Continuation-Passing Style

COS 326
David Walker
Princeton University

TAIL CALLS AND CONTINUATIONS

Tail Recursion

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

work to do after the function call

```
let sum_tail (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
;;
```

no work to do after the function call

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?

```
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
```

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

Question

Key idea: 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
```

Focus on what happens after the recursive call.

Question

Key idea: 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

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?

```
hd + 
```

result of recursive call gets plugged in here

```
fun s -> 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
```

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]
```

result of non-tail call

final result

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

let rec sum_cont (l:int list) (k:cont): int =
  match l with
    [] -> ...
  | hd::tail -> ...
```

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]
```

result of non-tail call

final result

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

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

result of non-tail call

final result

```
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 -> 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
```

result of non-tail call

final result

```
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 = sum_cont l ( ... )
```

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 x -> x)
```

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?

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 s1 -> (fun s2 -> (fun s3 -> s3) (1+s2)) (2+s1))
-->
    ...
--> 3
```

free variables stored in closure

Where did the stack space go?

there are 3 closures here.
each contains an environment

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 s1 -> (fun s2 -> (fun s3 -> s3) (1+s2)) (2+s3))
-->
    ...
-->
    3
```

free variables stored in closure

is/was variable hd stored in closure

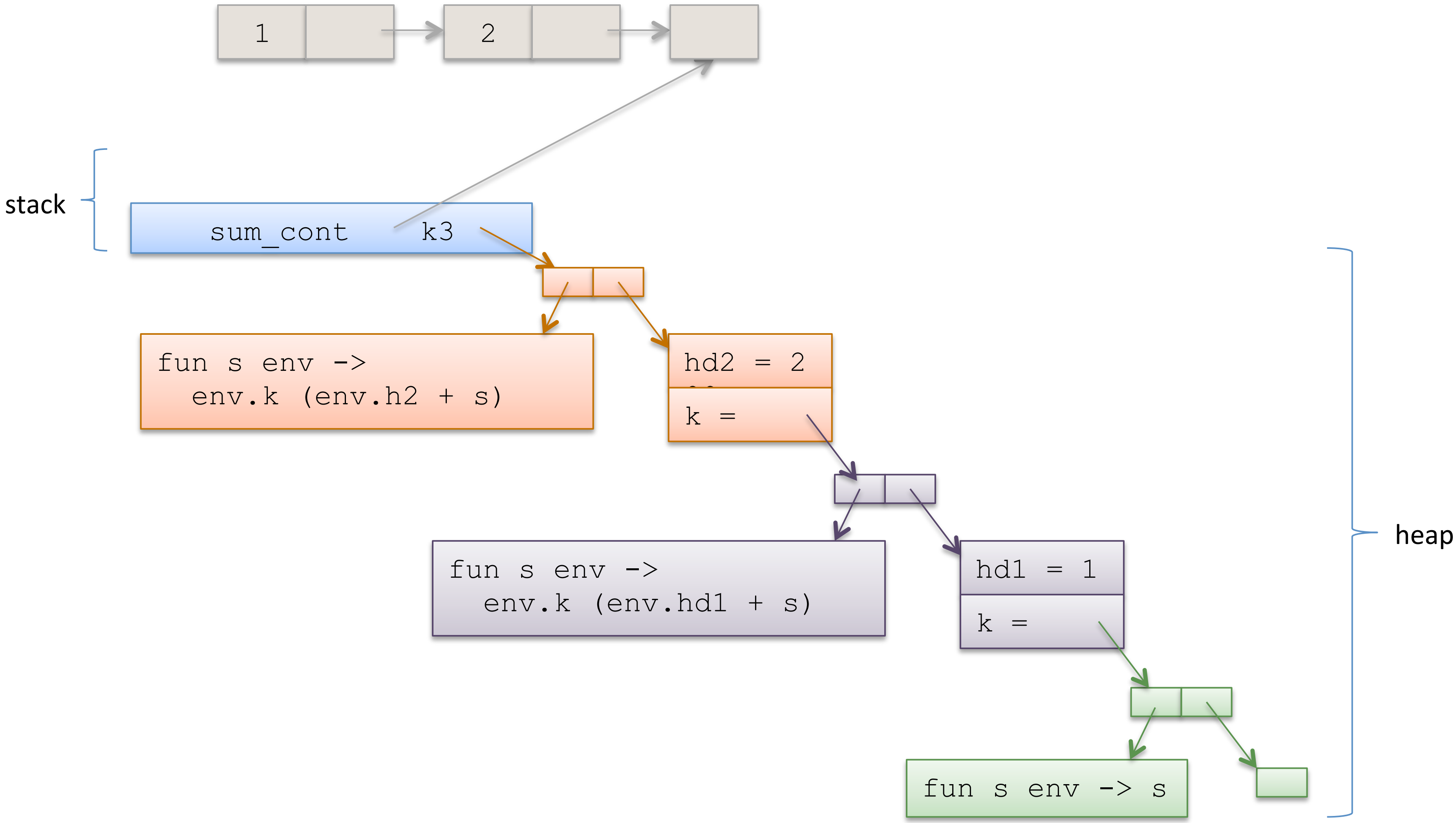
every time you call a function
you allocate a continuation to do what's next!
the stack shows up on the heap!!!

there are 3 closures here.
each contains an environment

function inside
function inside
function inside
expression

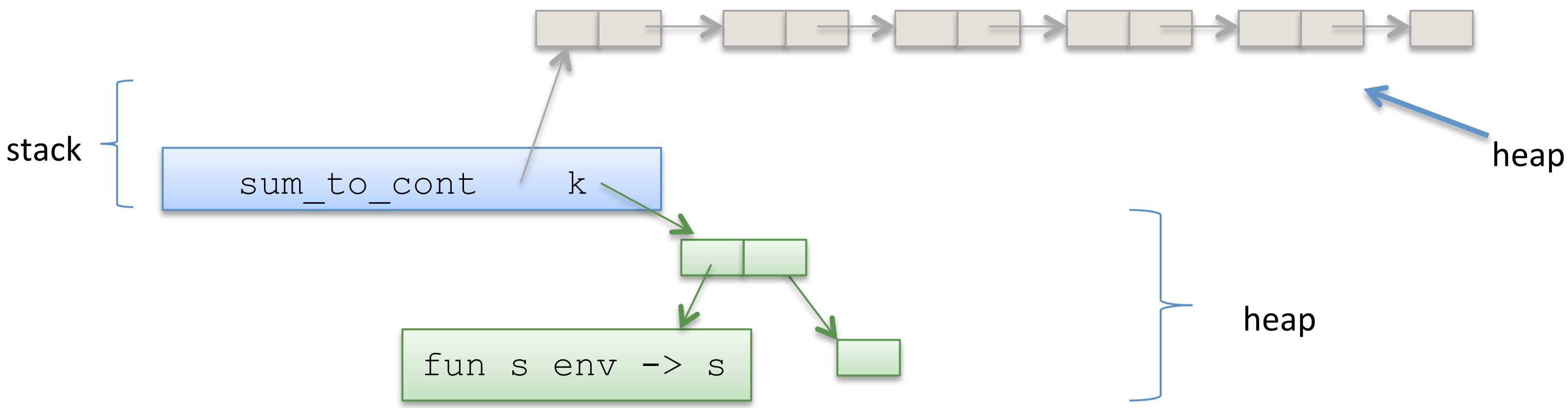
a stack of
closures on
the heap

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

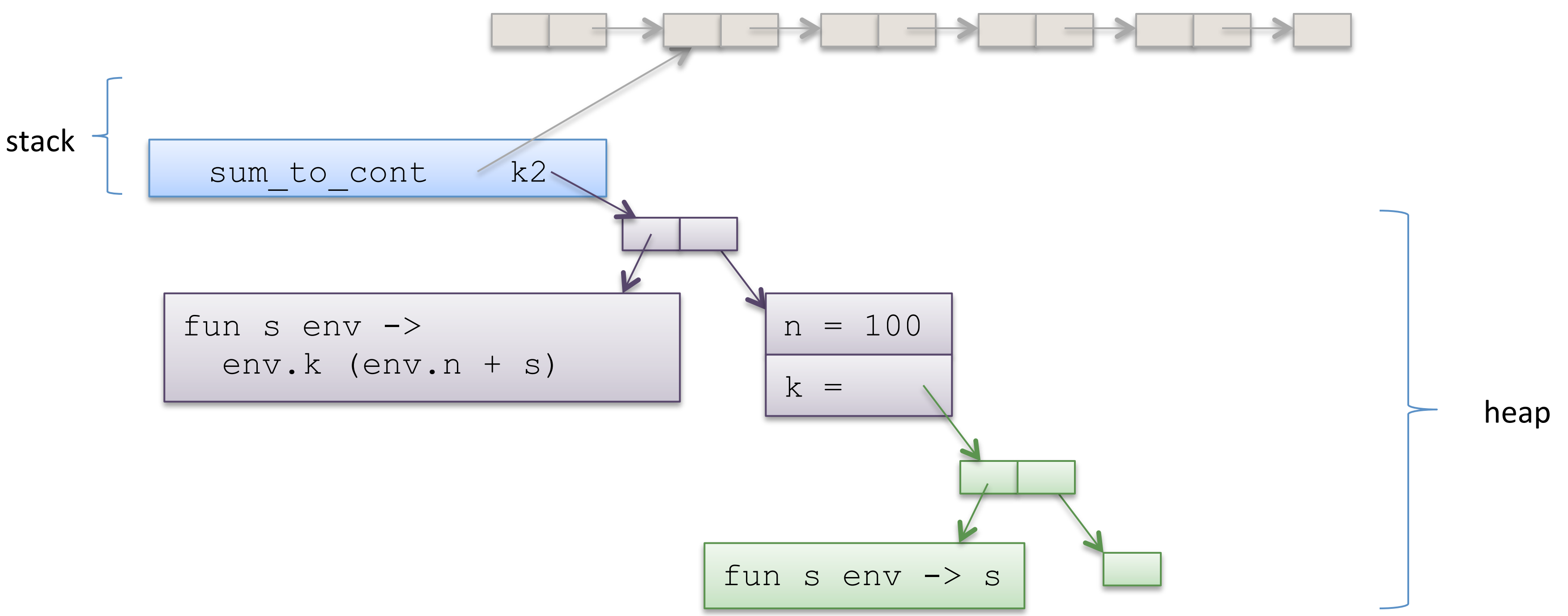


Continuation-passing style

