Continuation-Passing Style

COS 326
David Walker
Princeton University

Midterm Exam

Wed Oct 25, 2017

In Class (11:00-12:20)

Midterm Week

Be there or be square!

Some Innocuous Code


```
(* sum of 0..n *)

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;

let big_int = 1000000;;

sum big_int;;
```

Let’s try it.

(Go to tail.ml)

Some Other Code

Four functions: Green works on big inputs; Red doesn't.

```
let sum_to2 (n: int) : int =
  let rec aux (n:int) (a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
```

```
let rec sum2 (l:int list) : int =
  match l with
      [] -> 0
    | hd::tail -> hd + sum2 tail
```

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
```

```
let sum (l:int list) : int =
  let rec aux (l:int list) (a:int) : int =
    match l with
        [] -> a
      | hd::tail -> aux tail (a+hd)
  in
  aux l 0
```

Some Other Code

Four functions: Green works on big inputs; Red doesn't.

```
let sum_to2 (n: int) : int =
  let rec aux (n:int) (a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
```

```
let rec sum2 (l:int list) : int =
  match l with
      [] -> 0
    | hd::tail -> hd + sum2 tail
```

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
```

```
let sum (l:int list) : int =
  let rec aux (l:int list) (a:int) : int =
    match l with
        [] -> a
      | hd::tail -> aux tail (a+hd)
  in
  aux l 0
```

code that works:
no computation after recursive function call

Tail Recursion

A *tail-recursive function* does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
    sum_to 1000000
```

```
(* sum of 0..n *)

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0

let big_int = 1000000;;

sum big_int
```

Tail Recursion

A *tail-recursive function* does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
    sum_to 1000000
-->
    1000000 + sum_to 99999
```

```
(* sum of 0..n *)

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;

let big_int = 1000000;;

sum big_int;;
```

Tail Recursion

A *tail-recursive function* does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
    sum_to 1000000
-->
    1000000 + sum_to 99999
-->
    1000000 + 99999 + sum_to 99998
```

```
(* sum of 0..n *)

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;

let big_int = 1000000;;

sum big_int;;
```

expression size grows
at every recursive call ...

lots of adding to do after
the call returns"

Tail Recursion

A *tail-recursive function* does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
    sum_to 1000000
-->
    1000000 + sum_to 99999
-->
    1000000 + 99999 + sum_to 99998
-->
    ...
-->
    1000000 + 99999 + 99998 + ... + sum_to 0
```

```
(* sum of 0..n *)

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;

let big_int = 1000000;;

sum big_int;;
```

Tail Recursion

A *tail-recursive function* does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
    sum_to 1000000
-->
    1000000 + sum_to 99999
-->
    1000000 + 99999 + sum_to 99998
-->
    ...
-->
    1000000 + 99999 + 99998 + ... + sum_to 0
-->
    1000000 + 99999 + 99998 + ... + 0
```

```
(* sum of 0..n *)

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;

let big_int = 1000000;;

sum big_int;;
```

recursion
finally bottoms out

Tail Recursion

A *tail-recursive function* does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

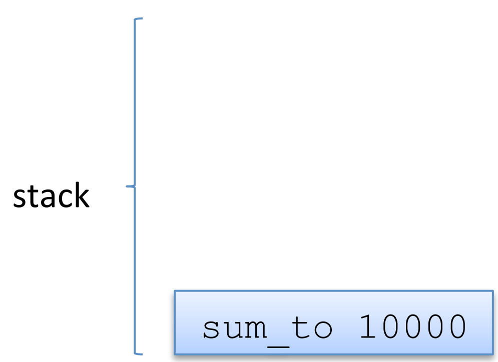
```
    sum_to 1000000
-->
    1000000 + sum_to 99999
-->
    1000000 + 99999 + sum_to 99998
-->
    ...
-->
    1000000 + 99999 + 99998 + ... + sum_to 0
-->
    1000000 + 99999 + 99998 + ... + 0
-->
    ... add it all back up ...
```

```
(* sum of 0..n *)

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;

let big_int = 1000000;;

sum big_int;;
```

do a long series of additions to get back an int

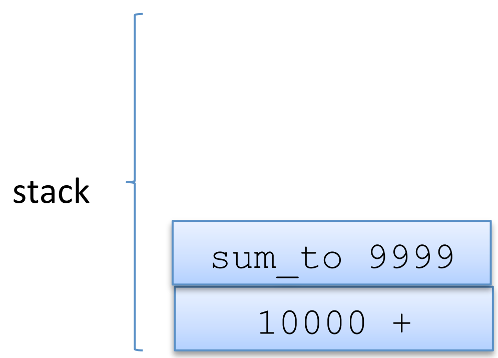
Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```



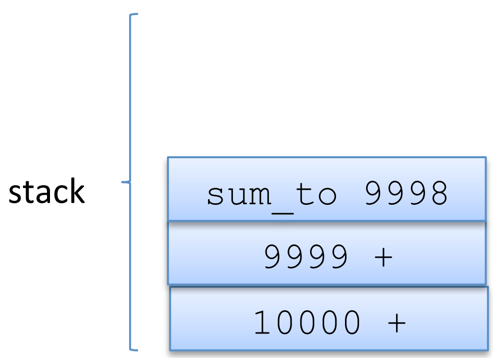
Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```



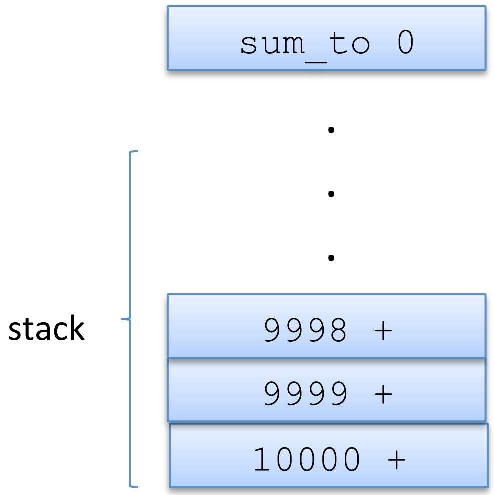
Non-tail recursive


```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```



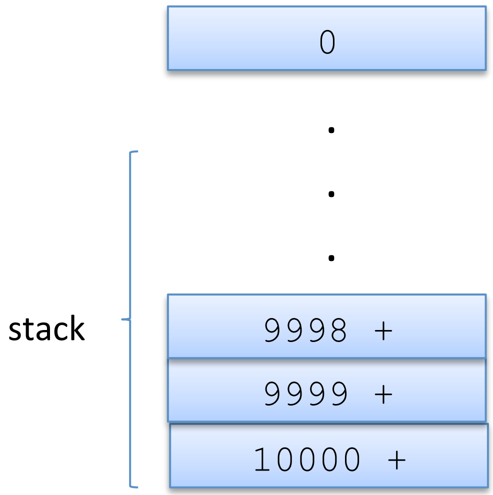
Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```



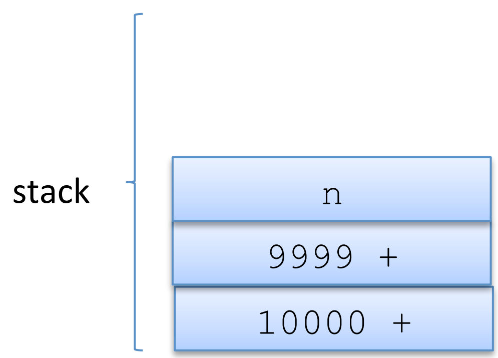
Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```



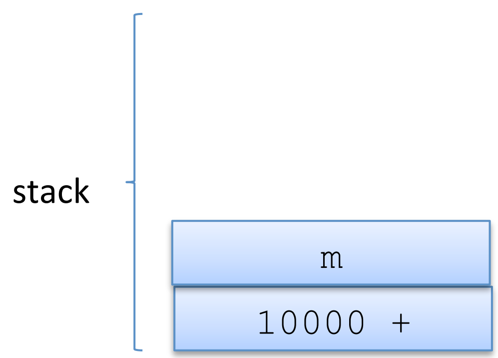
Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```



