

# Programming Languages and Techniques (CIS120)

## Lecture 12

Feb 14, 2014

Mutable State  
Abstract Stack Machine

# Announcements

- “Make up” office hours for Thursday recitation
  - Probably Sunday afternoon, watch Piazza for details
- Homework 4 due Tuesday
- Read Chapters 14 & 15 in the lecture notes
- Midterm 1
  - Scheduled *in class* on *Friday, Feb 21st*
  - Review material posted on course website
  - Review session Wednesday, Feb 19<sup>th</sup>, 7-9PM in Levine 101

# HW 03 feedback

- Comments
  - FUN but challenging
  - loved it
  - This was quite interesting and after working with lists and trees, felt easier than the last assignment.
  - Interesting but time-consuming.
  - This homework took much longer than the previous ones.
  - I need to stop underestimating how long these assignments will take.
  - This homework was very difficult.
  - I choked on a banana in the middle of the assignment. College life is rough.
  - UUUUUUUUUUGGGGGGHHHHHHHHGGGGHHGHGHHGGGGHHHH
  - (Also, difficulties with project configuration, compiler error messages, long lines at OH)
- Timespent
  - min: 1, avg:12, max:100, n=117

# Mutable state

# Why Use Declarative Programming?

- Simple
  - small language: arithmetic, local variables, recursive functions, datatypes, pattern matching, polymorphism and modules
  - simple substitution model of computation
- Persistent data structures
  - Nothing changes, so can remember all intermediate results
  - Good for version control, fault tolerance, etc.
- Typecheckers give more helpful errors
  - Once your program compiles, it needs less testing
  - failwith vs. NullPointerException
- Easier to parallelize and distribute
  - No implicit interactions between parts of the program. All of the behavior of a function is specified by its arguments

# Why Use Mutable State?

- Action at a distance
  - allow remote parts of a program to communicate / share information without threading the information through all the points in between
- Direct manipulation of hardware (device drivers, etc.)
- Data structures with explicit sharing
  - e.g. graphs
  - without mutation, it is only possible to build trees – no cycles
- Efficiency/Performance
  - a few data structures have imperative versions with better asymptotic efficiency than the best declarative version
- Re-using space (in-place update)
- Random-access data (arrays)

# A new view of imperative programming

## Java (and C, C++, C#)

- Null is contained in (almost) every type. Partial functions can return **null**.
- Code is a sequence of **statements** that do something, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

## OCaml (and Haskell, etc.)

- No null. Partiality must be made explicit with **options**.
- Code is an **expression** that has a value. Sometimes computing that value has other effects.
- References are **immutable** by default, must be explicitly declared to be mutable

Records



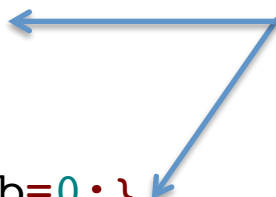
# Immutable Records

- Records are like tuples with named fields:

```
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int;}

(* some example rgb values *)
let red   : rgb = {r=255; g=0;   b=0;}
let blue  : rgb = {r=0;   g=0;   b=255;}
let green : rgb = {r=0;   g=255; b=0;}
let black : rgb = {r=0;   g=0;   b=0;}
let white : rgb = {r=255; g=255; b=255;}
```

Curly braces  
around record.  
Semicolons after  
record components.



- The type `rgb` is a record with three fields: `r`, `g`, and `b`
  - fields can have any types; they don't all have to be the same
- Record values are created using this notation:  
`{field1=val1; field2=val2;...}`

# Field Projection

- The value in a record field can be obtained by using “dot” notation: `record.field`

```
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int;}

(* using 'dot' notation to project out components *)
(* calculate the average of two colors *)
let average_rgb (c1:rgb) (c2:rgb) : rgb =
  {r = (c1.r + c2.r) / 2;
   g = (c1.g + c2.g) / 2;
   b = (c1.b + c2.b) / 2;}
```

# Mutable Record Fields

- By default, all record fields are *immutable*—once initialized, they can never be modified.
- OCaml supports *mutable* fields that can be imperatively updated by the “set” command: `record.field <- val`

note the ‘mutable’ keyword

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

let p0 = {x=0; y=0}
(* set the x coord of p0 to 17 *)
;; p0.x <- 17
;; print_endline ("p0.x = " ^ (string_of_int p0.x))
```

“in-place” update of p0.x

# Defining new Commands

- Functions can assign to mutable record fields
- Note that the return type of '`<-`' is `unit`

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

(* a command to shift a point by dx,dy *)
let shift (p:point) (dx:int) (dy:int) : unit =
  p.x <- p.x + dx;
  p.y <- p.y + dy
```

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

(* a command to shift a point by dx,dy *)
let shift (p:point) (dx:int) (dy:int) : unit =
  p.x <- p.x + dx;
  p.y <- p.y + dy
```

What answer does the following function produce when called?

```
let f (p1:point) : int =
  p1.x <- 17;
  p1.x
```

1. 17
2. 34
3. sometimes 17 and sometimes 34
4. f is ill typed

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

(* a command to shift a point by dx,dy *)
let shift (p:point) (dx:int) (dy:int) : unit =
  p.x <- p.x + dx;
  p.y <- p.y + dy
```

What answer does the following function produce when called?

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  p2.x <- 34;
  p1.x
```

1. 17
2. 34
3. sometimes 17 and sometimes 34
4. f is ill typed

# Issue with Mutable State: Aliasing

- What does this function return?

```
let f (p1:point) (p2:point) : int =  
  p1.x <- 17;  
  p2.x <- 42;  
  p1.x
```

```
(* Consider this call to f *)  
let ans = f p0 p0
```

Two identifiers are said to be *aliases* if they both name the *same* mutable record. Inside `f`, `p1`, and `p2` might be aliased, depending on which arguments are passed to `f`.

# Modeling Computation with Mutable State



Have you used the substitution model to reason about how functions evaluate?

```
total_secs (2 + 3) 12 17
  ↳ total_secs 5 12 17
  ↳ (5 * 60 + 12) * 60 + 17 subst. the args
  ↳ (300 + 12) * 60 + 17
  ↳ 312 * 60 + 17
  ↳ 18720 + 17
  ↳ 18737
```

```
let total_secs (hours:int)
               (minutes:int)
               (seconds:int)
               : int =
  (hours * 60 + minutes) * 60 + seconds
```

1. yes, every single step
2. yes, but skipping some steps
3. no, it seems useless to me
4. what is the substitution model?

# Mutable Records

- *Mutable* (updateable) state means that the *locations* of values becomes important.