```
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)) ;;
```

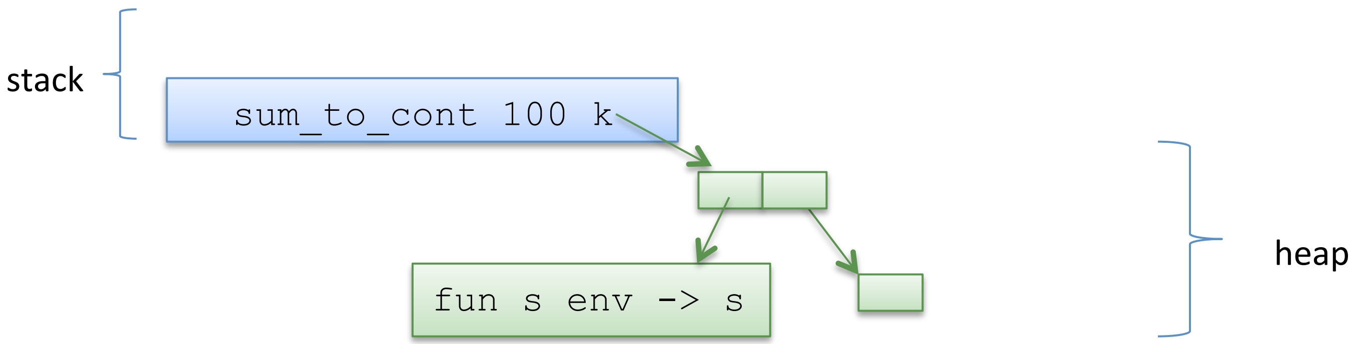


Continuation-passing style



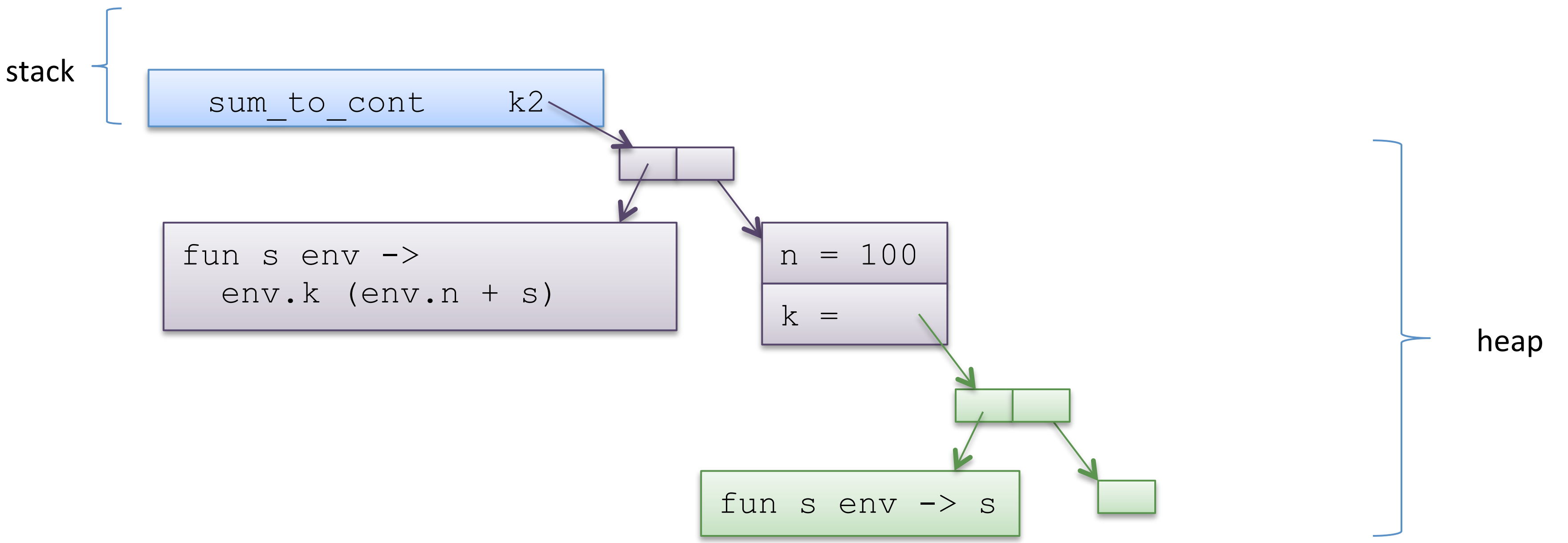
Continuation-passing style

