$$\text{sum } 4 \Rightarrow \text{if } \underbrace{4=0}_{\text{Evaluate this first}} \text{ then } \underbrace{0} \text{ else } \underbrace{4+\text{sum}(4-1)}$$

“Thunks”

$\Rightarrow$ if false then 0 else 4+sum(4−1)

$\Rightarrow$ 4+sum(4−1)

(+) 4 (sum ((−) 4 1))

$\Rightarrow$ 4 + sum 3

$\Rightarrow$ 4+(if 3=0 then 0 else 3+sum(3-1))

$\Rightarrow$ . . .

Buildup

$\Rightarrow$ 4 + (3 + sum 2)

$\Rightarrow$ . . .

$\Rightarrow$ 4+(3 + (2+sum 1))

$\Rightarrow$ . . .

$\Rightarrow$ 4+(3+(2+(1+0)))

↑

$\Rightarrow 4 + (3 + (2 + 1))$

$\Rightarrow 4 + (3 + 3)$

Winddown

Unfolding

$\Rightarrow 4 + 6$

$\Rightarrow 10$

↓

```
let rec sum n
    = if n=0 then 0 else n+sum(n-1)
```

$\underbrace{\text{sum(n-1)}}_{\text{Make recursive call}}$

Everything to do once the function call returns.

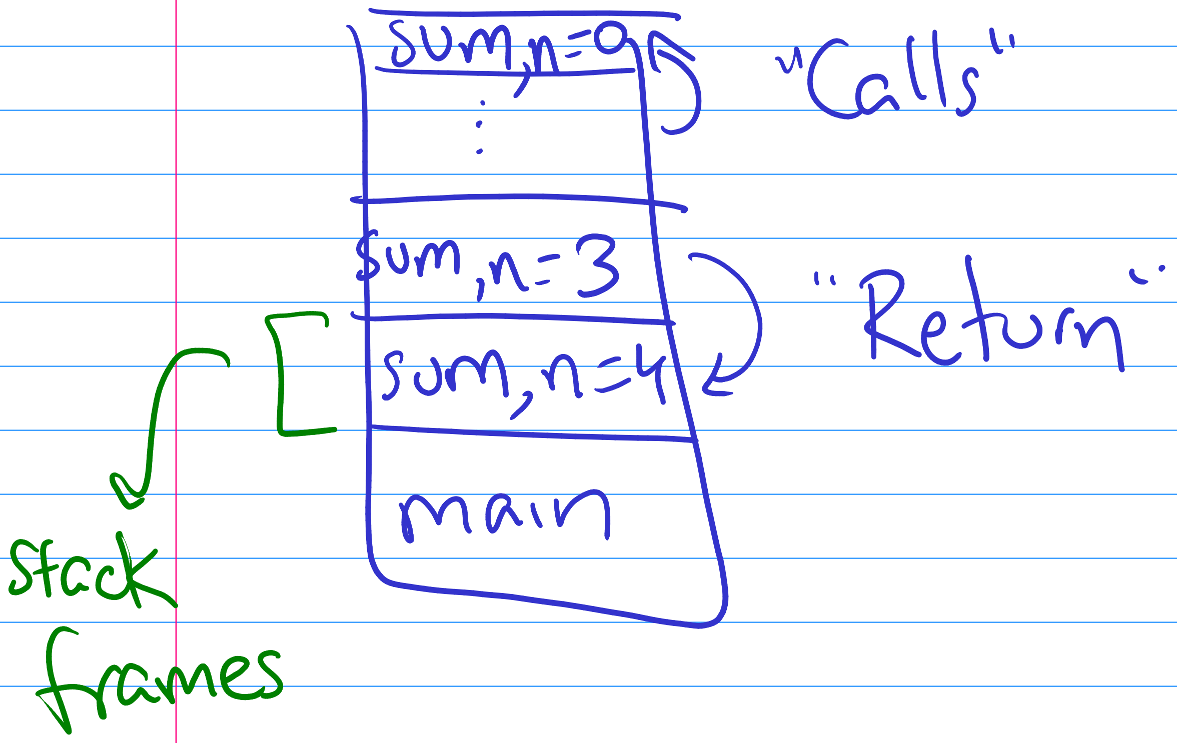
"Continuation"

