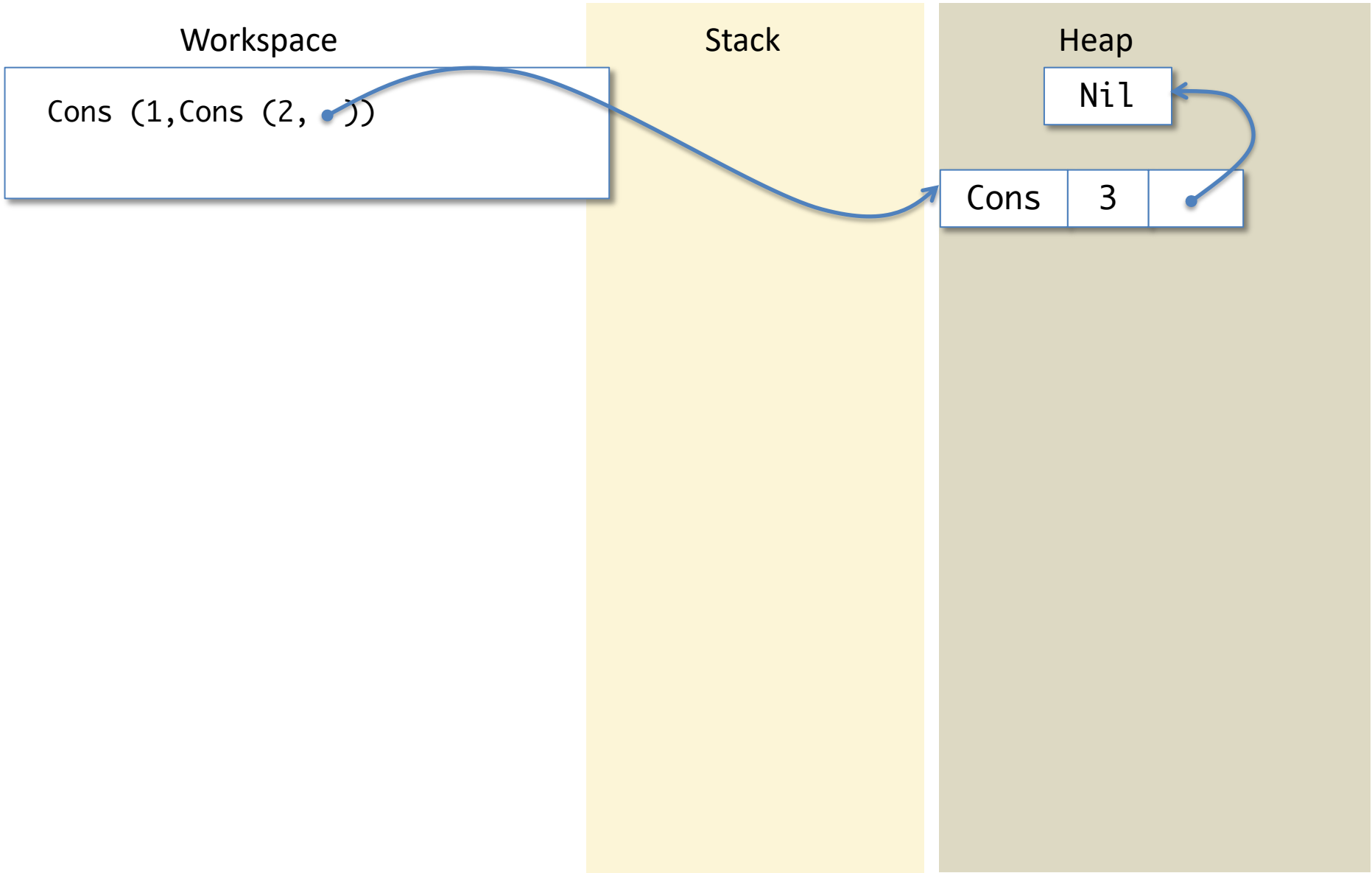
Simplification



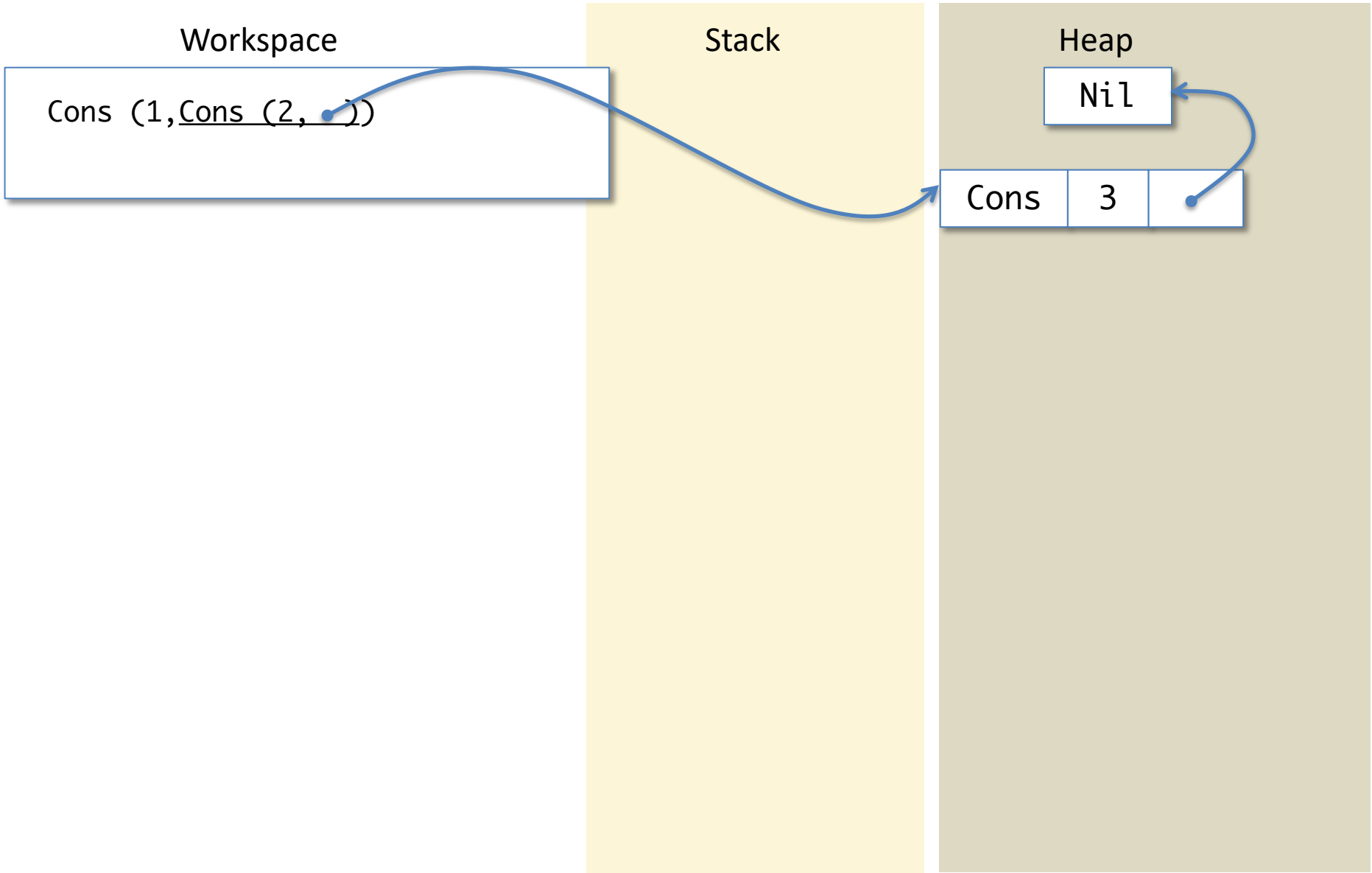
Simplification



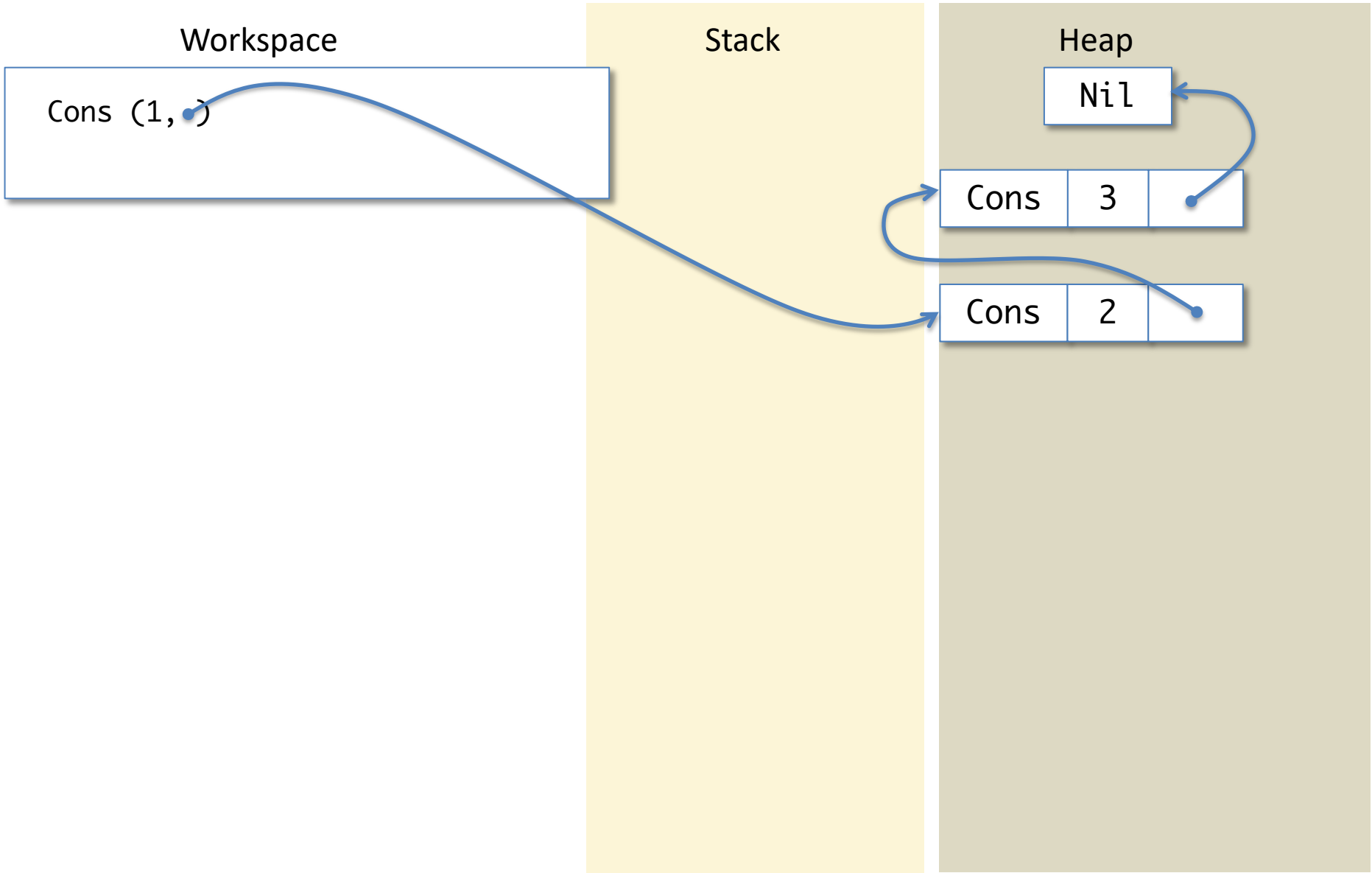
Simplification



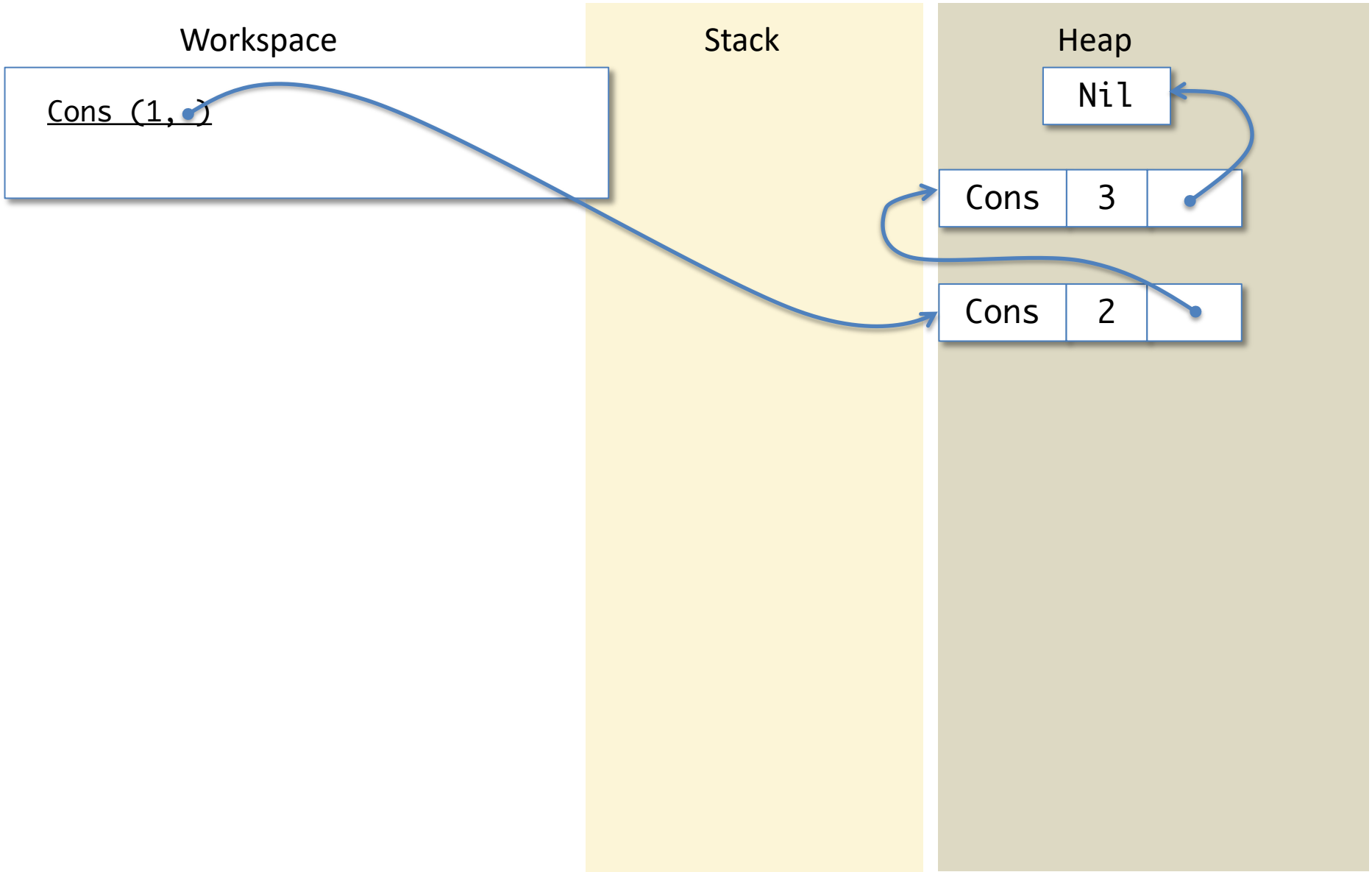
Simplification



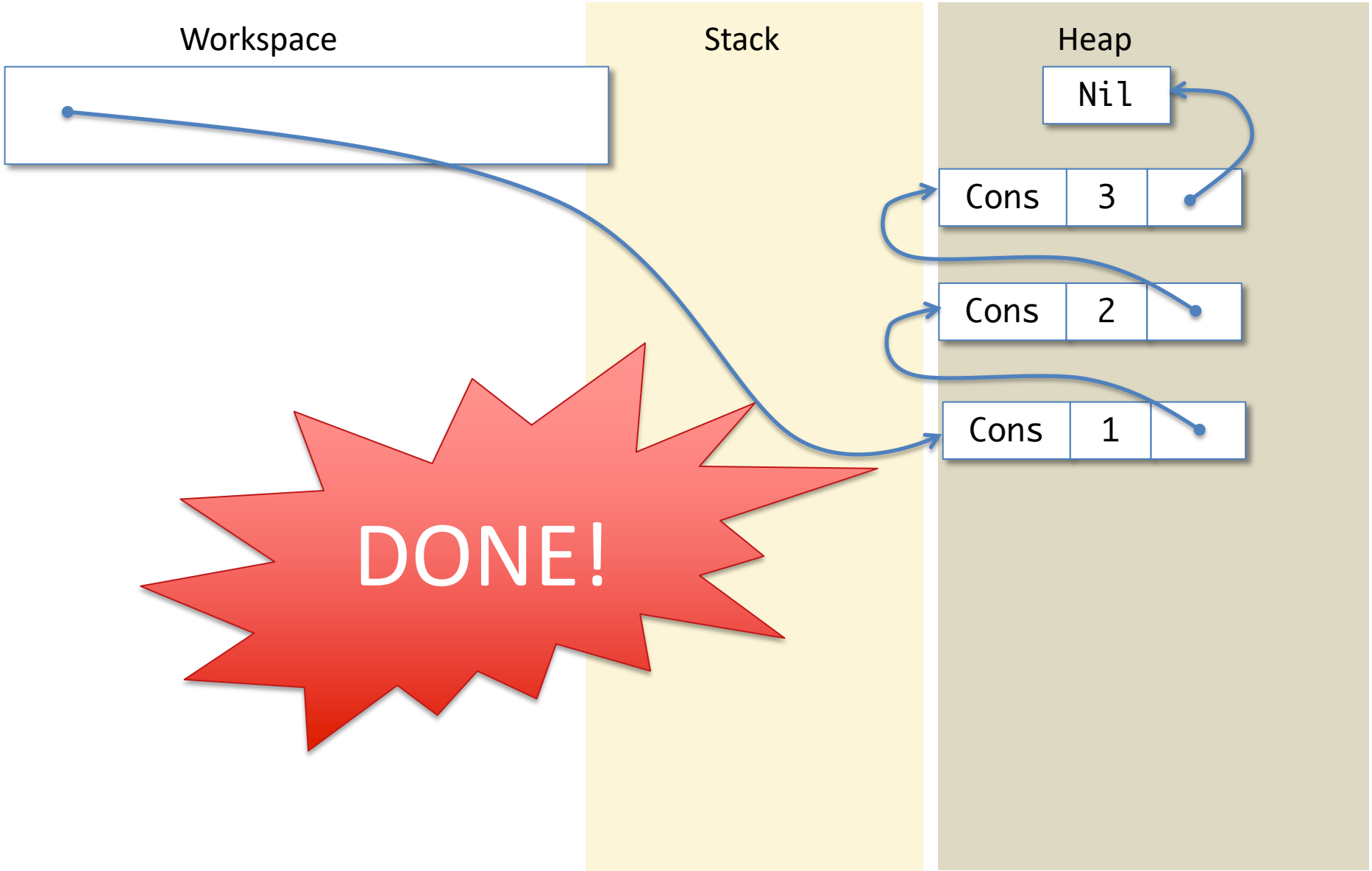
Simplification



Simplification



Simplification



An Optimization

- Datatype constructors that carry no extra information can be treated as “small” values.
- Examples:

```
type ‘a list =
| Nil
| Cons of ‘a * ‘a list
```

```
type ‘a option =
| None
| Some of ‘a
```

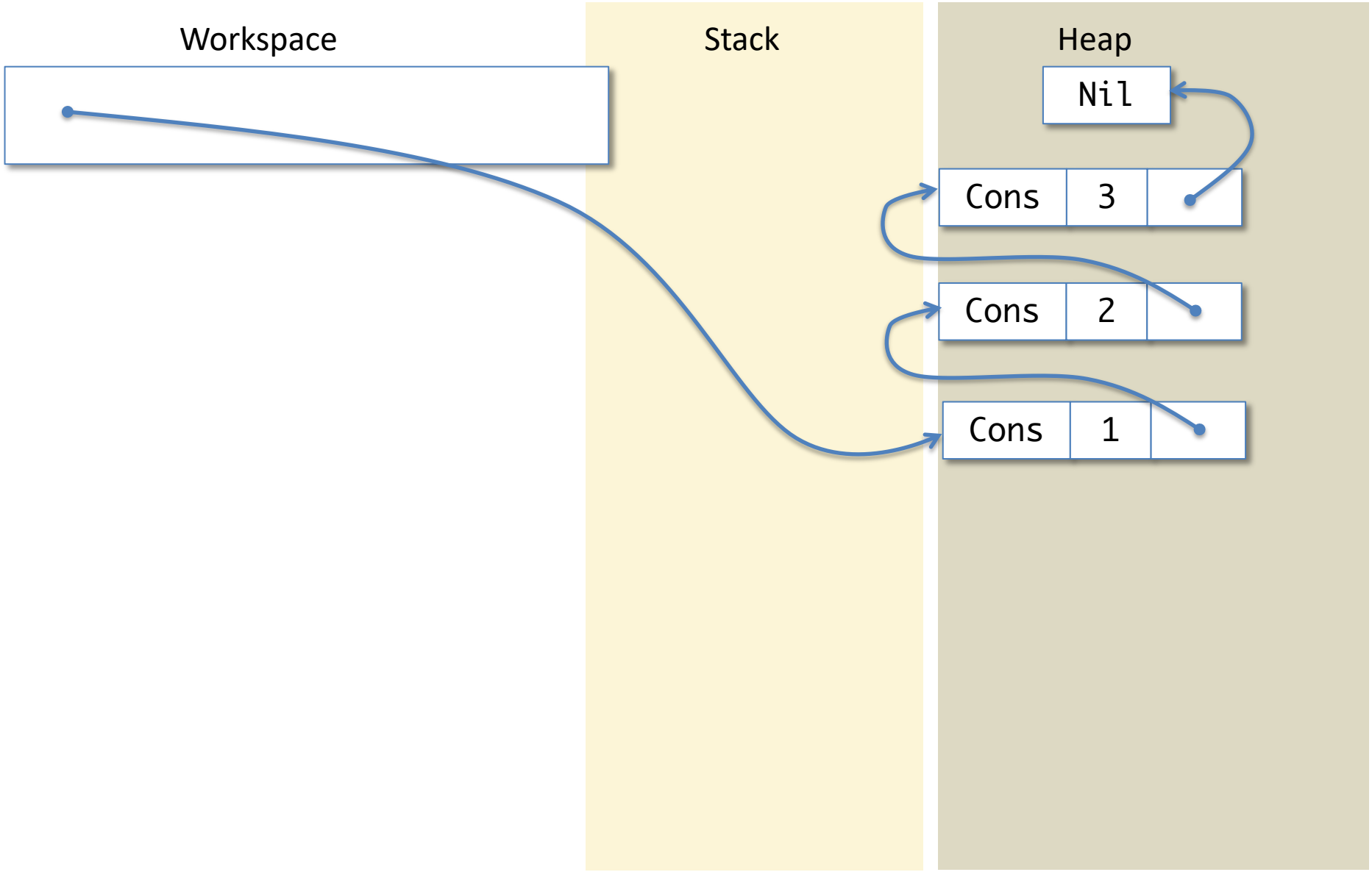
```
type ‘a tree =
| Empty
| Node of ‘a tree * ‘a * ‘a tree
```

- They can be placed directly in the stack.
- They don’t require a reference in the heap.

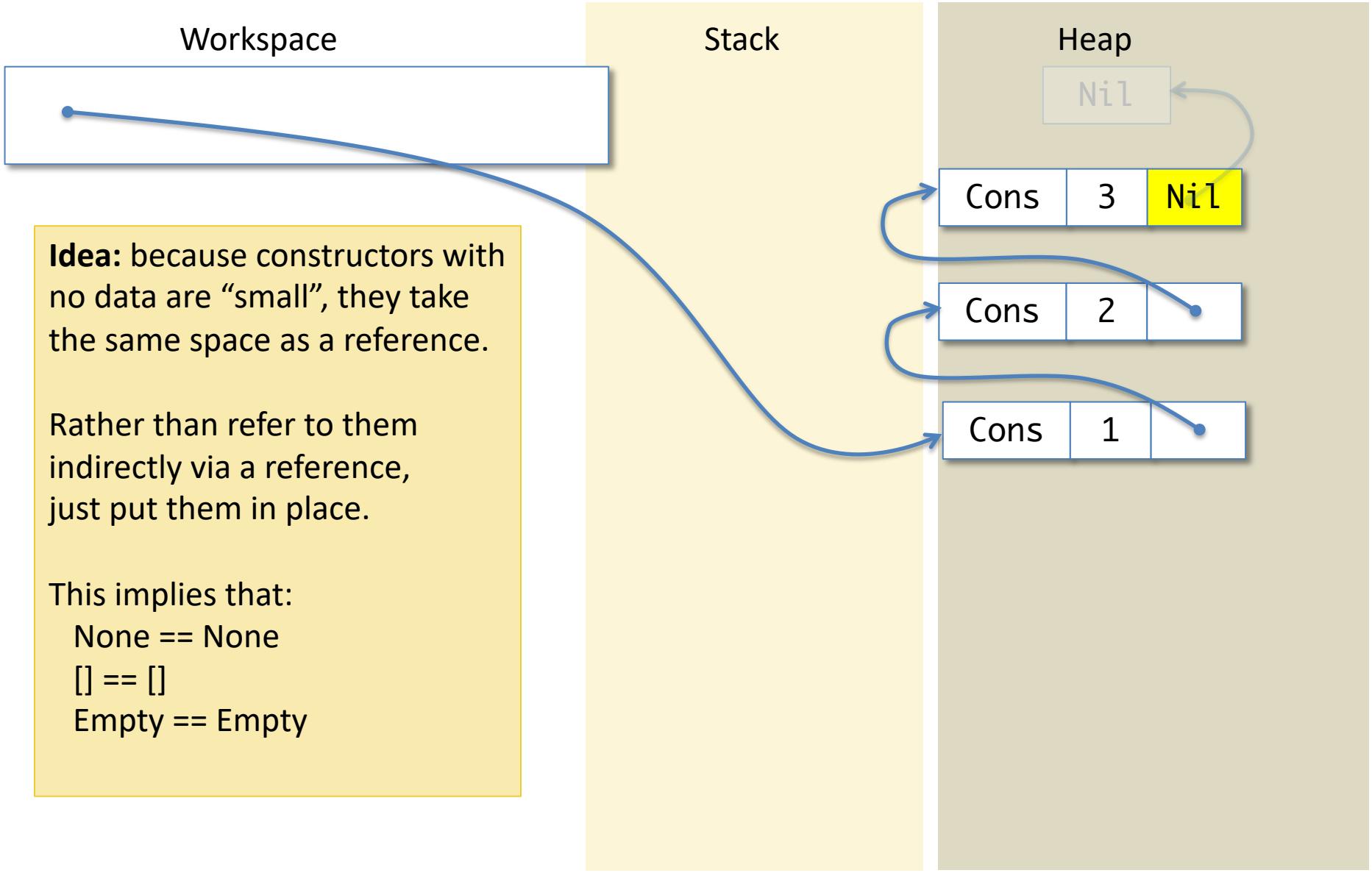
} Saves space!

- N.b.: This optimization affects reference equality.

Example Optimization



Example Optimization



Idea: because constructors with no data are "small", they take the same space as a reference.

Rather than refer to them indirectly via a reference, just put them in place.

