Programming Languages and Techniques (CIS120)

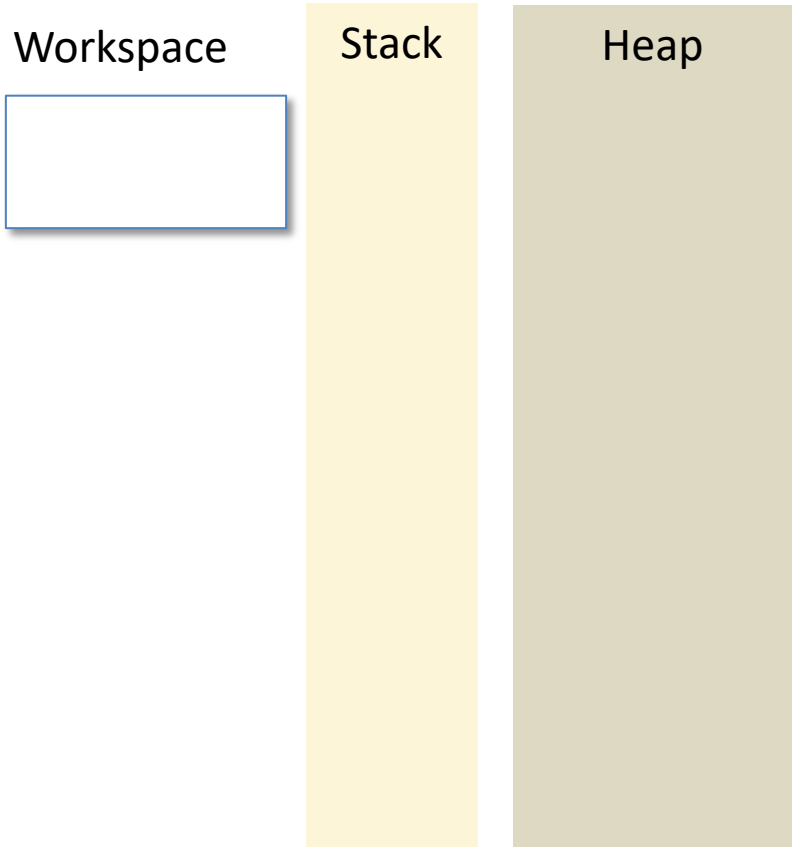
Lecture 14

Abstract Stack Machine, Reference Equality

Lecture notes: Chapter 15

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



Review: Values and References

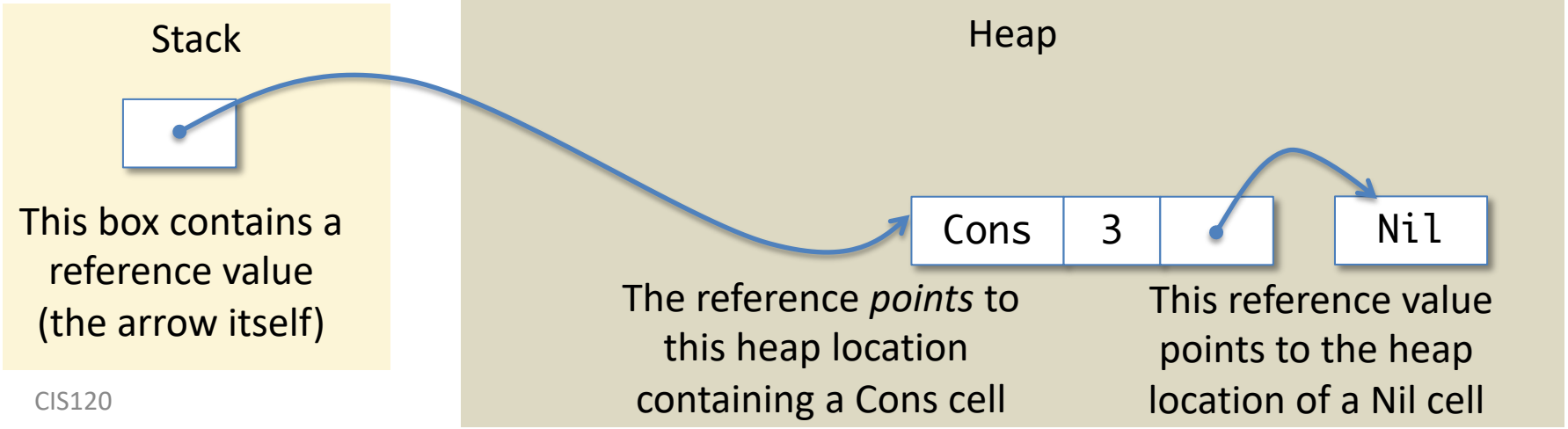
A *value* is either:

- 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:



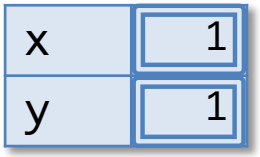
Mutable Records

- The reason for introducing the ASM model is to make heap locations and sharing *explicit*.
  - Now we can say what it means to mutate a heap value *in place*.

```
type point = {mutable x:int; mutable y:int}

let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int = (p2.x <- 17; p1.x)
```

- We draw a record in the heap like this:
  - The doubled outlines indicate that those cells are mutable
  - Everything else is immutable



A point record in the heap.

Aliasing Example

Workspace

```
let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```

Stack

Heap

Allocate a Record

Workspace

```
let p1 : point = {x=1; y=1;}
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```

Stack

Heap

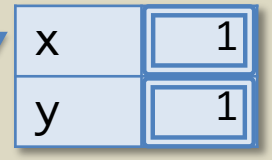
Allocate a Record

Workspace

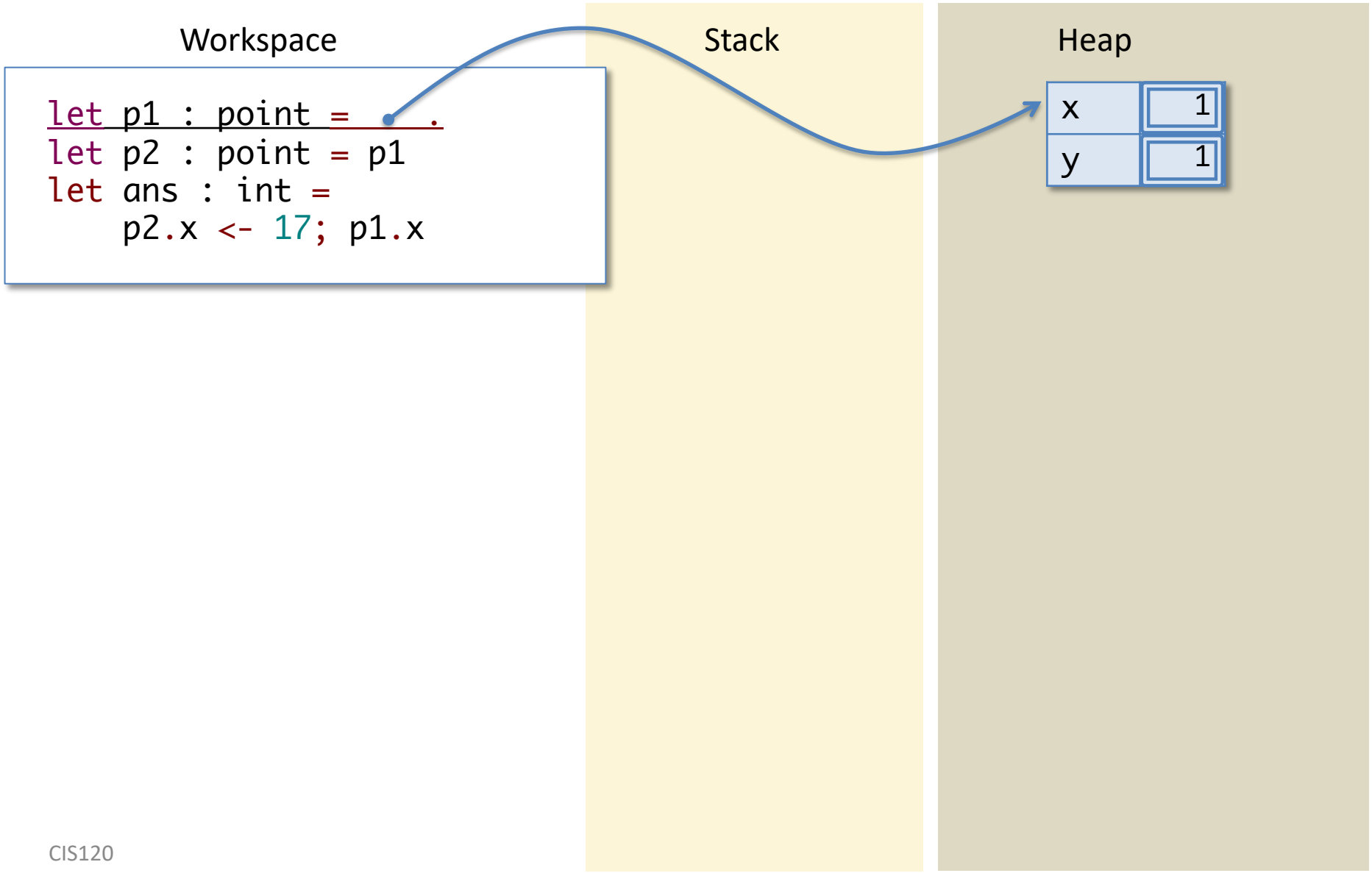
```
let p1 : point =
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```

Stack

Heap



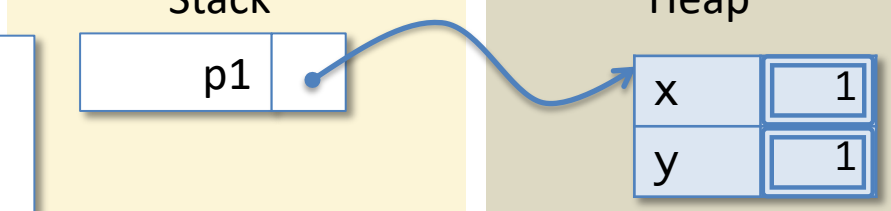
Let Expression



Push p1

Workspace

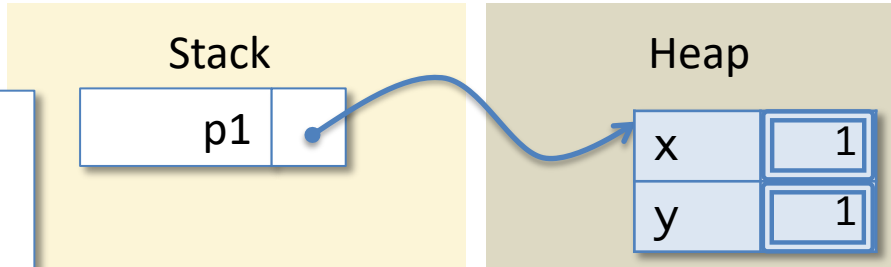
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



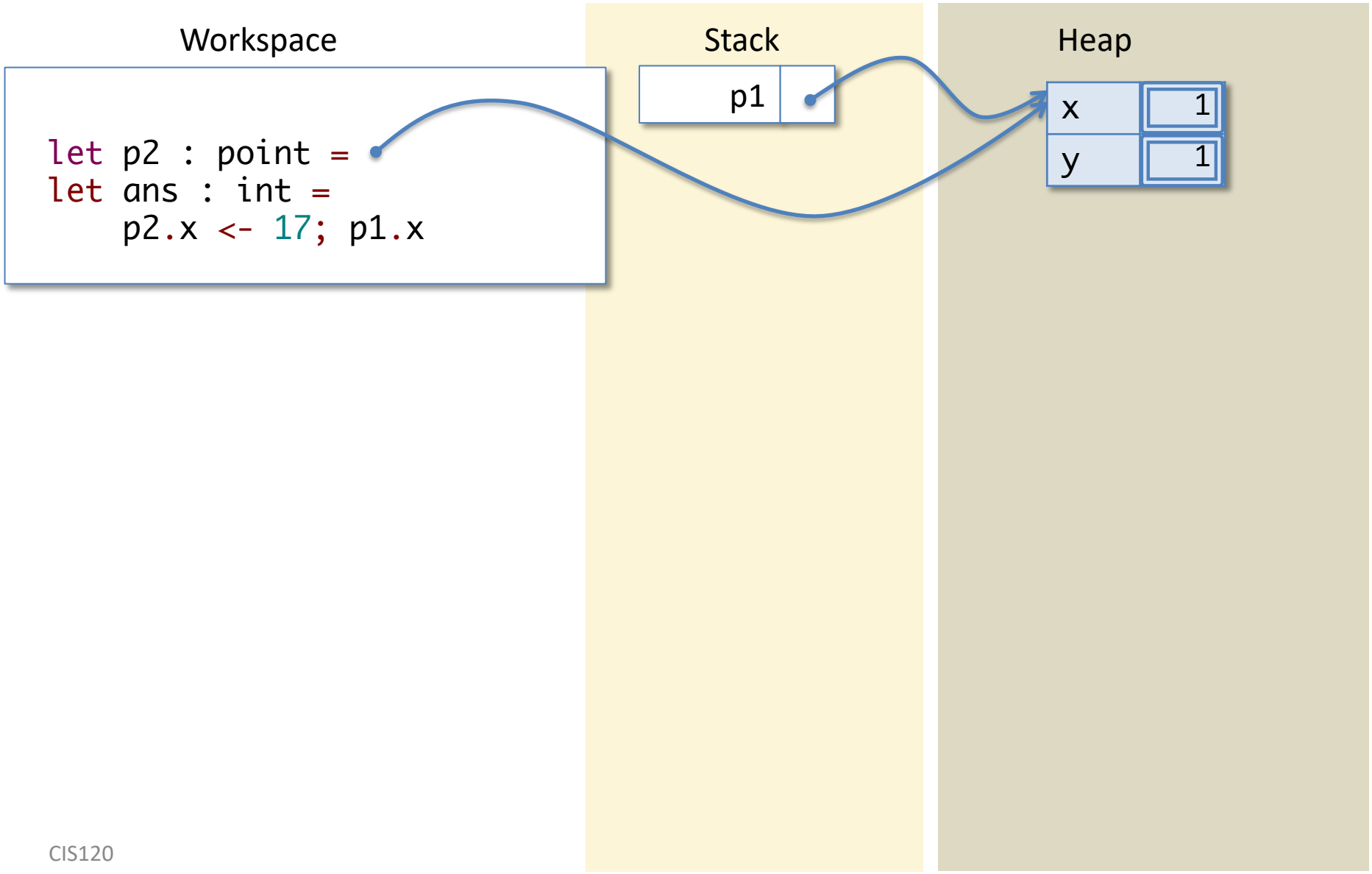
Look Up ‘p1’

Workspace

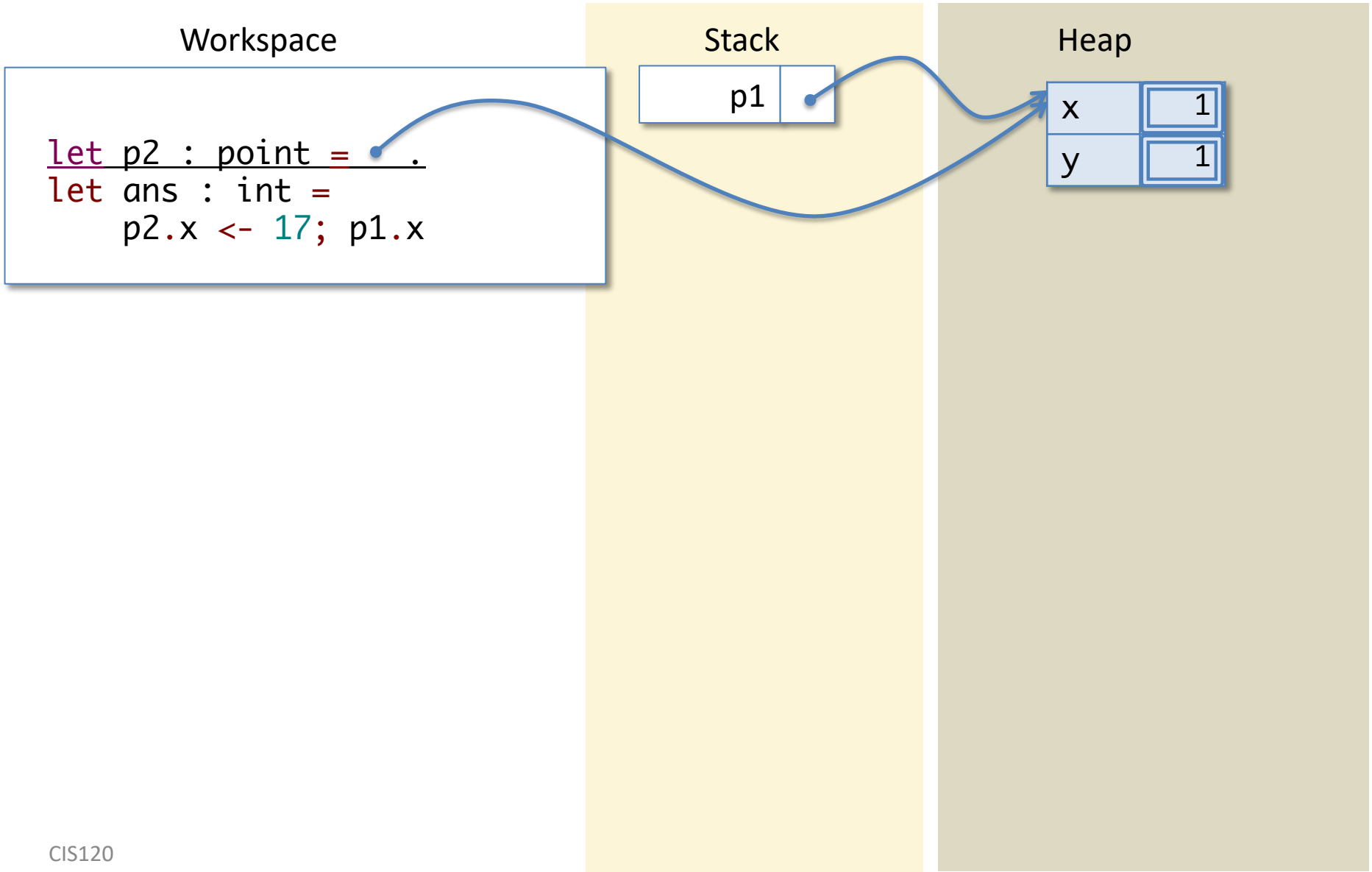
```
let p2 : point = p1
let ans : int =
    p2.x <- 17; p1.x
```



Look Up ‘p1’


Workspace
let p2 : point =
let ans : int =
p2.x <- 17; p1.x
Stack
p1
Heap
x
1
y
1

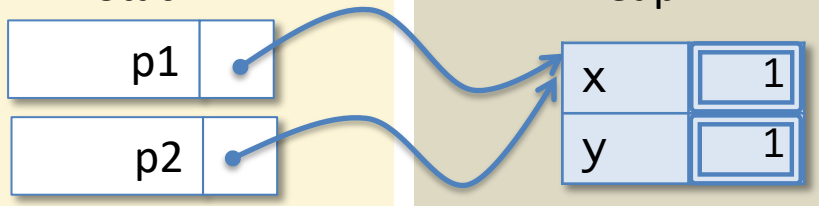
Let Expression



Push p2

Workspace

```
let ans : int =
    p2.x <- 17; p1.x
```

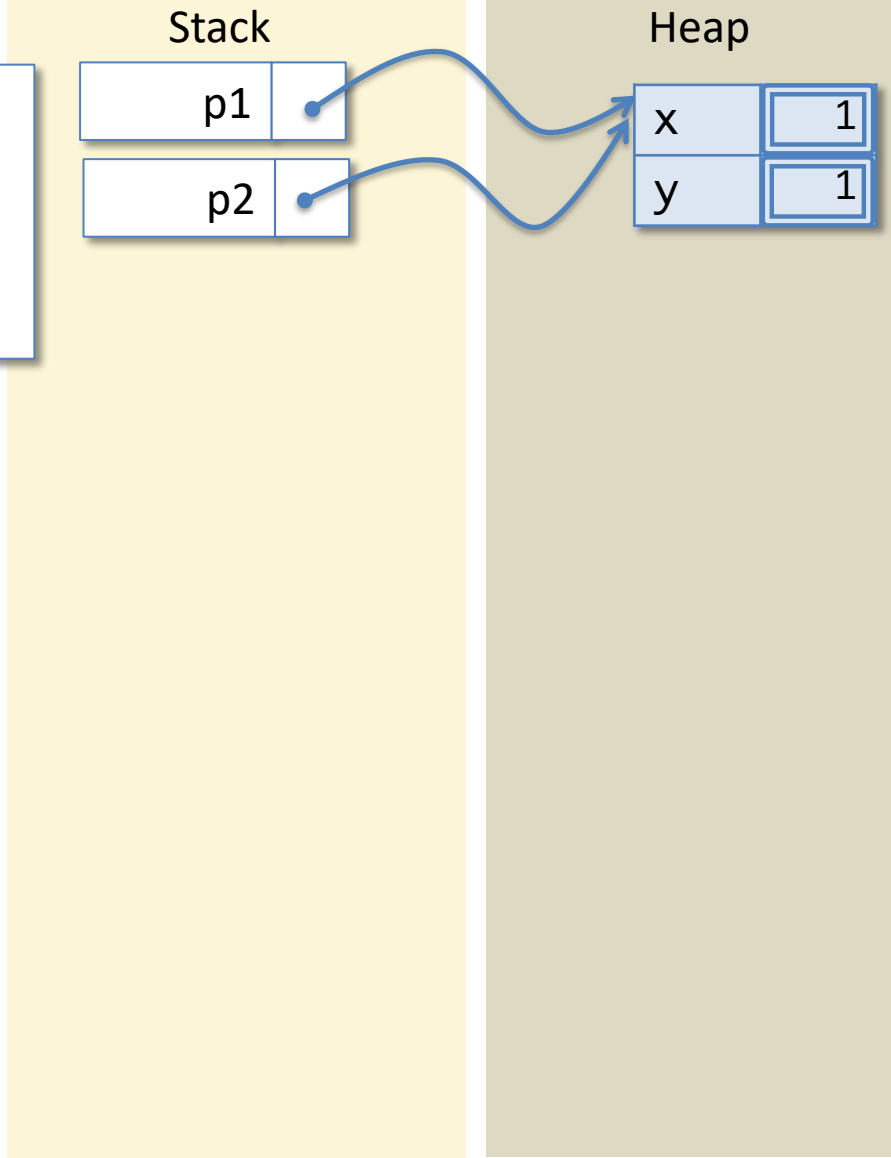


Note: p1 and p2 are references to the *same* heap record.
They are *aliases* – two different names for the *same thing*.

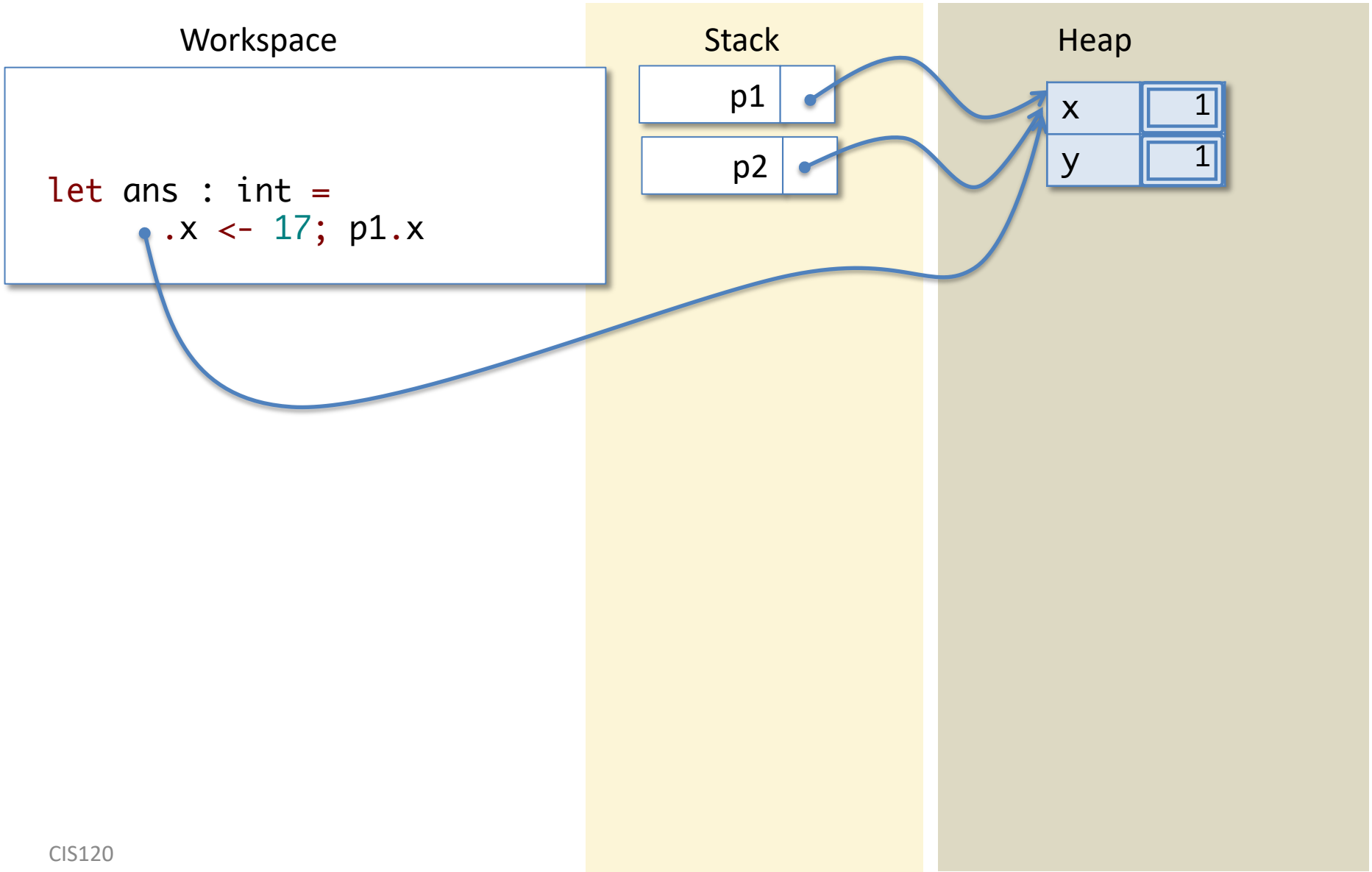
Look Up ‘p2’

Workspace

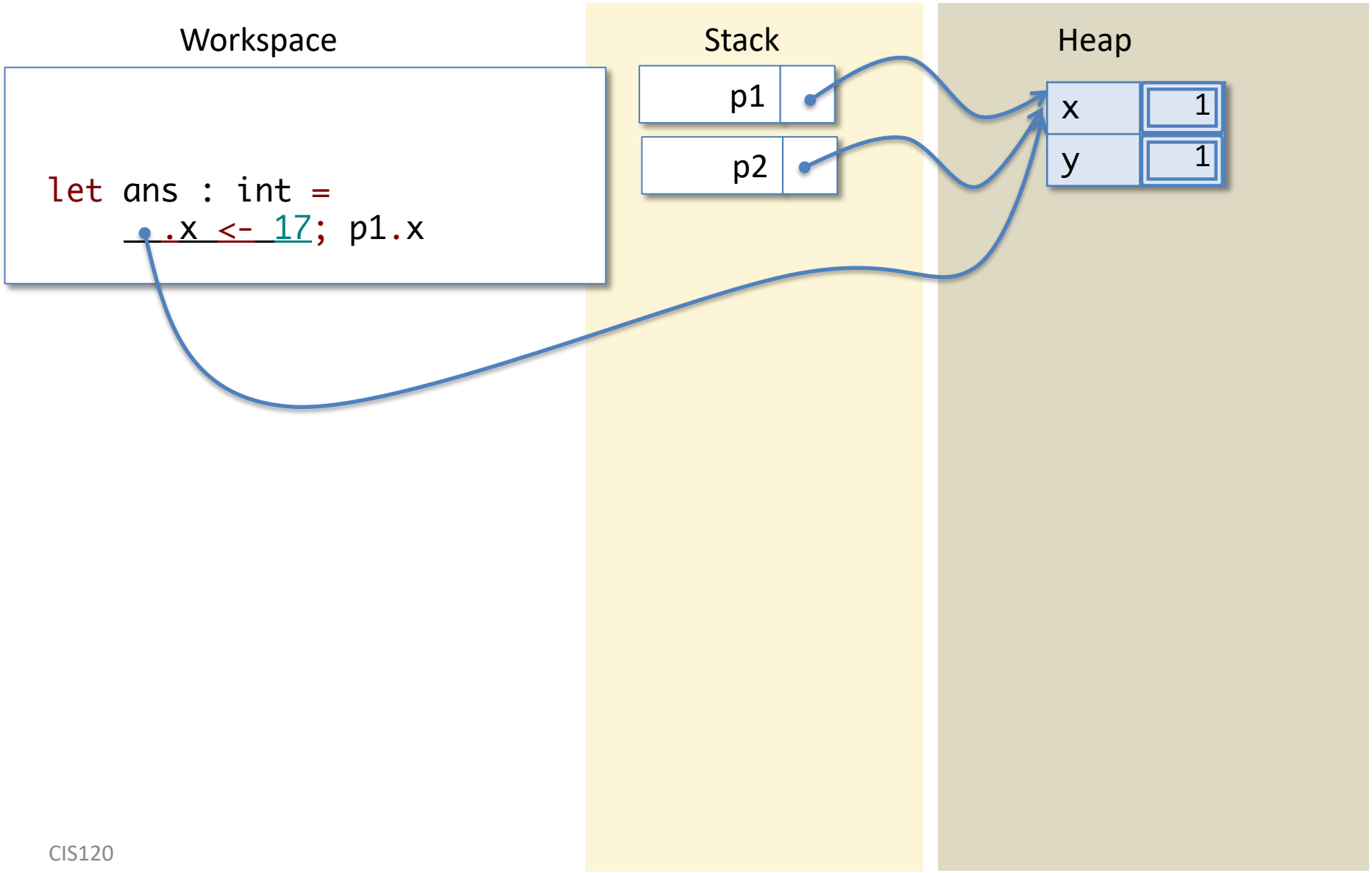
```
let ans : int =
    p2.x <- 17; p1.x
```



Look Up ‘p2’



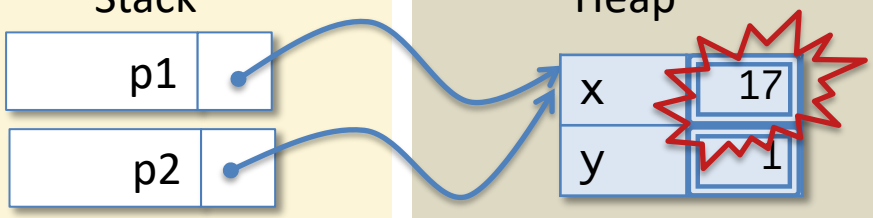
Assign to x field



Assign to x field

Workspace

```
let ans : int =
    (); p1.x
```

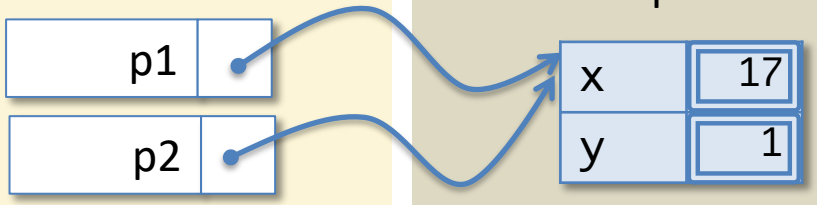


This is the step in which the 'imperative' update occurs. The mutable field x has been modified in place to contain the value 17.

Sequence ‘;’ Discards Unit

Workspace

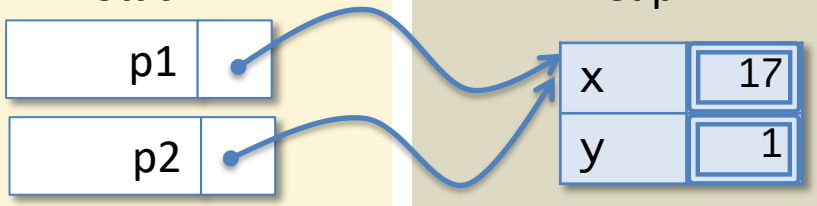
```
let ans : int =
    (); p1.x
```



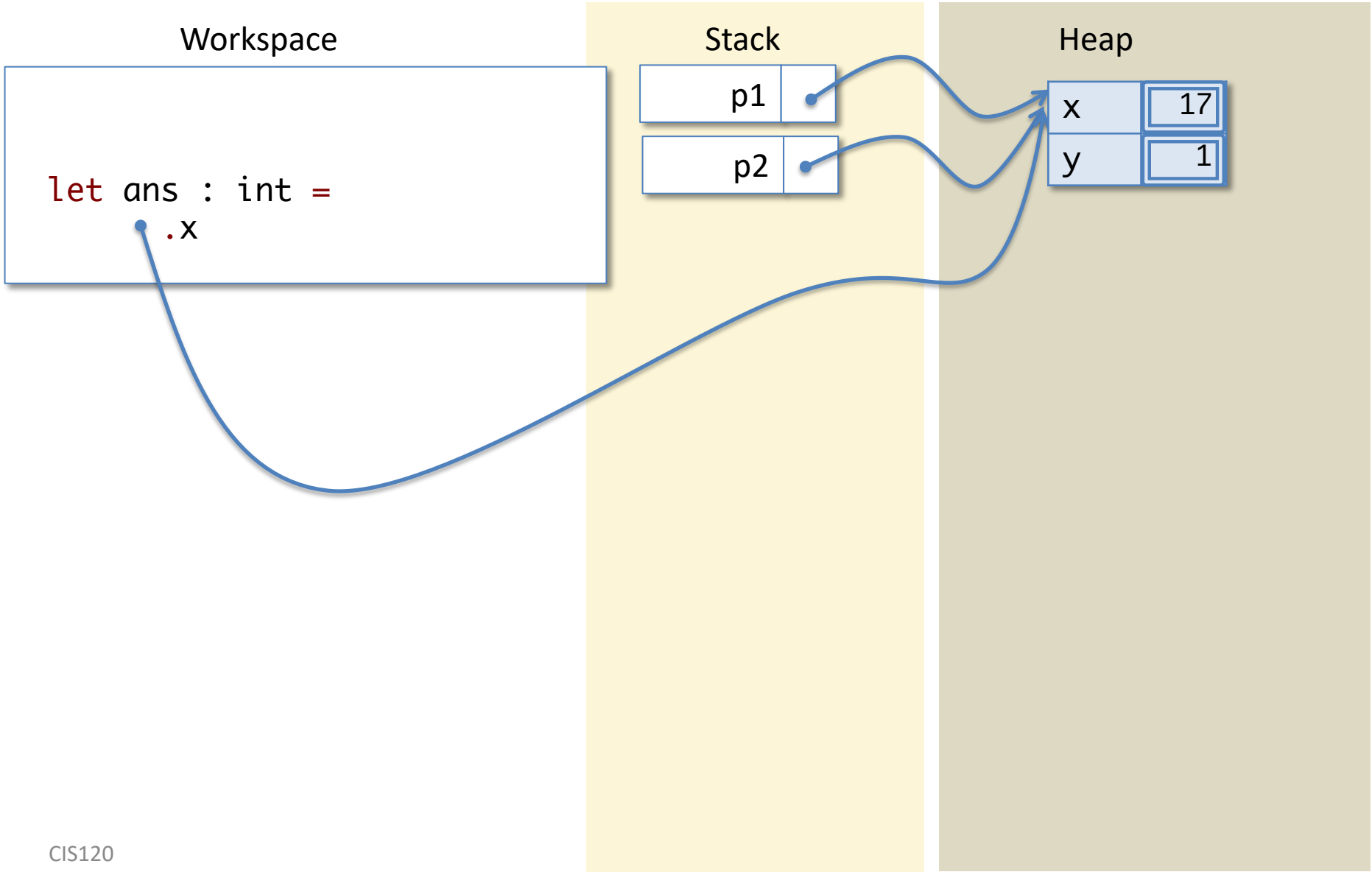
Look Up ‘p1’

Workspace

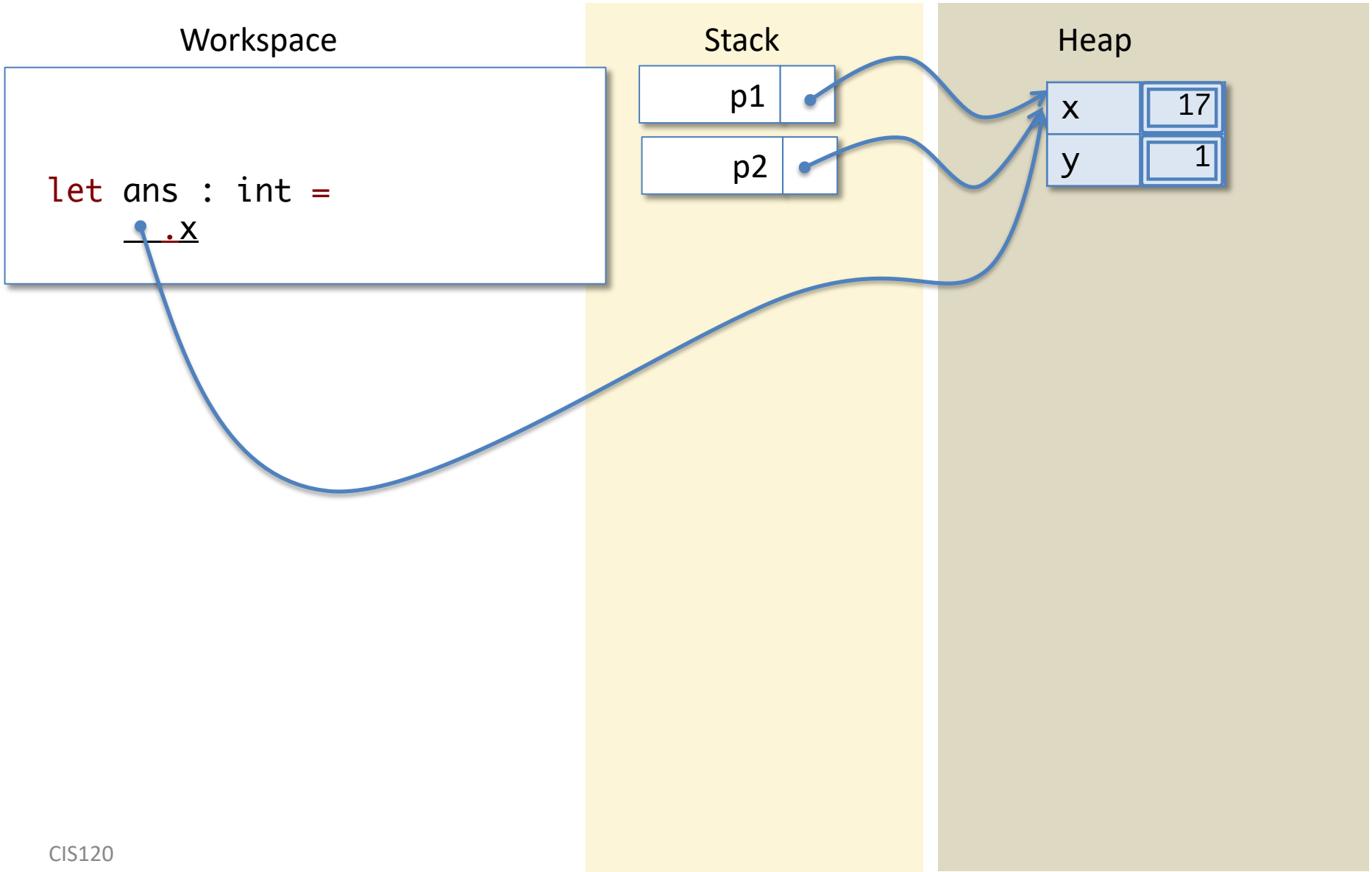
```
let ans : int =
    p1.x
```



Look Up ‘p1’



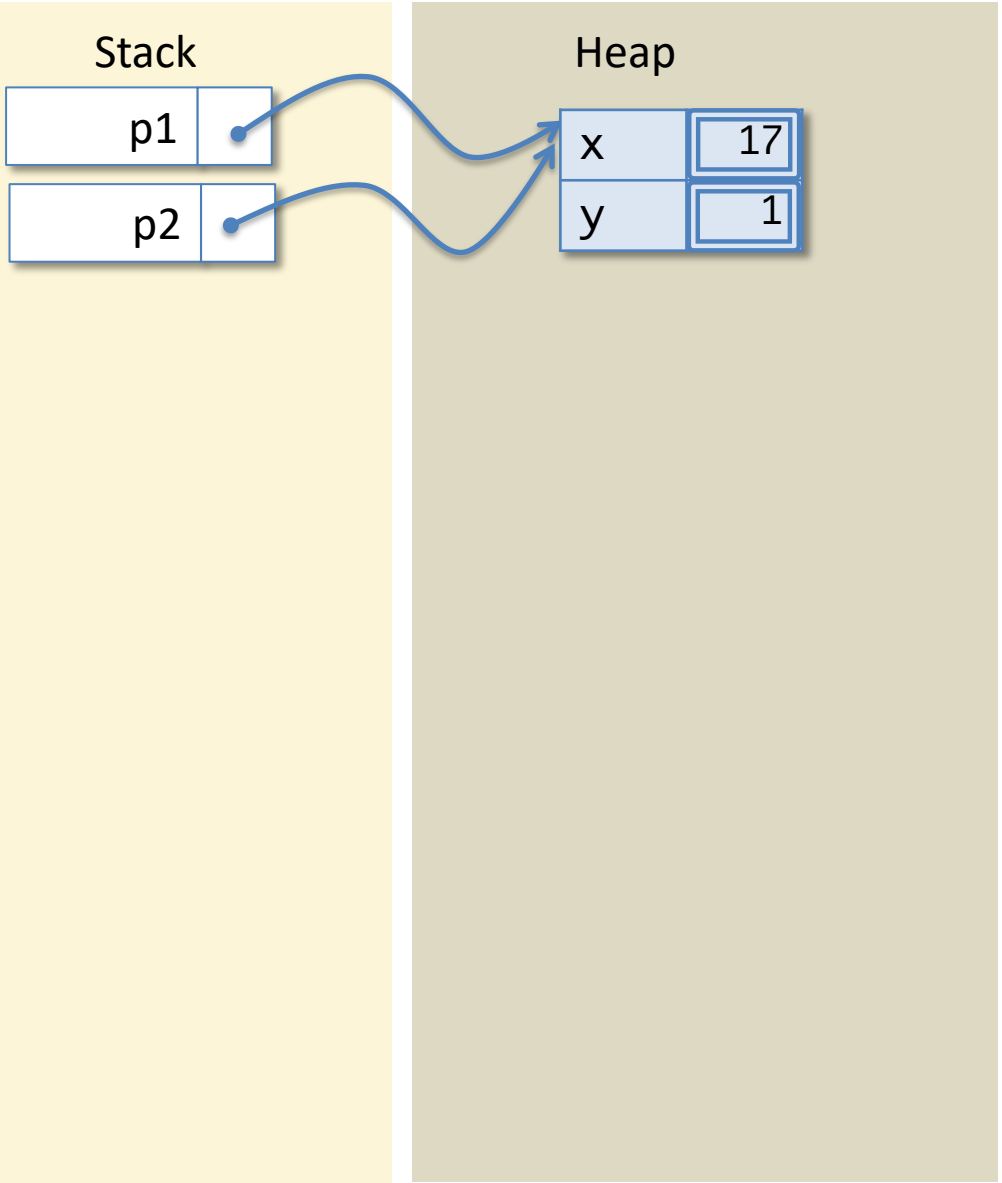
Project the ‘x’ field



Project the ‘x’ field

Workspace

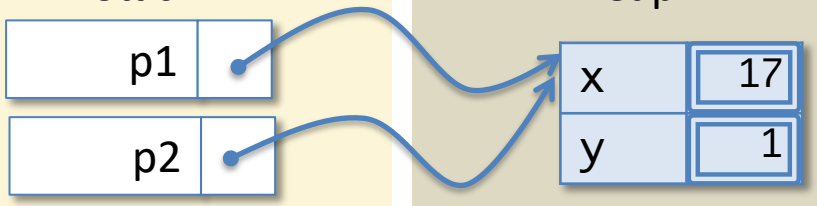
```
let ans : int =
    17
```



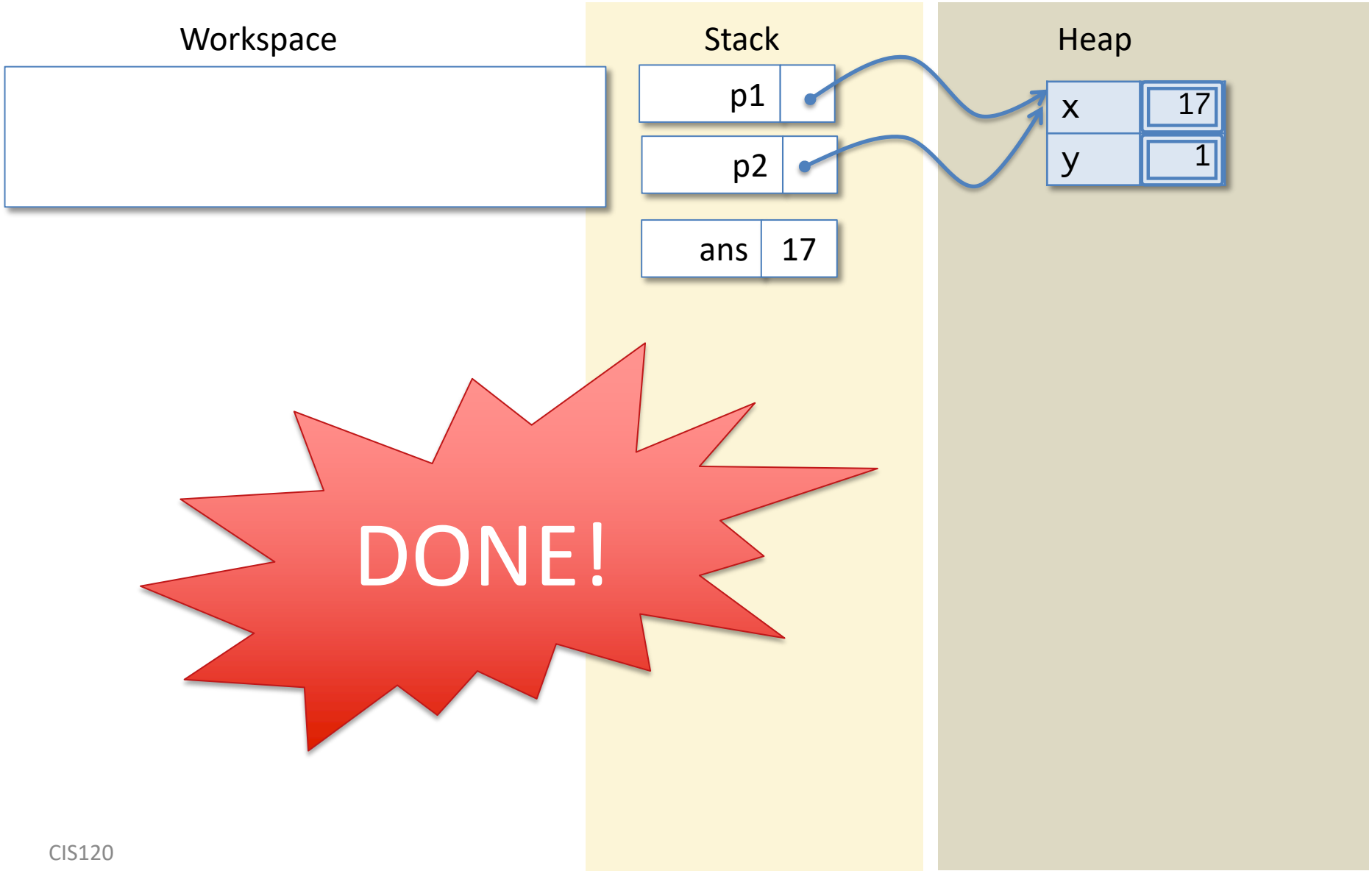
Let Expression

Workspace

```
let ans : int =
    17
```

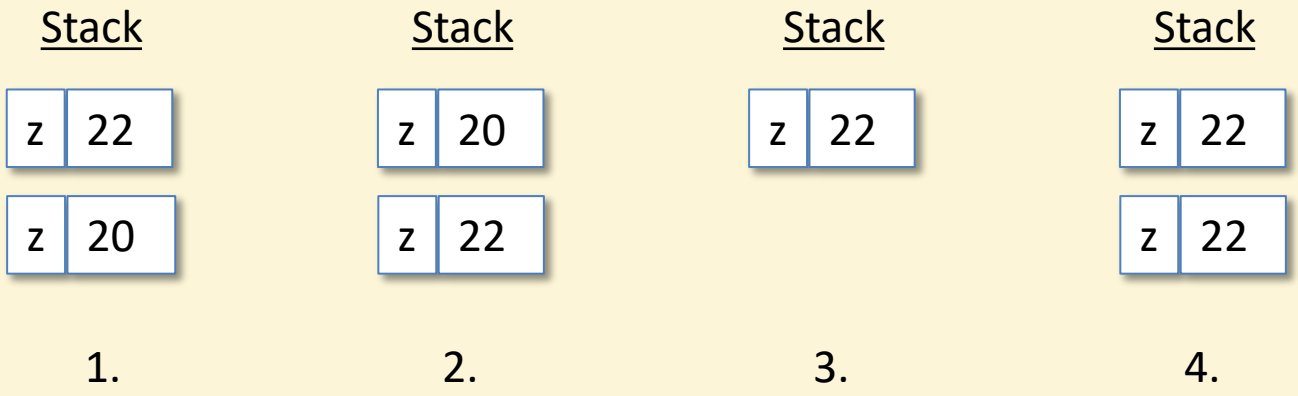


Push ans



What does the Stack look like after simplifying the following code on the workspace?