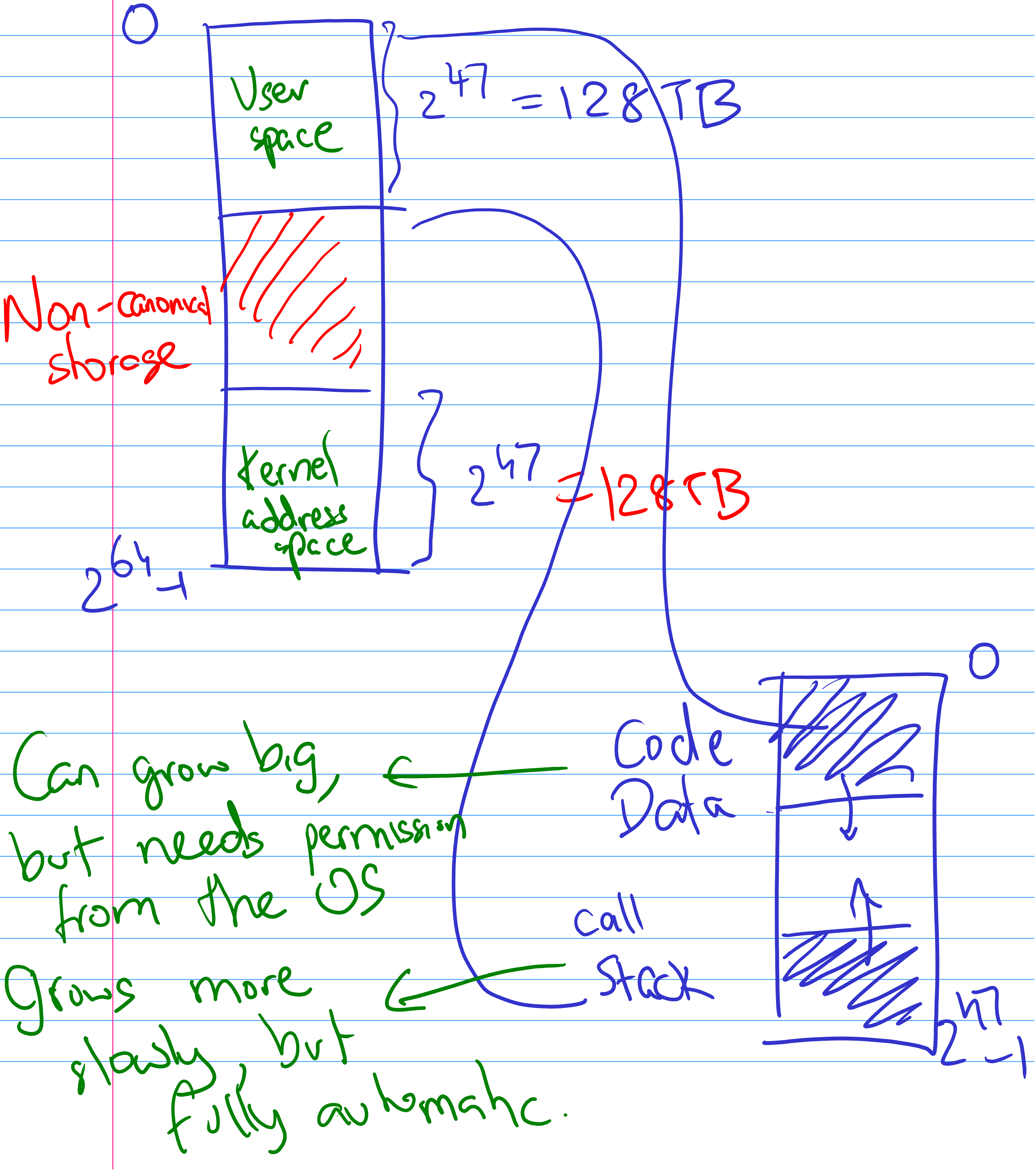
Add n to the result

Return

Continuation = Traditionally maintained as a stack

call





Can grow big, but needs permission from the OS

Grows more slowly, but fully automatic.

```
  let sum2 n =
 let rec helper acc n =
       if n=0 then acc else
          helper (n+acc) (n-1) in
  helper 0 n
```

Q1 : What does sum2 do?

Q2 : Why doesn't sum2 explode like sum?

Q1: What does sum2 do?

helper 0 4 ⇒

$\text{if } 4=0 \text{ then } 0 \text{ else } \underbrace{\text{helper } (0+4)\ (4-1)}_{\text{Thunks}}$

⇒ ... ⇒ helper (0+4) (4-1)

⇒ helper 4 3

⇒ if 3=0 then 4 else helper (4+3) (3-1)

⇒ ... ⇒ helper (4+3) (3-1)