This implies that:

None == None

[] == []

Empty == Empty

ASM: functions

Function Simplification

Workspace

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

Stack

Heap

Function Simplification

Workspace

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

Stack

Heap

Rewrite add1 as an anonymous function

Function Simplification

Workspace

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

Stack

Heap

Function Simplification

Workspace

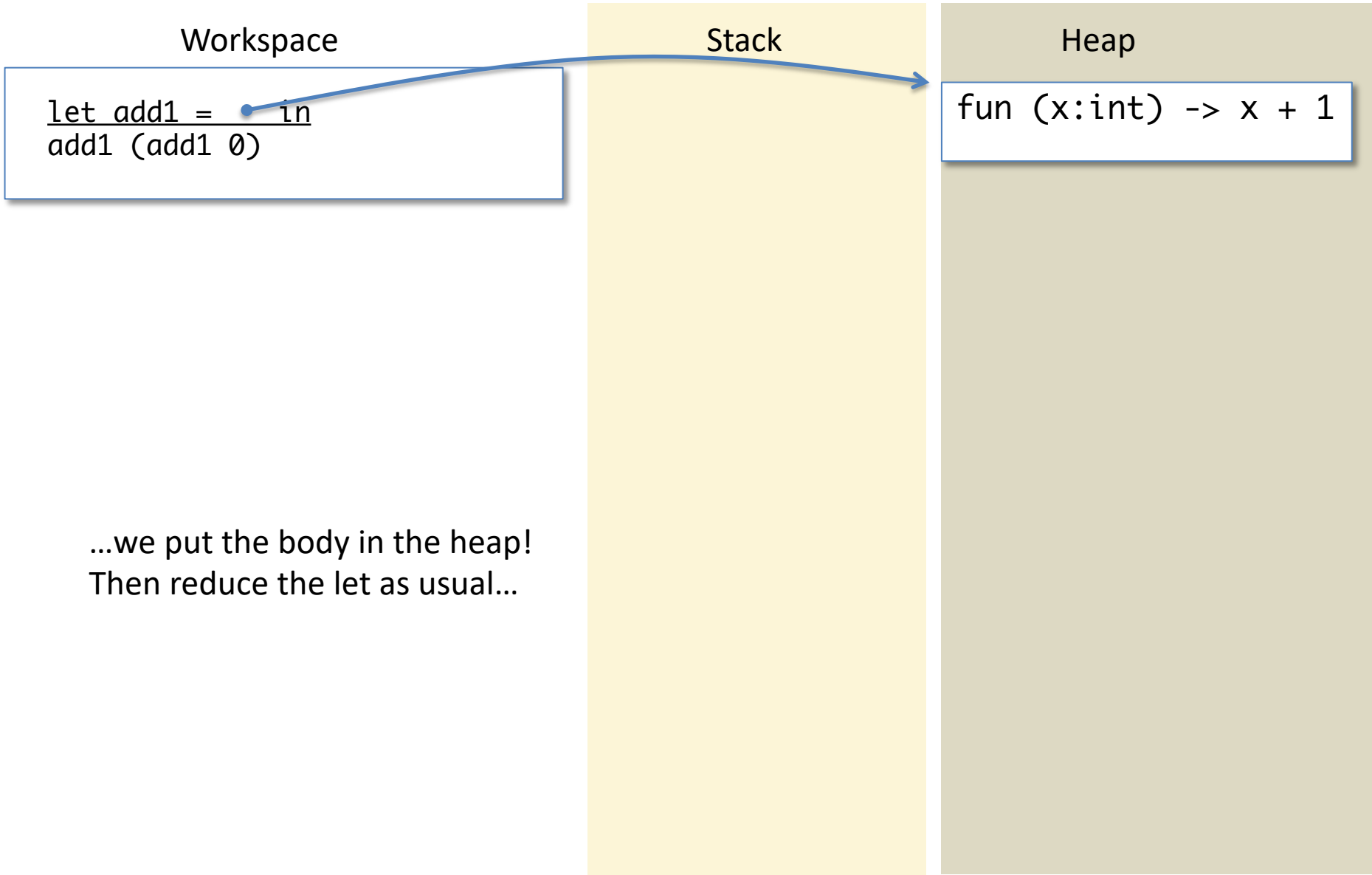
```
let add1 = fun (x : int) ->
  x + 1 in
add1 (add1 0)
```

Stack

Heap

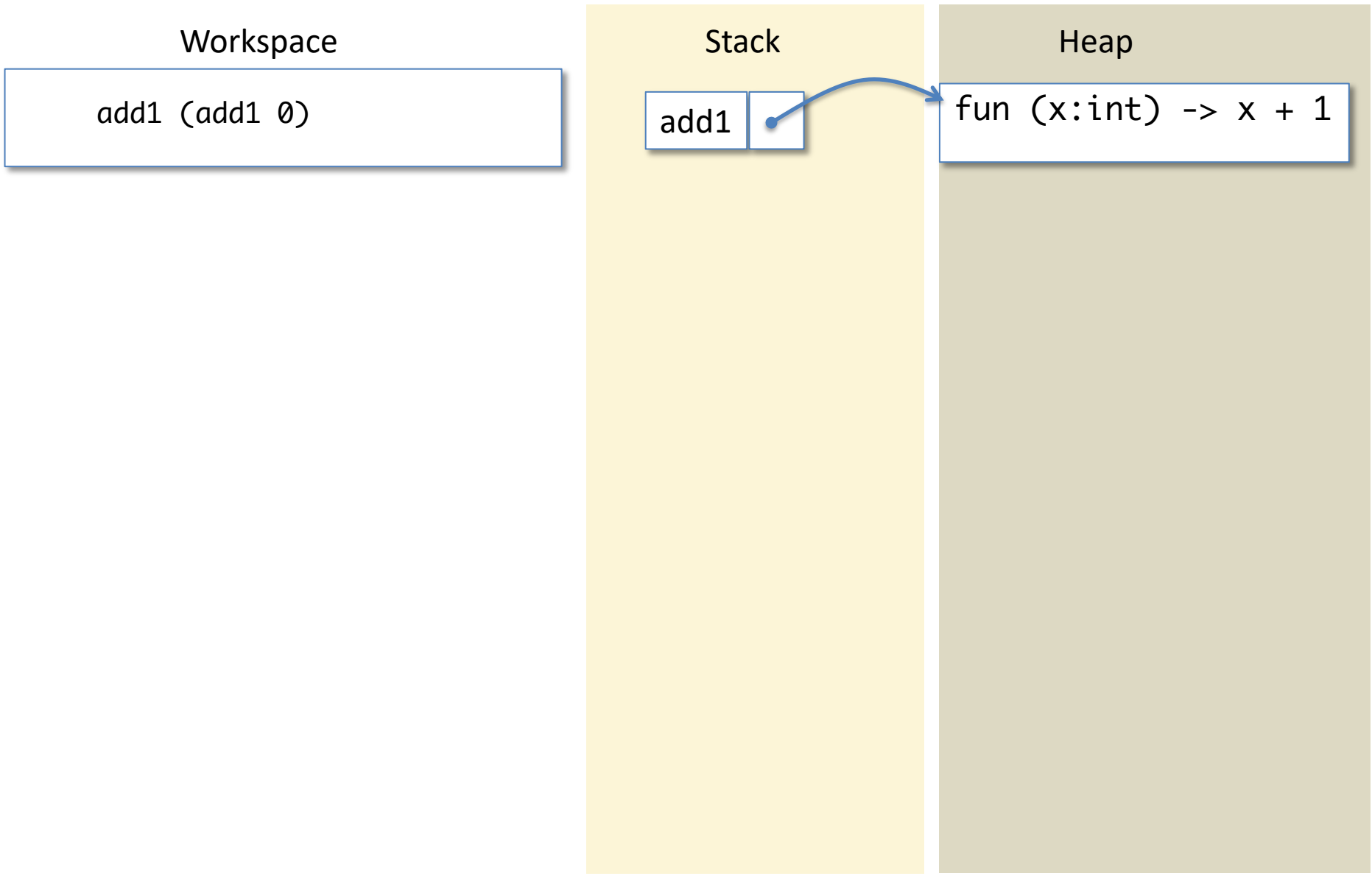
Function values are large, so…

Function Simplification

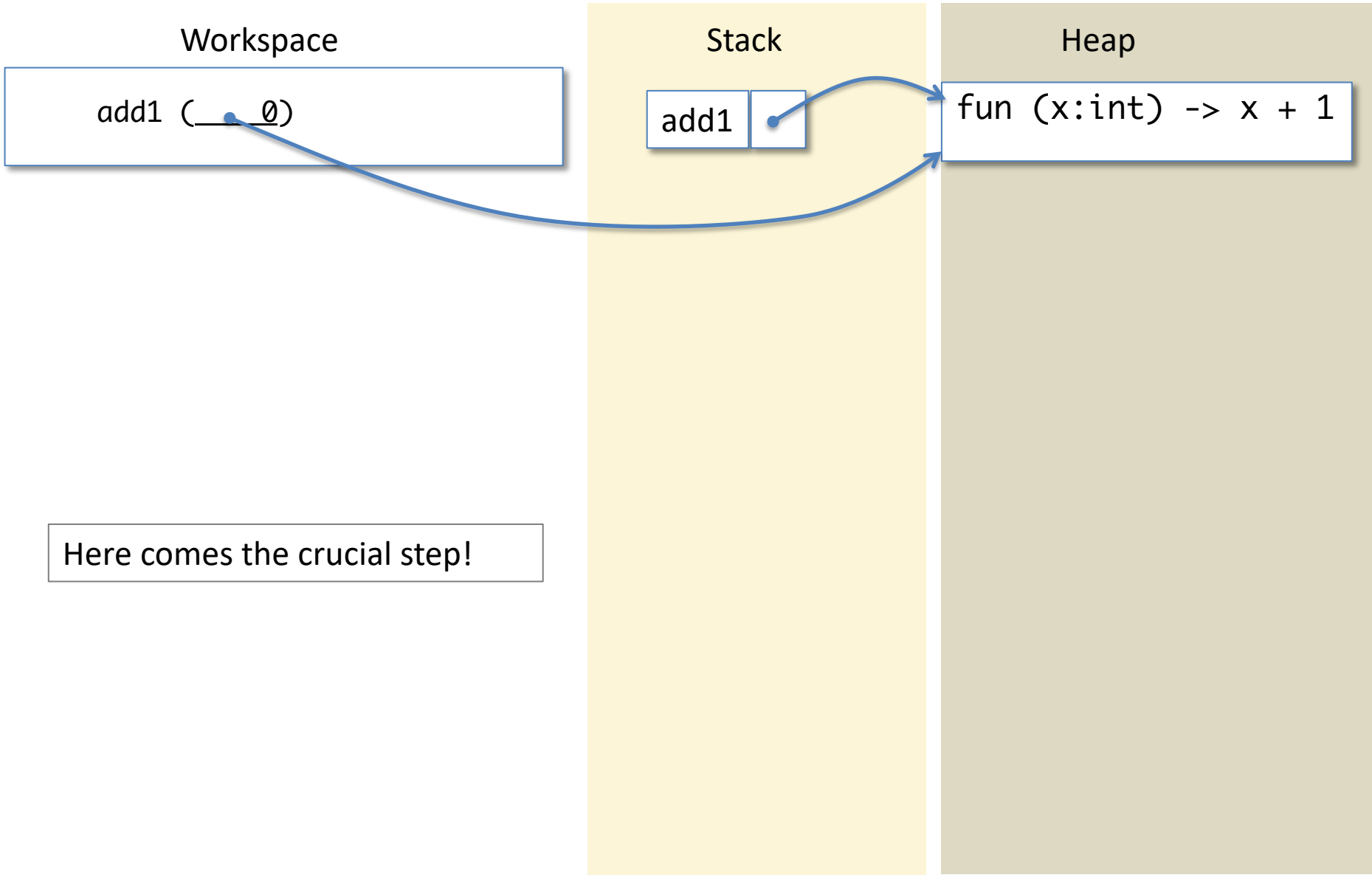


…we put the body in the heap!
Then reduce the let as usual…

Function Simplification

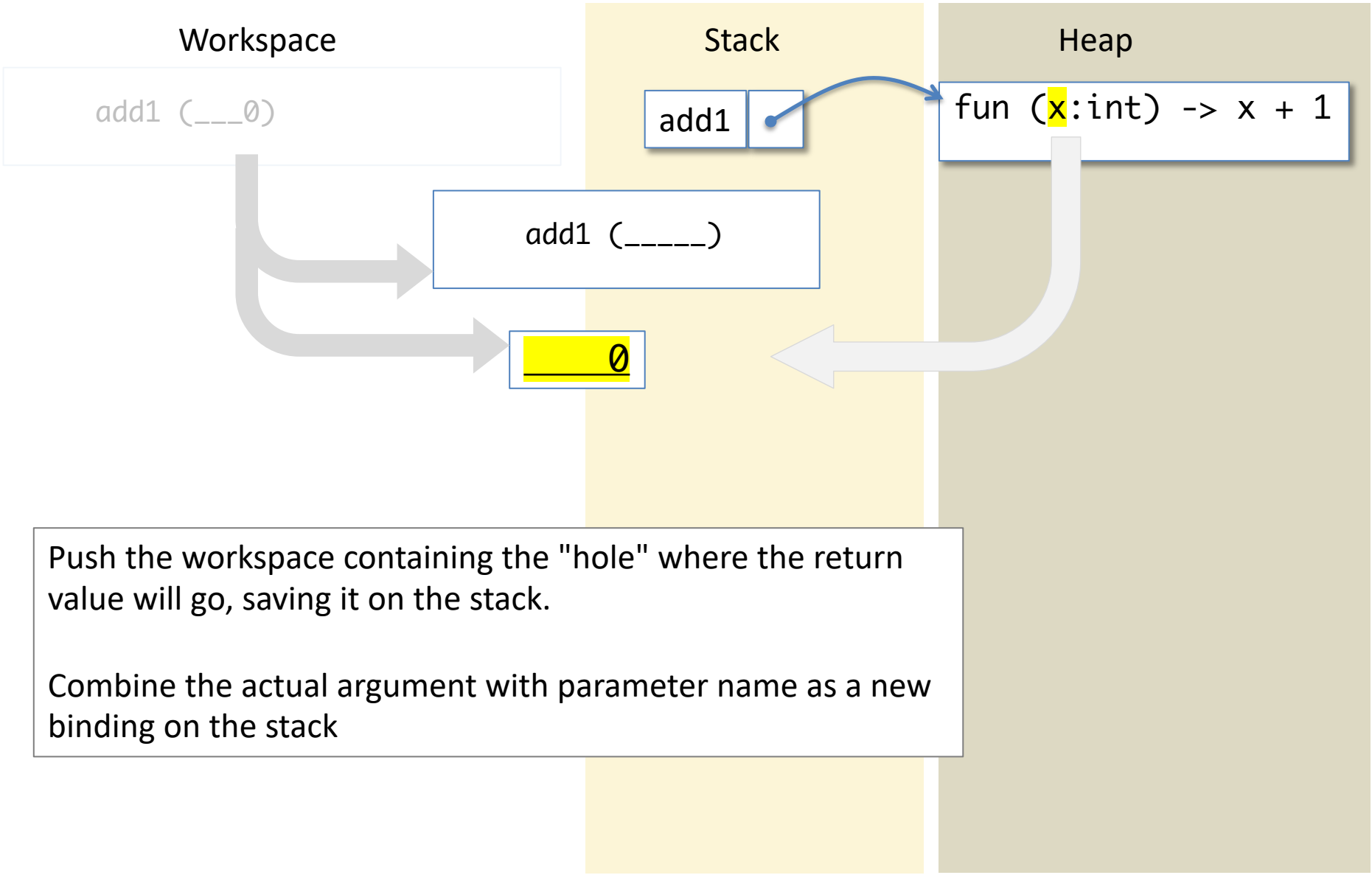


Function Simplification



Here comes the crucial step!

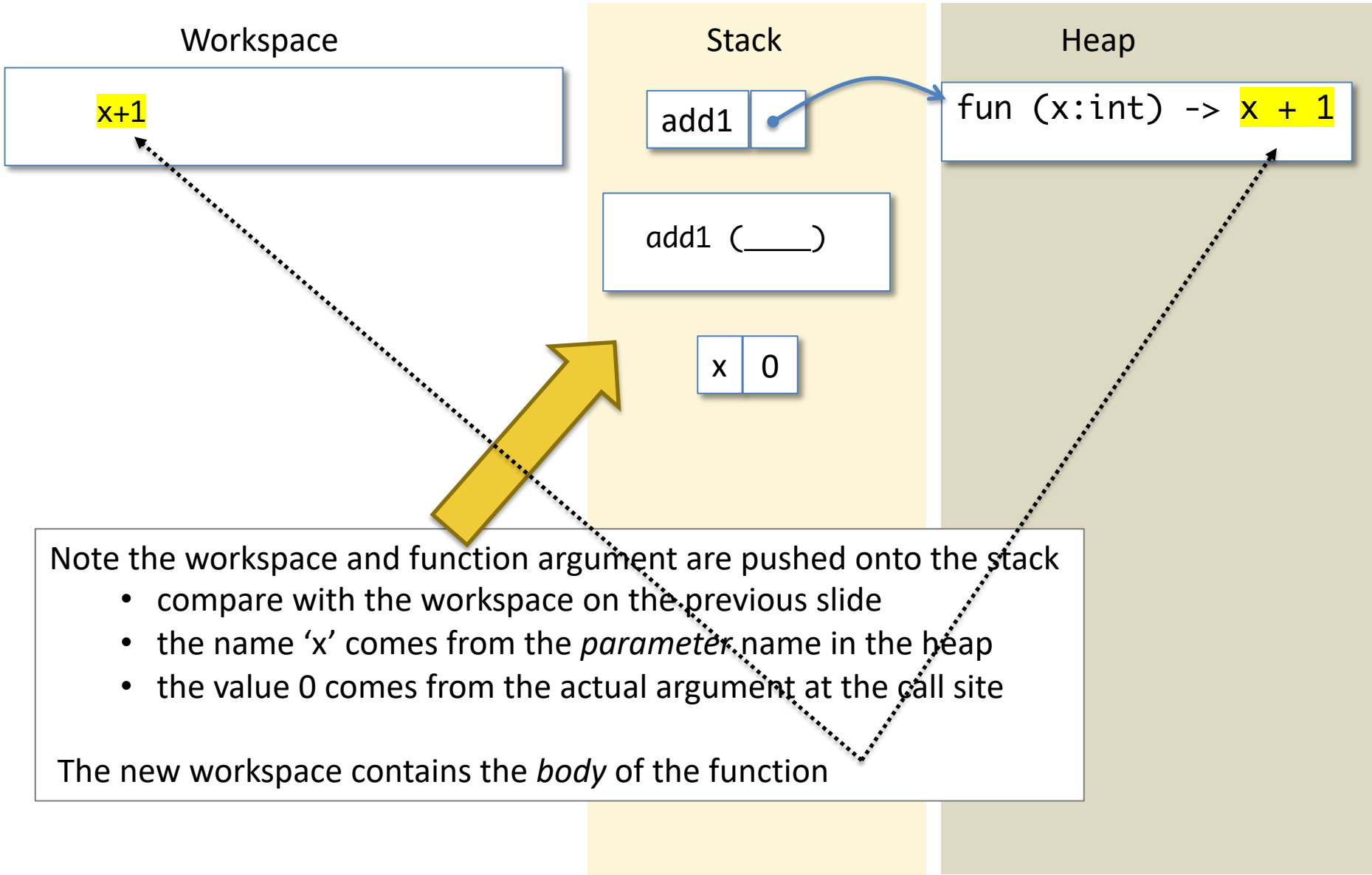
Push the Workspace & Argument



Push the workspace containing the "hole" where the return value will go, saving it on the stack.

Combine the actual argument with parameter name as a new binding on the stack

Do the Call, Saving the Workspace

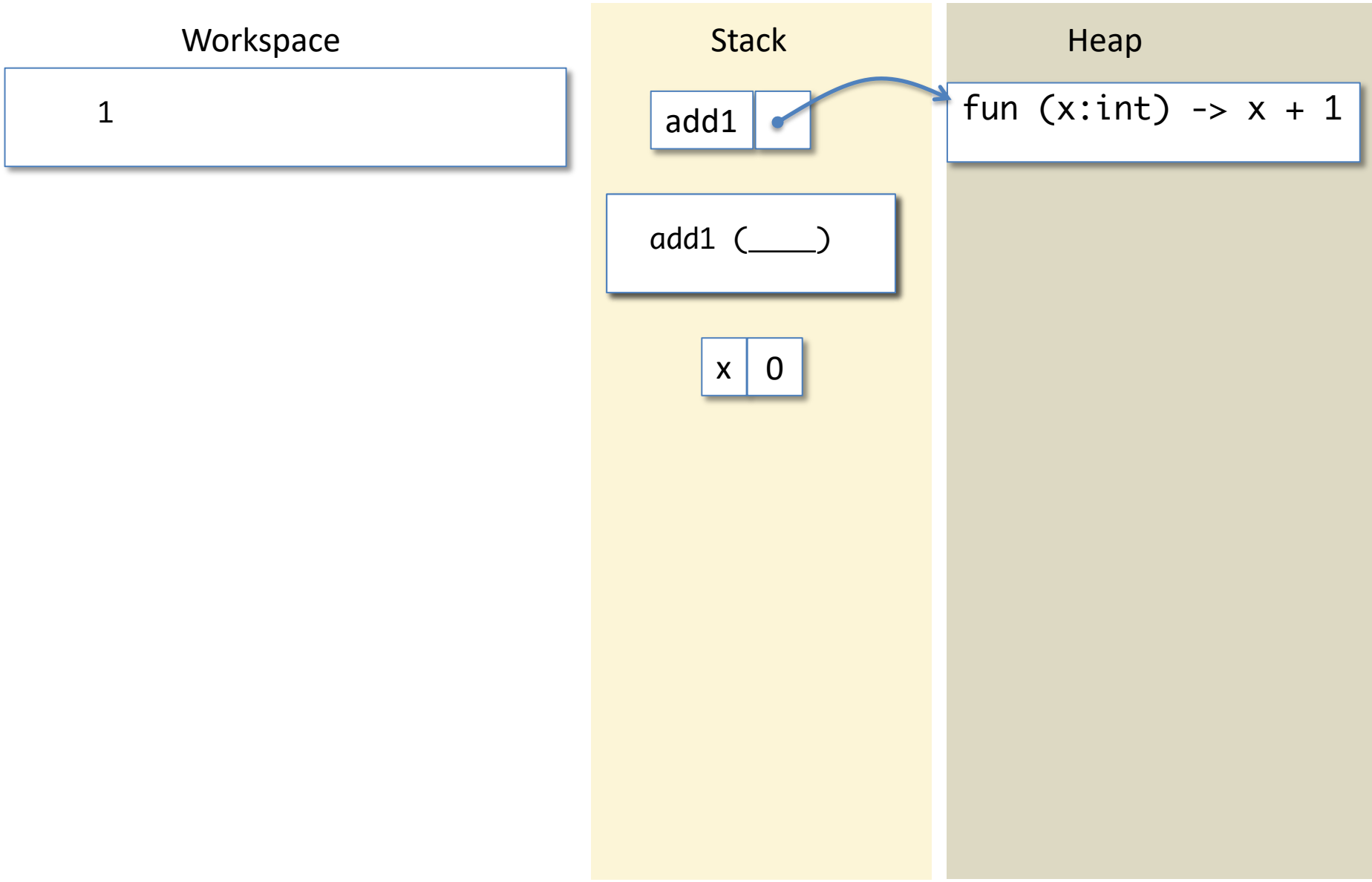


Note the workspace and function argument are pushed onto the stack

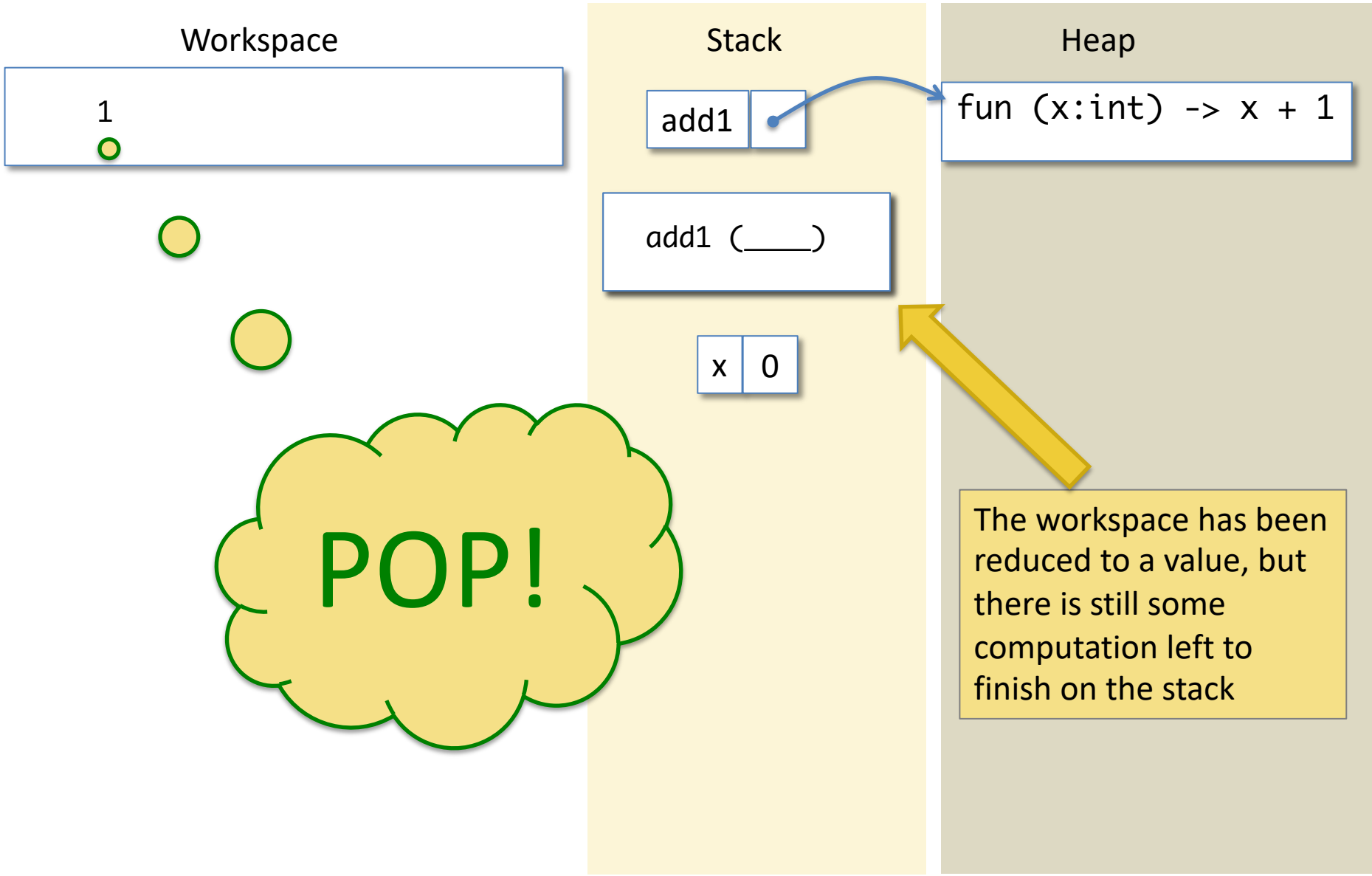
- compare with the workspace on the previous slide
- the name ‘x’ comes from the *parameter* name in the heap
- the value 0 comes from the actual argument at the call site

The new workspace contains the *body* of the function

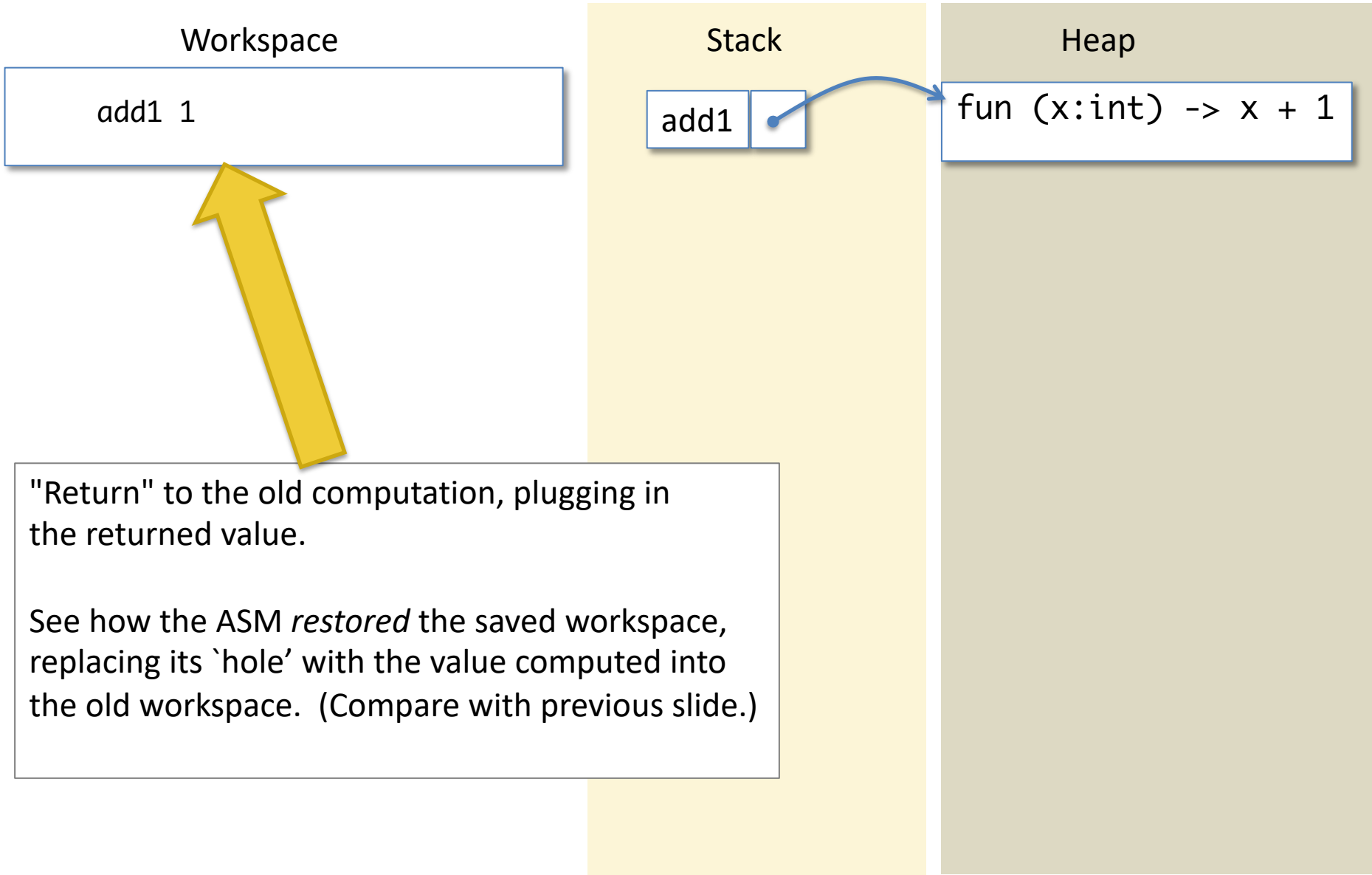
After a few more steps…



Function Simplification



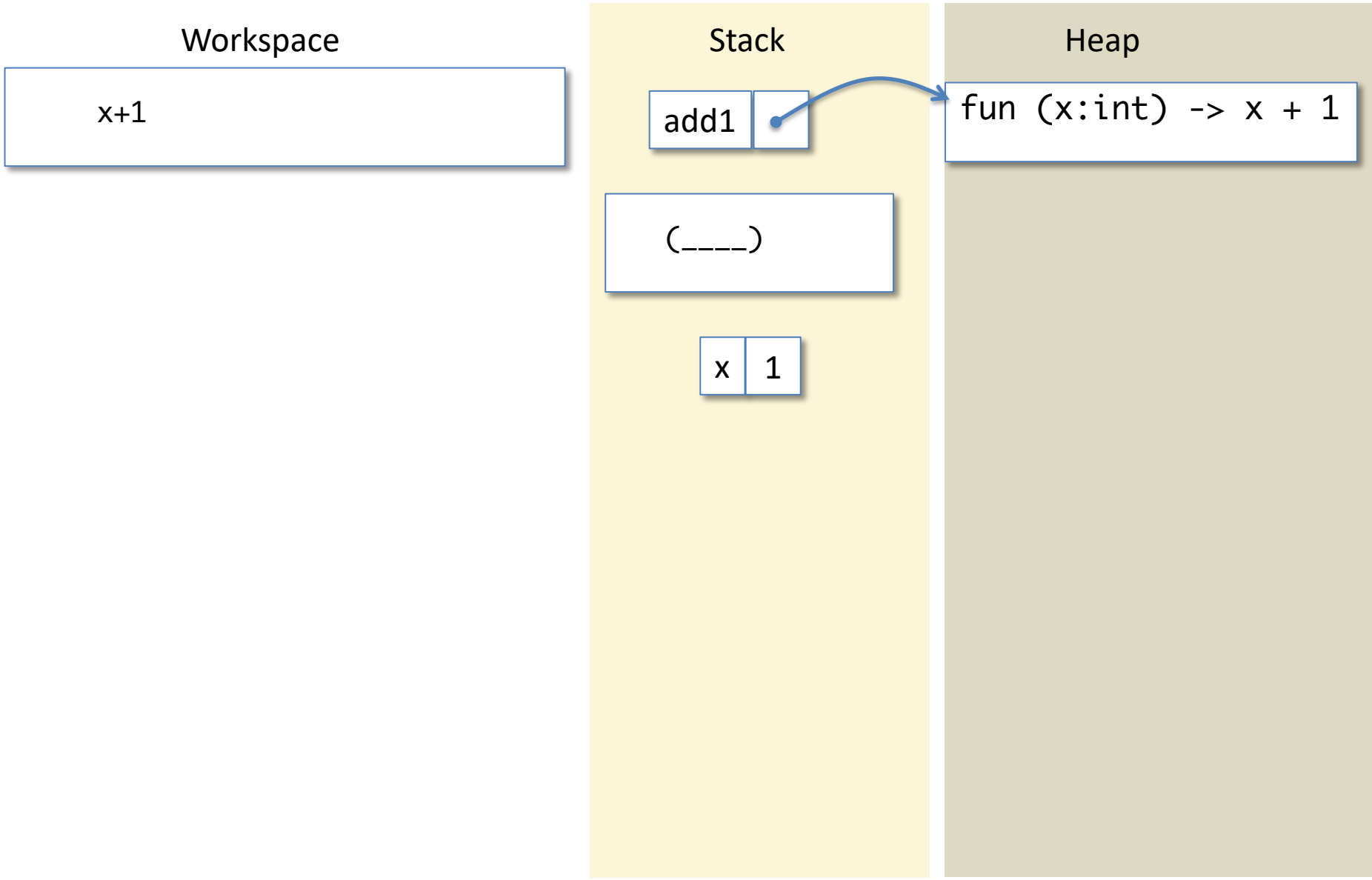
Function Simplification



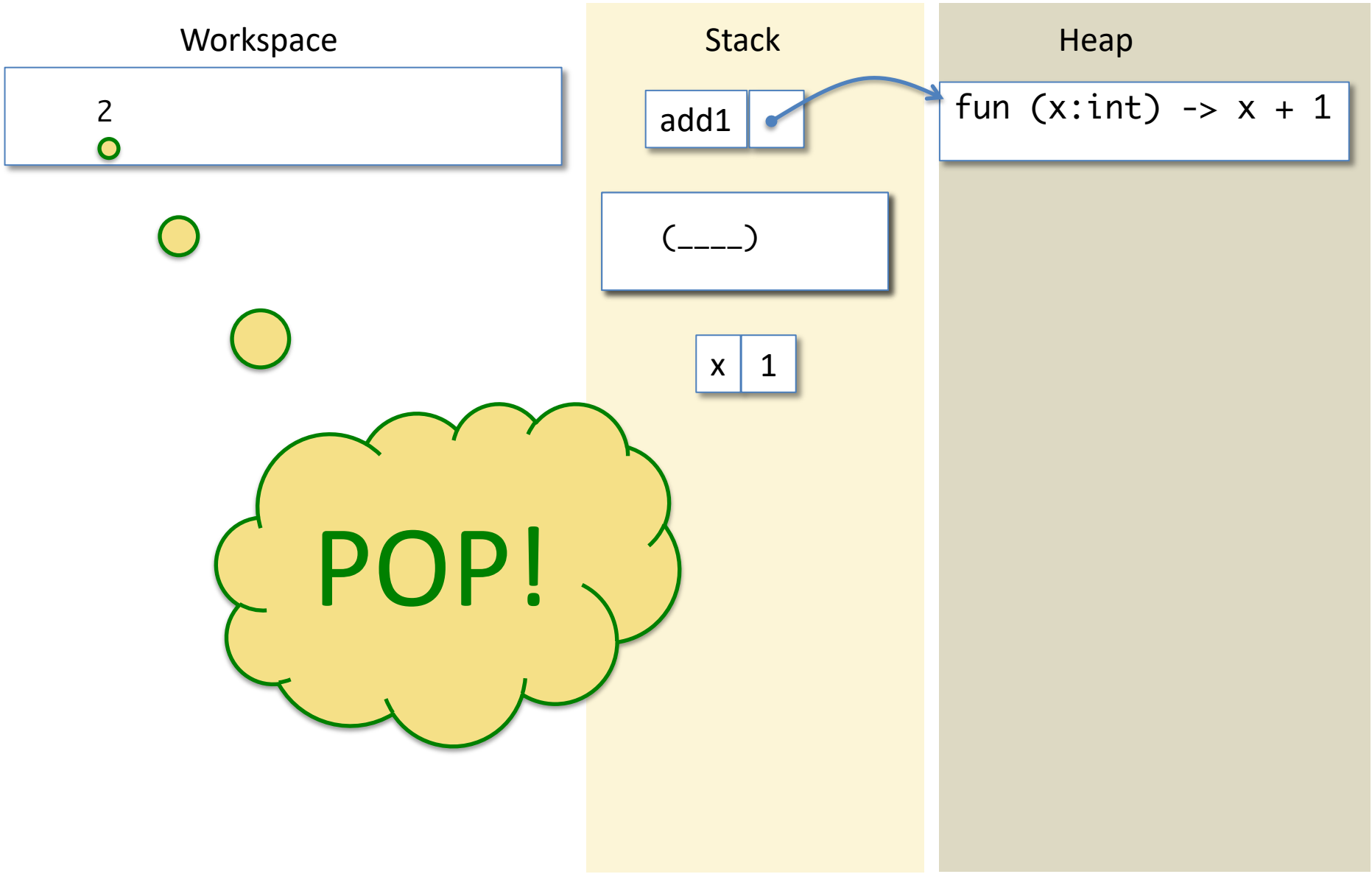
"Return" to the old computation, plugging in the returned value.

