A Space Model

COS 326
David Walker
Princeton University

Midterm Exam

Instructions to download will be on Piazza

You have 24 hours once you begin

Take-home. You will download the exam to begin your 24 hours.

Earliest Start Time: Sun Oct 21 (12 noon)

Latest Start Time: Tues Oct 23, (11:59pm)

Latest End time: Wed Oct 24 (11:59pm)

(You must hand in the midterm by the end time,
like it is an assignment – no late days allowed.)

Lecture on Wednesday Oct 24 will be cancelled.

Because Halloween draws nigh:

Serial killer or programming languages researcher?

http://www.malevole.com/mv/misc/killerquiz/

Space

Understanding the space complexity of functional programs

- At least two interesting components:
  - the amount of *live space* at any instant in time
  - the *rate of allocation*
    - a function call may not change the amount of live space by much but may allocate at a substantial rate
    - because functional programs act by generating new data structures and discarding old ones, they often allocate a lot
      - OCaml garbage collector is optimized with this in mind
      - interesting fact: at the assembly level, the number of writes by a functional program is roughly the same as the number of writes by an imperative program

Space

Understanding the space complexity of functional programs

- At least two interesting components:
    - the amount of *live space* at any instant in time
    - the *rate of allocation*
        - a function call may not change the amount of live space by much but may allocate at a substantial rate
        - because functional programs act by generating new data structures and discarding old ones, they often allocate a lot
            - » OCaml garbage collector is optimized with this in mind
            - » interesting fact: at the assembly level, the number of writes by a functional program is roughly the same as the number of writes by an imperative program
- *What takes up space*?
    - conventional first-order data: tuples, lists, strings, datatypes
    - function representations (closures)
    - the call stack

CONVENTIONAL DATA

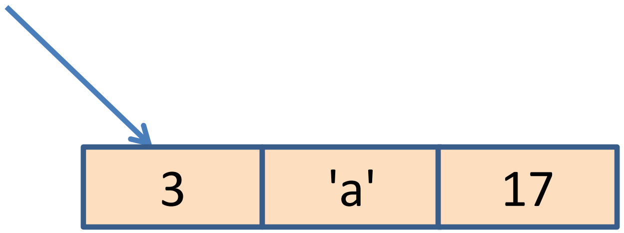
OCaml Representations for Data Structures

Type:

```
type triple = int * char * int
```

Representation:

(3, 'a', 17)



OCaml Representations for Data Structures

Type:

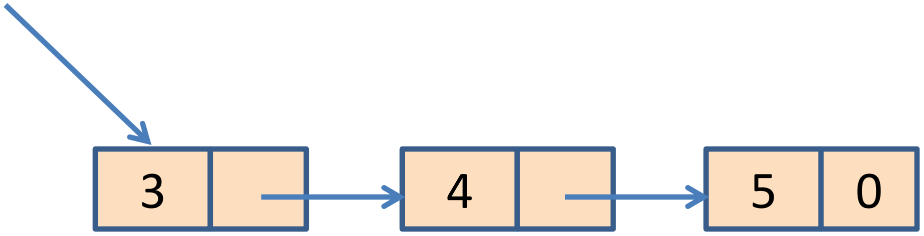
```
type mylist = int list
```

Representation:

[ ]

[3; 4; 5]

0



Space Model

Type:

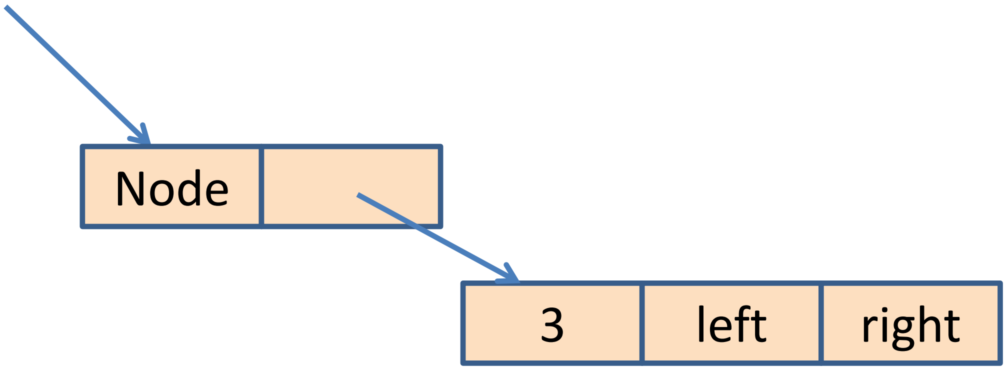
```
type tree = Leaf | Node of int * tree * tree
```

Representation:

Leaf

Node(3, left, right)

0



Node

3

left

right

Allocating space

In C, you allocate when you call “malloc”

In Java, you allocate when you call “new”

What about ML?

Allocating space

Whenever you *use a constructor*, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

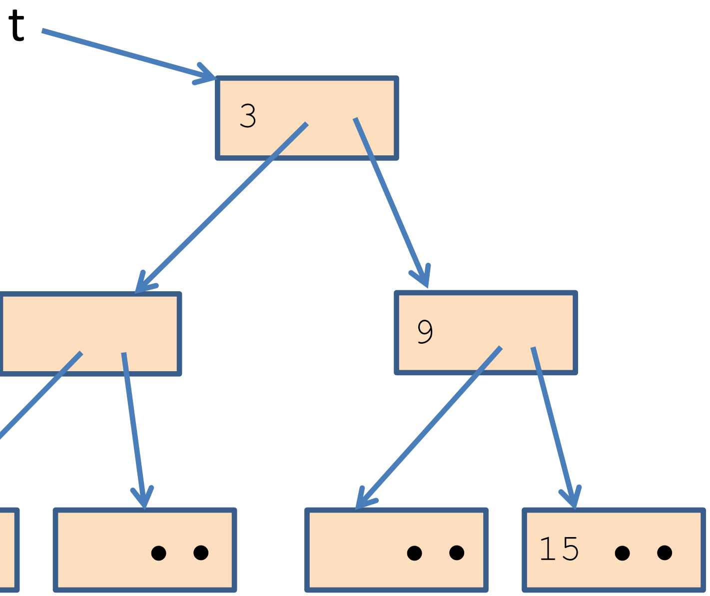
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



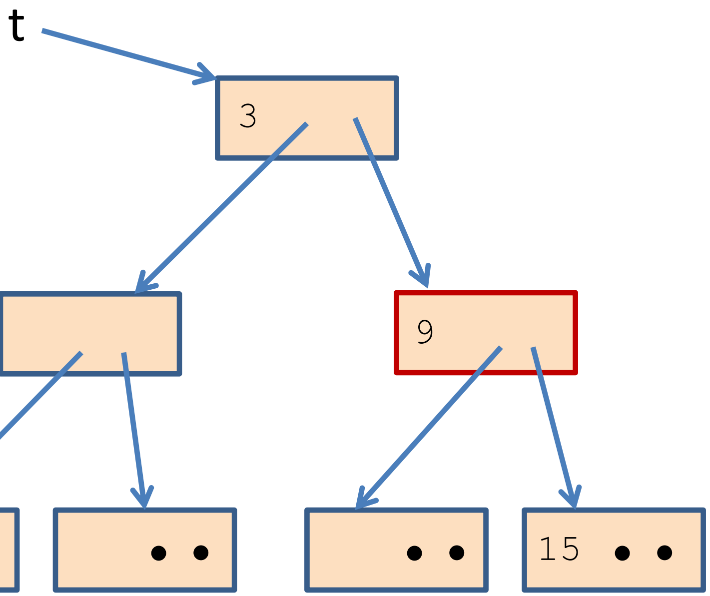
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



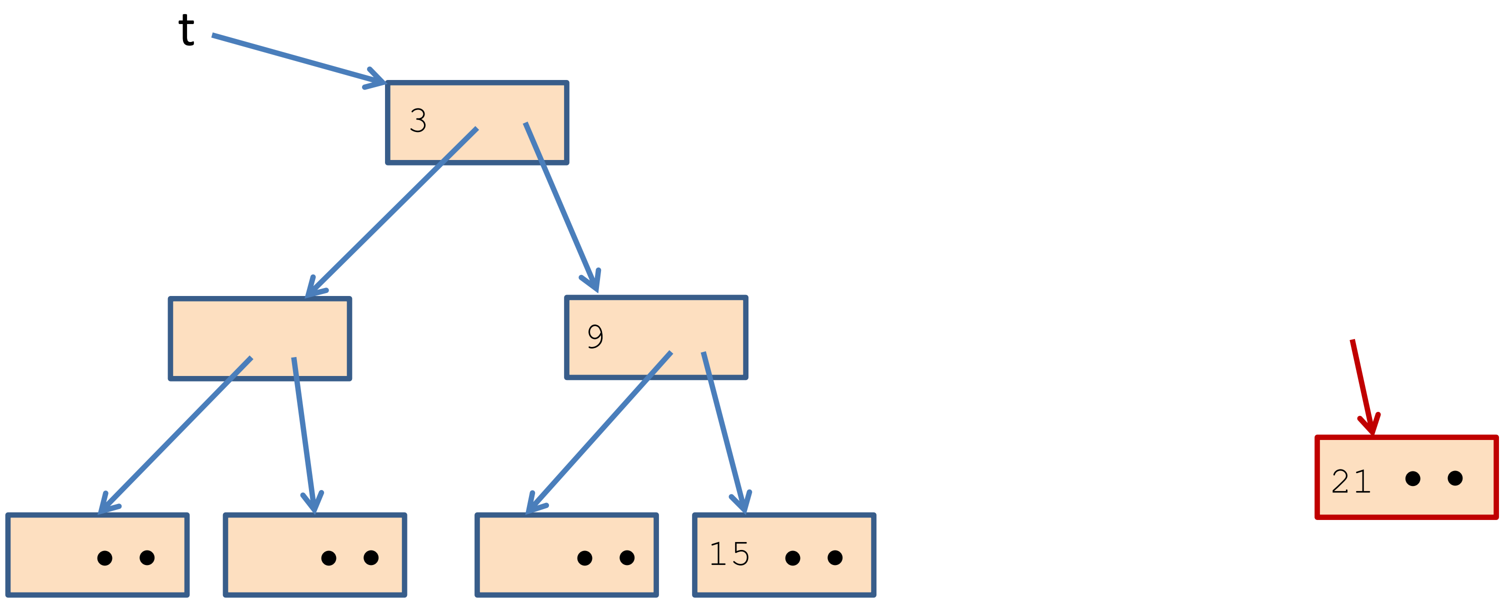
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



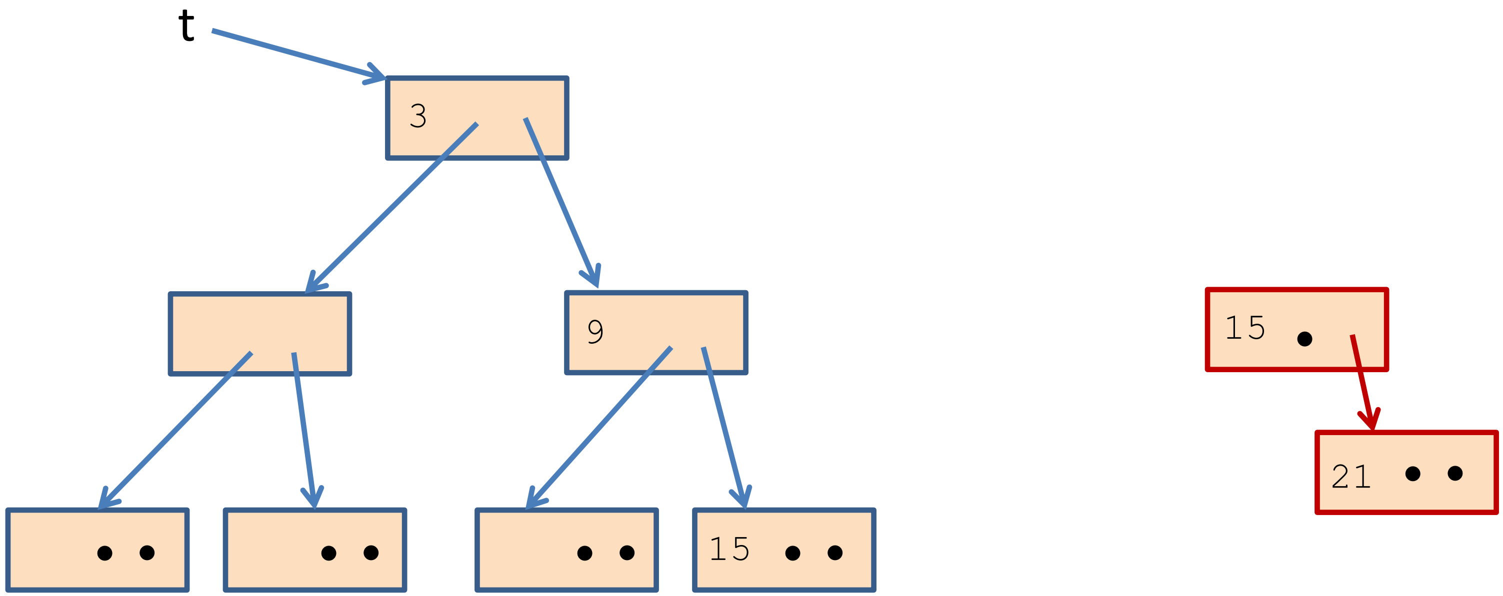
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