```
let z = 20 in
let z = 2 + z in
  z
```



ANSWER: 2

References and Equality

= vs. ==

Reference Equality

- Suppose we have two counters. Do they share state?

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

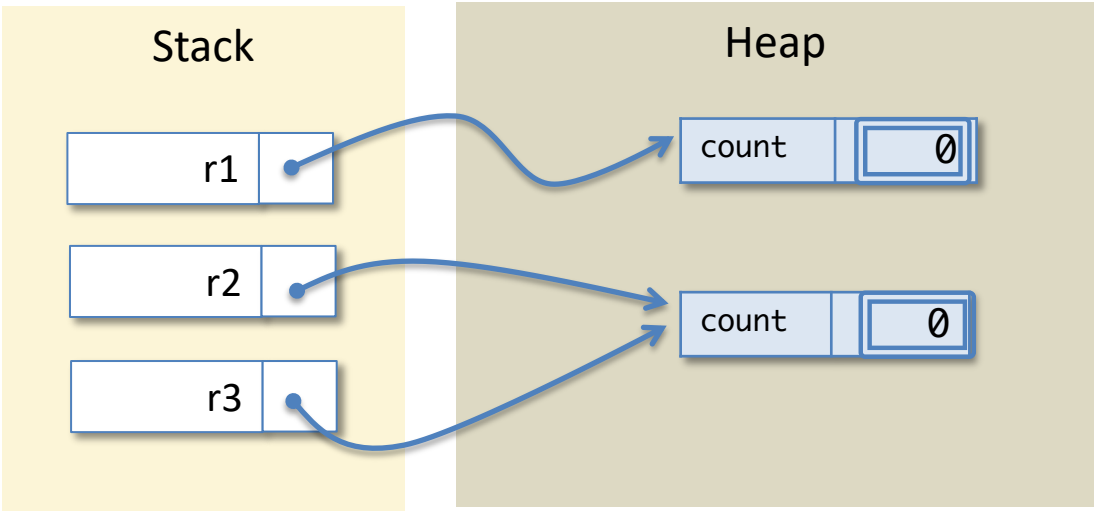
  - We could increment one and see whether the other's value changes.
  - But we could also just test whether the references are aliases.

- Ocaml uses '==' to mean *reference* equality:
  - two reference values are '==' if they point to the same object 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 are never structurally equivalent to anything
  - 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
  - equates strictly fewer things than structural equality
  - appropriate for comparing *mutable* datatypes

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 = 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

Ah… Refs!

OCaml provides syntax for working with updatable *references*:

```
type 'a ref = {mutable contents:'a}
```

`ref e`	means	`{contents = e}`	has type `t ref` when `(e : t)`
`e1 := e2`	means	`(e1).contents <- e2`	has type `unit` when `(e1 : t ref)` and `(e2 : t)`
`!e`	means	`(e).contents`	has type `t` when `(e : t ref)`

"is defined to be"
(not Ocaml syntax)

OCaml "syntactic sugar"

equivalent expressions

type constraints

Comparison To Java (or C, C++, …)

Java

```
int f() {
  int x = 3;
  x = x + 1;
  return x;
}
```

- x has type int
- meaning on left of = different than on right
- *implicit* dereference

OCaml

```
let f () : int =
  let x = ref 3 in
  x := !x + 1;
  !x
```

- x has type (int ref)
- use := for update
- *explicit* dereference

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

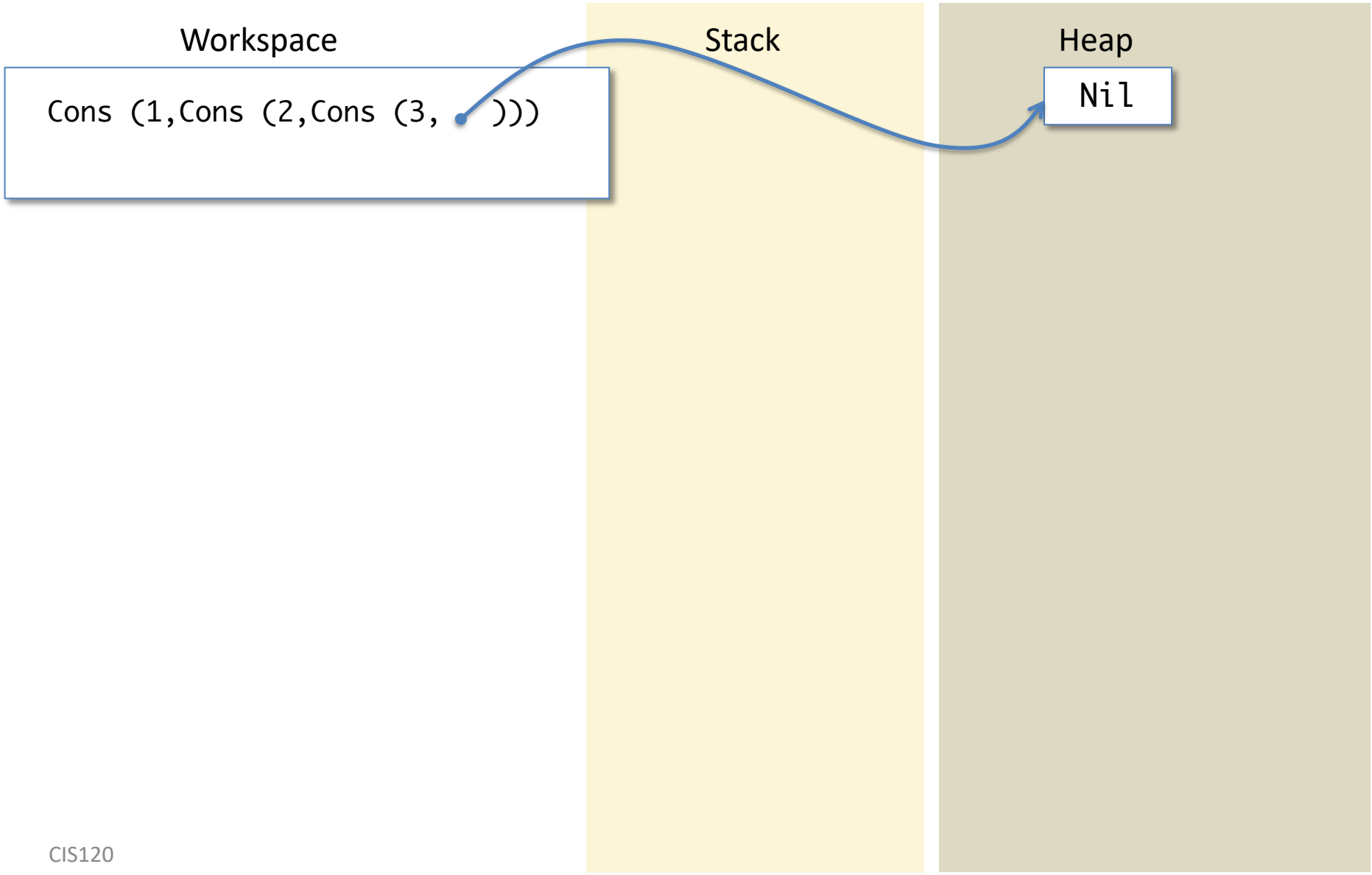
Workspace

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

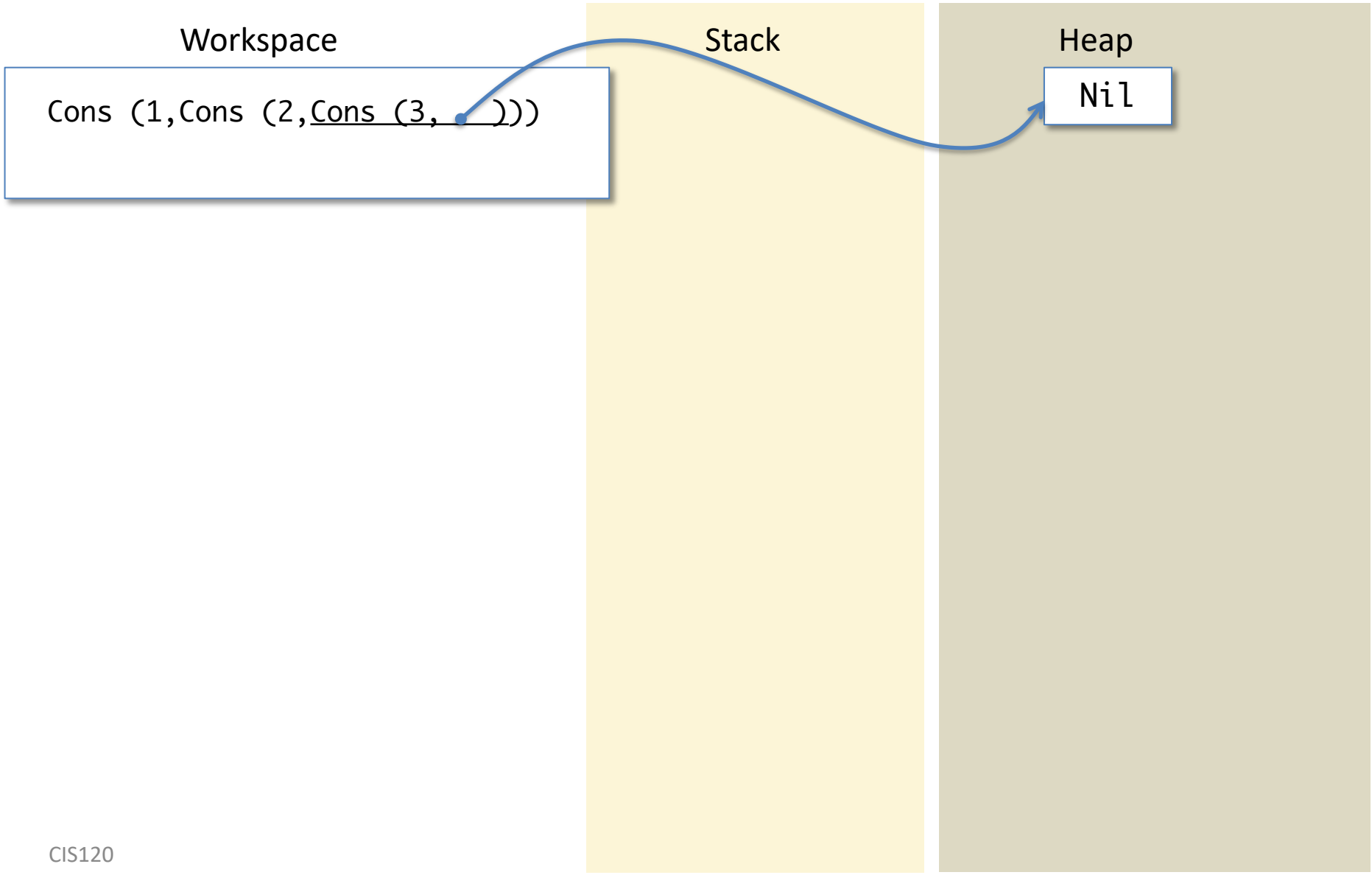
Stack

Heap

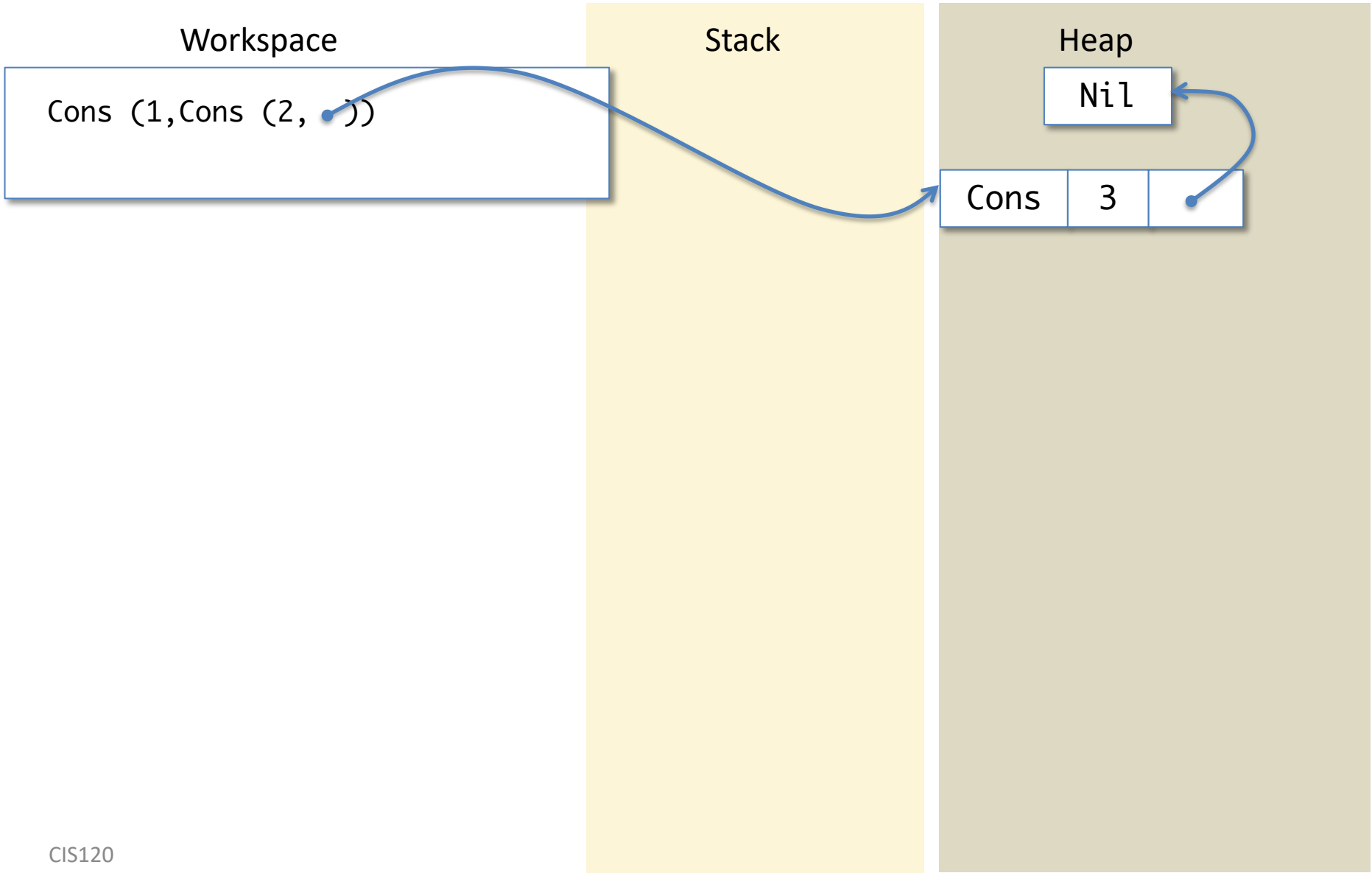
Simplification



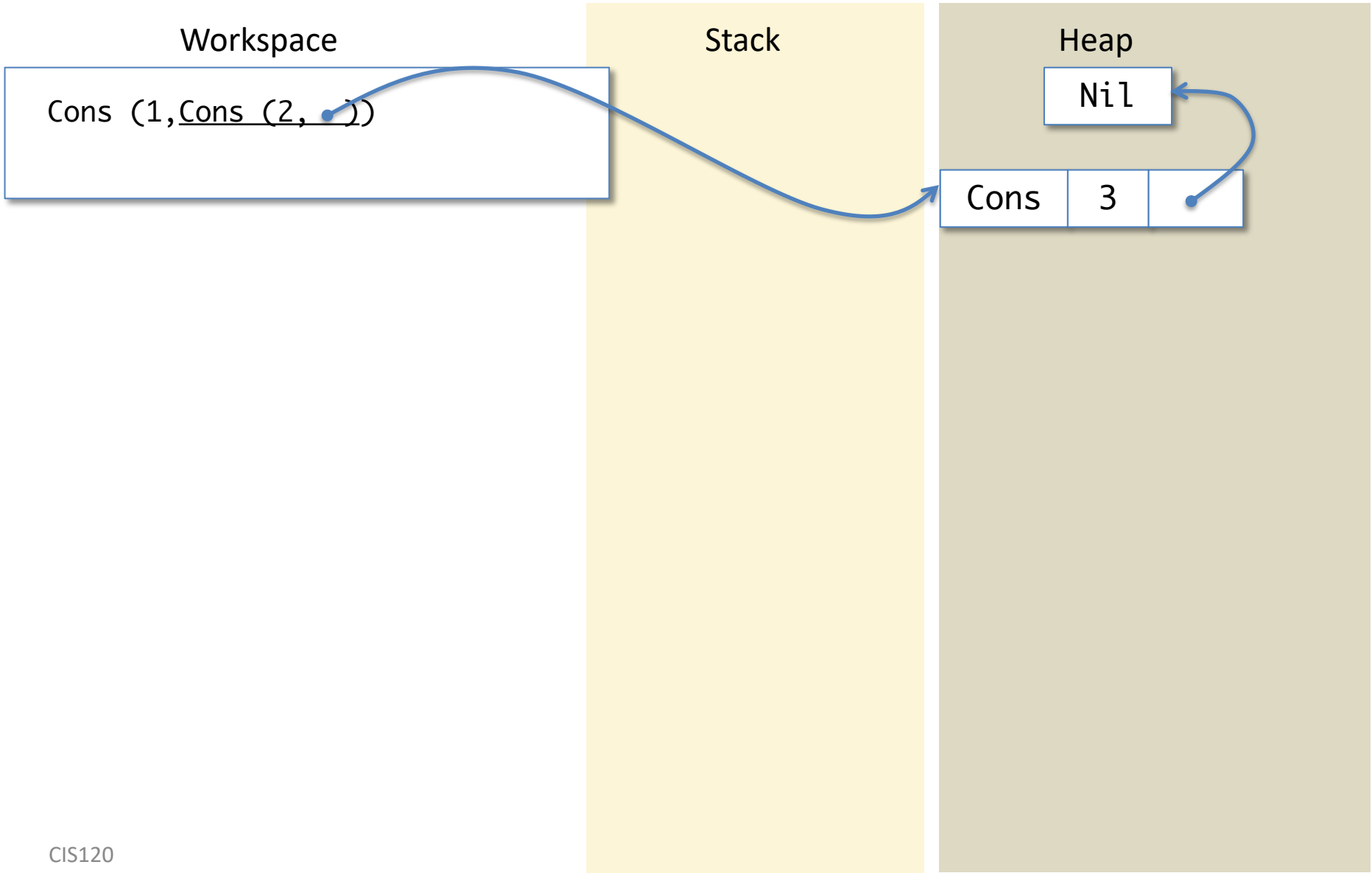
Simplification



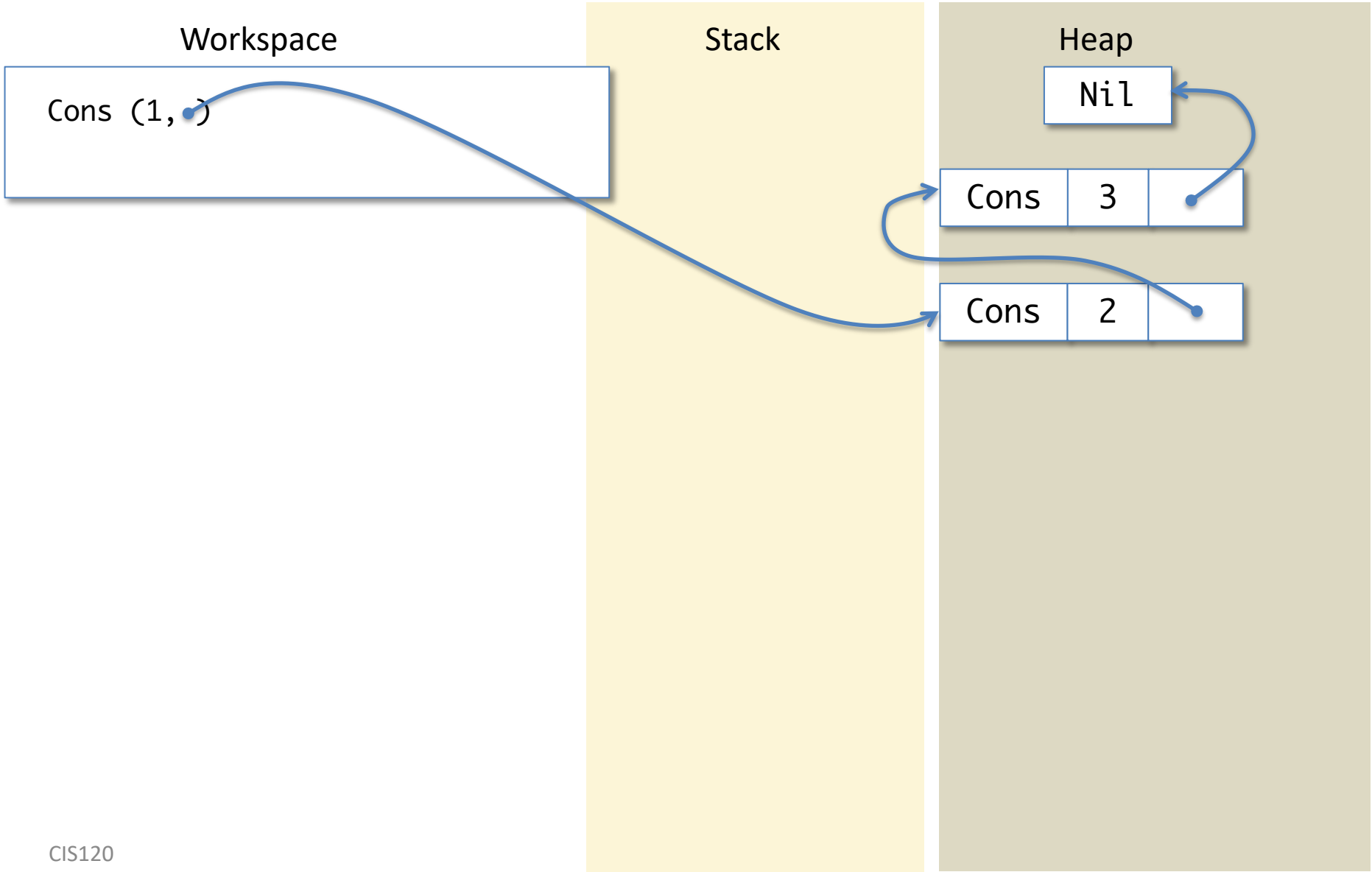
Simplification



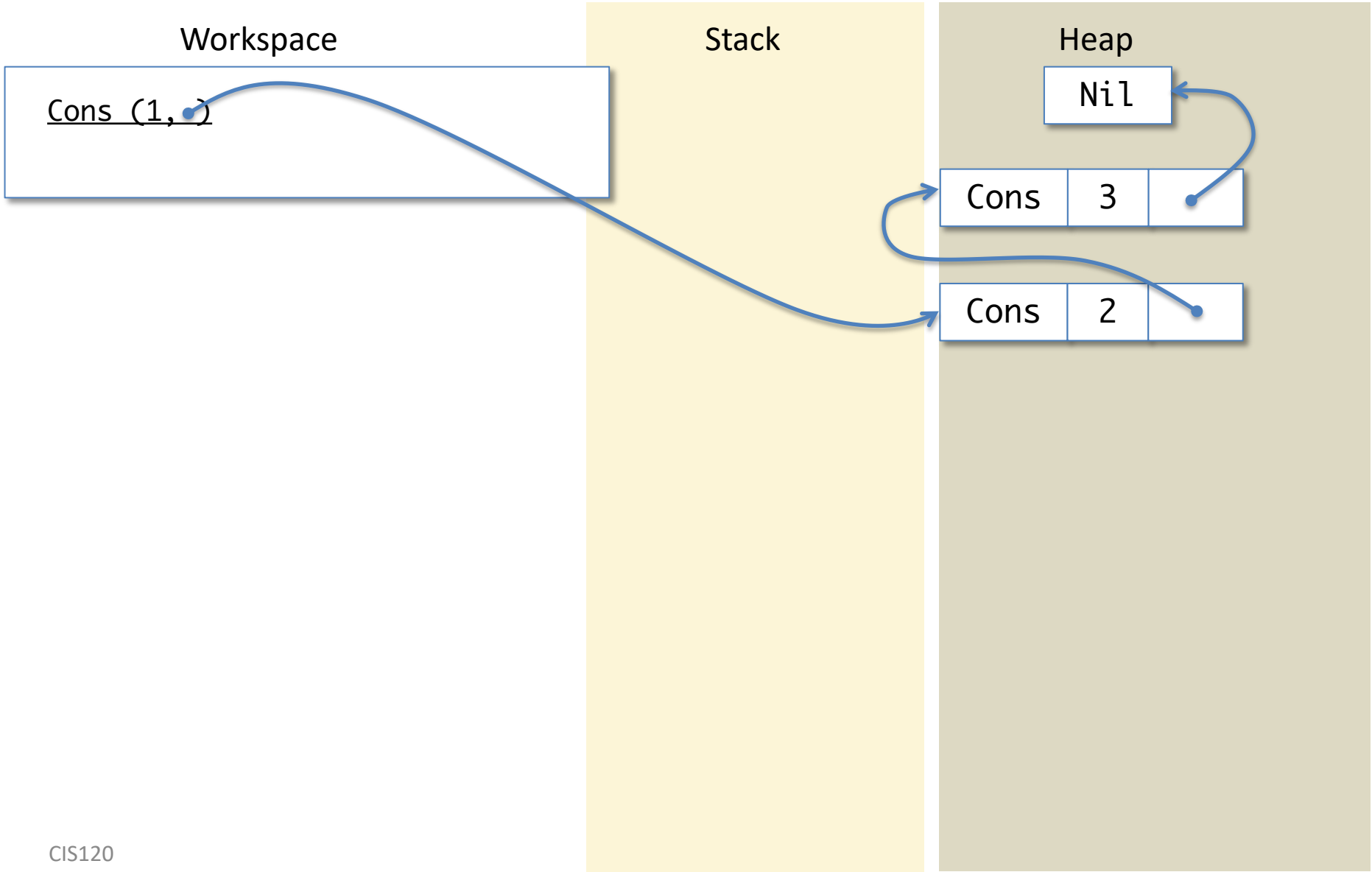
Simplification



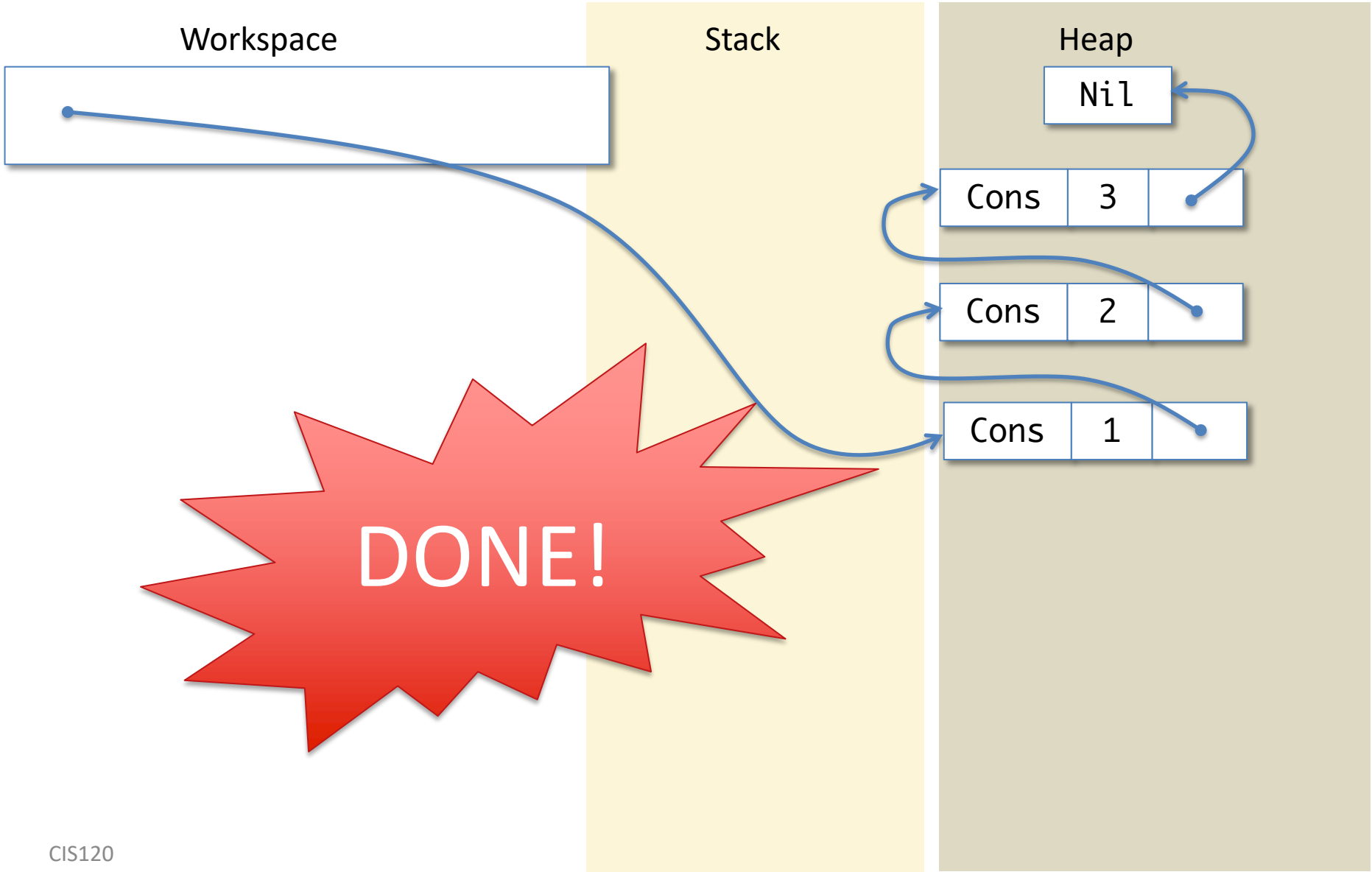
Simplification



Simplification



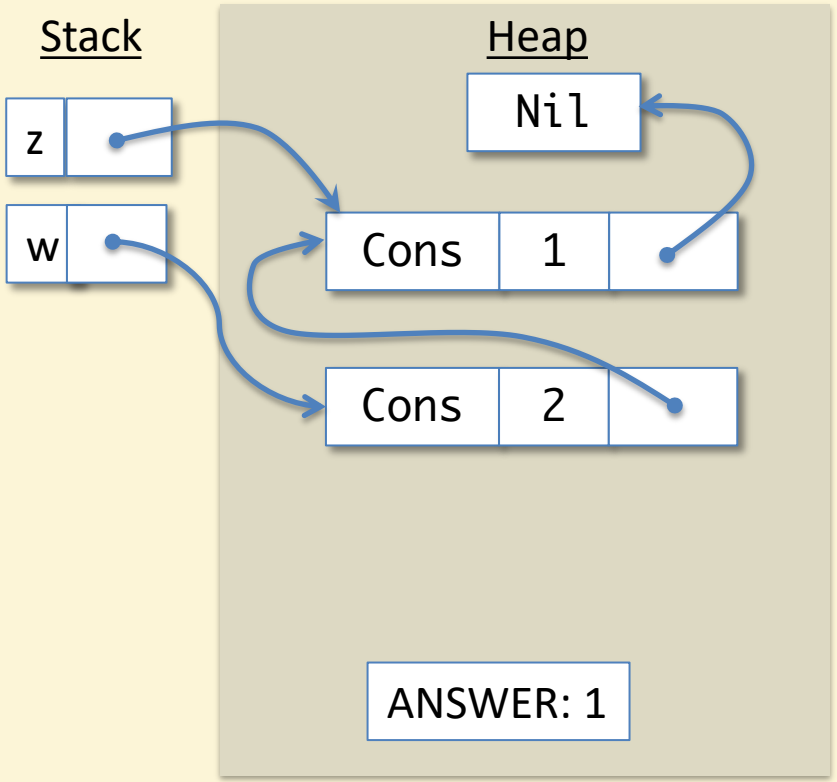
Simplification



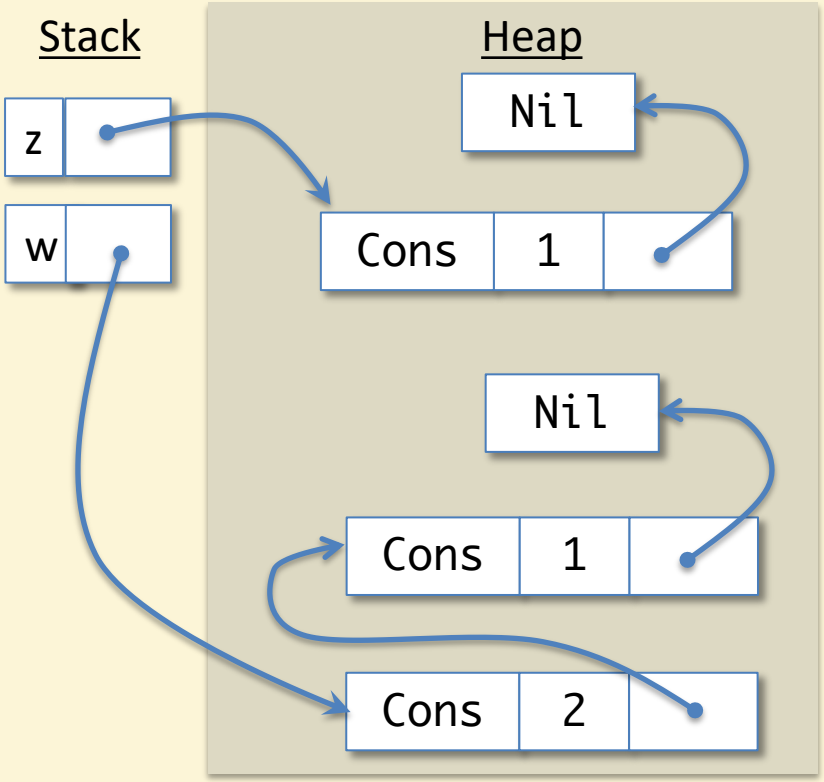
What do the Stack and Heap look like after simplifying the following code on the workspace?