See how the ASM *restored* the saved workspace, replacing its `hole' with the value computed into the old workspace. (Compare with previous slide.)

Function Simplification



Function Simplification



Function Simplification

Simplifying Functions

- A function definition "let f $(x_1:t_1)\ldots(x_n:t_n)$ = e in body" is always ready.
  - It is simplified by replacing it with "let f = fun $(x:t_1)\ldots(x:t_n)$ = e in body"
- A function "fun $(x_1:t_1)\ldots(x_n:t_n)$ = e" is always ready.
  - It is simplified by moving the function to the heap and replacing the function expression with a pointer to that heap data.
- A function *call* is ready if the function and its arguments are all values
  - it is simplified by
    - saving the current workspace contents on the stack
    - adding bindings for the function's parameter variables (to the actual argument values) to the end of the stack
    - copying the function's body to the workspace

Function Completion

- When the workspace contains just a single value, we *pop the stack* by removing everything back to (and including) the last saved workspace contents.

- The value currently in the workspace is substituted for the function application expression in the saved workspace contents, which are put back into the workspace.

- If there aren't any saved workspaces in the stack, then the whole computation is finished and the value in the workspace is its final result.

Putting State to Work: Mutable Queues

A design problem

Suppose you are implementing a website for constituents to submit questions to their political representatives. To be fair, you would like to deal with questions in first-come, first-served order. How would you do it?

- Understand the problem
  - Need to keep track of pending questions, in the order in which they were submitted
- Define the interface
  - Need a data structure to store questions
  - Need to add questions to the *end* of the queue
  - Need to allow responders to retrieve questions from the *beginning* of the queue
  - Both kinds of access must be efficient to handle large volume

Design Process Step 1:
Understand the problem

(Mutable) Queue Interface

```
module type QUEUE =
sig
  (* abstract type *)
  type 'a queue

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

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

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

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

end
```

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

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

A: Adding an element to a queue returns `unit` because it *modifies* the given queue.

Design Process Step 2: specify the interface

Specify the behavior via test cases

```
let test () : bool =
 let q = create () in
  enq 1 q;
  begin match deq q with
  | None -> failwith "deq failed"
  | Some hd -> hd = 1 && is_empty q
  end
;; 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
  begin match deq q with
  | None -> false
  | Some hd -> hd = 2 && is_empty q
  end
;; run_test "queue test 2" test
```

Design Process Step 3: write test cases

Implementing Linked Queues

Representing links

Data Structure for Mutable Queues

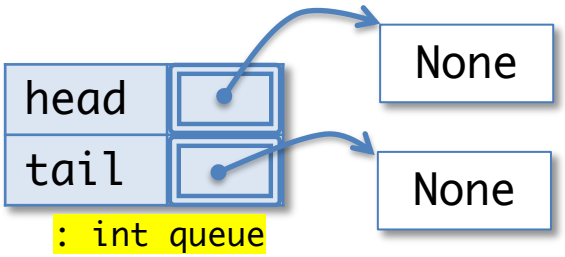
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

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
2. a record with links to the head and tail nodes

All of these links are *optional* so that the queue can be empty

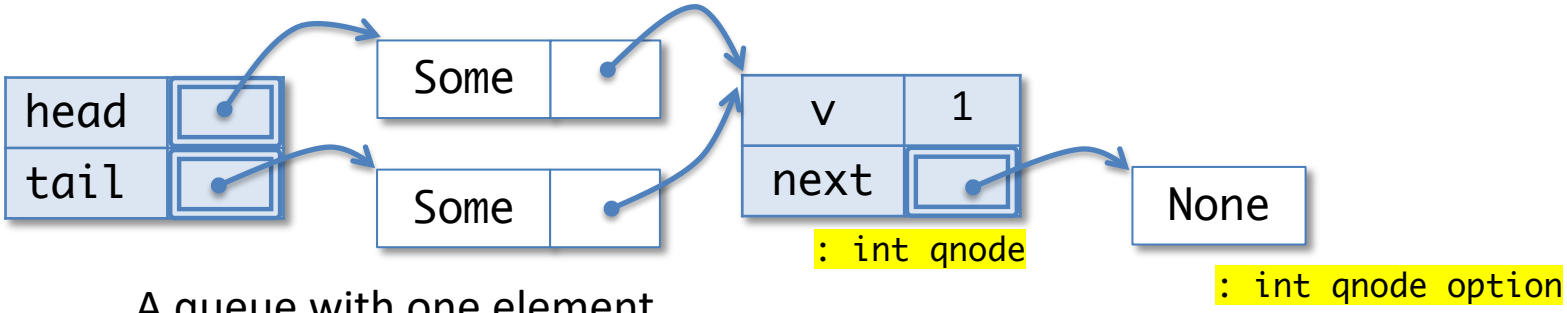
Queues in the Heap



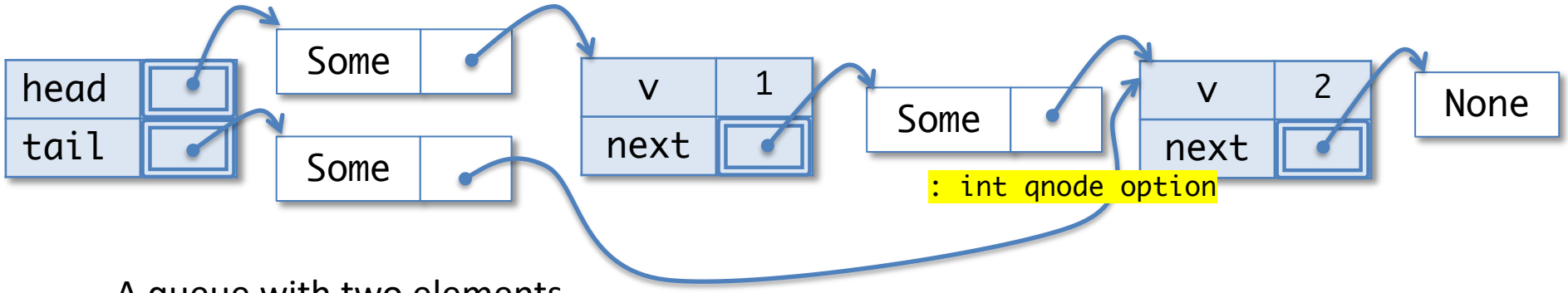
An empty queue

Type Information

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

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

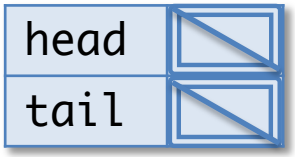


A queue with one element

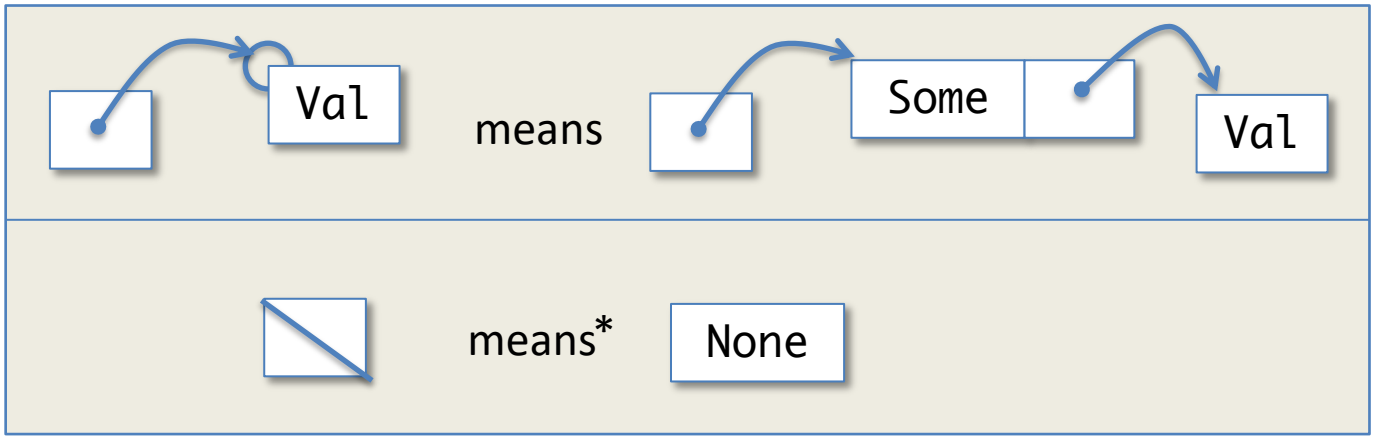


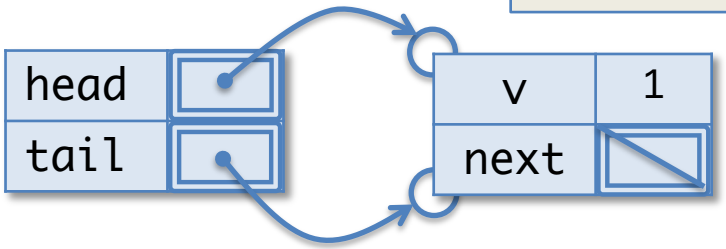
A queue with two elements

Visual Shorthand: Abbreviating Options



An empty queue





A queue with one element

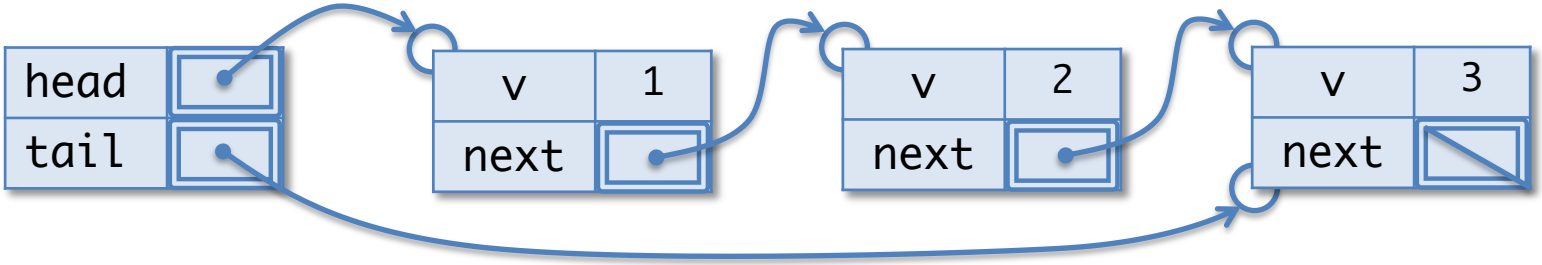
*Note: Ocaml can optimize "nullary" constructors like Nil, None, Empty so that they aren't allocated in the heap. This is why

```
None == None
```

even though

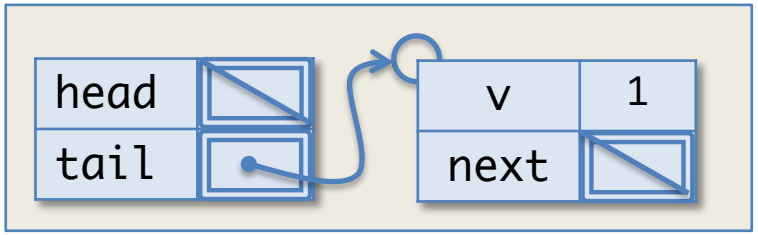
```
not ((Some x) == (Some x)).
```

Be careful with reference equality and options!

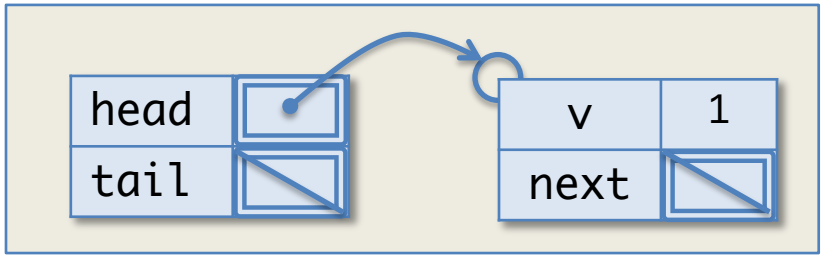


A queue with three elements

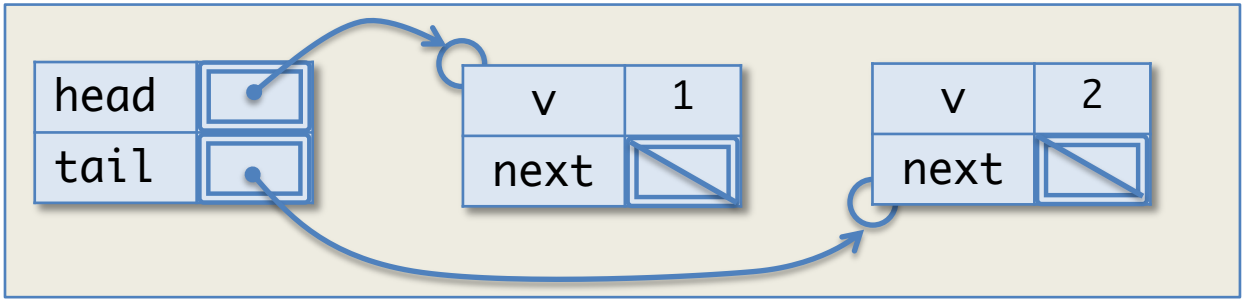
“Bogus” values of type `int queue`



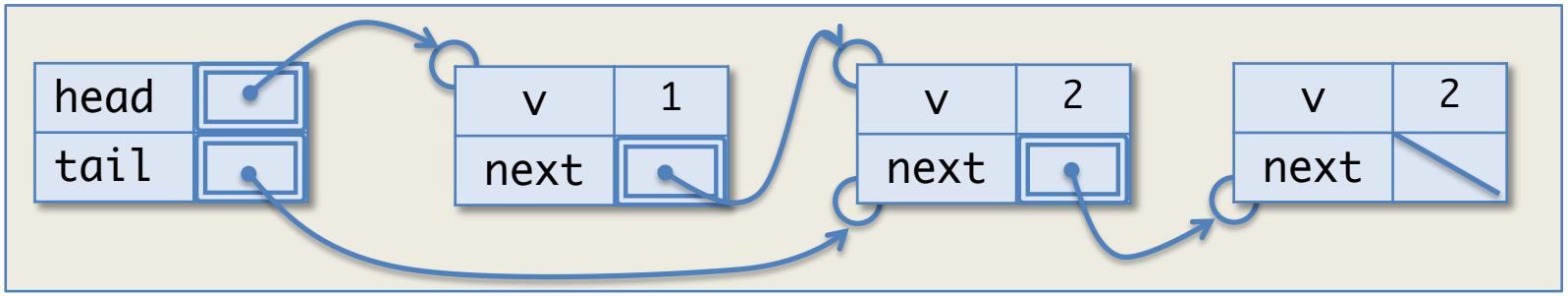
head is None, tail is Some



head is Some, tail is None



tail is not reachable from head



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

15: Given the queue datatype shown below, is it possible to create a cycle of references in the heap. (i.e. a way to get back to the same place by following references.)

0

yes

0%

no

0%

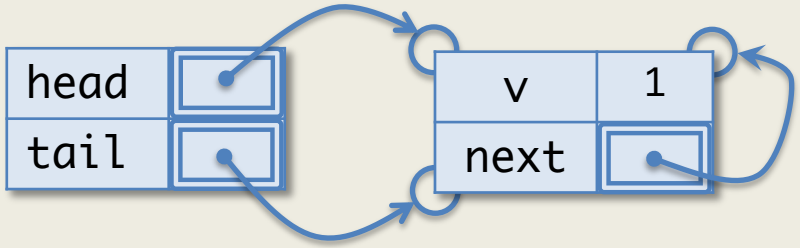
not sure

0%

Given the queue datatype shown below, is it possible to create a *cycle* of references in the heap. (i.e. a way to get back to the same place by following references.)

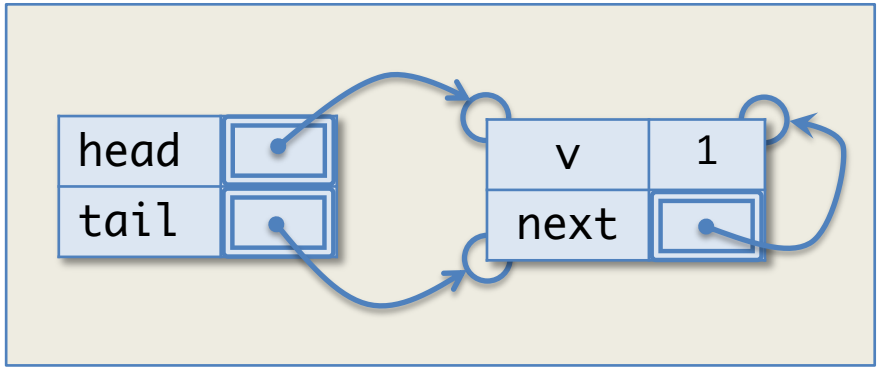
```
type 'a qnode = {
     v: 'a;
     mutable next : 'a qnode option
}

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

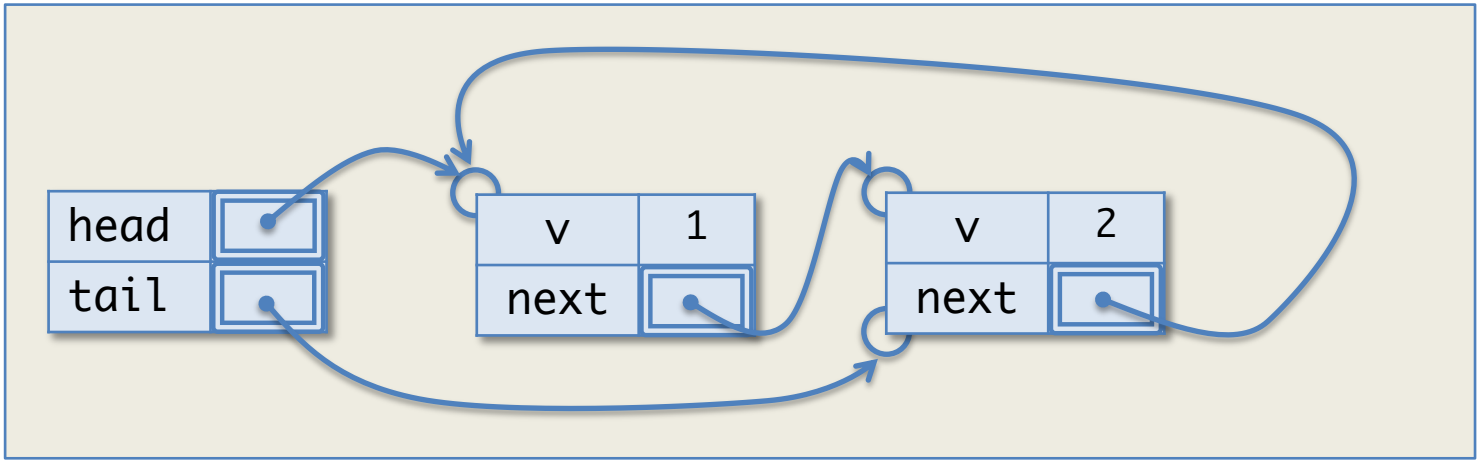
1. yes
2. no
3. not sure



Answer: 1

Cyclic `int` queue values





(And many, many others…)

Linked Queue Invariants

Just as we imposed some restrictions on which trees count as legitimate Binary Search Trees, we require that Linked Queues satisfy the following representation *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 prove that these properties suffice to rule out all of the "bogus" examples.
- Each queue operation may assume that these invariants hold on its inputs and must ensure that the invariants hold when it's done.

15: Is this a valid queue?

0

yes

0%

no

0%

Either:

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

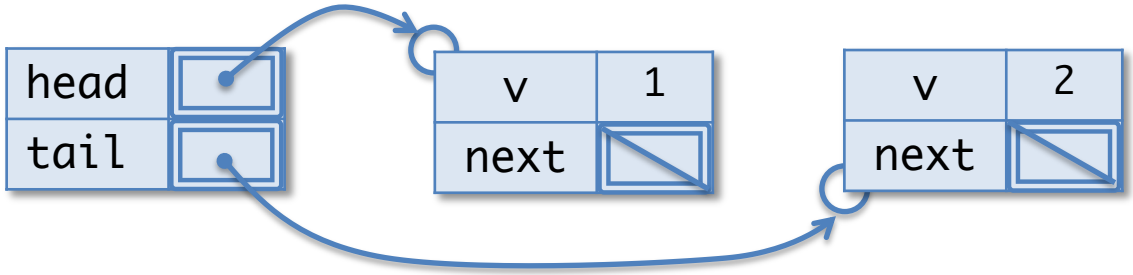
or

(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



ANSWER: No

15: Is this a valid queue?

0

yes

0%

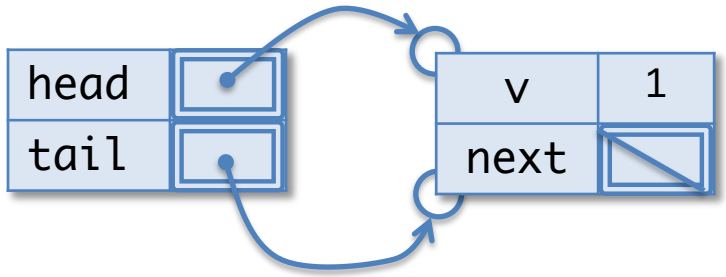
no

0%

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



ANSWER: Yes

15: Is this a valid queue?

0

yes

0%

no

0%

Either:

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

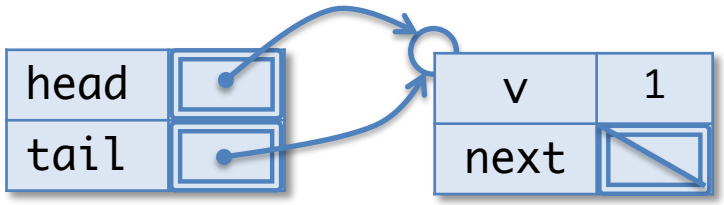
or

(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



ANSWER: Yes

Implementing Linked Queues

q.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

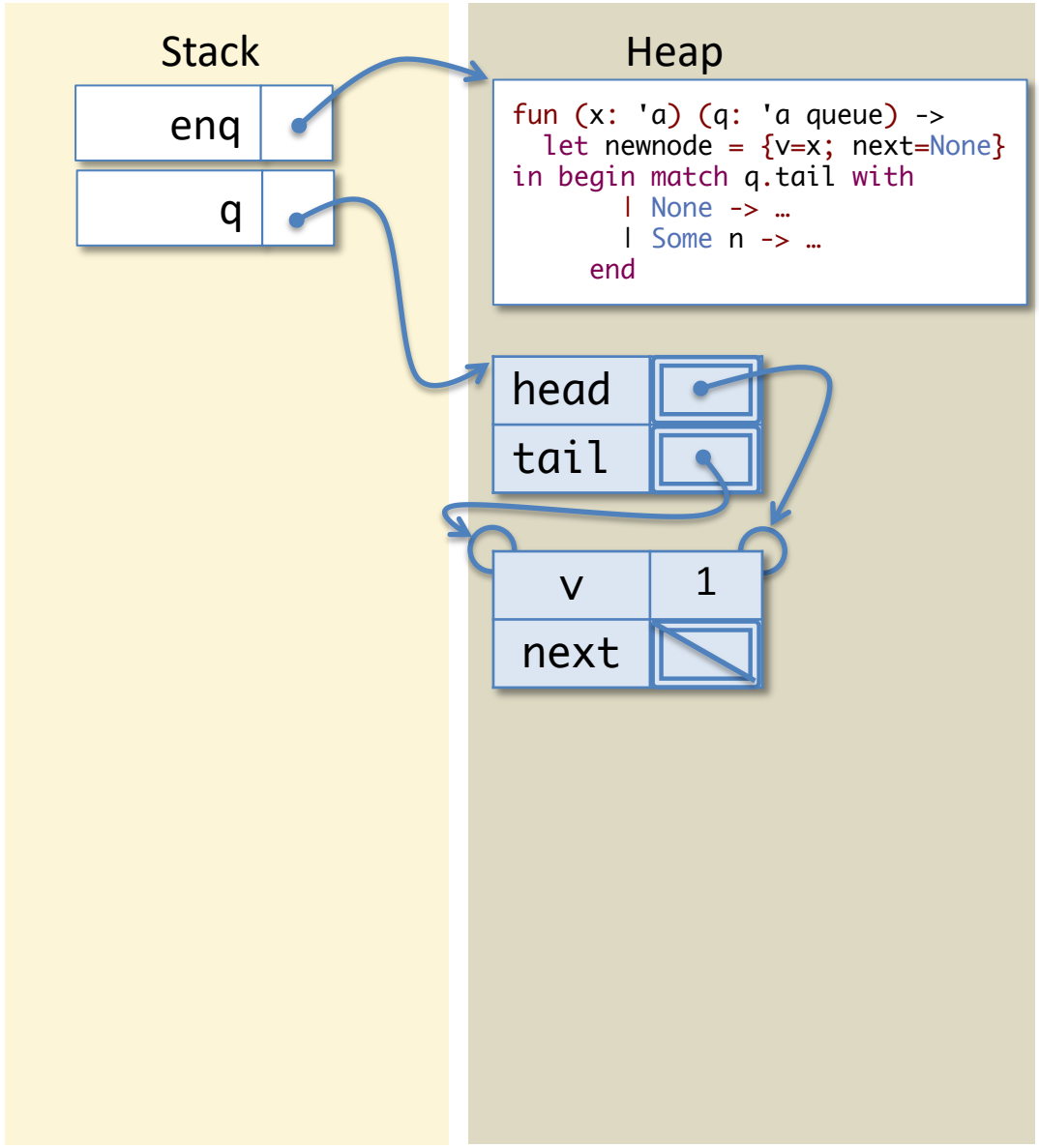
```
(* add an element to the tail of a queue *)
let enq (x: 'a) (q: 'a queue) : unit =
   let newnode = {v=x; next=None} in
   begin match q.tail with
     | None ->
         q.head <- Some newnode;
         q.tail <- Some newnode
     | Some n ->
         n.next <- Some newnode;
         q.tail <- Some 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 must “patch up” the “next” link of the old tail node to maintain the queue invariant.