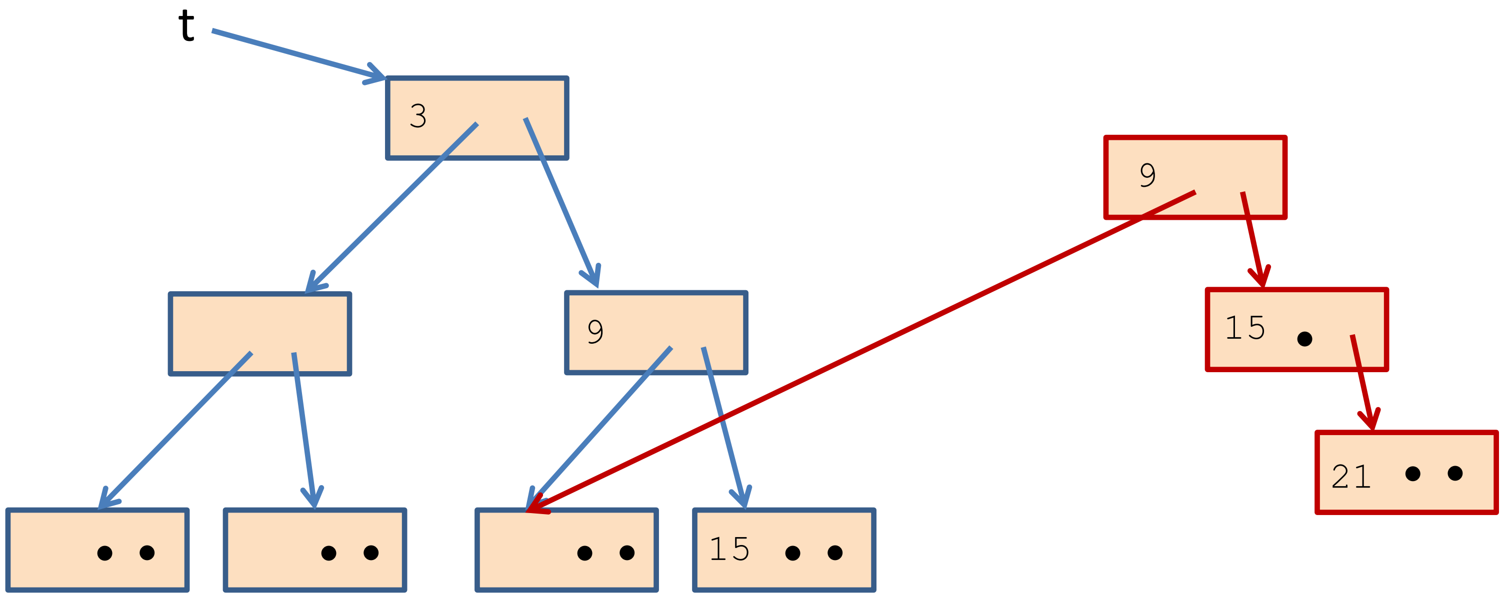
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



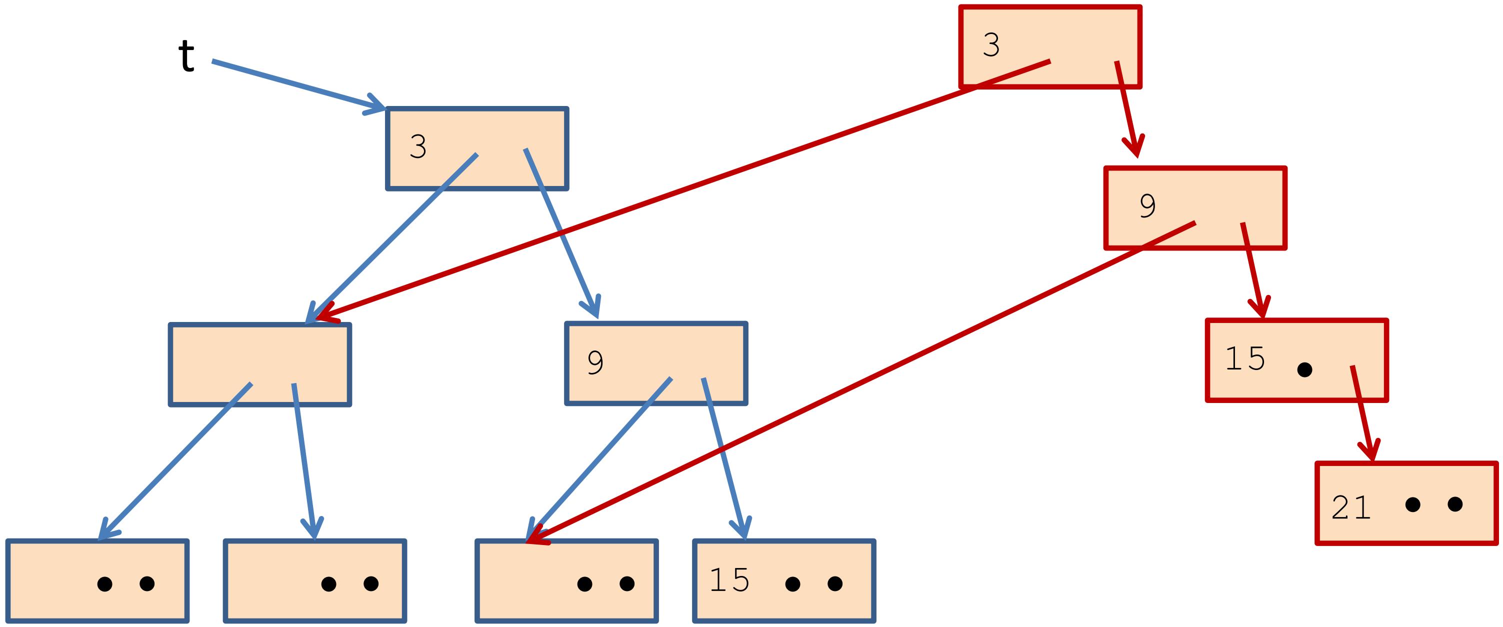
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Consider:

insert t 21



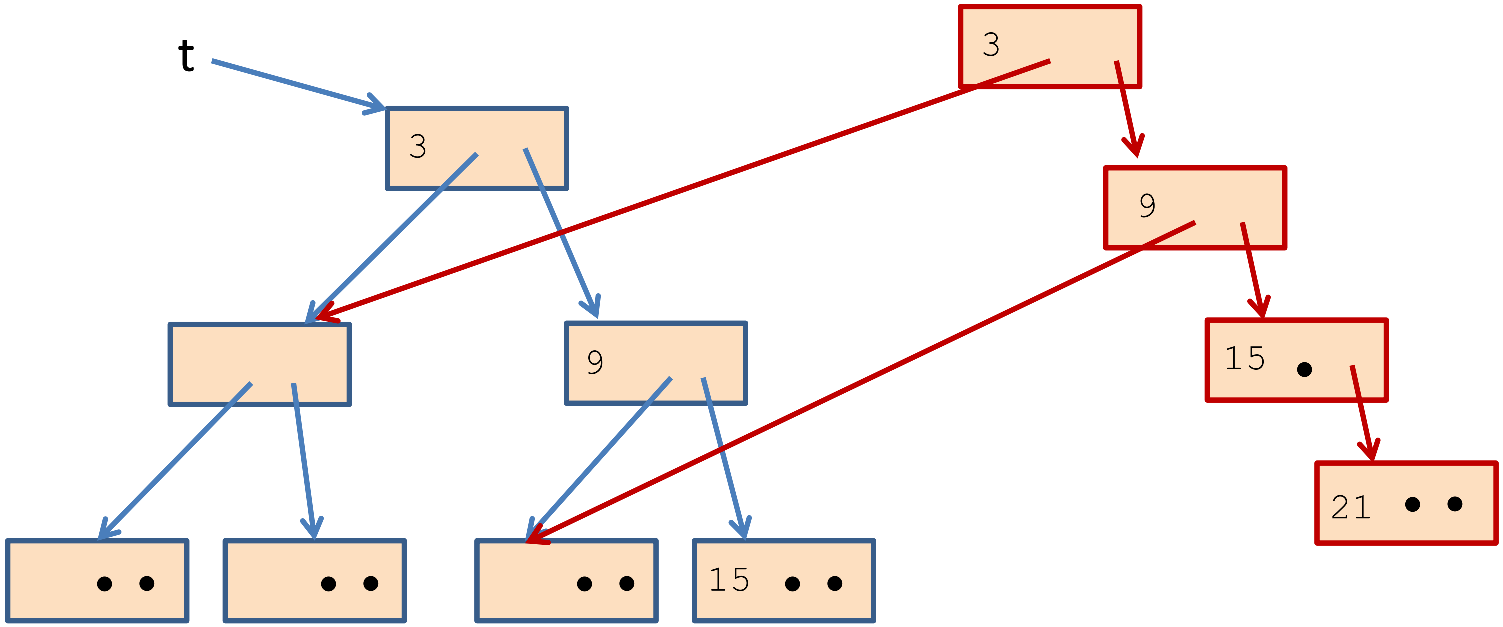
Allocating space

Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
      if i <= j then
        Node (j, insert left i, right)
      else
        Node (j, left, insert right i)
```

Total space allocated is proportional to the height of the tree.

~ log n, if tree with n nodes is balanced



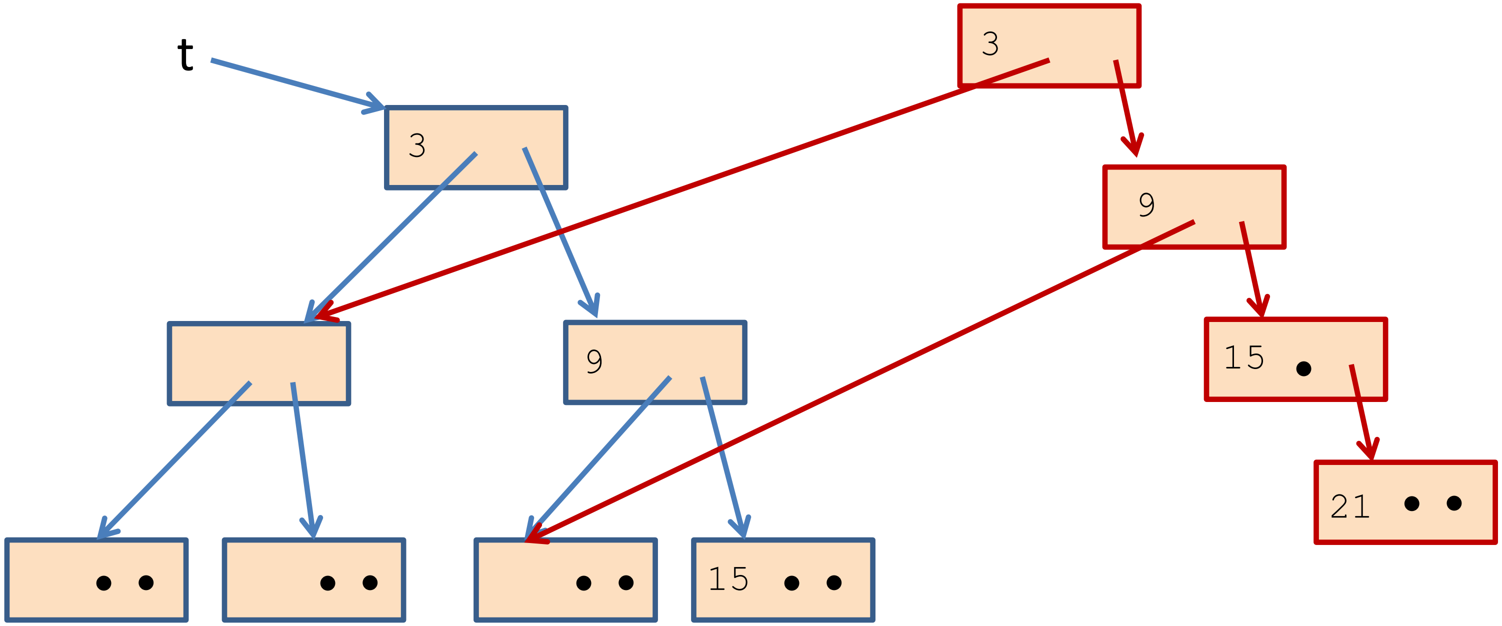
Net space allocated

The garbage collector reclaims

unreachable data structures on the heap.