Non-tail recursive

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```



Non-tail recursive

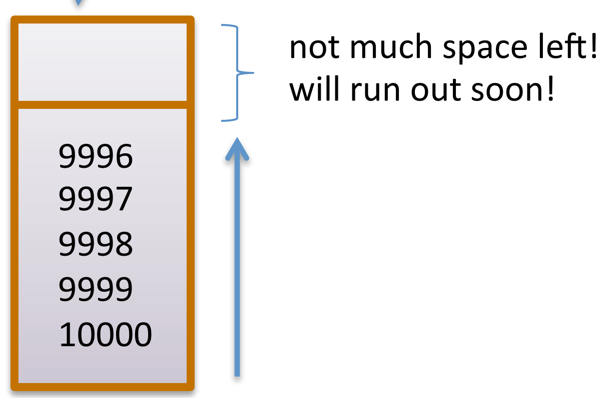

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```

stack

result

Data Needed on Return Saved on Stack

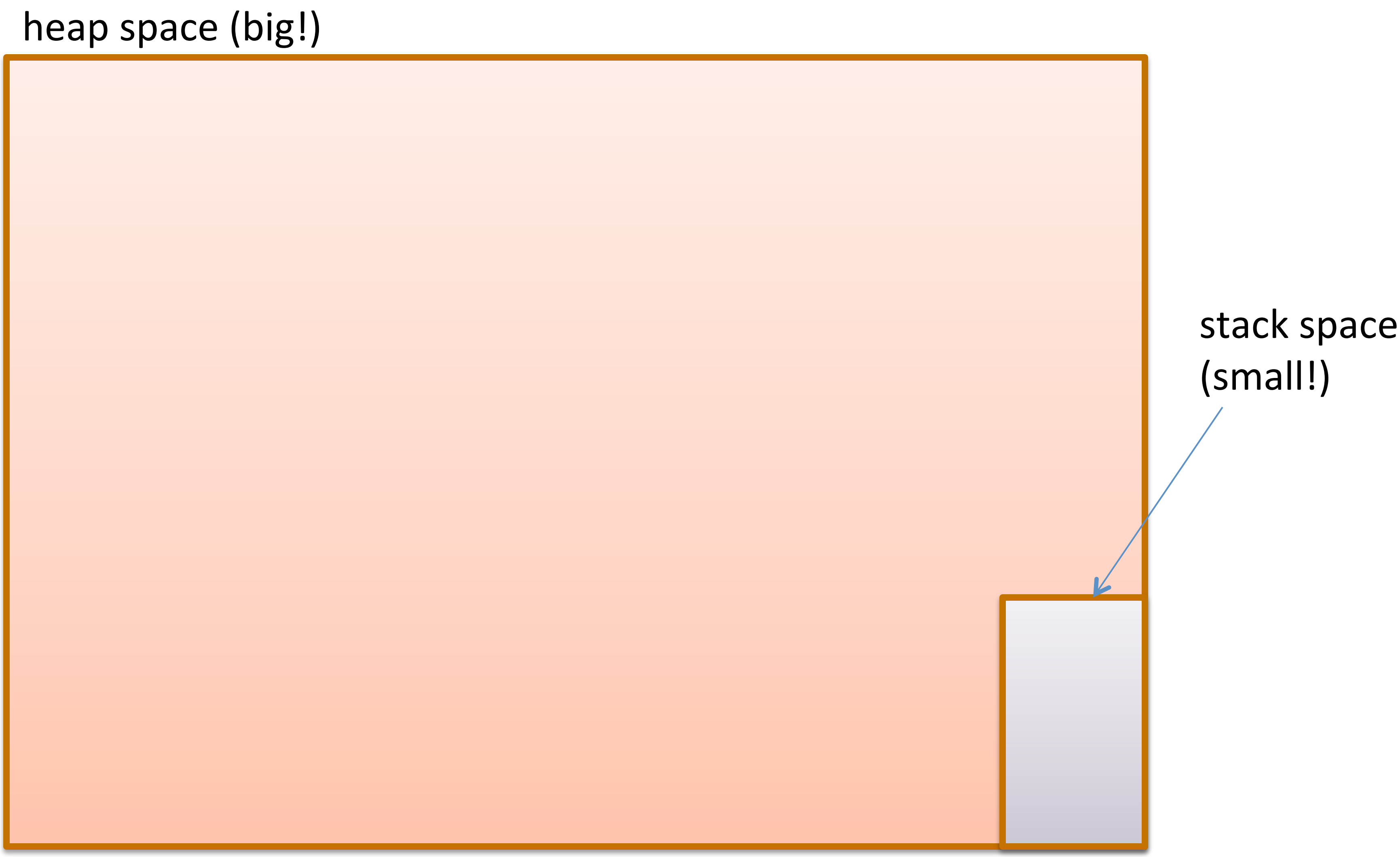

```
    sum_to 10000
-->
    ...
-->
    10000 + 9999 + 9998 + 9997 + ... +
-->
    ...
-->
    ...
```



the stack

every non-tail call puts the data from the calling context on the stack

Memory is partitioned: Stack and Heap



Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

```
  sum_to2 1000000
```

```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int)
                : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```

Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

```
    sum_to2 1000000
-->
    aux 1000000 0
```

```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int)
                 : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```

Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

```
    sum_to2 1000000
-->
    aux 1000000 0
-->
    aux 99999 1000000
```

```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int)
                  : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```

Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

```
    sum_to2 1000000
-->
    aux 1000000 0
-->
    aux 99999 1000000
-->
    aux 99998 1999999
```

```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int)
                 : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```

Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

```
    sum_to2 1000000
-->
    aux 1000000 0
-->
    aux 99999 1000000
-->
    aux 99998 1999999
-->
    ...
-->
    aux 0 (-363189984)
-->
    -363189984
```

```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int)
                : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```

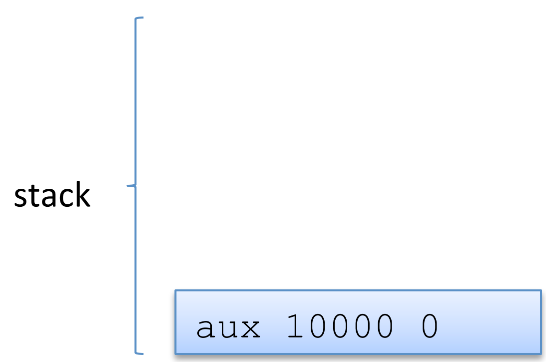
constant size expression
in the substitution model

(addition overflow occurred
at some point)

Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

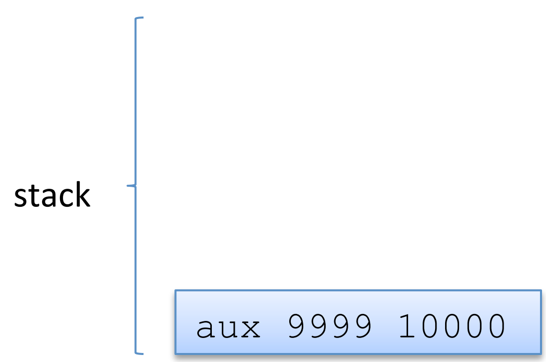
```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int)
                    : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```



Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

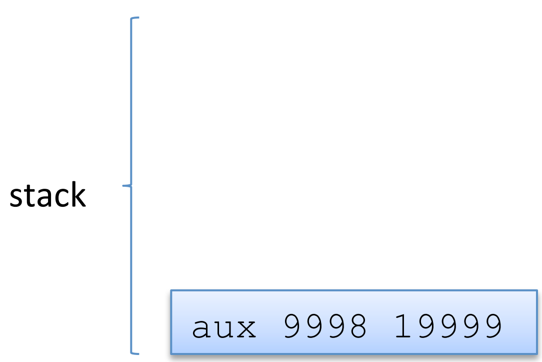
```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int)
                  : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```



Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int)
                  : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```



Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int)
                : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```

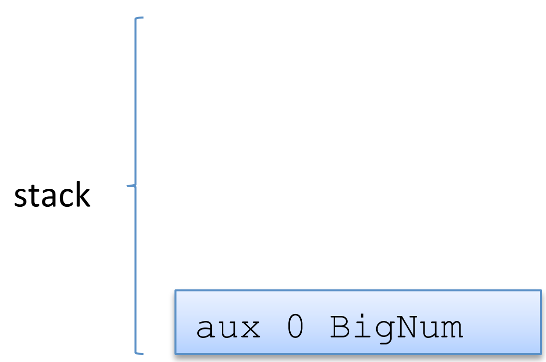
stack

```
aux 9997 29998
```

Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int)
                    : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```



Question

We used human ingenuity to do the tail-call transform.

Is there a mechanical procedure to transform *any* recursive function in to a tail-recursive one?

not only is sum2 tail-recursive but it reimplements an algorithm that took *linear space* (on the stack) using an algorithm that executes in *constant space*!

```
let rec sum_to (n: int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;
```

human ingenuity

```
let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```

CONTINUATION-PASSING STYLE
CPS!

CPS

CPS:

- Short for *Continuation-Passing Style*
- Every function takes a *continuation* (a function) as an argument that expresses "what to do next"
- CPS functions only call other functions as the last thing they do
- All CPS functions are tail-recursive

Goal:

- Find a mechanical way to translate any function in to CPS

Serial Killer or PL Researcher?

Serial Killer or PL Researcher?

Gordon Plotkin
Programming languages researcher
Invented CPS conversion.

Call-by-Name, Call-by Value
and the Lambda Calculus. TCS, 1975.

Robert Garrow
Serial Killer

Killed a teenager at a campsite
in the Adirondacks in 1974.
Confessed to 3 other killings.

Serial Killer or PL Researcher?

Gordon Plotkin
Programming languages researcher
Invented CPS conversion.

Call-by-Name, Call-by Value
and the Lambda Calculus. TCS, 1975.

Robert Garrow
Serial Killer

Killed a teenager at a campsite
in the Adirondacks in 1974.
Confessed to 3 other killings.

Question

Can any non-tail-recursive function be transformed in to a tail-recursive one? Yes, if we can capture the *differential* between a tail-recursive function and a non-tail-recursive one.

```
let rec sum (l:int list) : int =
  match l with
    [] -> 0
  | hd::tail -> hd + sum tail
;;
```

Idea: Focus on what happens after the recursive call.

Question

Can any non-tail-recursive function be transformed in to a tail-recursive one? Yes, if we can capture the *differential* between a tail-recursive function and a non-tail-recursive one.

```
let rec sum (l:int list) : int =
  match l with
    [] -> 0
  | hd::tail -> hd + sum tail
;;
```

what happens next

Idea: Focus on what happens after the recursive call.