Calling Enq on a non-empty queue

Workspace

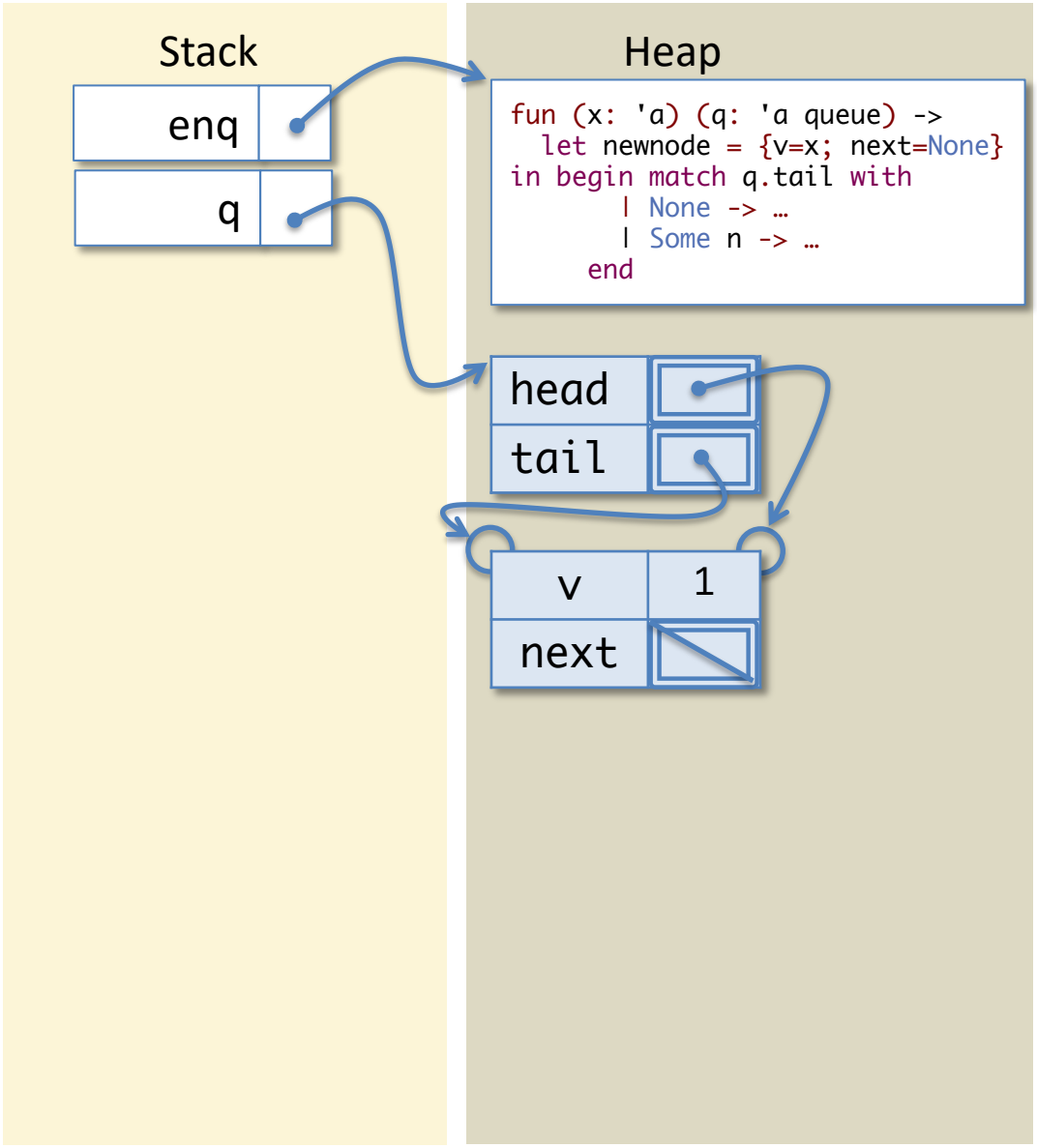
```
enq 2 q
```



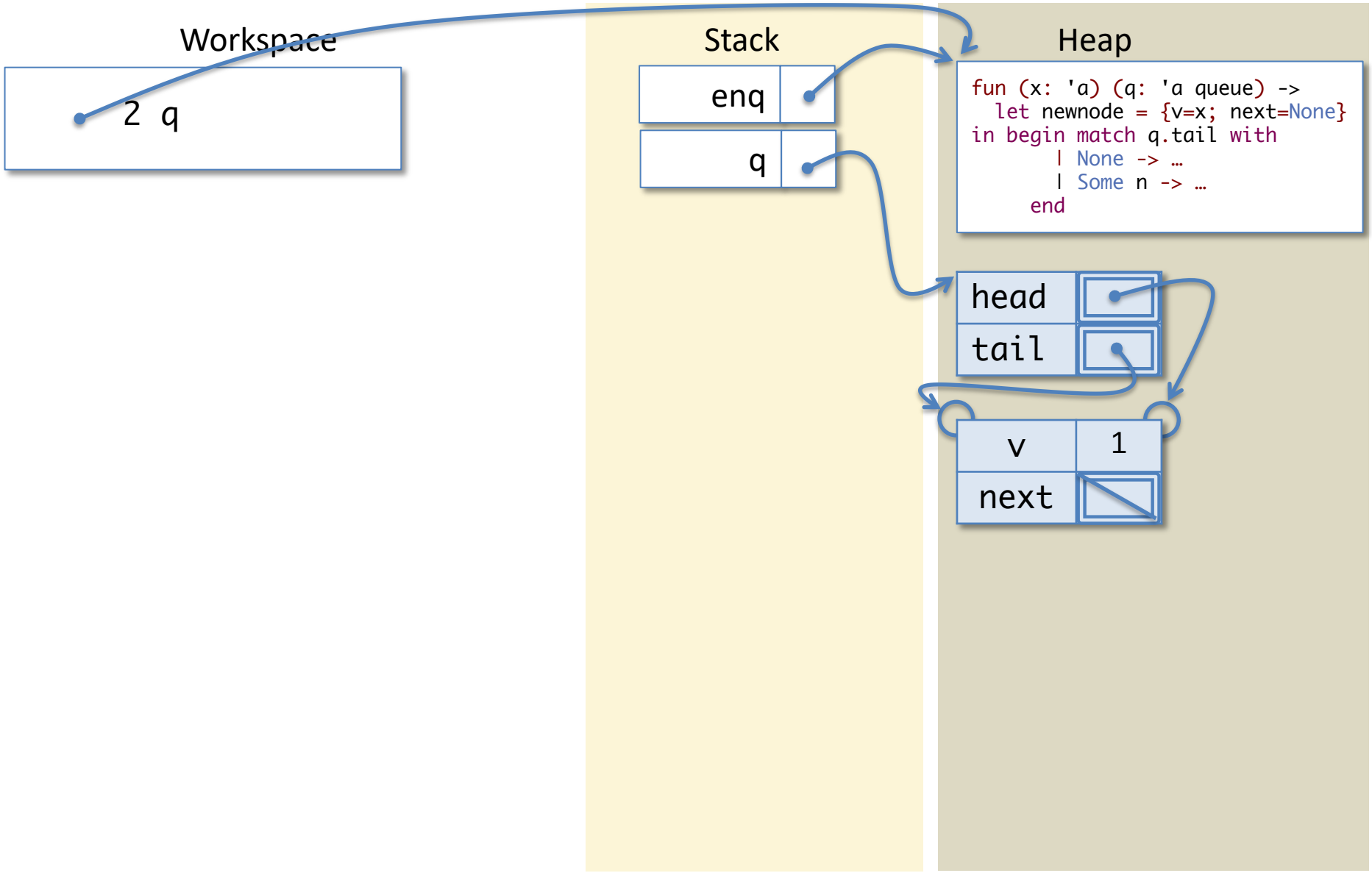
Calling Enq on a non-empty queue

Workspace

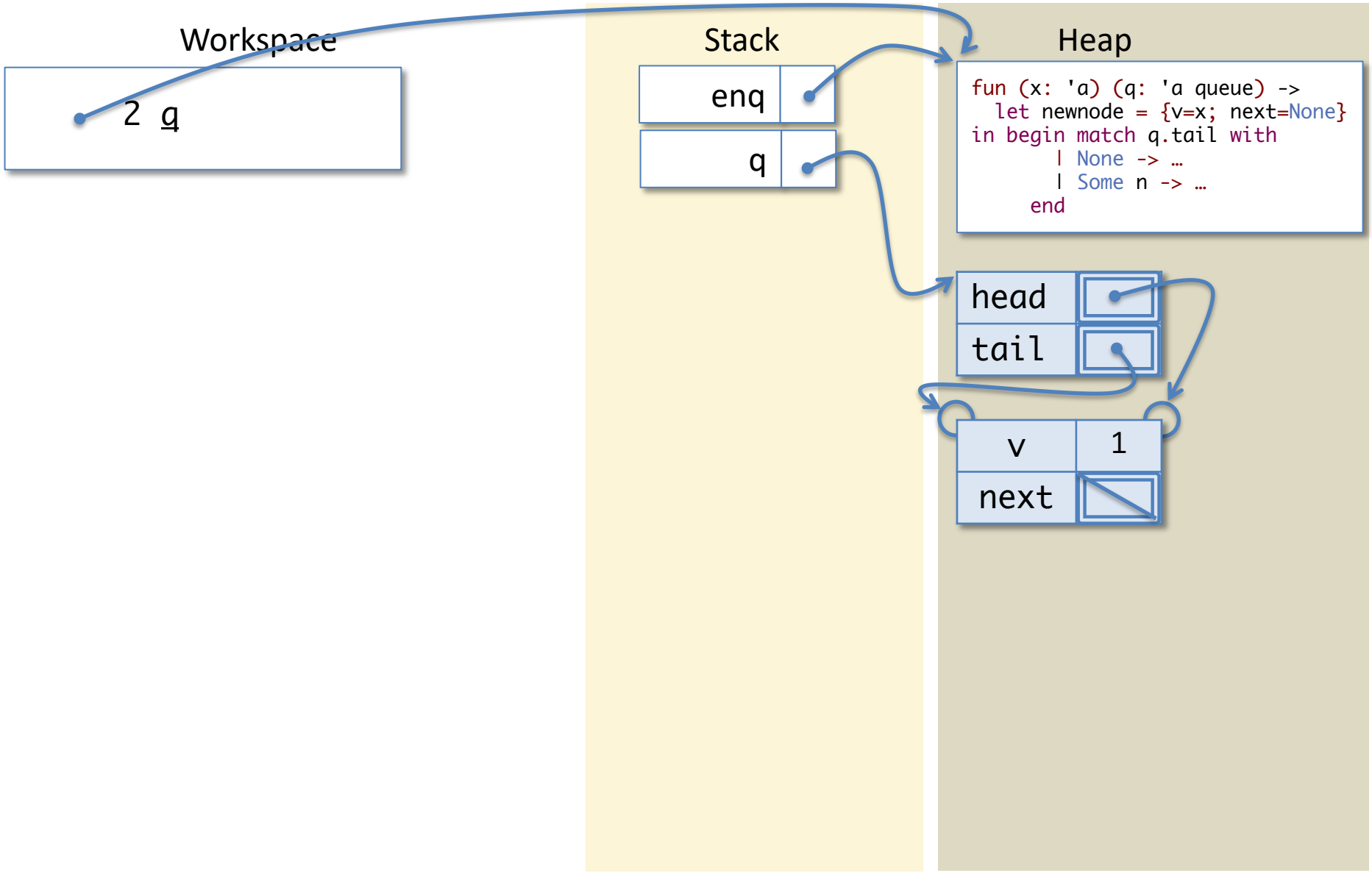
```
enq 2 q
```



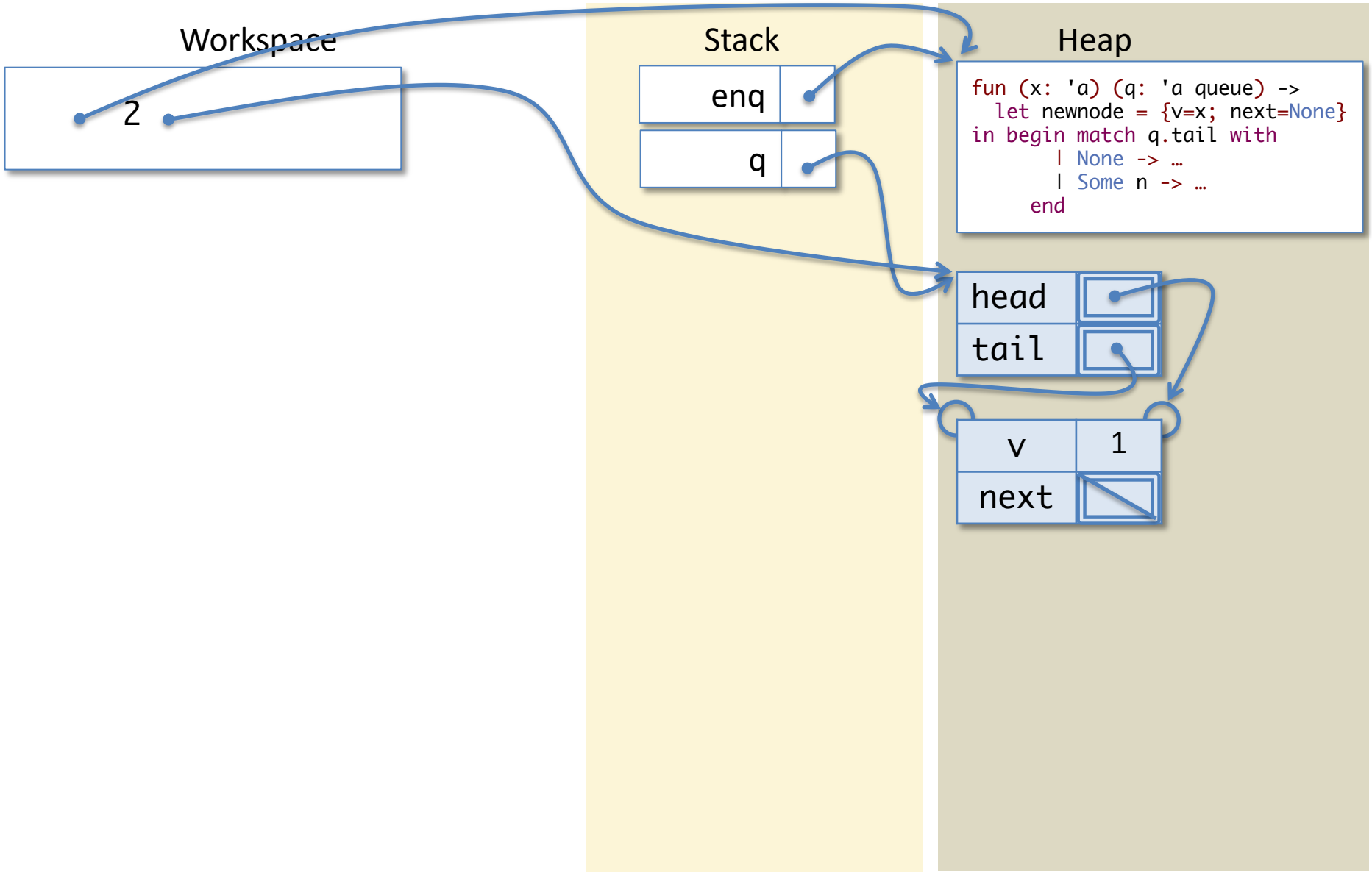
Calling Enq on a non-empty queue



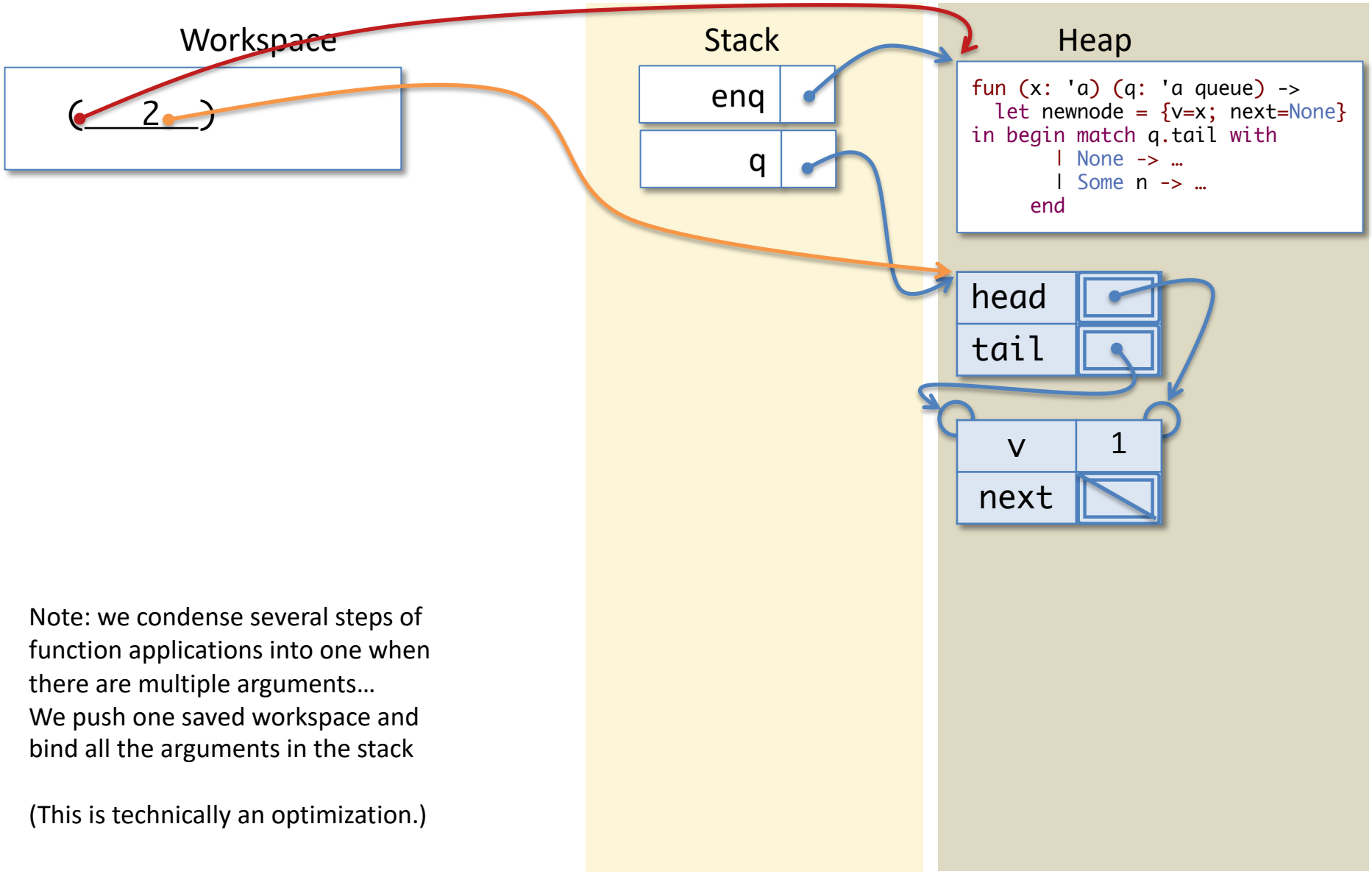
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



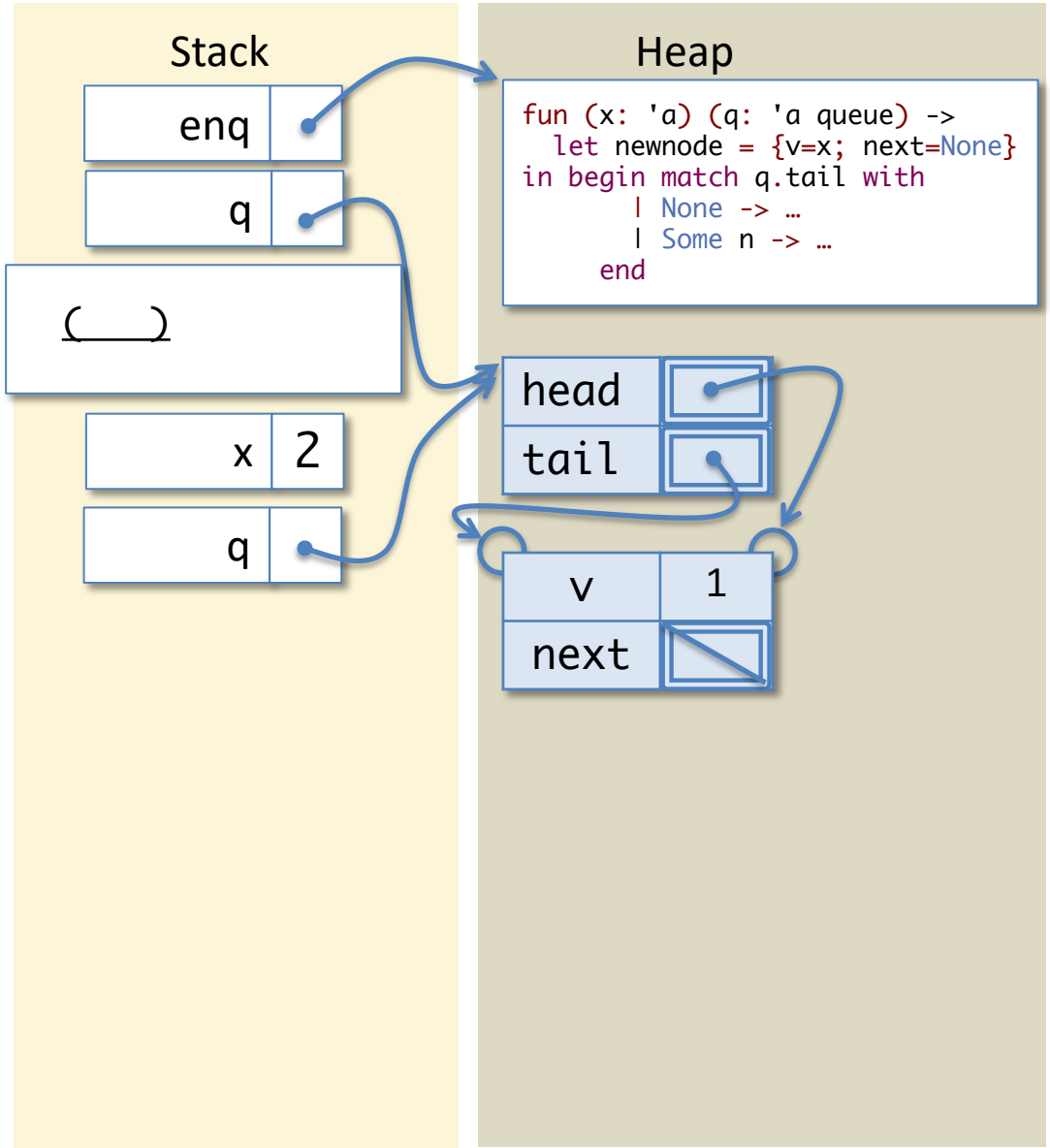
Note: we condense several steps of function applications into one when there are multiple arguments…
We push one saved workspace and bind all the arguments in the stack

(This is technically an optimization.)

Calling Enq on a non-empty queue

Workspace

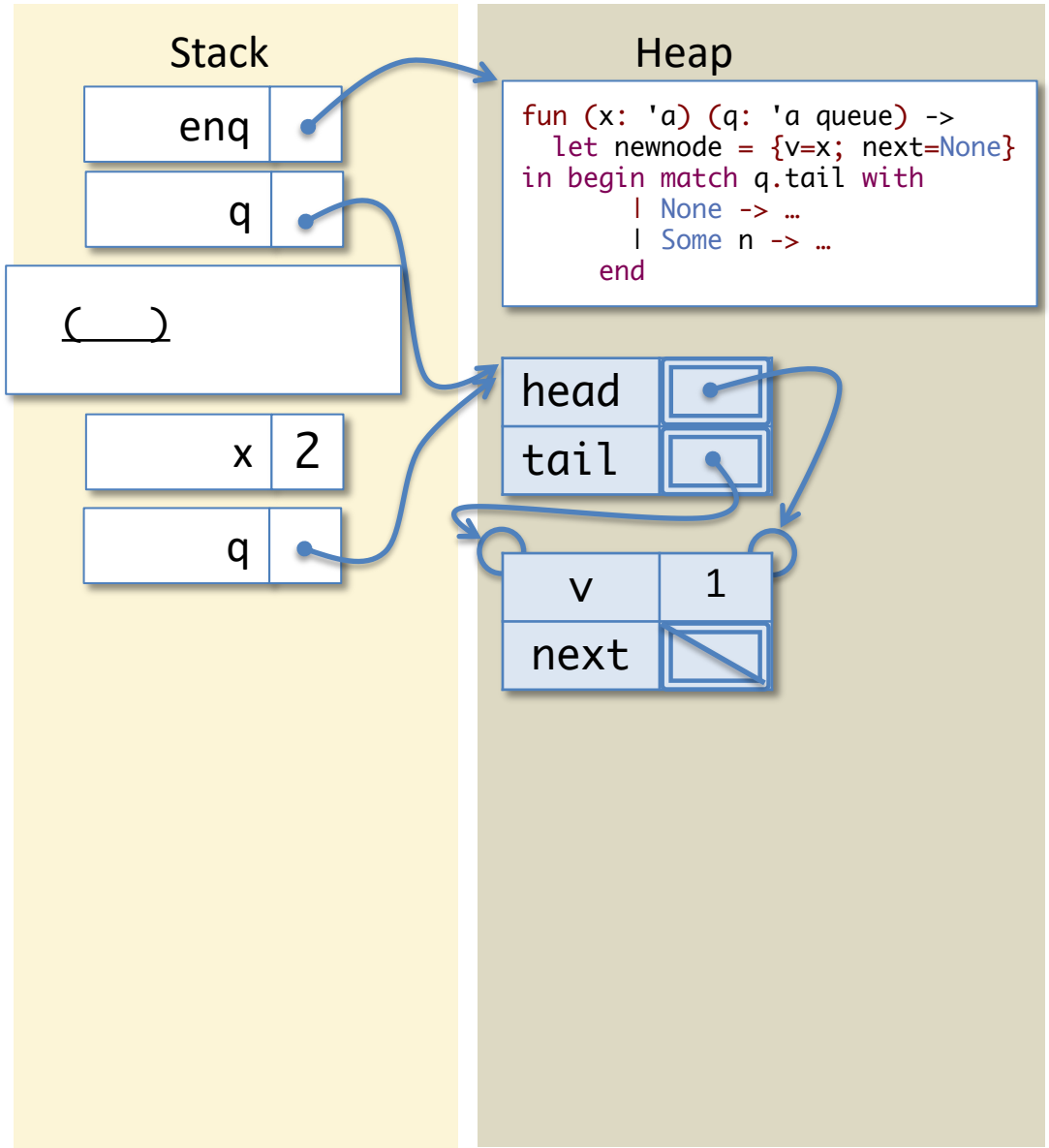
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

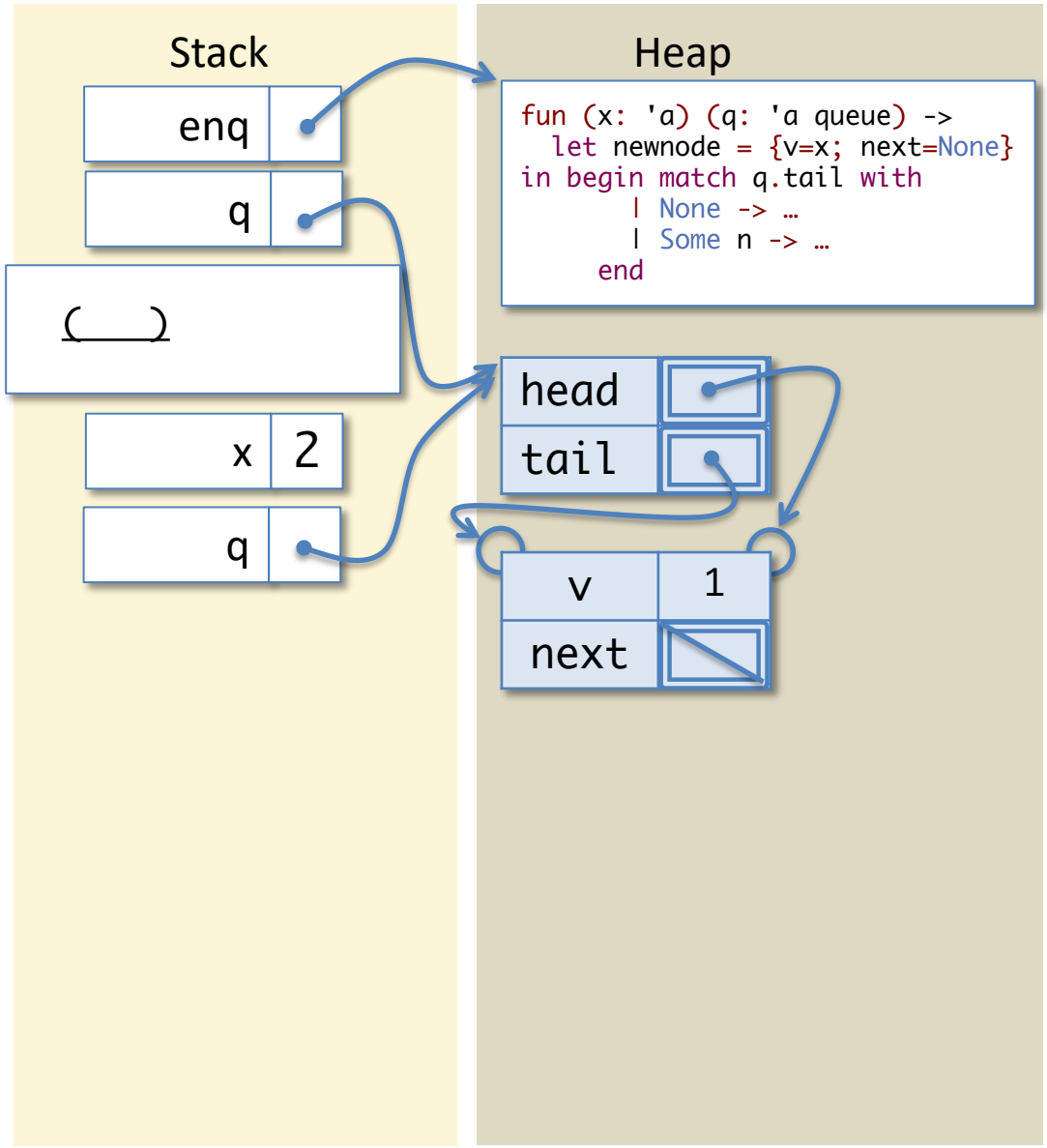
```
let newnode = {v=x; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue

Workspace

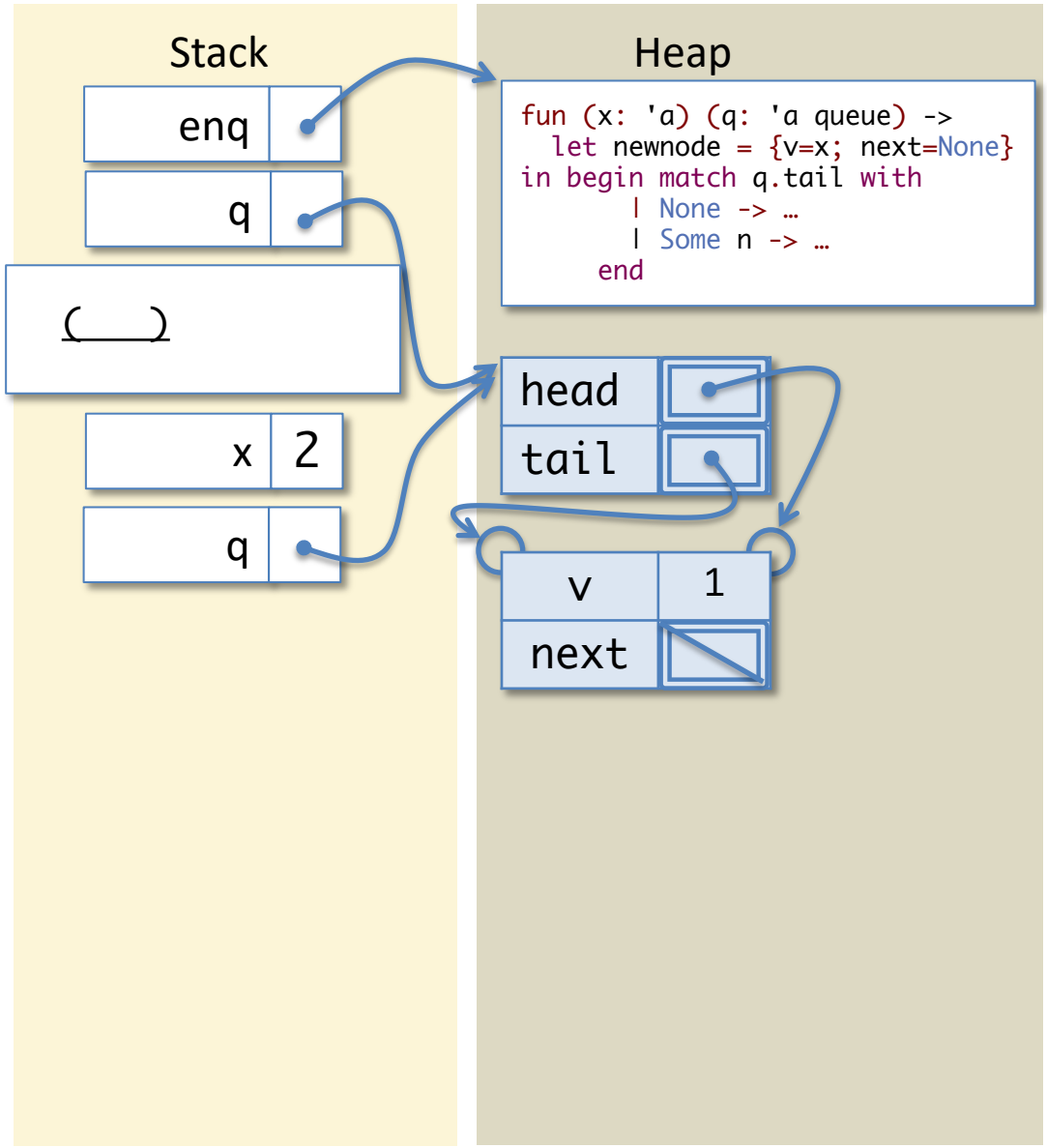
```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