```
let fiddle (t: tree) =
  insert t 21
```

John McCarthy
invented g.c.
1960



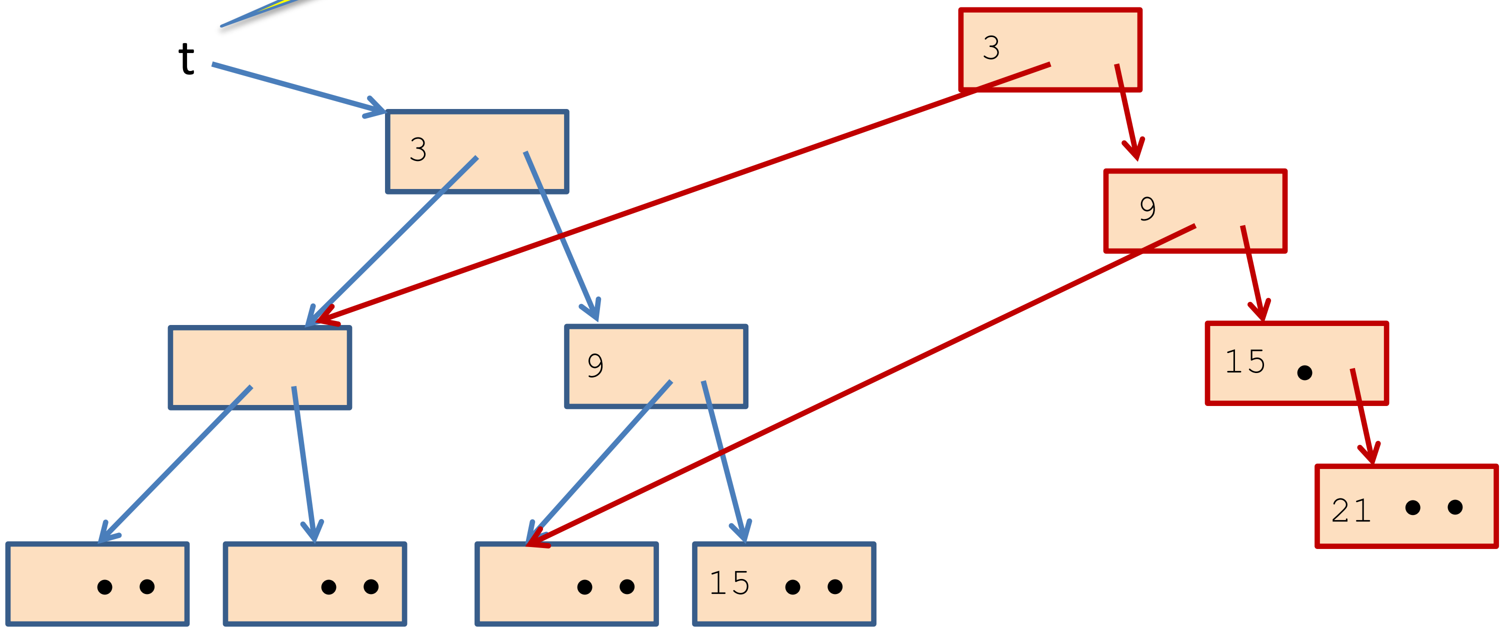
Net space allocated

The garbage collector reclaims

unreachable data structures on the heap.

```
let fiddle (t: tree) =
  insert t 21
```

If t is dead
(unreachable),

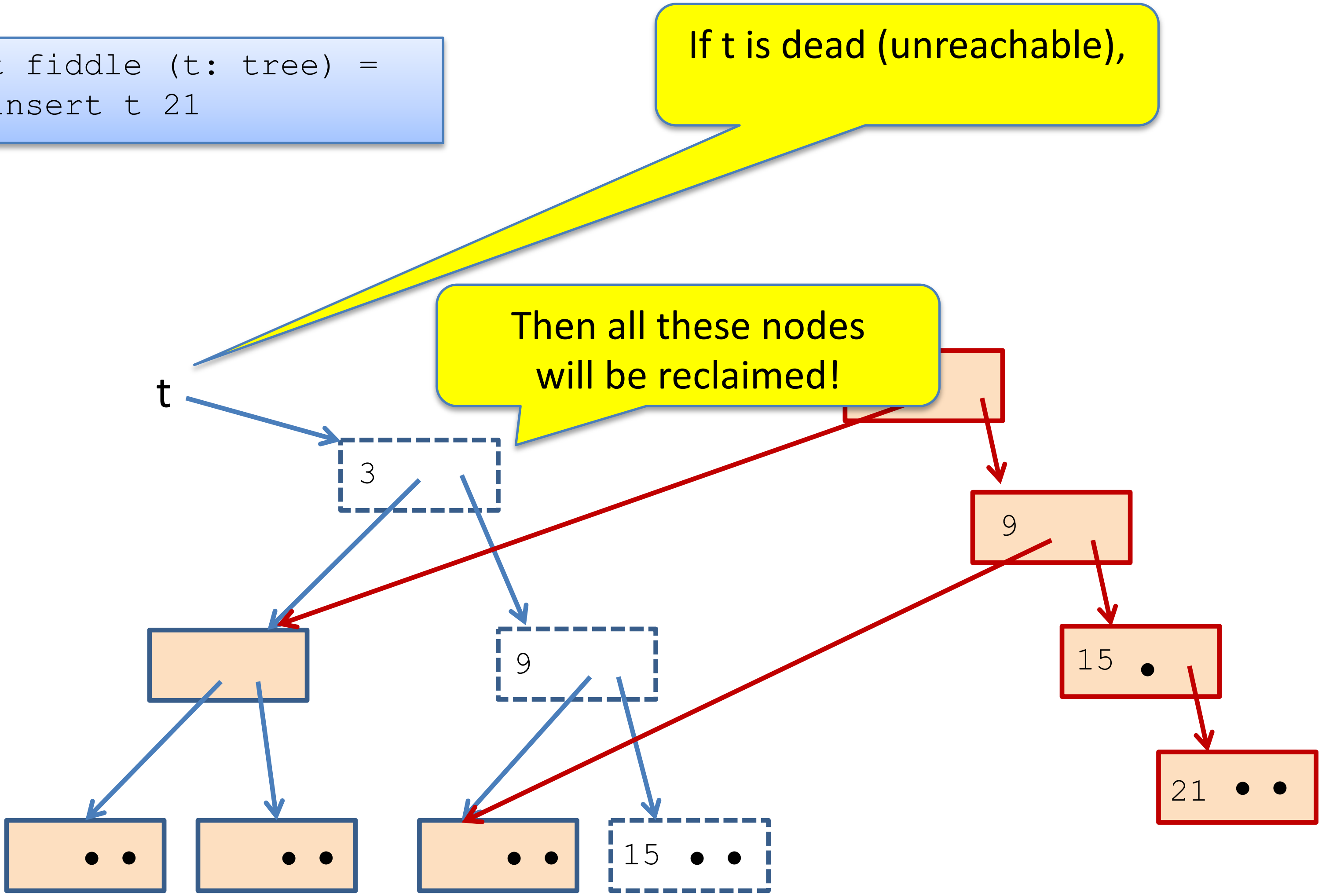


Net space allocated

The garbage collector reclaims

unreachable data structures on the heap.

```
let fiddle (t: tree) =
  insert t 21
```



If t is dead (unreachable),

Then all these nodes will be reclaimed!

Net space allocated

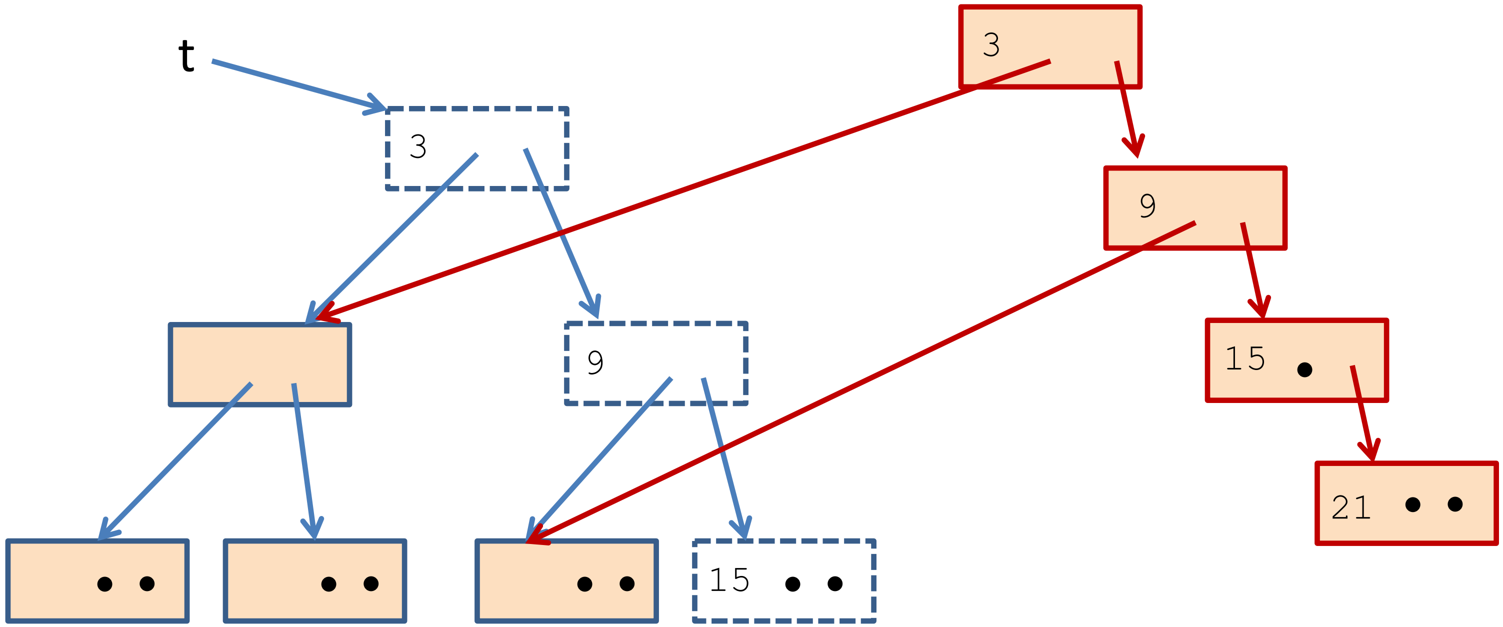
The garbage collector reclaims

unreachable data structures on the heap.