```
let z = Cons (1, Nil) in
let w = Cons (2, z) in
    w
```

1.



2.



ASM: functions

Tracking the space usage of function calls

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

First step: replace declaration of add1 with more primitive version

Function Simplification

Workspace

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

Stack

Heap

Function Simplification

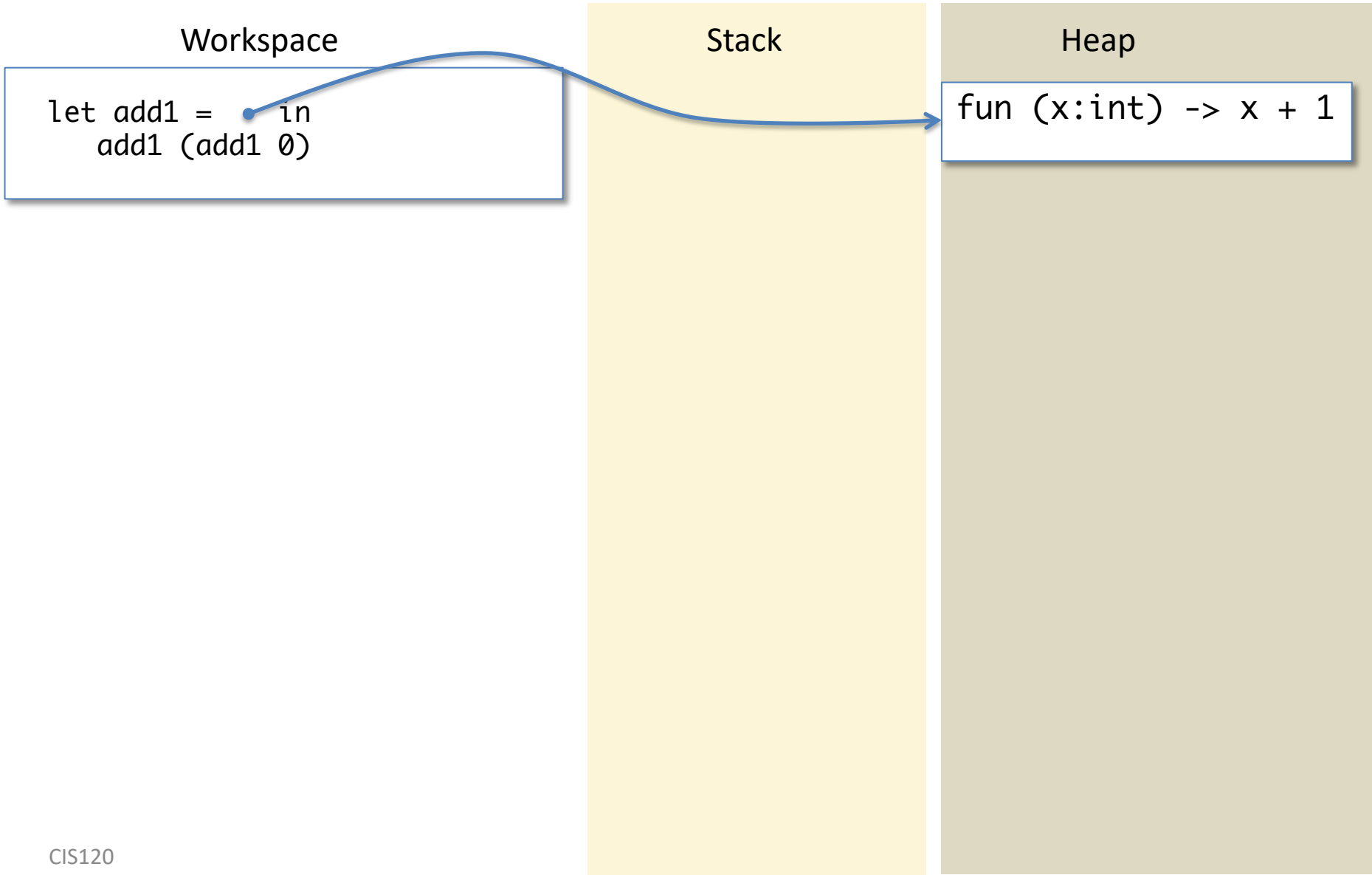
Workspace

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

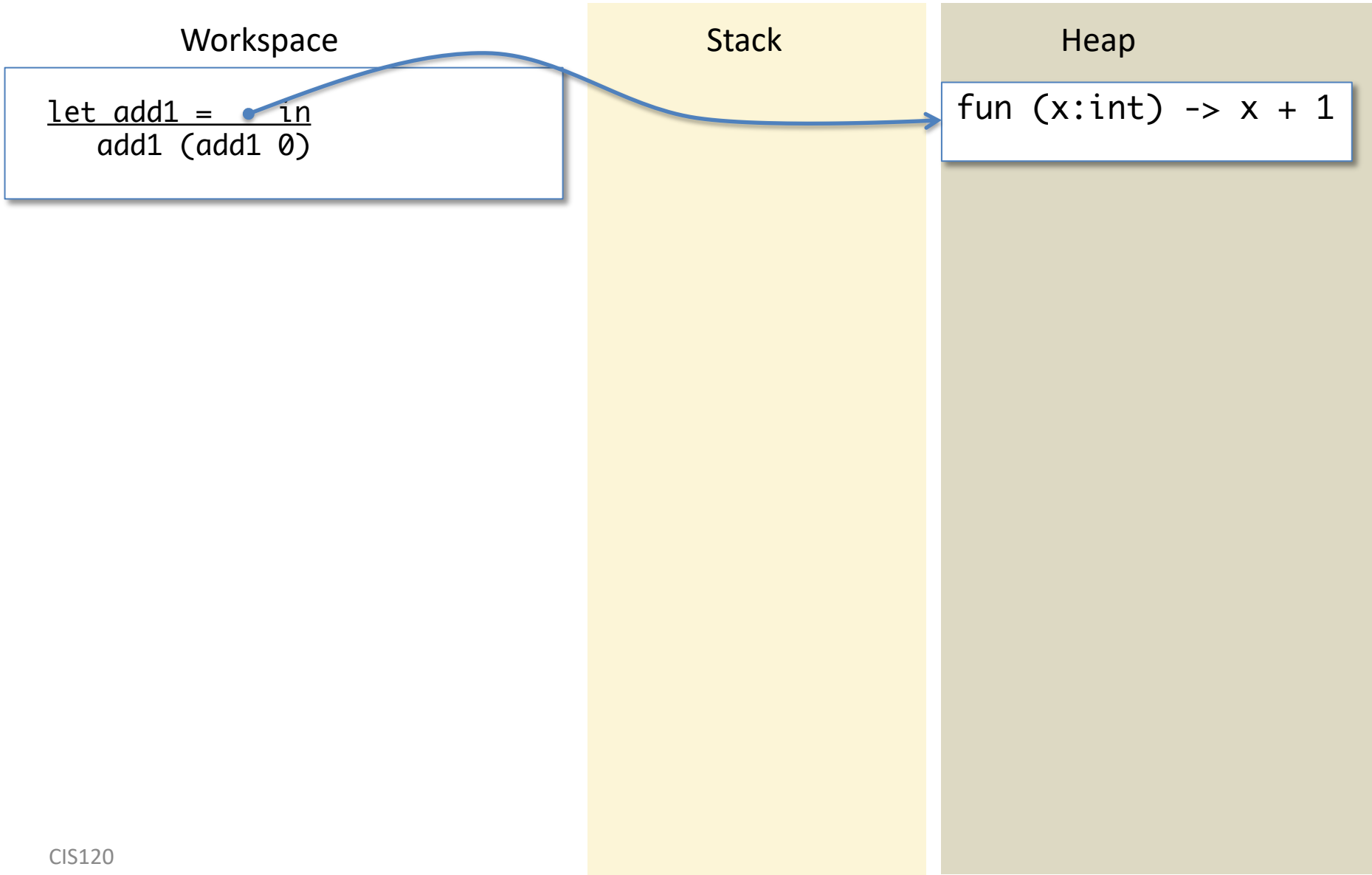
Stack

Heap

Function Simplification



Function Simplification

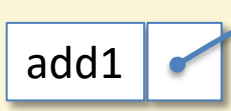


Function Simplification

Workspace

```
add1 (add1 0)
```

Stack



Heap

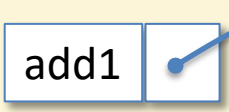
```
fun (x:int) -> x + 1
```

Function Simplification

Workspace

```
add1 (add1 0)
```

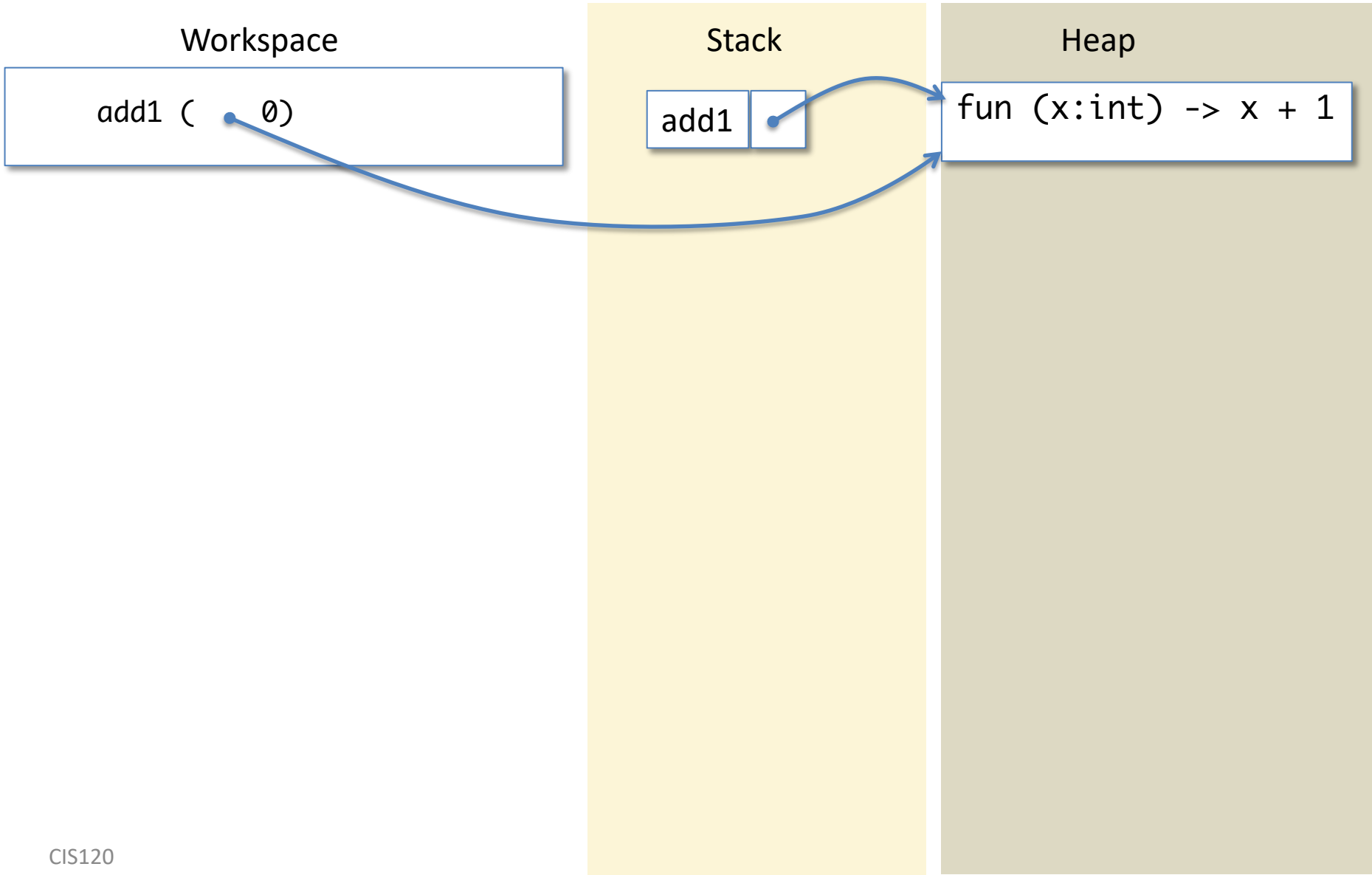
Stack



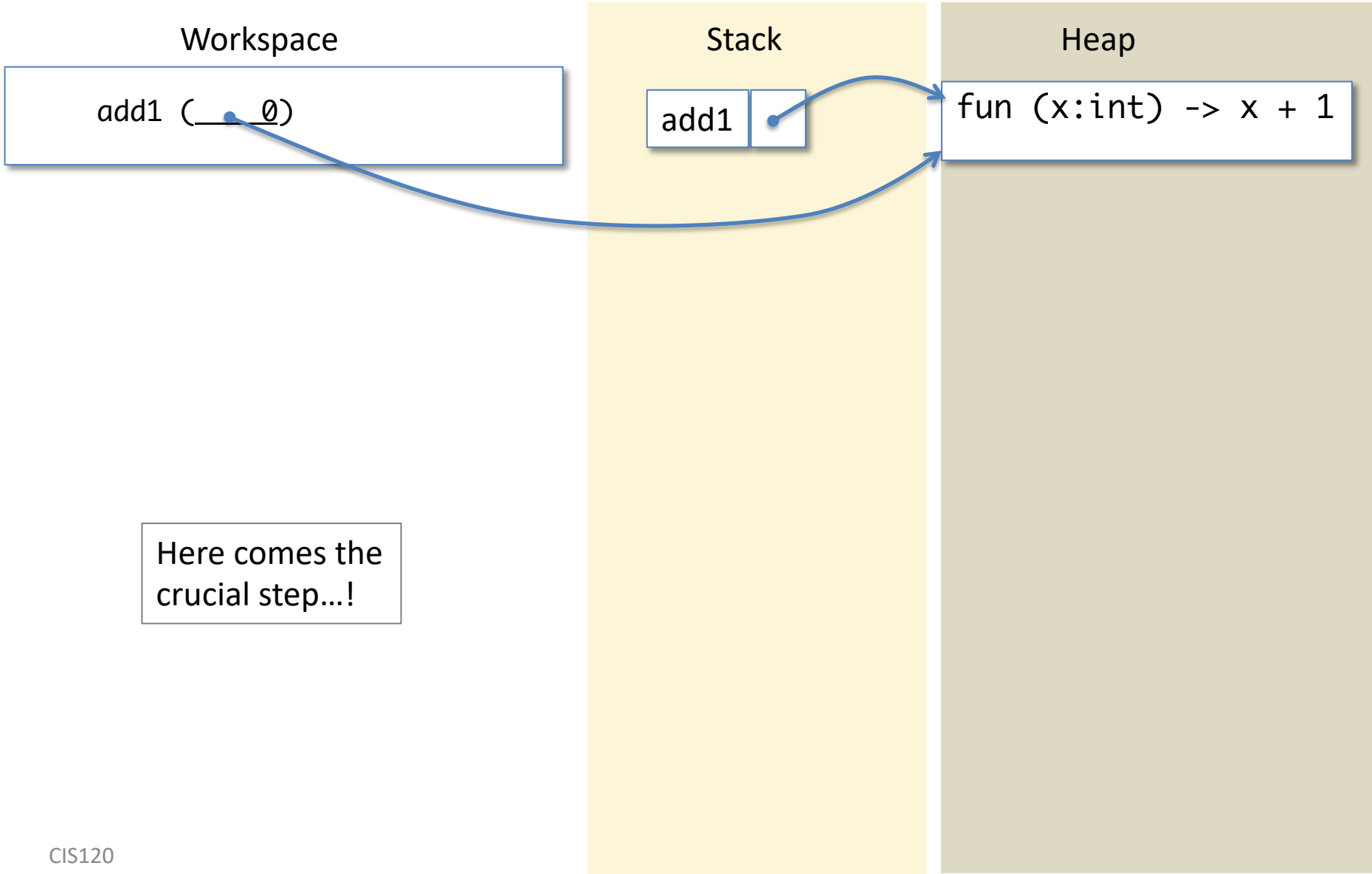
Heap

```
fun (x:int) -> x + 1
```

Function Simplification

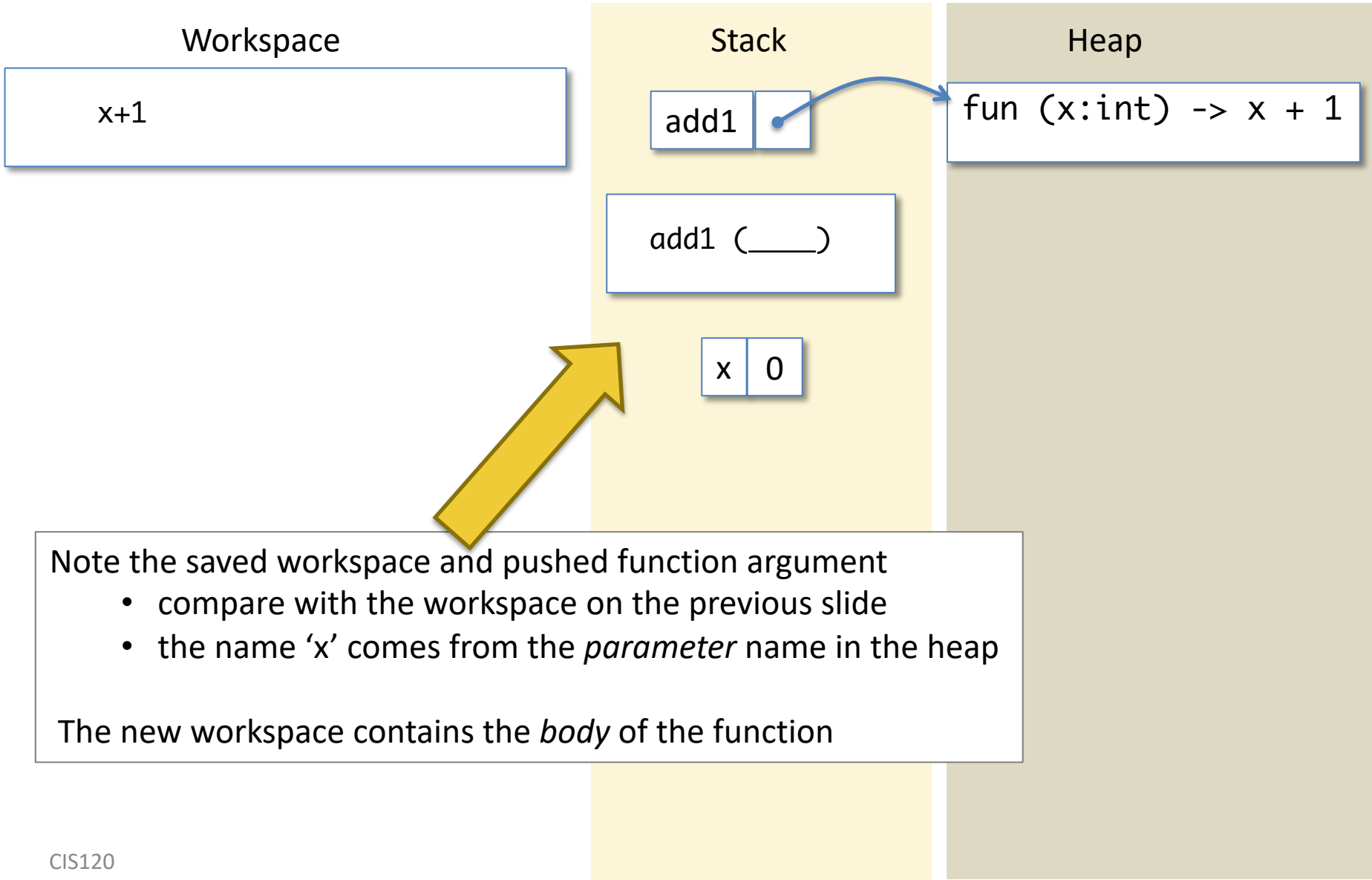


Function Simplification



Here comes the crucial step…!

Do the Call, Saving the Workspace

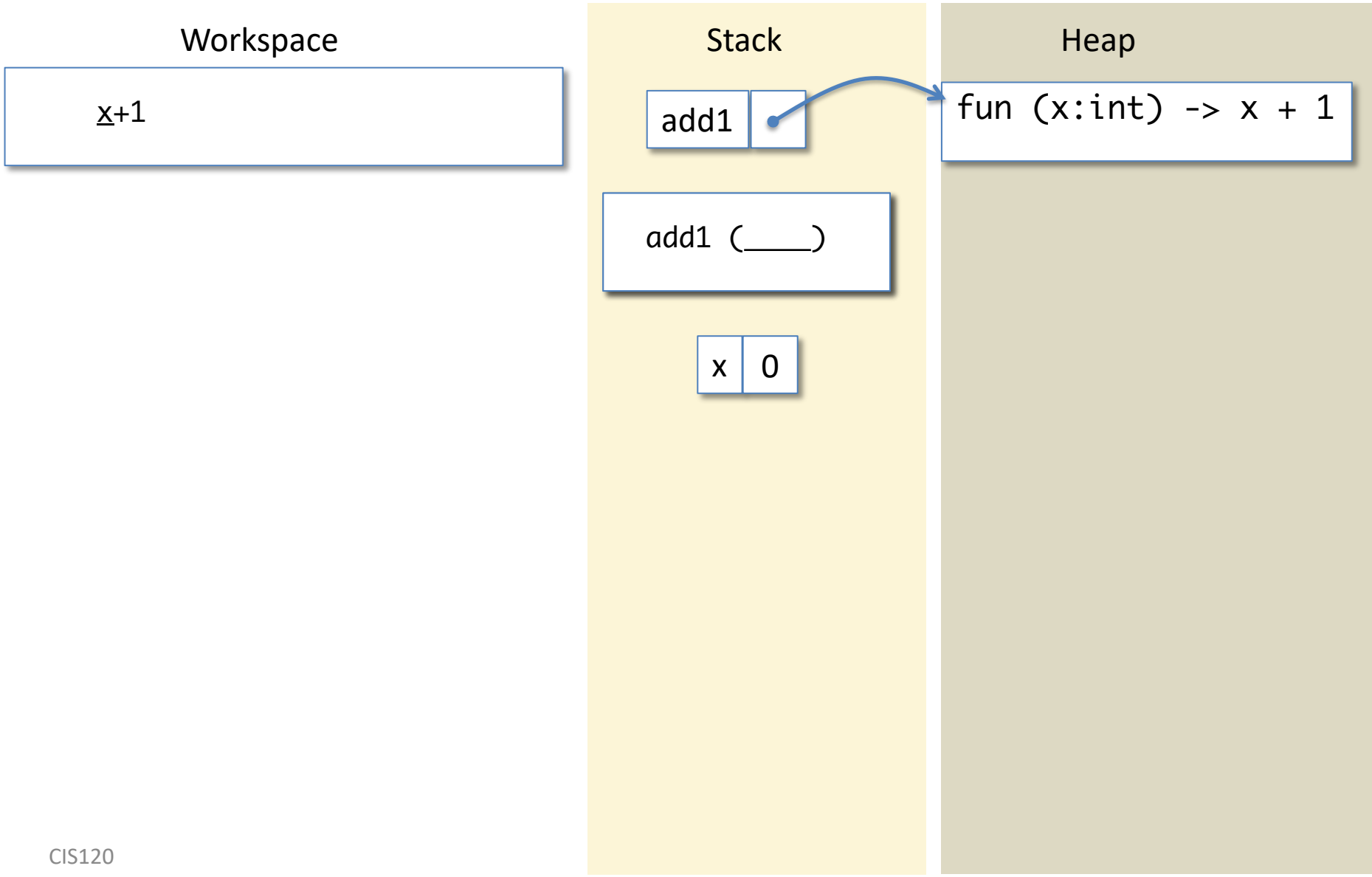


Note the saved workspace and pushed function argument

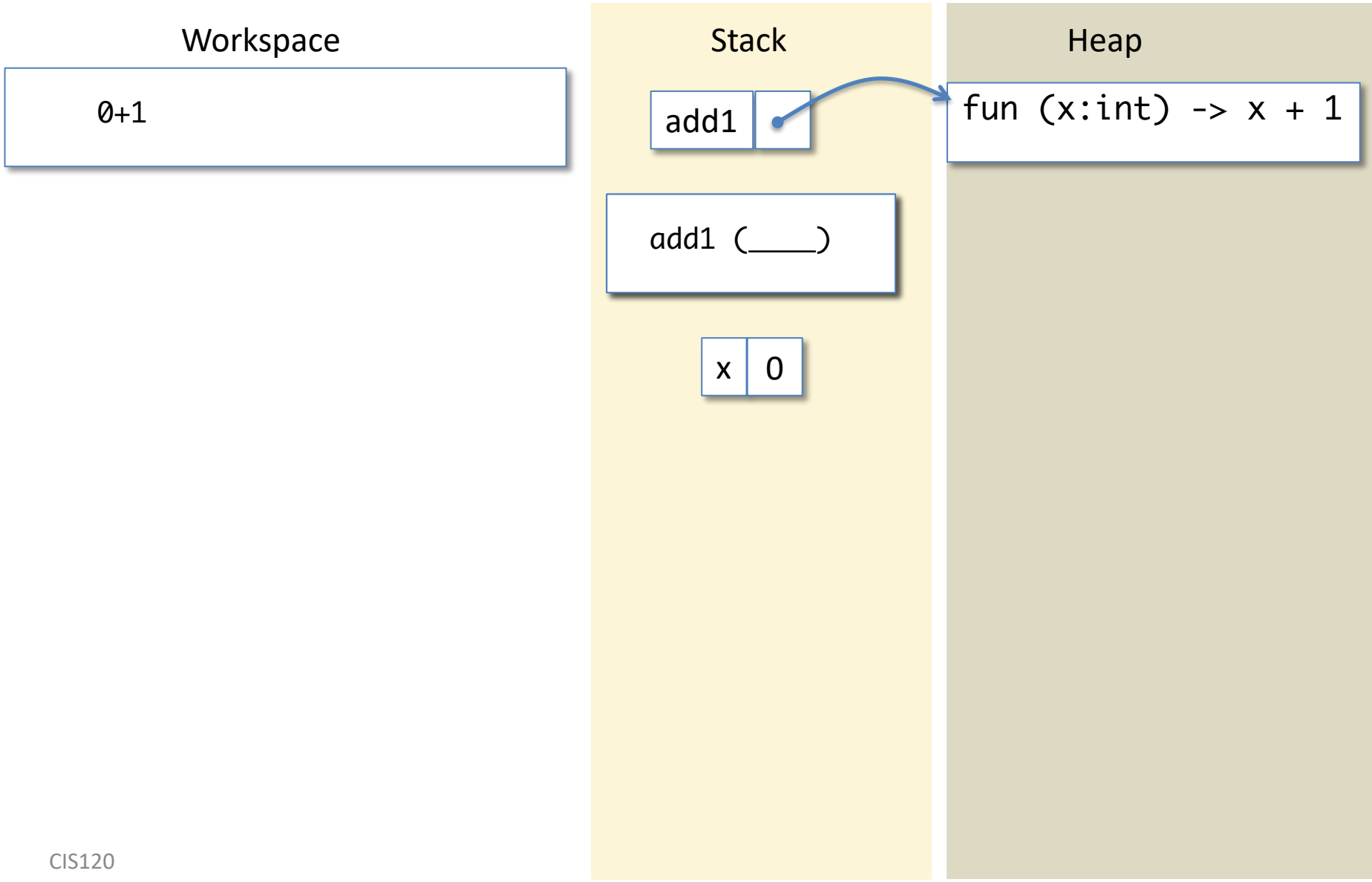
- compare with the workspace on the previous slide
- the name 'x' comes from the *parameter* name in the heap

The new workspace contains the *body* of the function

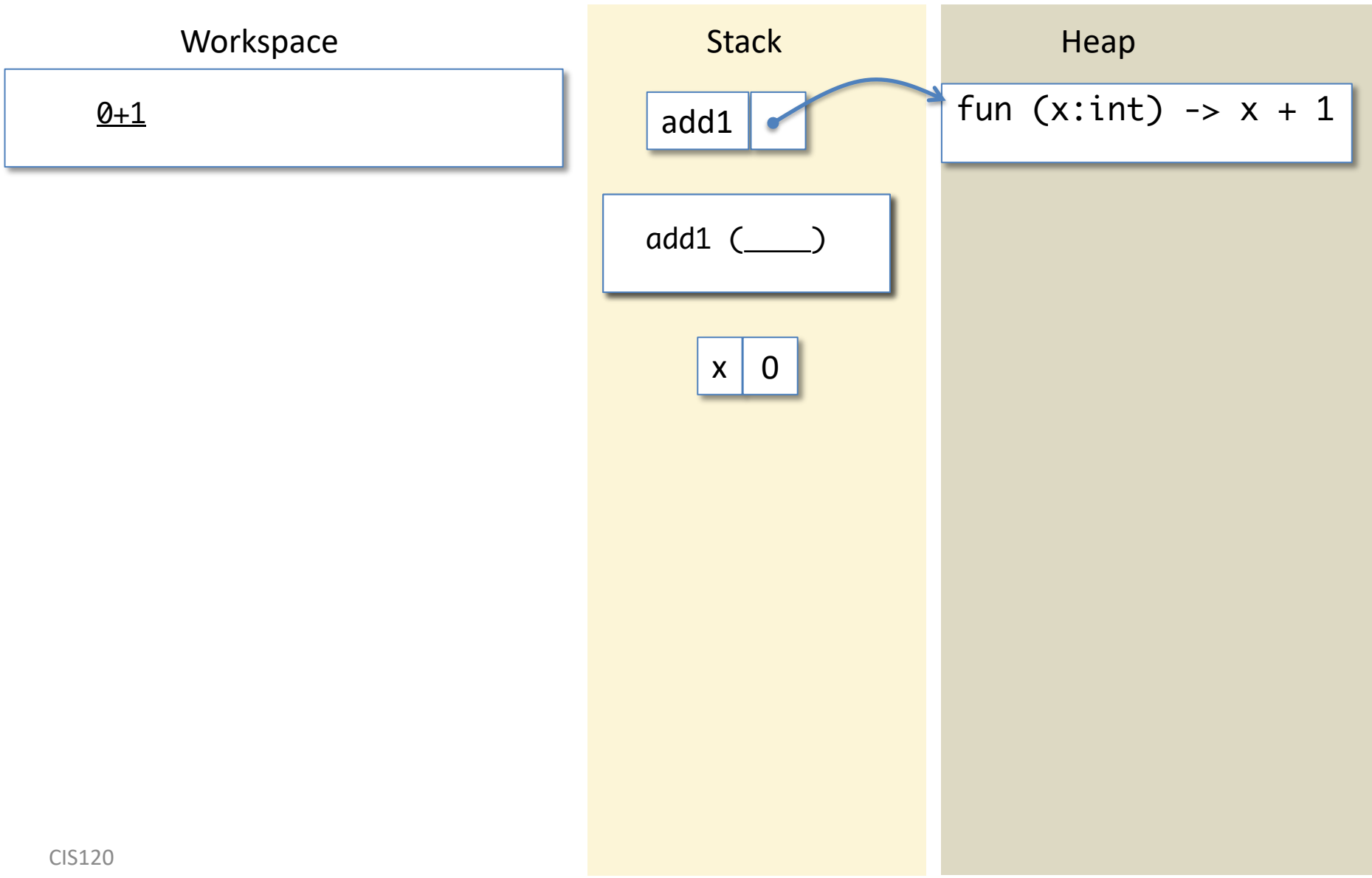
Function Simplification



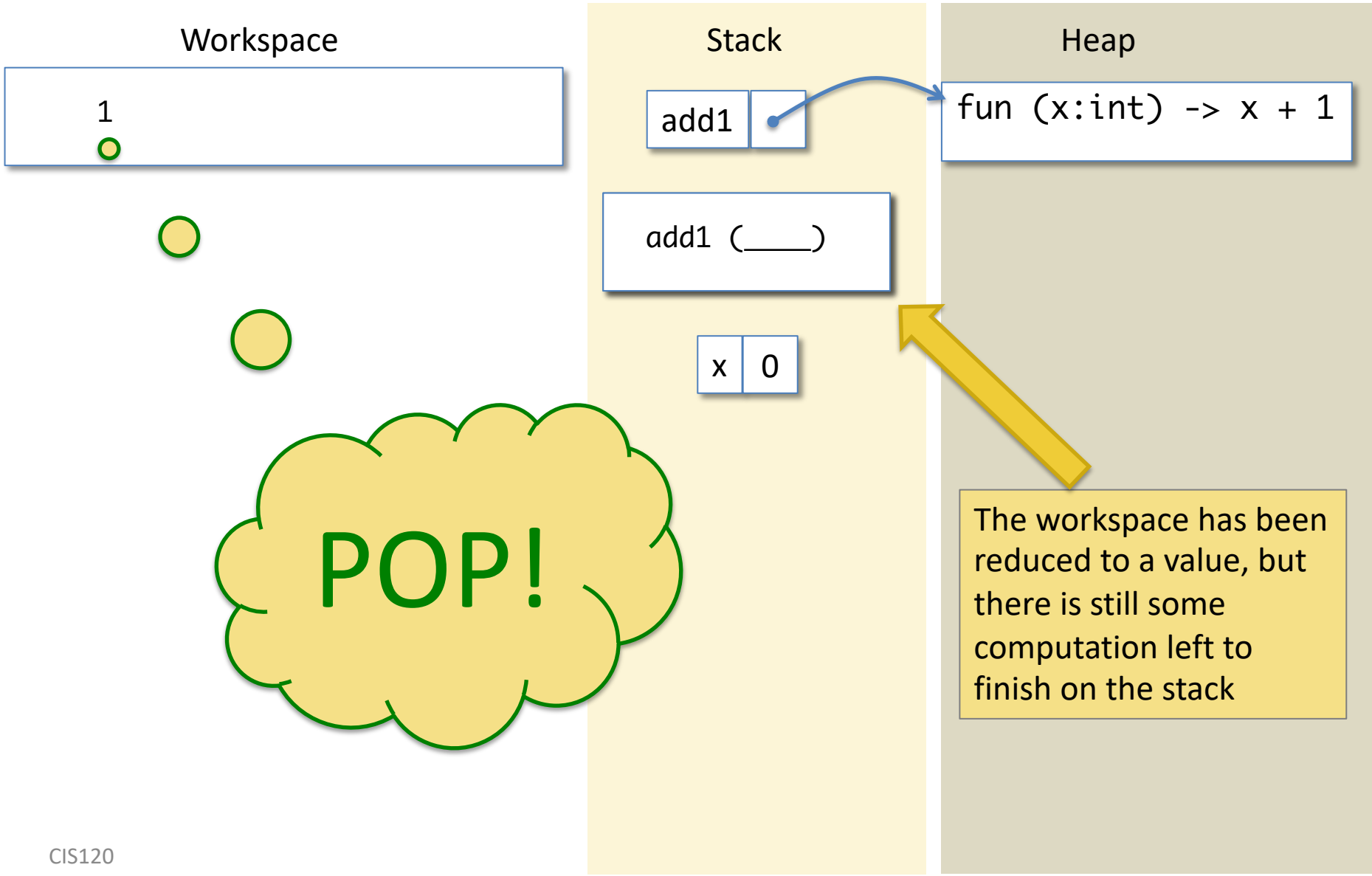
Function Simplification



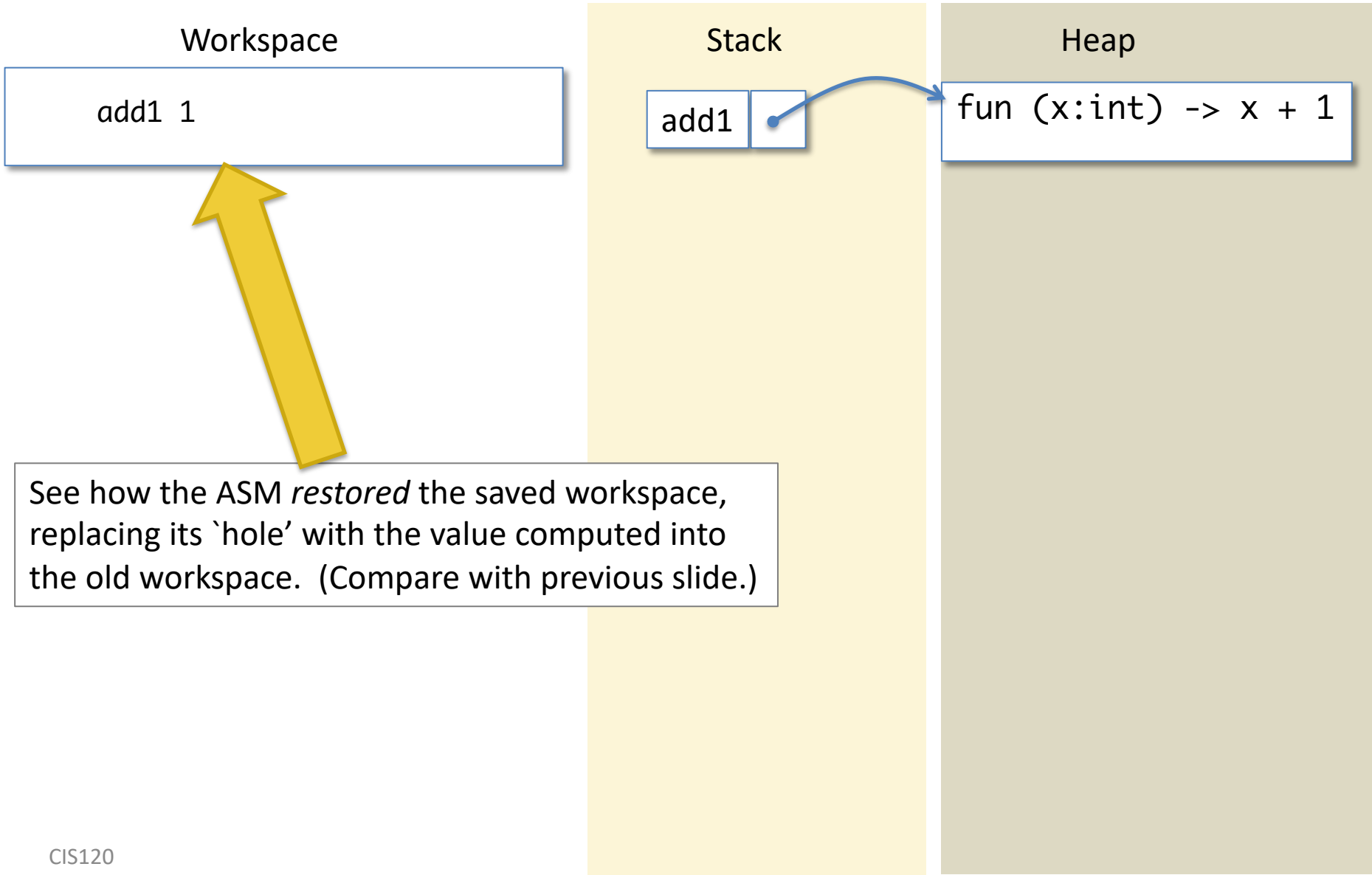
Function Simplification



Function Simplification



Function Simplification

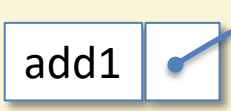


Function Simplification

Workspace

add1 1

Stack

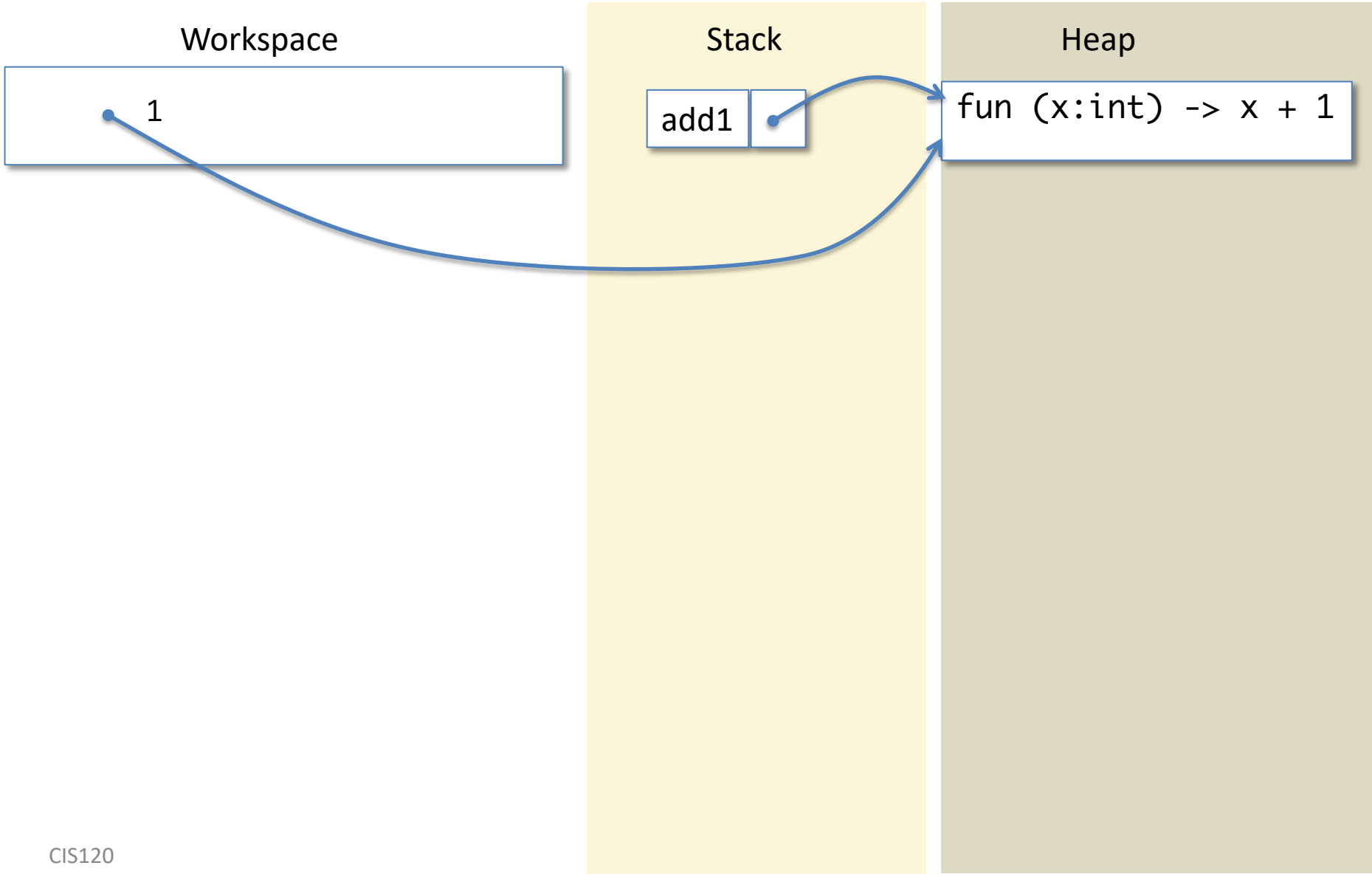


Heap

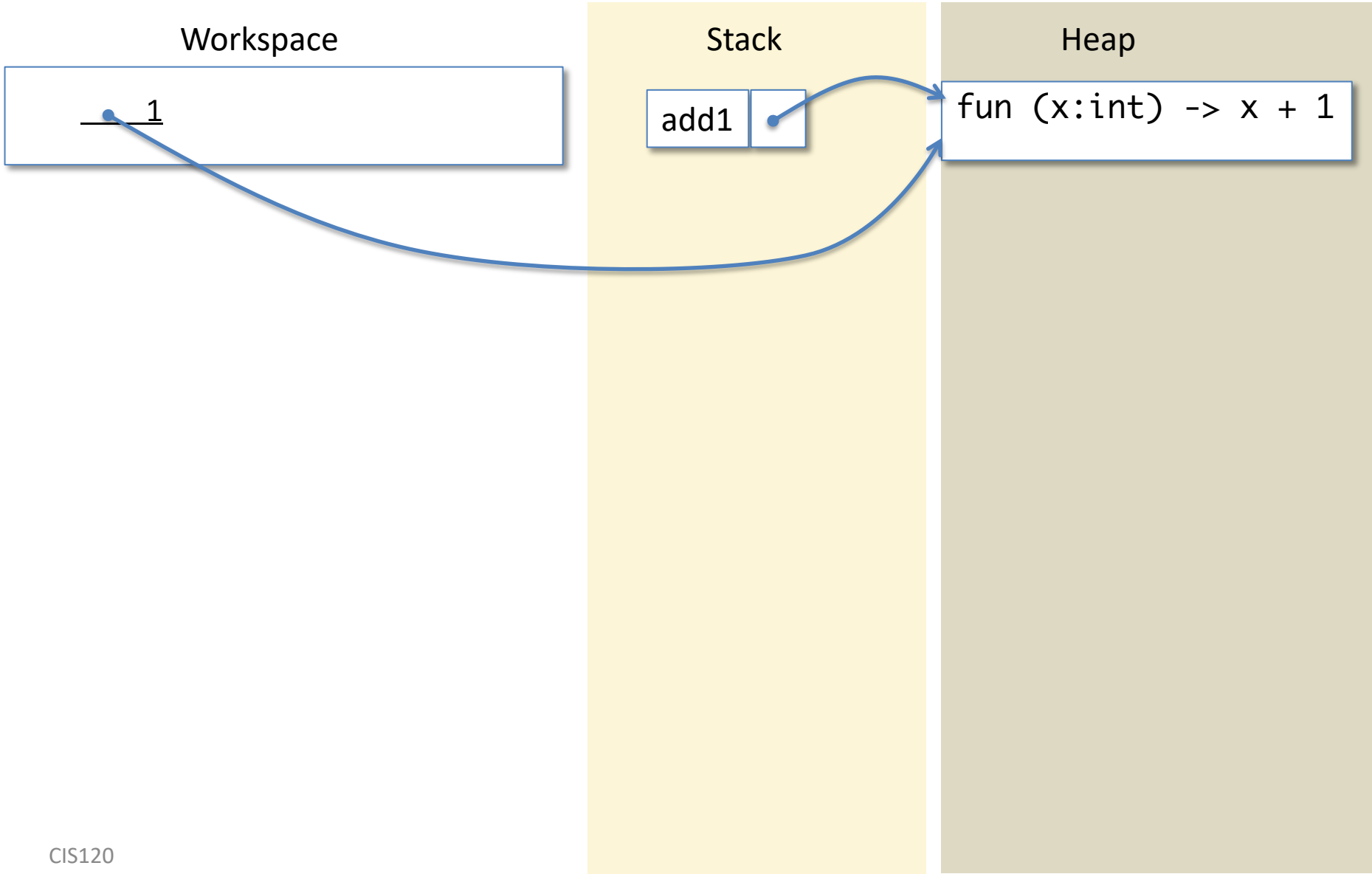
```
fun (x:int) -> x + 1
```

Now we have to do it all over again for the second invocation of add1…

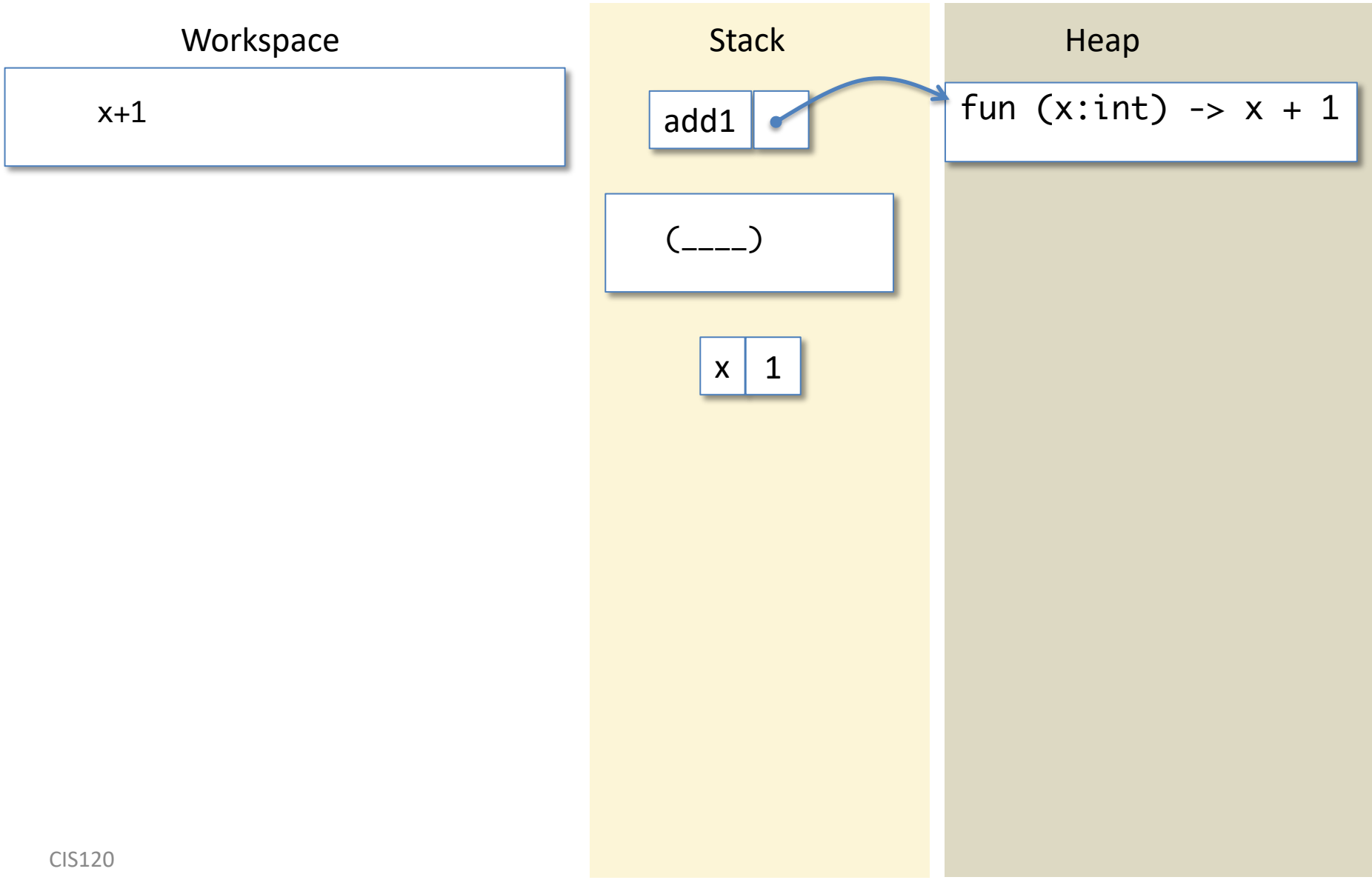
Function Simplification



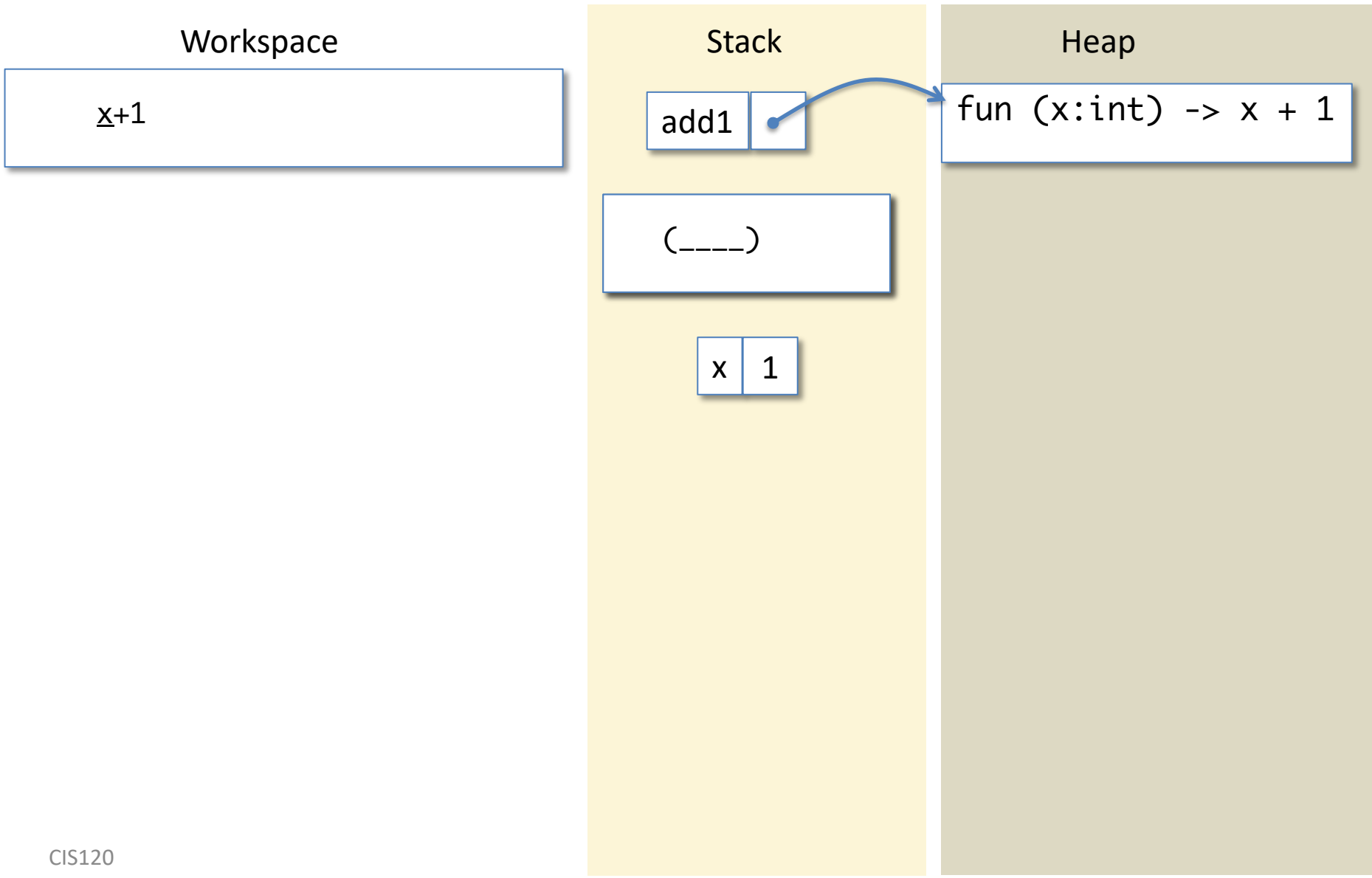
Function Simplification



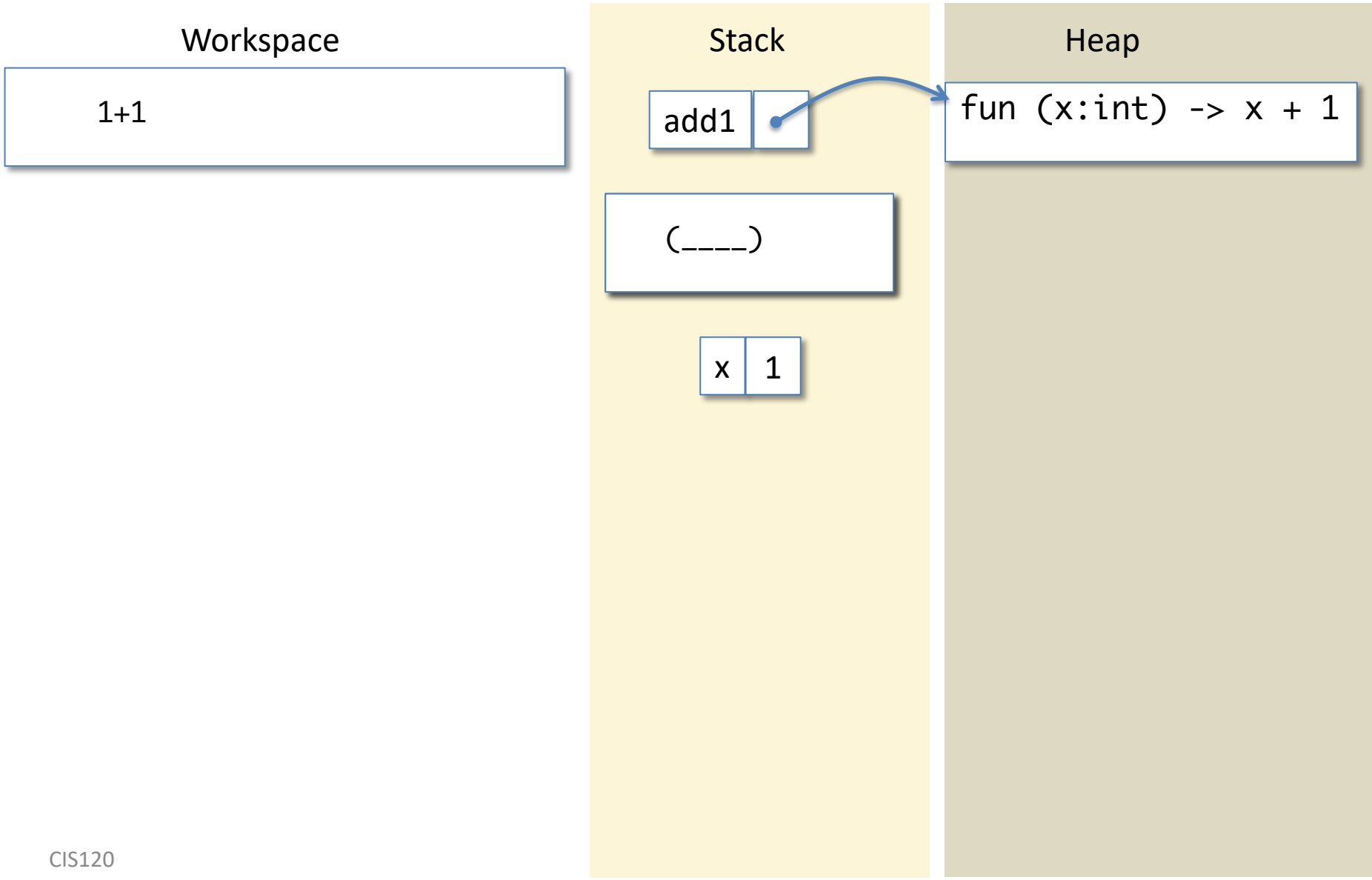
Function Simplification



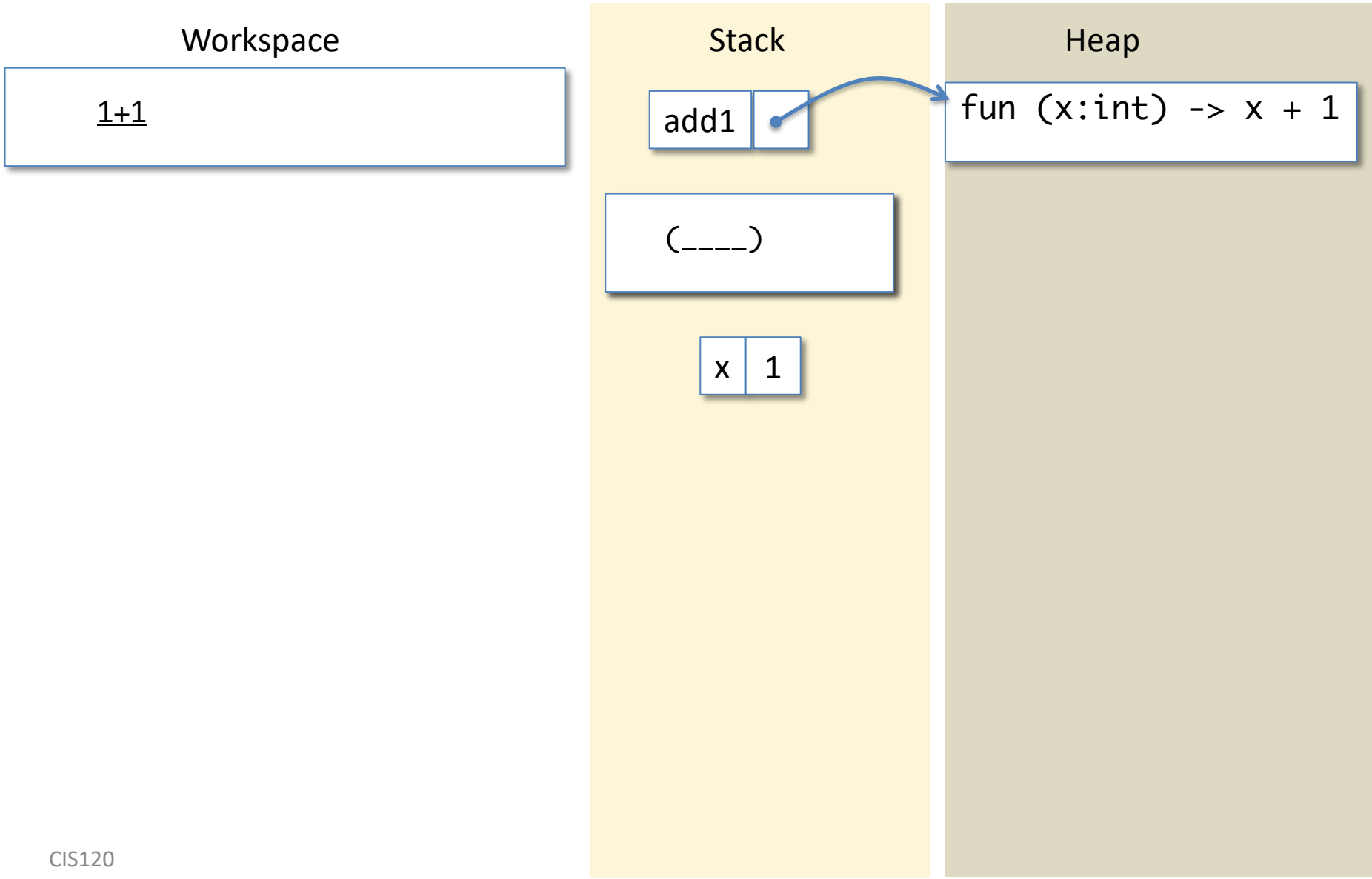
Function Simplification



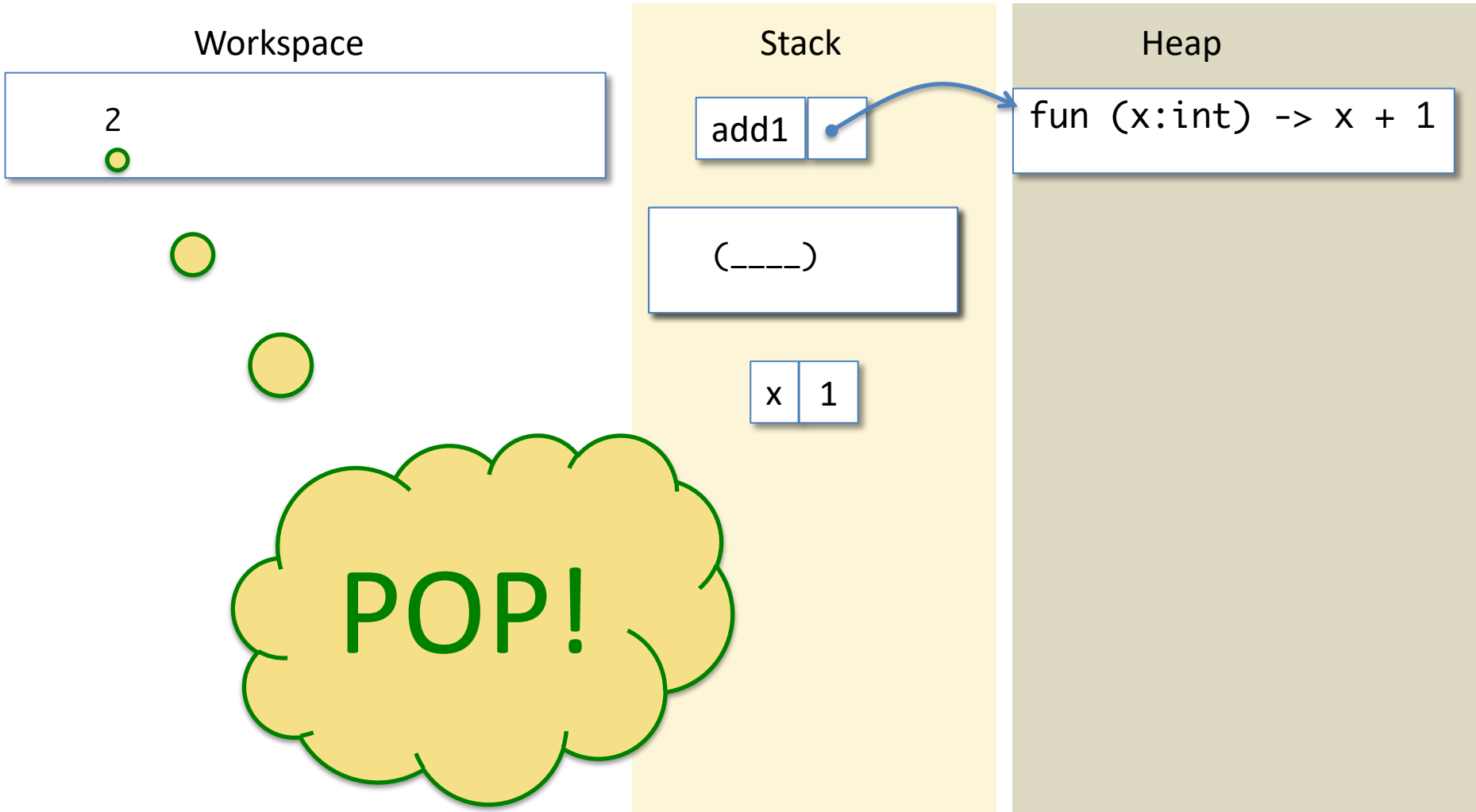
Function Simplification



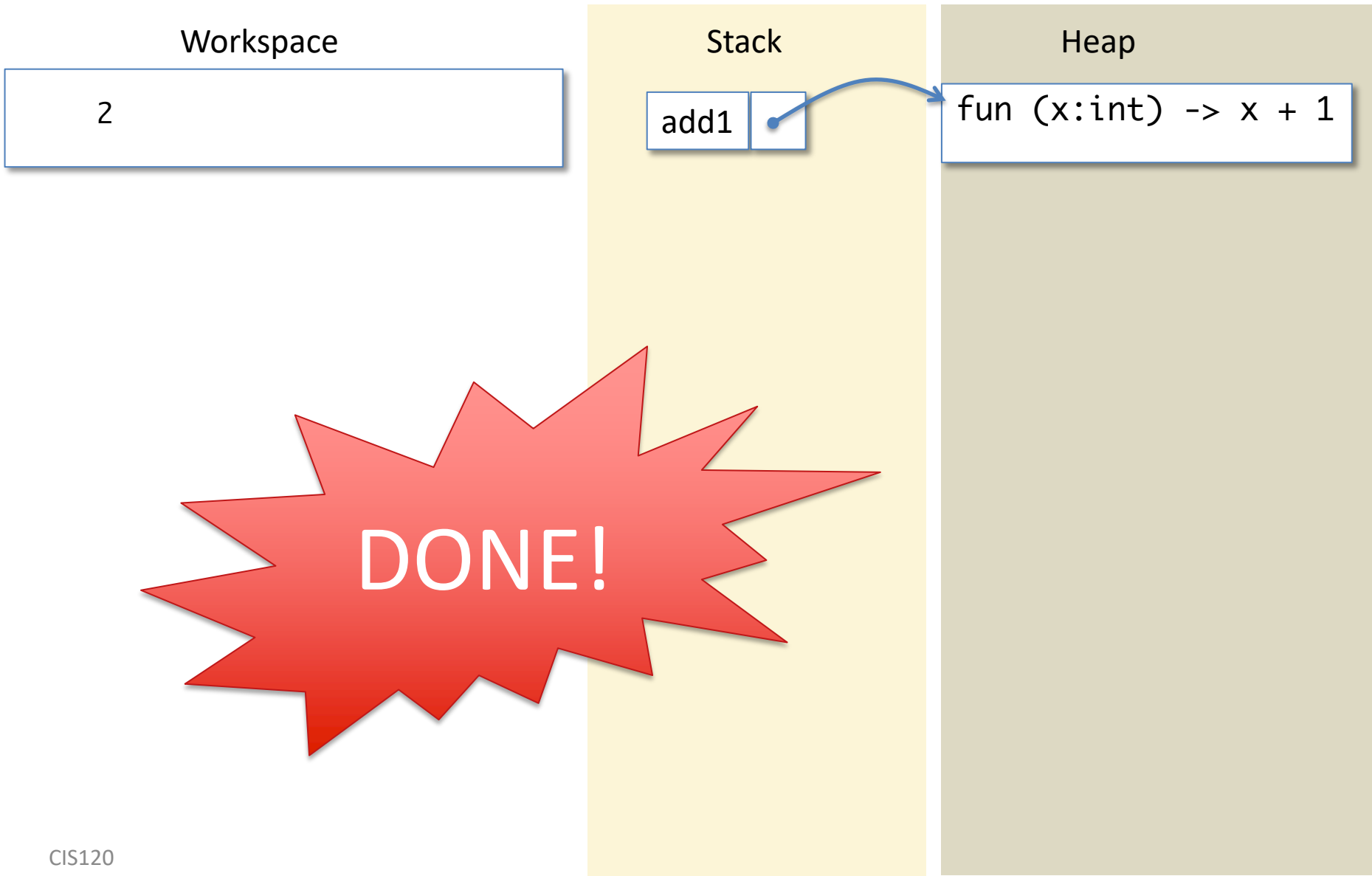
Function Simplification



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 workspace contents in the stack, the whole computation is finished and the value in the workspace is its final result.

ASM:
pattern matching and recursion

Example

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

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

append a b
```