```
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)
```



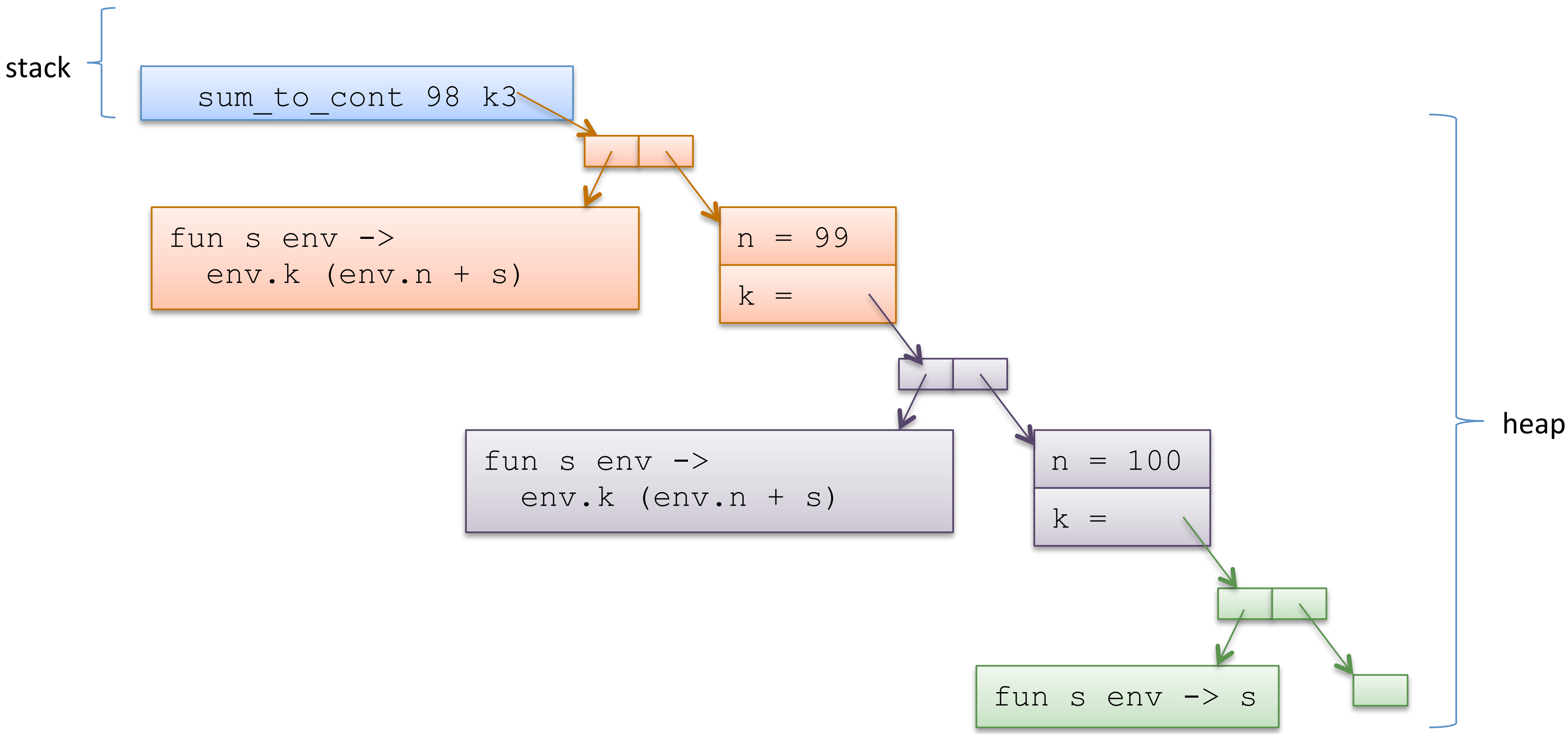
Continuation-passing style

```
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)
```



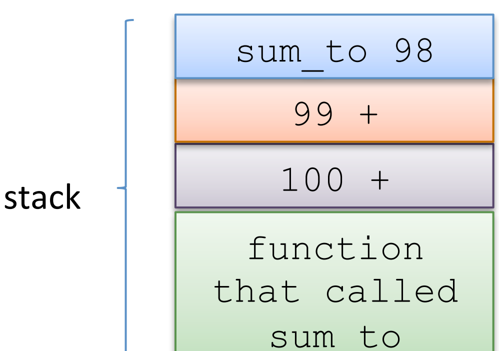
Continuation-passing style