```
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
```

- The simple substitution model of program evaluation breaks down – it doesn't account for locations and can't explain aliasing.
- We need to refine our model of how to understand programs.

# Abstract Stack Machine

A new model of computation

# Abstract Machines

- The job of a programming language is to provide some *abstraction* of the underlying hardware
- An *abstract machine* hides the details of the real machine
  - model doesn't depend on the hardware
  - easier to reason about the behavior of programs
- There are lots of ways of visualizing machine evaluation
  - e.g. the substitution model we've been using until now
- An Abstract *Stack* Machine
  - is a good way of understanding how recursive functions work
  - gives an accurate picture of how OCaml data structures are shared internally (which helps predict how fast programs will run), and
  - extends smoothly to include imperative features (assignment, pointer manipulation) and objects (for Java)

# Stack Machine

- Three “spaces”
  - workspace
    - the expression the computer is currently working with
  - 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
  - Stop when there are no more simplifications

# Abstract Stack Machine

The abstract stack machine operates by simplifying the expression in the workspace...

... but instead of substitution, it records the values of variables on the stack

... values themselves are divided into primitive values (also on the stack) and reference values (on the heap).

For immutable structures, this model is just a complicated way of doing substitution

... but we need the extra complexity to understand mutable state.

We'll start with examples first, and then define general rules

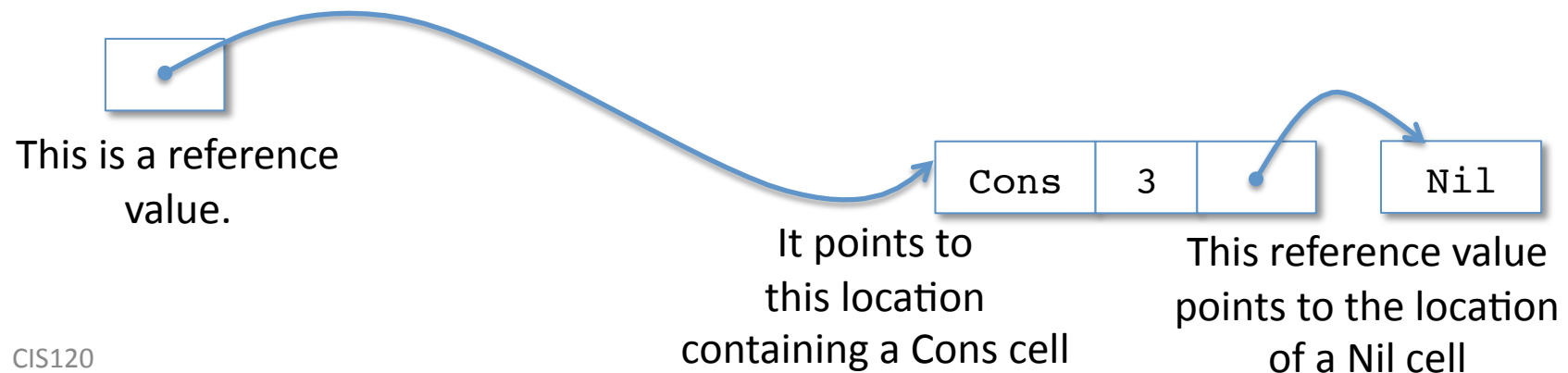
# Values and References

A *value* is either:

- a *primitive* value like an integer or,
- a *reference* (or *pointer*) into the heap

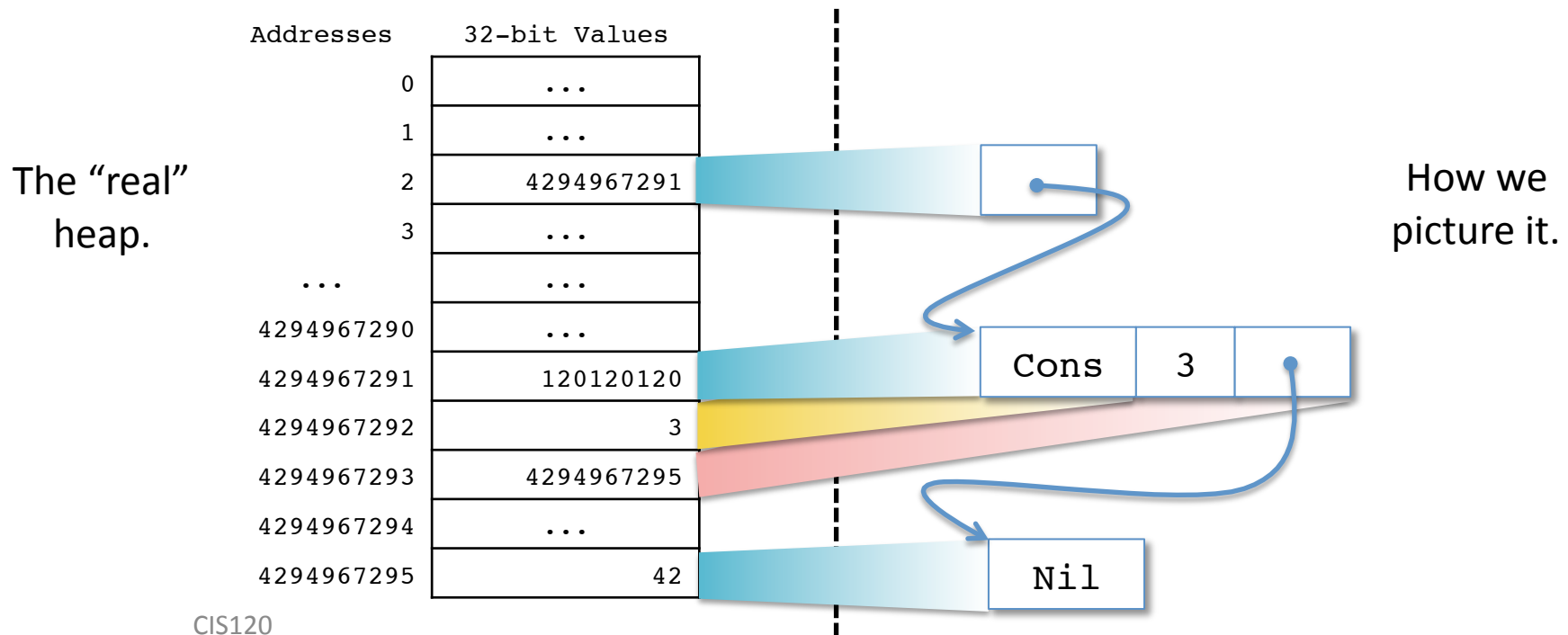
A reference is the *address* (or *location* of) a piece of data in the heap. We draw a reference as an “arrow”:

- The start of the arrow is the reference itself (i.e. the address).
- The arrow “points” to the value located at the reference’s address.



# References as an Abstraction

- In a real computer, the memory consists of an array of 32-bit words, numbered  $0 \dots 2^{32}-1$  (for a 32-bit machine)
  - A reference is just an address that tells you where to look up a value
  - Data structures are usually laid out in contiguous blocks of memory
  - Constructor tags are just numbers chosen by the compiler  
e.g. Nil = 42 and Cons = 120120120





# Simplifying let, variables, operators, and if expressions

# Simplification

Workspace

```
let x = 10 + 12 in  
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

Heap

# Simplification

Workspace

```
let x = 10 + 12 in  
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

Heap

# Simplification

Workspace

```
let x = 22 in  
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

Heap

# Simplification

Workspace

```
let x = 22 in  
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

Heap

# Simplification

Workspace

```
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

x	22
---	----

Heap

# Simplification

Workspace

```
let y = 2 + x in  
  if x > 23 then 3 else 4
```

Stack

x	22
---	----

Heap

x is not a value: so look it up in the stack

# Simplification

Workspace

```
let y = 2 + 22 in  
  if x > 23 then 3 else 4
```

Stack

x	22
---	----

Heap



# Simplification

Workspace

```
let y = 2 + 22 in  
  if x > 23 then 3 else 4
```

Stack

x	22
---	----

Heap

# Simplification

Workspace

```
let y = 24 in  
  if x > 23 then 3 else 4
```

Stack

x	22
---	----

Heap

# Simplification

Workspace

```
let y = 24 in  
if x > 23 then 3 else 4
```

Stack

x	22
---	----

Heap

# Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

x	22
---	----

y	24
---	----

Heap

# Simplification

Workspace

```
if x > 23 then 3 else 4
```

Stack

x	22
---	----

y	24
---	----

Heap

# Simplification

Workspace

```
if 22 > 23 then 3 else 4
```

Stack

x	22
---	----

y	24
---	----

Heap

# Simplification

Workspace

```
if 22 > 23 then 3 else 4
```

Stack

x	22
---	----

y	24
---	----

Heap

# Simplification

Workspace

```
if false then 3 else 4
```

Stack

x	22
---	----

y	24
---	----

Heap



# Simplification

Workspace

```
if false then 3 else 4
```

Stack

x	22
---	----

y	24
---	----

Heap

# Simplification

Workspace

4

Stack

x	22
---	----

y	24
---	----

Heap



# Simplification Rules

- A let-expression “let  $x = e$  in body” is ready if the expression  $e$  is a *value*
  - it is simplified by adding a binding of  $x$  to  $e$  at the end of the stack and leaving body in the workspace
- A variable is always ready
  - it is simplified by replacing it with its value from the stack, where binding lookup goes in order from most recent to least recent
- A primitive operator (like  $+$ ) is ready if both of its arguments are values
  - it is simplified by replacing it with the result of the operation
- An “if” expression is ready if the test is true or false
  - if it is true, it is simplified by replacing it with the then branch
  - if it is false, it is simplified by replacing it with the else branch