Simplification

Workspace

```
let rec append (l1: 'a list)
    (l2: 'a list) : 'a list =
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
      Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

Heap

Function Definition

Workspace

```
let rec append (l1: 'a list)
    (l2: 'a list) : 'a list =
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

Heap

Rewrite to a “fun”

Workspace

```
let rec append =
  fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

Heap

Function Expression

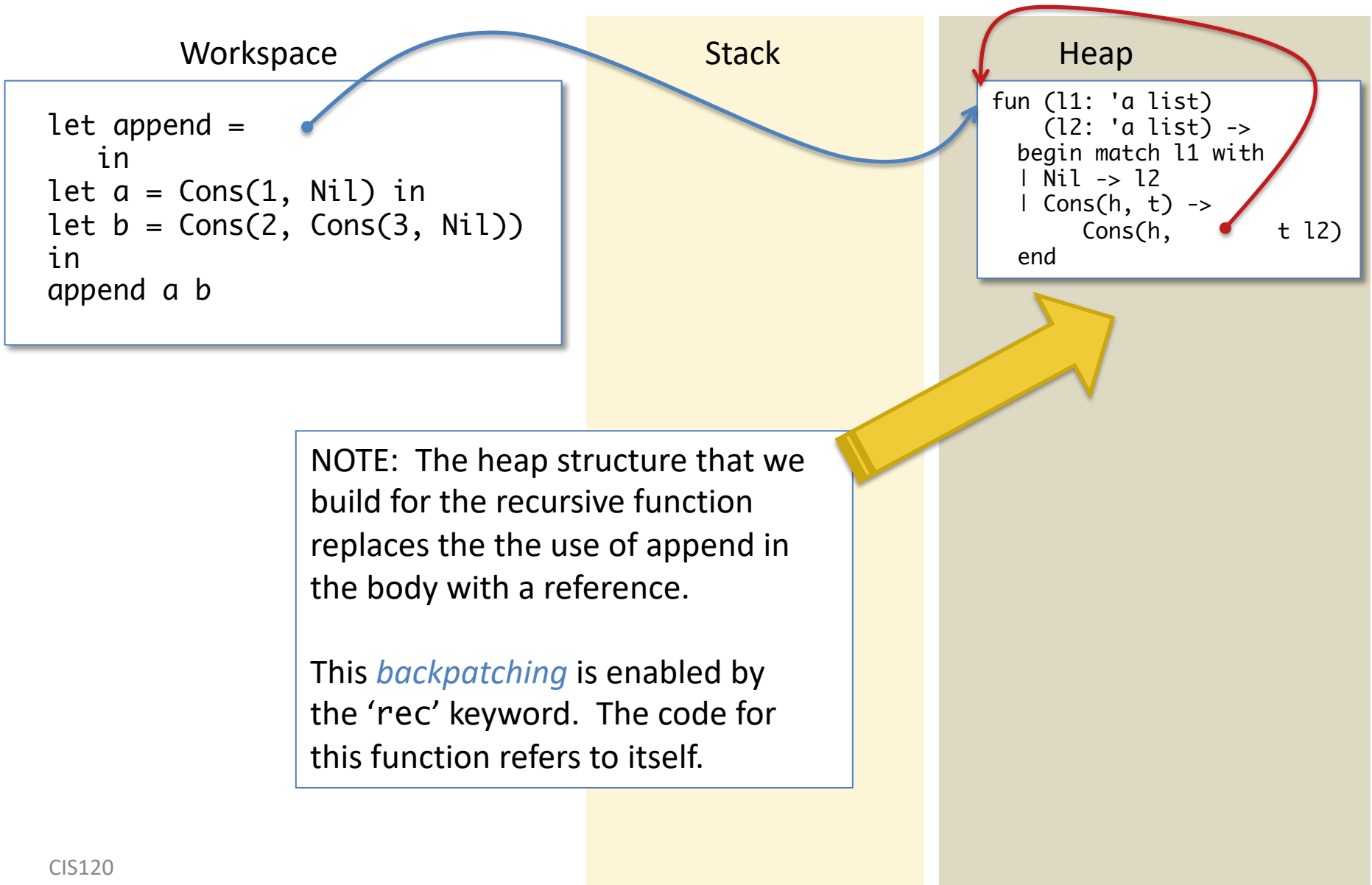
Workspace

```
let rec append =
  fun (l1: 'a list)
    (l2: 'a list) ->
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) ->
       Cons(h, append t l2)
  end in
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

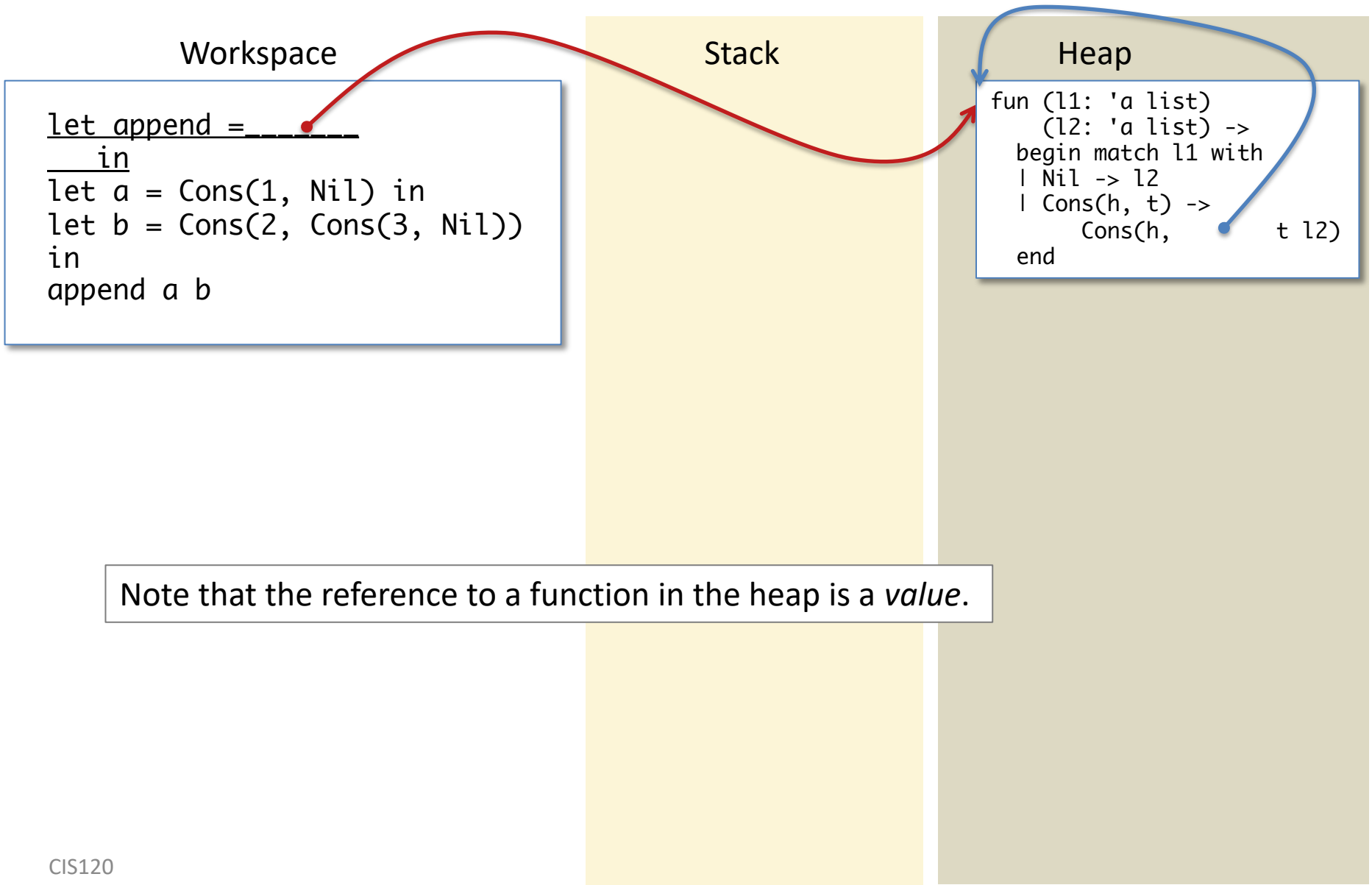
Stack

Heap

Copy to the Heap, Replace w/Reference



Let Expression



Note that the reference to a function in the heap is a *value*.

Create a Stack Binding

Workspace

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

Stack

Heap

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

Allocate a Nil cell

Workspace

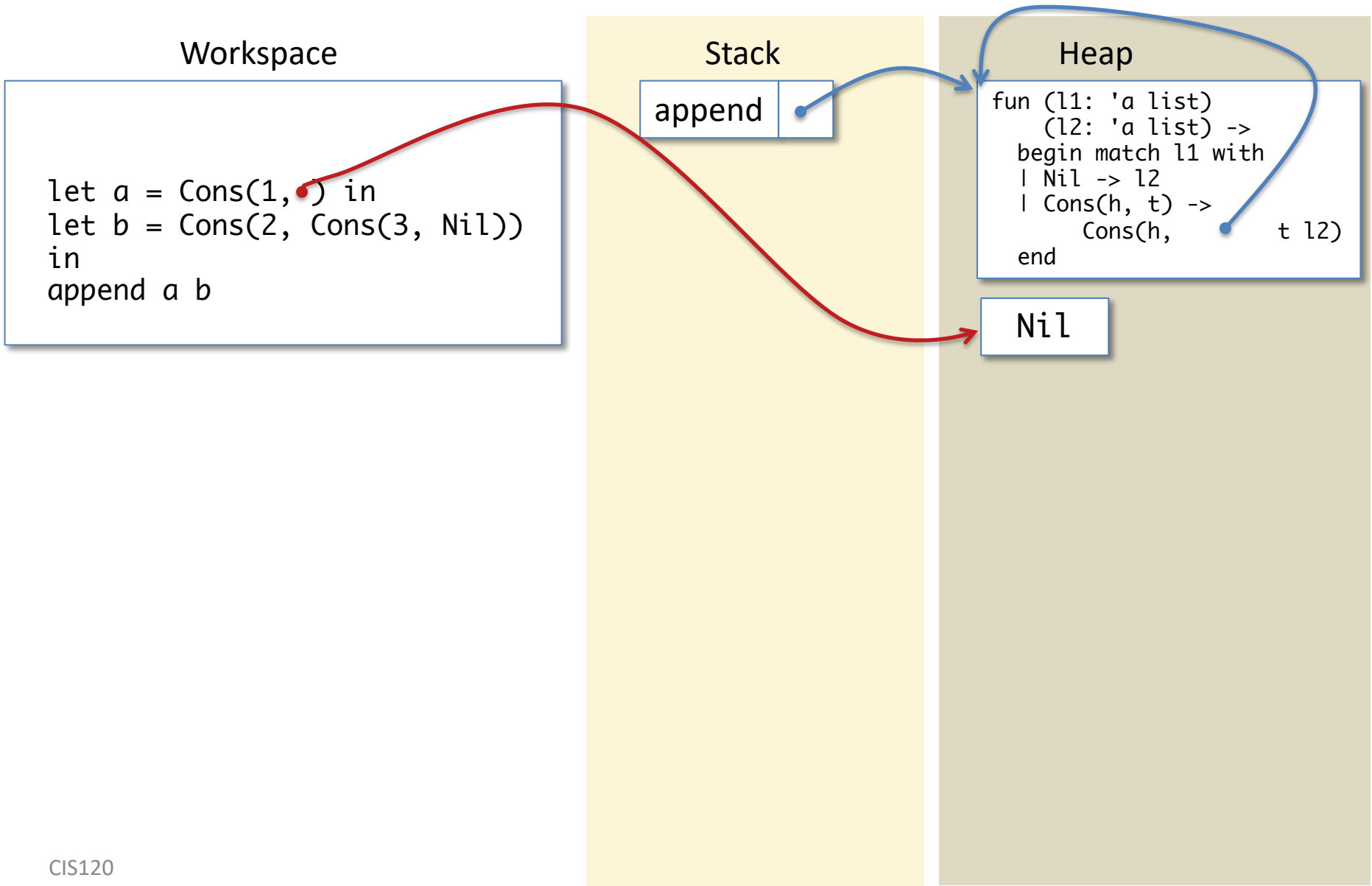
```
let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil))
in
append a b
```

Stack

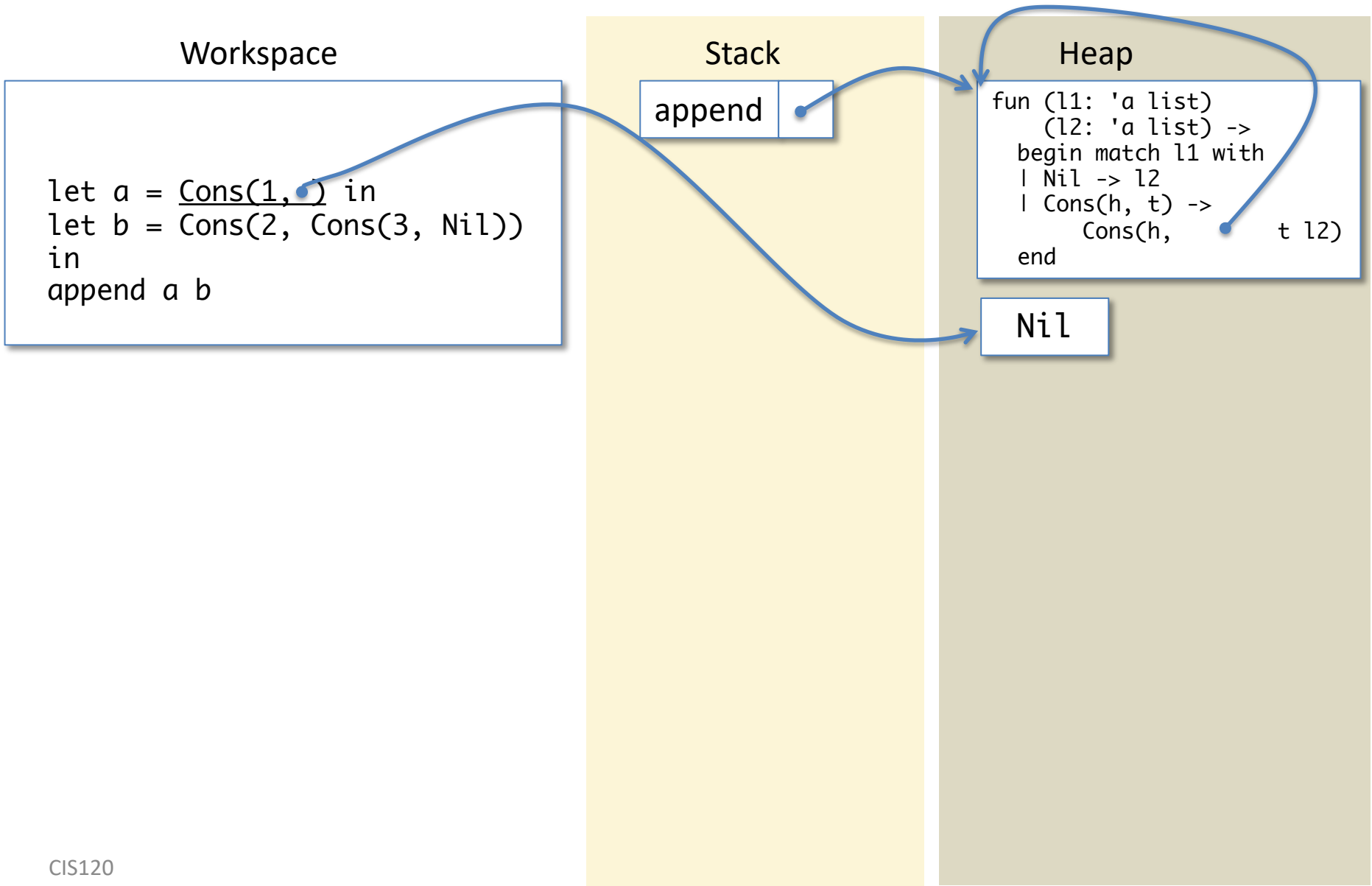
Heap

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

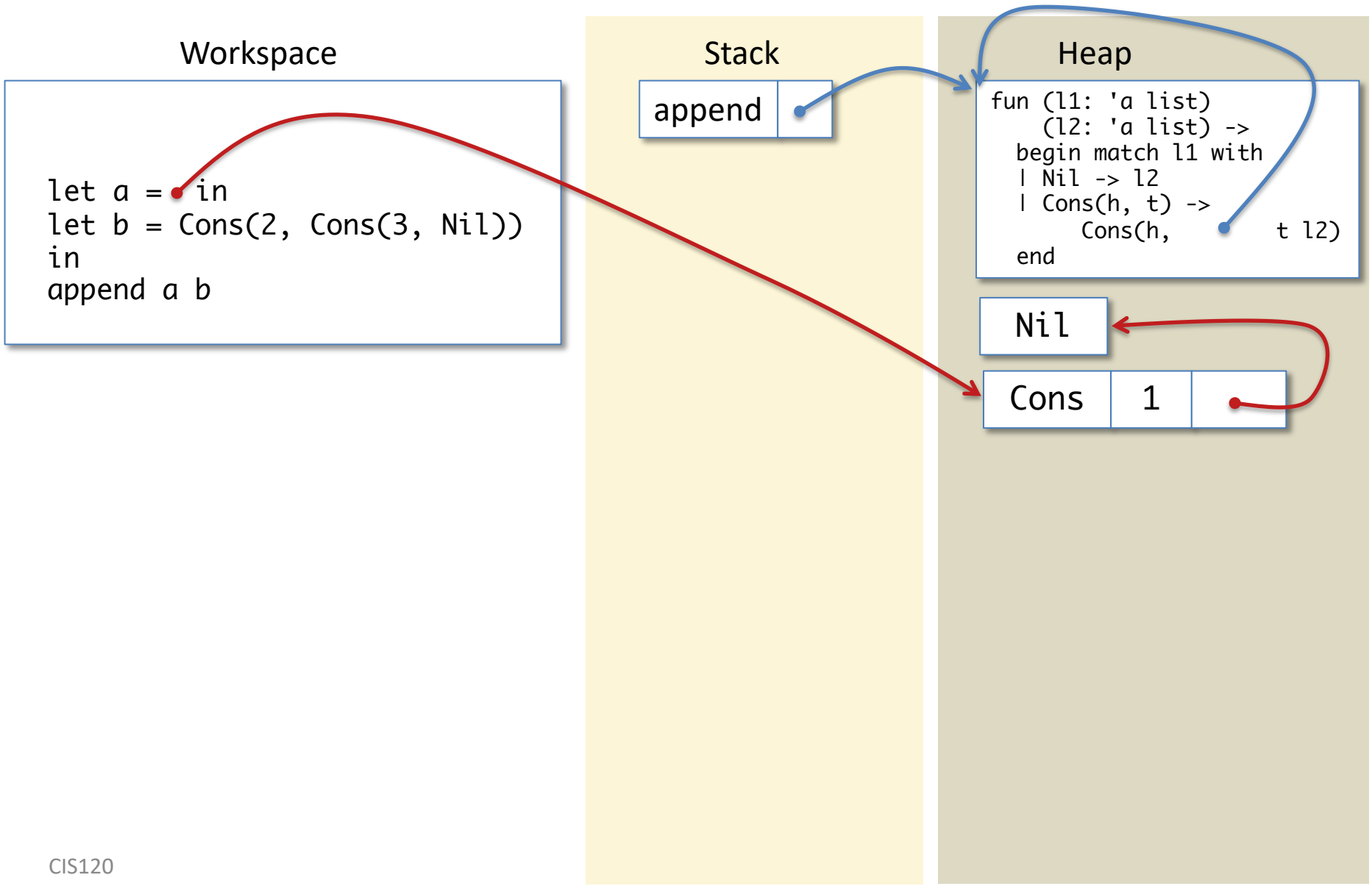
Allocate a Nil cell



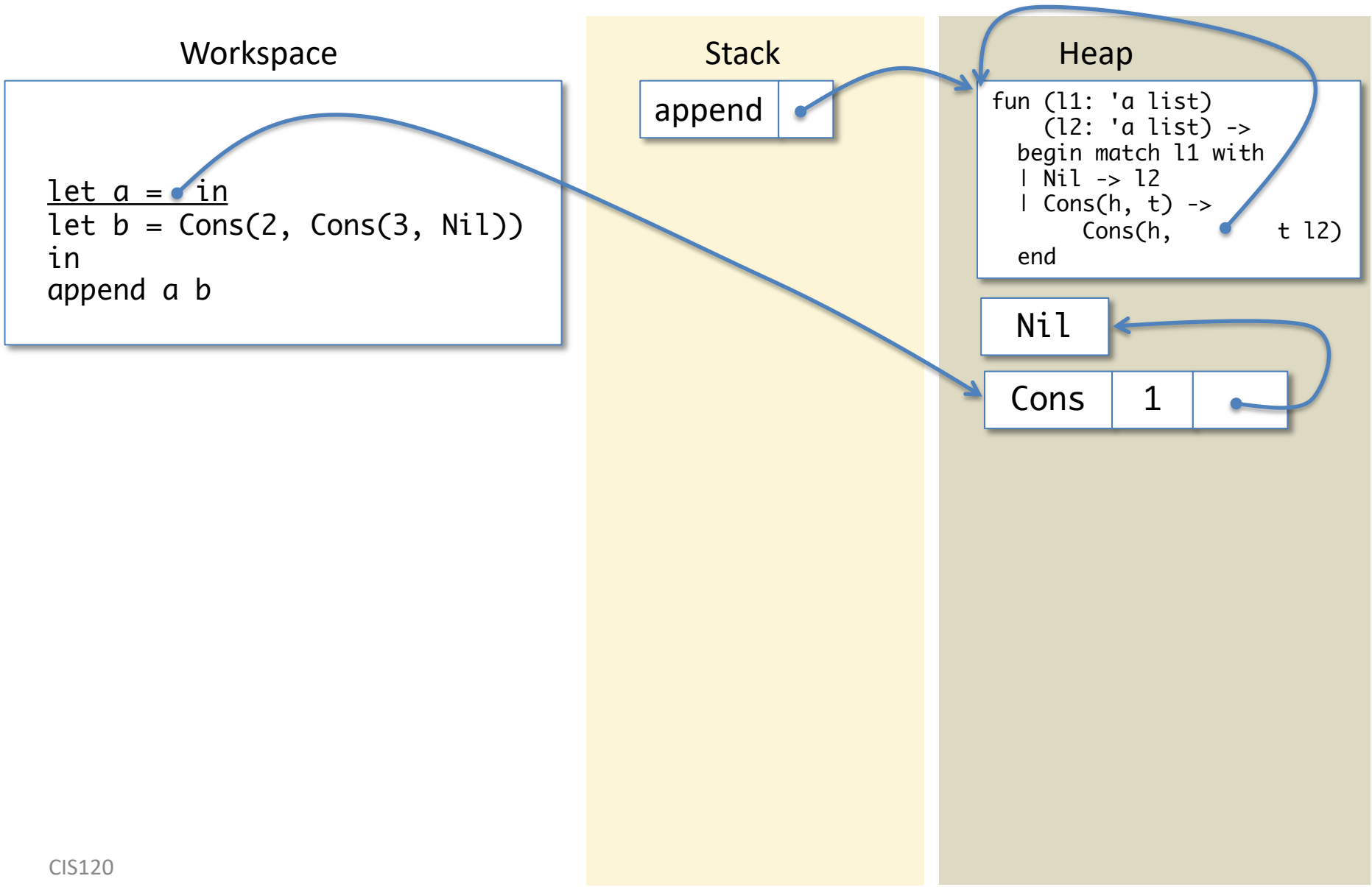
Allocate a Cons cell



Allocate a Cons cell



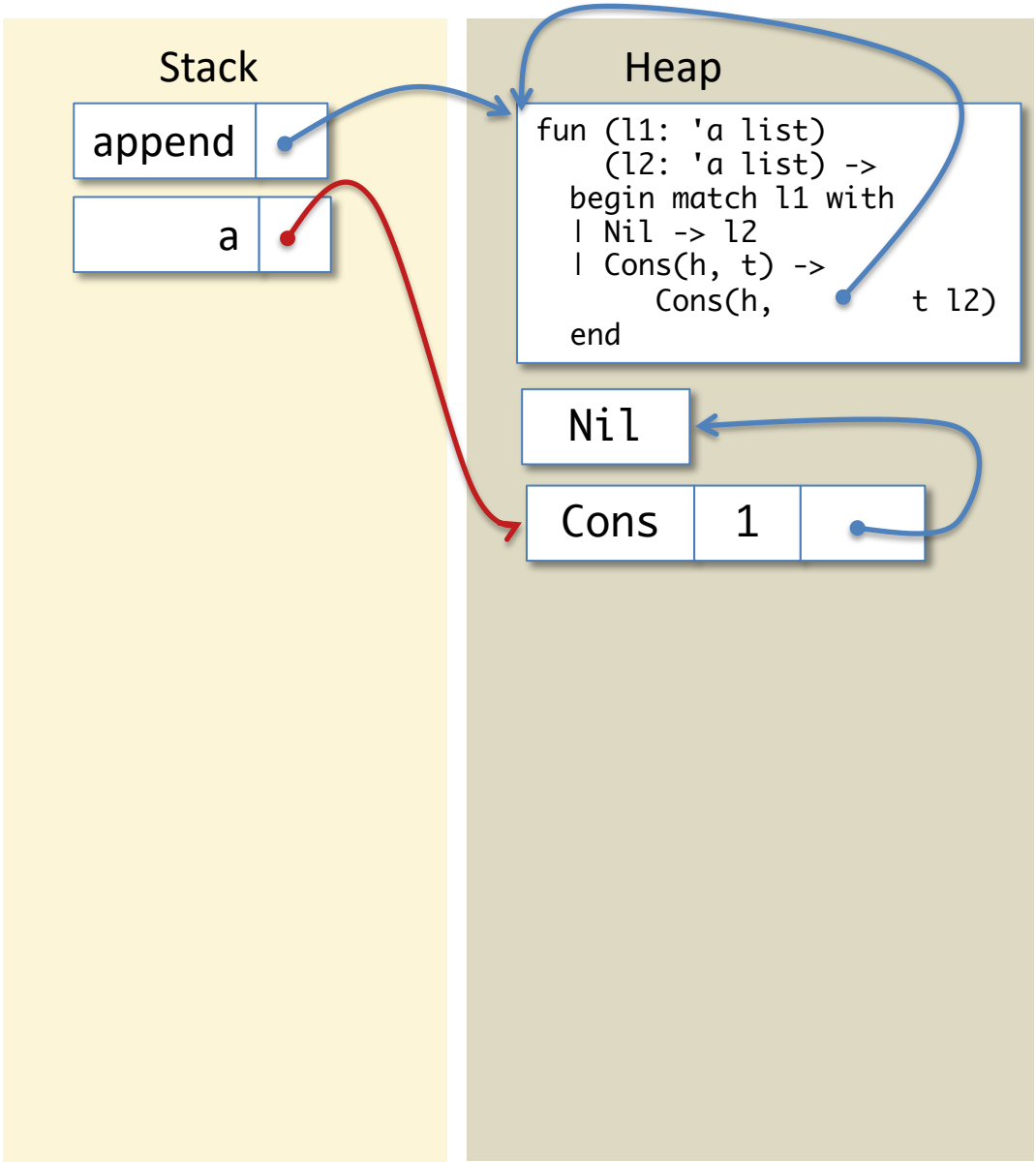
Let Expression



Create a Stack Binding

Workspace

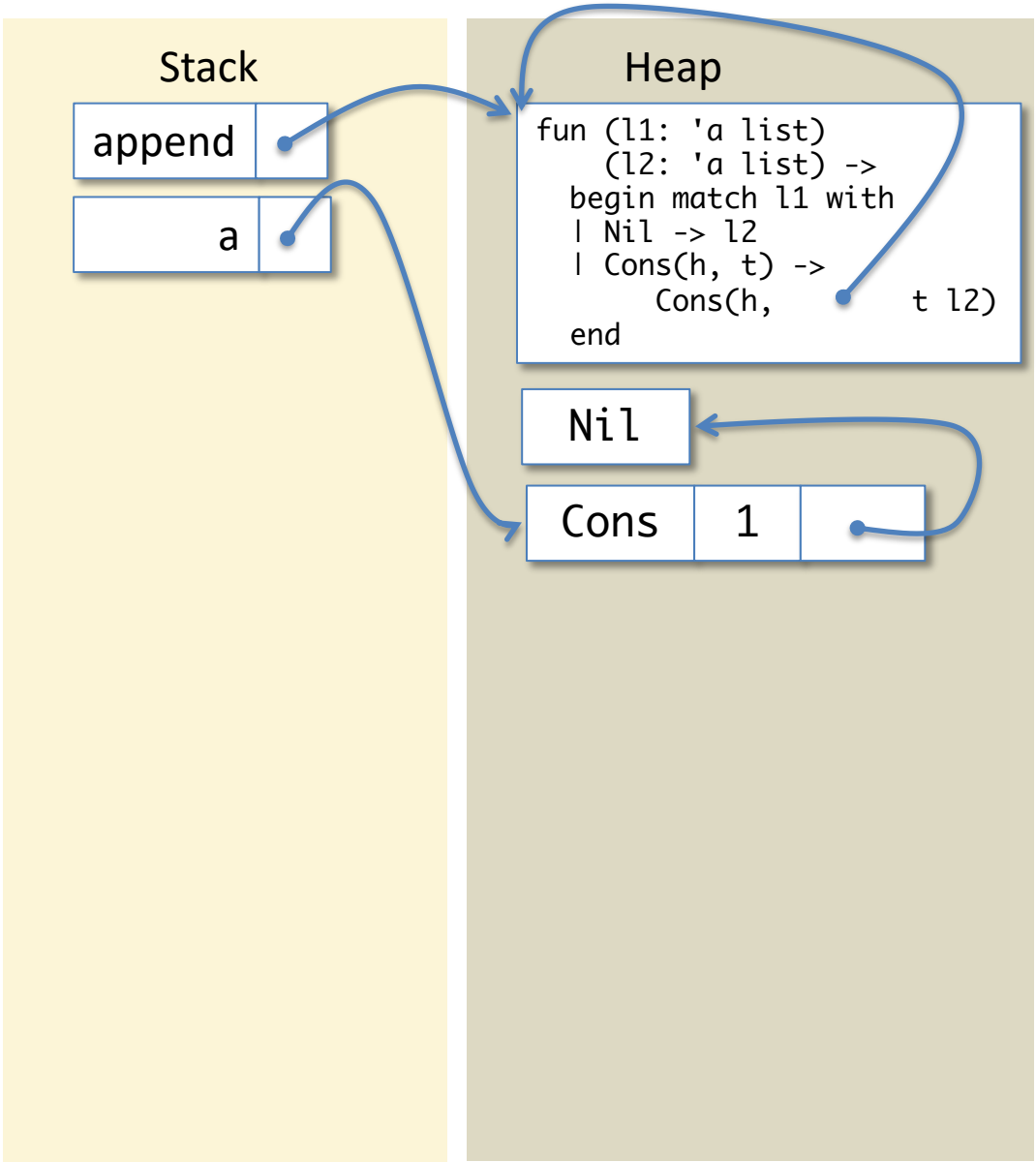
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```