```
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

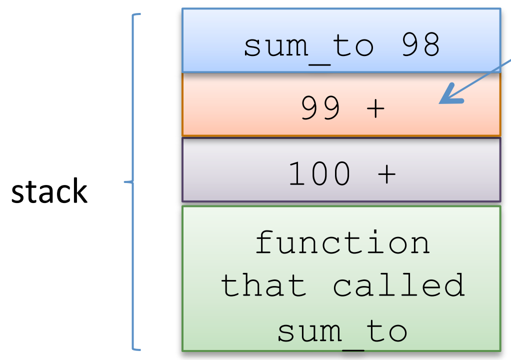
```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



Back to stacks

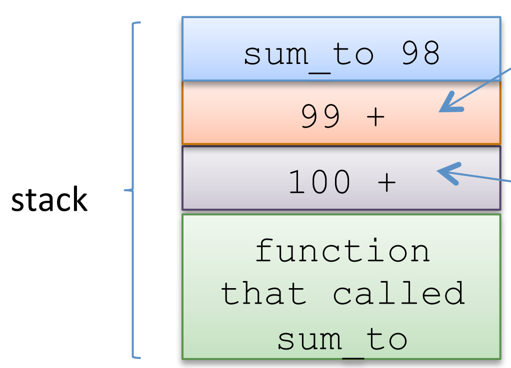
```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