# Simplifying lists and datatypes using the heap

# 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

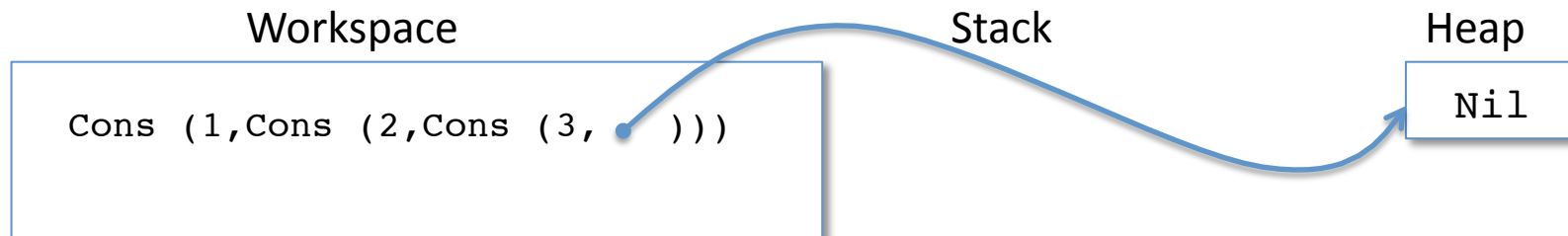
Workspace

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

Stack

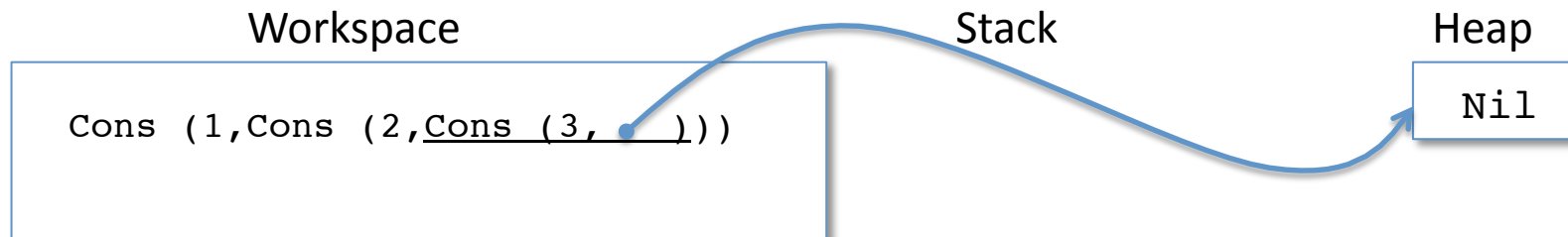
Heap

# Simplification

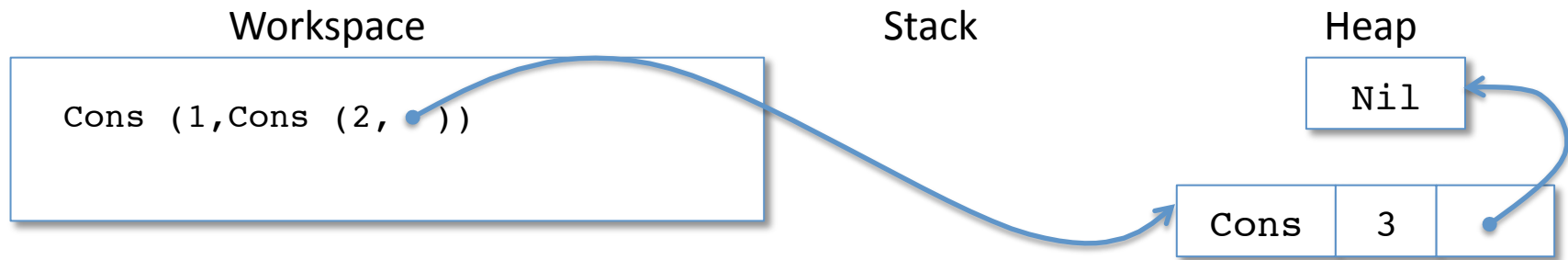




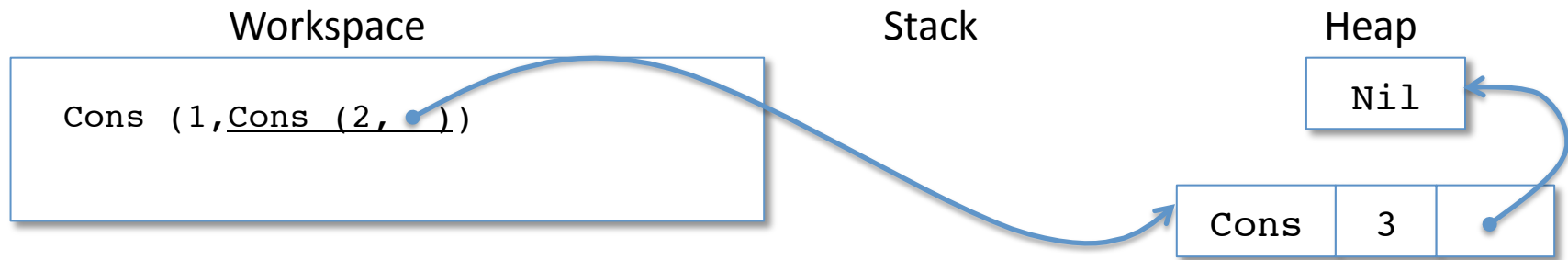
# Simplification



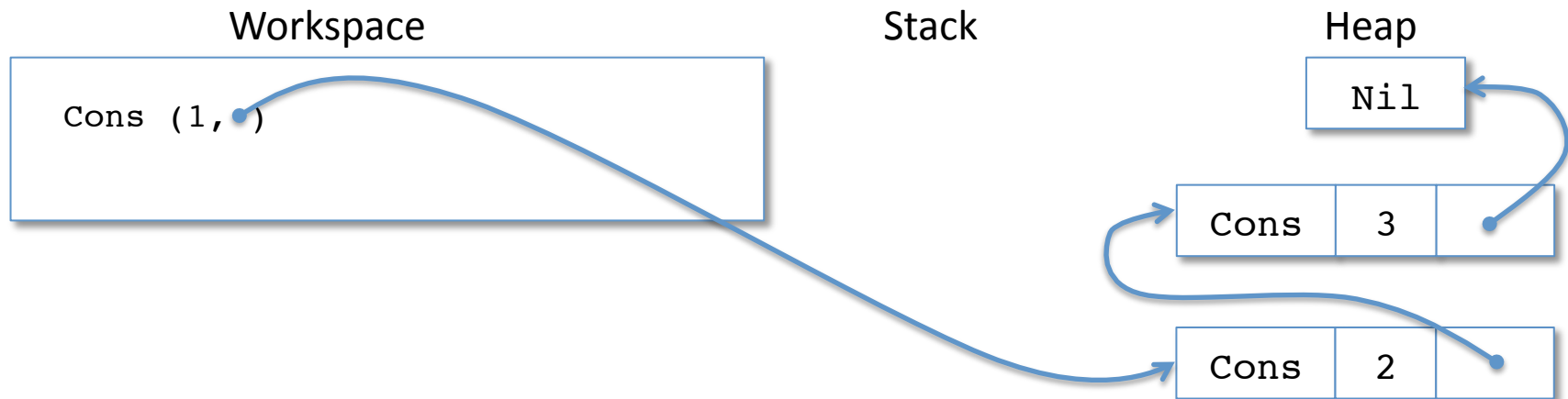
# Simplification



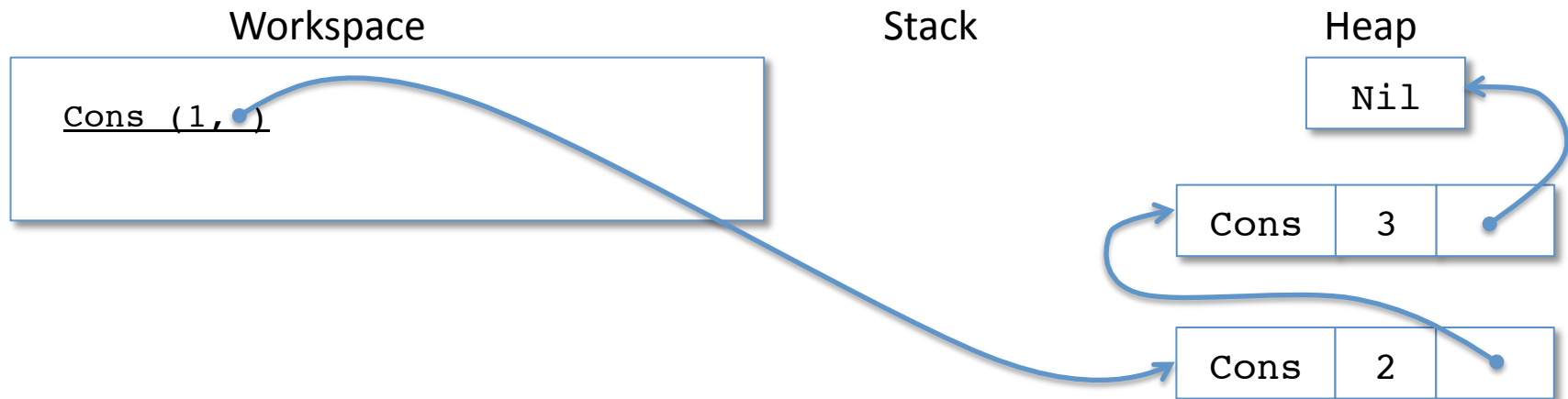
# Simplification



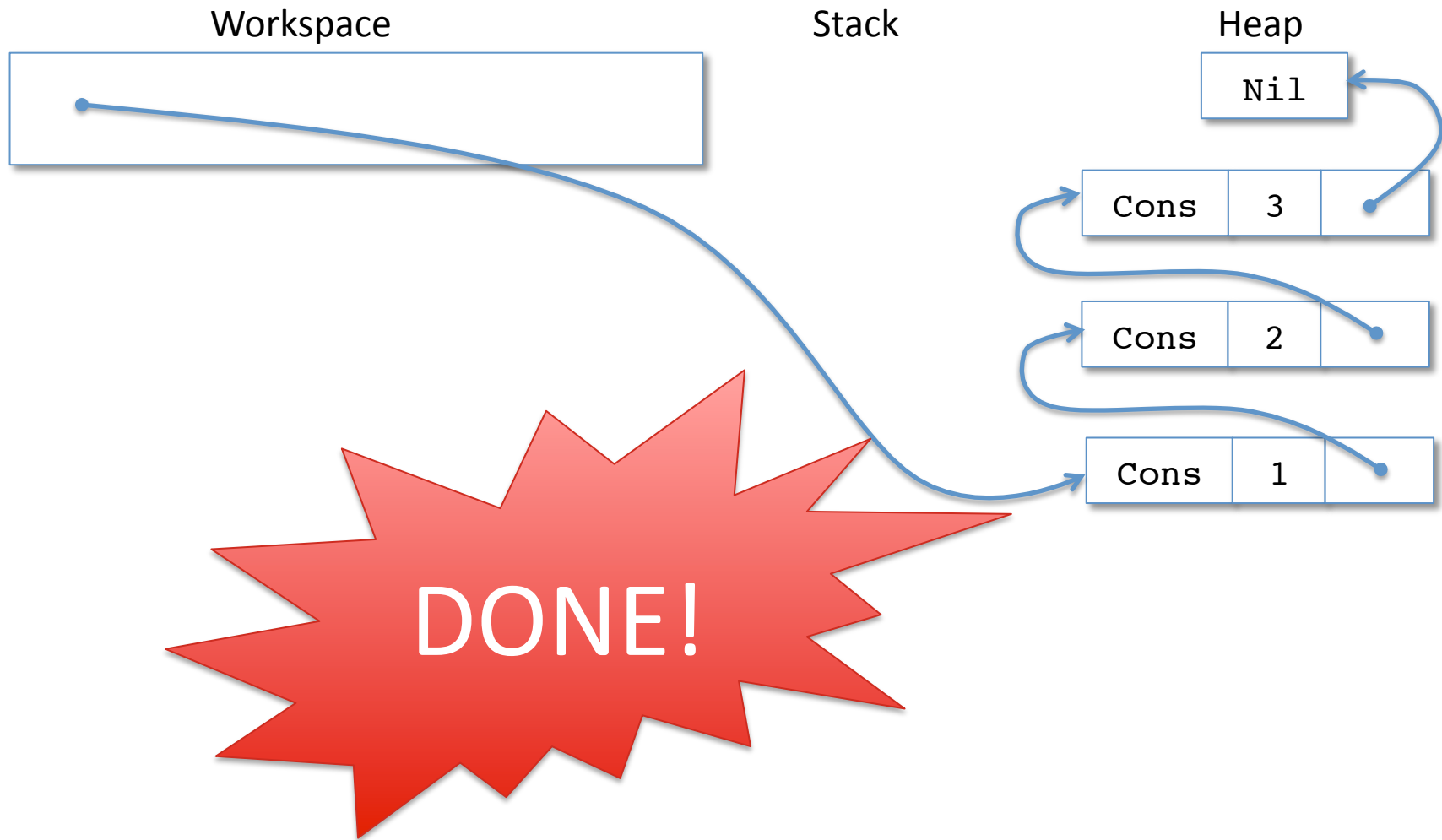
# Simplification



# Simplification



# Simplification



# Simplifying Datatypes

- A datatype constructor (like `Nil` or `Cons`) is ready if all its arguments are values
  - It is simplified by:
    - creating a new heap cell labeled with the constructor and containing the argument values\*
    - replacing the constructor expression in the workspace by a reference to this heap cell

\*Note: in OCaml, using a datatype constructor causes some space to be automatically allocated on the heap. Other languages have different mechanisms for accomplishing this: for example, the keyword 'new' in Java works similarly (as we'll see in a few weeks).