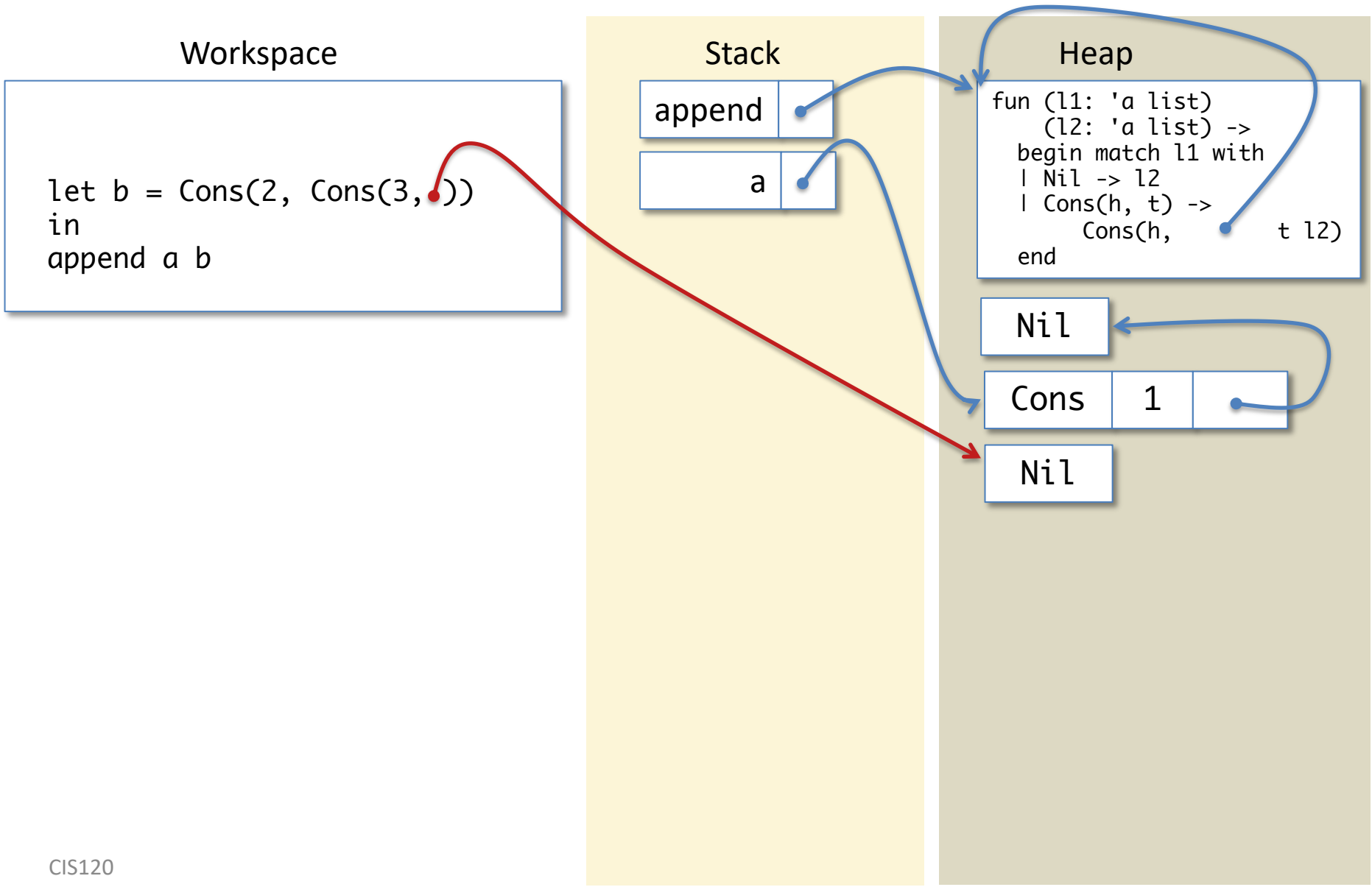
Allocate a Nil cell

Workspace

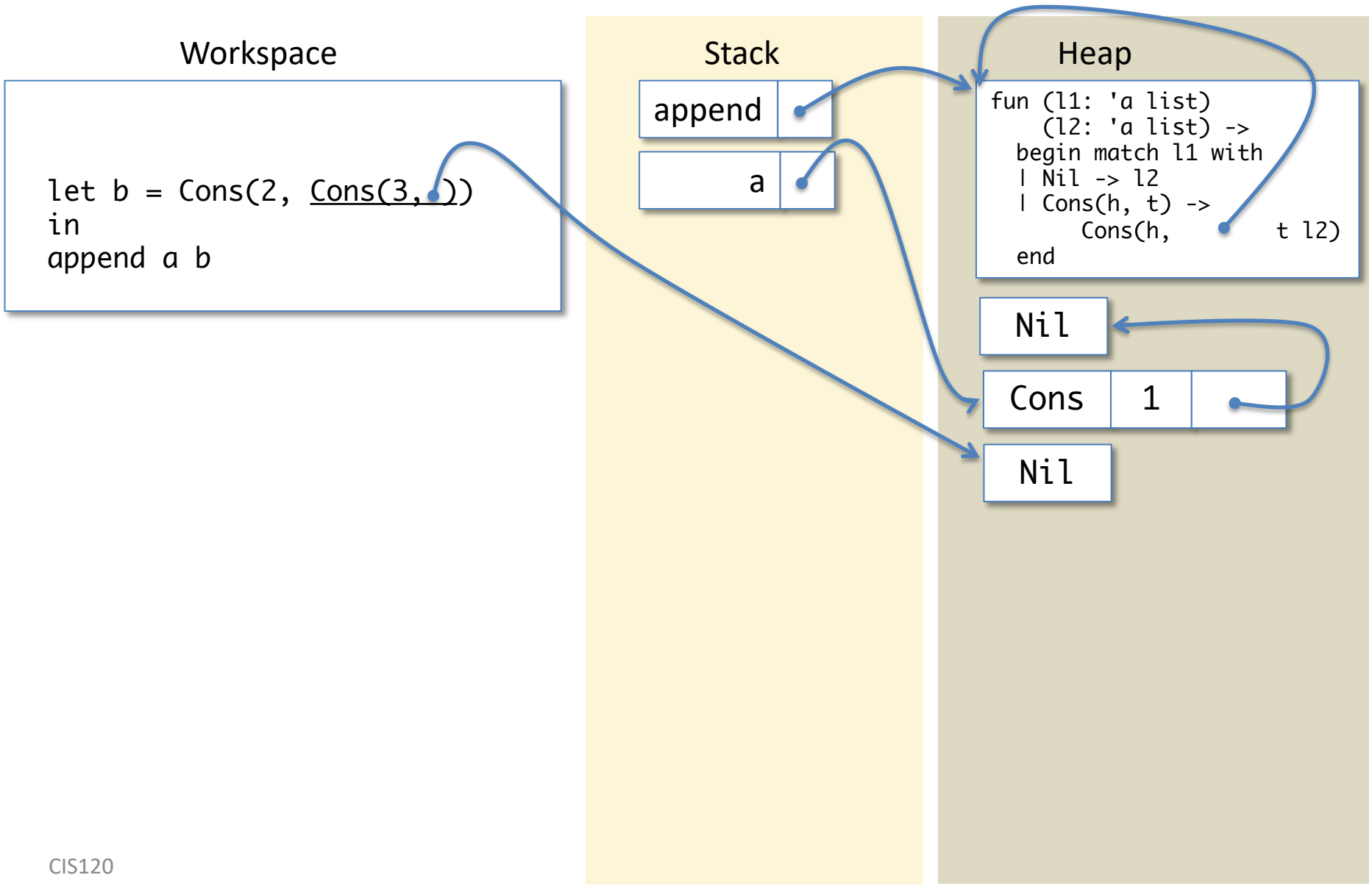
```
let b = Cons(2, Cons(3, Nil))
in
append a b
```



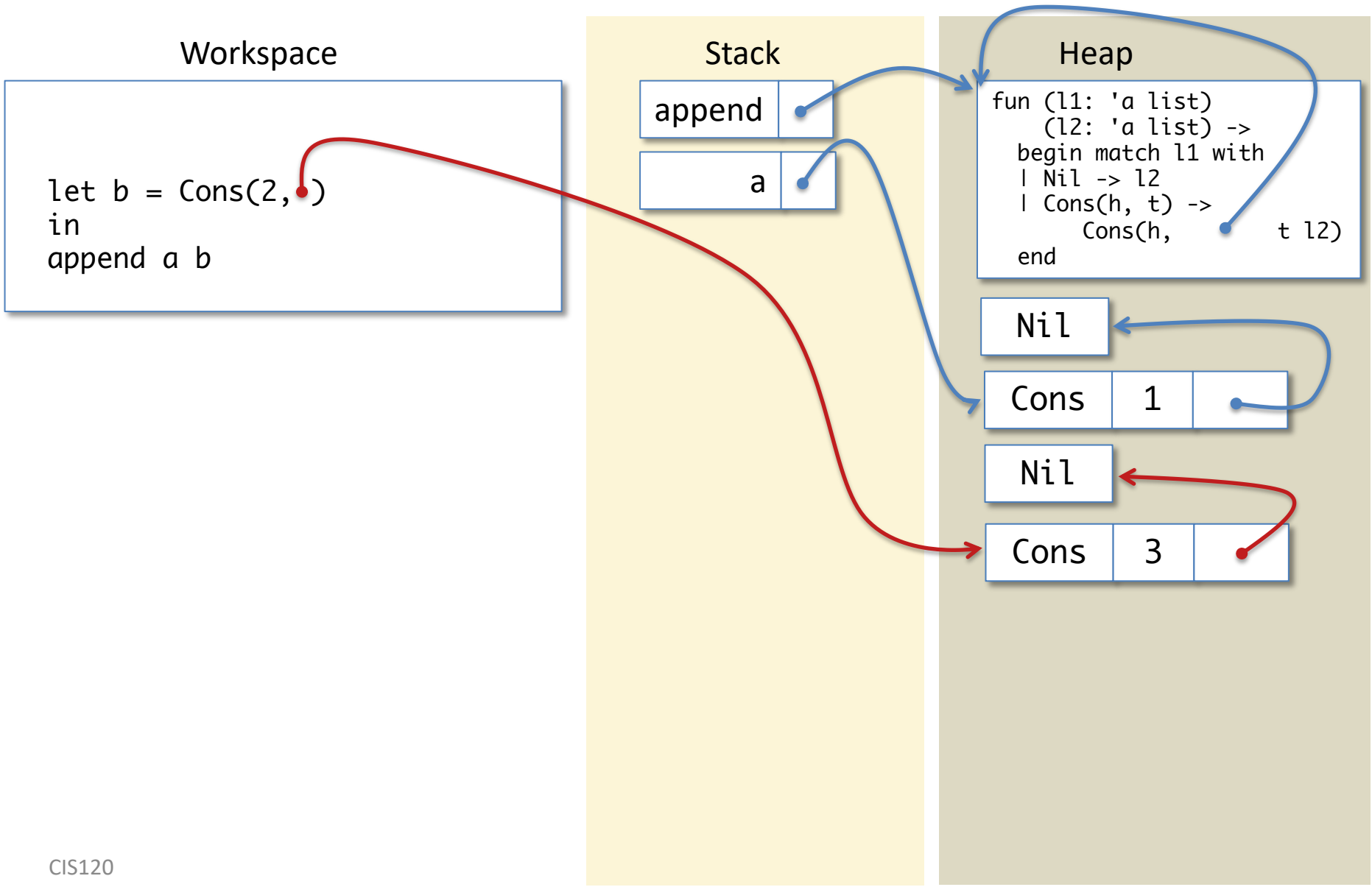
Allocate a Nil cell



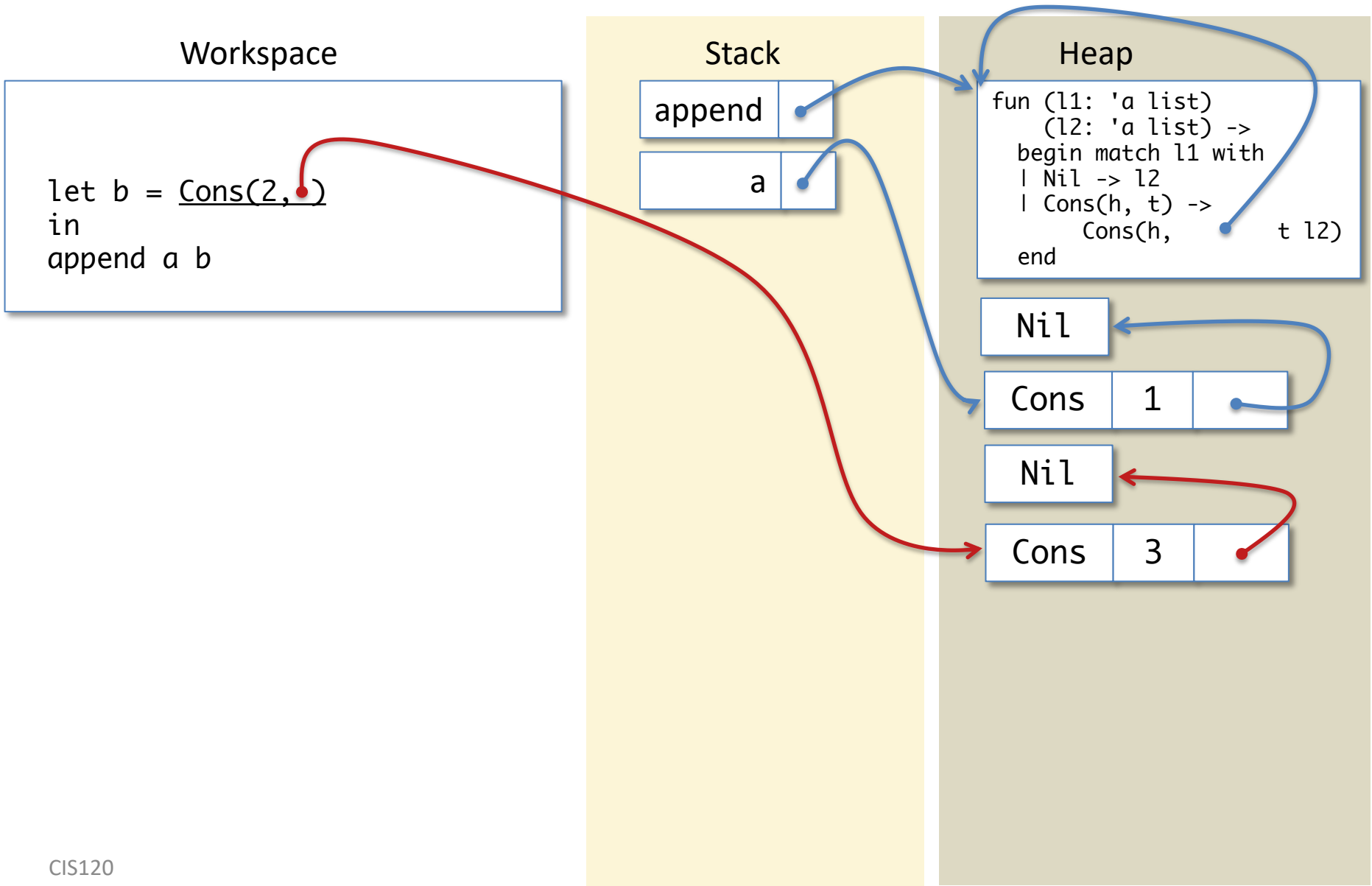
Allocate a Cons cell



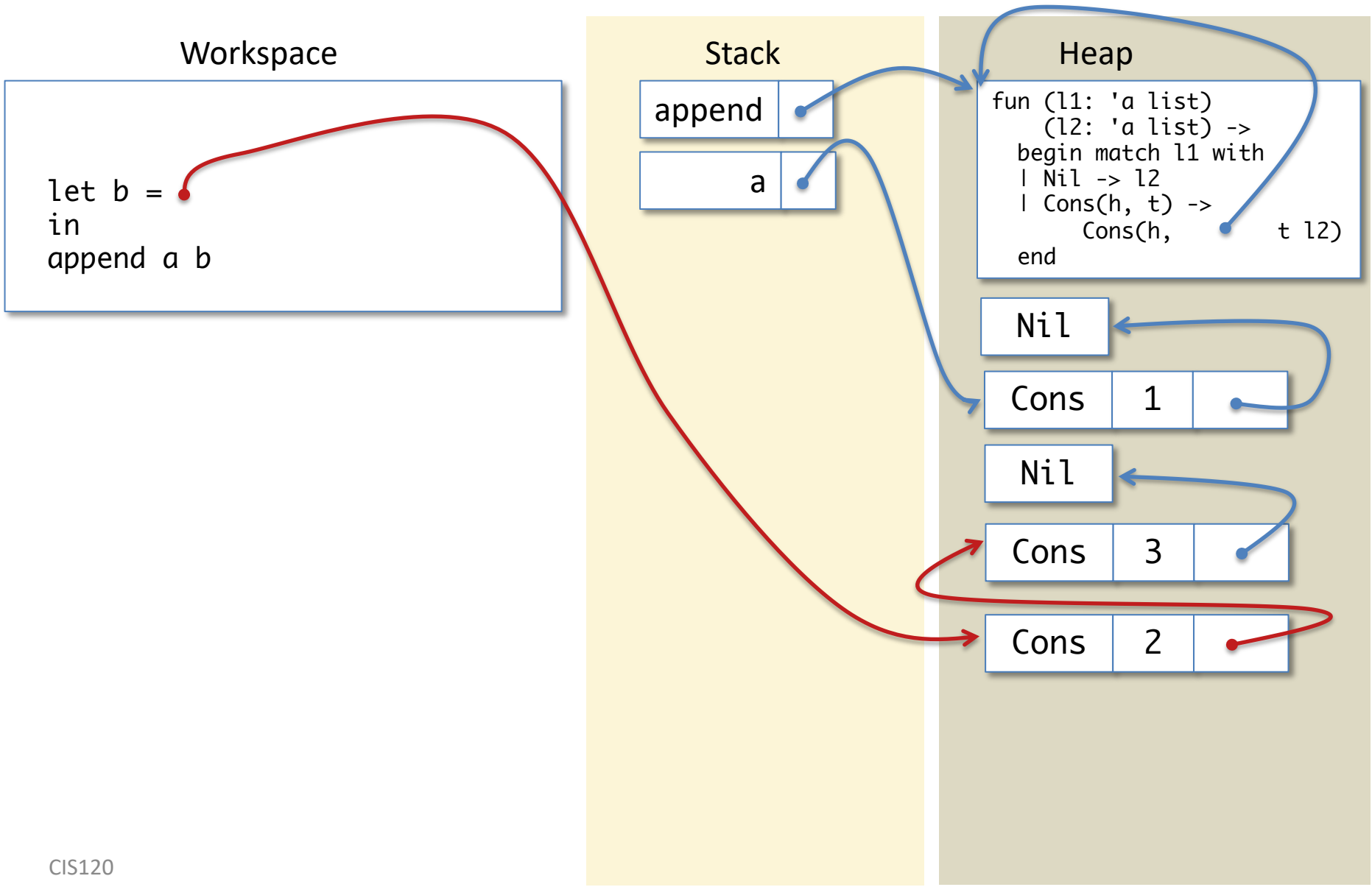
Allocate a Cons cell



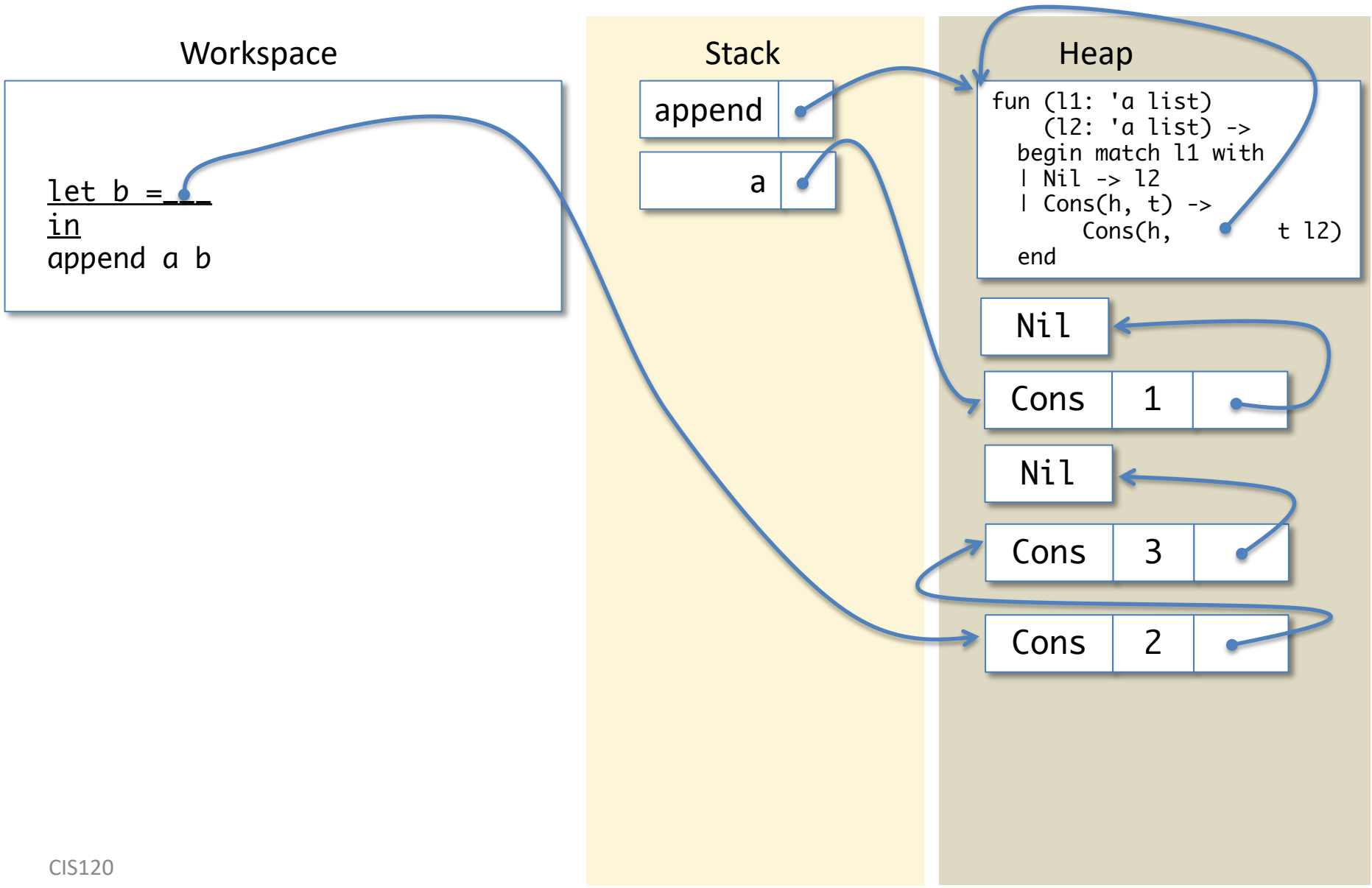
Allocate a Cons cell



Allocate a Cons cell



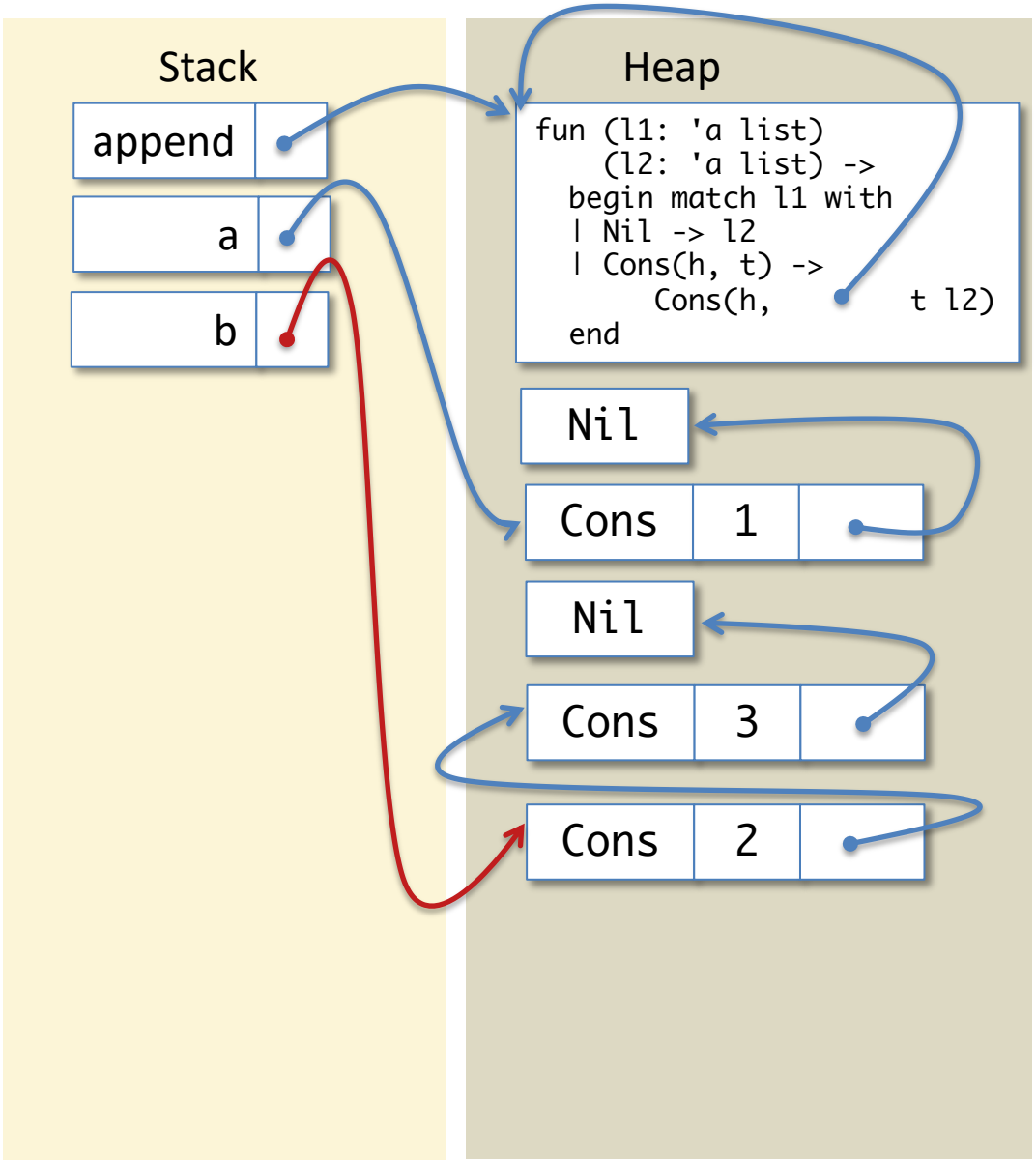
Let Expression



Create a Stack Binding

Workspace

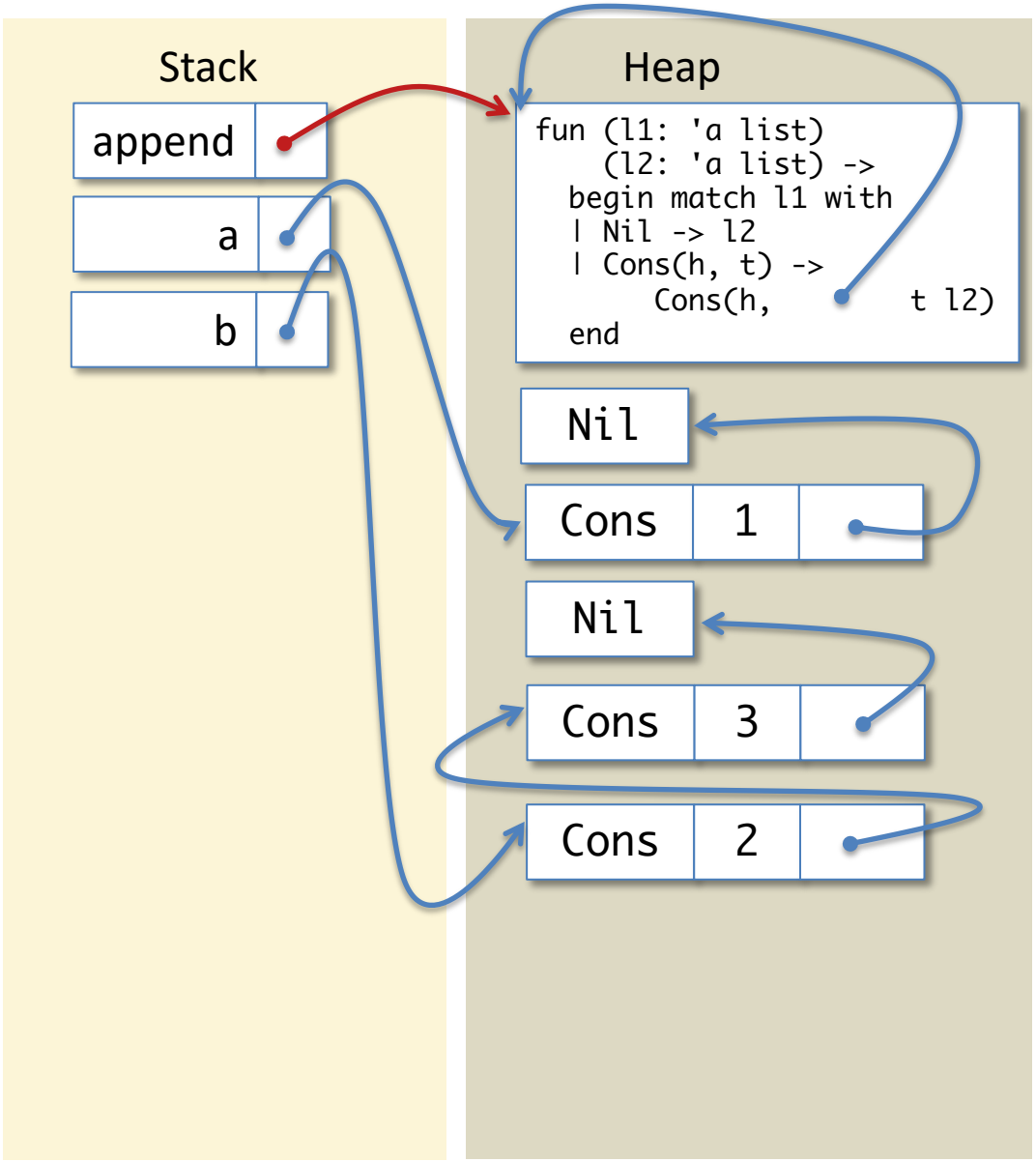
```
append a b
```



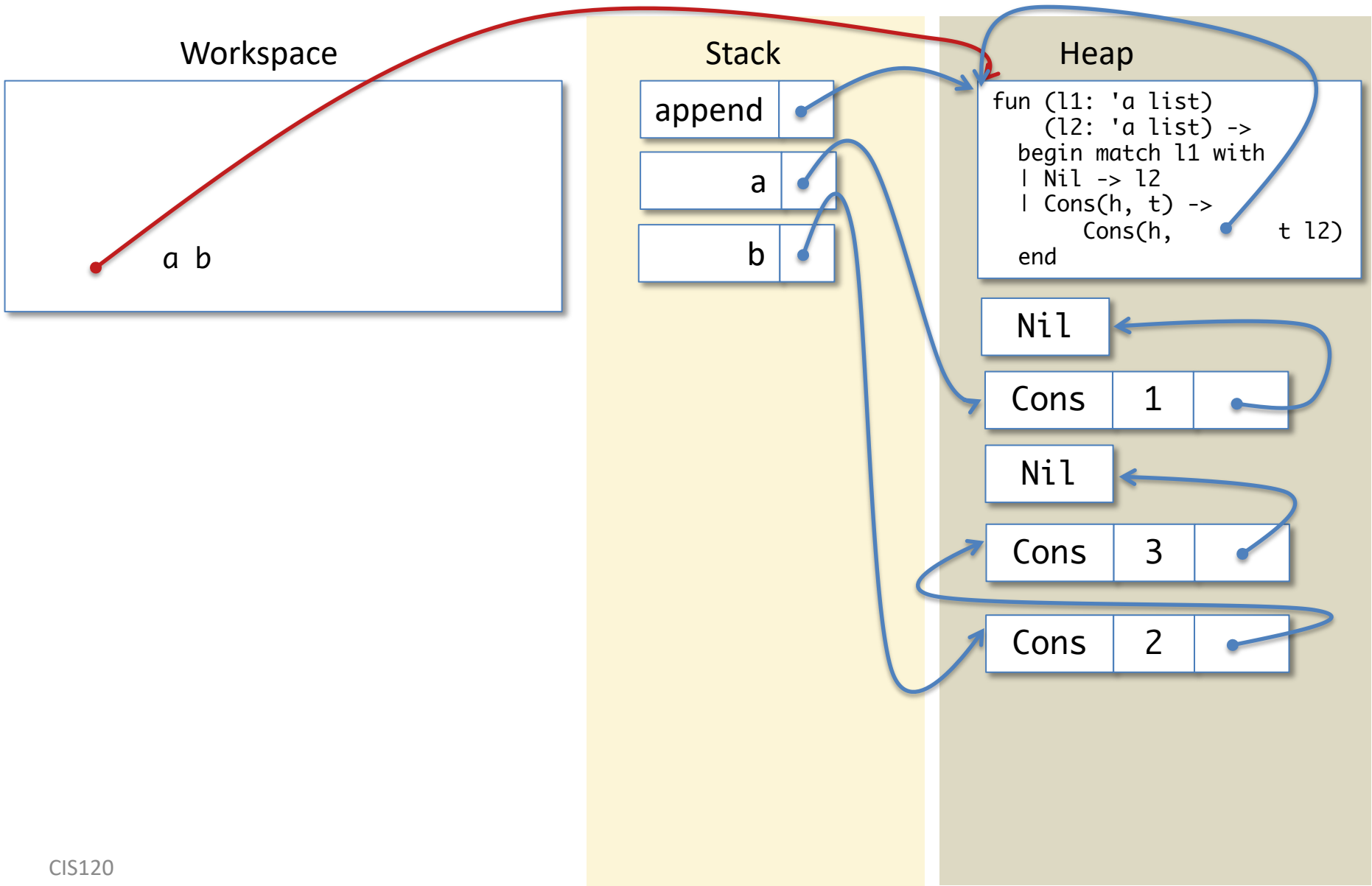
Lookup ‘append’

Workspace

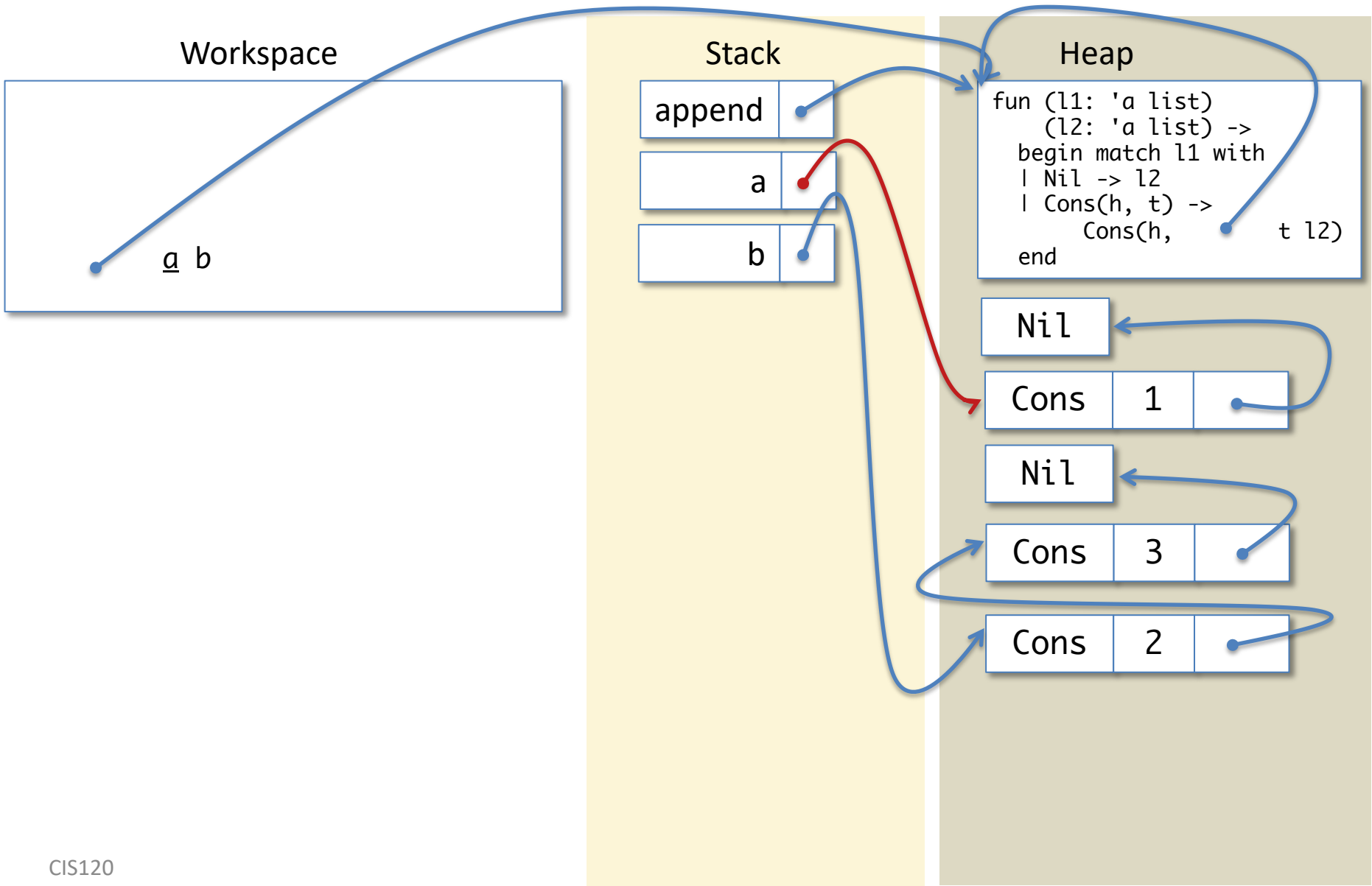
append a b



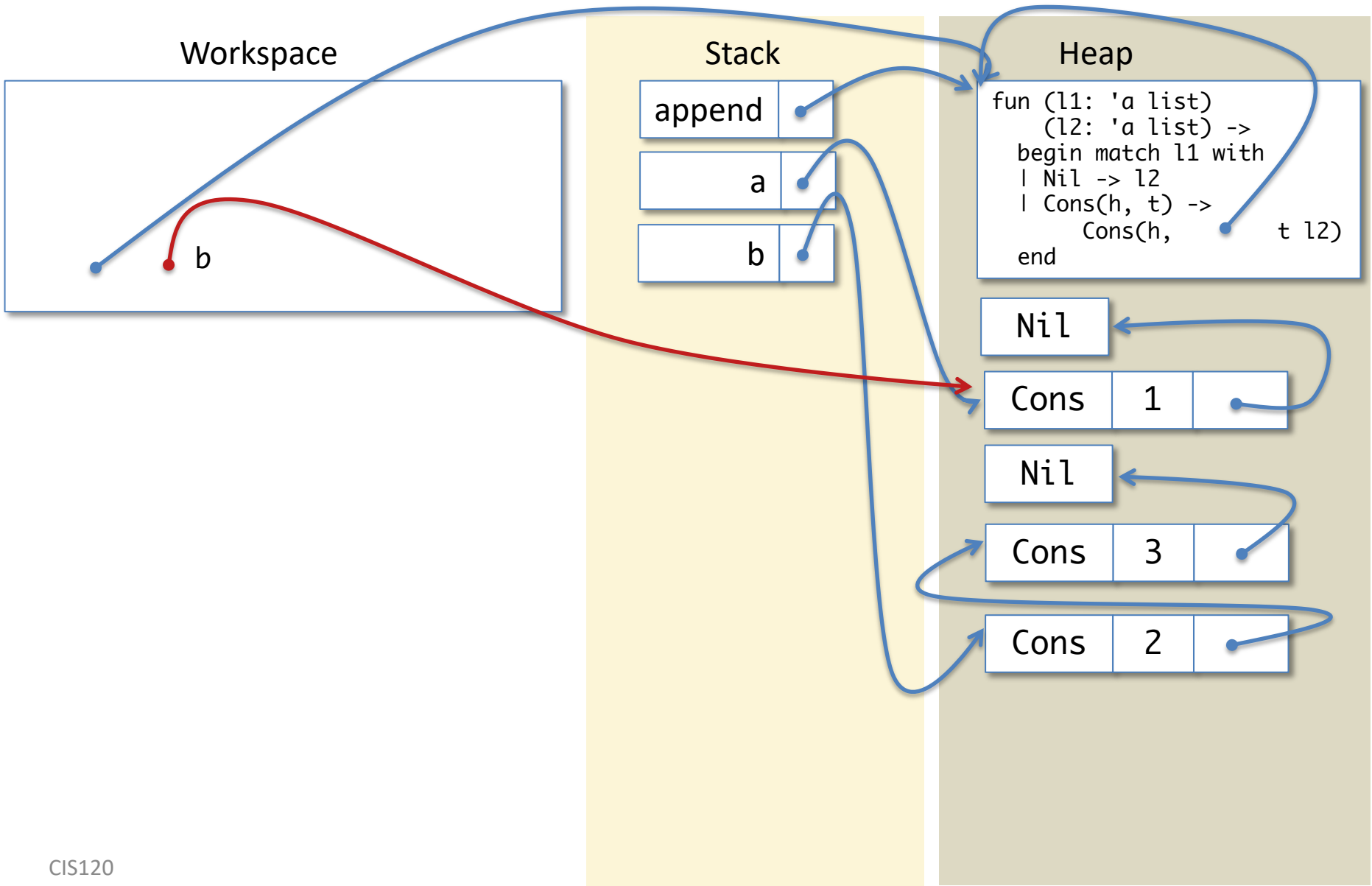
Lookup ‘append’