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
```



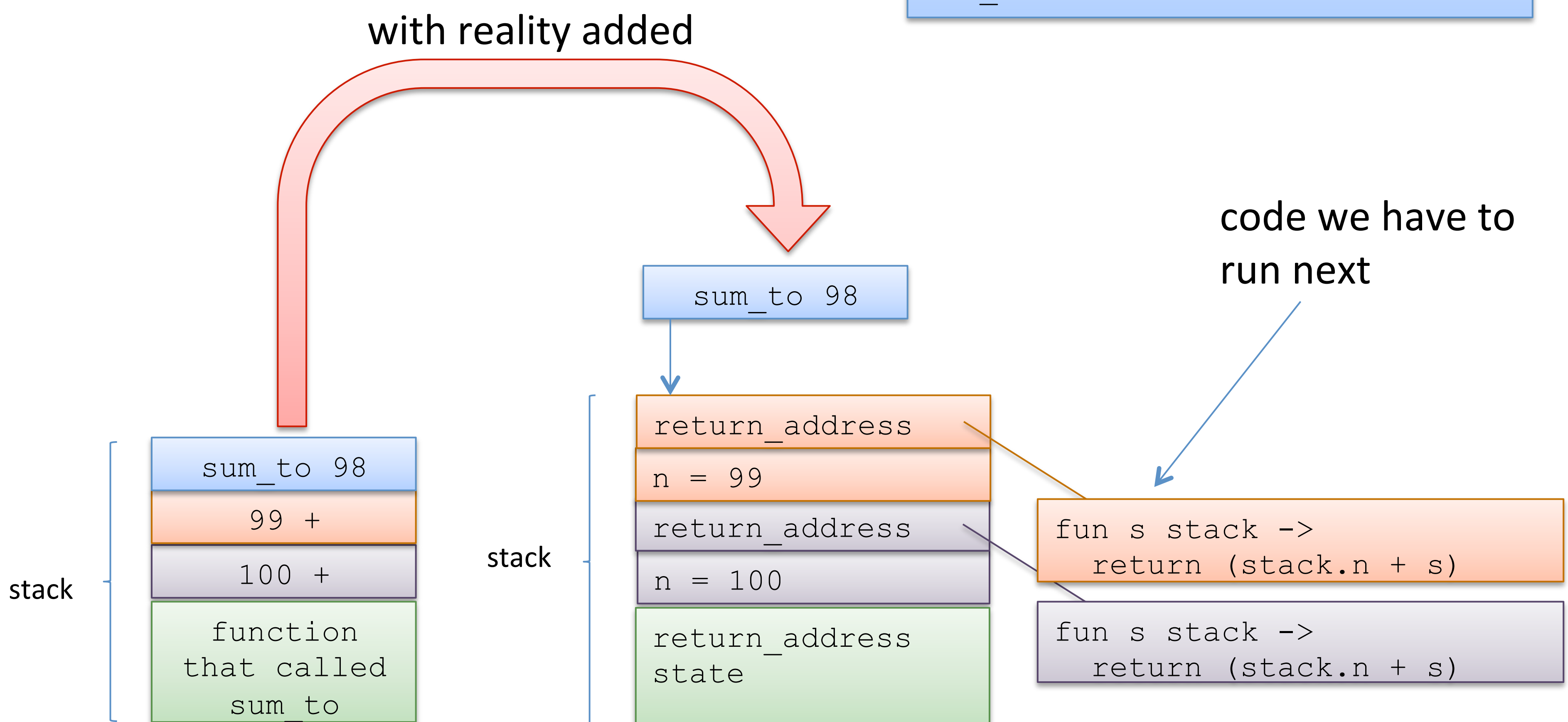
but how do you really implement that?

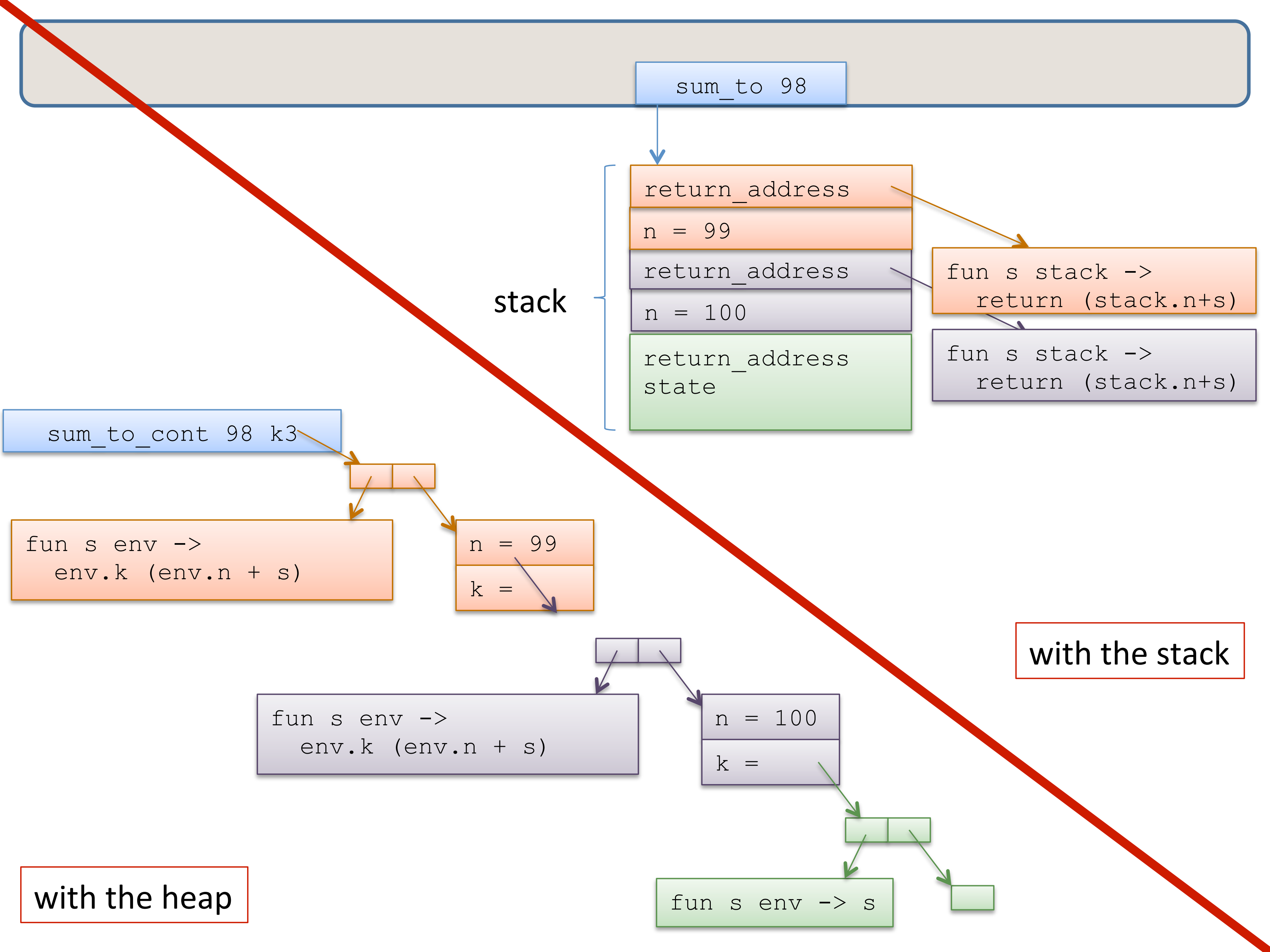
there is two bits of information here:

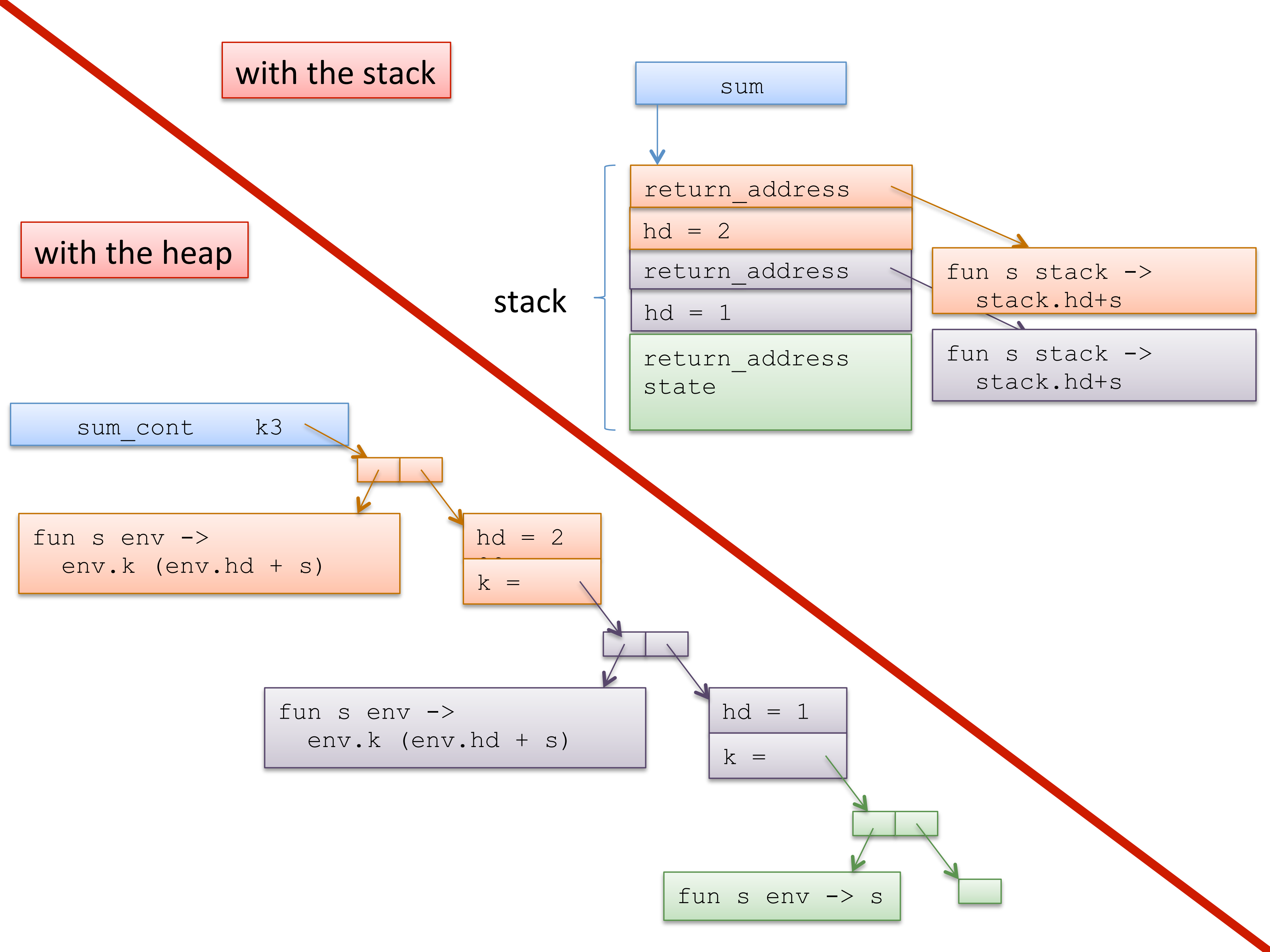
(1) some state (n=100) we had to remember
(2) some code we have to run later

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



with the stack
with the heap
sum
stack
return_address
hd = 2
return_address
hd = 1
return_address
state
fun s stack ->
stack.hd+s
fun s stack ->
stack.hd+s
sum_cont k3
fun s env ->