```
let fiddle (t: tree) =
  insert t 21
```

Net new space allocated:
1 node

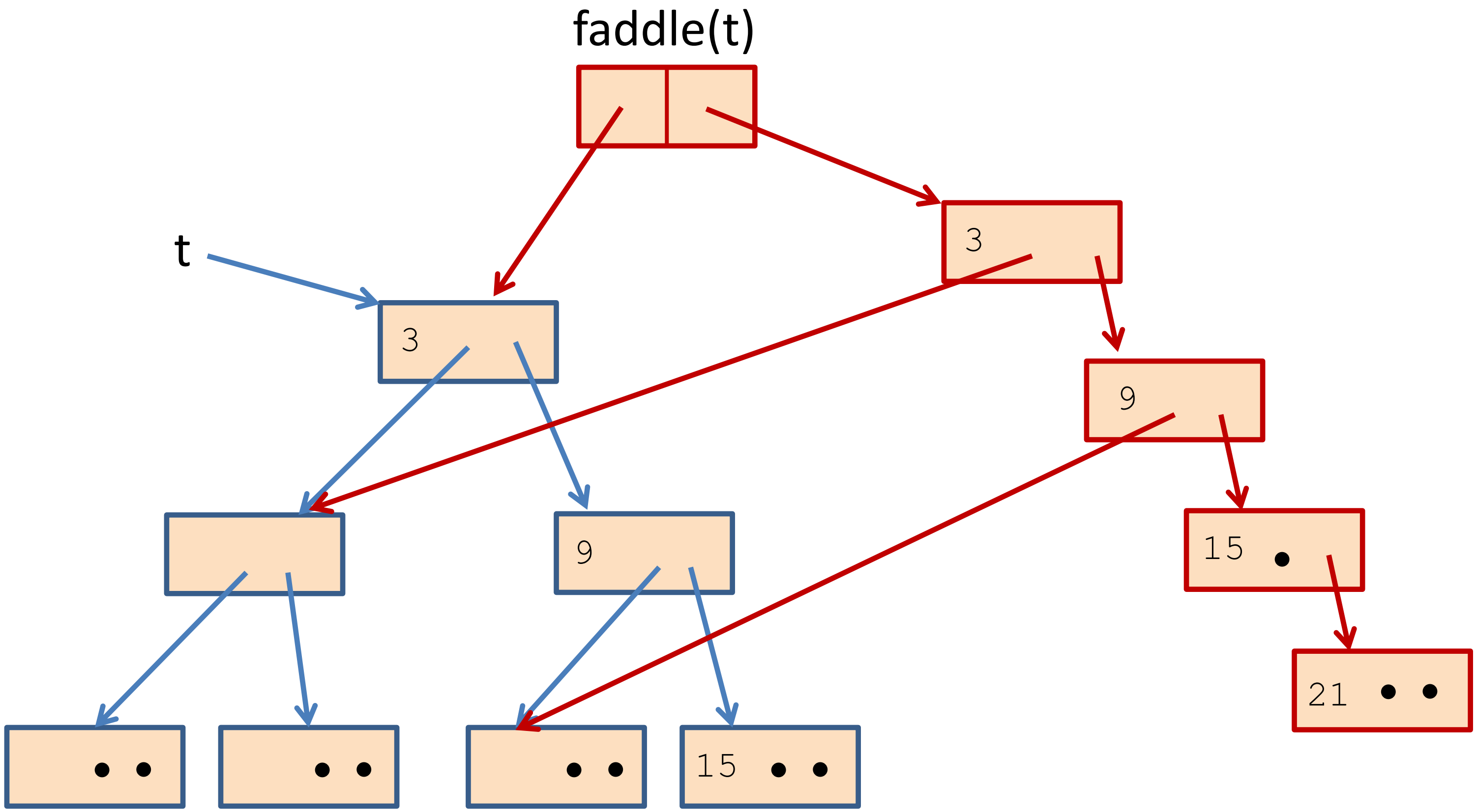
(just like "imperative" version
of binary search trees)



Net space allocated

But what if you want to keep the old tree?

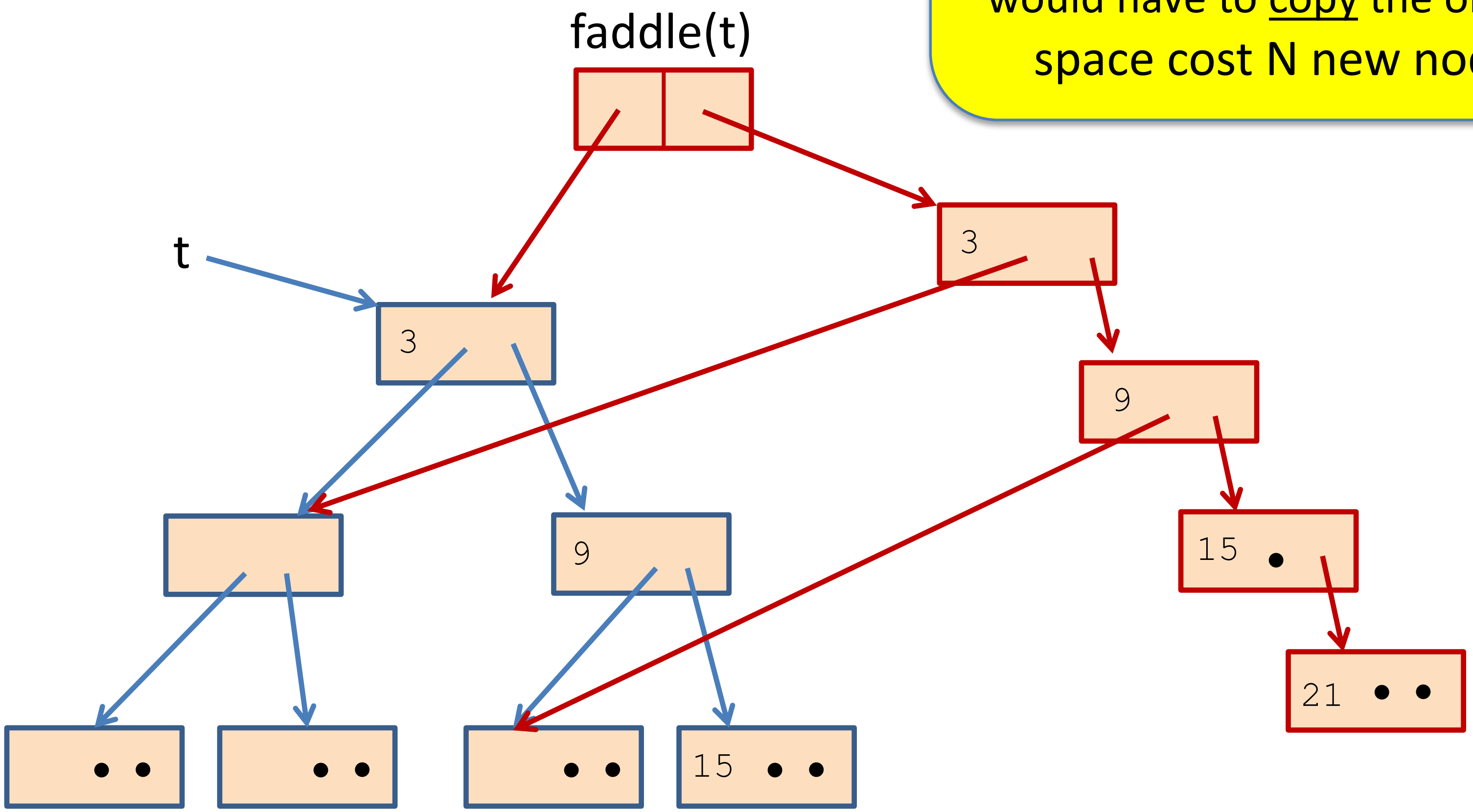
```
let faddle (t: tree) =
  (t, insert t 21)
```



Net space allocated

But what if you want to keep the old tree?

```
let faddle (t: tree) =
   (t, insert t 21)
```



Net new space allocated:
log(N) nodes

but note: “imperative” version would have to copy the old tree, space cost N new nodes!

Compare

```
let check_option (o:int option) : int option =
  match o with
    Some _ -> o
  | None -> failwith “found none”
```

```
let check_option (o:int option) : int option =
  match o with
    Some j -> Some j
  | None -> failwith “found none”
```

Compare

```
let check_option (o:int option) : int option =
  match o with
    Some _ -> o
  | None -> failwith “found none”
```

allocates nothing
when arg is Some i

```
let check_option (o:int option) : int option =
  match o with
    Some j -> Some j
  | None -> failwith “found none”
```

allocates an option
when arg is Some i

Another Example

```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

allocates
a new pair

Compare

```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
```

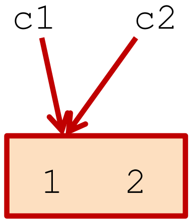
```
let double (c1:int*int) : int*int =
  cadd c1 c1
```

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```

Compare

```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
```



```
let double (c1:int*int) : int*int =
  cadd c1 c1
```

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```

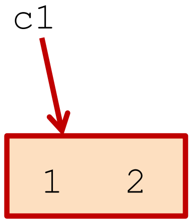
Compare

```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
```

```
let double (c1:int*int) : int*int =
  cadd c1 c1
```

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```



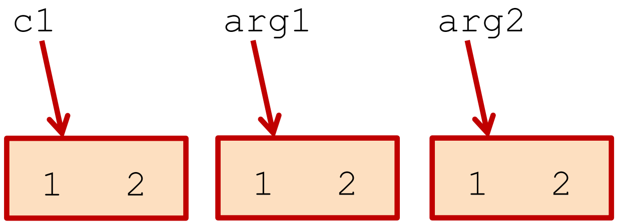
Compare

```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
```

```
let double (c1:int*int) : int*int =
  cadd c1 c1
```

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```



Compare

```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
```

no (extra) allocation

```
let double (c1:int*int) : int*int =
  cadd c1 c1
```

no (extra) allocation

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
```

allocates 2 pairs
(unless the compiler
happens to optimize…)

Compare

```
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
```

```
let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd c1 c1
```

double does not allocate

extracts components: it is a read

FUNCTION CLOSURES

Closures (A reminder)

Nested functions like bar often contain free variables:

```
let foo y =
  let bar x = x + y in
  bar
```

Here's bar on its own:

```
let bar x = x + y
```

y is *free* in the definition of bar

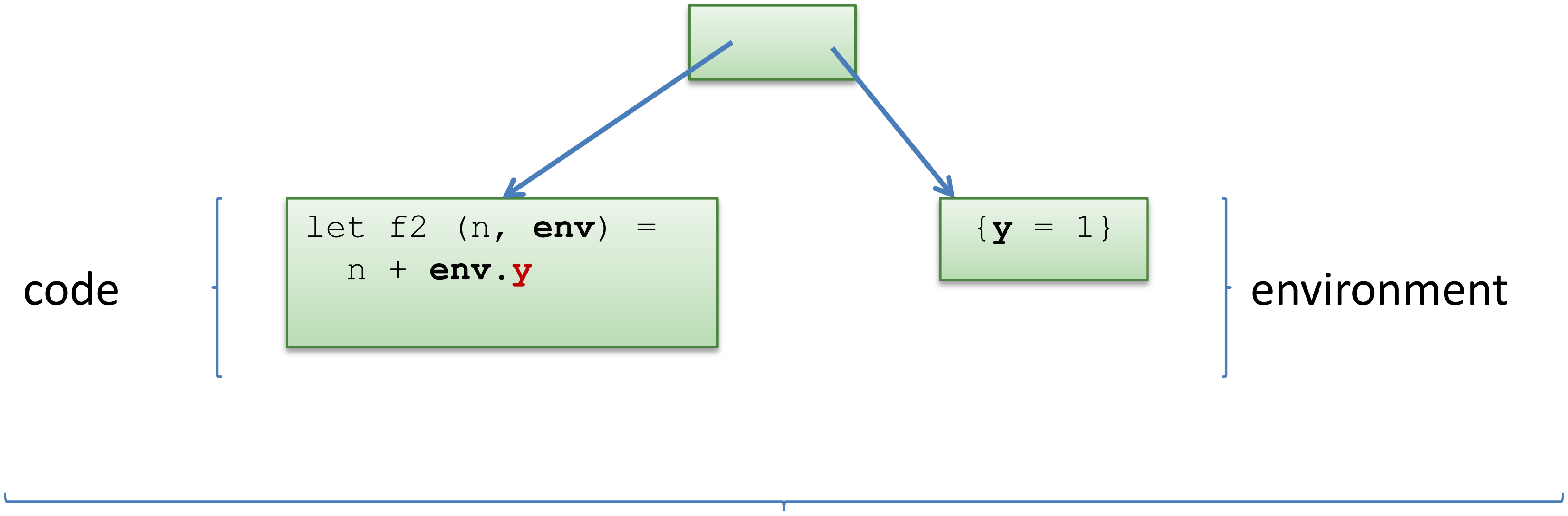
To implement bar, the compiler creates a *closure*, which is a pair of code for the function plus an environment holding the free variables.

But what about nested, higher-order functions?

bar again:

```
let bar x = x + y
```

bar's representation:



code

```
let f2 (n, env) =
  n + env.y
```

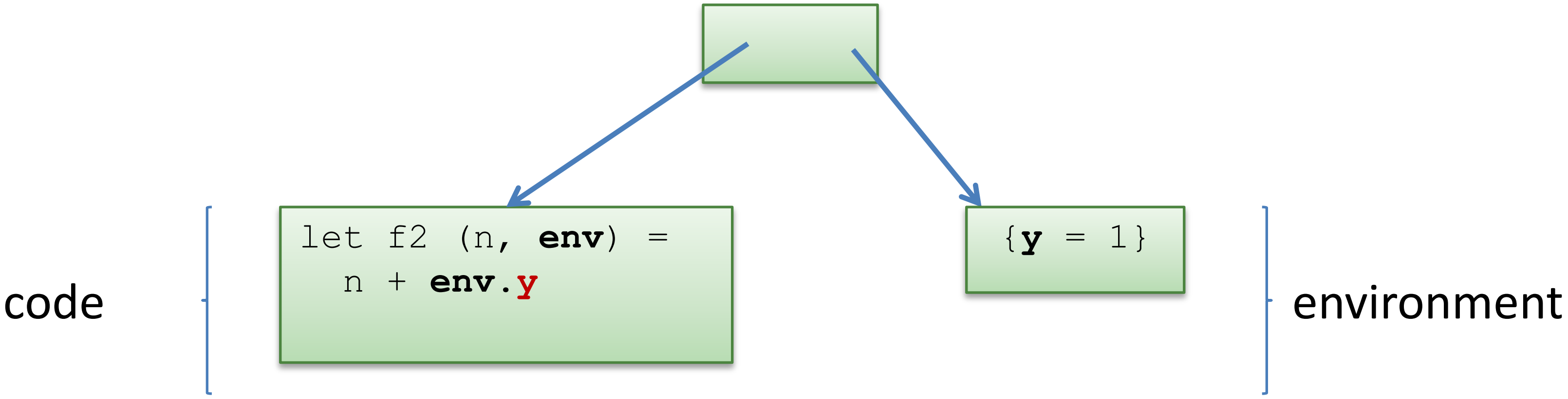
```
{y = 1}
```

environment

closure

But what about nested, higher-order functions?

To estimate the (heap) space used by a program, we often need to estimate the (heap) space used by its closures.



code

```
let f2 (n, env) =
  n + env.y
```

```
{y = 1}
```

environment

Our estimate will include the cost of the pair:

- two pointers = two 4-byte values = 8 bytes total +
- the cost of the environment (4 bytes in this case).

Space Model Summary

Understanding space consumption in FP involves:

- understanding the difference between
  - live space
  - rate of allocation
- understanding where allocation occurs
  - any time a constructor is used
  - whenever closures are created
- understanding the costs of
  - data types (fairly similar to Java)
  - costs of closures (pair + environment)

CONTINUATIONS

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

let _ = sum big_int
```

What’s going to happen when we run this code?