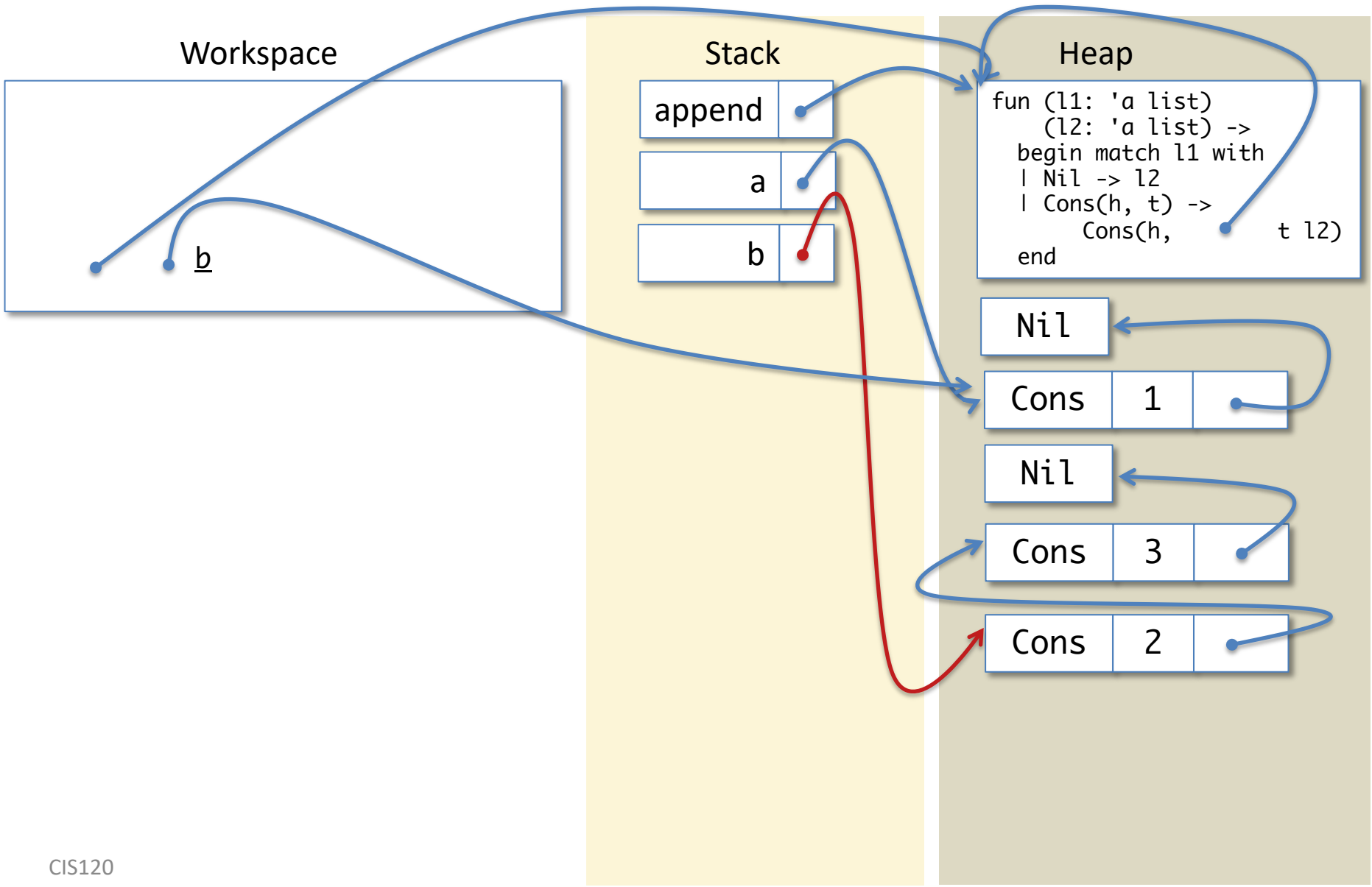
Lookup ‘a’


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

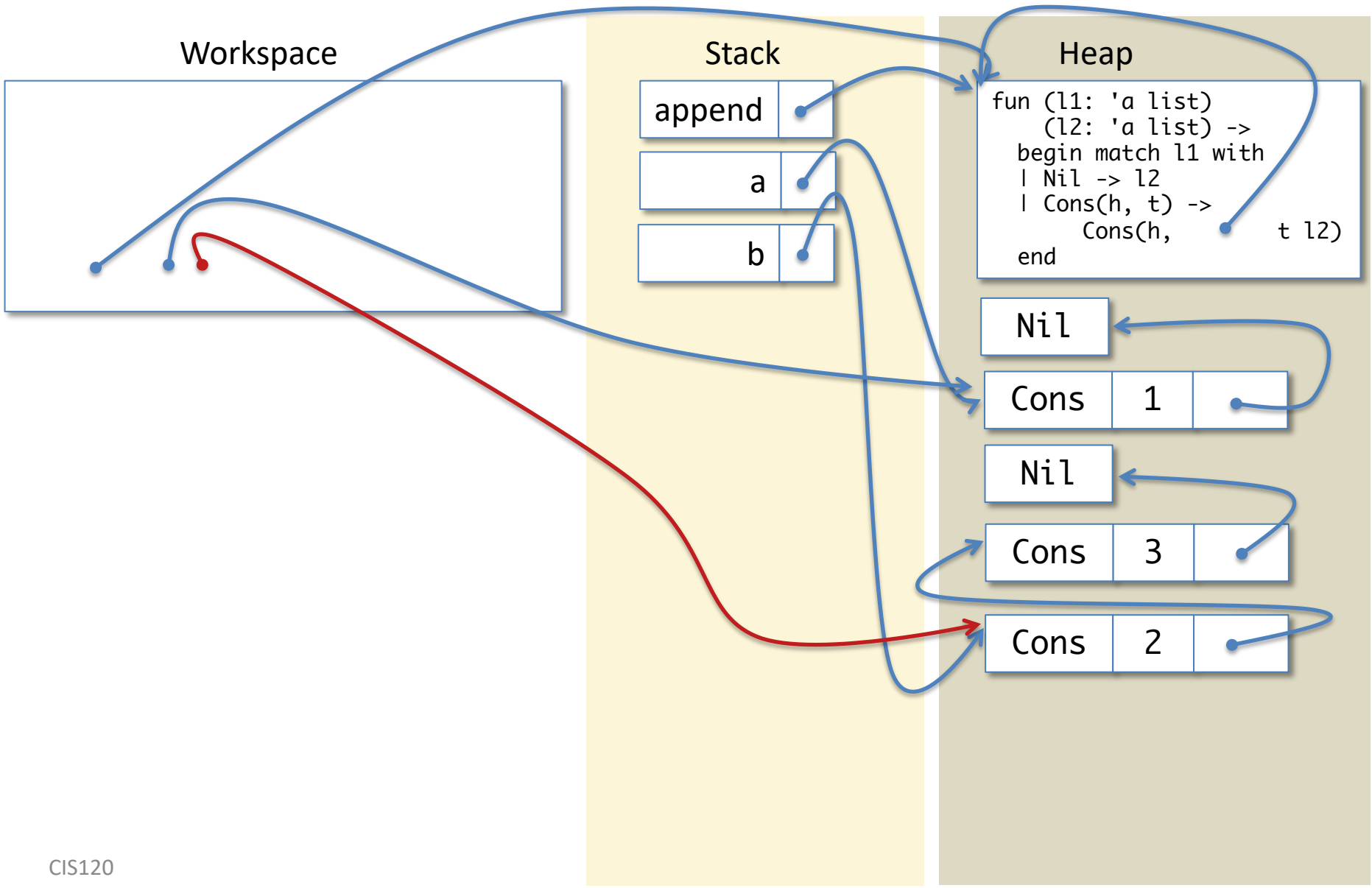
Lookup ‘a’


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

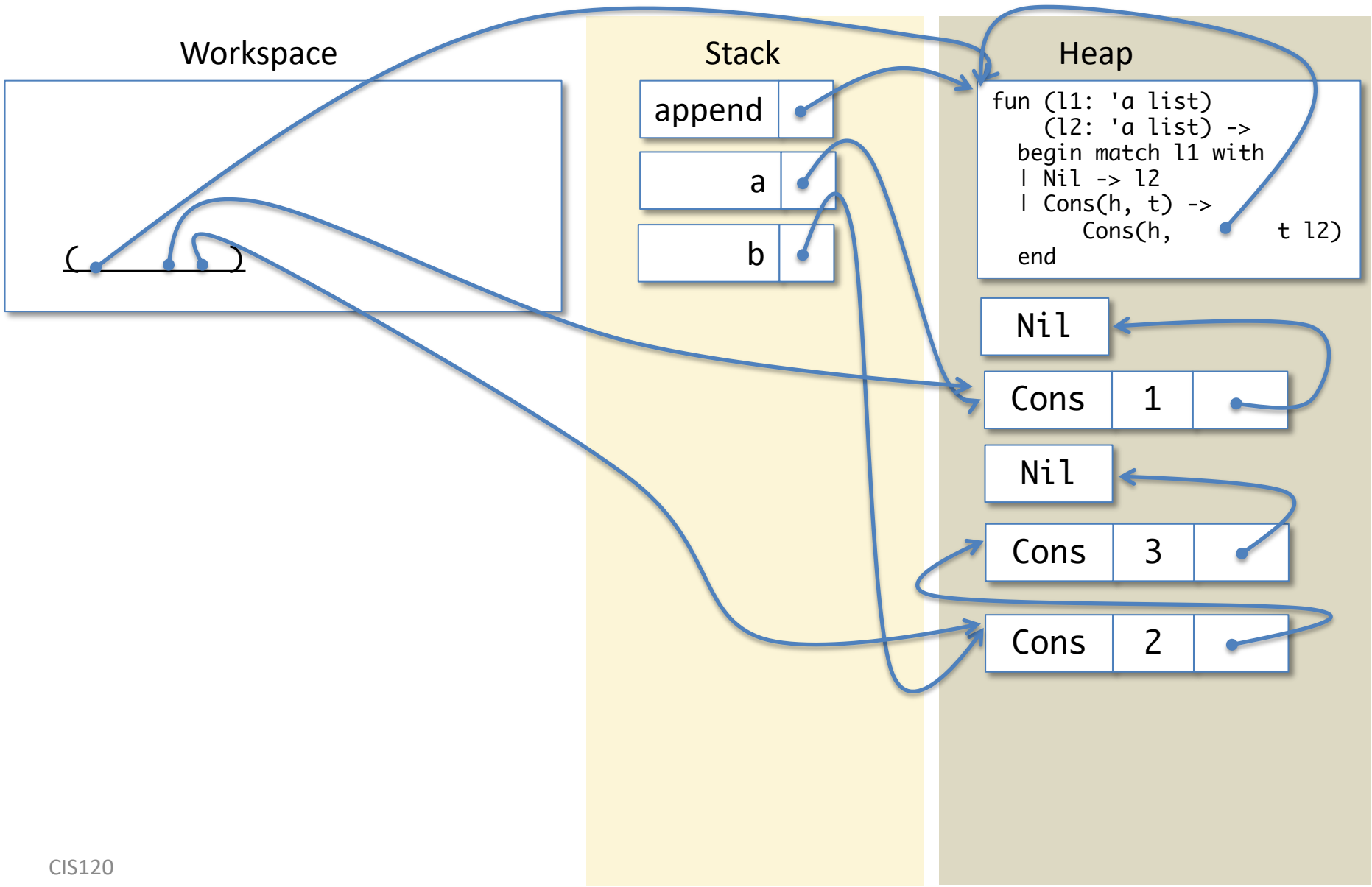
Lookup ‘b’


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

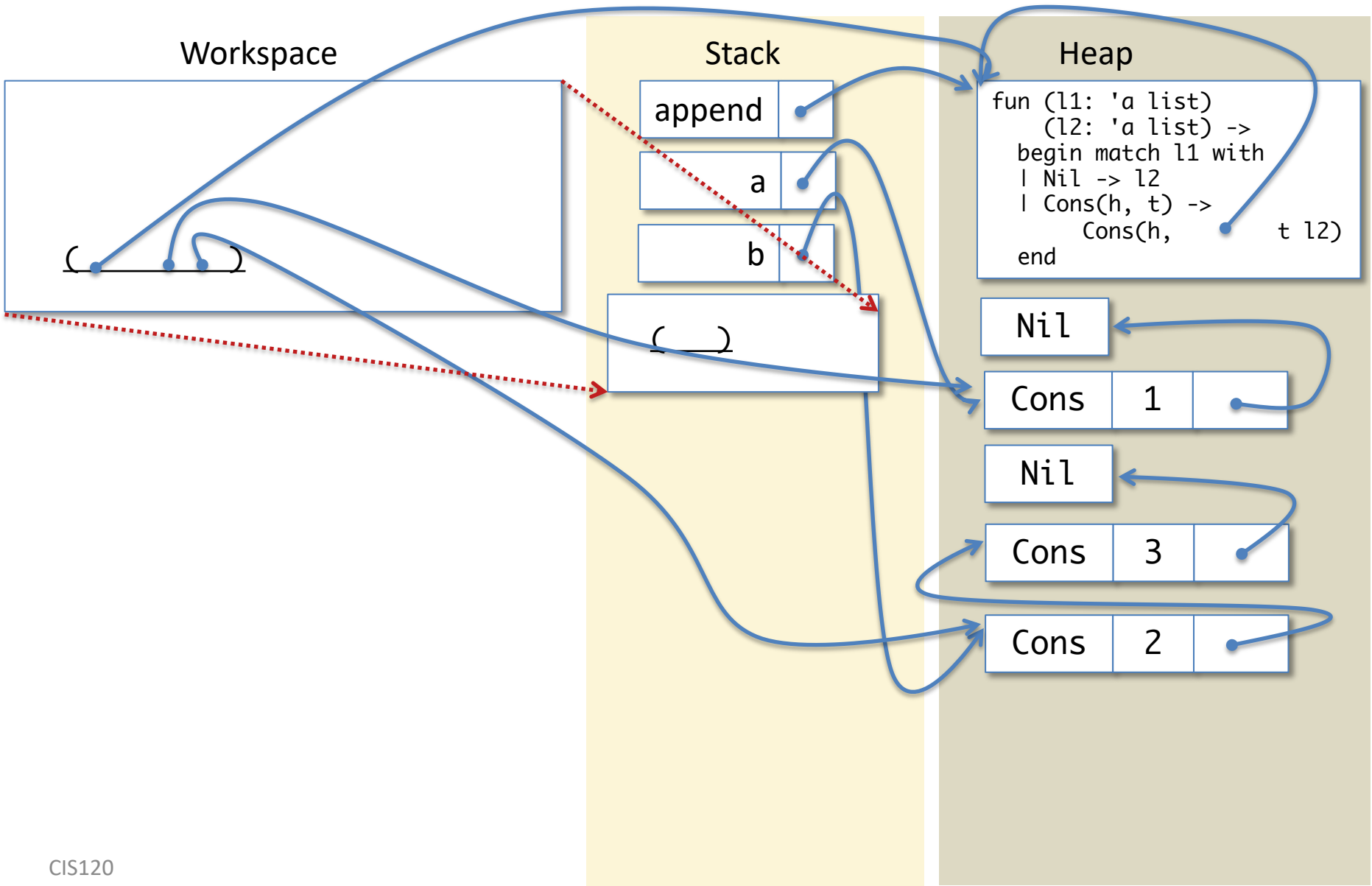
Lookup ‘b’


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

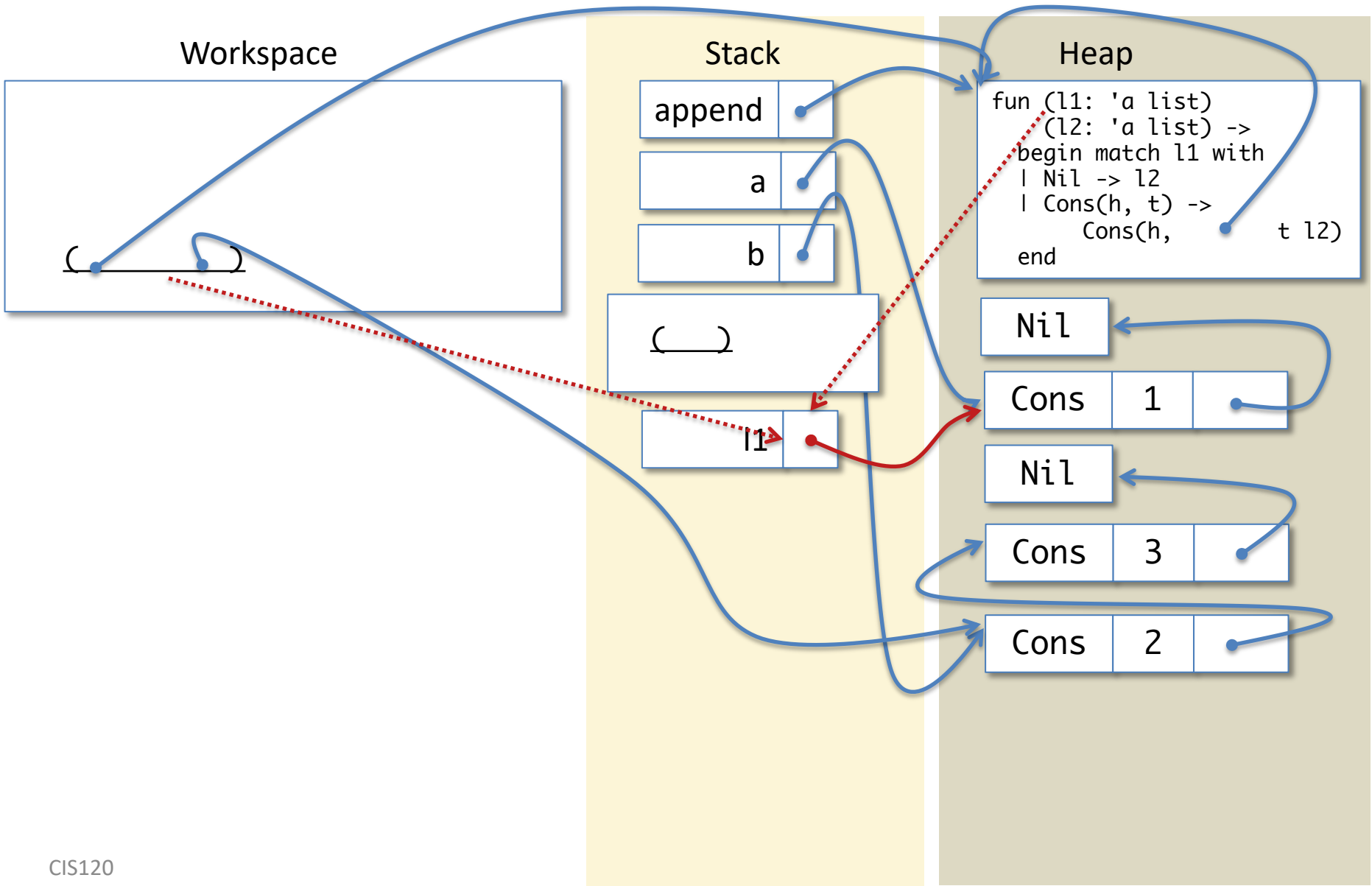
Do the Function call



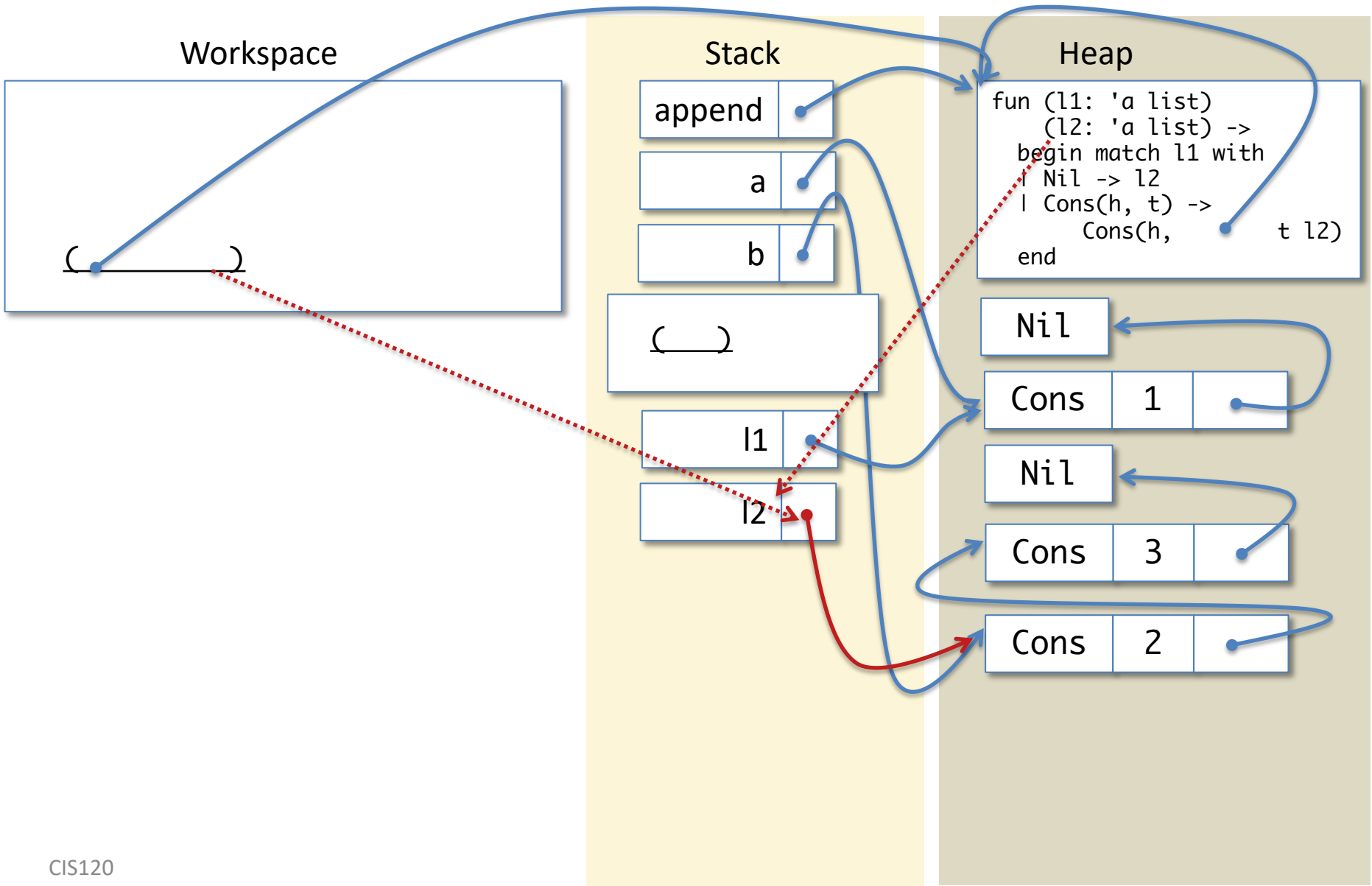
Call (1): Save Workspace



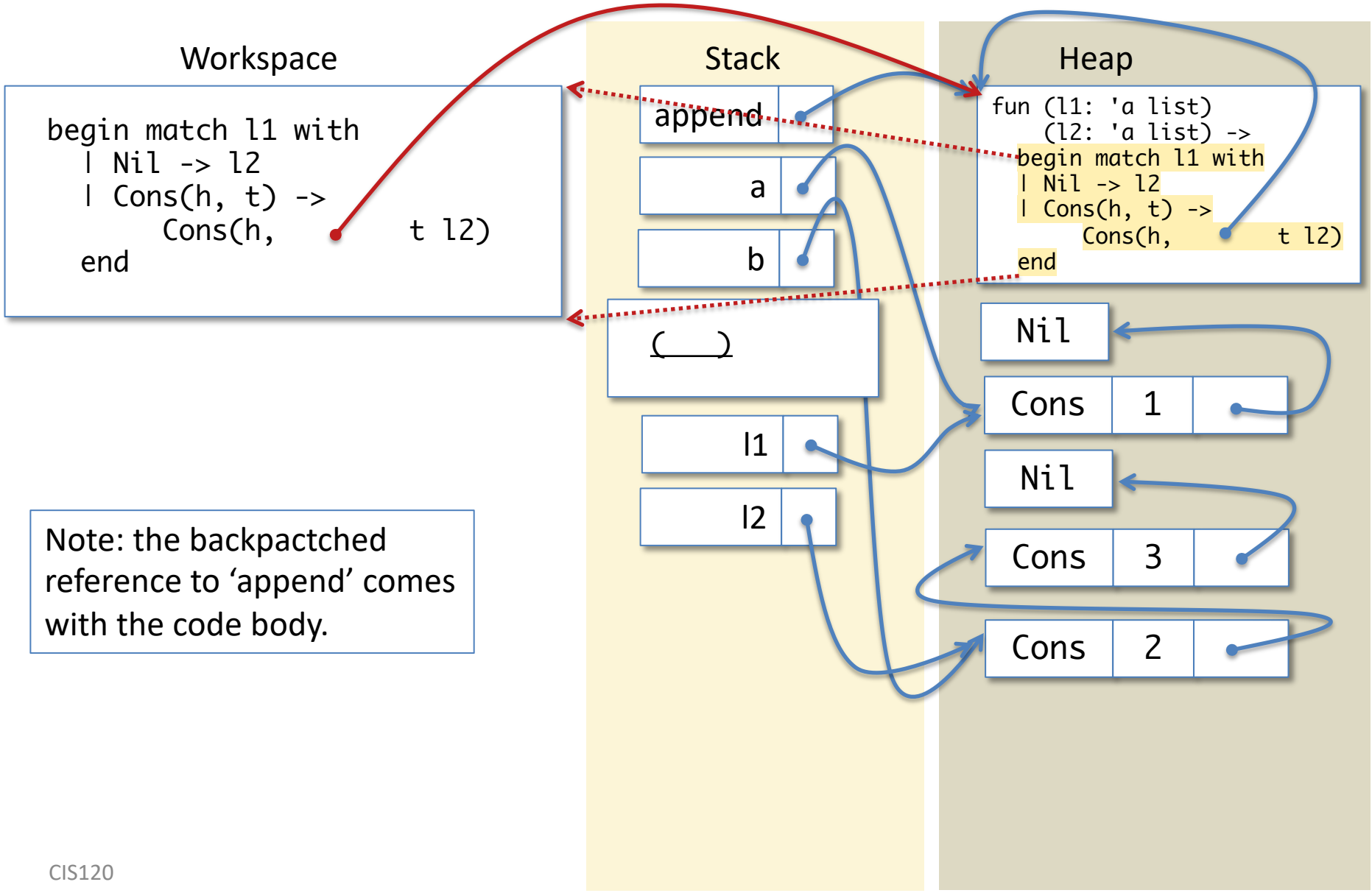
push l1


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

push l2

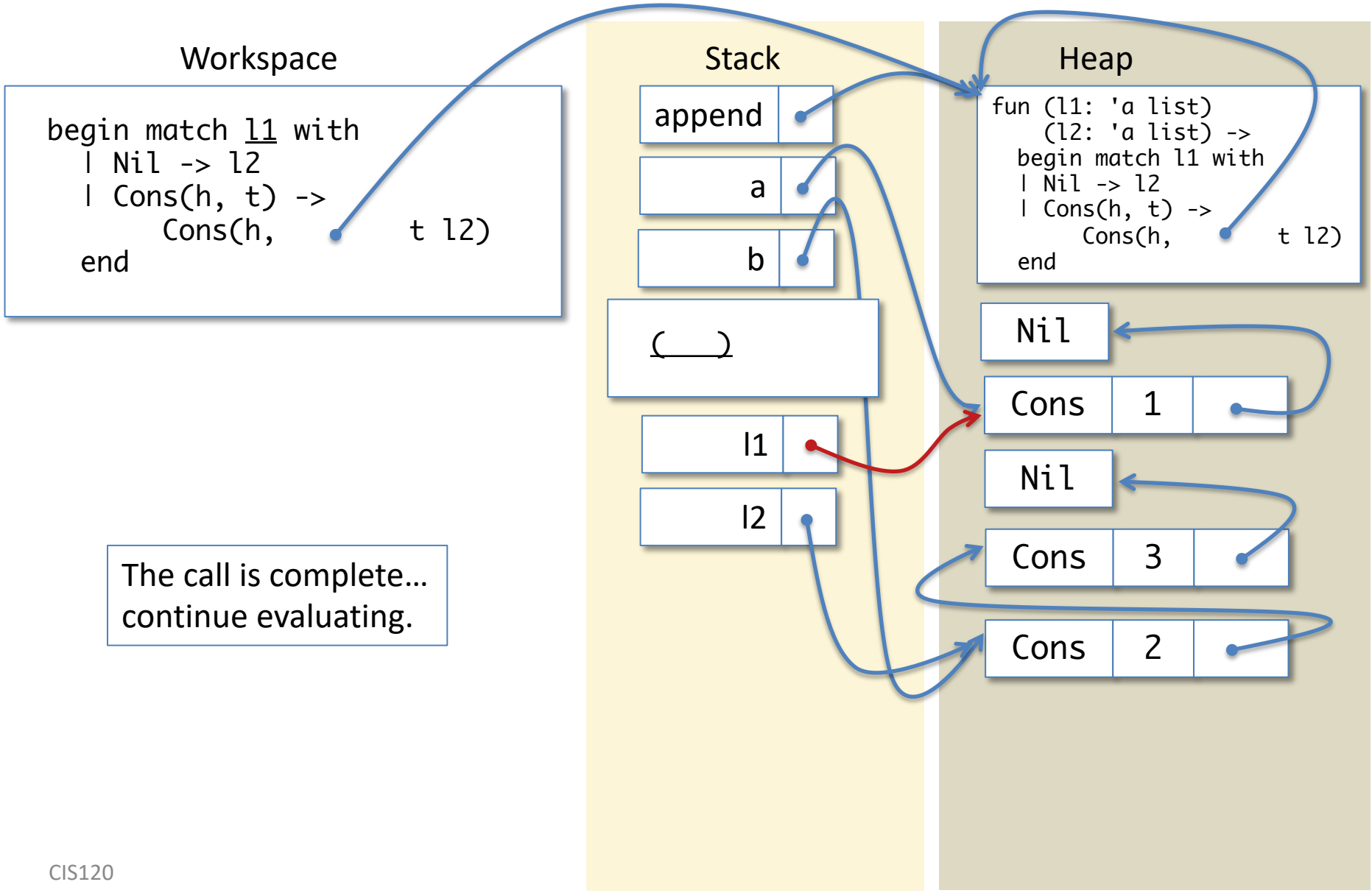

Workspace
Stack
Heap
append
a
b
l1
l2
fun (l1: 'a list)
(l2: 'a list) ->
begin match l1 with
| Nil -> l2
| Cons(h, t) ->
Cons(h, t l2)
end
Nil
Cons 1
Nil
Cons 3
Cons 2

Install Function Body in Workspace



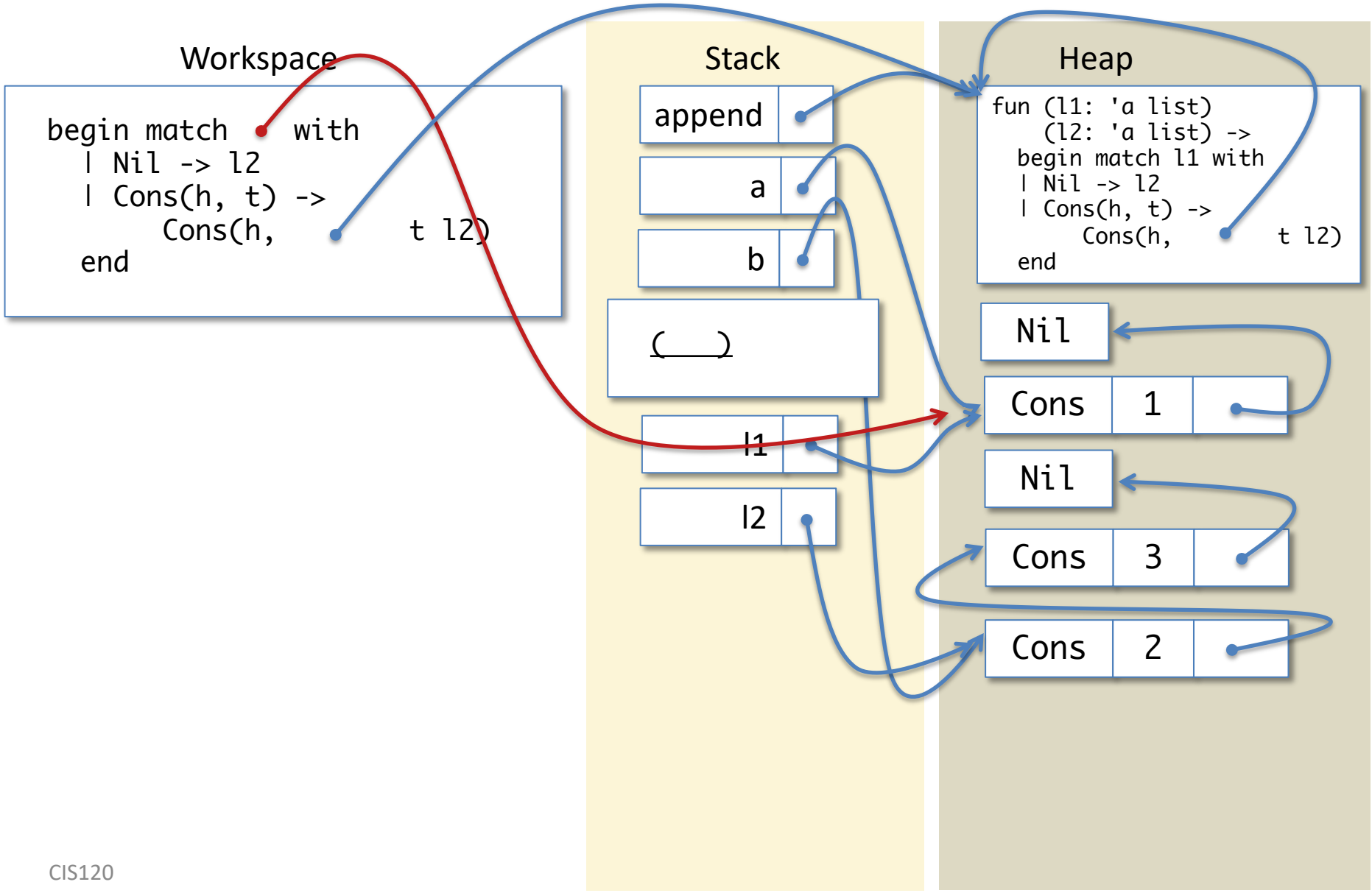
Note: the backpactched reference to 'append' comes with the code body.

Lookup l1

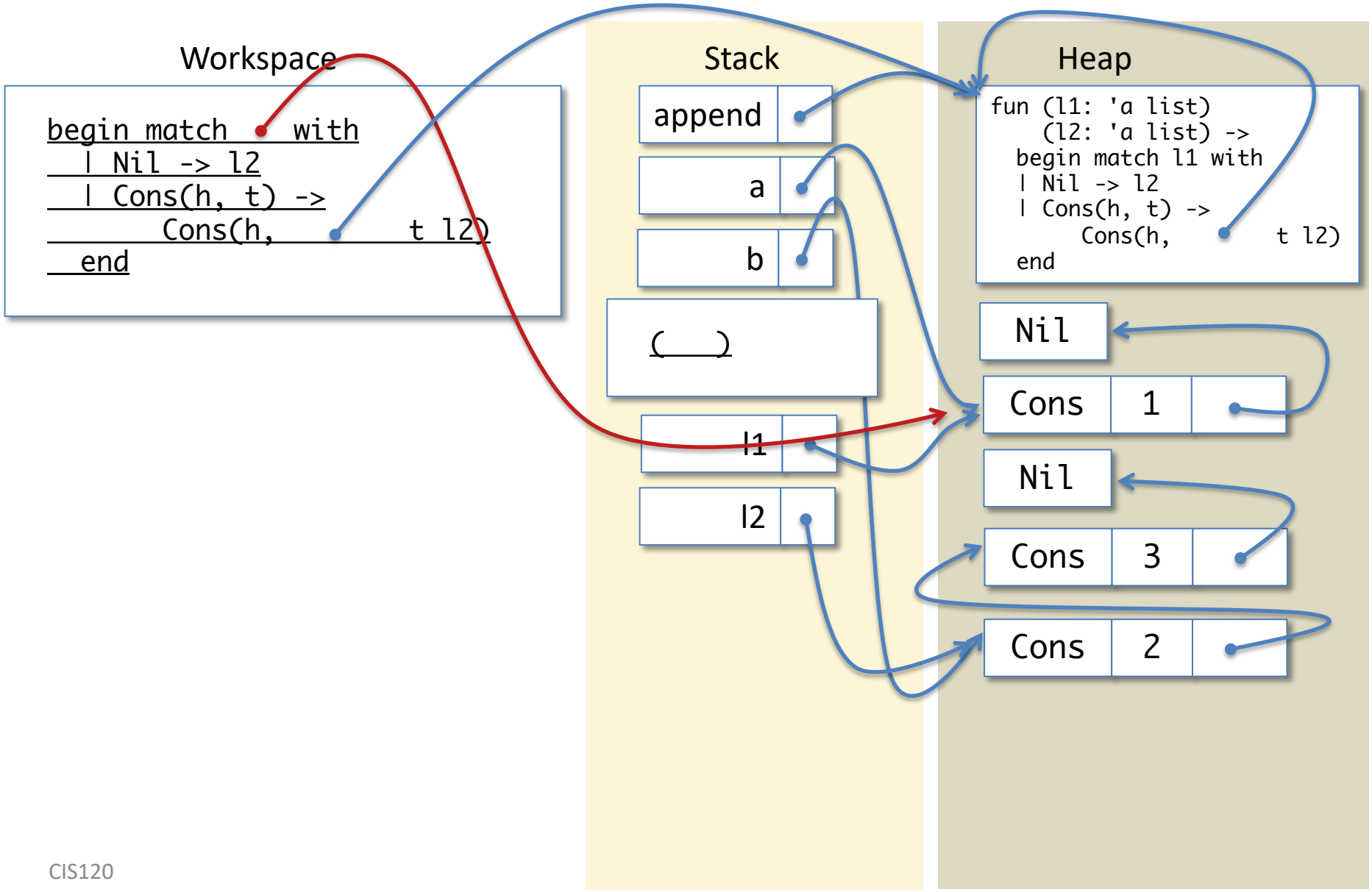


The call is complete…
continue evaluating.