env.k (env.hd + s)
hd = 2
k =
fun s env ->
env.k (env.hd + s)
hd = 1
k =
fun s env -> s

Why CPS?

Continuation-passing style is *inevitable*.

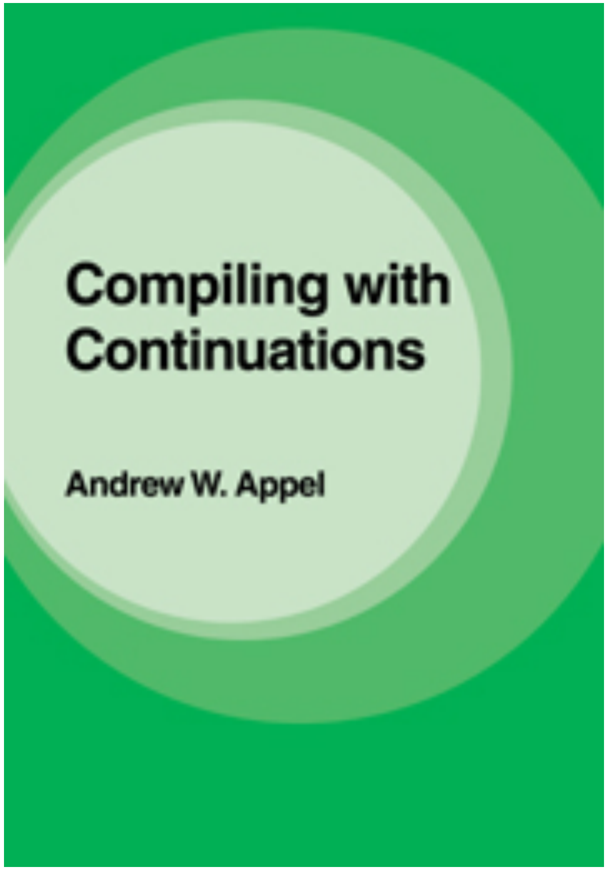
It does not matter whether you program in Java or C or OCaml -- there's code around that tells you "*what to do next*"

- 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:

- linked list of closures can be less space-efficient than stack
- hardware architectures optimized to use a stack
- 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.stuff.com/i.html
      (fun html -> process html)
```