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:

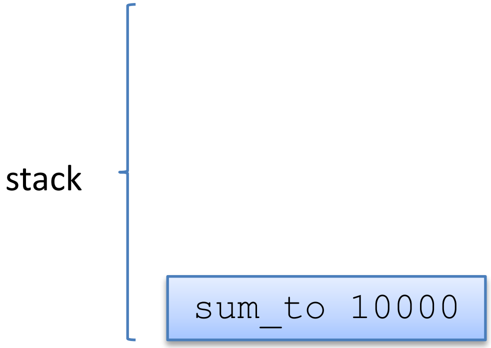
```
    sum_to 1000000
-->
    1000000 + sum_to 99999
-->
    1000000 + 99999 + sum_to 99998
-->
    ...
-->
    1000000 + 99999 + 99998 + ... + sum_to 0
-->
    1000000 + 99999 + 99998 + ... + 0
-->
    ... add it all back up ...
```

```
(* 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

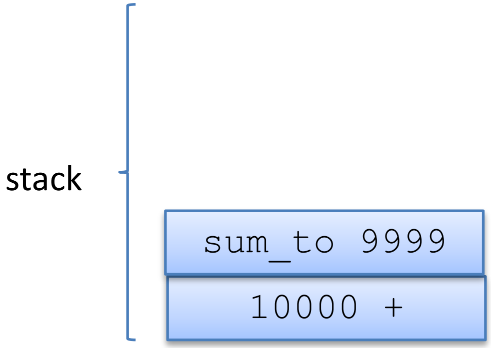
Non-tail recursive

```
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
```



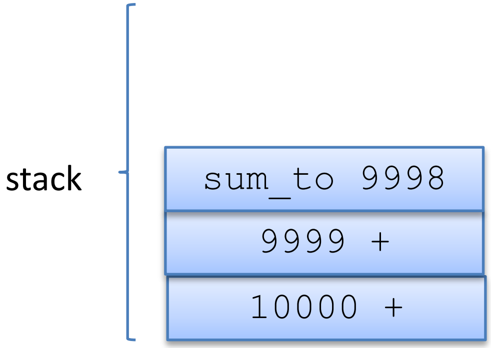
stack

sum_to 9999

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
```


stack

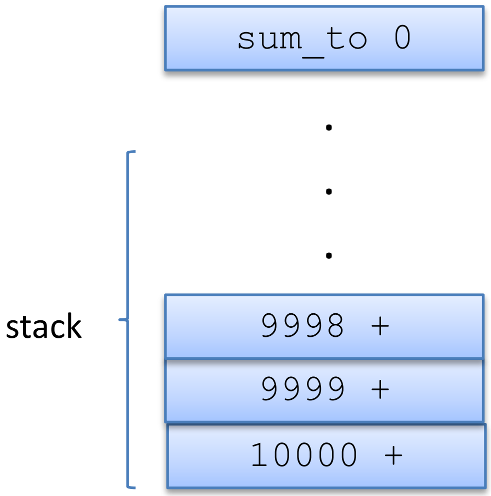
sum_to 9998

9999 +

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
```



sum_to 0

.
.
.

stack

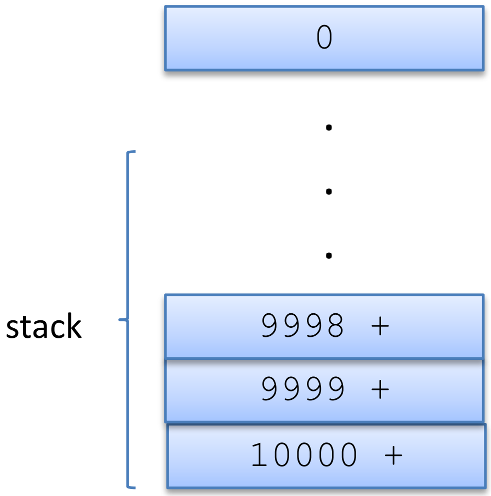
9998 +

9999 +

10000 +

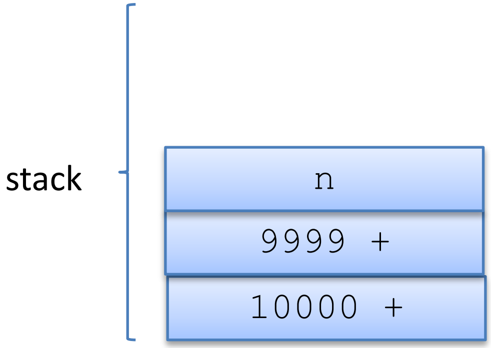
Non-tail recursive

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

sum_to 10000
```



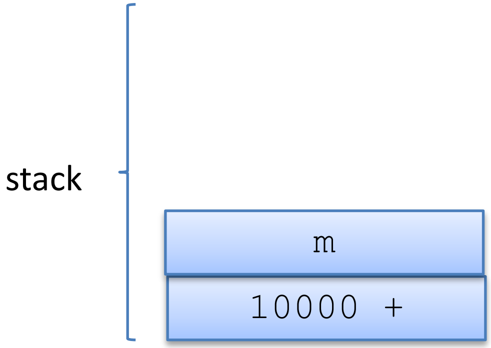
Non-tail recursive

```
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
```



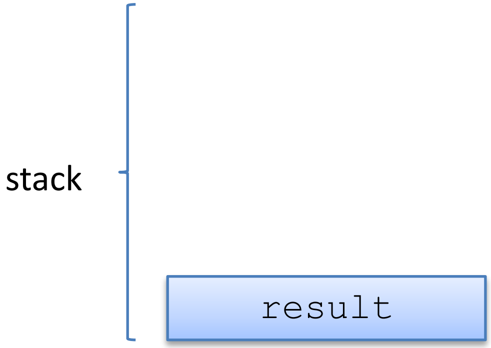
stack

m

10000 +

Non-tail recursive

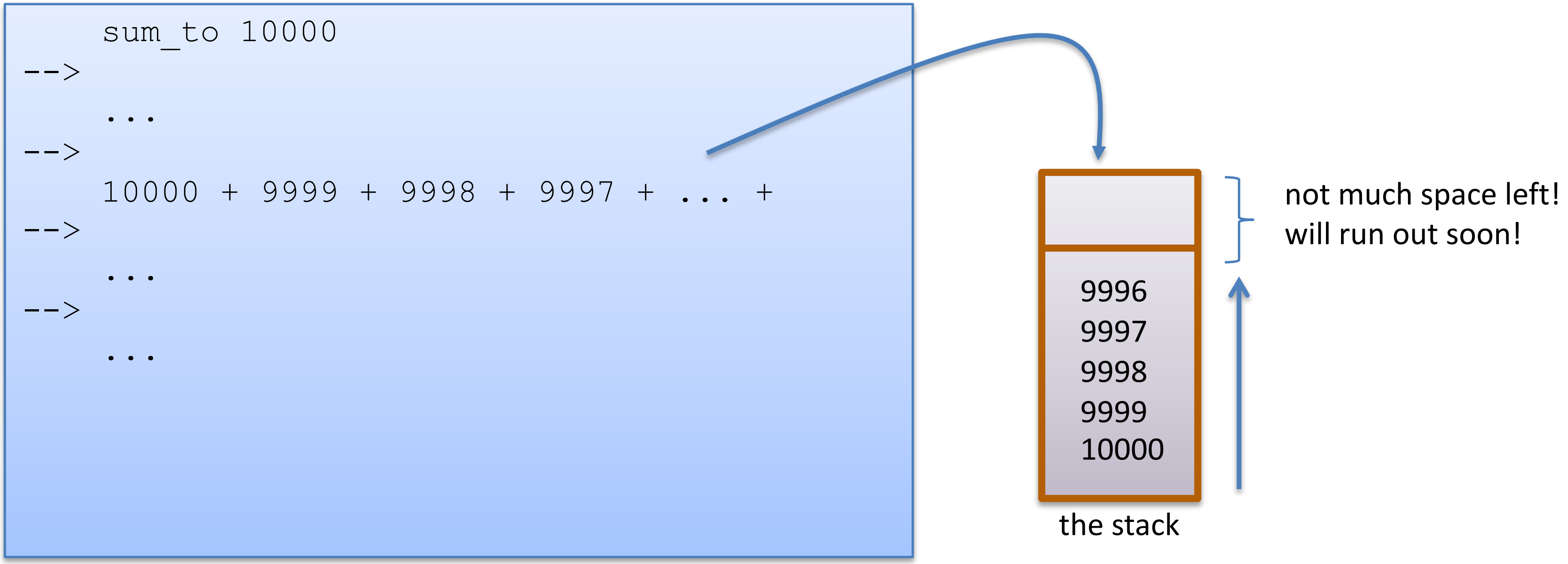
```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 100
```



stack

result

Data Needed on Return Saved on Stack



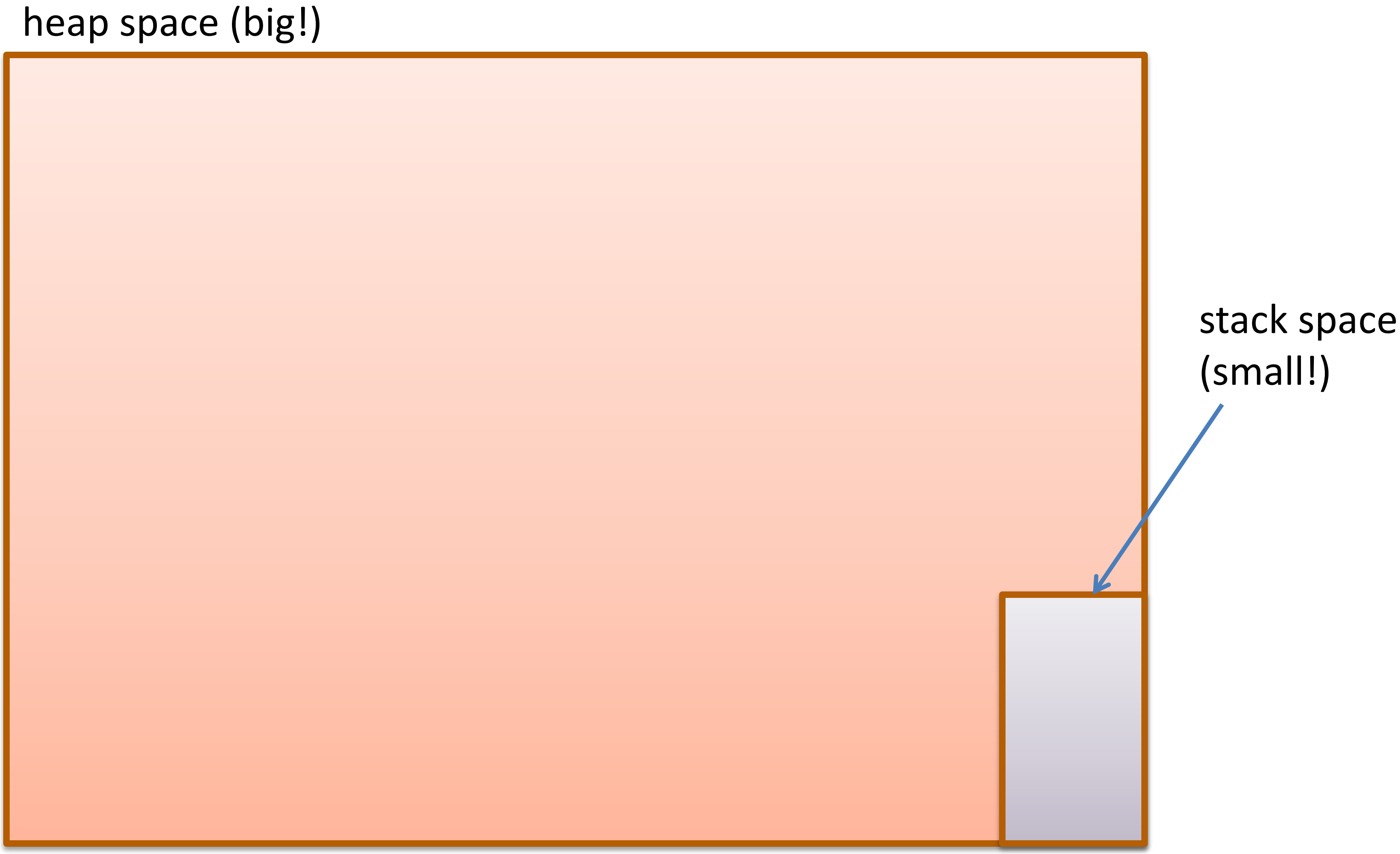
```
    sum_to 10000
-->
    ...
-->
    10000 + 9999 + 9998 + 9997 + ... +
-->
    ...
-->
    ...
```

the stack

not much space left!
will run out soon!