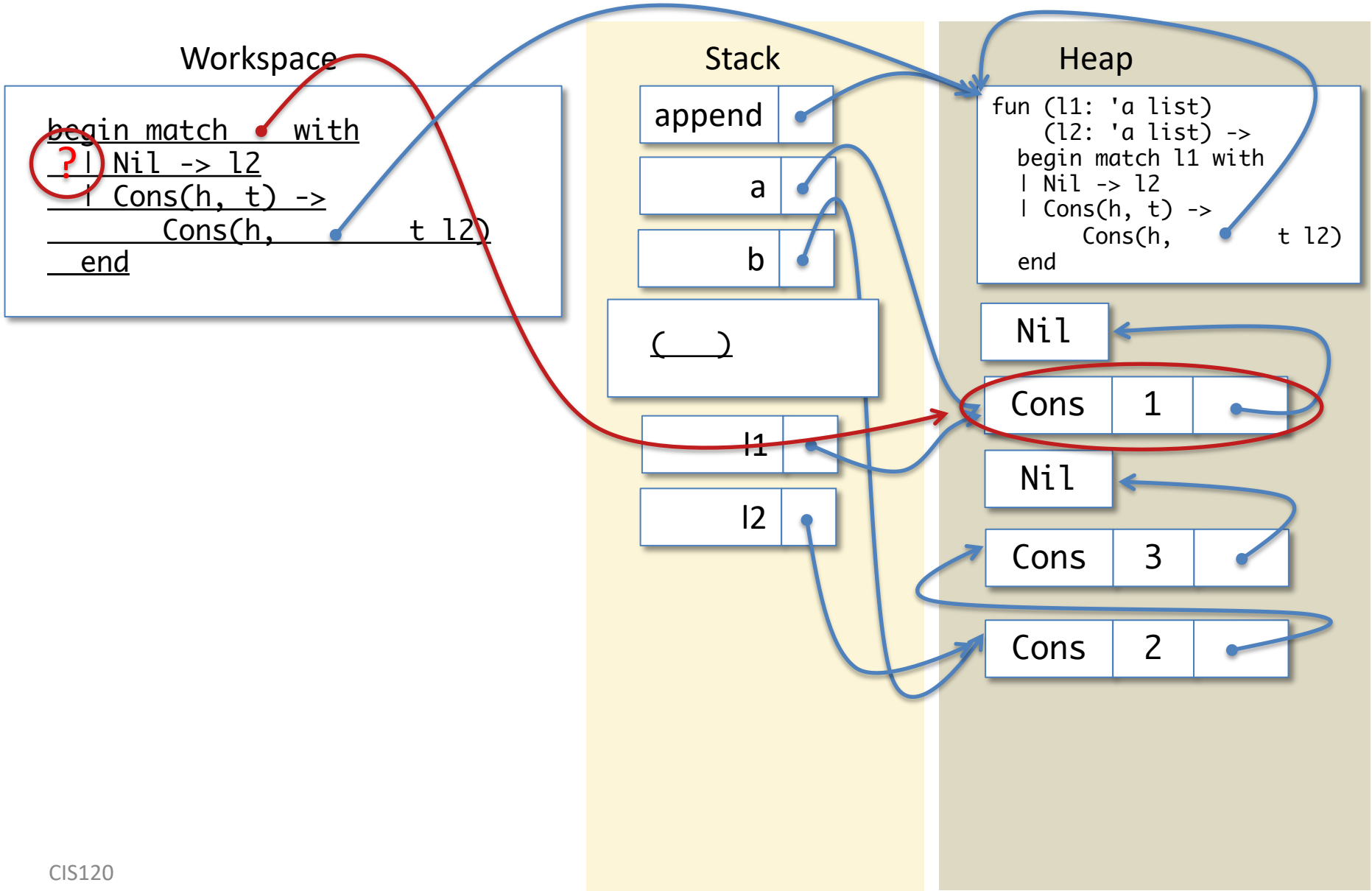
Lookup l1


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

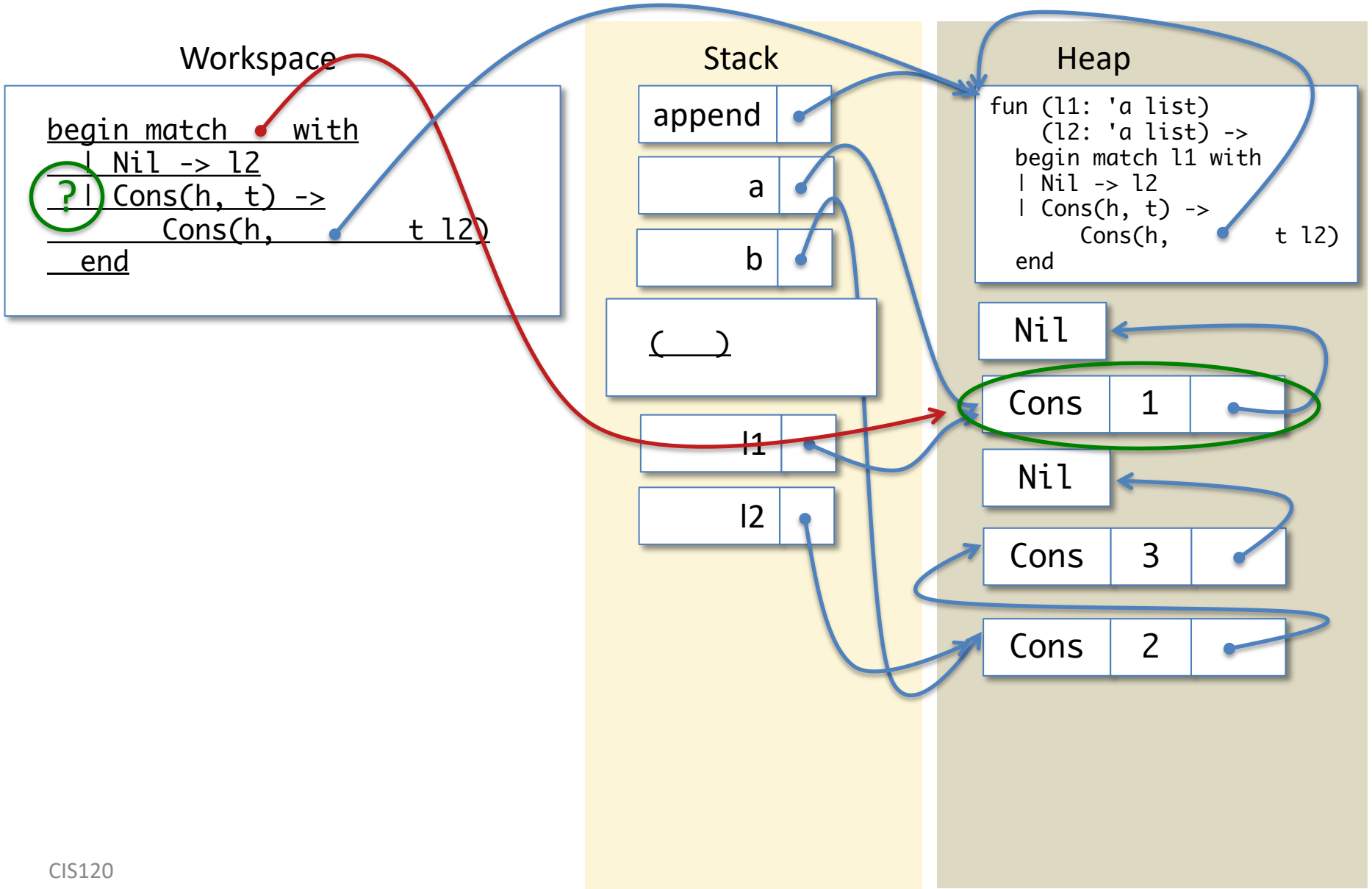
Match Expression



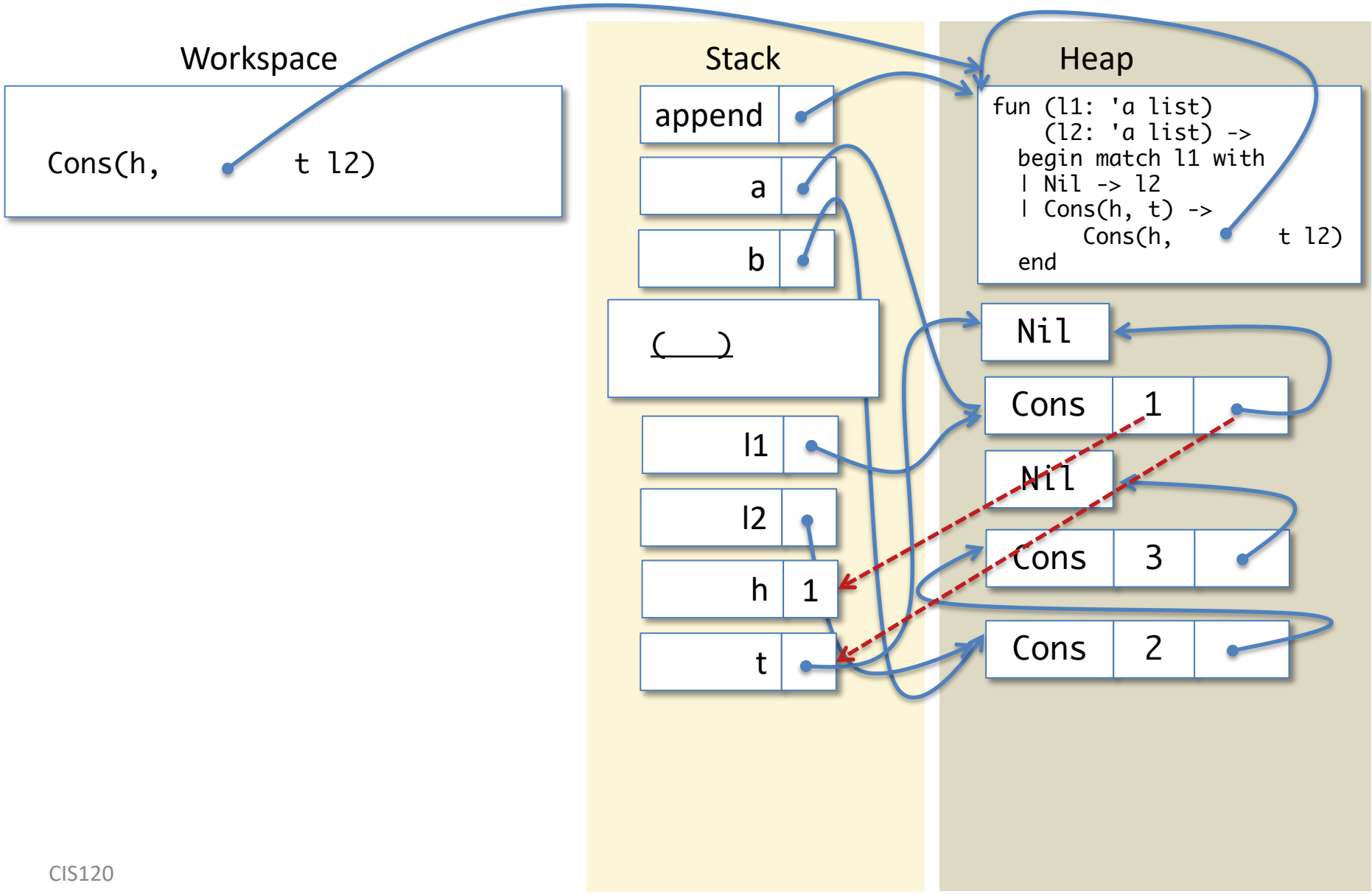
Nil case Doesn't Match



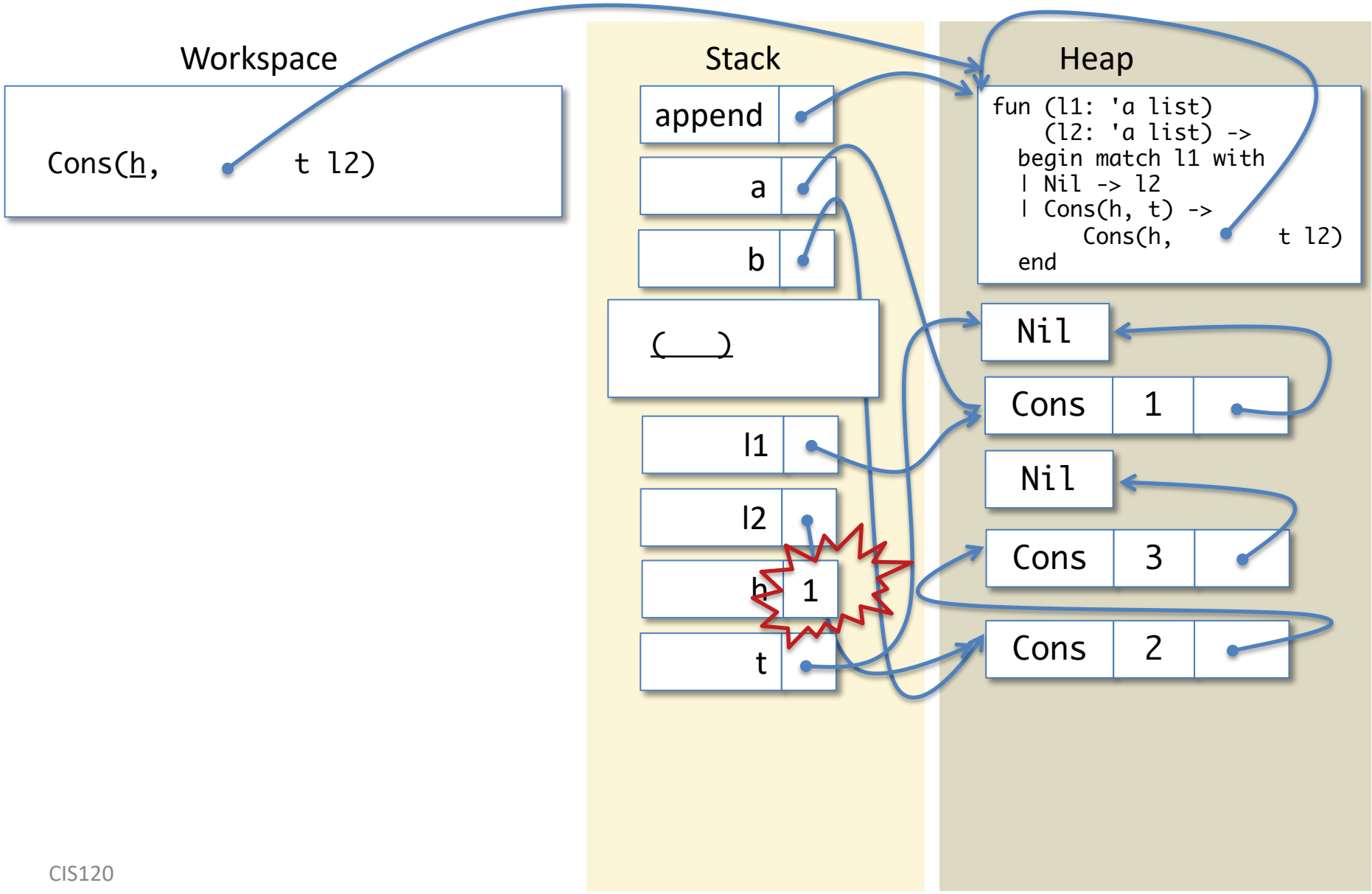
Cons case *Does* Match



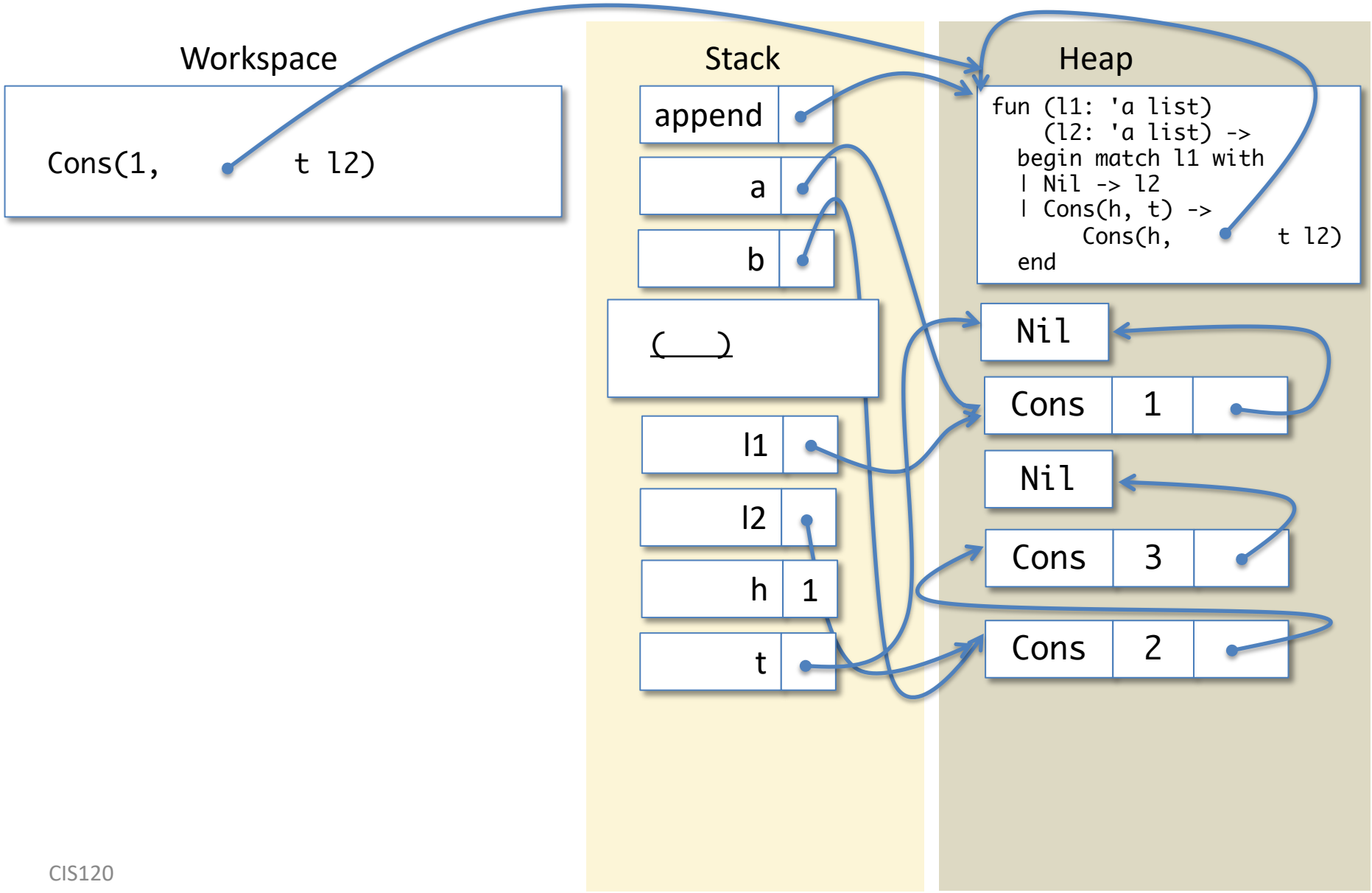
Simplify the Branch: push h, t



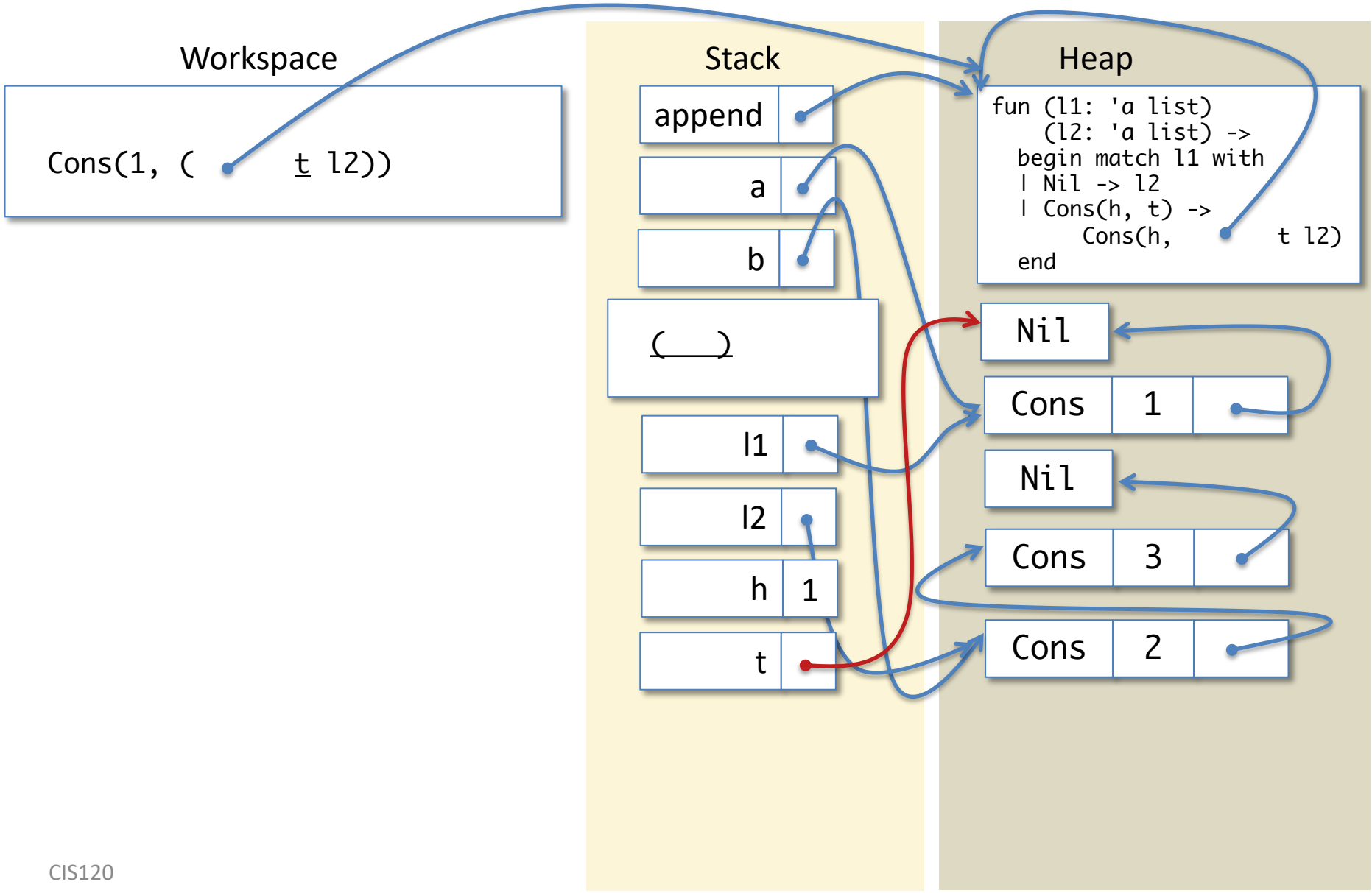
Lookup ‘h’


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

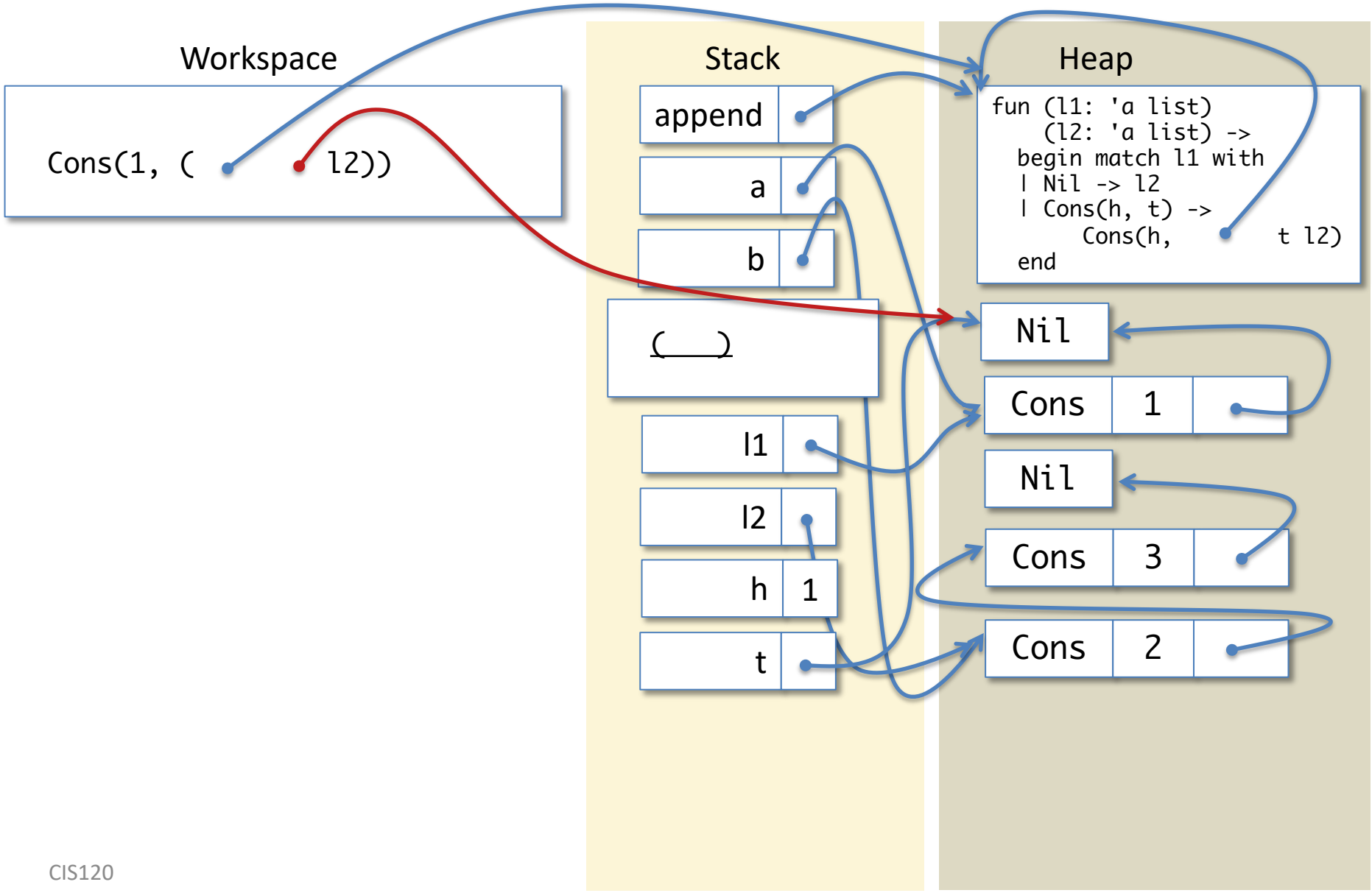
Lookup ‘h’


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

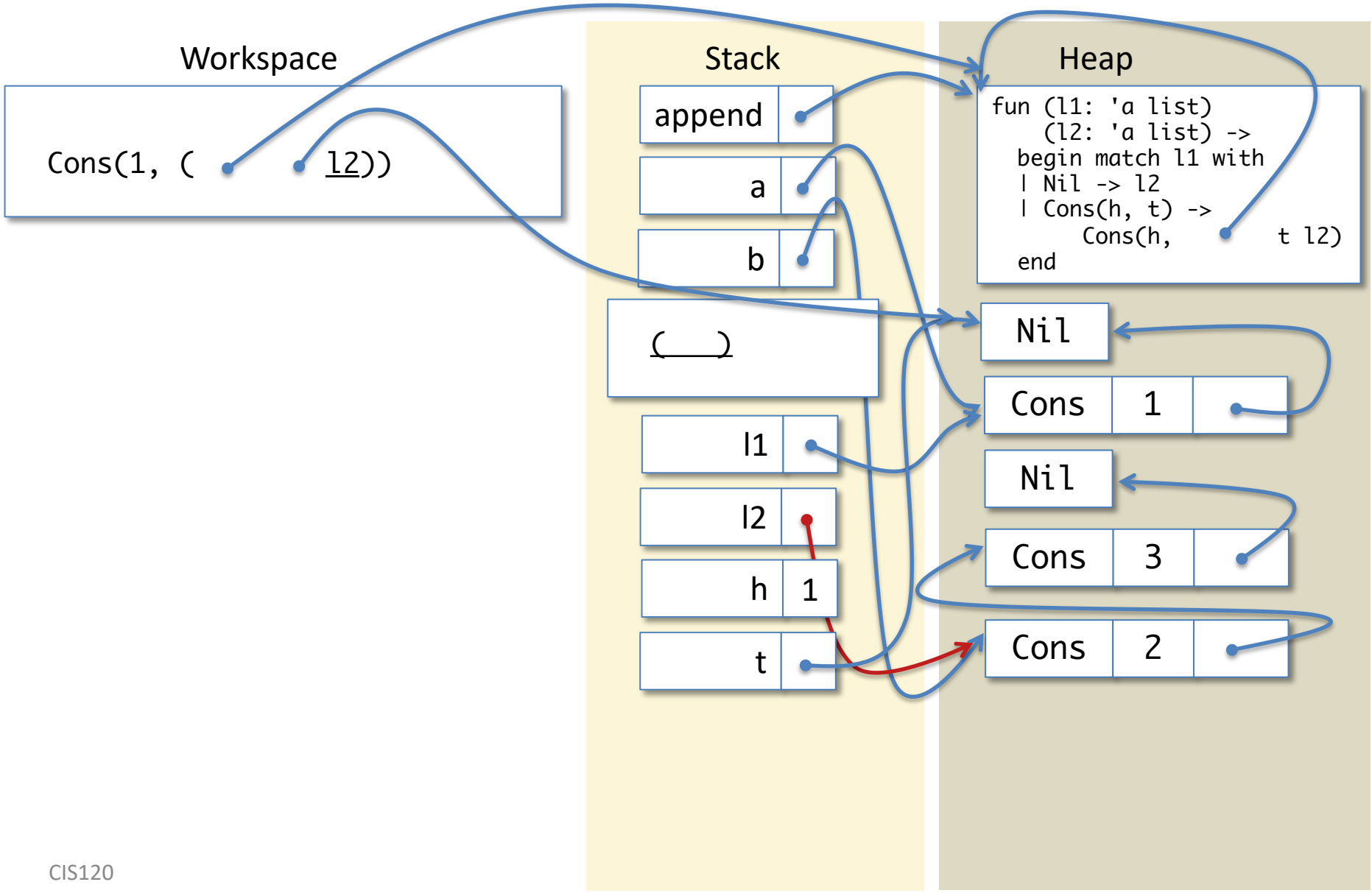
Lookup ‘t’


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

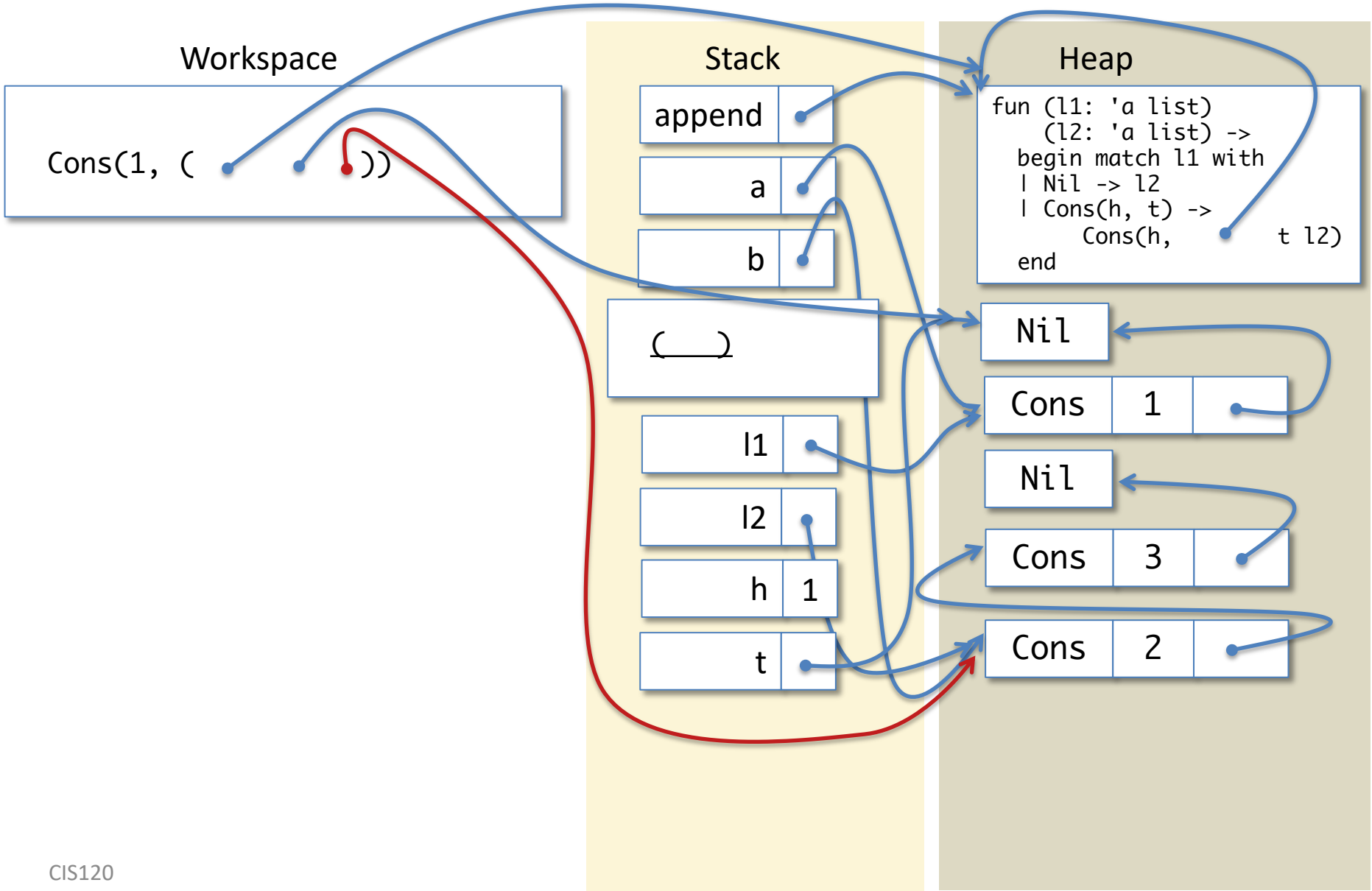
Lookup ‘t’


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

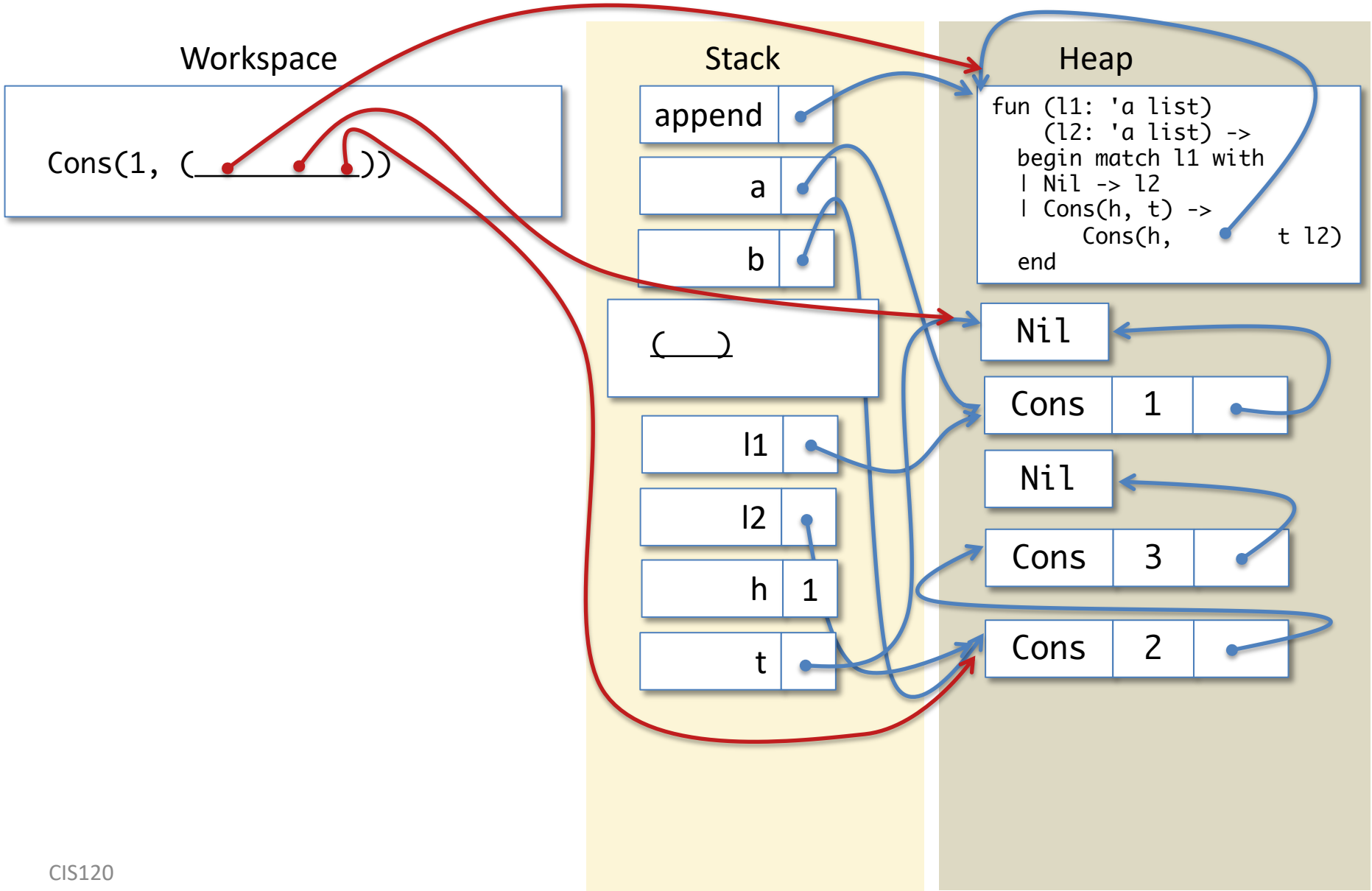
Lookup ‘l2’


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

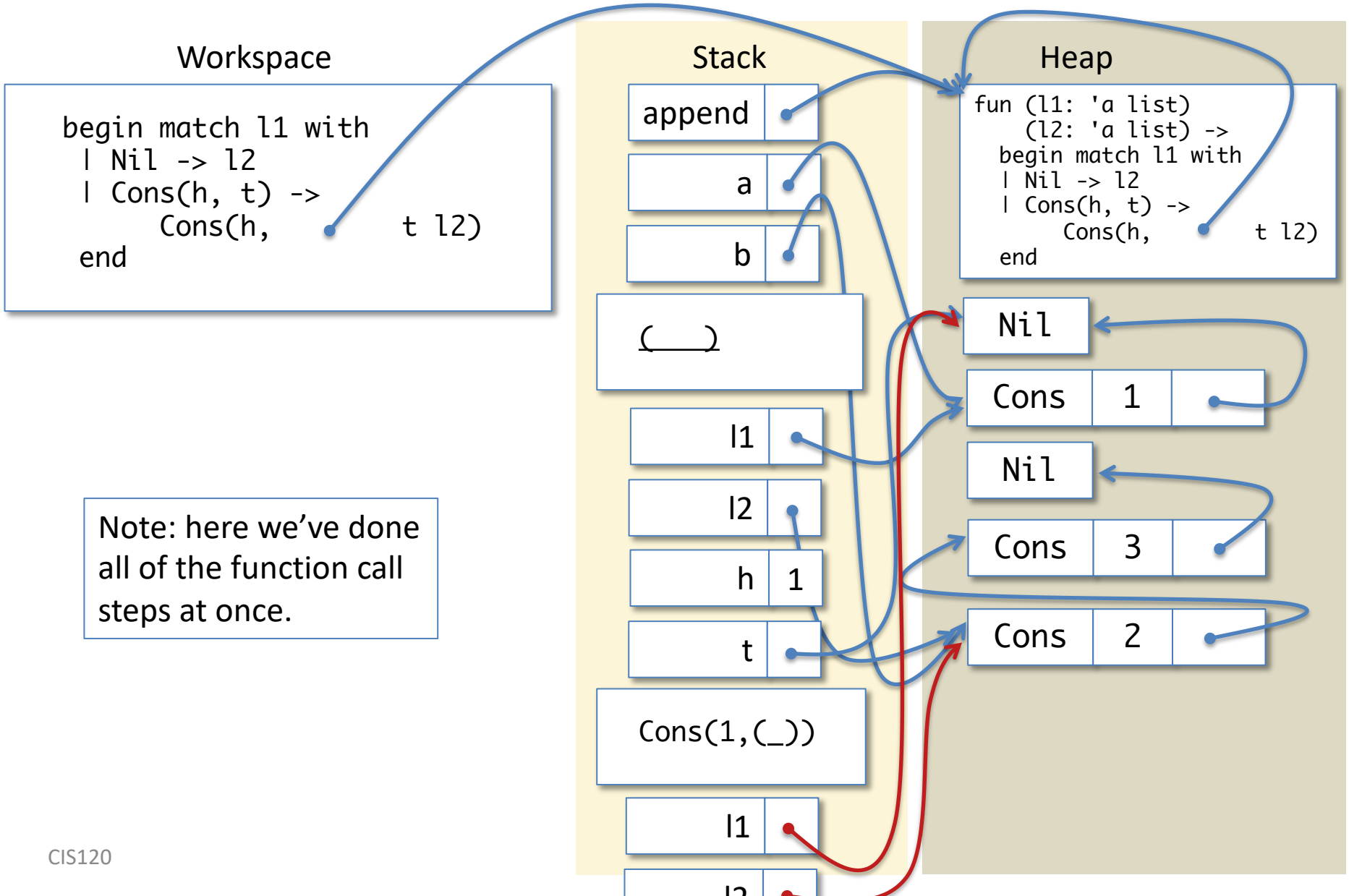
Lookup ‘l2’


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

Do the Function Call

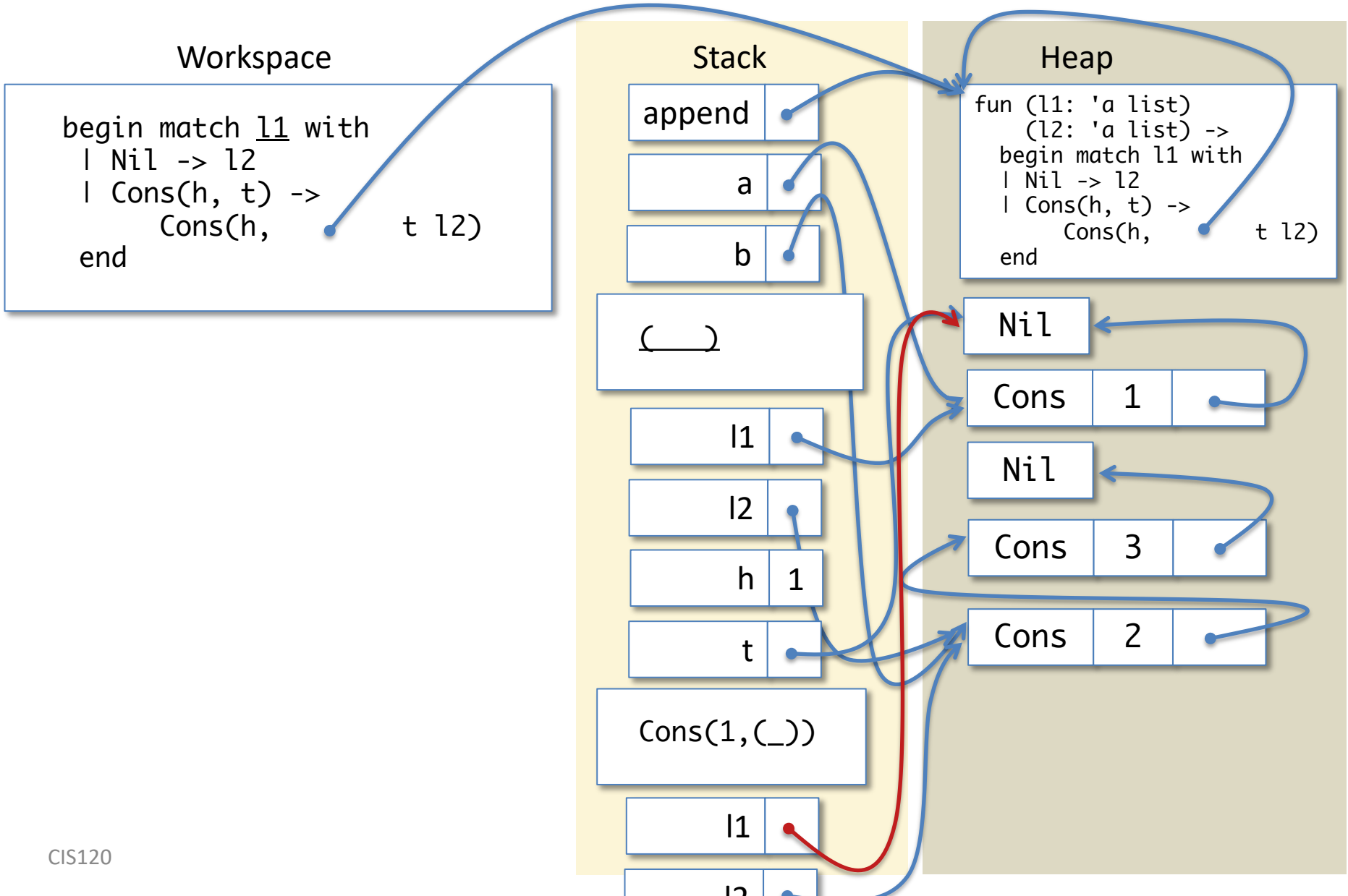


Save the Workspace; push l1, l2

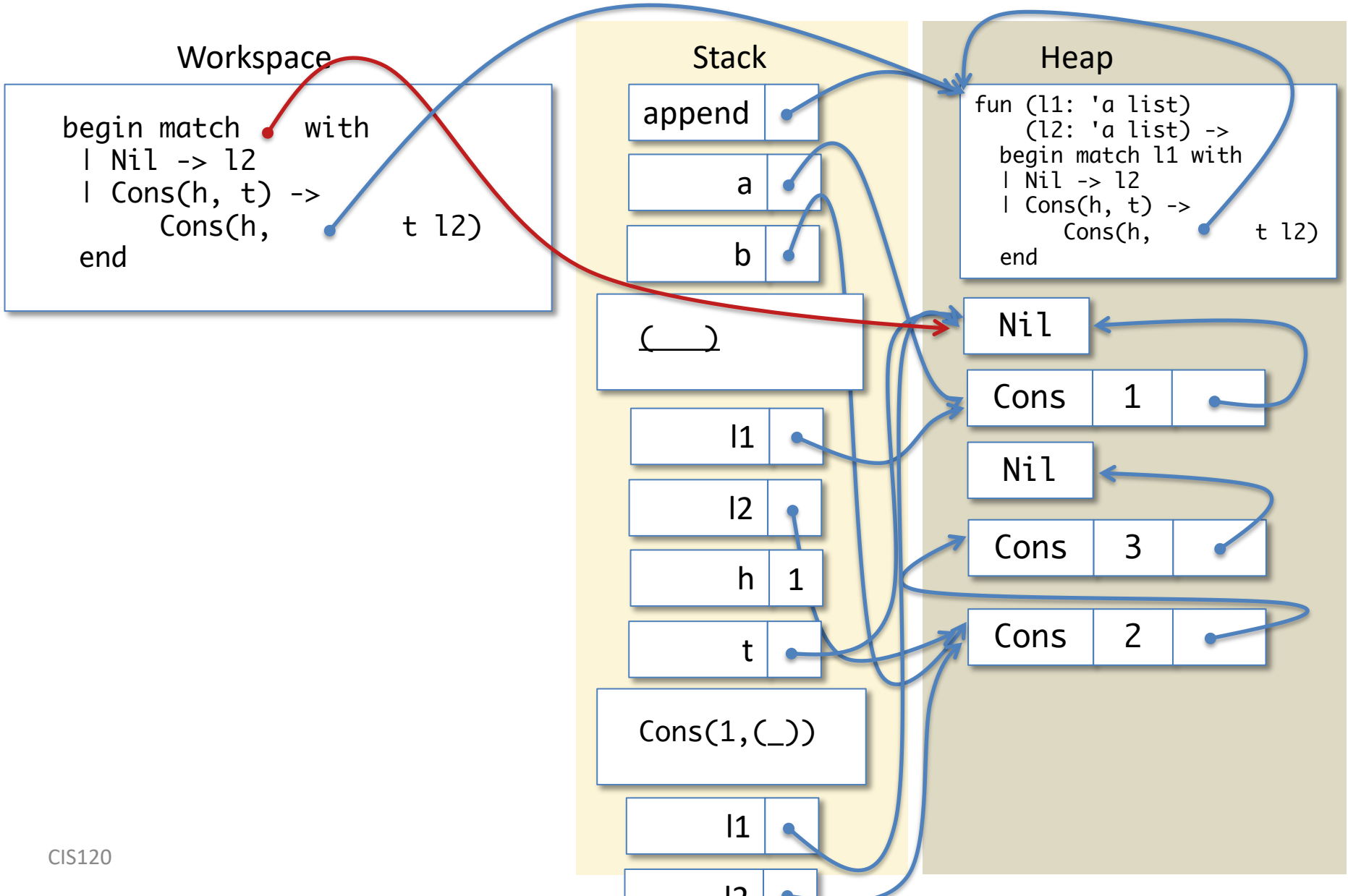


Note: here we've done all of the function call steps at once.