continuation

Summary

CPS is interesting and important:

- *unavoidable*
    - assembly language is continuation-passing
- *theoretical ramifications*
    - fixes evaluation order
    - call-by-value evaluation == call-by-name evaluation
    - work by Gordon Plotkin
- *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*

ANOTHER EXAMPLE

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: It is a little easier to put the continuations in the order in which they are called.

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) ->
      let t1 = incr left i in
      let t2 = incr right i in
      Node (i+j, t1, t2)
;;
```

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

let rec incr (t:tree) (i:int) (k: tree -> tree) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) ->
      let t1 = incr left i in
      let t2 = incr right i in
      Node (i+j, t1, t2)
```

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) ->
      let t1 = incr left i in
      let t2 = incr right i in
      Node (i+j, t1, t2)
;;
```

.

Hint: There are 2 function calls and so you need 2 continuations. Go.

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) ->
      let t1 = incr left i in
      let t2 = incr right i in
      Node (i+j, t1, t2)
;;
```

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

let rec incr (t:tree) (i:int) (k: tree -> tree) : tree =
  match t with
    Leaf -> Leaf
  | Node (j,left,right) ->
        let t1 = incr left i in
        let t2 = incr right i in
        Node (i+j, t1, t2))
```

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) ->
      let t1 = incr left i in
      let t2 = incr right i in
      Node (i+j, t1, t2)
;;
```

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

let rec incr (t:tree) (i:int) (k: tree -> tree) : tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) ->
        let t1 = incr left i in
        let t2 = incr right i in
        Node (i+j, t1, t2))
```

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) ->
      let t1 = incr left i in
      let t2 = incr right i in
      Node (i+j, t1, t2)
;;
```

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

let rec incr (t:tree) (i:int) (k: tree -> tree) : tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) ->
      incr left i (fun result1 ->
        let t1 = result1 in
        let t2 = incr right i in
        Node (i+j, t1, t2))
```

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) ->
      let t1 = incr left i in
      let t2 = incr right i in
      Node (i+j, t1, t2)
;;
```

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

let rec incr (t:tree) (i:int) (k: tree -> tree) : tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) ->
      incr left i (fun result1 ->
        let t1 = result1 in
        incr right i (fun result2 -> let t2 = result2 in
                                     Node (i+j, t1, t2))
```

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) ->
      let t1 = incr left i in
      let t2 = incr right i in
      Node (i+j, t1, t2)
;;
```

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

let rec incr (t:tree) (i:int) (k: tree -> tree) : tree =
  match t with
    Leaf -> k Leaf
  | Node (j,left,right) ->
      incr left i (fun result1 ->
        let t1 = result1 in
        incr right i (fun result2 -> let t2 = result2 in
                                      k (Node (i+j, t1, t2)))
```

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
```

```
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))
```

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)
==
```

```
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))
```

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 (fun s' -> (fun s -> s) (hd + s')) (eval)
```

```
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))
```

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

```
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))
```

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

== darn!
```

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

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

... but that's not going to work either

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

Need to Generalize the Theorem and IH

Original theorem:

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

Specific continuation

Need to Generalize the Theorem and IH

Original theorem:

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

Specific continuation

New theorem:

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

Prove it for *all* continuations. A more general theorem.
Sometimes more general theorems are easier to prove.

Need to Generalize the Theorem and IH

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

```
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))
```

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 [])
```

```
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))
```

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:
```

```
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))
```

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
```

```
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))
```

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)
```

```
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))
```

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 [])
```

```
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))
```

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

case done!
```

```
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))
```

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))
```

```
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))
```

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
```

```
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))
```

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 + x))      (eval)
```

```
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))
```

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 + x))       (eval)

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

```
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))
```

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 + x))      (eval)

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

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

Recall:

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

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
```

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

SUMMARY

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.

END