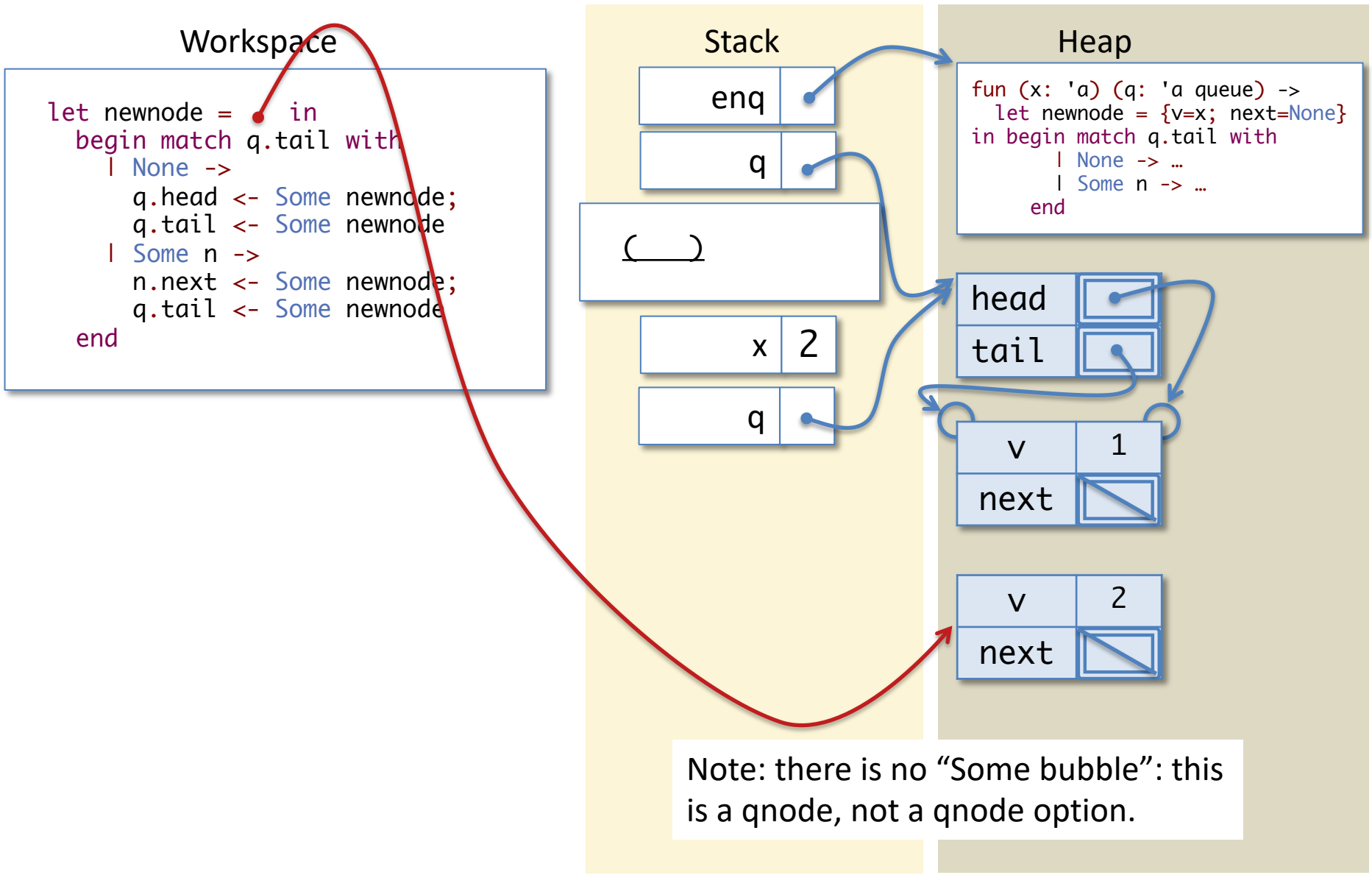
Calling Enq on a non-empty queue

Workspace

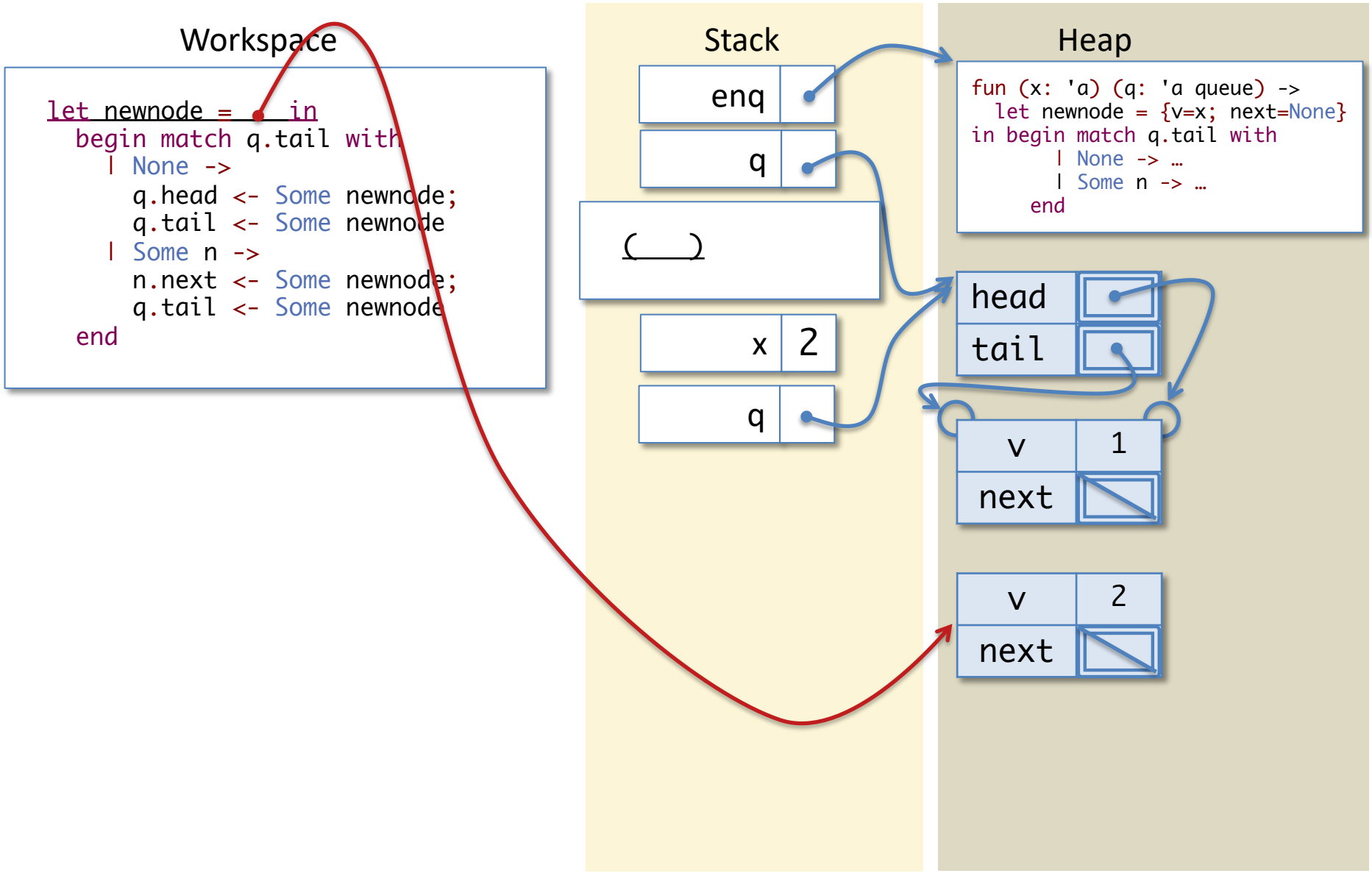
```
let newnode = {v=2; next=None} in
  begin match q.tail with
    | None ->
      q.head <- Some newnode;
      q.tail <- Some newnode
    | Some n ->
      n.next <- Some newnode;
      q.tail <- Some newnode
  end
```



Calling Enq on a non-empty queue



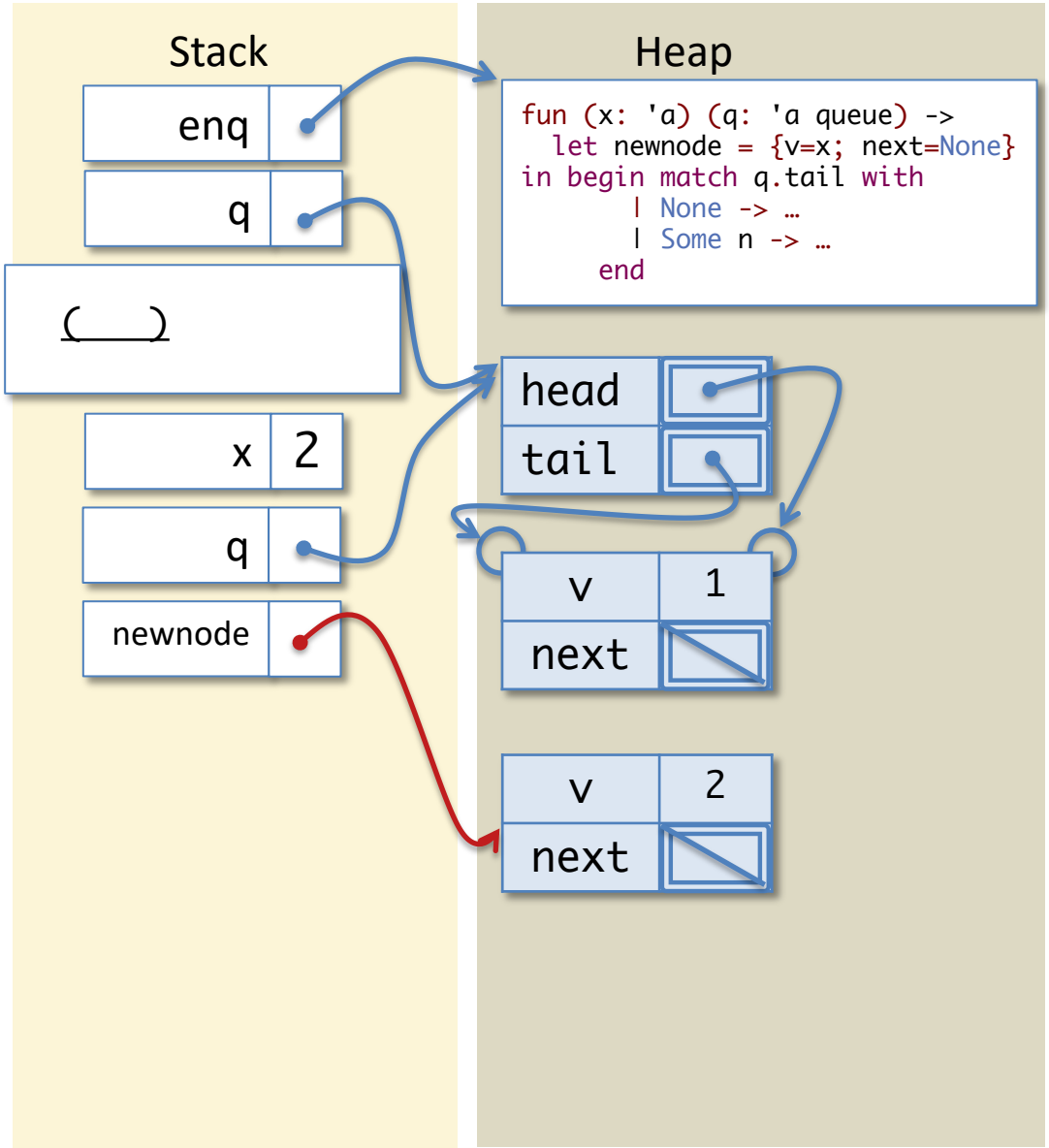
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