# Simplifying 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

# Function Simplification

Workspace

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

Stack

Heap

# Function Simplification

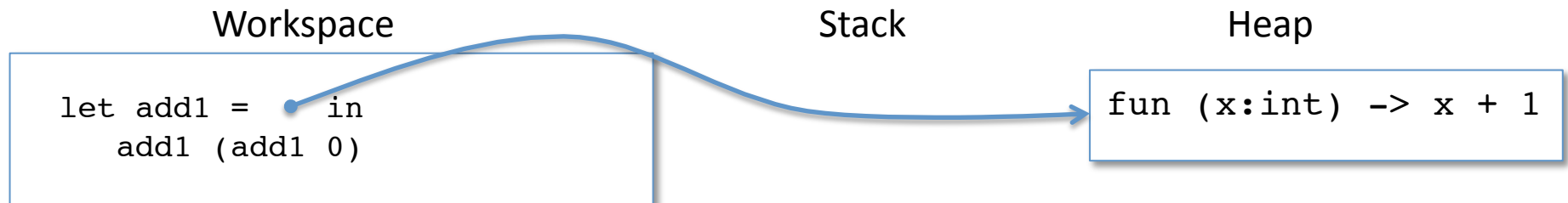
Workspace

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

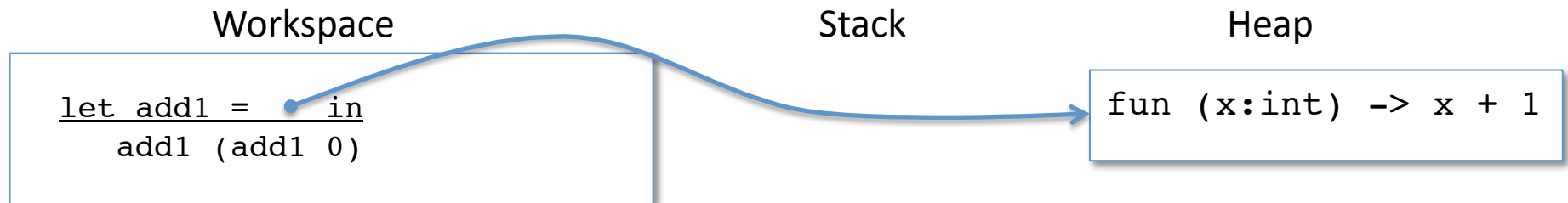
Stack

Heap

# Function Simplification



# Function Simplification



# Function Simplification

Workspace

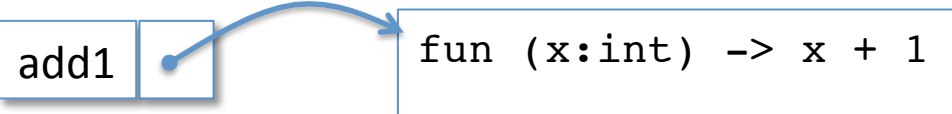
```
add1 (add1 0)
```

Stack

```
add1
```

Heap

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



# Function Simplification

Workspace

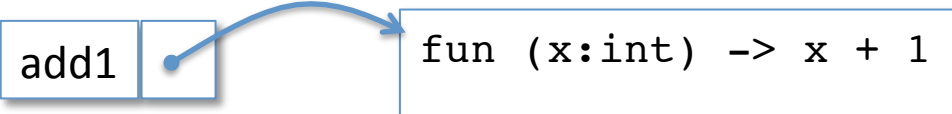
```
add1 (add1 0)
```

Stack

```
add1
```

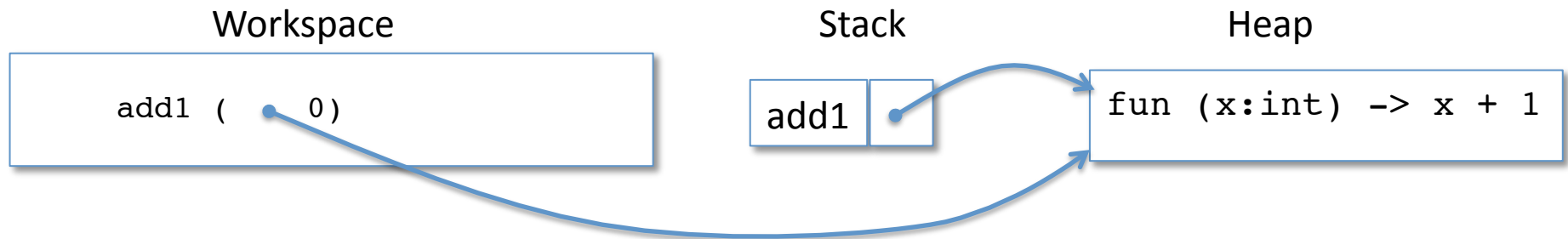
Heap

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

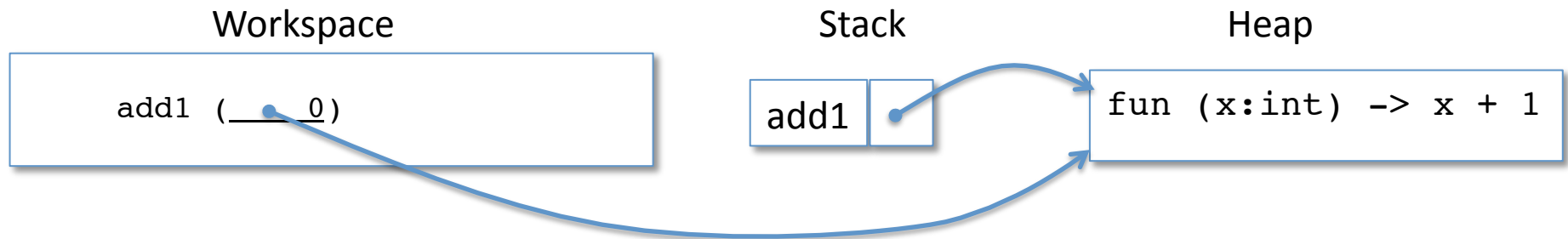




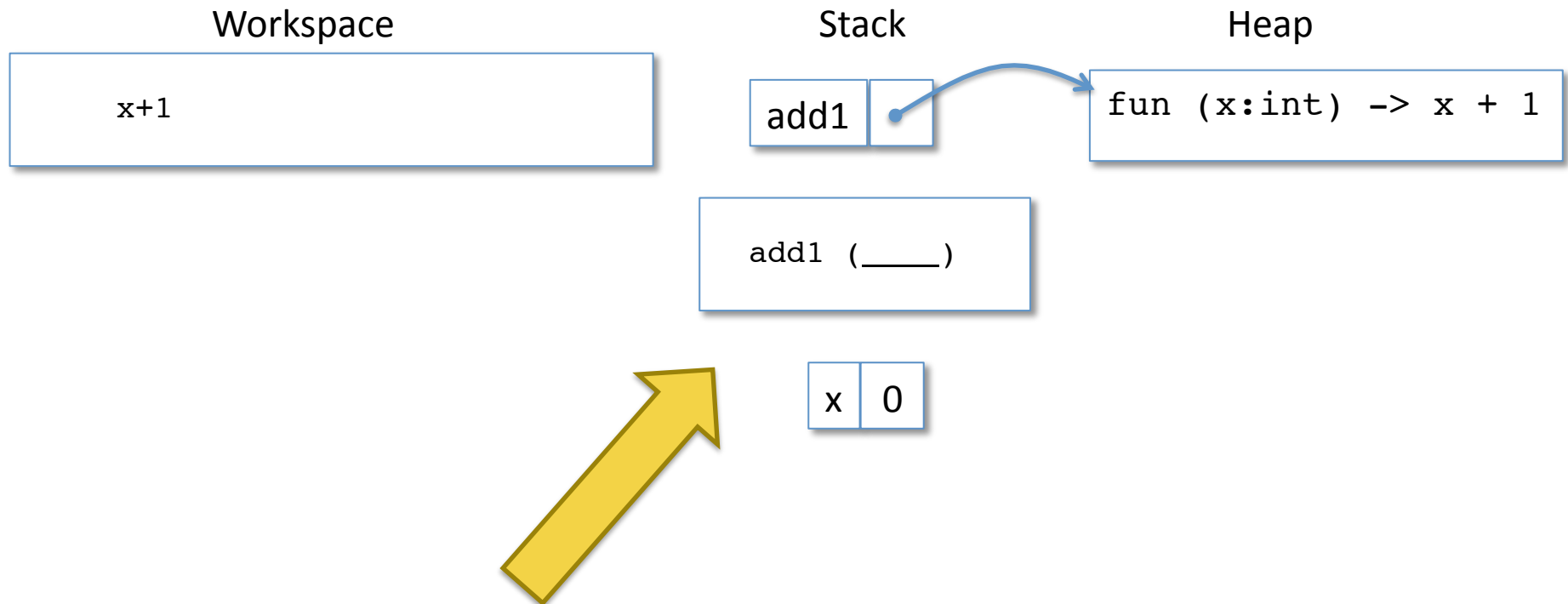
# Function Simplification



# Function Simplification



# Do the Call, Saving the Workspace

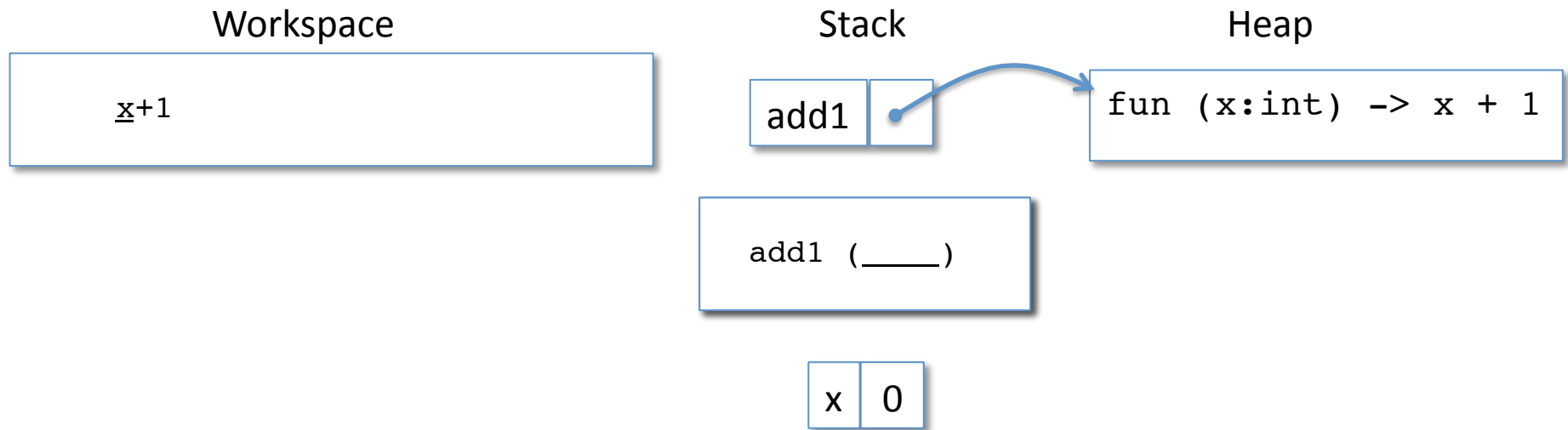


Note the saved workspace and pushed function argument.

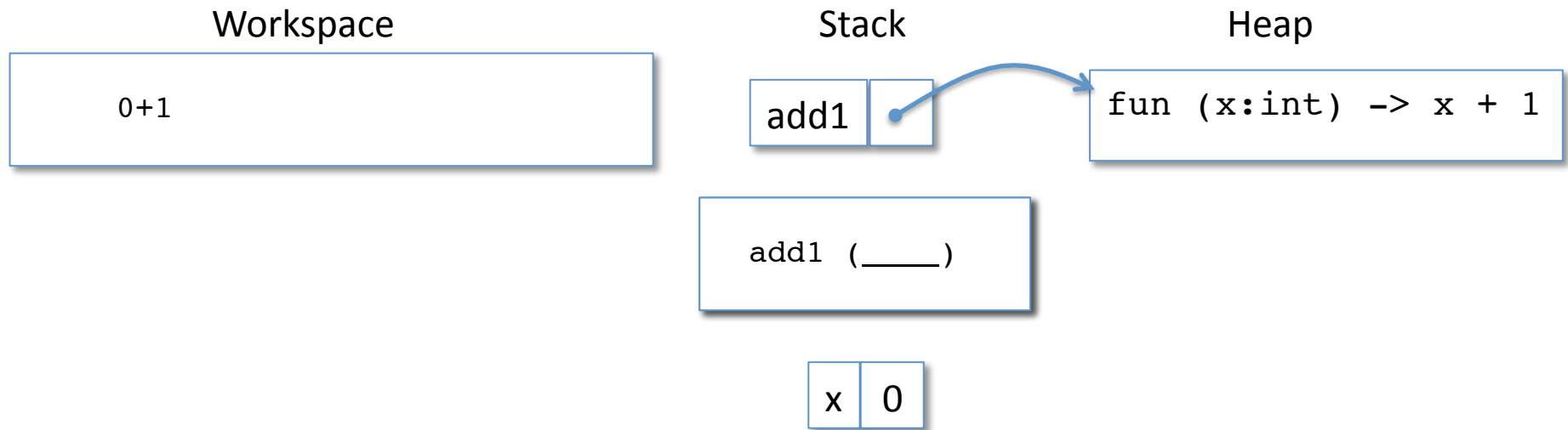
- compare with the workspace on the previous slide.
- the name 'x' comes from the name in the heap