Extracting that piece:

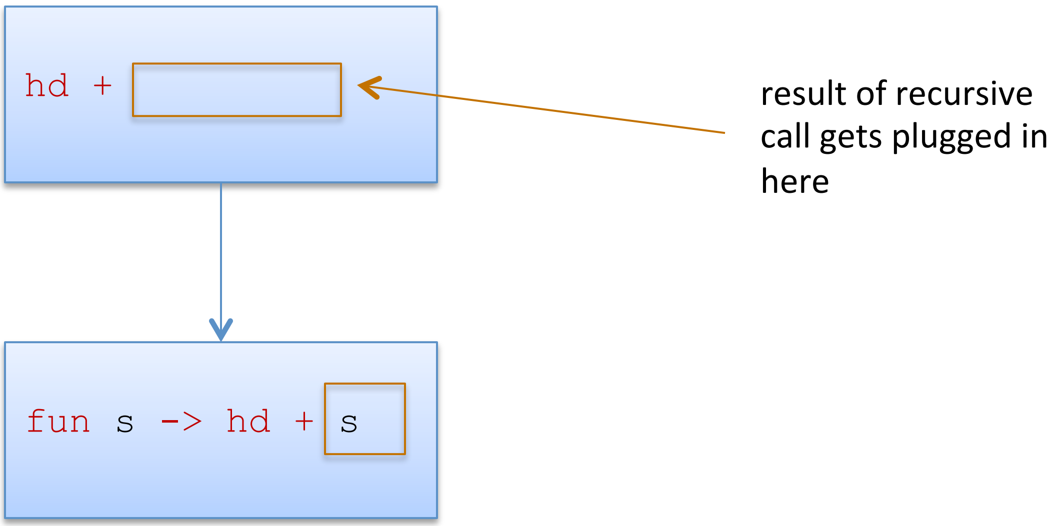
```
hd + [          ]
```

result of recursive call gets plugged in here

How do we capture it?

Question

How do we capture that computation?



Question

How do we capture that computation?

```
hd + [          ]
```

```
fun s -> hd + [s]
```

```
let rec sum (l:int list) : int =
  match l with
    [] -> 0
  | hd::tail -> hd + sum tail
;;
```

```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> ???) ;;
```

Question

How do we capture that computation?

```
hd + [    ]
```

```
fun s -> hd + [s]
```

```
let rec sum (l:int list) : int =
  match l with
    [] -> 0
  | hd::tail -> hd + [sum tail]
;;
```

```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;
```

Question

How do we capture that computation?

```
hd + [        ]
```

```
fun s -> hd + [s]
```

```
let rec sum (l:int list) : int =
  match l with
    [] -> 0
  | hd::tail -> hd + [sum tail]
;;
```

```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = ??
```

Question

How do we capture that computation?

```
hd +
```

```
fun s -> hd + s
```

```
let rec sum (l:int list) : int =
  match l with
    [] -> 0
  | hd::tail -> hd + sum tail
;;
```

```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

Execution


```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

```
    sum [1;2]
```

Execution


```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

```
    sum [1;2]
-->
    sum_cont [1;2] (fun s -> s)
```

Execution


```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

```
    sum [1;2]
-->
    sum_cont [1;2] (fun s -> s)
-->
    sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
```

Execution


```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

```
    sum [1;2]
-->
    sum_cont [1;2] (fun s -> s)
-->
    sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
-->
    sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
```

Execution


```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

```
    sum [1;2]
-->
    sum_cont [1;2] (fun s -> s)
-->
    sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
-->
    sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
-->
    (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s)) 0
```

Execution


```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

```
    sum [1;2]
-->
    sum_cont [1;2] (fun s -> s)
-->
    sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
-->
    sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
-->
    (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s)) 0
-->
    (fun s -> (fun s -> s) (1 + s)) (2 + 0))
```

Execution


```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

```
    sum [1;2]
-->
    sum_cont [1;2] (fun s -> s)
-->
    sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
-->
    sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
-->
    (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s)) 0
-->
    (fun s -> (fun s -> s) (1 + s)) (2 + 0))
-->
    (fun s -> s) (1 + (2 + 0))
```

Execution


```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

```
    sum [1;2]
-->
    sum_cont [1;2] (fun s -> s)
-->
    sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
-->
    sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
-->
    (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s)) 0
-->
    (fun s -> (fun s -> s) (1 + s)) (2 + 0))
-->
    (fun s -> s) (1 + (2 + 0))
-->
    1 + (2 + 0)
-->
    3
```

Question


```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

```
    sum [1;2]
-->
    sum_cont [1;2] (fun s -> s)
-->
    sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
-->
    sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
-->
    ...
-->
    3
```

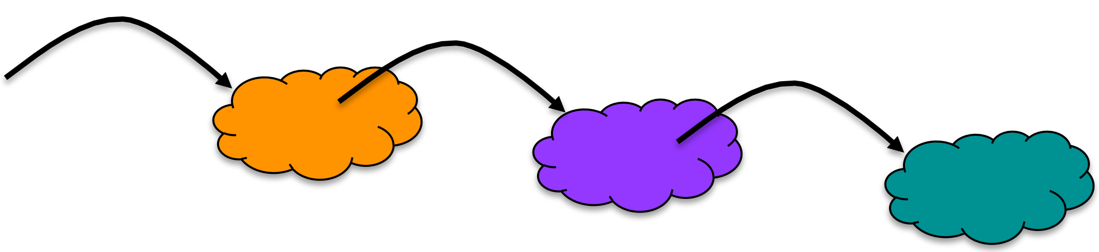
Where did the stack space go?

```
sum_cont []
  (fun s3 ->
    (fun s2 ->
      (fun s1 -> s1) (hd1 + s2)
    ) (hd2 + s3)
  )
```

function inside
function inside
function inside
expression

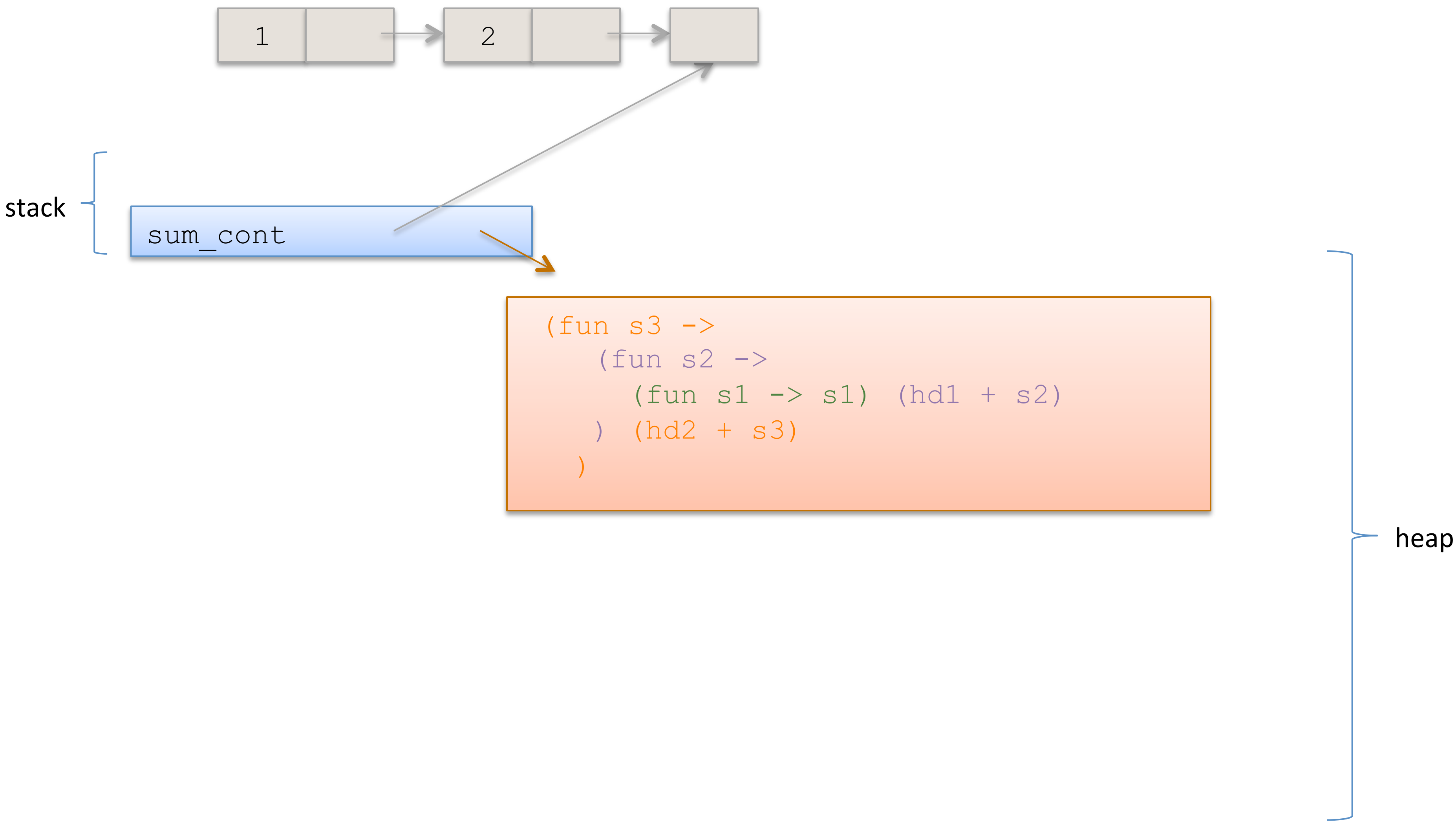
each function
is a closure;
points to the
closure inside it

a stack of
closures on
the heap



function inside function inside function inside expression → a stack of closures on the heap