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
;;
```

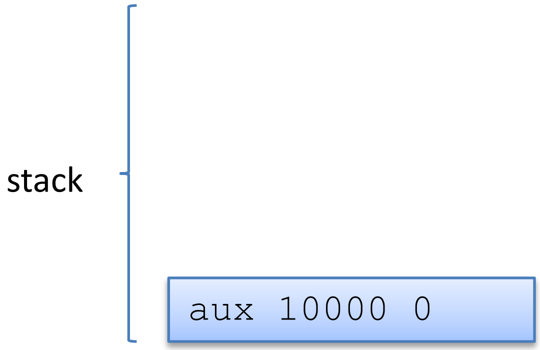
constant size expression
in the substitution model

(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
;;
```



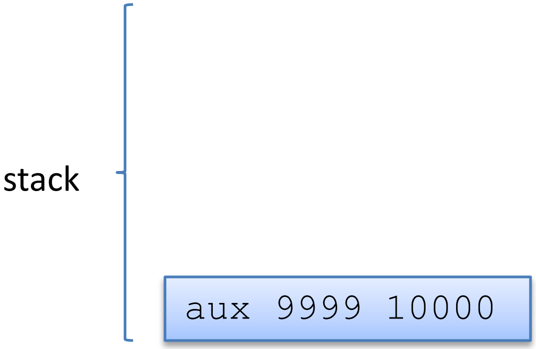
stack

aux 10000 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
;;
```



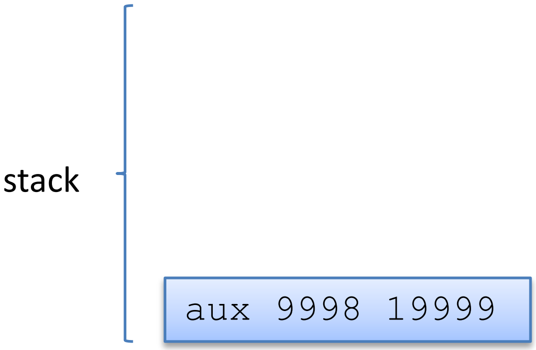
stack

```
aux 9999 10000
```

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
;;
```



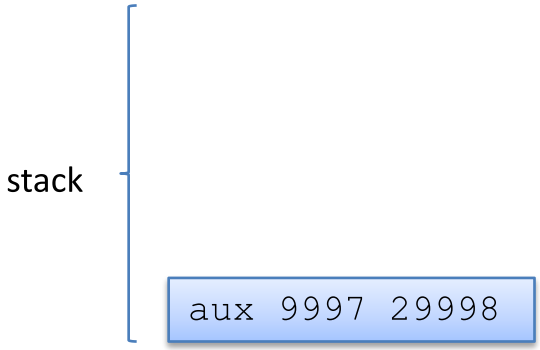
stack

aux 9998 19999

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
;;
```



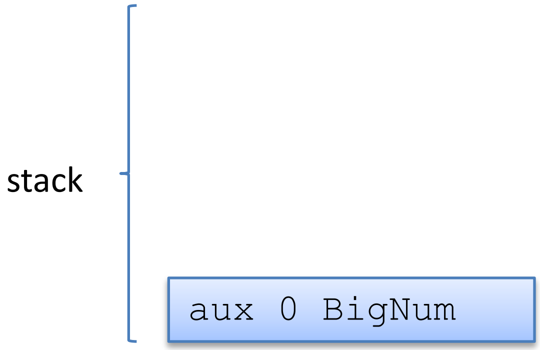
stack

`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
;;
```



stack

aux 0 BigNum

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:

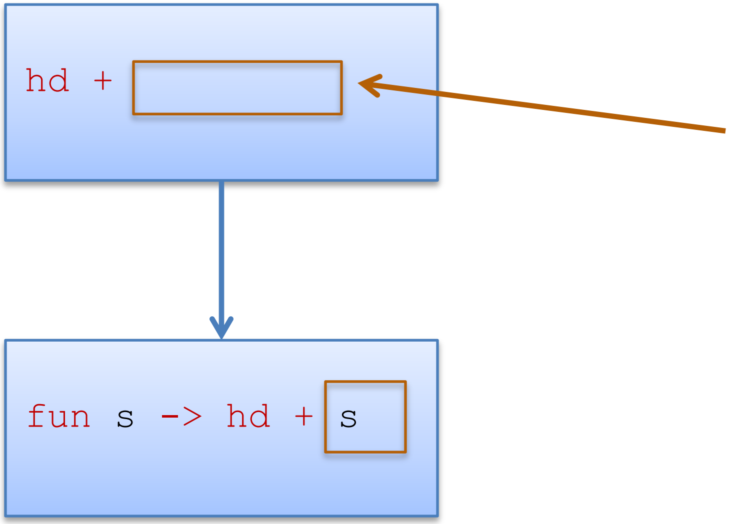
```
hd + 
```

result of recursive call gets plugged in here

How do we capture it?

Question

How do we capture that computation?



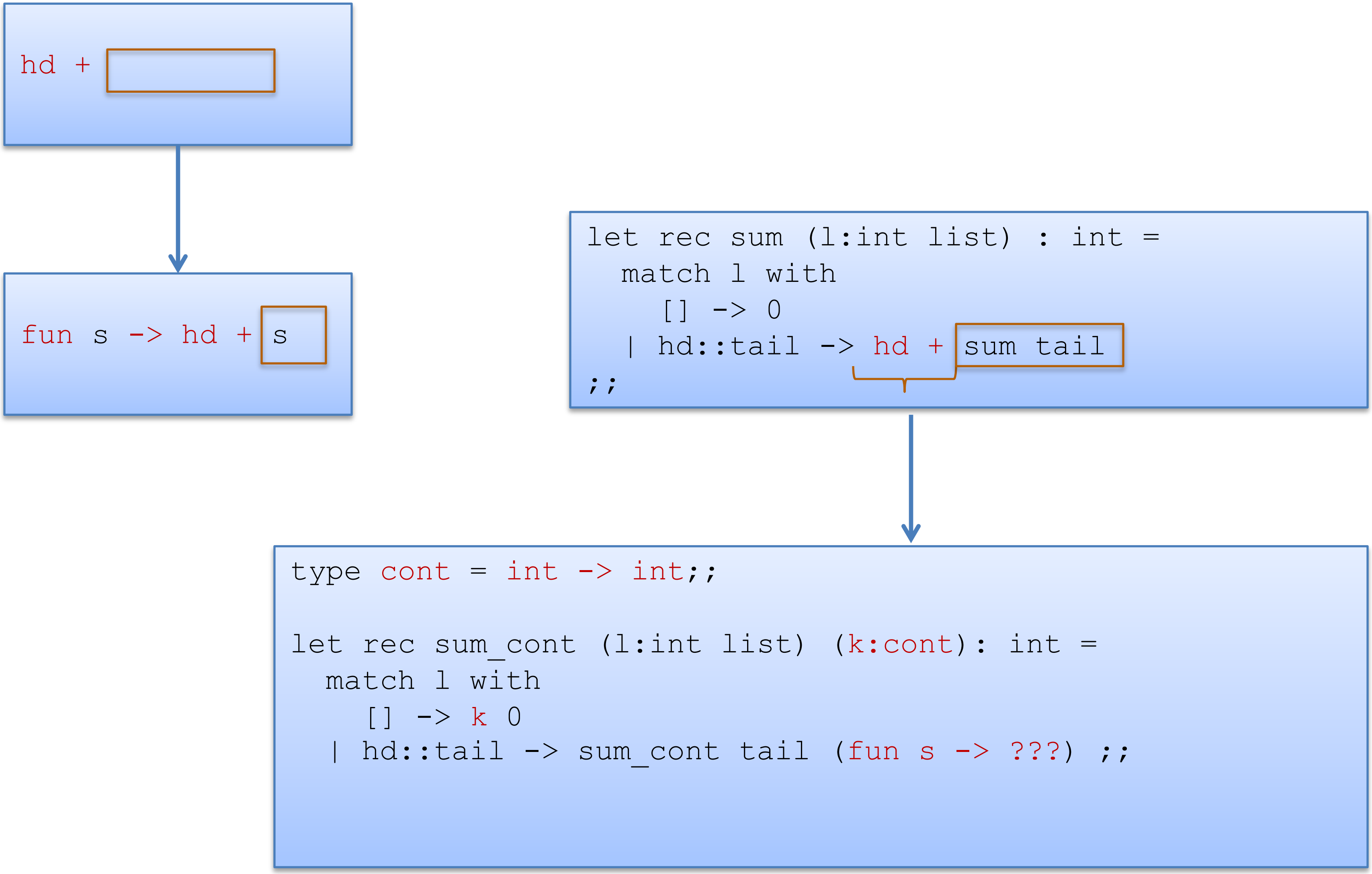
`hd + `

`fun s -> hd + s`

result of recursive call gets plugged in here

Question

How do we capture that computation?



```
hd +
```

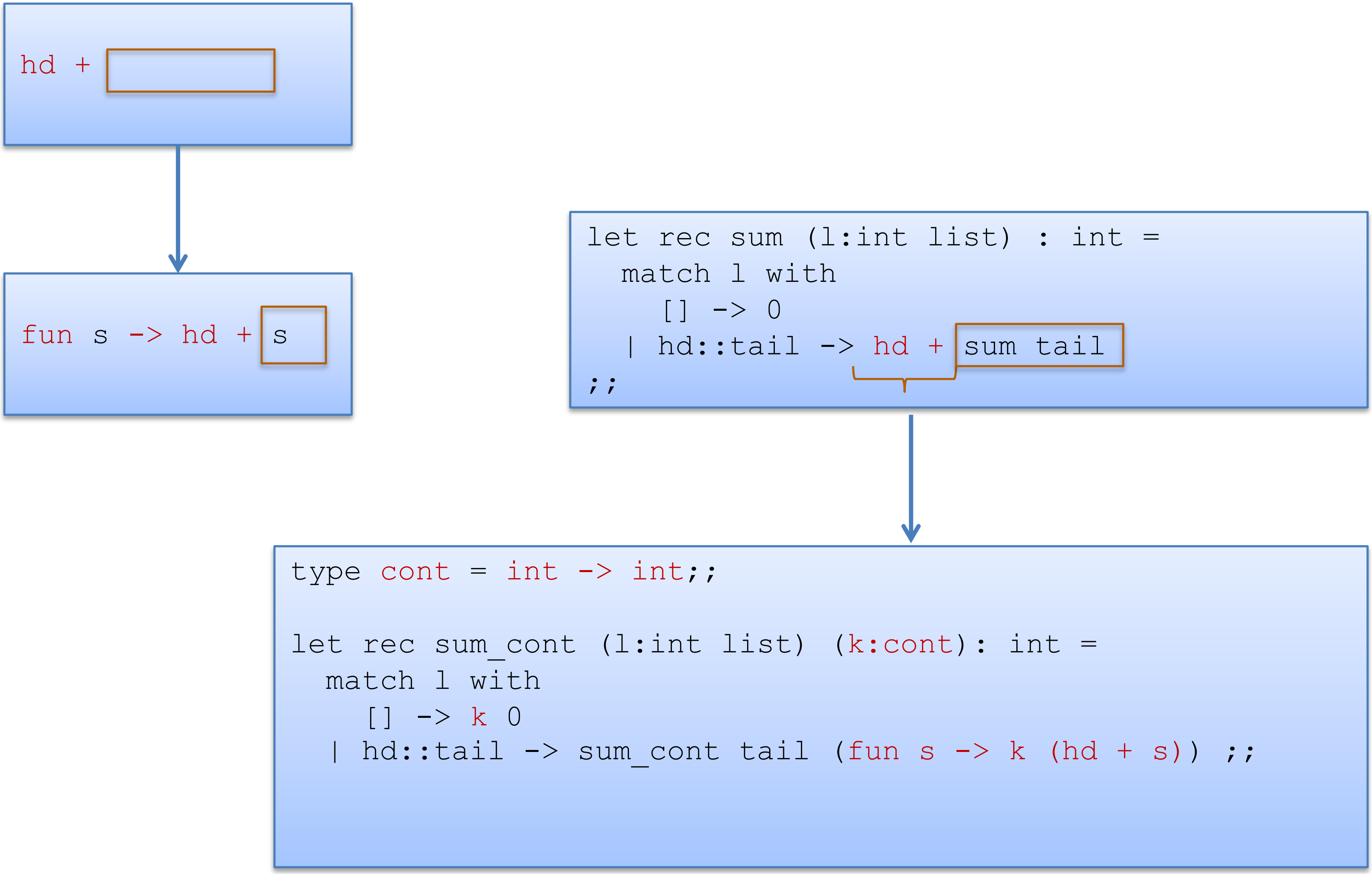
```
fun s -> hd + s
```

```
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 +
```

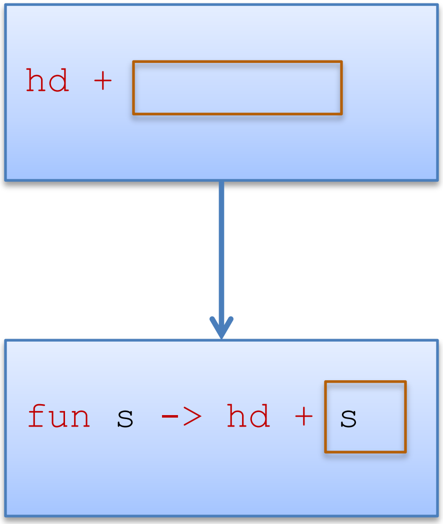
```
fun s -> hd + s
```

```
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?

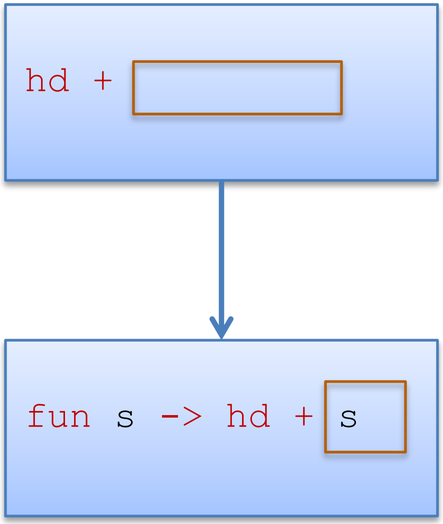


```
let rec sum (l:int list) : int =
  match l with
    [] -> 0
  | hd::tail -> hd + sum tail
;;
```

```
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
```

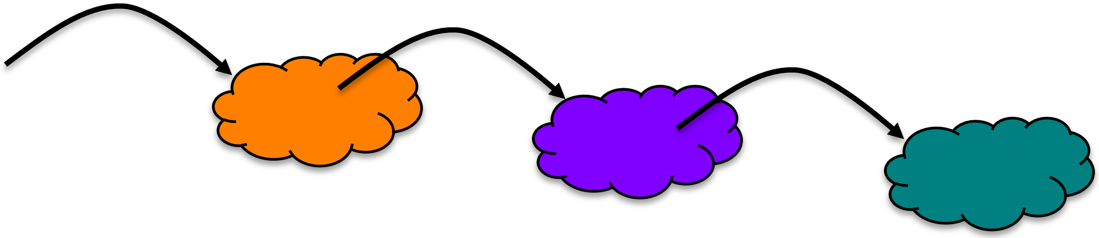
Where did the stack space go?