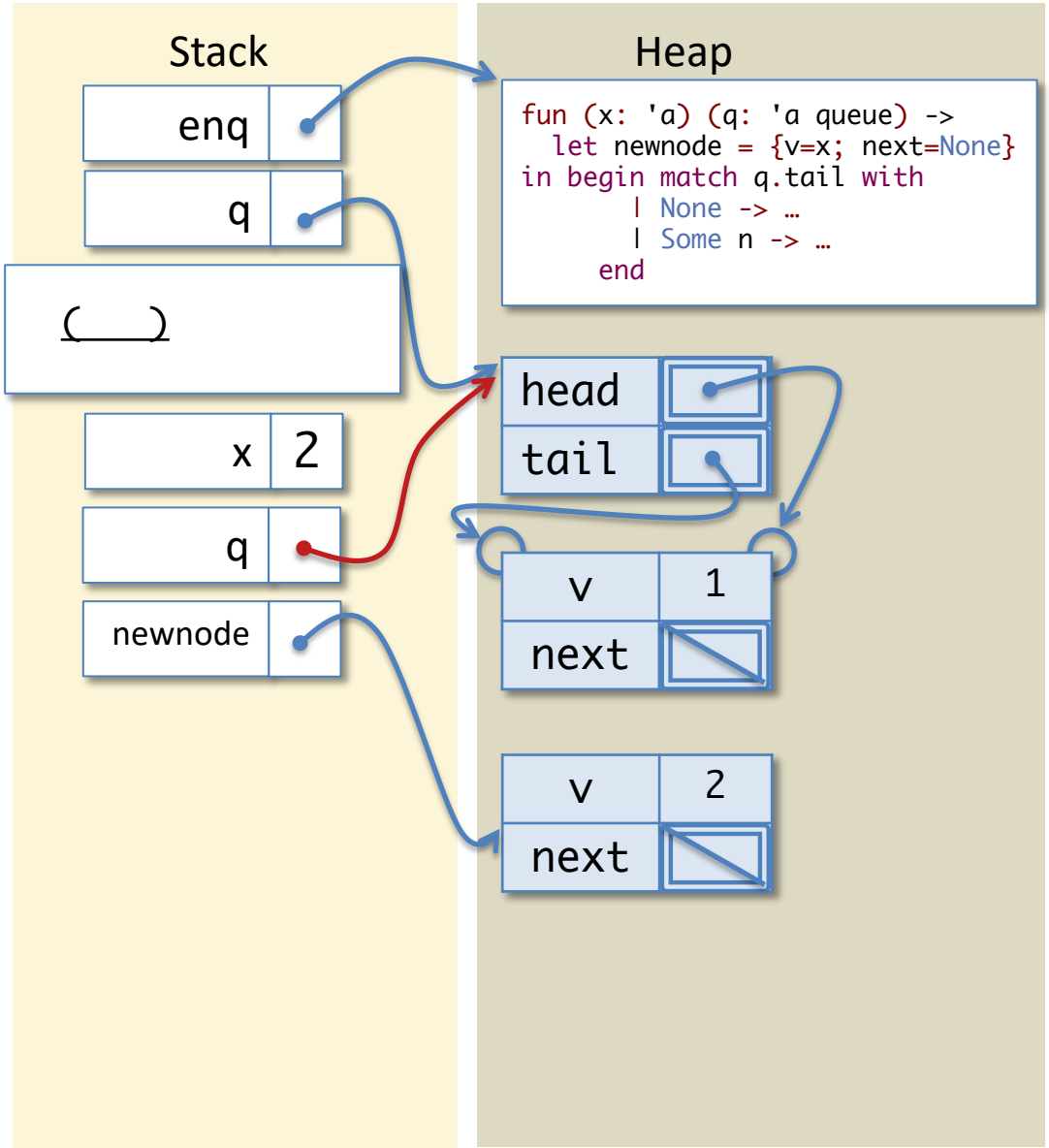
```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



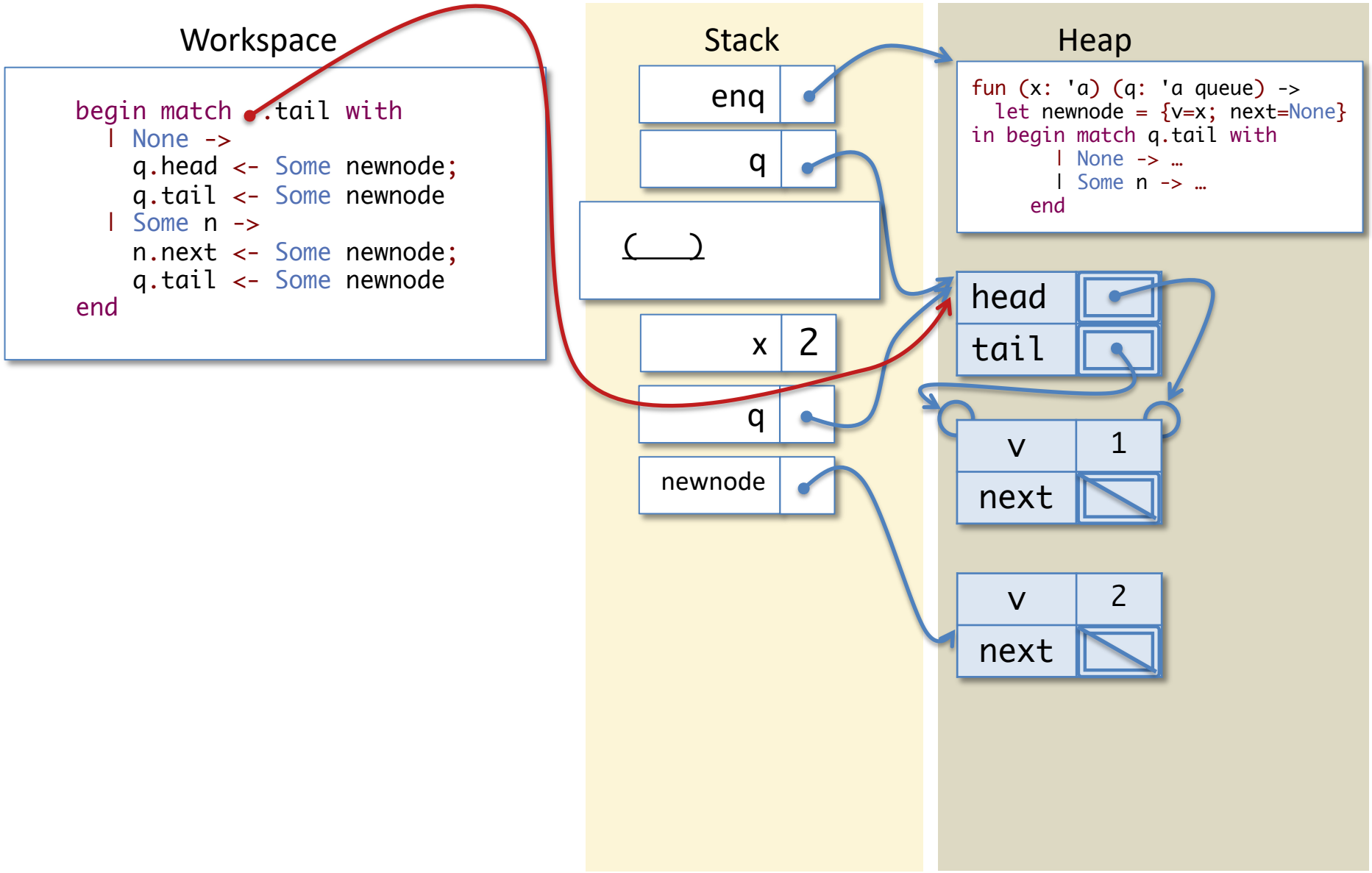
Calling Enq on a non-empty queue

Workspace

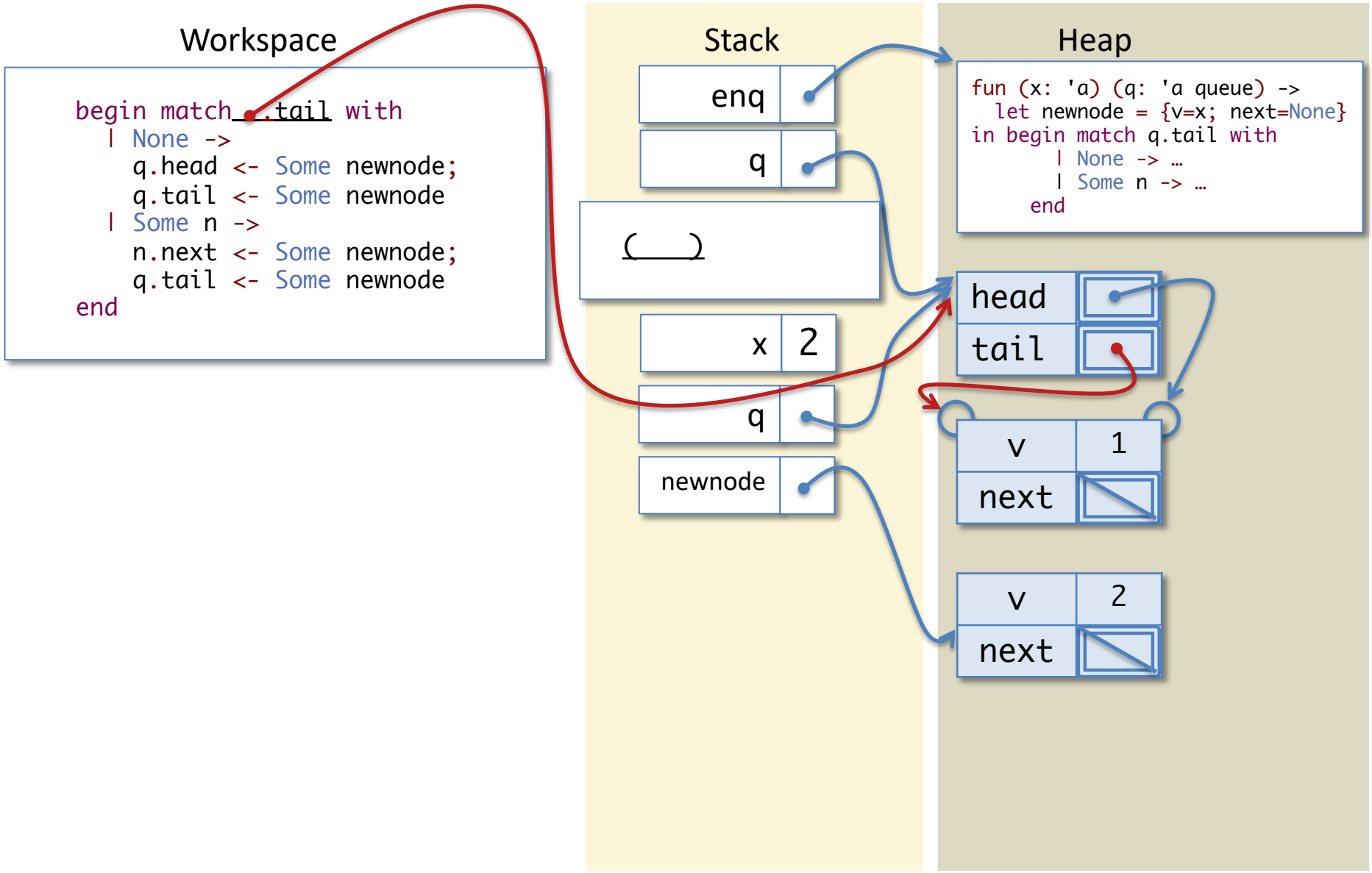
```
begin match q.tail with
  | None ->
    q.head <- Some newnode;
    q.tail <- Some newnode
  | Some n ->
    n.next <- Some newnode;
    q.tail <- Some newnode
end
```



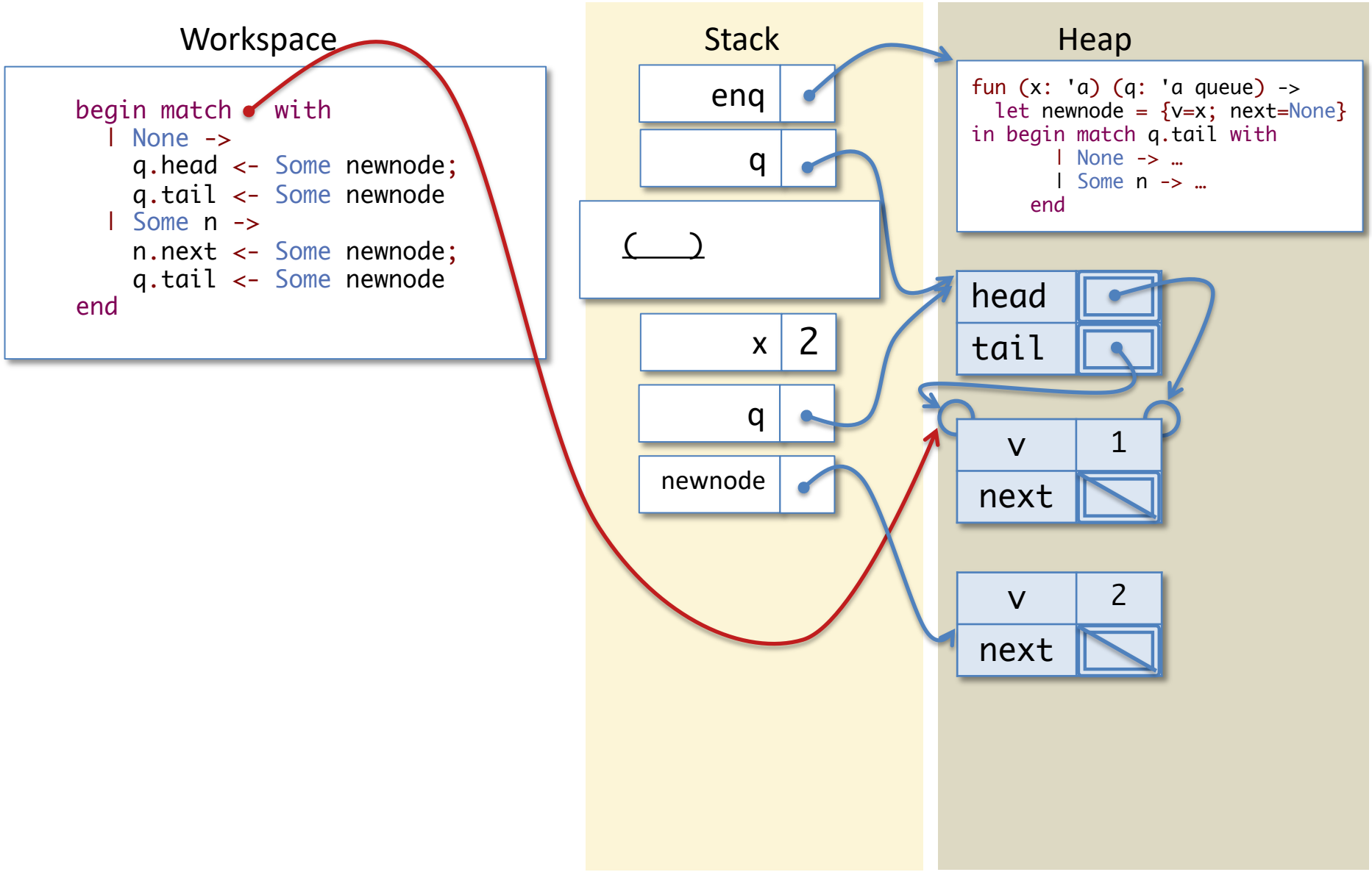
Calling Enq on a non-empty queue



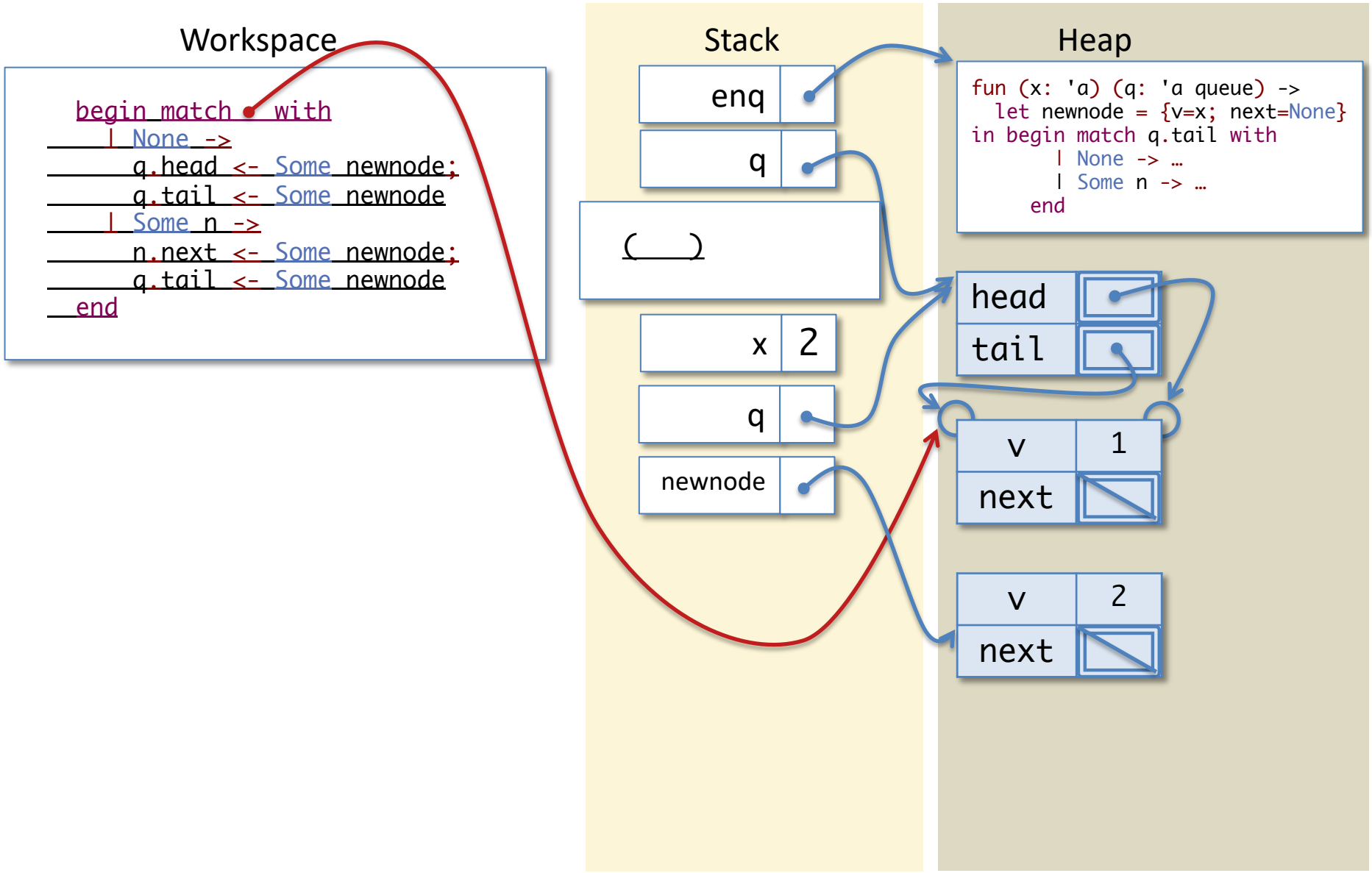
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Simplifying Match

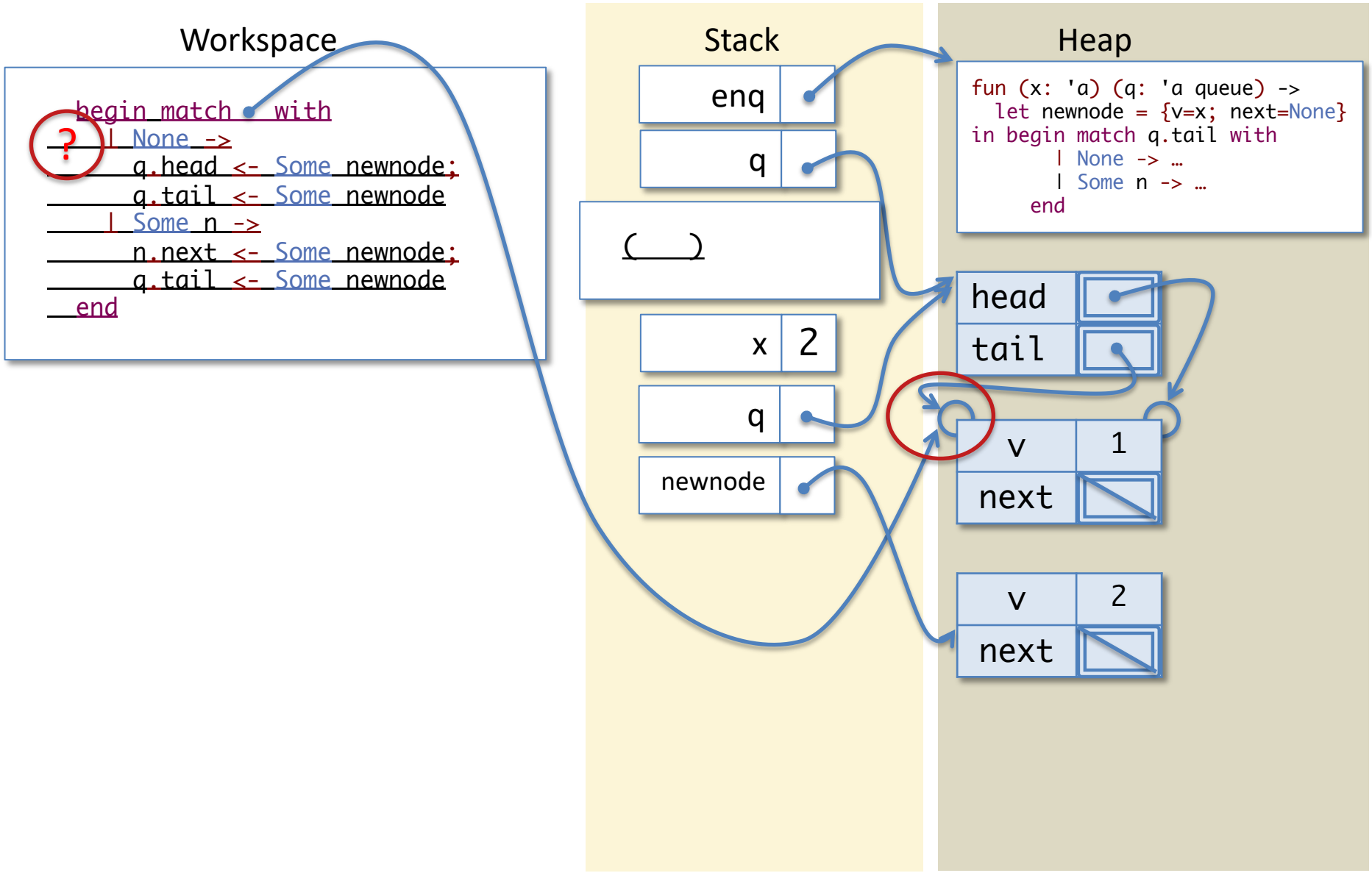
- A match expression

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

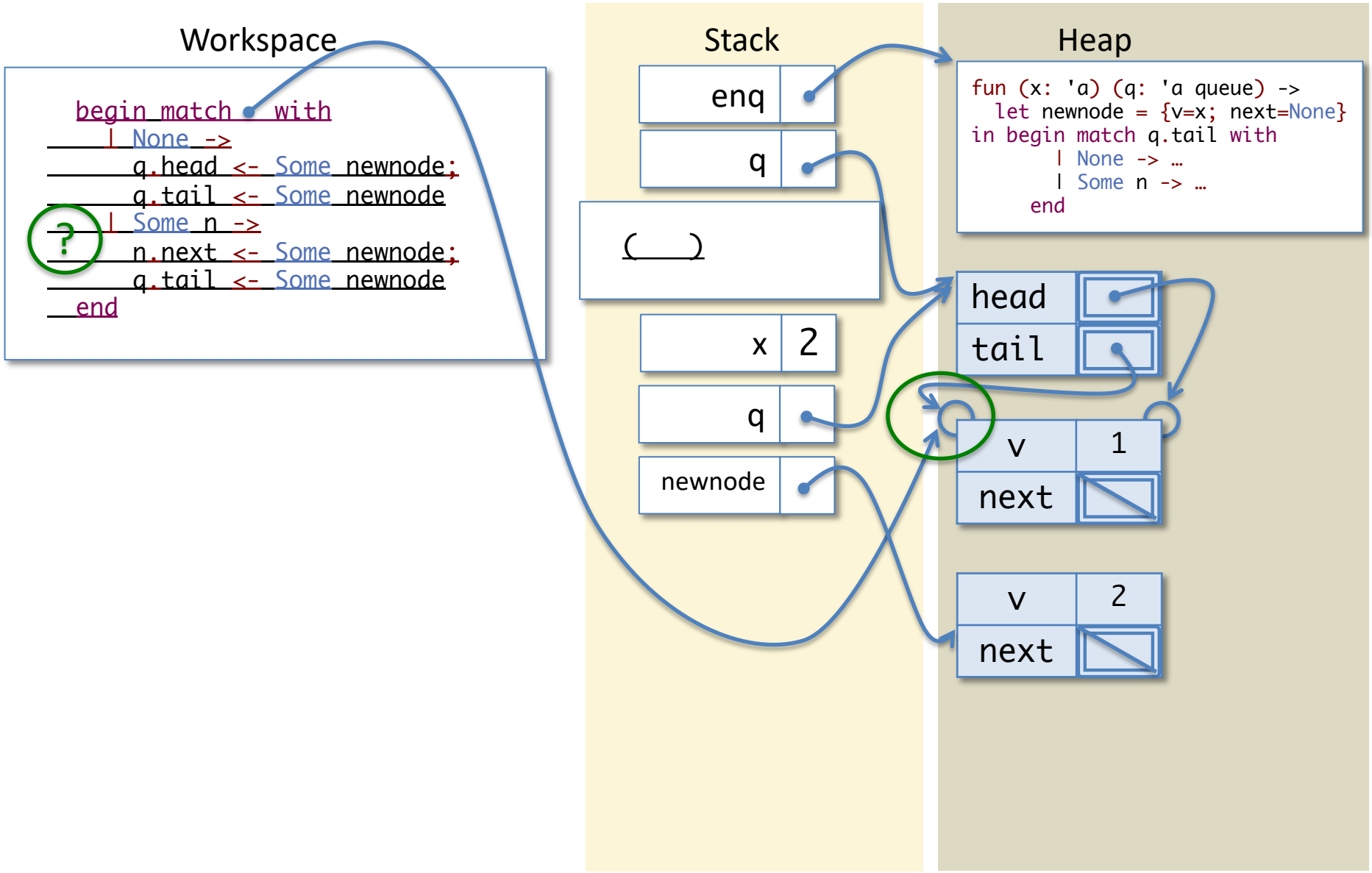
  is ready if e is a value

  - Note that e will always be a pointer to a constructor cell in the heap
  - This expression is simplified by finding the first pattern pat_i that matches the cell and adding new bindings for the pattern variables (to the parts of e that line up) to the end of the stack
  - replacing the whole match expression in the workspace with the corresponding $branch_i$