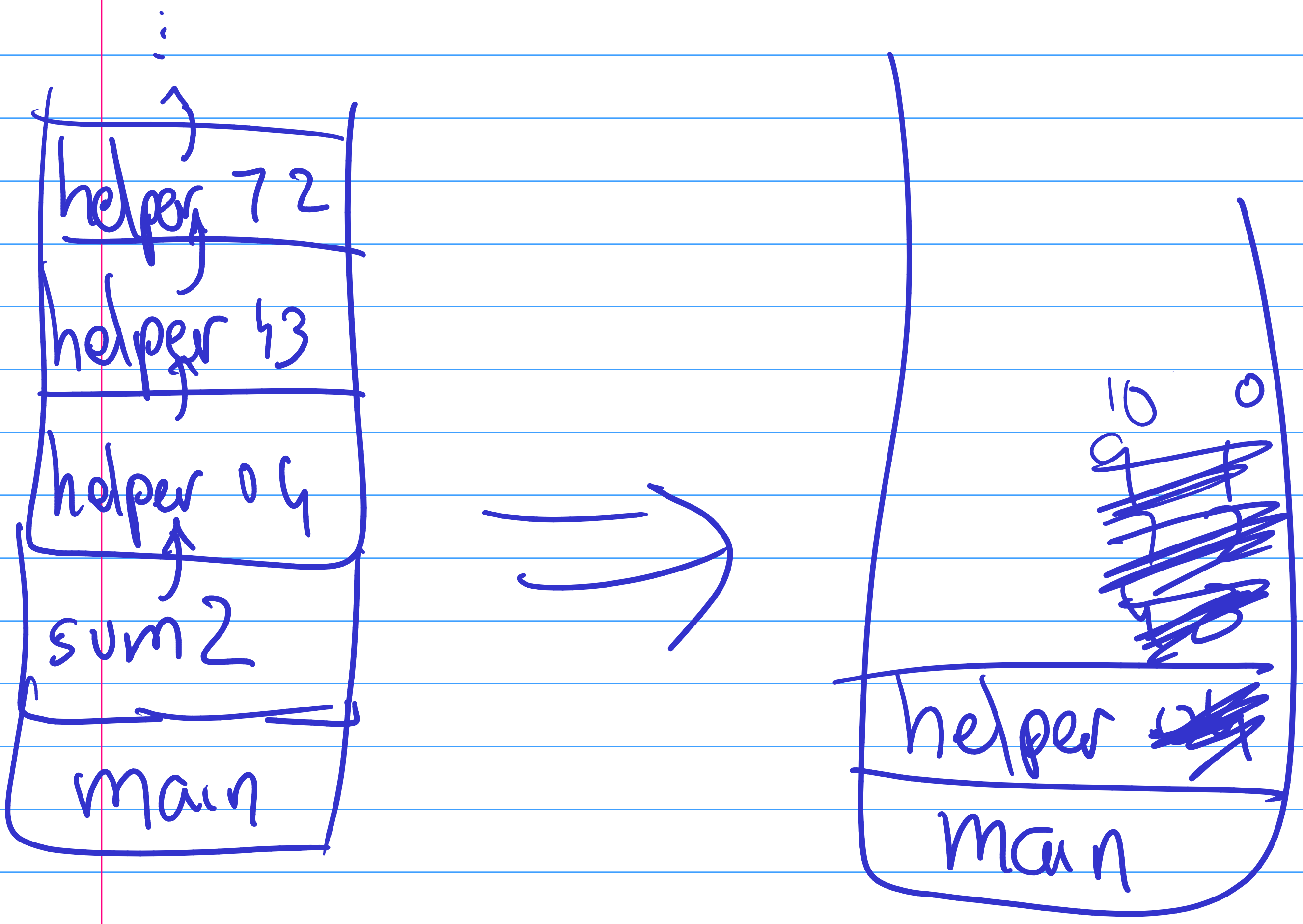
~~⇒ h 7 2~~

~~⇒ ...~~

⇒ h 9 1 ⇒ ... ⇒ h 10 0

⇒ 10

Q2: Why doesn't sum2 explode like sum?



Continuation is empty!

Nothing more to be done!

Just let the return value pass through!

If the last activity within a fn call is a call to another fn, just reuse the old stack frame.

Tail call optimization (TCO)

Iteration vs. tail calls

```
int acc = 0 ;
while (n > 0) {
    acc = n + acc ;
    n = n-1
}
```

```
let rec helper acc n
= if n = 0 then acc
  else helper (n+acc)
              (n-1)
```

Converting functions to be tail recursive

Ex : Checking if a number is even.

```
let rec even n =
  if n = 0 then true else not (even (n - 1))
```

Continuation "What needs to be done after."

Recursive call

Base case of recursion "What to do with an empty todo list"

Store continuation in a todo list

```
let even n =
  let rec helper todoList n =
    if n = 0 then todoList else helper ("not" :: todoList) (n - 1) in
  let todoList = helper [] n in
  let rec unwind todoList acc =
    match todoList with
    | [] -> acc
    | "not" :: todoList2 -> unwind todoList2 (not acc) in
  unwind afters true
```

Unwind the todo list

Both functions are tail recursive!

Jared's optimization: Better representations of the todo list

- Instead of an explicit list of "not"s, keep track of "how many" nots to apply to the base case

New todo list

```
let even n =
 let rec helper numNots n =
   if n = 0 then numNots else helper (1 + numNots) (n - 1) in
 let numNots = helper 0 n in
 let rec unwind numNots acc =
   match numNots with
   | 0 -> acc
   | _ -> unwind (numNots - 1) (not acc) in
 unwind numNots true
```