The new workspace is the *body* of the function

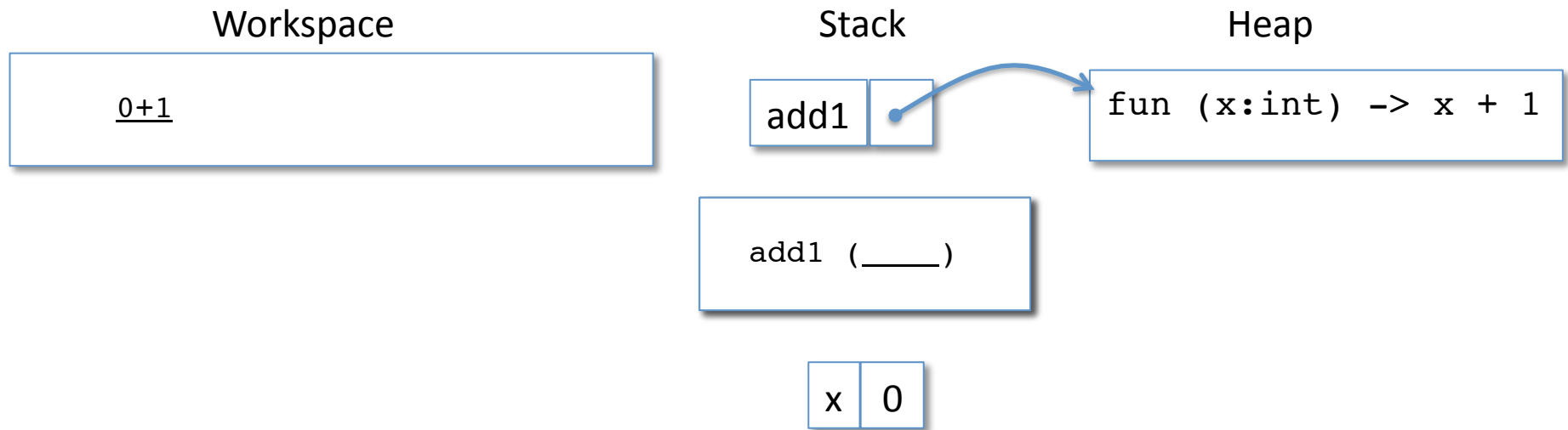
# Function Simplification



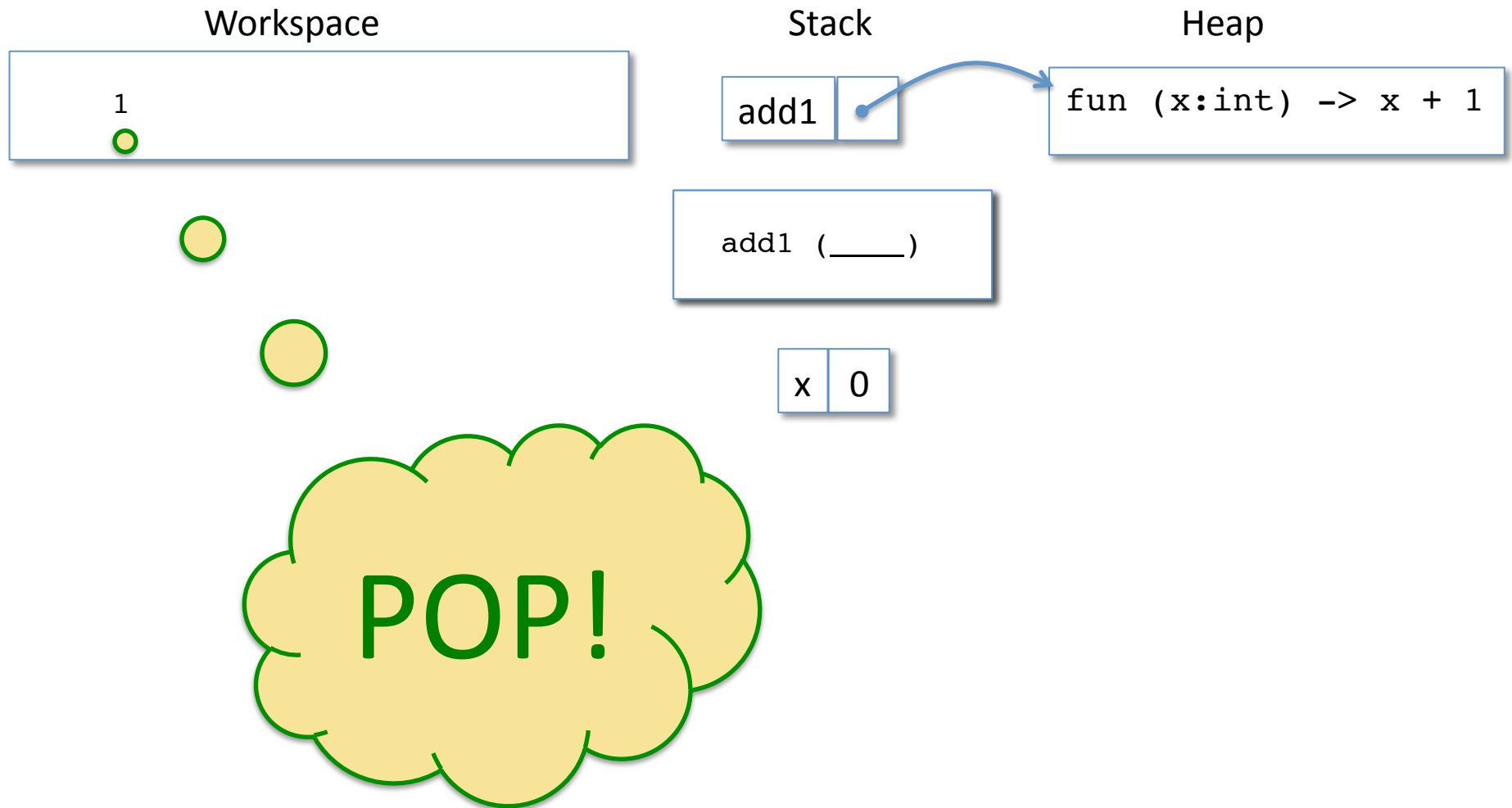
# Function Simplification



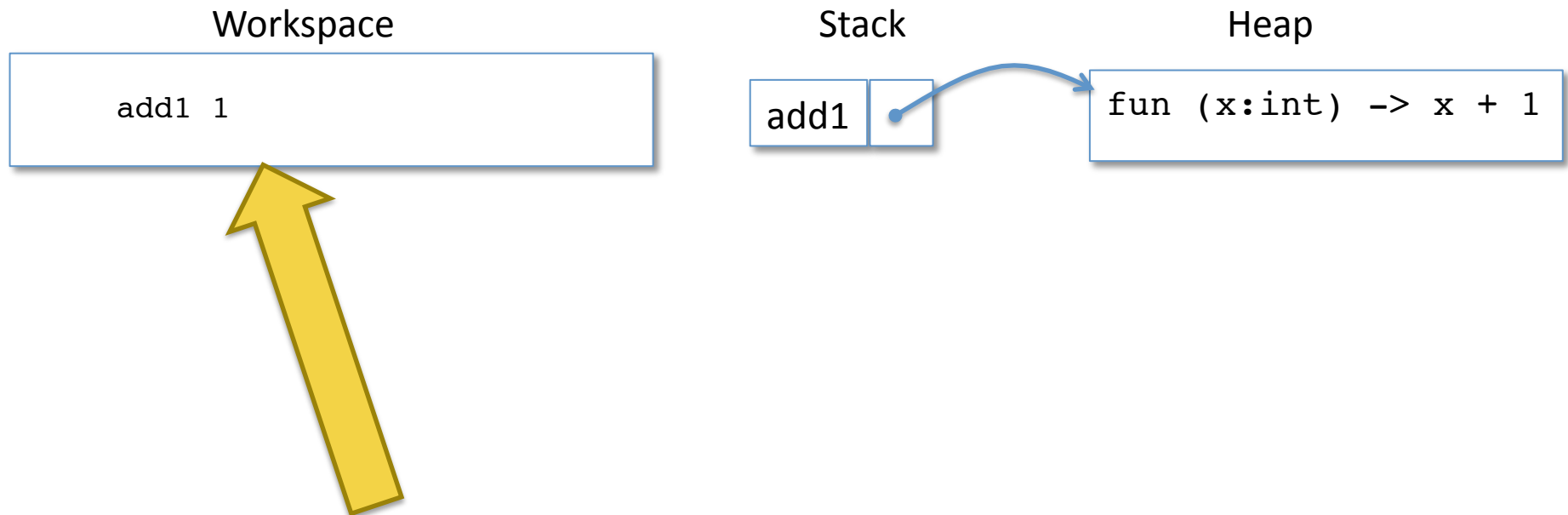
# Function Simplification



# Function Simplification



# Function Simplification



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

Workspace

```
add1 1
```

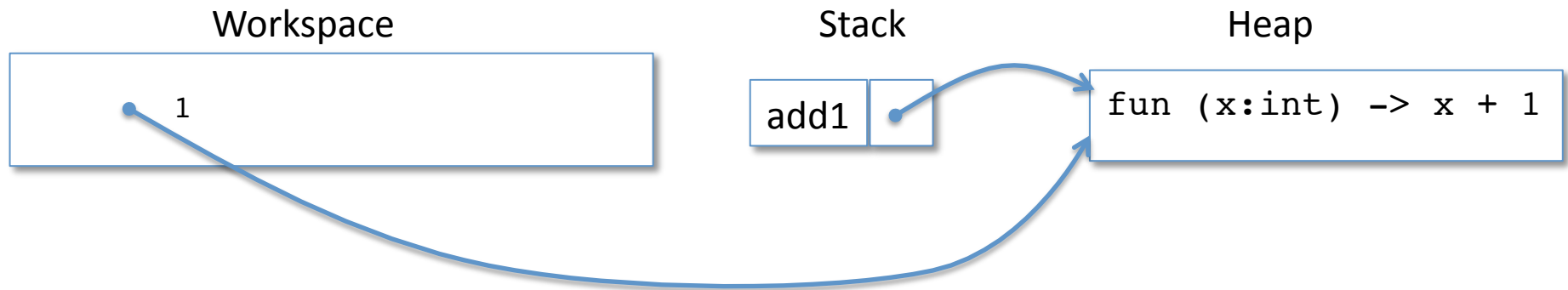
Stack

```
add1
```