```
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

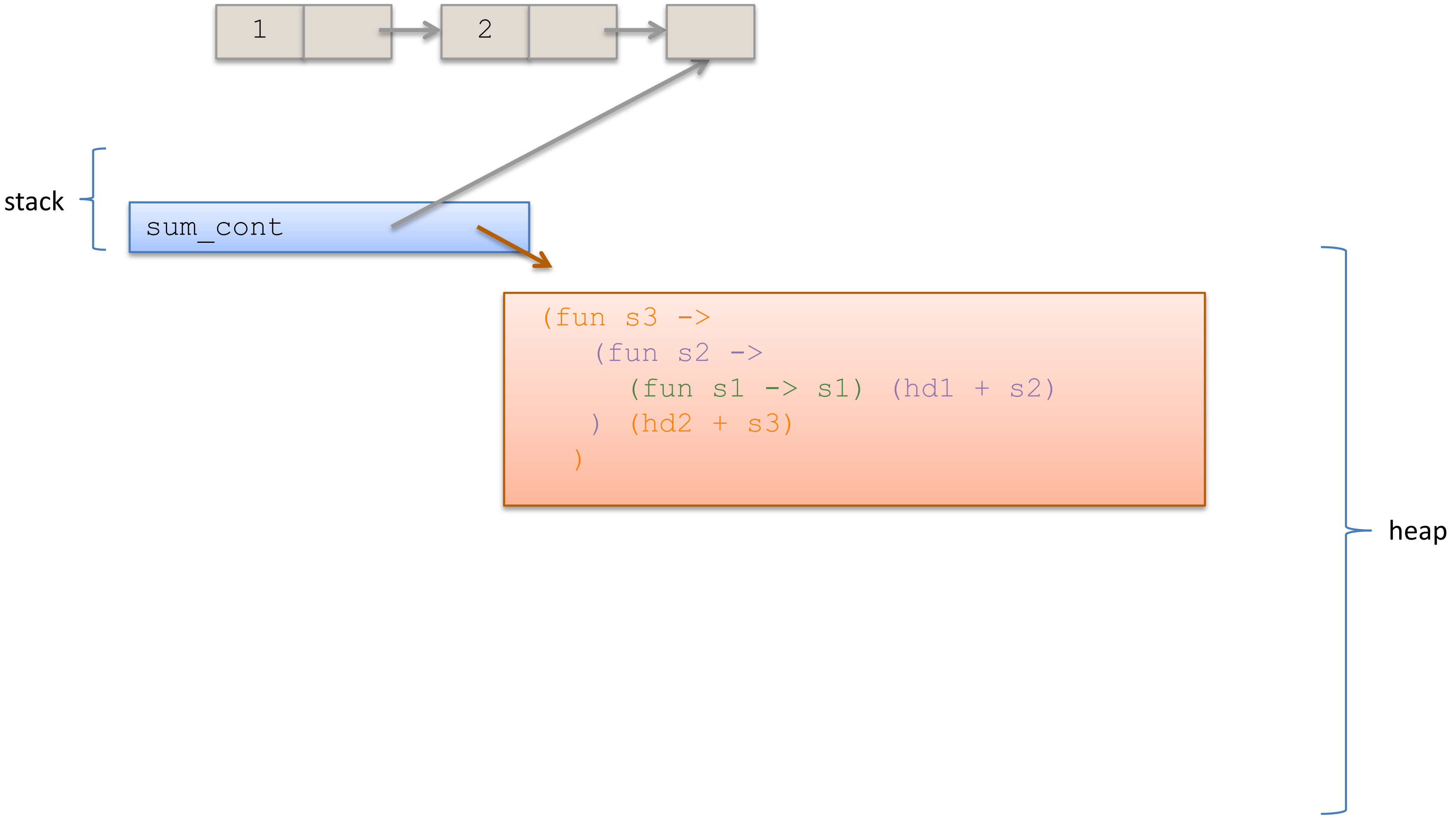
a stack of
closures on
the heap



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)
   )
```

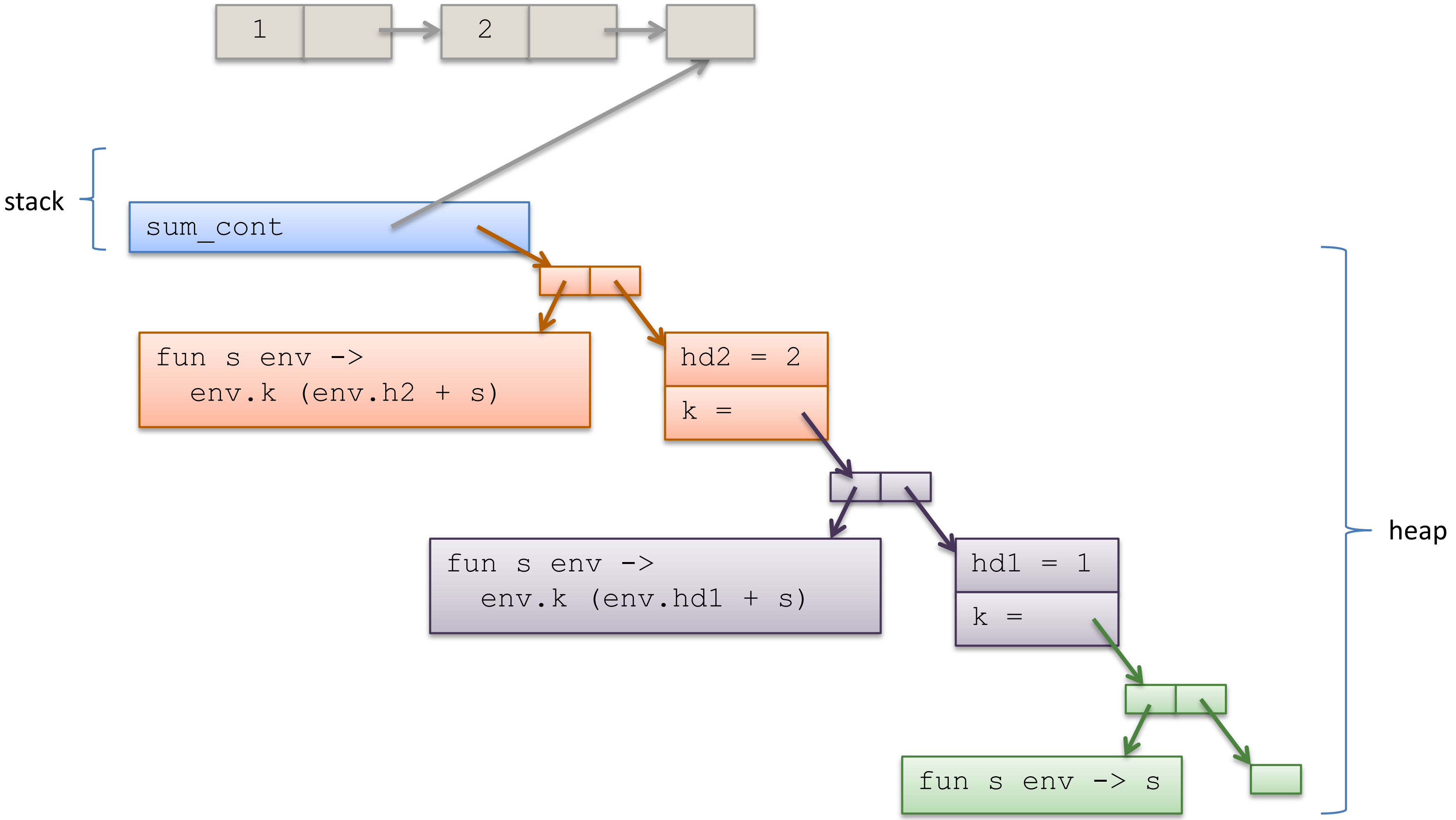


```
(fun s3 ->
   (fun s2 ->
     (fun s1 -> s1) (hd1 + s2)
   ) (hd2 + s3)
  )
```

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)
   )
```



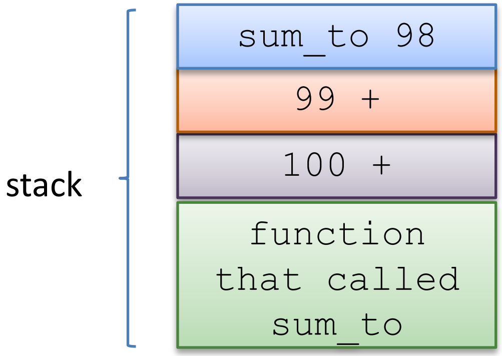
```
fun s env ->
  env.k (env.h2 + s)
```

```
fun s env ->
  env.k (env.hd1 + s)
```

```
fun s env -> 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
```



stack

sum_to 98

99 +

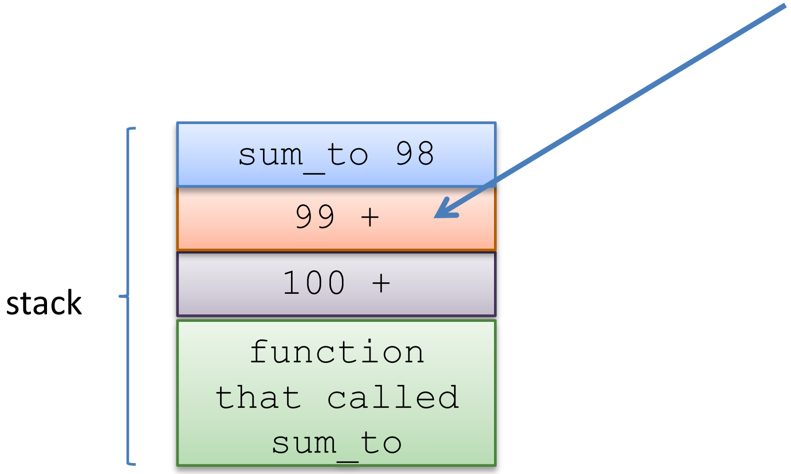
100 +

function that called sum_to

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?



stack

sum_to 98

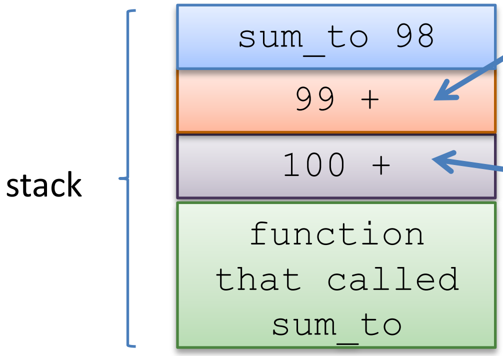
99 +

100 +

function that called sum_to

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?