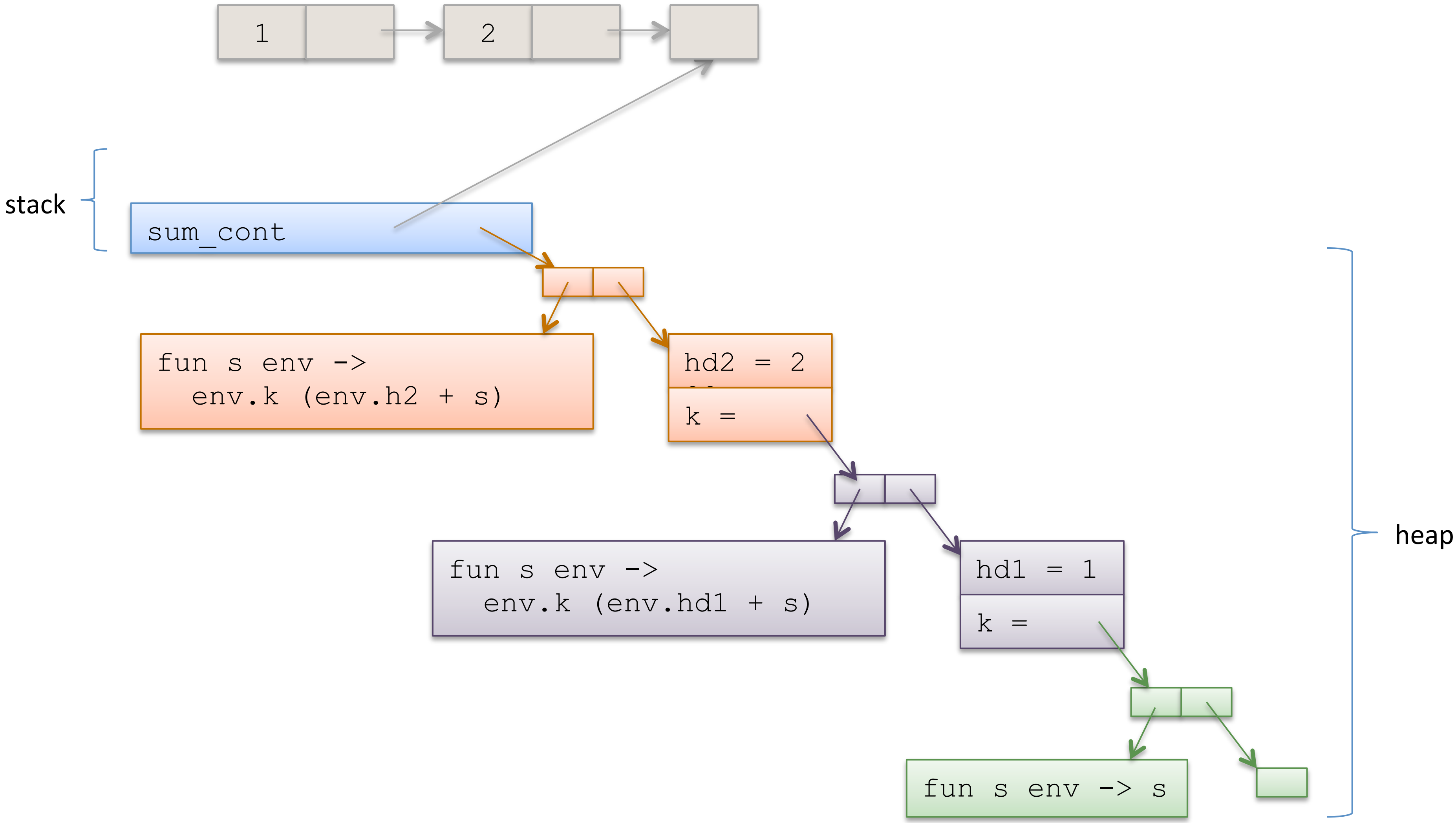
```
sum_cont []
  (fun s3 ->
    (fun s2 ->
      (fun s1 -> s1) (hd1 + s2)
    ) (hd2 + s3)
  )
```



function inside
function inside
function inside
expression

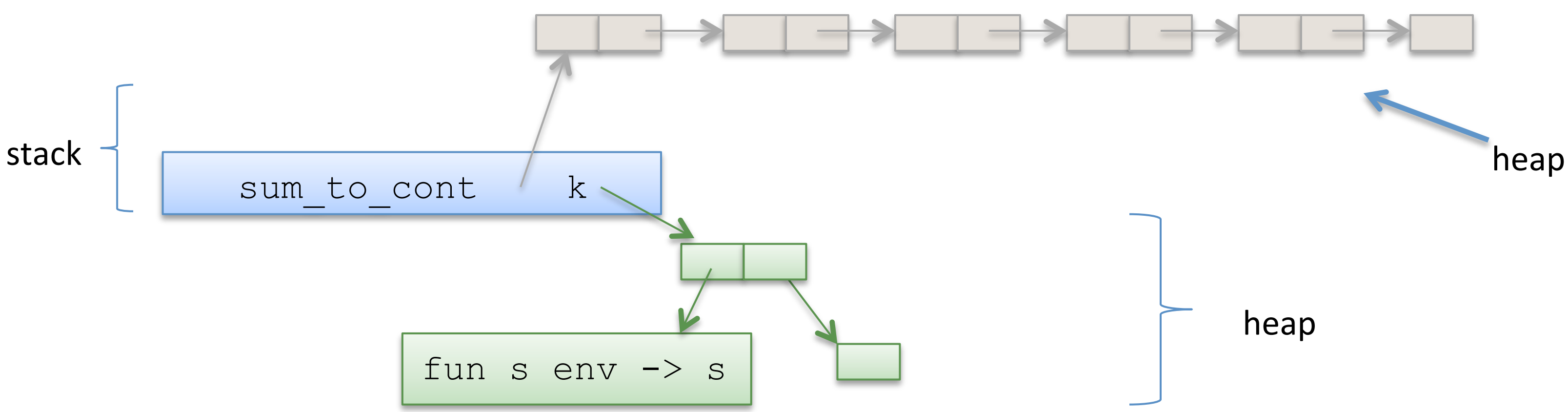
a stack of
closures on
the heap

```
sum_cont []
  (fun s3 ->
    (fun s2 ->
      (fun s1 -> s1) (hd1 + s2)
    ) (hd2 + s3)
  )
```

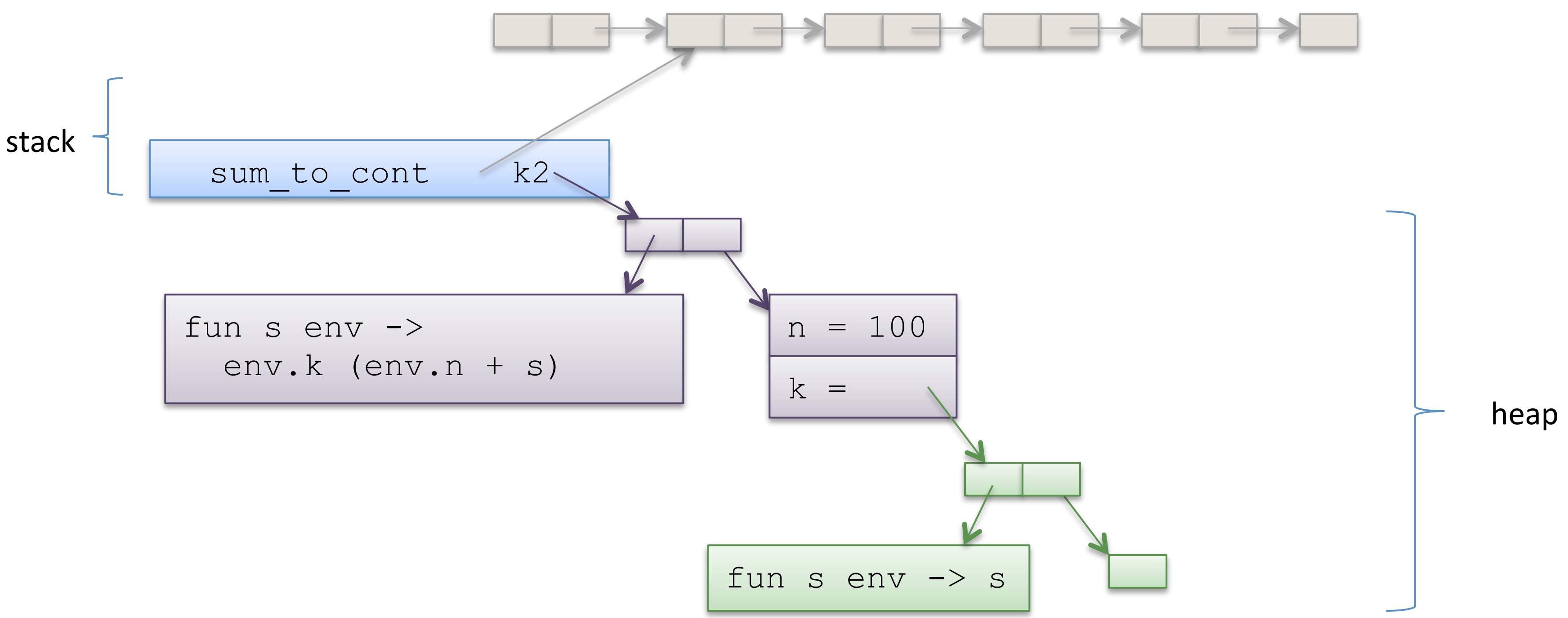


Continuation-passing style


```
let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;
```