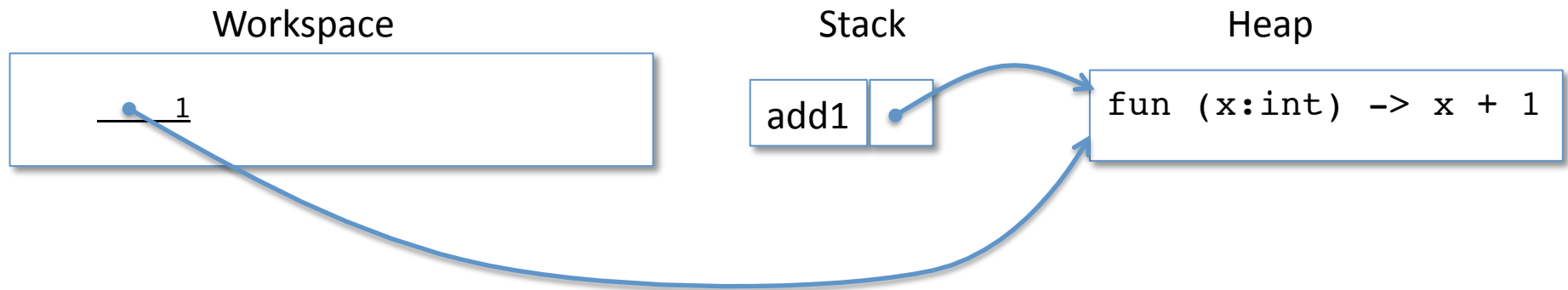
Heap

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

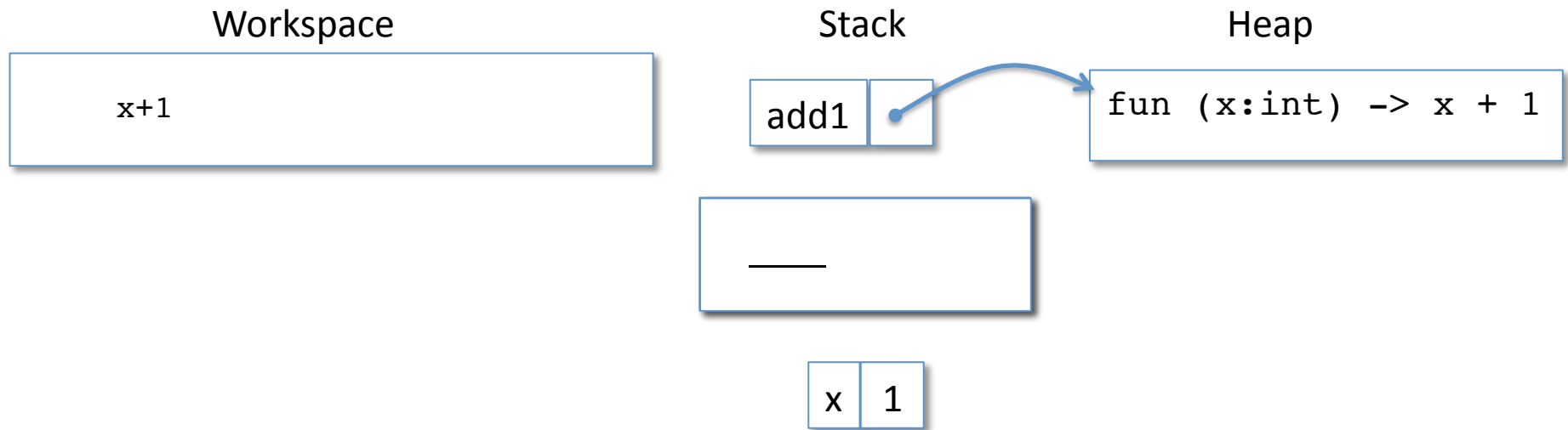
# Function Simplification



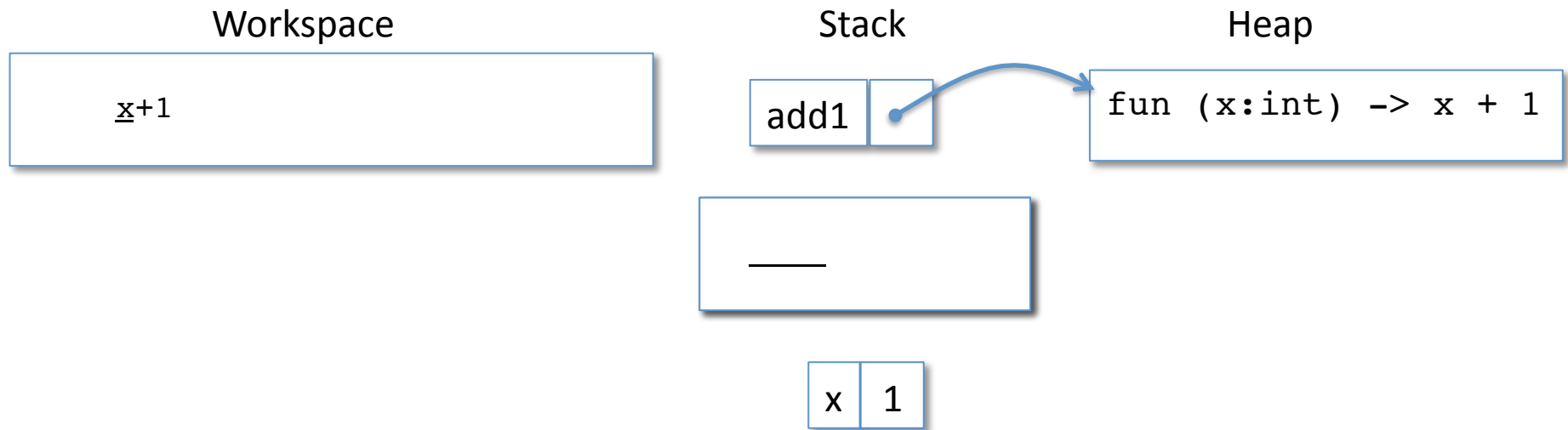
# Function Simplification



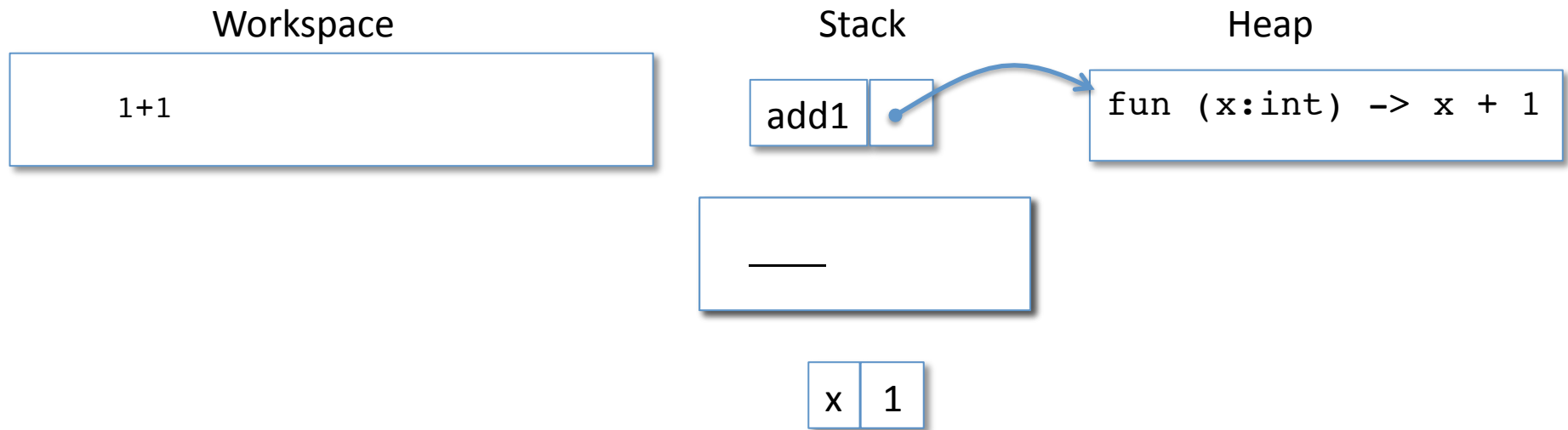
# Function Simplification



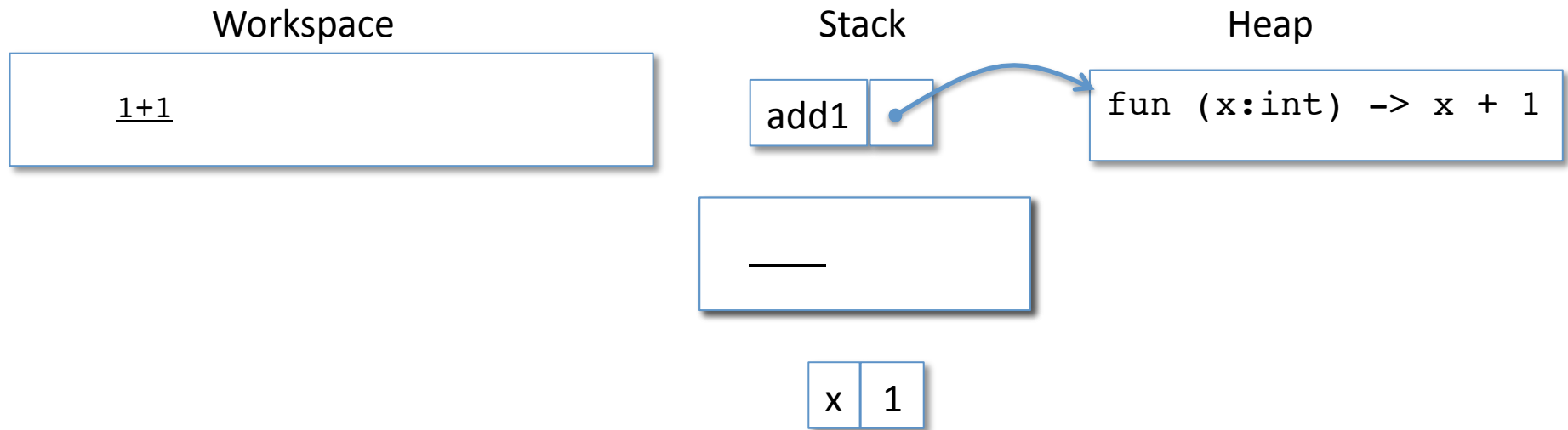
# Function Simplification



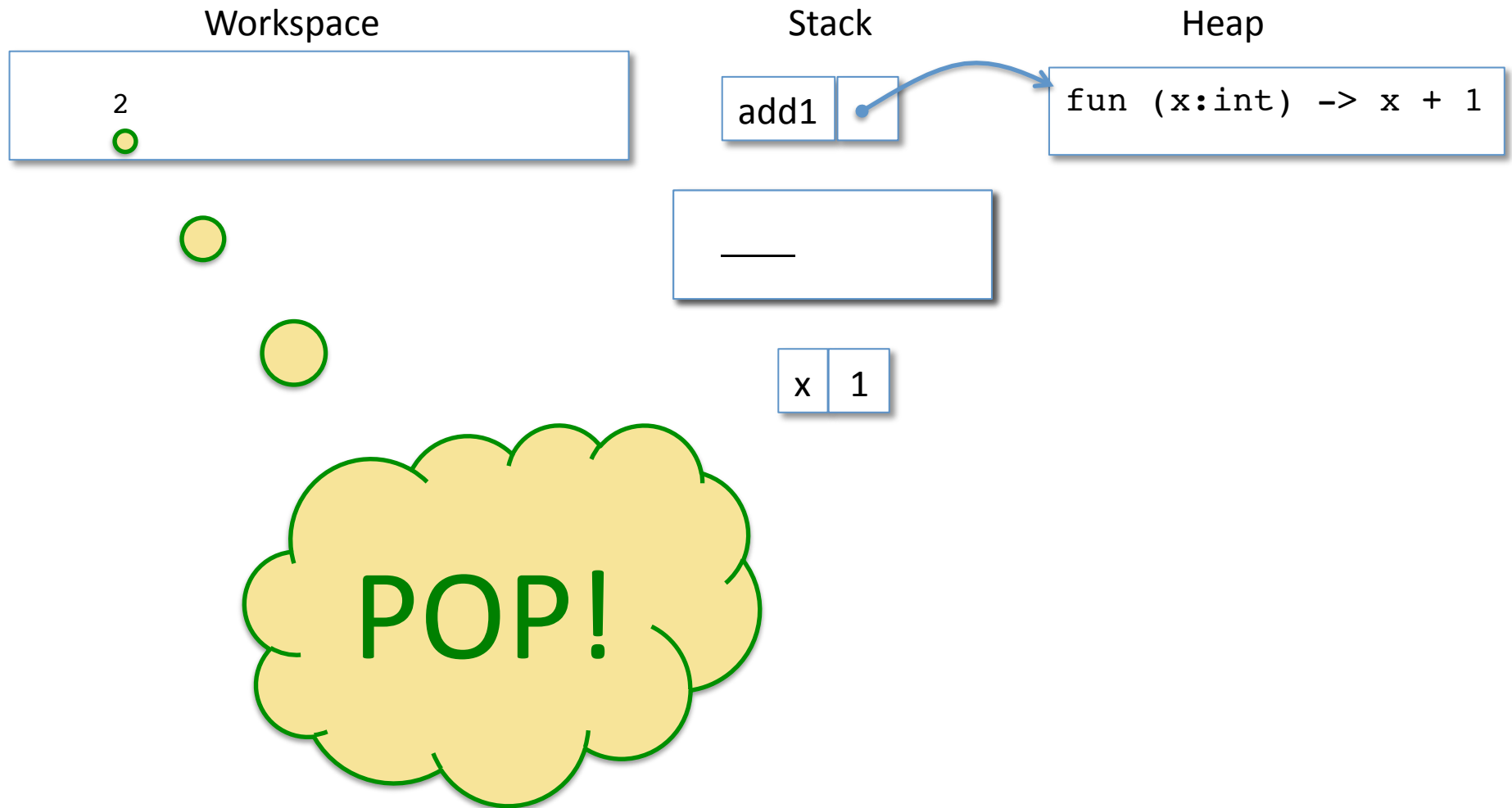