Calling Enq on a non-empty queue



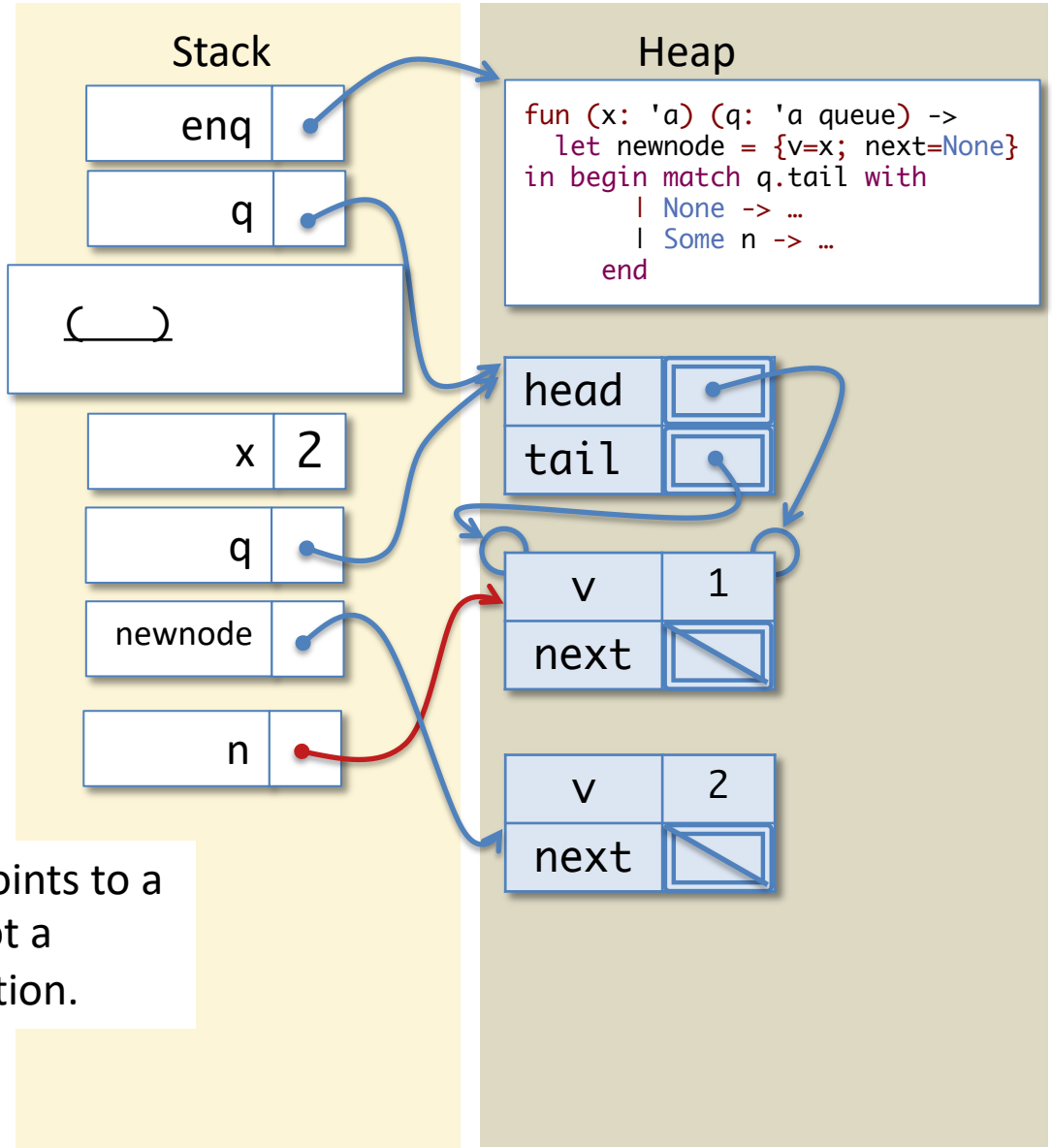
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

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

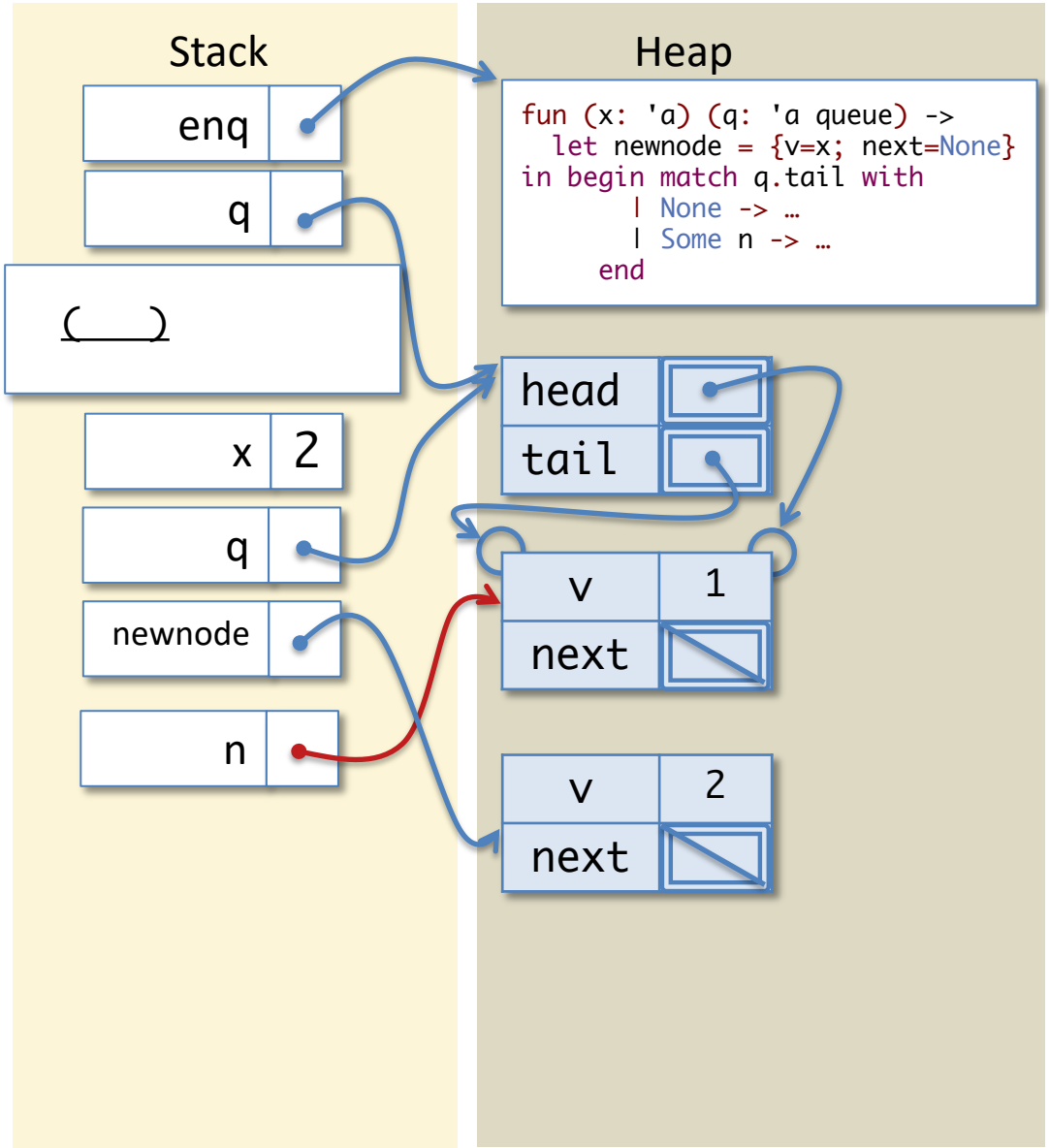


Note: n points to a qnode, not a qnode option.

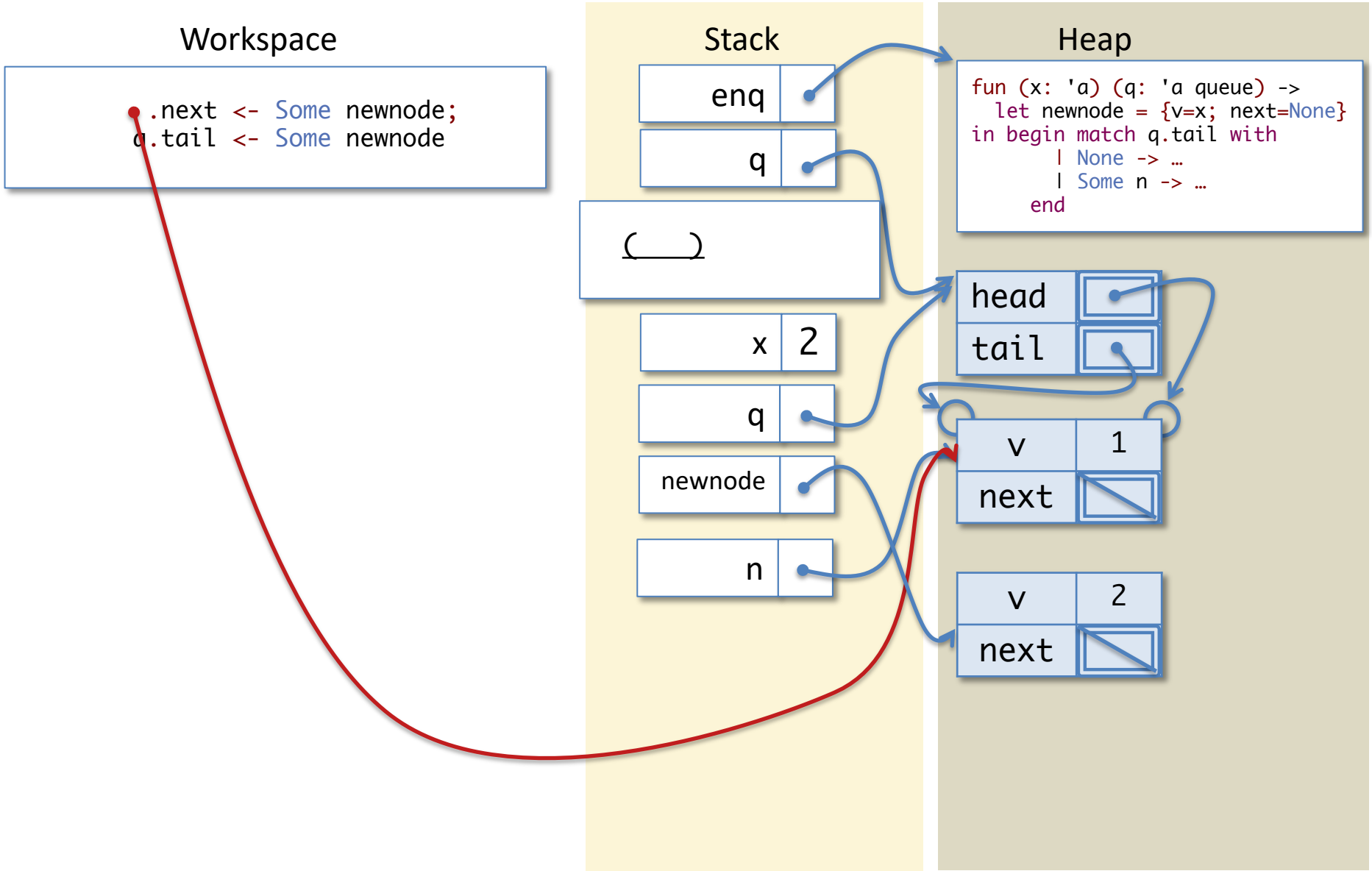
Calling Enq on a non-empty queue

Workspace

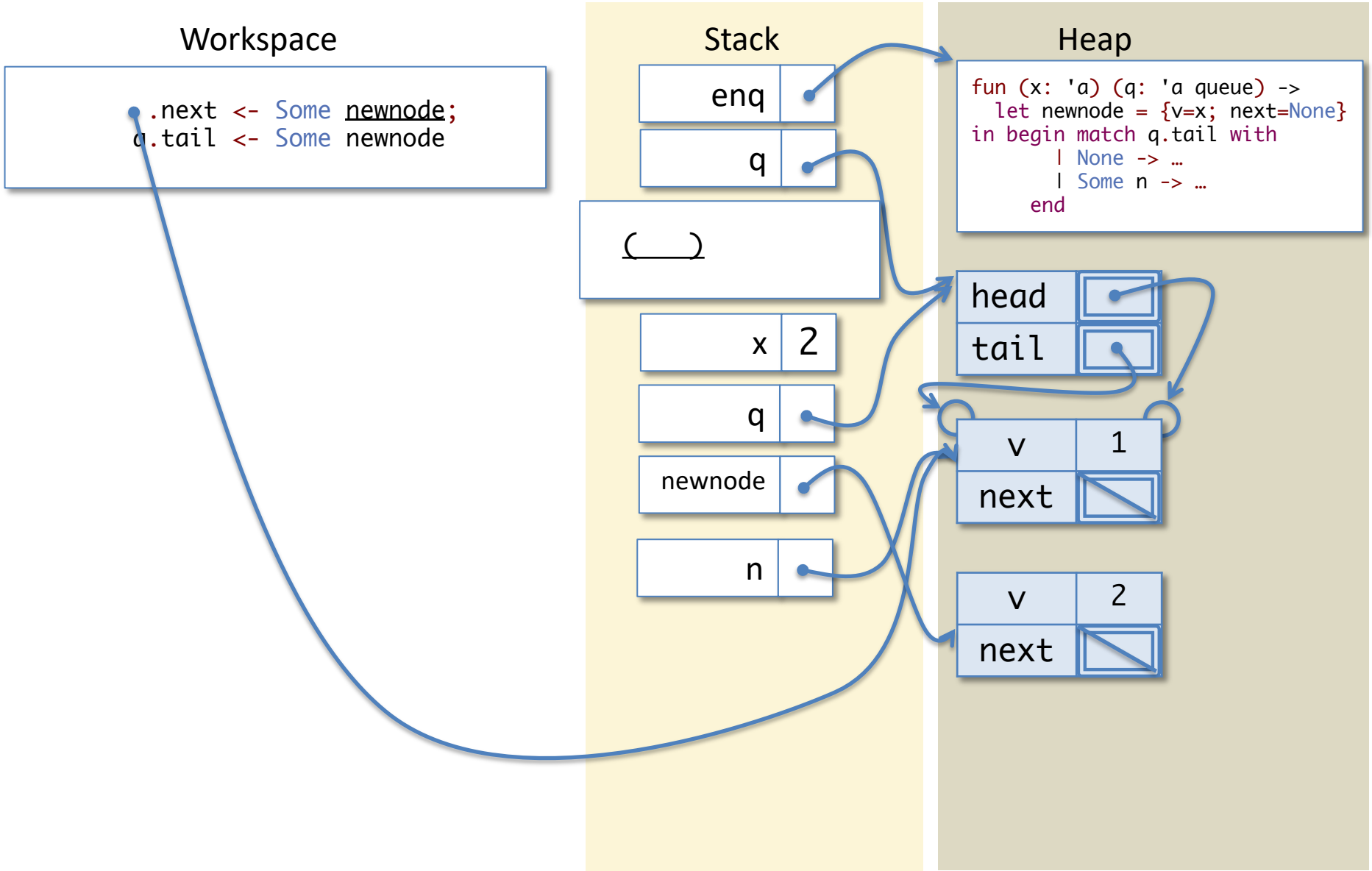
```
n.next <- Some newnode;
q.tail <- Some newnode
```



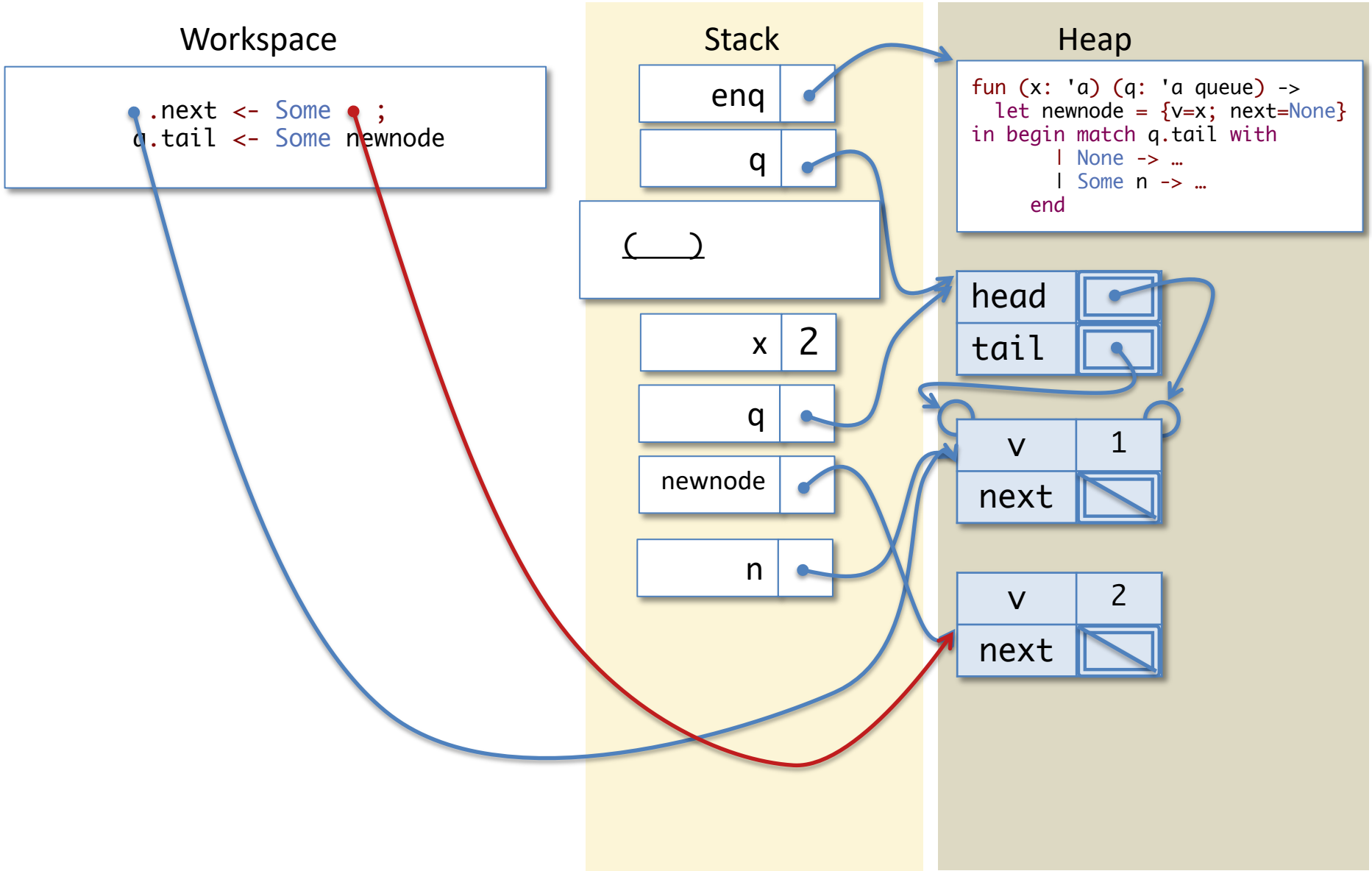
Calling Enq on a non-empty queue



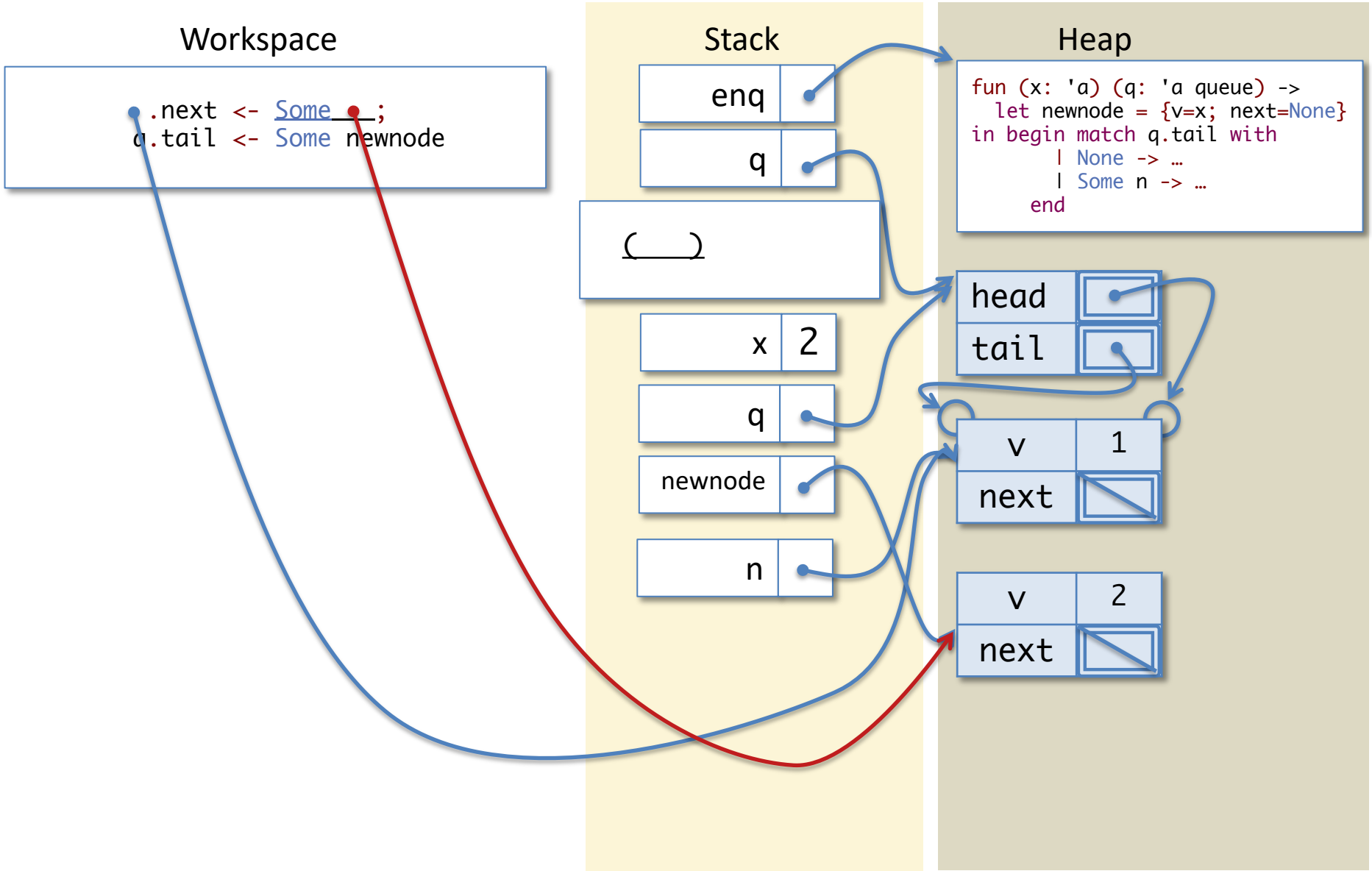
Calling Enq on a non-empty queue



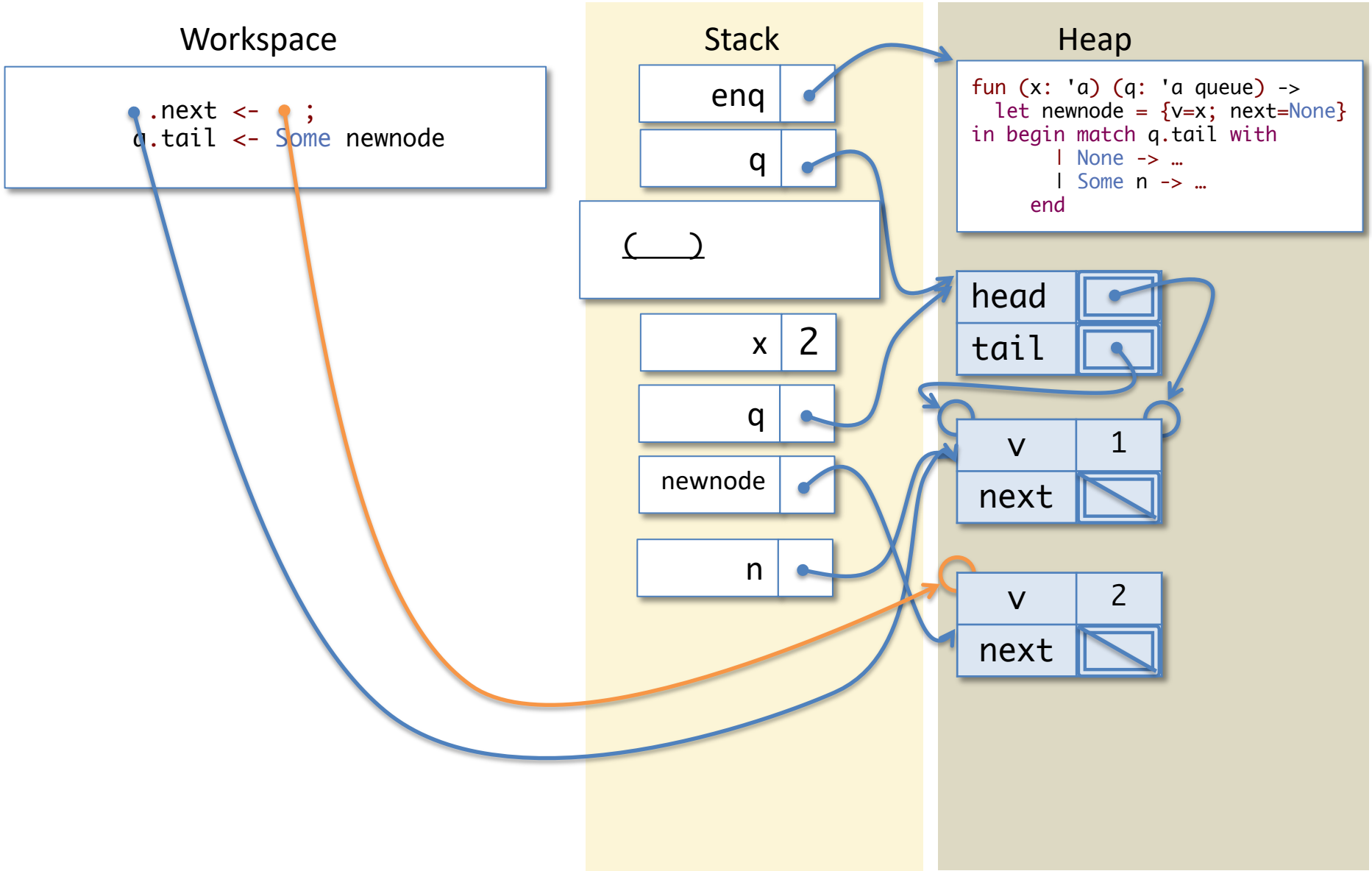
Calling Enq on a non-empty queue



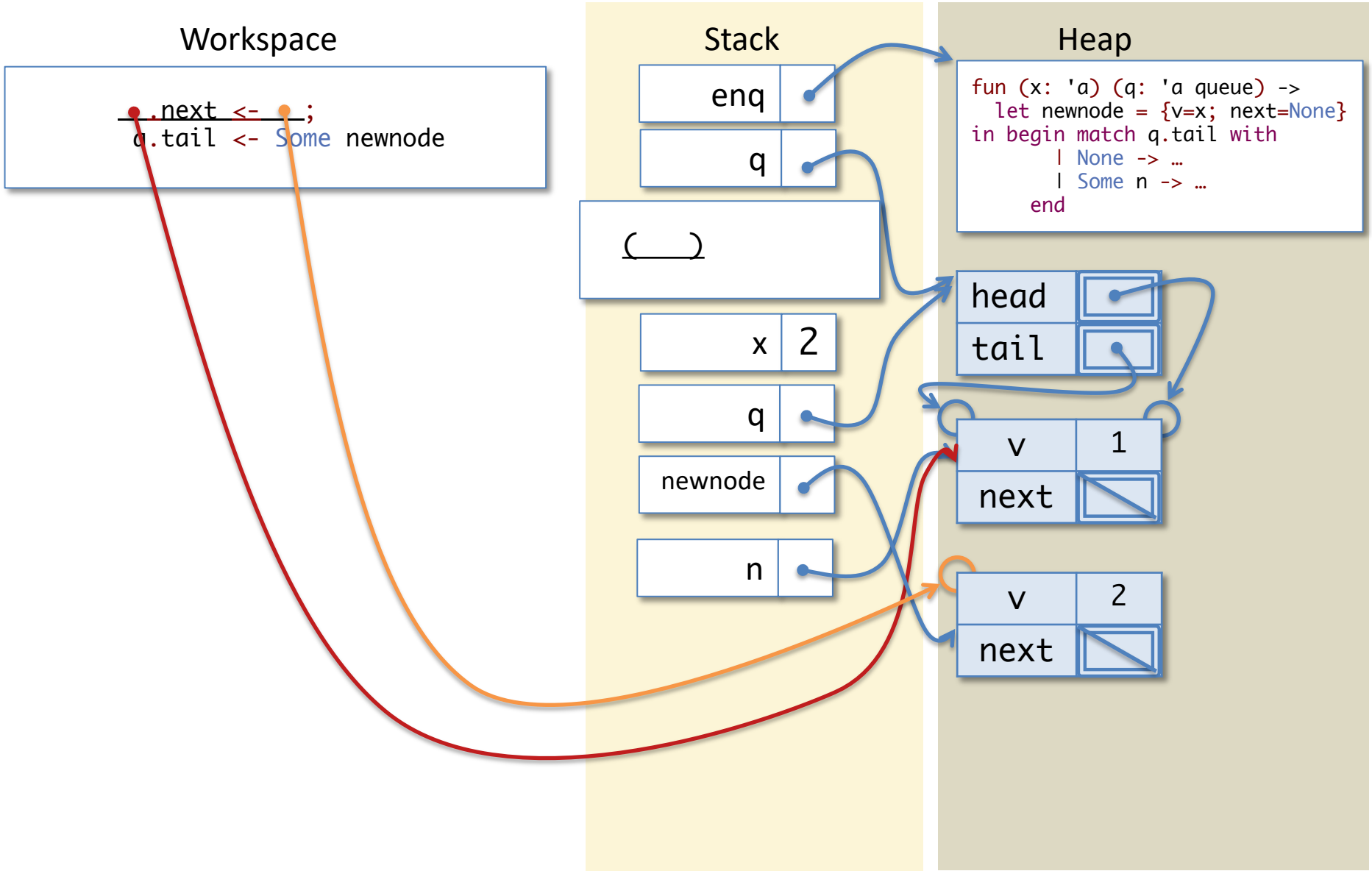
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



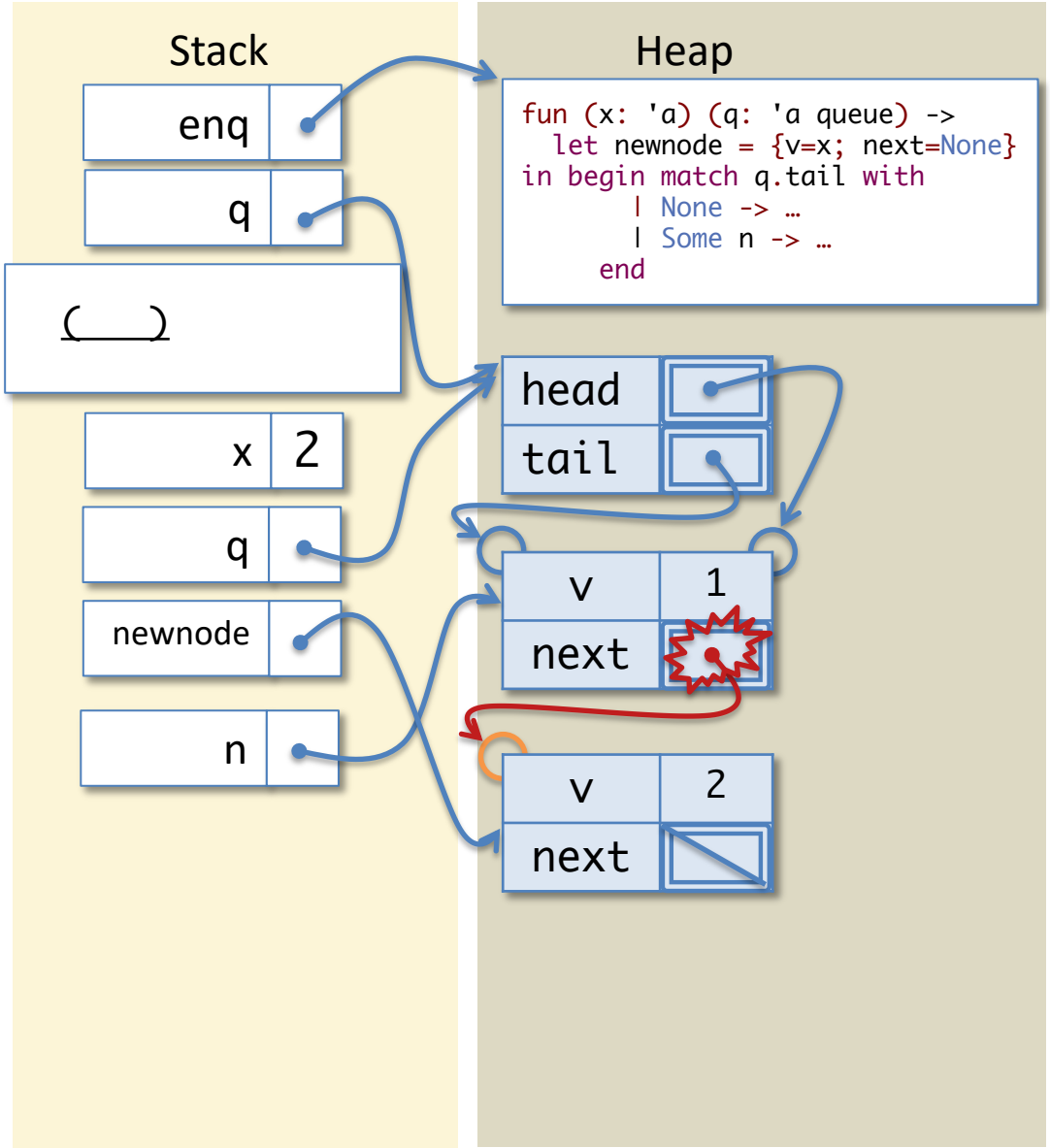
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue

Workspace

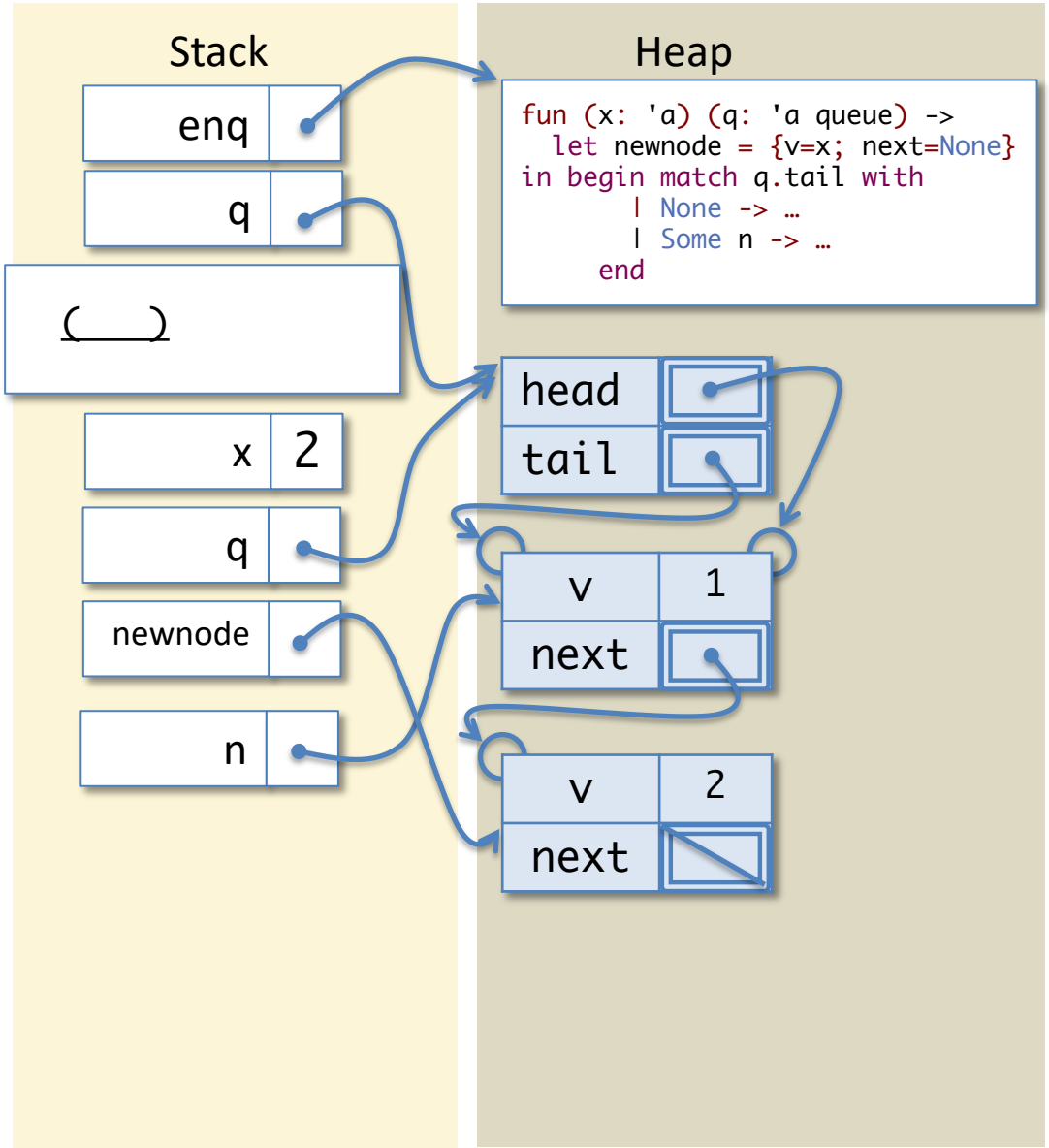
```
();
 q.tail <- Some newnode
```



Calling Enq on a non-empty queue

Workspace

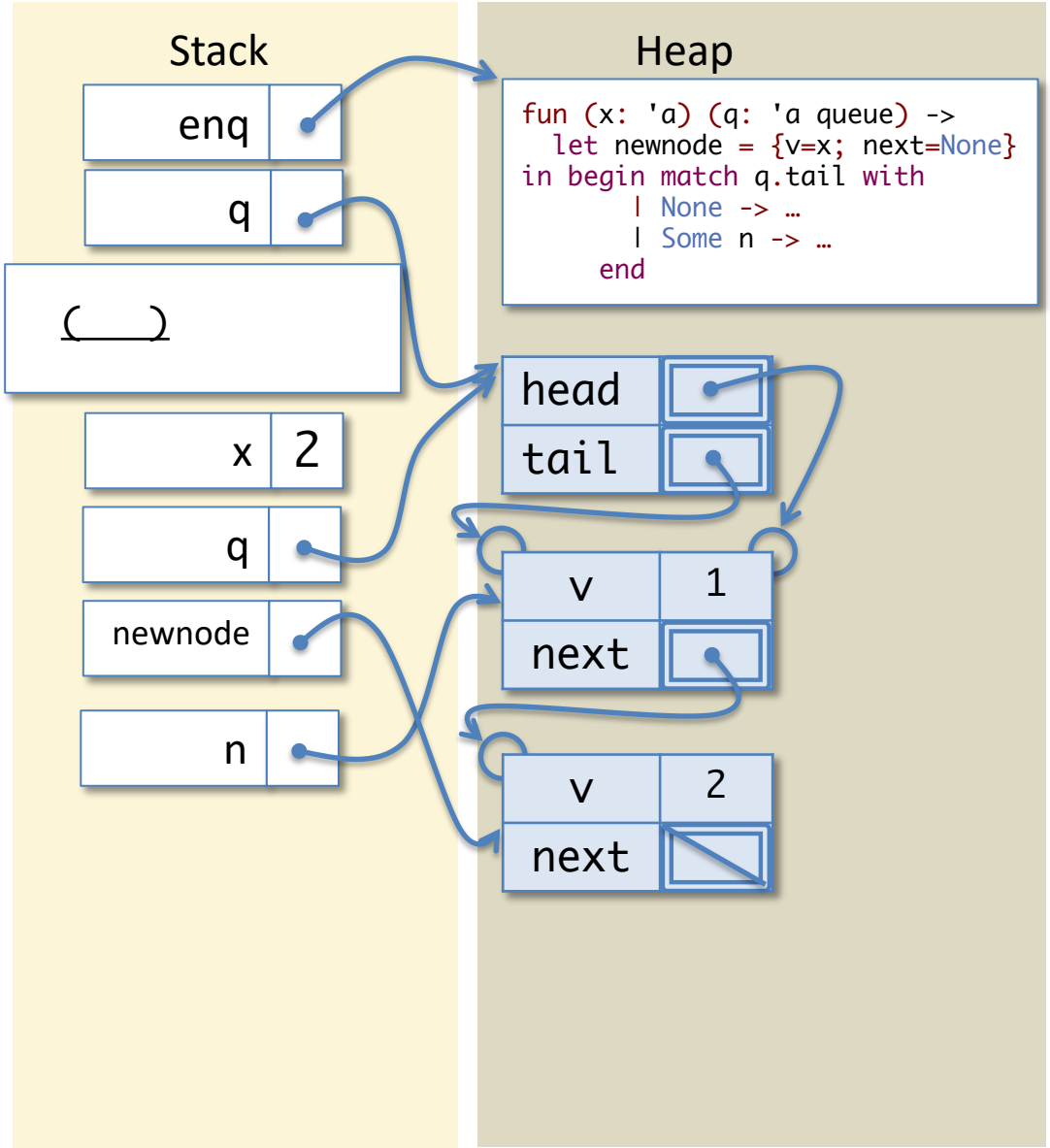
```