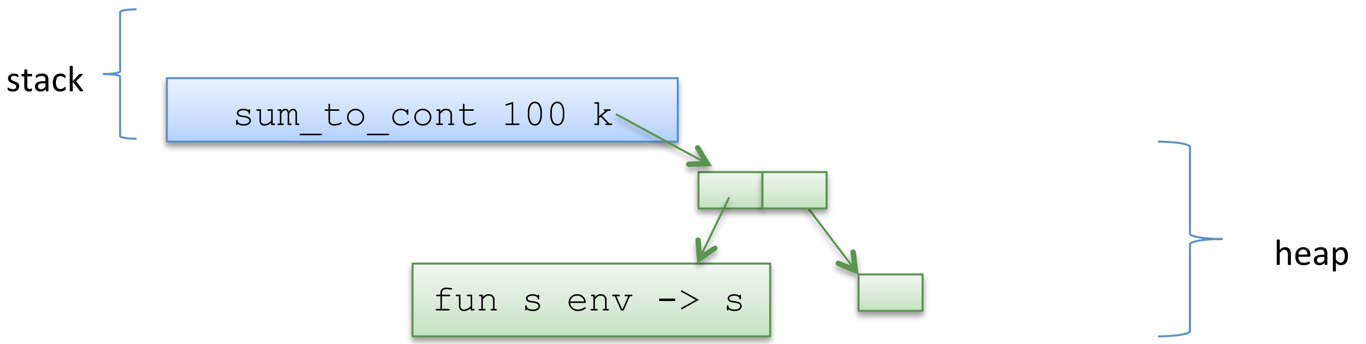


Continuation-passing style



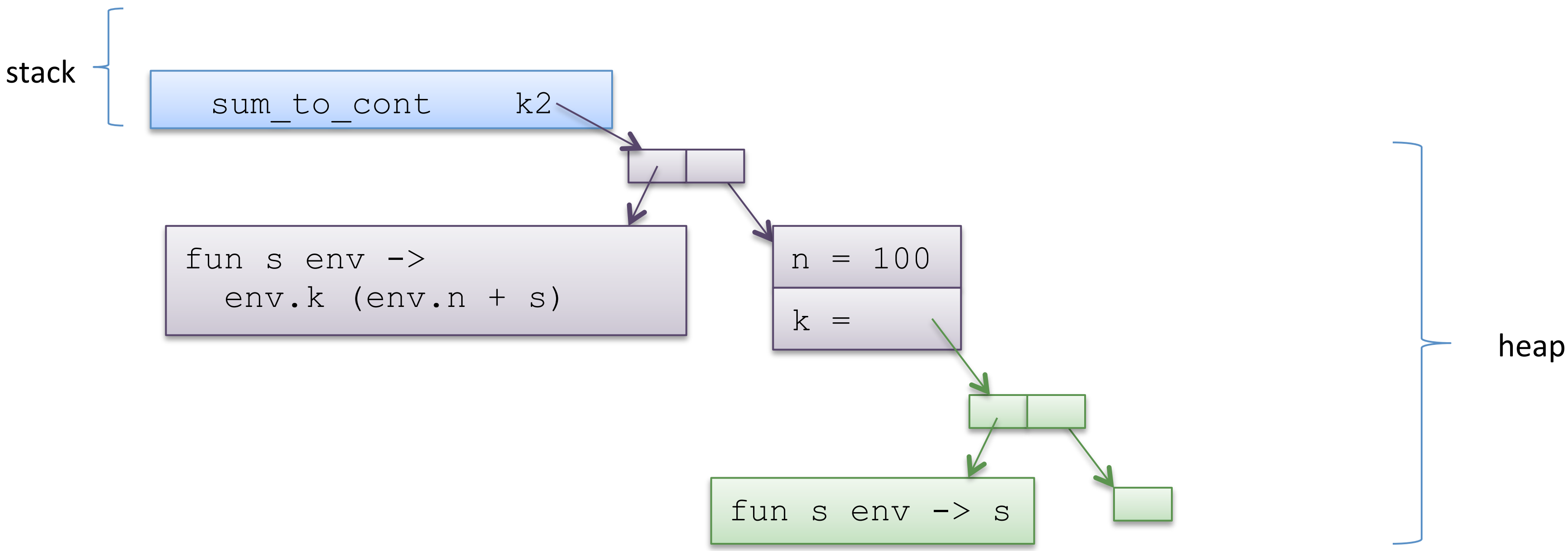
Continuation-passing style


```
let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    k 0  ;;

sum_to_cont 100 (fun s -> s)
```



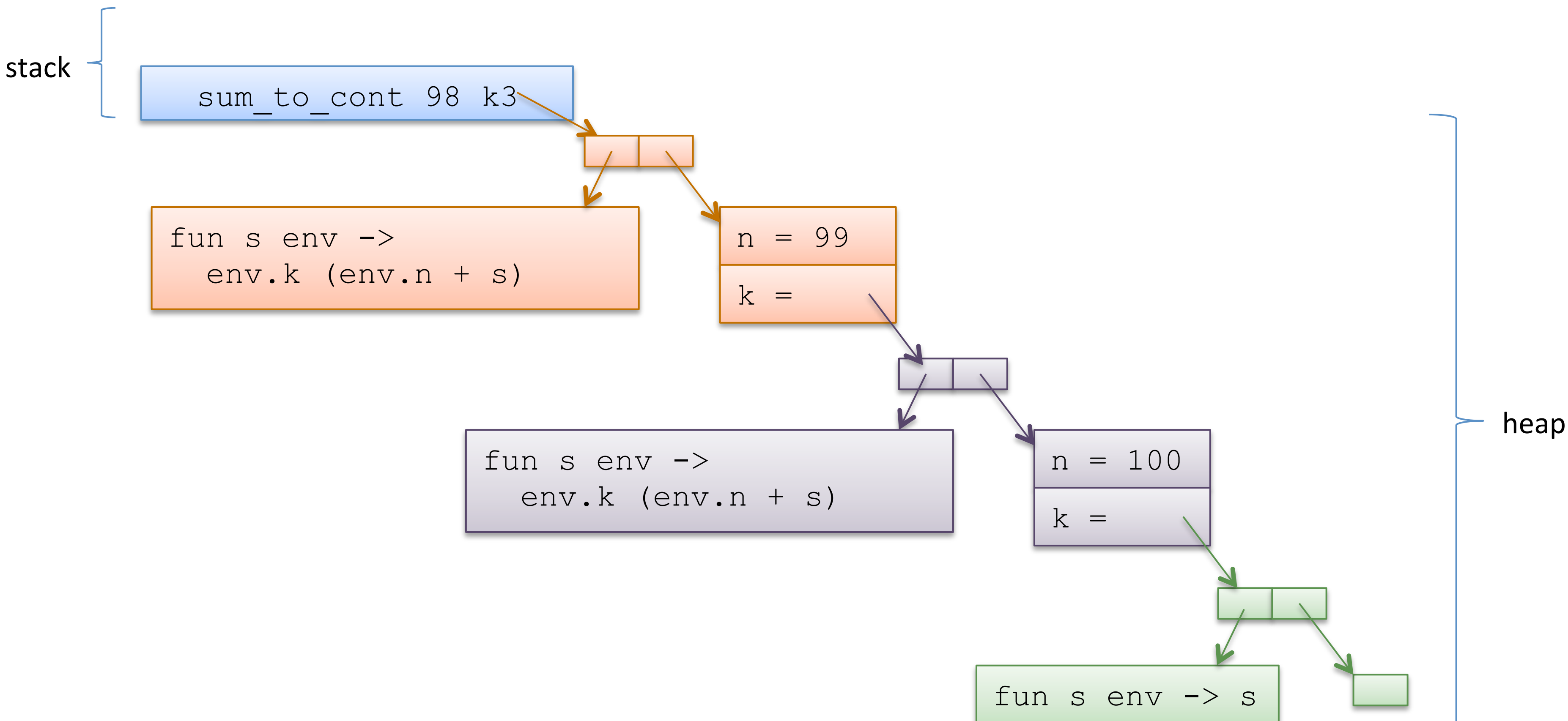
Continuation-passing style


```
let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    k 0  ;;

sum_to_cont 100 (fun s -> s)
```



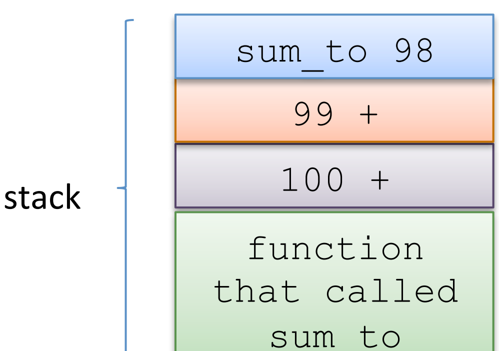
Continuation-passing style


```
let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    k 0  ;;

sum_to 100 (fun s -> s)
```