# Function Simplification



# Function Simplification

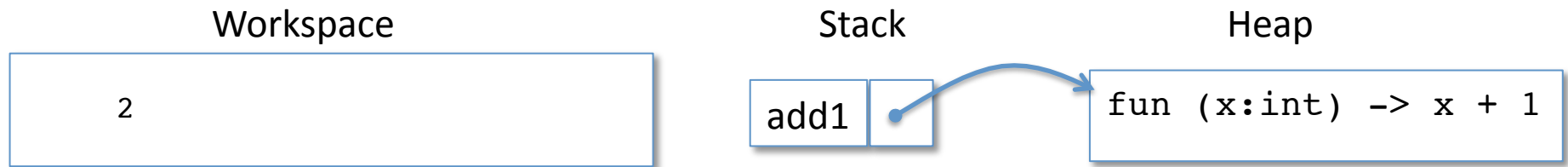


# Function Simplification





# Function Simplification



**DONE!**

# Simplifying Functions

- A function definition “let rec  $f(x_1:t_1)\dots(x_n:t_n) = e$  in body” is always ready.
  - It is simplified by replacing it with “let  $f = \text{fun } (x:t_1)\dots(x:t_n) = e$  in body”
- A function “fun  $(x_1:t_1)\dots(x_n:t_n) = e$ ” 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.