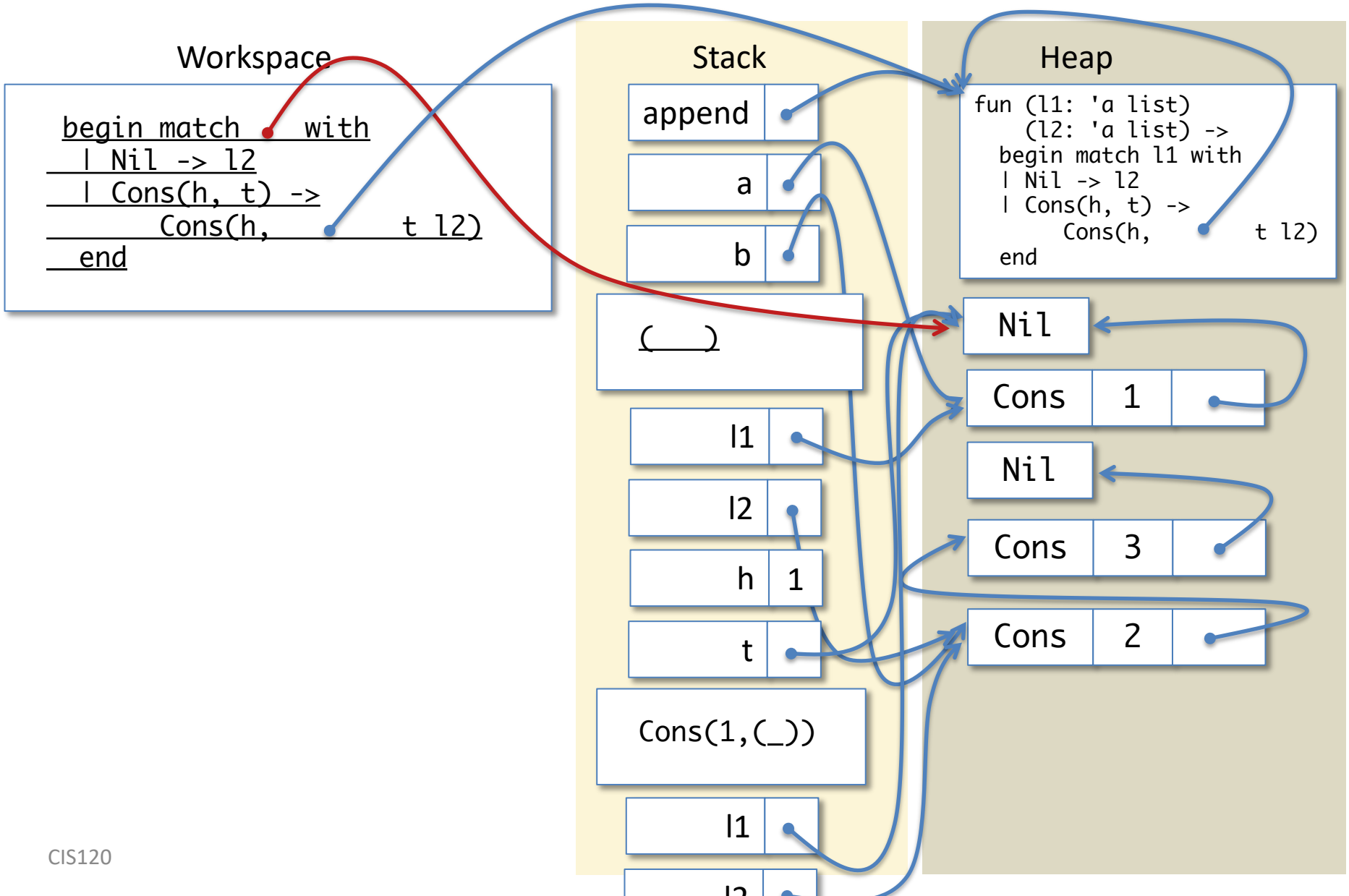
Lookup ‘l1’


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

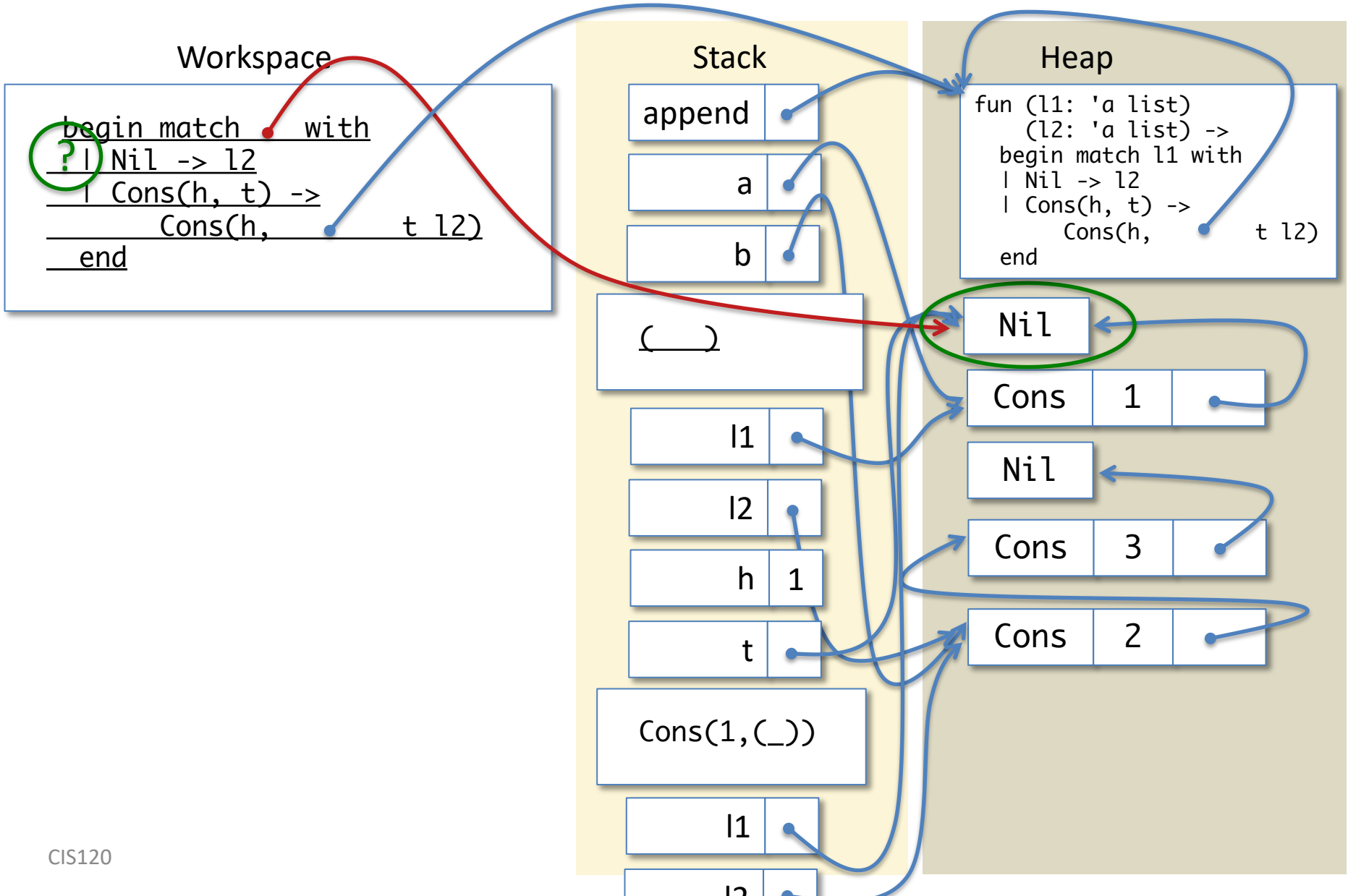
Lookup ‘l1’


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

Match Expression



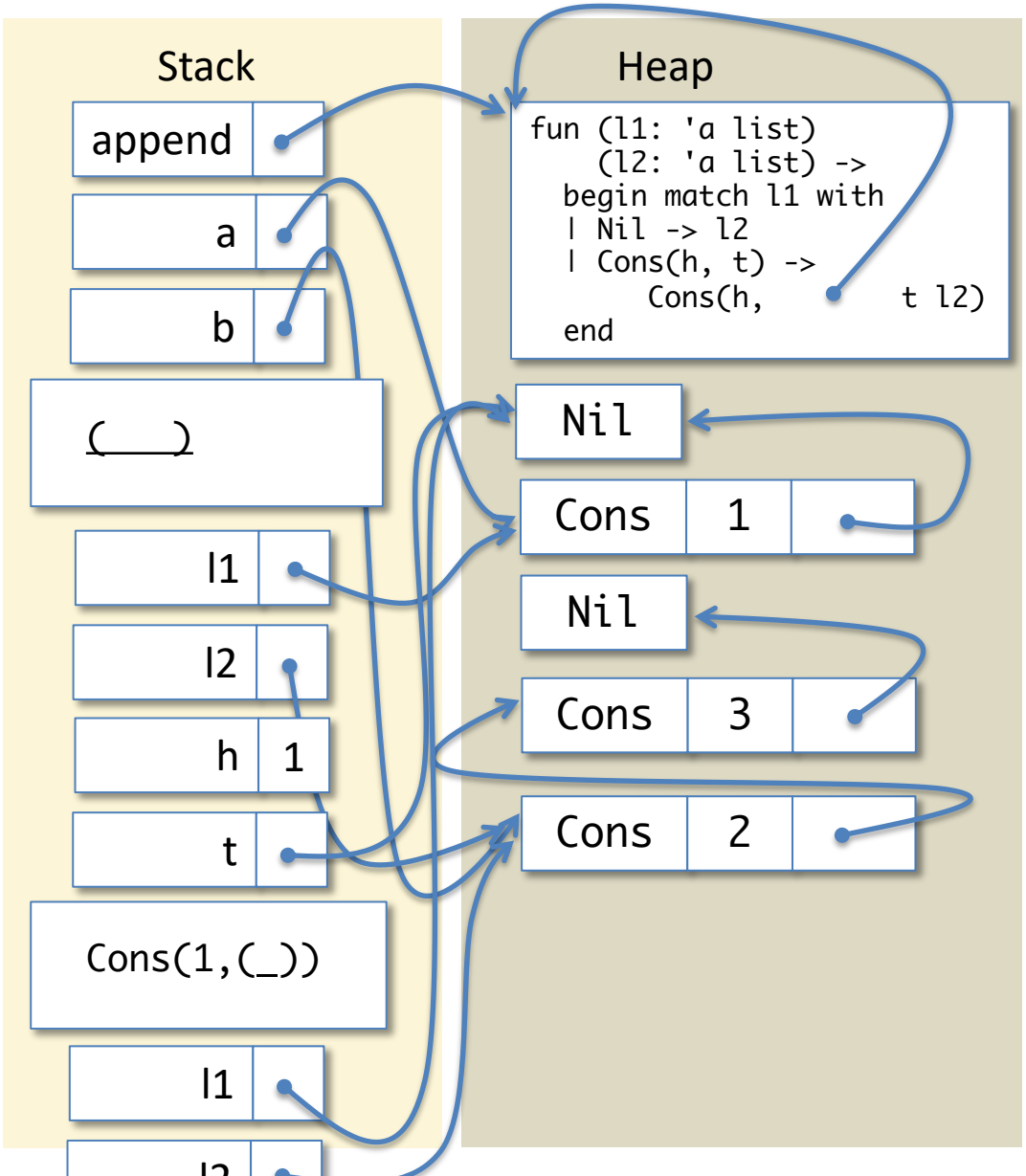
The Nil case Matches



Simplify the Branch (nothing to push)

Workspace

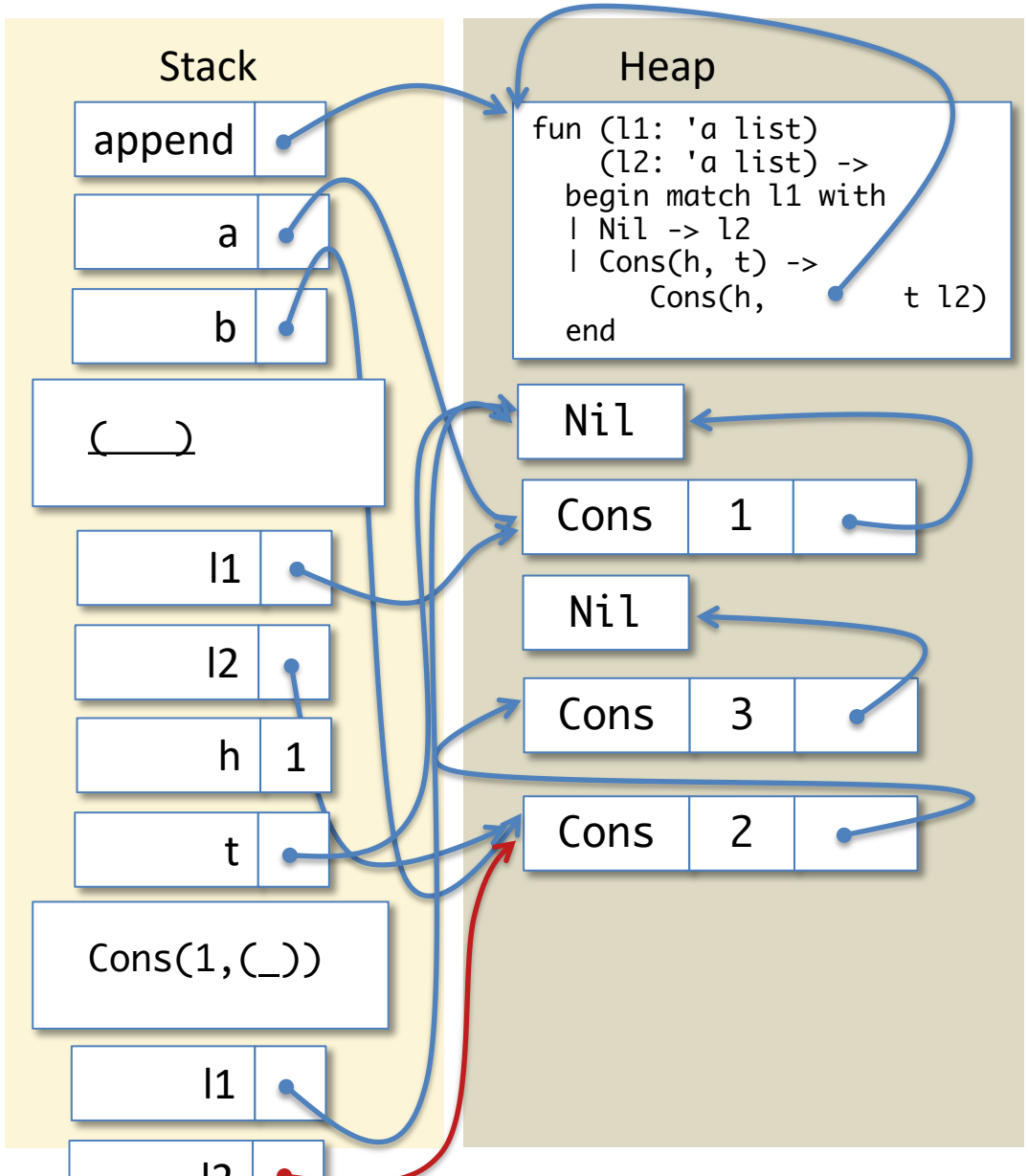
```
l2
```



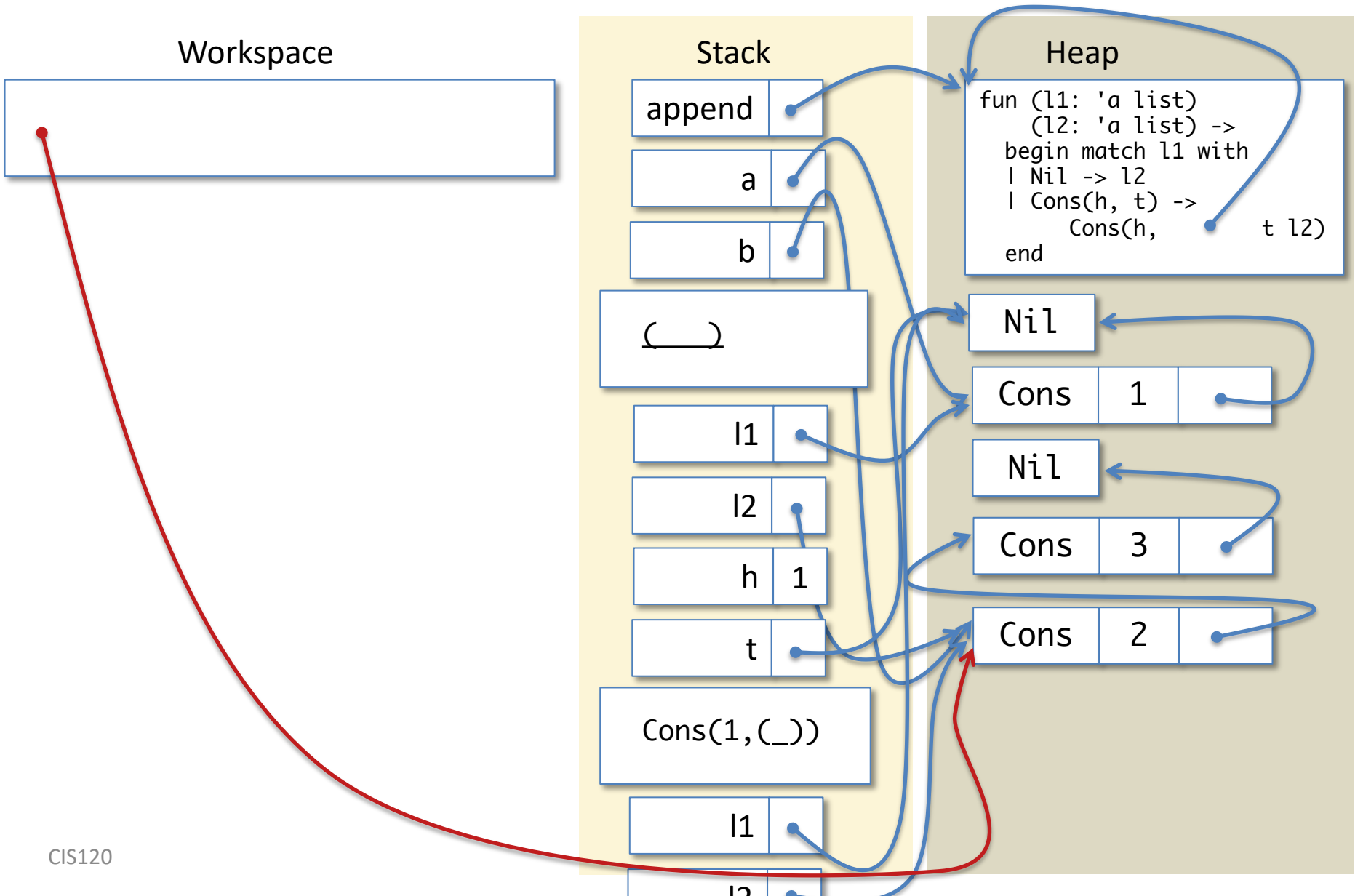
Lookup ‘l2’

Workspace

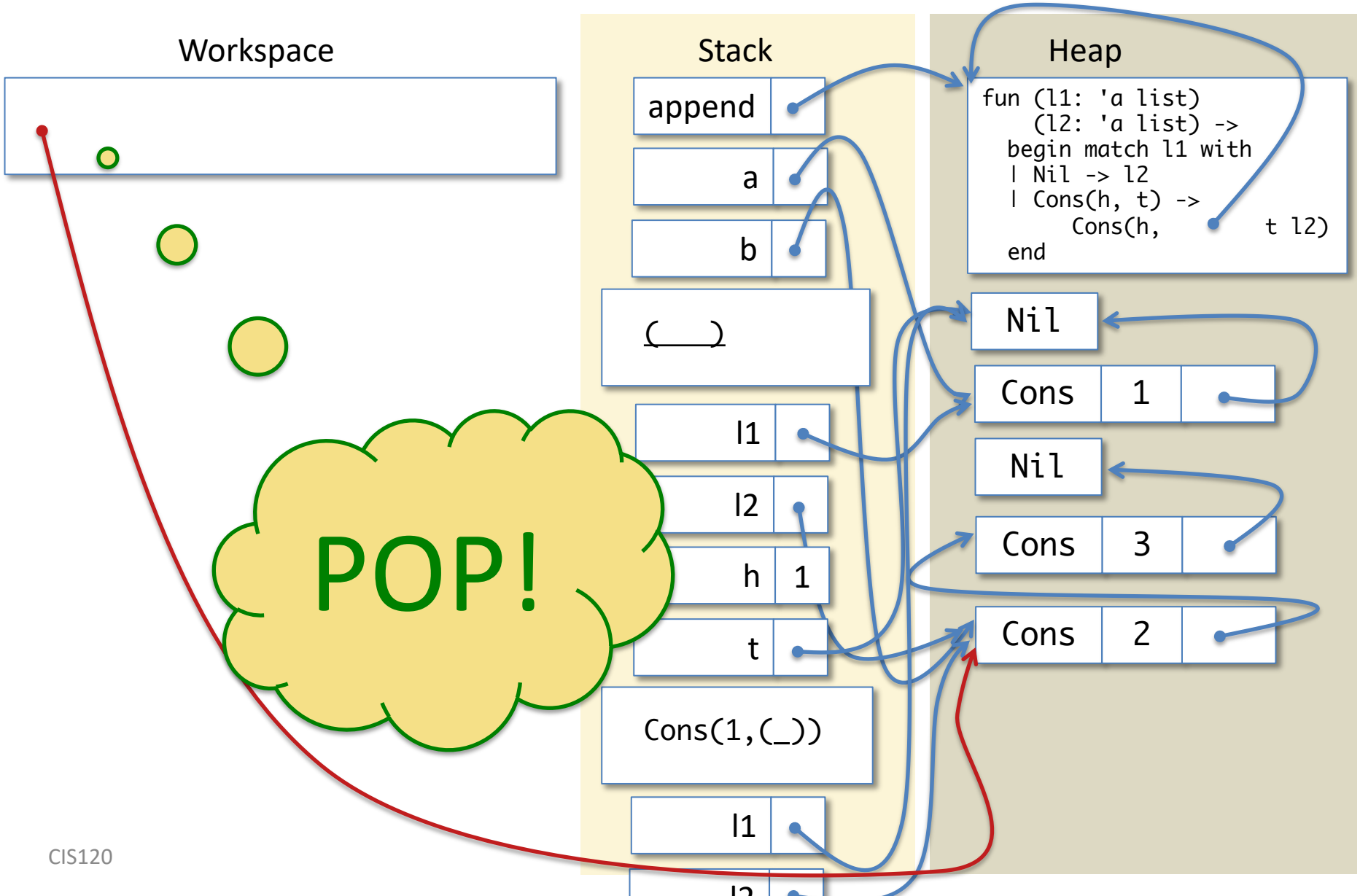
l2



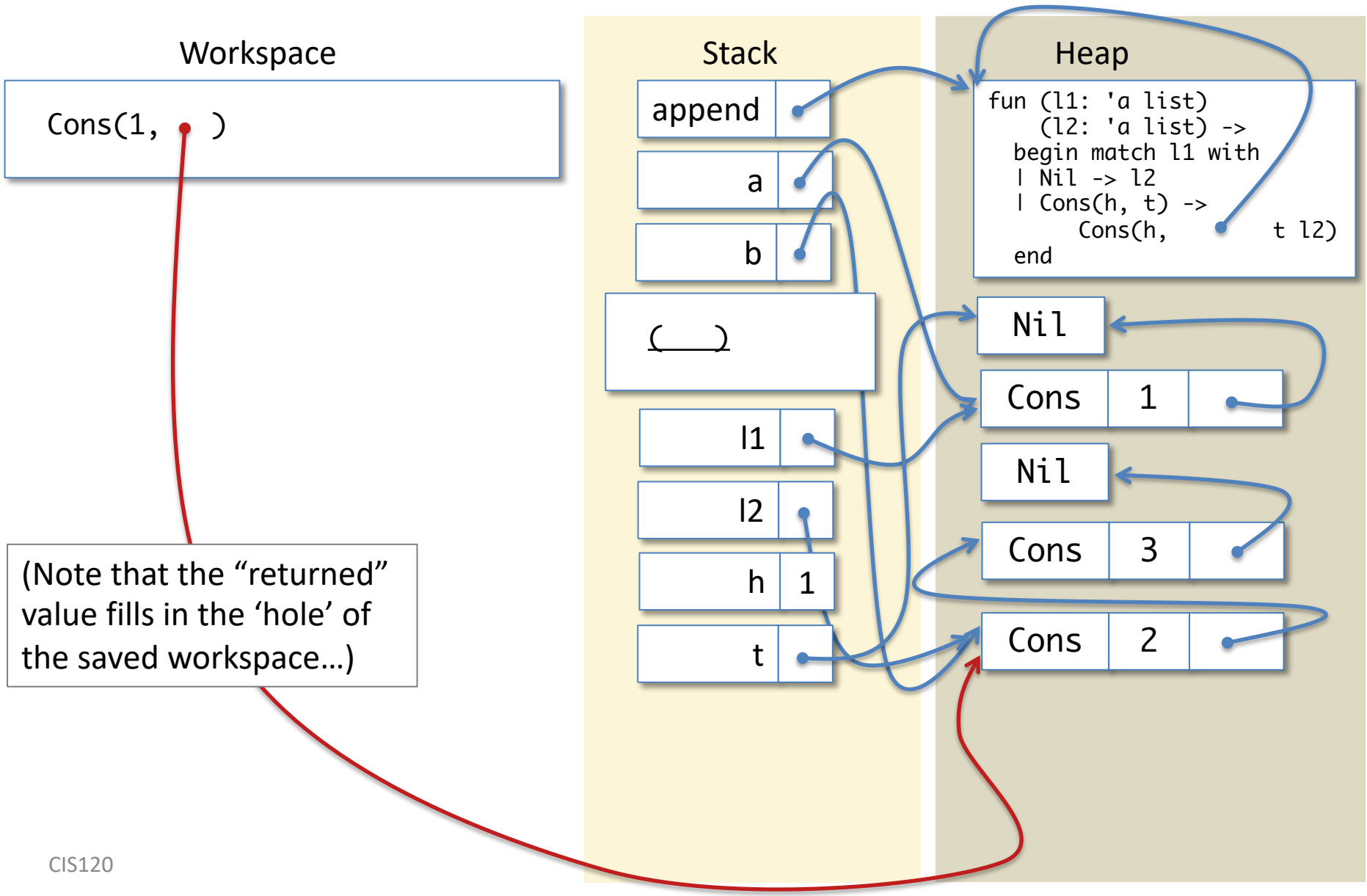
Lookup ‘l2’


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

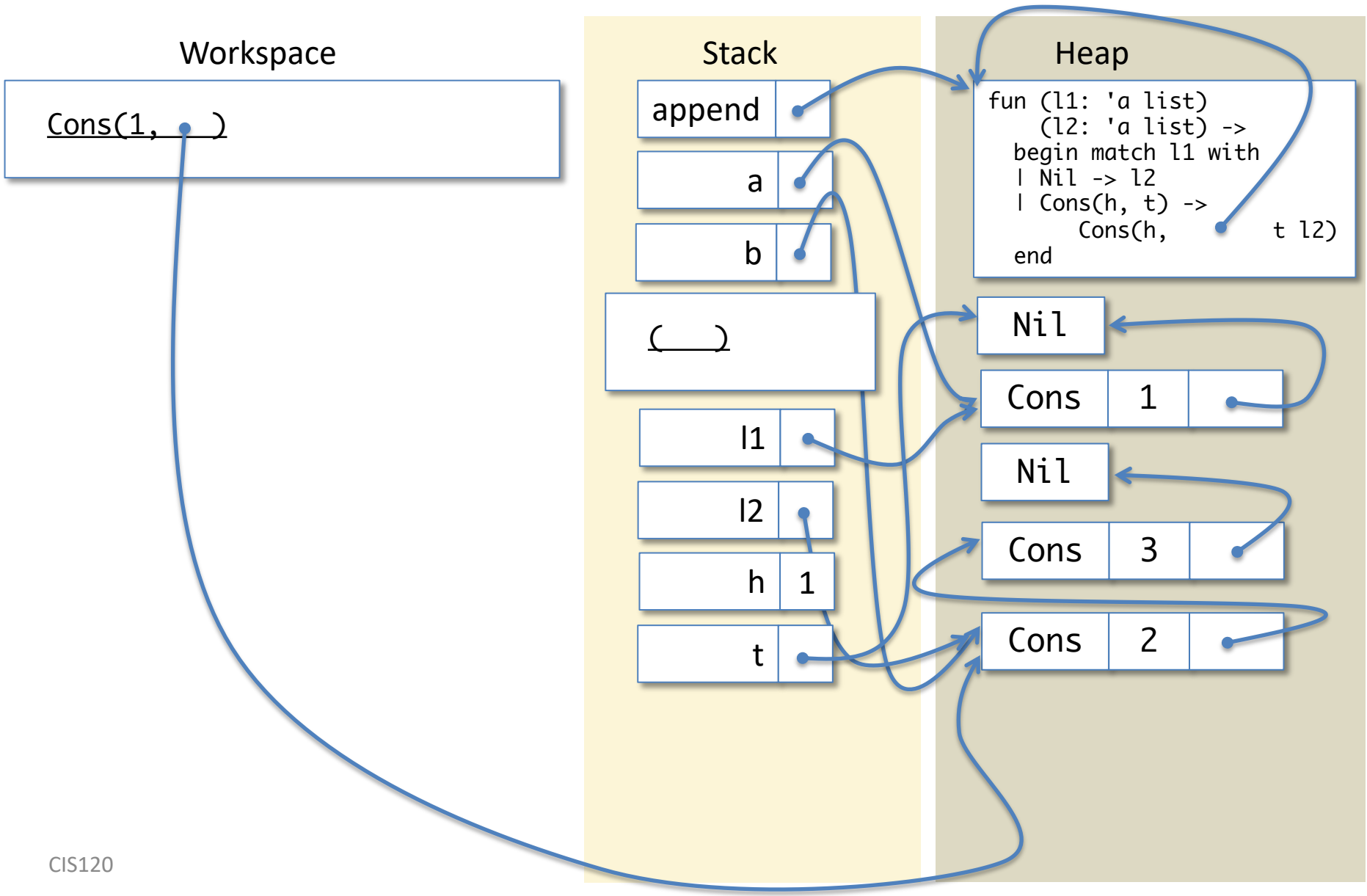
Done! Pop stack to last Workspace



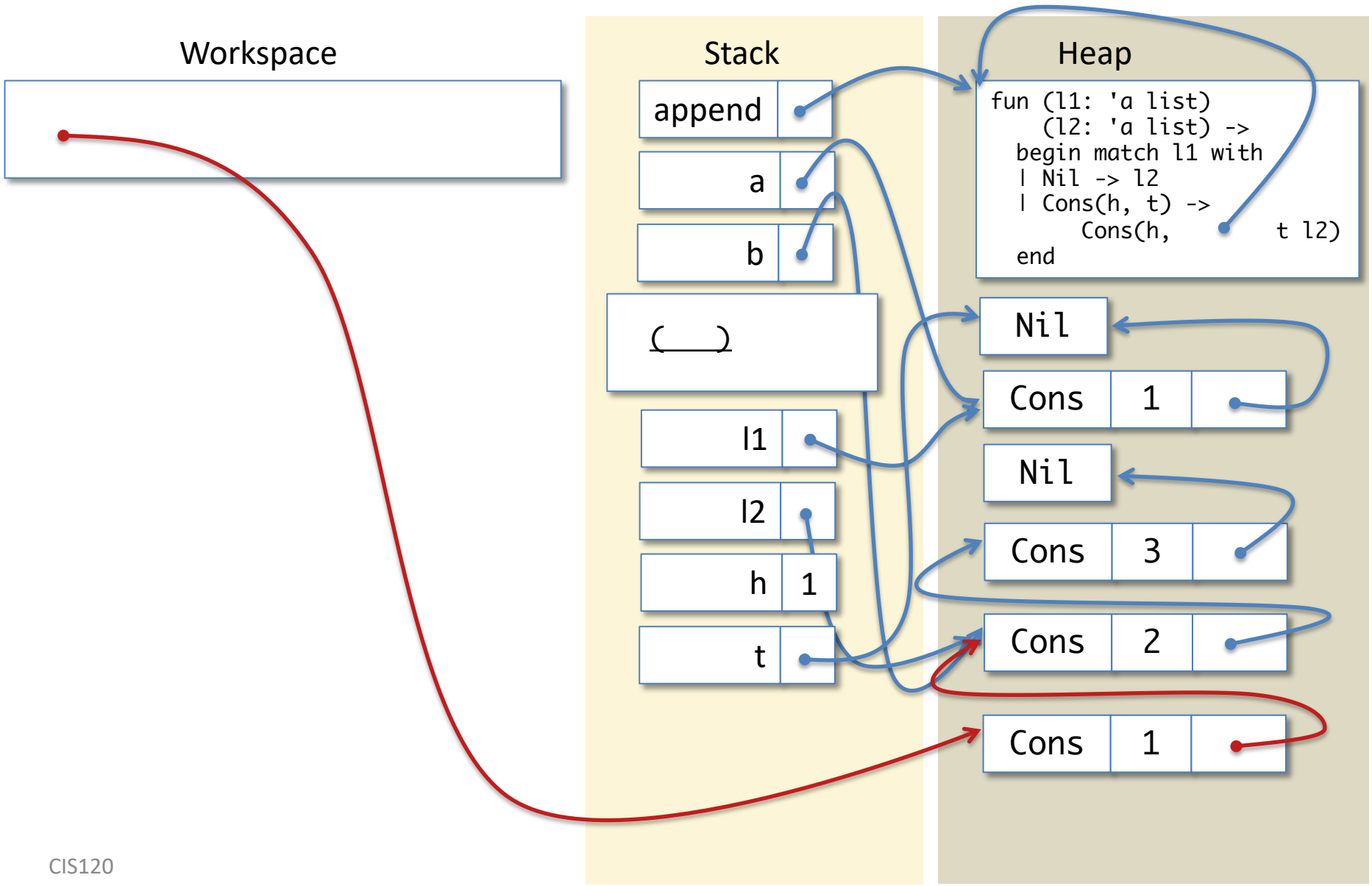
Done! Pop stack to last Workspace



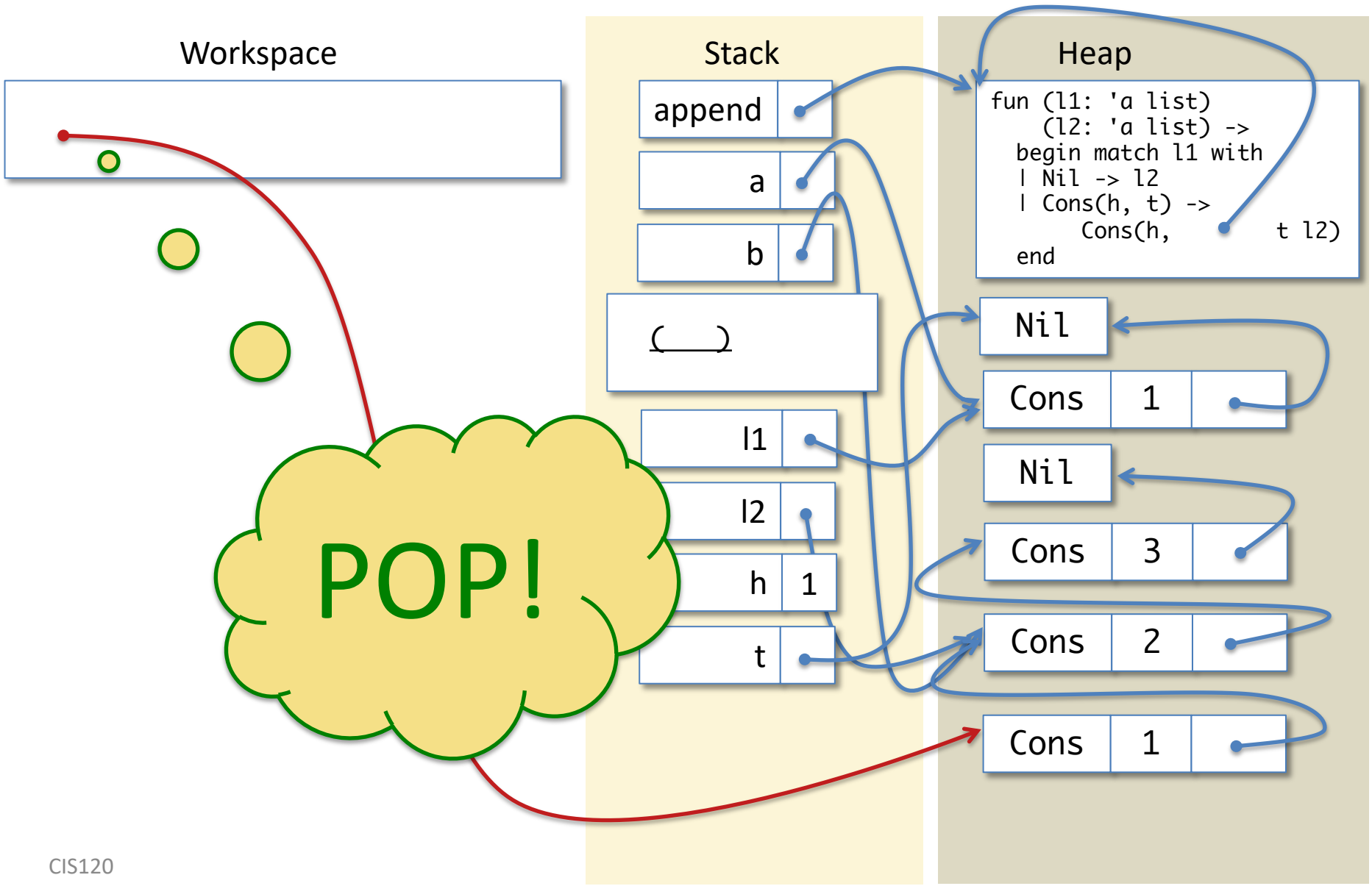
Allocate a Cons cell



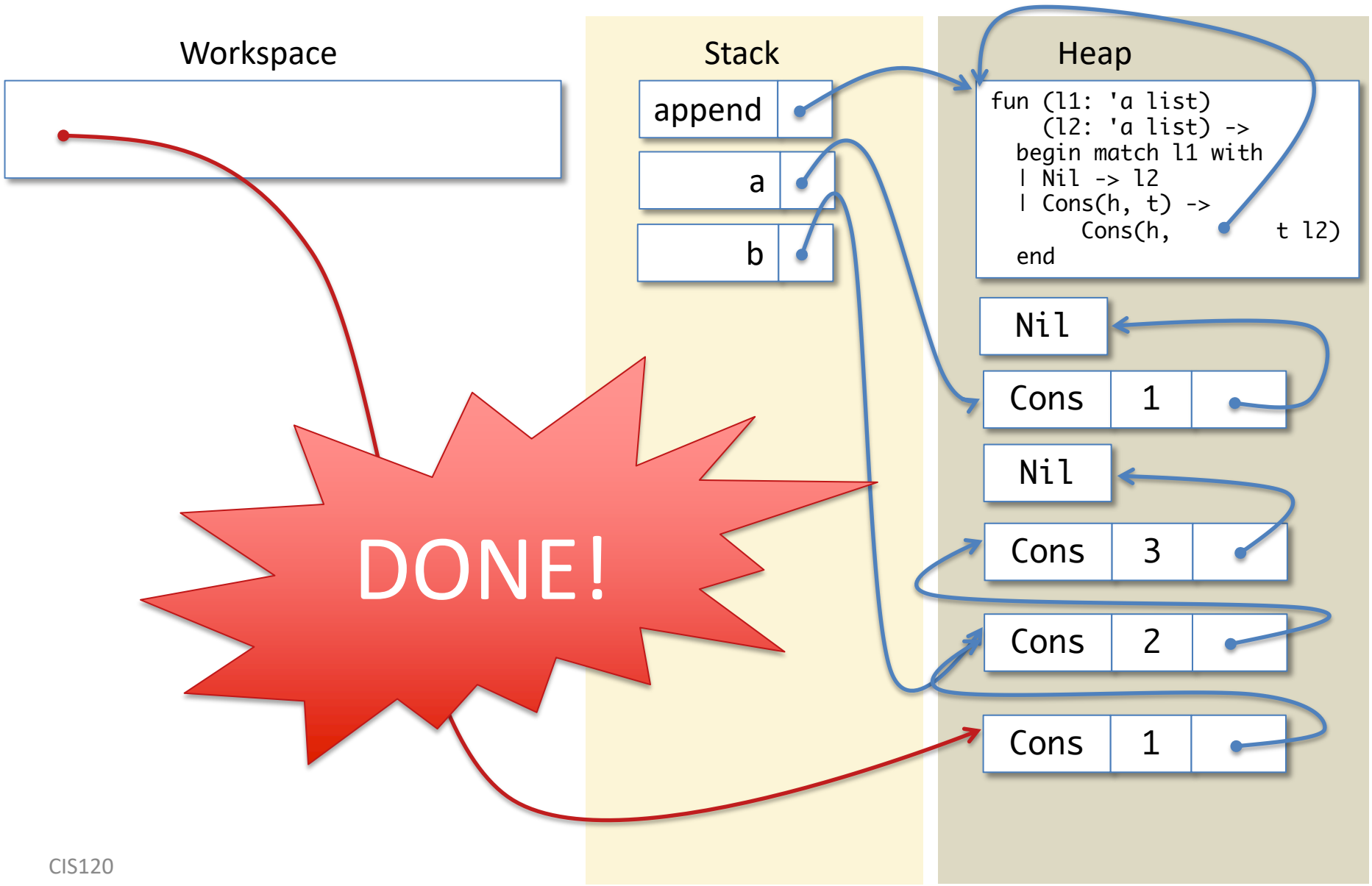
Allocate a Cons cell



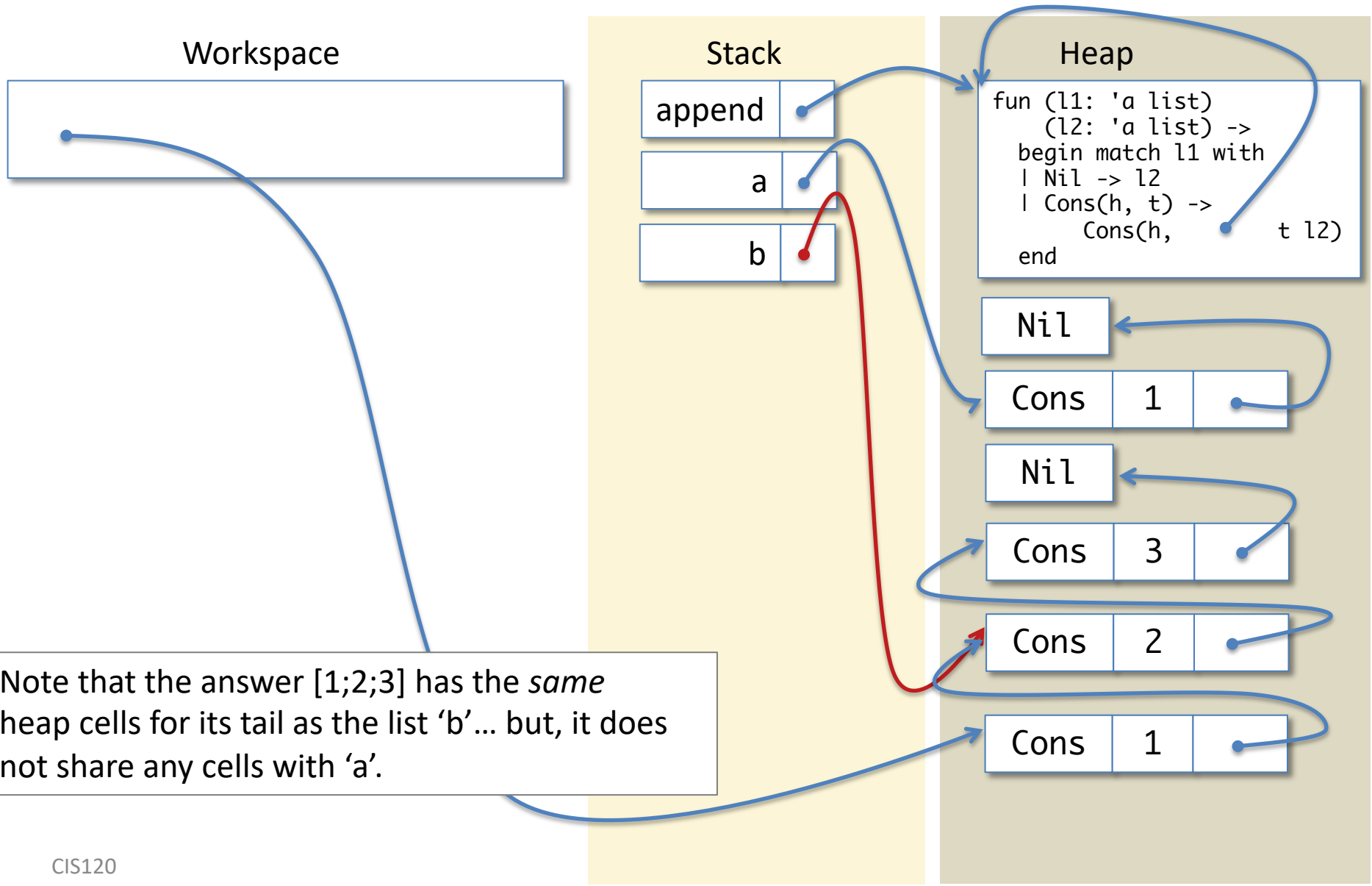
Done! Pop stack to last Workspace



Done! (PHEW!)


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

Done! (PHEW!)


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