Back to stacks

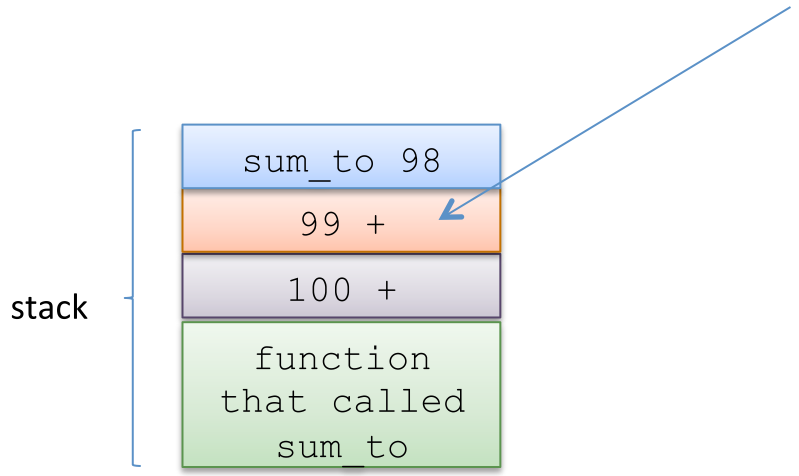
```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



Back to stacks

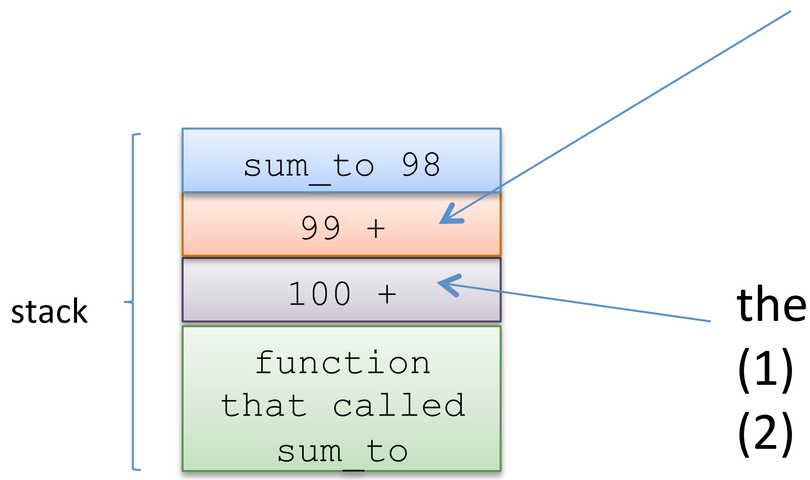
```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```

but how do you really implement that?



Back to stacks


```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



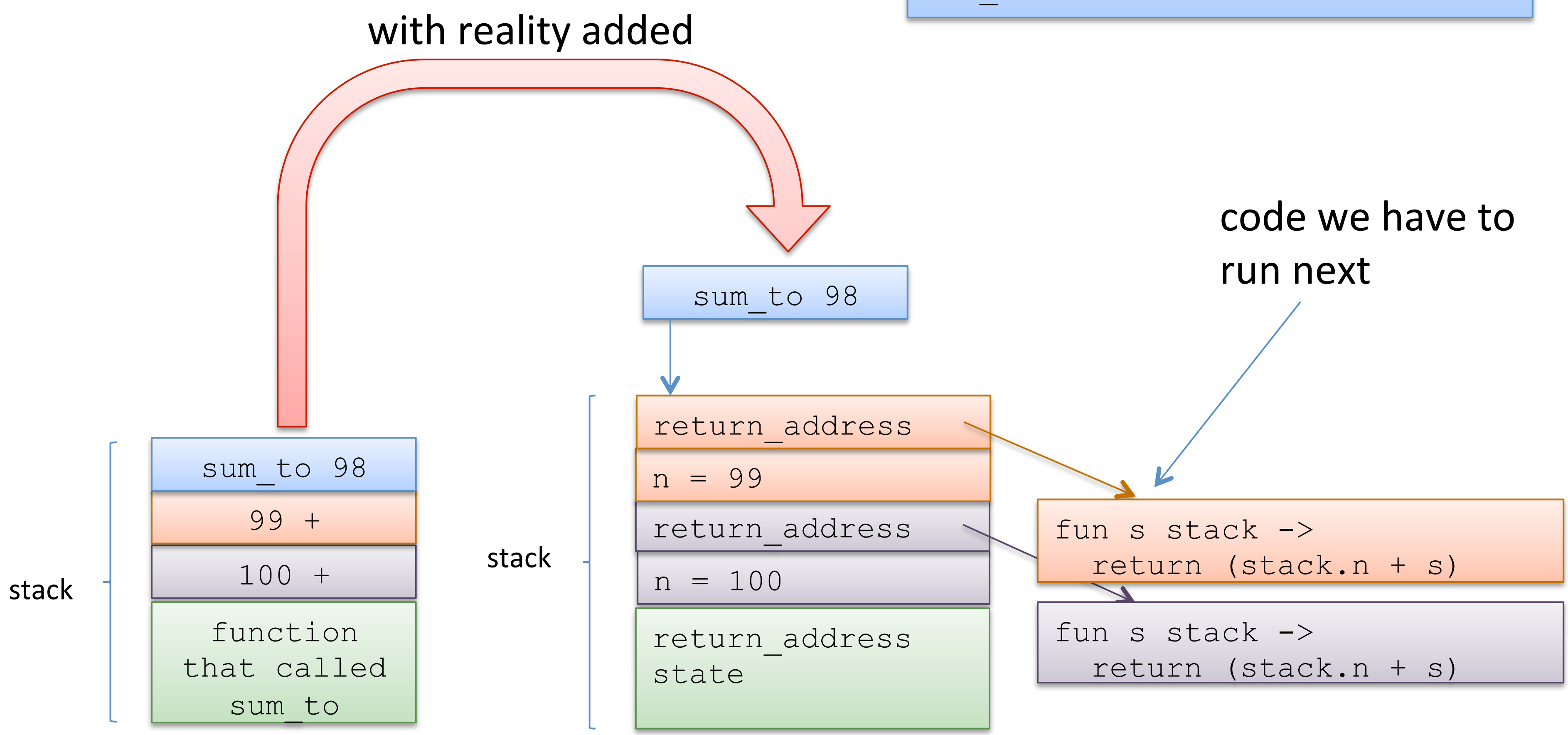
but how do you really implement that?

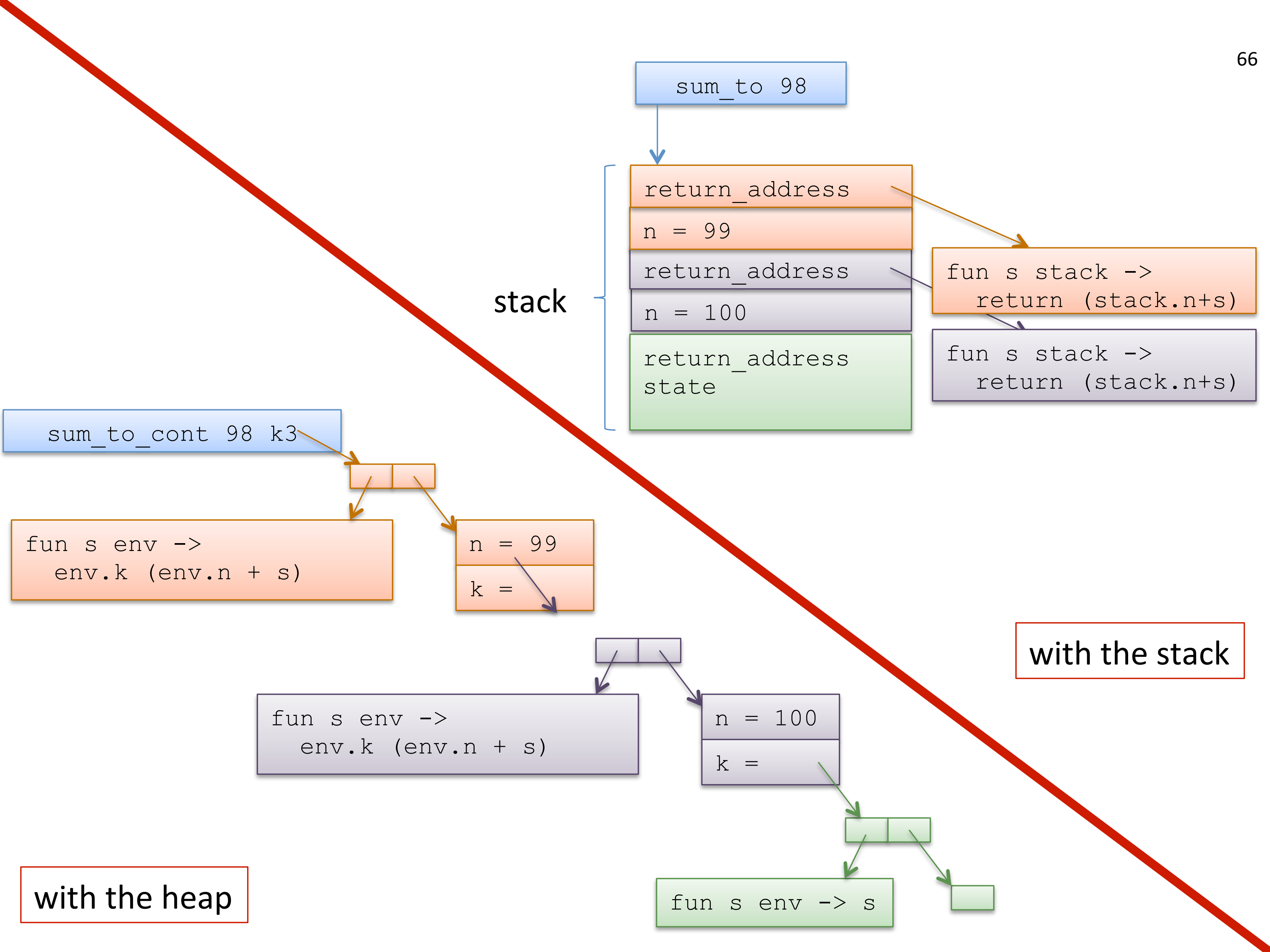
there is two bits of information here:

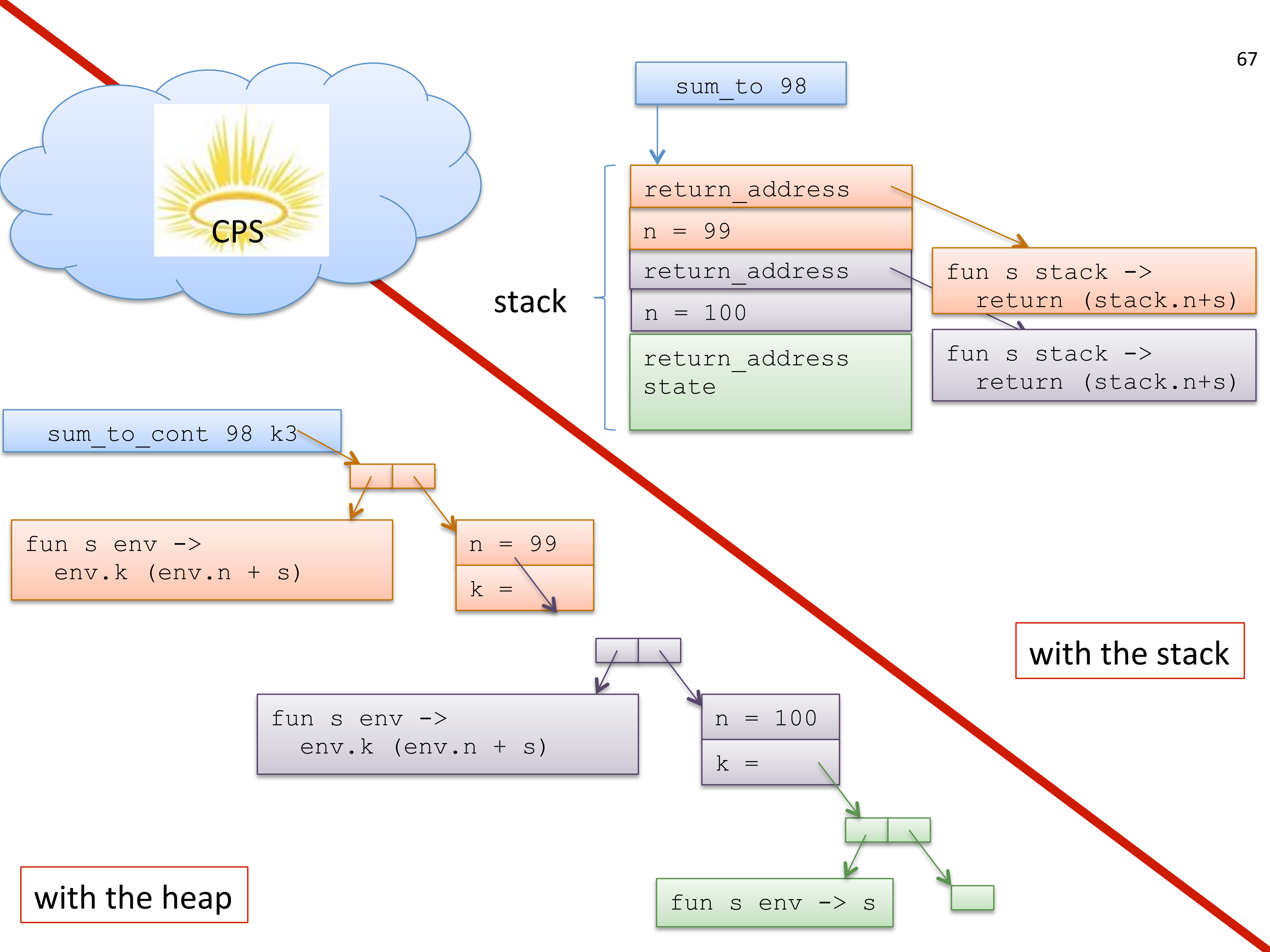
(1) some state (n=100) we had to remember
(2) some code we have to run later

Back to stacks