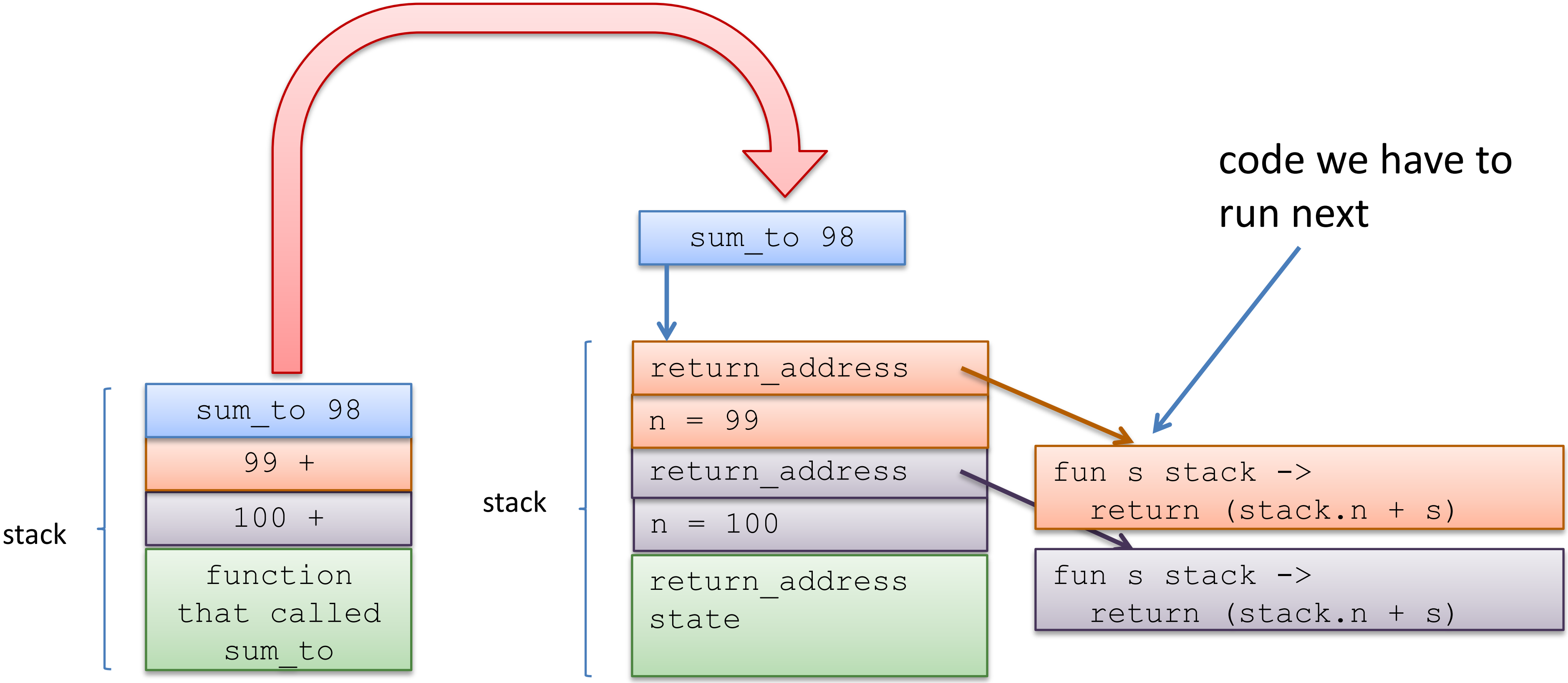
there is two bits of information here:

(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
```

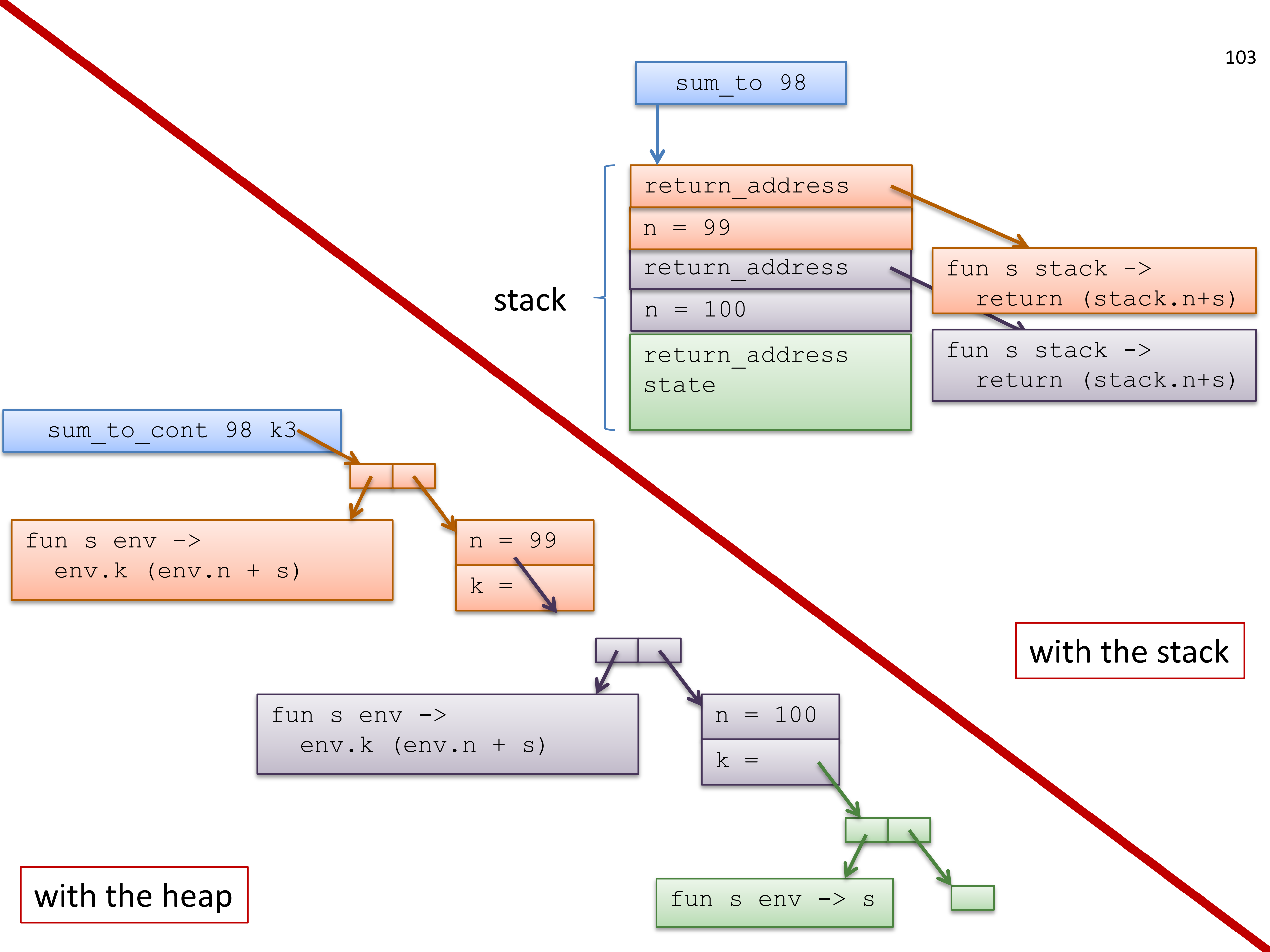


with reality added

code we have to run next

```
fun s stack ->
  return (stack.n + s)
```

```
fun s stack ->
  return (stack.n + s)
```



```
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)
```

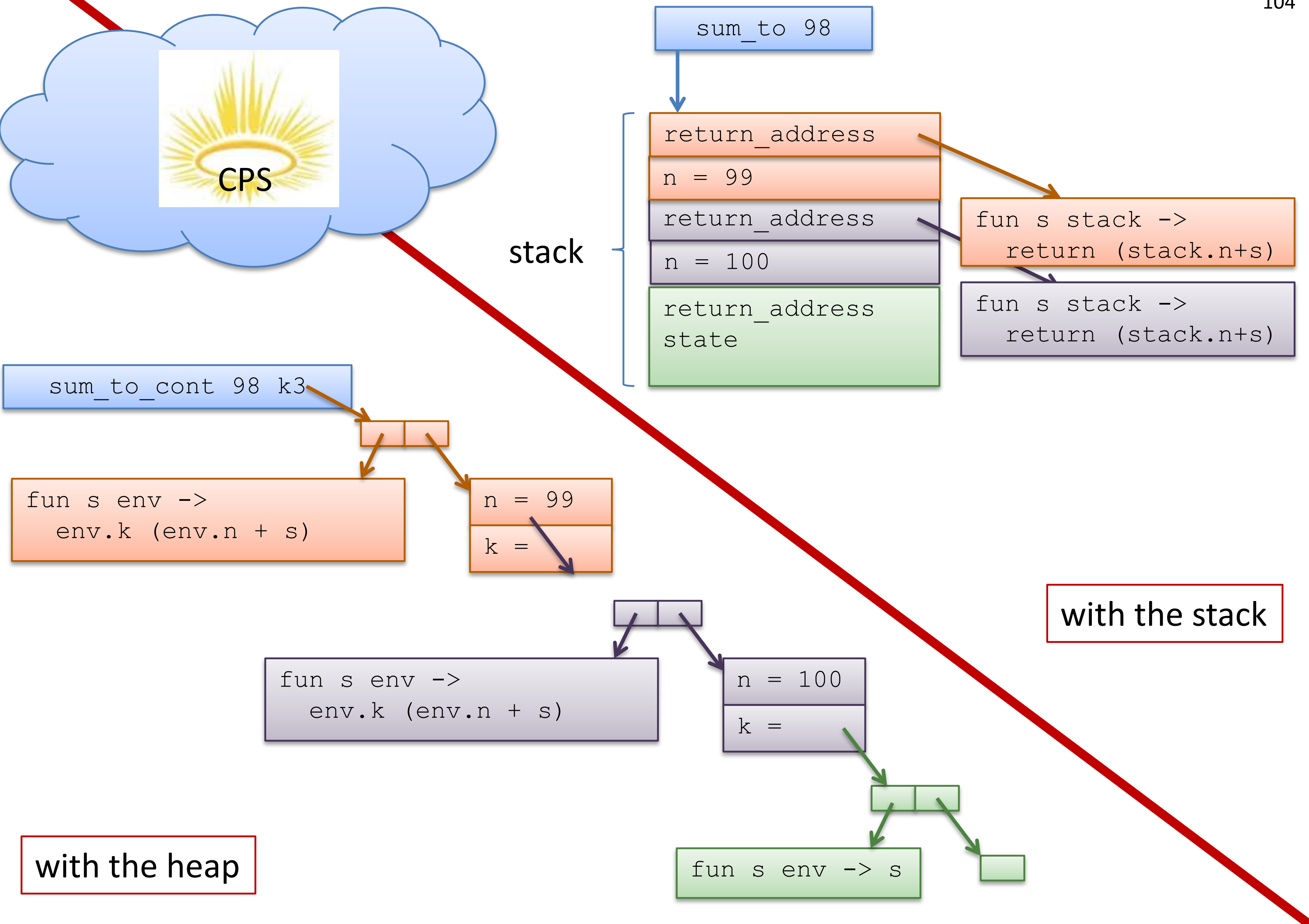
```
n = 99
k =
```

```
fun s env ->
  env.k (env.n + s)
```

```
n = 100
k =
```

```
fun s env -> s
```

with the heap



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*.

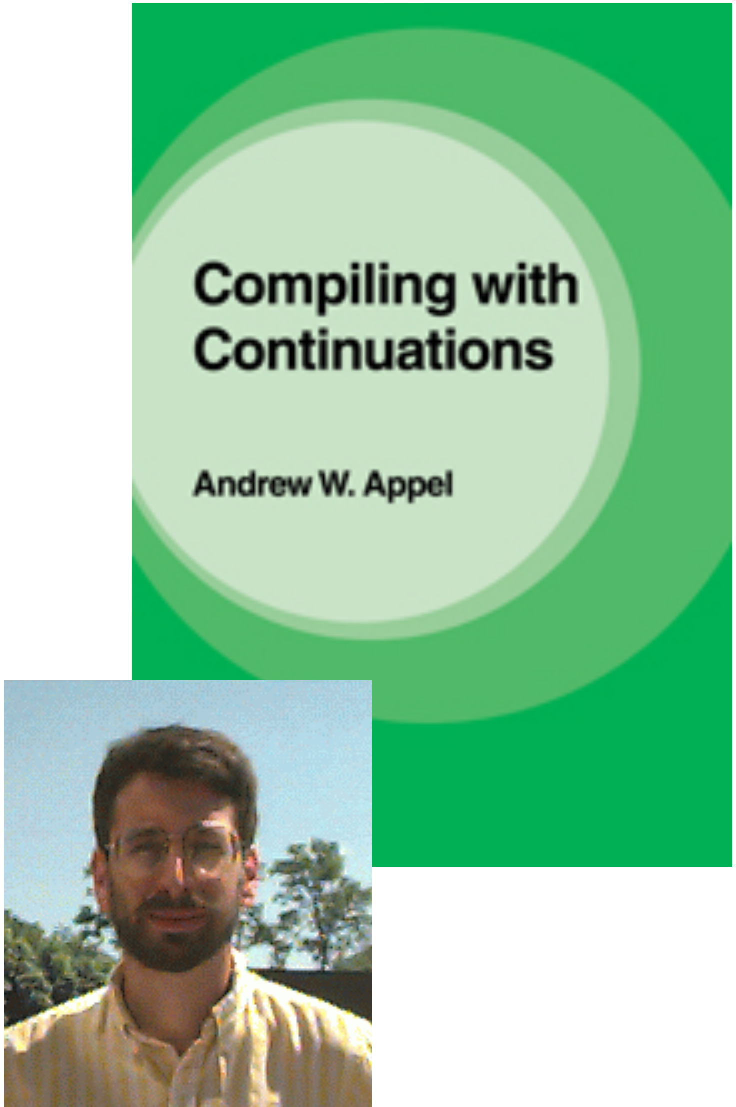
It does not matter whether you program in Java or C or OCaml -- there's code around that tells you "*what to do next*"

- 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



Compiling with Continuations

Andrew W. Appel

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

seeEmpirical 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

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