();
q.tail <- Some newnode
```



Calling Enq on a non-empty queue

Workspace

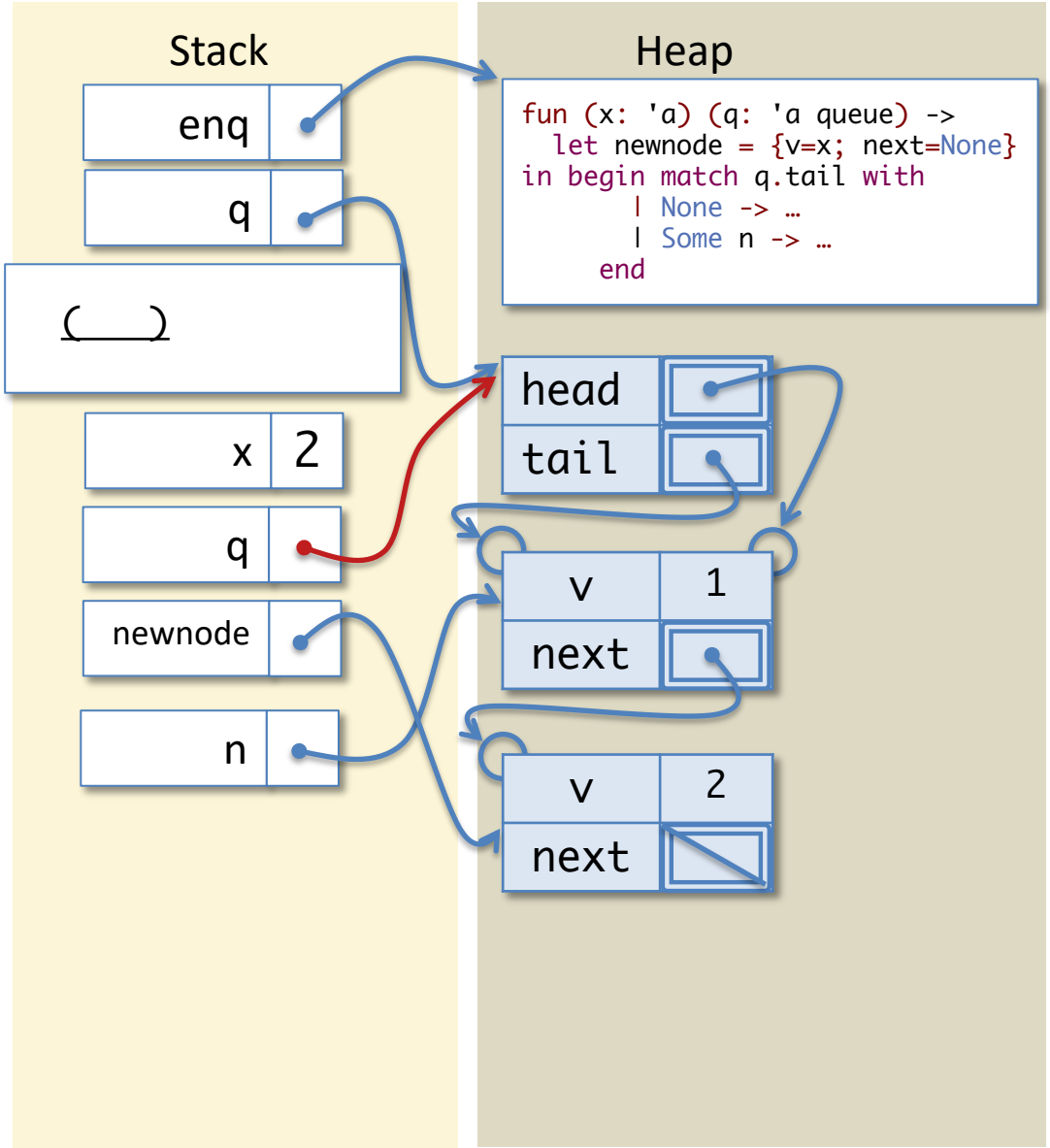
```
q.tail <- Some newnode
```



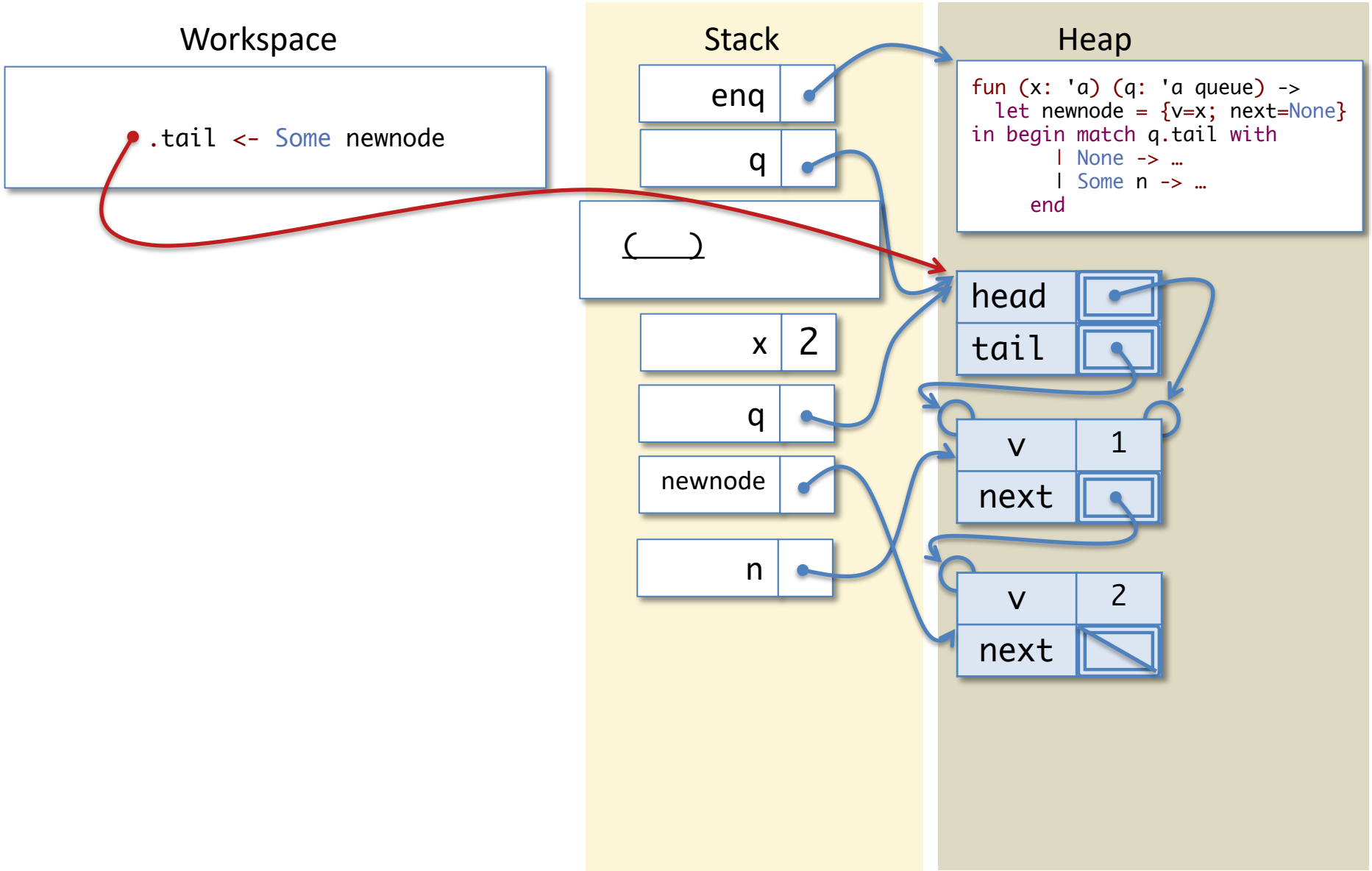
Calling Enq on a non-empty queue

Workspace

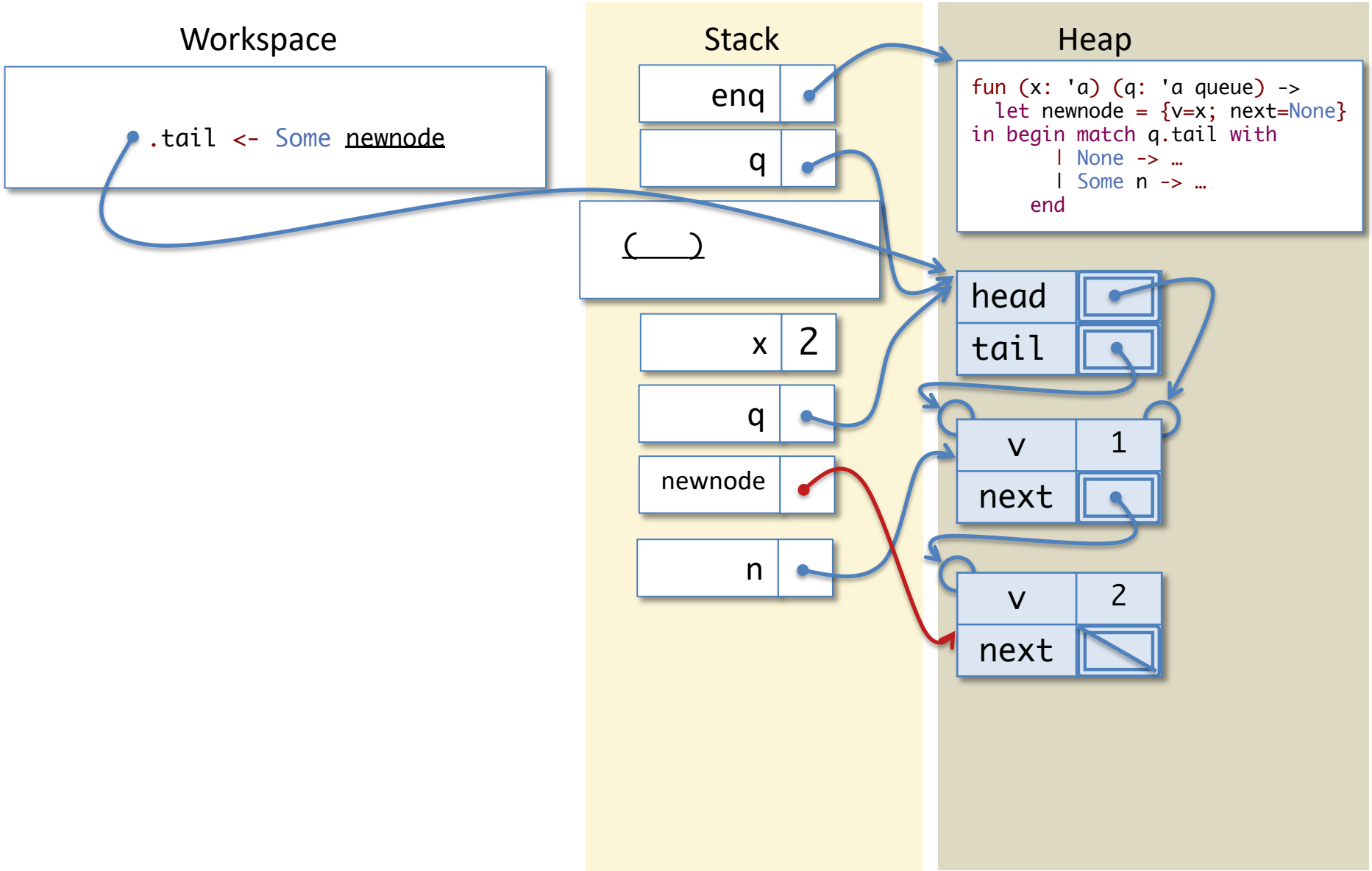
```
q.tail <- Some newnode
```



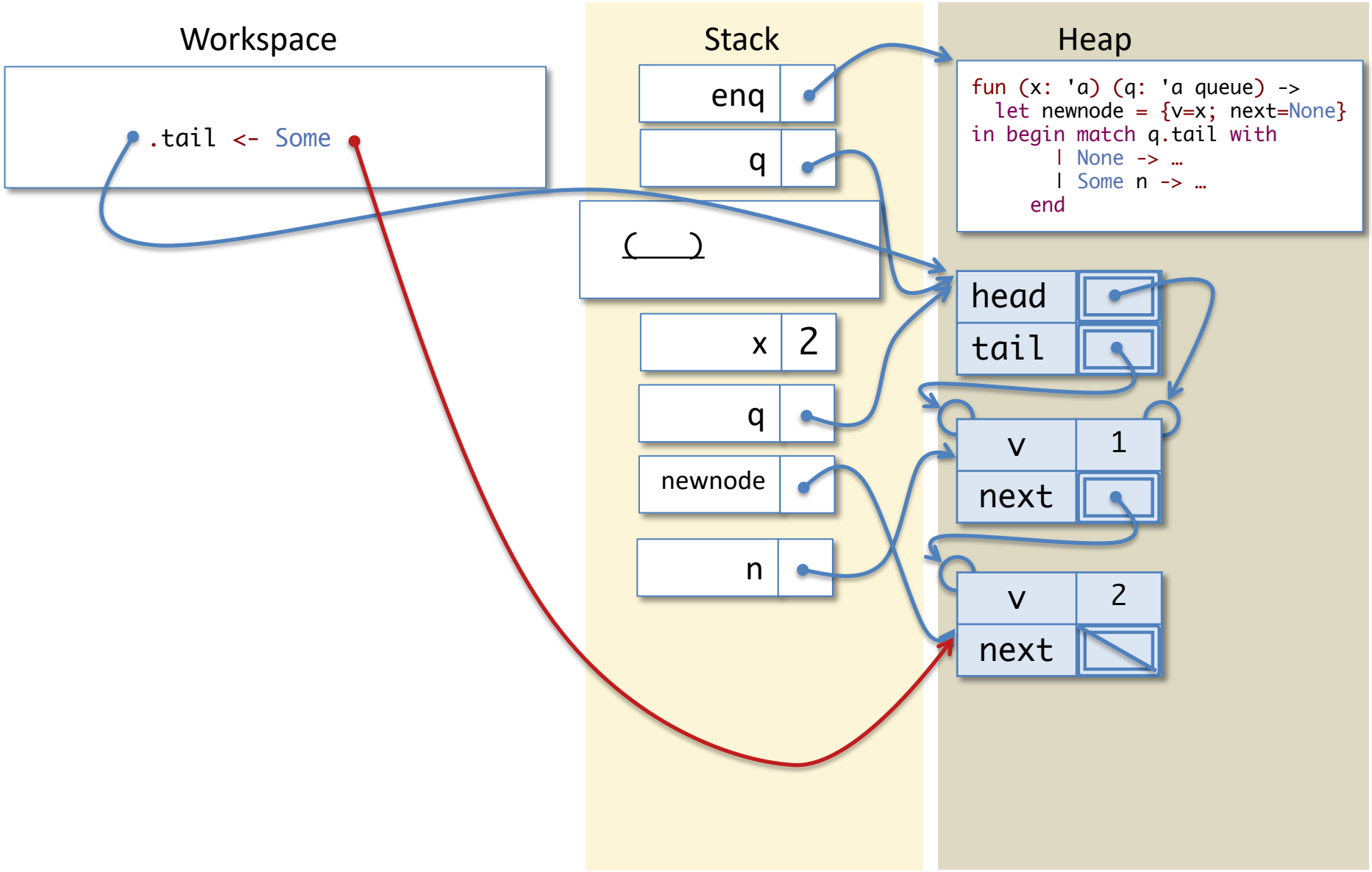
Calling Enq on a non-empty queue



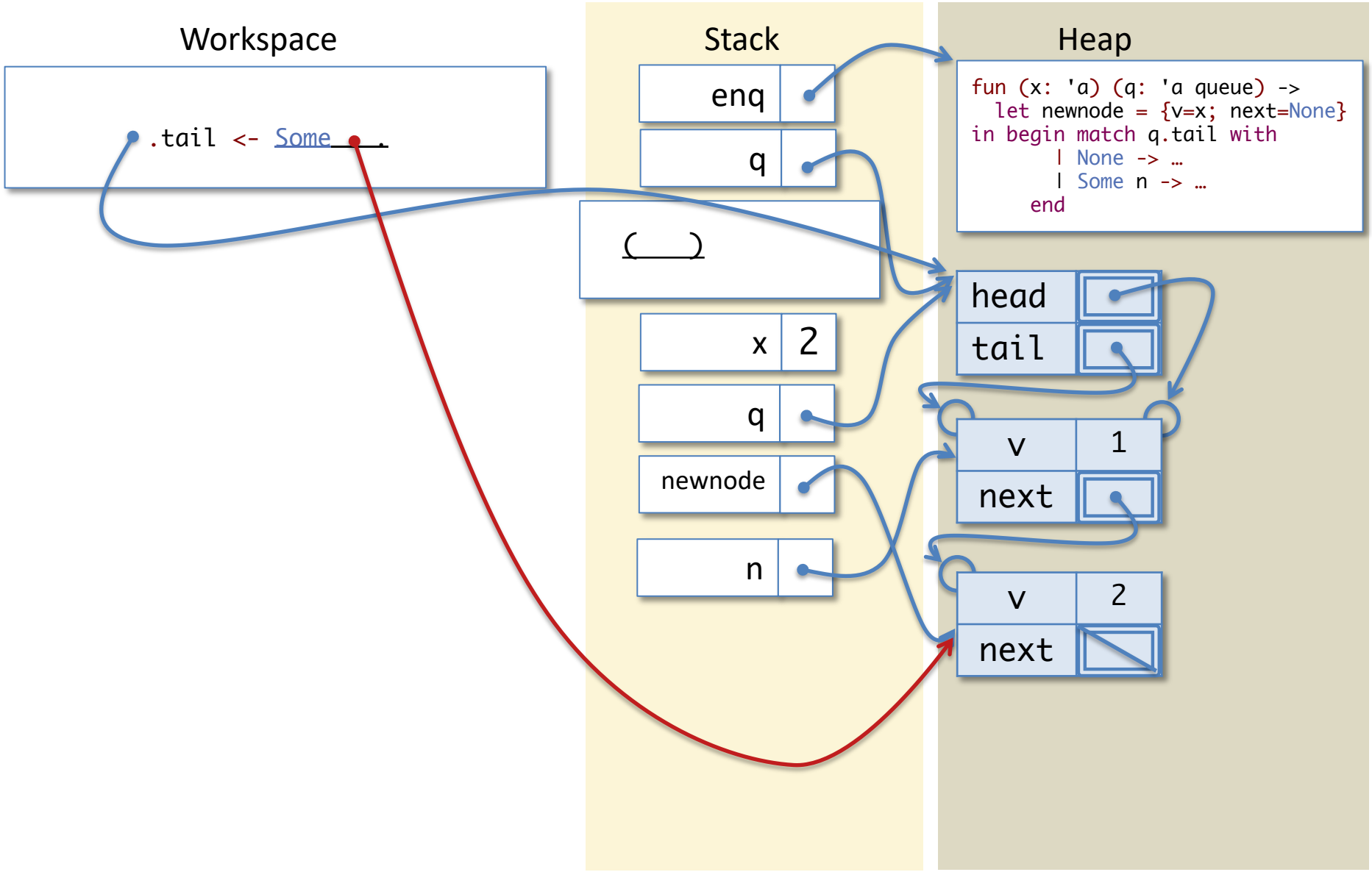
Calling Enq on a non-empty queue



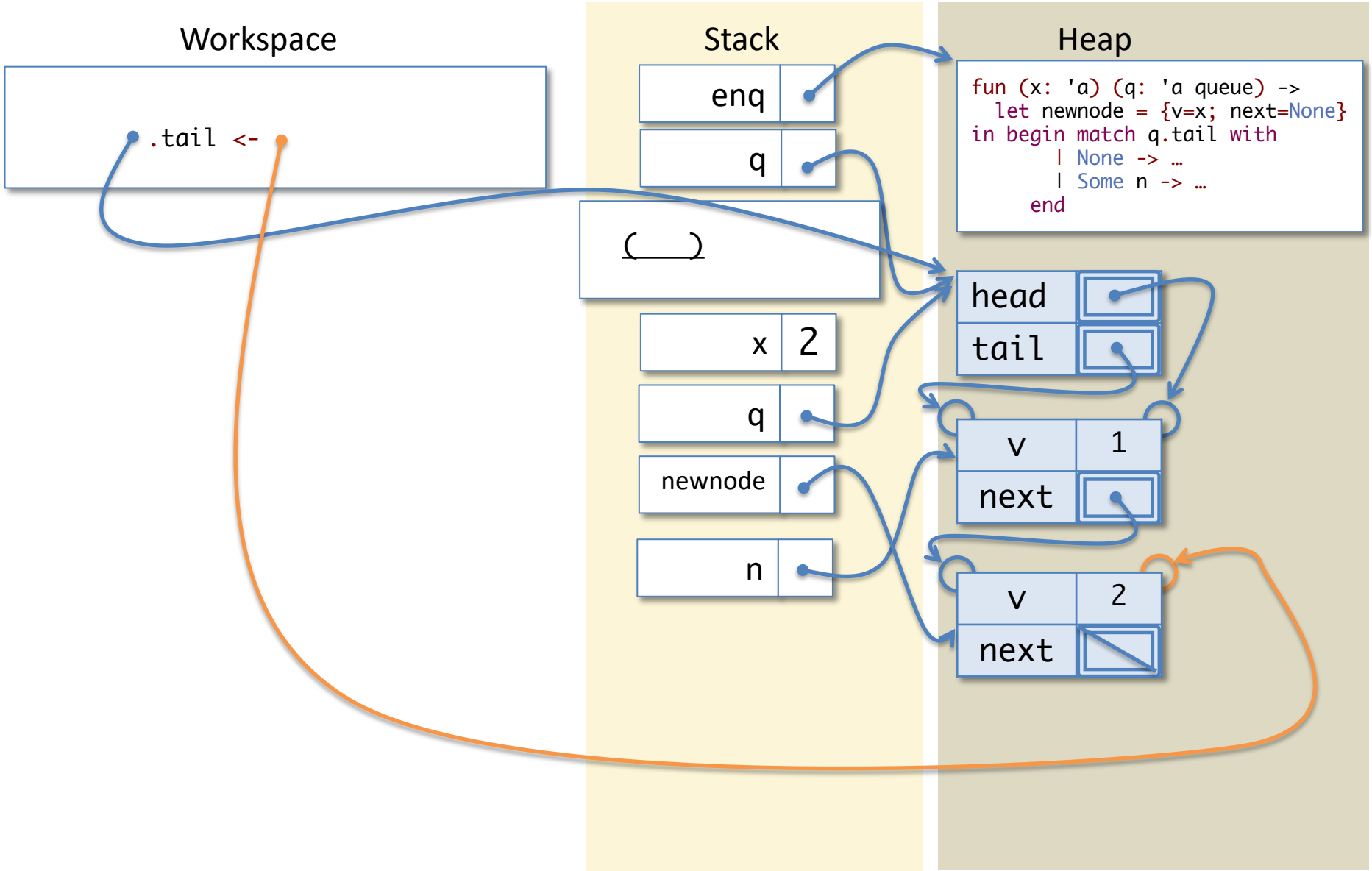
Calling Enq on a non-empty queue



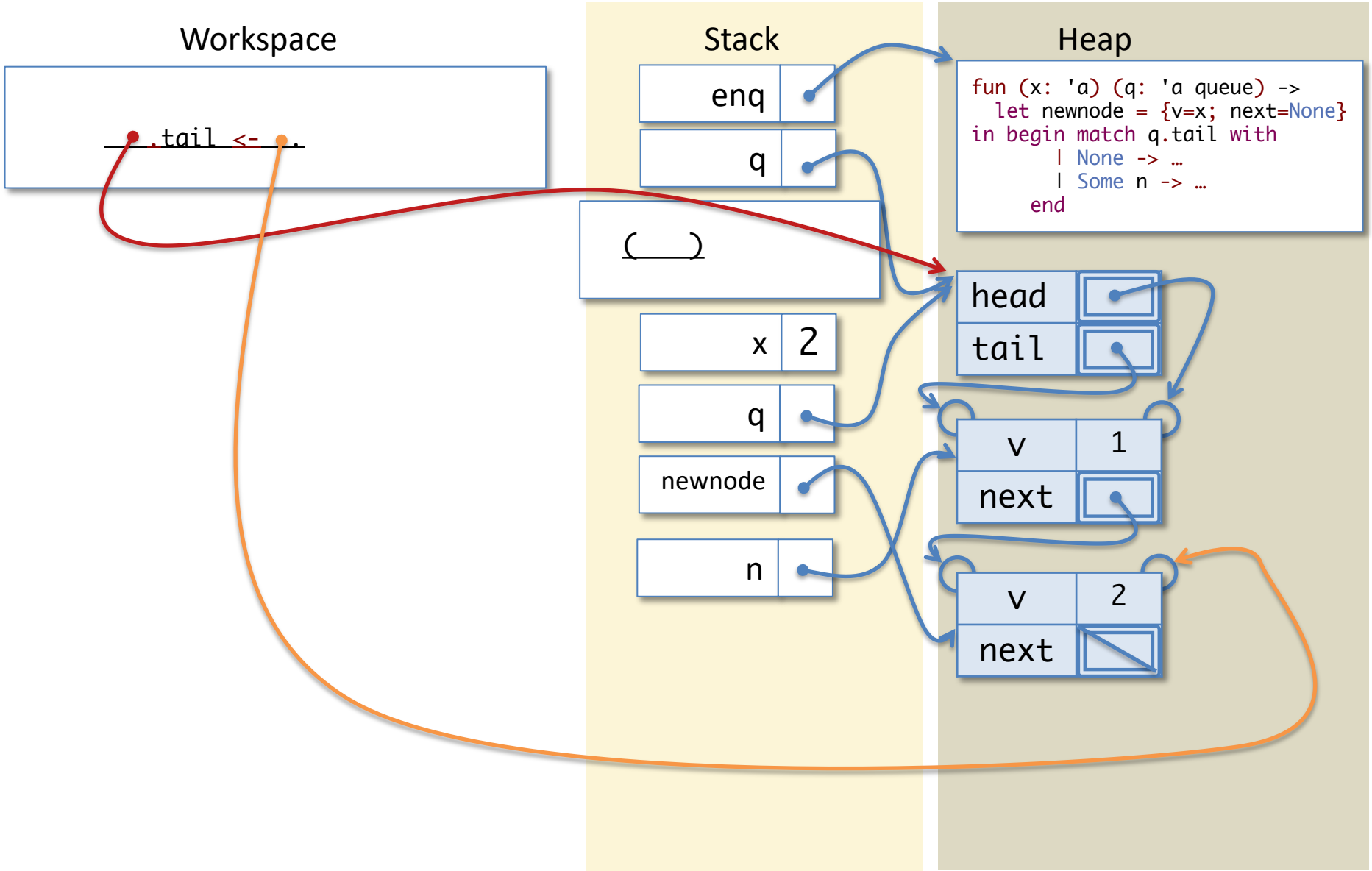
Calling Enq on a non-empty queue



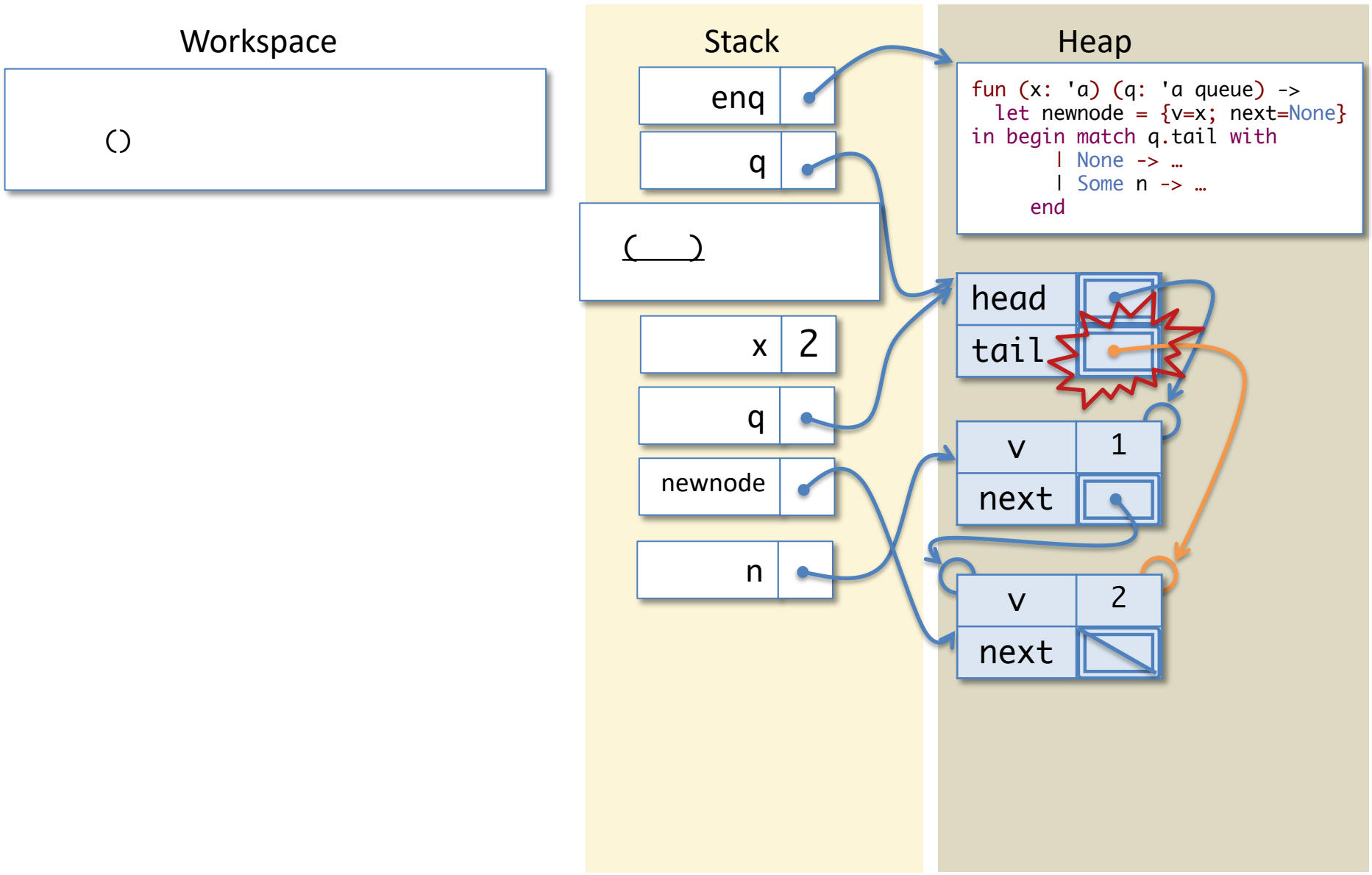
Calling Enq on a non-empty queue



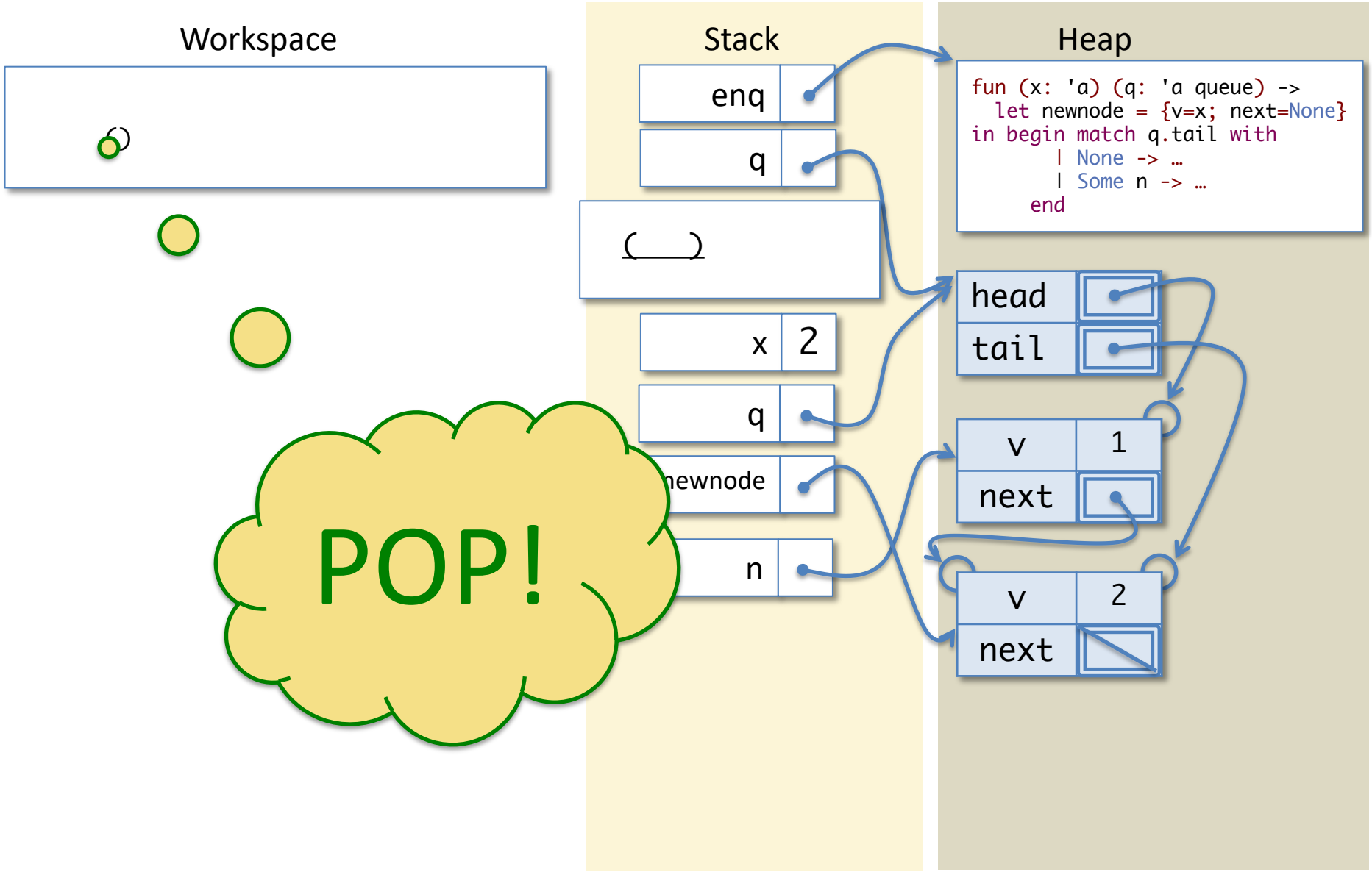
Calling Enq on a non-empty queue



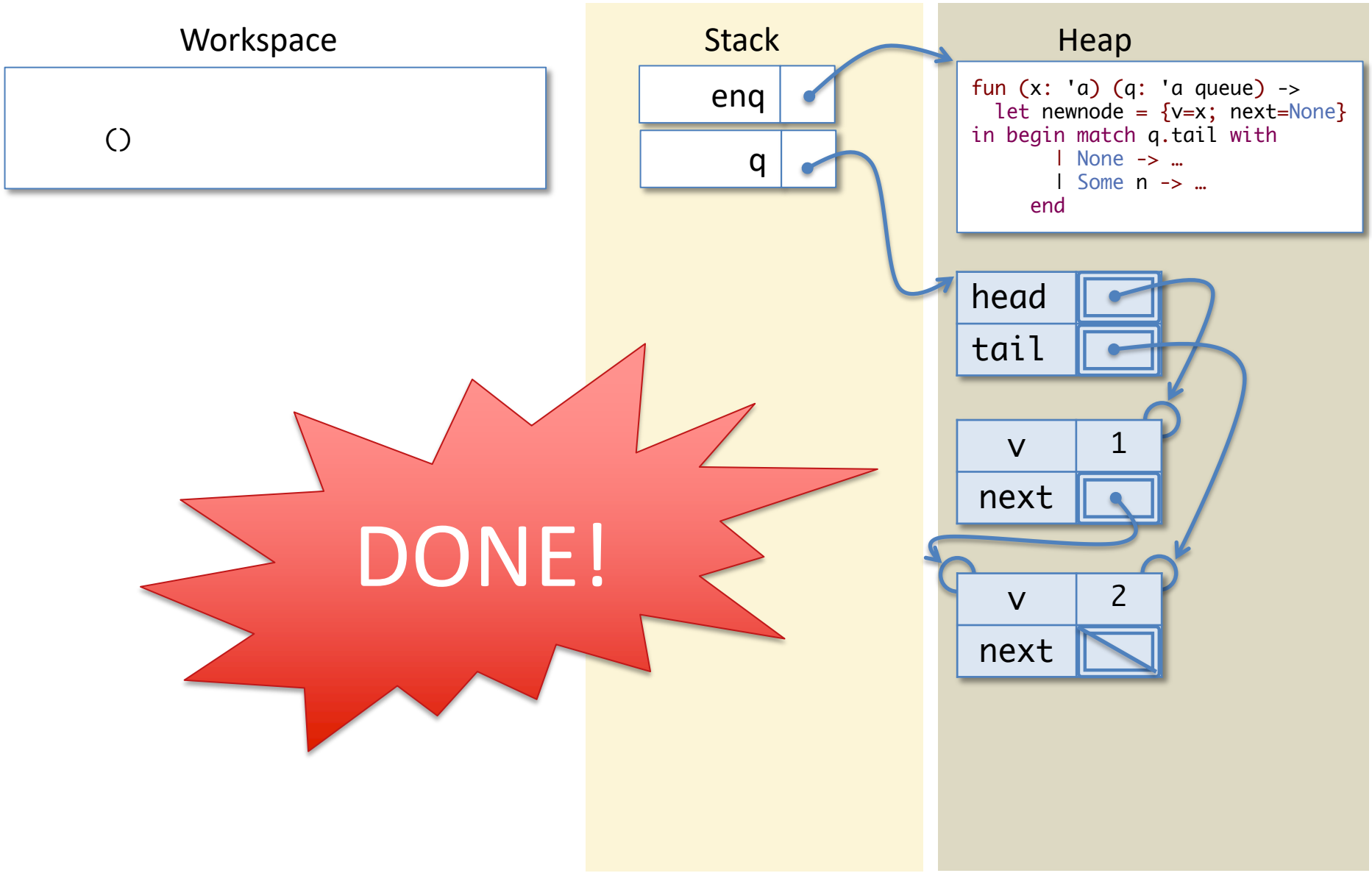
Calling Enq on a non-empty queue



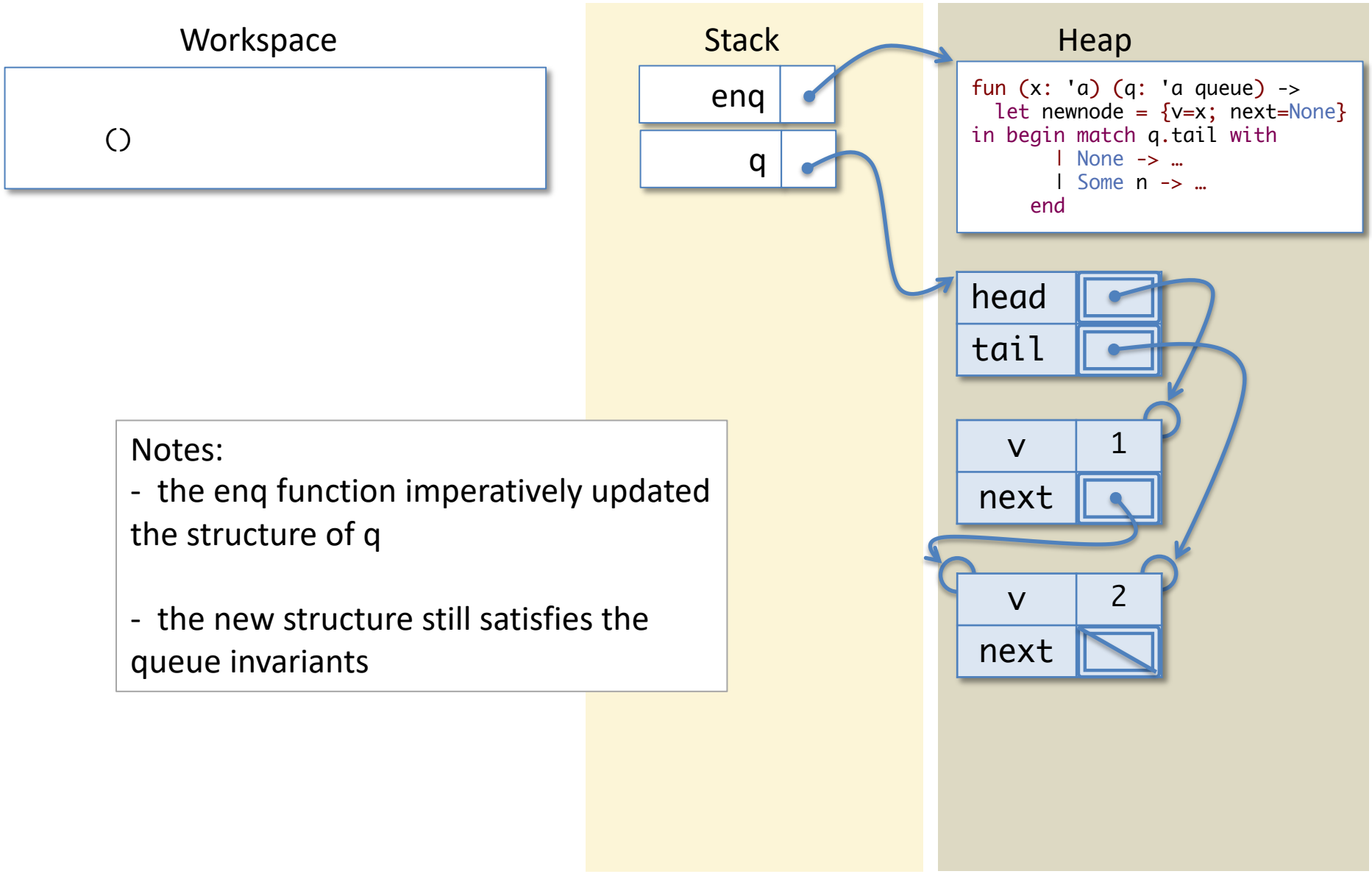
Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Calling Enq on a non-empty queue



Notes:

- the enq function imperatively updated the structure of q

- the new structure still satisfies the queue invariants