```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```




sum_to 98
stack
return_address
n = 99
return_address
n = 100
return_address
state
fun s stack ->
return (stack.n+s)
fun s stack ->
return (stack.n+s)
sum_to_cont 98 k3
fun s env ->
env.k (env.n + s)
n = 99
k =
fun s env ->
env.k (env.n + s)
n = 100
k =
fun s env -> s
with the stack
with the heap


CPS
sum_to 98
stack
return_address
n = 99
return_address
n = 100
return_address
state
fun s stack ->
return (stack.n+s)
fun s stack ->
return (stack.n+s)
with the stack
sum_to_cont 98 k3
fun s env ->
env.k (env.n + s)
n = 99
k =
fun s env ->
env.k (env.n + s)
n = 100
k =
fun s env -> s
with the heap

Why CPS?

Continuation-passing style is *inevitable*.

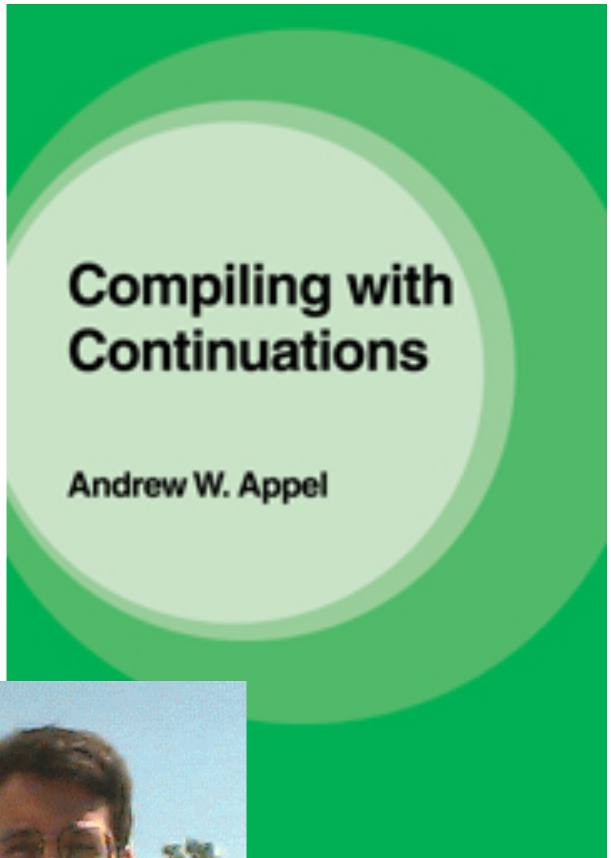
It does not matter whether you program in Java or C or OCaml -- there's code around that tells you "*what to do next*"

- If you explicitly CPS-convert your code, "*what to do next*" is stored on the heap
- If you don't, it's stored on the stack

If you take a conventional compilers class, the continuation will be called a *return address* (but you'll know what it really is!)

The idea of a *continuation* is much more general!

Standard ML of New Jersey



Your compiler can put all the continuations in the heap so you don't have to (and you don't run out of stack space)!

Other pros:

- light-weight concurrent threads

Some cons:

- hardware architectures optimized to use a stack
- need tight integration with a good garbage collector

see
Empirical and Analytic Study of Stack versus Heap Cost for Languages with Closures. Shao & Appel

Call-backs: Another use of continuations

Call-backs:

```
request_url : url -> (html -> 'a) -> 'a

request_url "http://www.s.com/i.html" (fun html -> process html)
```

continuation

Overall Summary

We developed techniques for reasoning about the space costs of functional programs

- the cost of *manipulating data types* like tuples and trees
- the cost of allocating and *using function closures*
- the cost of *tail-recursive* and non-tail-recursive *functions*

We also talked about some important program transformations:

- *closure conversion* makes nested functions with free variables in to pairs of closed code and environment
- the *continuation-passing style* (CPS) transformation turns non-tail-recursive functions in to tail-recursive ones that use no stack space
  - the stack gets moved in to the function closure
- since stack space is often small compared with heap space, it is often necessary to use *continuations and tail recursion*
  - but full CPS-converted programs are unreadable: use judgement

Challenge: CPS Convert the incr function


```
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) -> Node (i+j, incr left i, incr right i)
;;
```

Hint 1: introduce one let expression for each function call:

let x = incr left i in ...

Hint 2: you will need two continuations

CPS Convert the incr function


```
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) -> Node (i+j, incr left i, incr right i)
;;
```

```
type cont = tree -> tree ;;

let rec incr_cps (t:tree) (i:int) (k:cont) : tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) -> ...
;;
```

```
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) -> Node (i+j, incr left i, incr right i)
;;
```

first continuation:

```
Node (i+j, ___________ , incr right i)
```

second continuation:

```
Node (i+j, left_done, ______________ )
```

```
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) -> Node (i+j, incr i left, incr i right)
;;
```

first continuation:

```
fun left_done -> Node (i+j, left_done , incr right i)
```

second continuation:

```
fun right_done -> k (Node (i+j, left_done, right_done))
```

```
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) -> Node (i+j, incr left i, incr right i)
;;
```

second continuation

inside

first continuation:

```
fun left_done ->
  let k2 =
    (fun right_done ->
       k (Node (i+j, left_done, right_done))
    )
  in
  incr right i k2
```

```
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) -> Node (i+j, incr left i, incr right i)
;;
```

```
type cont = tree -> tree ;;

let rec incr_cps (t:tree) (i:int) (k:cont) : tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) ->
      let k1 = (fun left_done ->
                  let k2 = (fun right_done ->
                              k (Node (i+j, left_done, right_done)))
                  in
                  incr_cps right i k2
                )
      in
      incr_cps left i k1
;;


let incr_tail (t:tree) (i:int) : tree = incr_cps t i (fun t -> t);;
```

CORRECTNESS OF A CPS TRANSFORM

Are the two functions the same?

```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum2 (l:int list) : int = sum_cont l (fun s -> s)
```

```
let rec sum (l:int list) : int =
  match l with
    [] -> 0
  | hd::tail -> hd + sum tail
;;
```

Here, it is really pretty tricky to be sure you've done it right if you don't prove it. Let's try to prove this theorem and see what happens:

```
for all l:int list,
  sum_cont l (fun x -> x) == sum l
```

Attempting a Proof


```
for all l:int list, sum_cont l (fun s -> s) == sum l

Proof: By induction on the structure of the list l.

case l = []
  ...

case: hd::tail
  IH: sum_cont tail (fun s -> s) == sum tail
```

Attempting a Proof


```
for all l:int list, sum_cont l (fun s -> s) == sum l

Proof: By induction on the structure of the list l.

case l = []
  ...

case: hd::tail
  IH: sum_cont tail (fun s -> s) == sum tail

   sum_cont (hd::tail) (fun s -> s)
==
```

Attempting a Proof


```
for all l:int list, sum_cont l (fun s -> s) == sum l

Proof: By induction on the structure of the list l.

case l = []
  ...

case: hd::tail
  IH: sum_cont tail (fun s -> s) == sum tail

   sum_cont (hd::tail) (fun s -> s)
== sum_cont tail (fn s' -> (fn s -> s) (hd + s'))   (eval)
```

Attempting a Proof


```
for all l:int list, sum_cont l (fun s -> s) == sum l

Proof: By induction on the structure of the list l.

case l = []
  ...

case: hd::tail
  IH: sum_cont tail (fun s -> s) == sum tail

   sum_cont (hd::tail) (fun s -> s)
== sum_cont tail (fn s' -> (fn s -> s) (hd + s'))   (eval)
== sum_cont tail (fn s' -> hd + s')                 (eval)
```

Need to Generalize the Theorem and IH


```
for all l:int list, sum_cont l (fun s -> s) == sum l

Proof: By induction on the structure of the list l.

case l = []
  ...

case: hd::tail
  IH: sum_cont tail (fun s -> s) == sum tail

   sum_cont (hd::tail) (fun s -> s)
== sum_cont tail (fn s' -> (fn s -> s) (hd + s'))   (eval)
== sum_cont tail (fn s' -> hd + s')                 (eval)

== darn!
```

we'd like to use the IH, but we can't!
we might like:

sum_cont tail (fn s' -> hd + s') == sum tail

... but that's not even true

not the identity continuation (fun s -> s) like the IH requires

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)
```

Need to Generalize the Theorem and IH

```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

  must prove:  for all k:int->int, sum_cont [] k == k (sum [])
```

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

  must prove:  for all k:int->int, sum_cont [] k == k (sum [])

  pick an arbitrary k:
```

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

  must prove:  for all k:int->int, sum_cont [] k == k (sum [])

  pick an arbitrary k:

     sum_cont [] k
```

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

  must prove:  for all k:int->int, sum_cont [] k == k (sum [])

  pick an arbitrary k:

     sum_cont [] k
  == match [] with [] -> k 0 | hd::tail -> ...      (eval)
  == k 0                                            (eval)
```

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

  must prove:  for all k:int->int, sum_cont [] k == k (sum [])

  pick an arbitrary k:

     sum_cont [] k
  == match [] with [] -> k 0 | hd::tail -> ...      (eval)
  == k 0                                            (eval)



  == k (sum [])
```

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

  must prove:  for all k:int->int, sum_cont [] k == k (sum [])

  pick an arbitrary k:

     sum_cont [] k
  == match [] with [] -> k 0 | hd::tail -> ...      (eval)
  == k 0                                            (eval)

  == k (0)                                          (eval, reverse)
  == k (match [] with [] -> 0 | hd::tail -> ...)    (eval, reverse)
  == k (sum [])

case done!
```

Need to Generalize the Theorem and IH

```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

  IH:    for all k':int->int, sum_cont tail k' == k' (sum tail)

  Must prove:  for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))
```

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

  IH:   for all k':int->int, sum_cont tail k' == k' (sum tail)

  Must prove:  for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))

  Pick an arbitrary k,

     sum_cont (hd::tail) k
```

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

  IH:   for all k':int->int, sum_cont tail k' == k' (sum tail)

  Must prove:  for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))

  Pick an arbitrary k,

     sum_cont (hd::tail) k
  == sum_cont tail (fun s -> k (hd + s))      (eval)
```

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

  IH:   for all k':int->int, sum_cont tail k' == k' (sum tail)

  Must prove:  for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))

  Pick an arbitrary k,

     sum_cont (hd::tail) k
  == sum_cont tail (fun s -> k (hd + s))      (eval)

  == (fun s -> k (hd + s)) (sum tail)         (IH with IH quantifier k'
                                               replaced with (fun s -> k (hd+s))
```

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

  IH:   for all k':int->int, sum_cont tail k' == k' (sum tail)

  Must prove:  for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))

  Pick an arbitrary k,

     sum_cont (hd::tail) k
  == sum_cont tail (fun s -> k (hd + s))      (eval)

  == (fun s -> k (hd + s)) (sum tail)         (IH with IH quantifier k'
                                               replaced with (fun s -> k (hd+s))
  == k (hd + (sum tail))                      (eval, since sum total and
                                                     and sum tail valuable)
```

Need to Generalize the Theorem and IH


```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

  IH:   for all k':int->int, sum_cont tail k' == k' (sum tail)

  Must prove:  for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))

  Pick an arbitrary k,

     sum_cont (hd::tail) k
  == sum_cont tail (fun s -> k (hd + s))      (eval)

  == (fun s -> k (hd + s)) (sum tail)         (IH with IH quantifier k'
                                               replaced with (fun s -> k (hd+s))
  == k (hd + (sum tail))                      (eval, since sum total and
                                                     and sum tail valuable)
  == k (sum (hd::tail))                        (eval sum, reverse)

case done!
QED!
```

Finishing Up

Ok, now what we have is a proof of this theorem:

```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)
```

We can use that general theorem to get what we really want:

```
for all l:int list,
   sum2 l
== sum_cont l (fun s -> s)      (by eval sum2)
== (fun s -> s) (sum l)        (by theorem, instantiating k with (fun s -> s)
== sum l                       (by eval, since sum l valuable)
```

So, we've show that the function sum2, which is tail-recursive, is functionally equivalent to the non-tail-recursive function sum.

SUMMARY

CPS

CPS is interesting and important:

- *unavoidable*
  - assembly language is continuation-passing
- *theoretical ramifications*
  - fixes evaluation order
  - call-by-value evaluation == call-by-name evaluation
- *efficiency*
  - generic way to create tail-recursive functions
  - Appel's SML/NJ compiler based on this style
- *continuation-based programming*
  - call-backs
  - programming with "*what to do next*"
- *implementation-technique for concurrency*

Summary of the CPS Proof

We tried to prove the *specific* theorem we wanted:

```
for all l:int list, sum_cont l (fun s -> s) == sum l
```

But it didn't work because in the middle of the proof, *the IH didn't apply* -- inside our function we had the wrong kind of continuation -- not (fun s -> s) like our IH required. So we had to *prove a more general theorem* about *all* continuations.

```
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)
```

This is a common occurrence -- *generalizing the induction hypothesis* -- and it requires human ingenuity. It's why proving theorems is hard. It's also why writing programs is hard -- you have to make the proofs and programs work more generally, around every iteration of a loop.

Overall Summary

We developed techniques for reasoning about the space costs of functional programs

- the cost of *manipulating data types* like tuples and trees
- the cost of allocating and using *function closures*
- the cost of *tail-recursive* and non-tail-recursive *functions*

We also talked about some important program transformations:

- *closure conversion* makes nested functions with free variables into pairs of closed code and environment
- the *continuation-passing style* (CPS) transformation turns non-tail-recursive functions in to tail-recursive ones that use no stack space
  - the stack gets moved in to the function closure
- since stack space is often small compared with heap space, it is often necessary to use *continuations and tail recursion*
  - but full CPS-converted programs are unreadable: use judgement