Continuations

COS 326
David Walker
Princeton University

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:

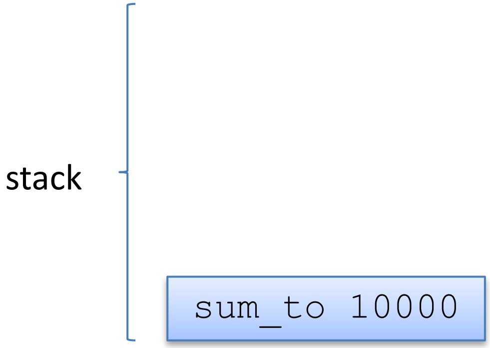
```
    sum_to 1000000
-->
    1000000 + sum_to 99999
-->
    1000000 + 99999 + sum_to 99998
-->
    ...
-->
    1000000 + 99999 + 99998 + ... + sum_to 0
-->
    1000000 + 99999 + 99998 + ... + 0
-->
    ... add it all back up ...
```

```
(* 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

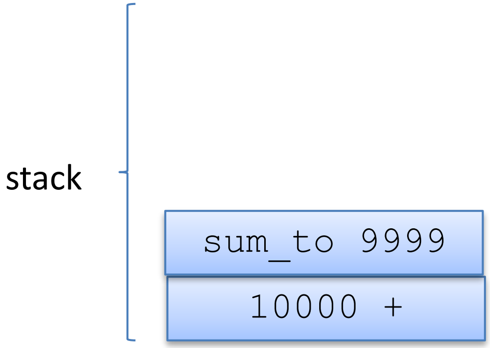
Non-tail recursive

```
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
```



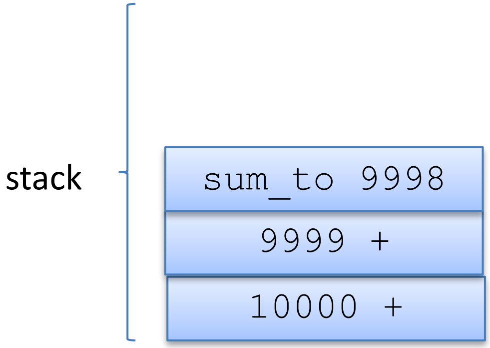
stack

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

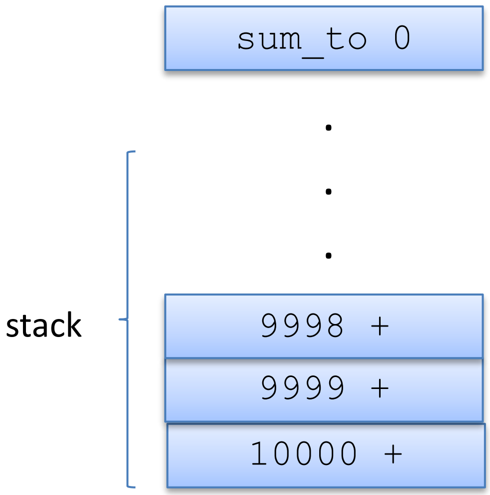
sum_to 9998

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

.
.
.

9998 +

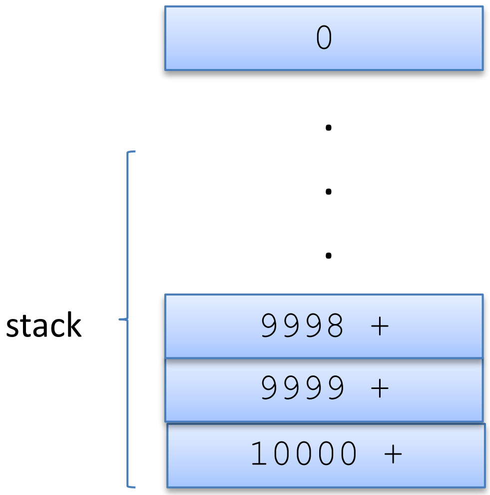
9999 +

10000 +

stack

Non-tail recursive

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

sum_to 10000
```



0

.

.

.

stack

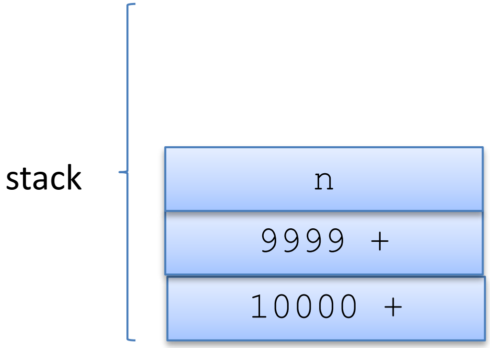
9998 +

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
```



stack

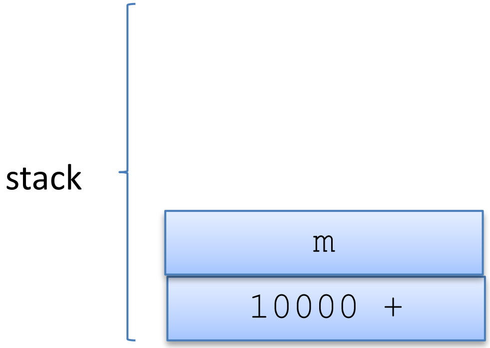
n

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
```



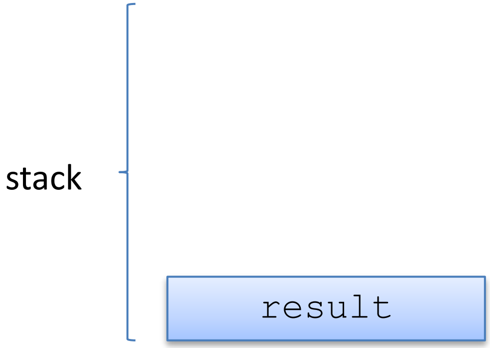
stack

m

10000 +

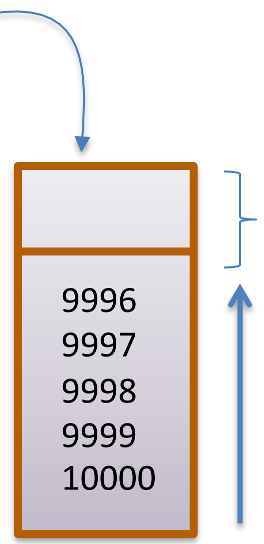
Non-tail recursive

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

sum_to 100
```



Data Needed on Return Saved on Stack

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



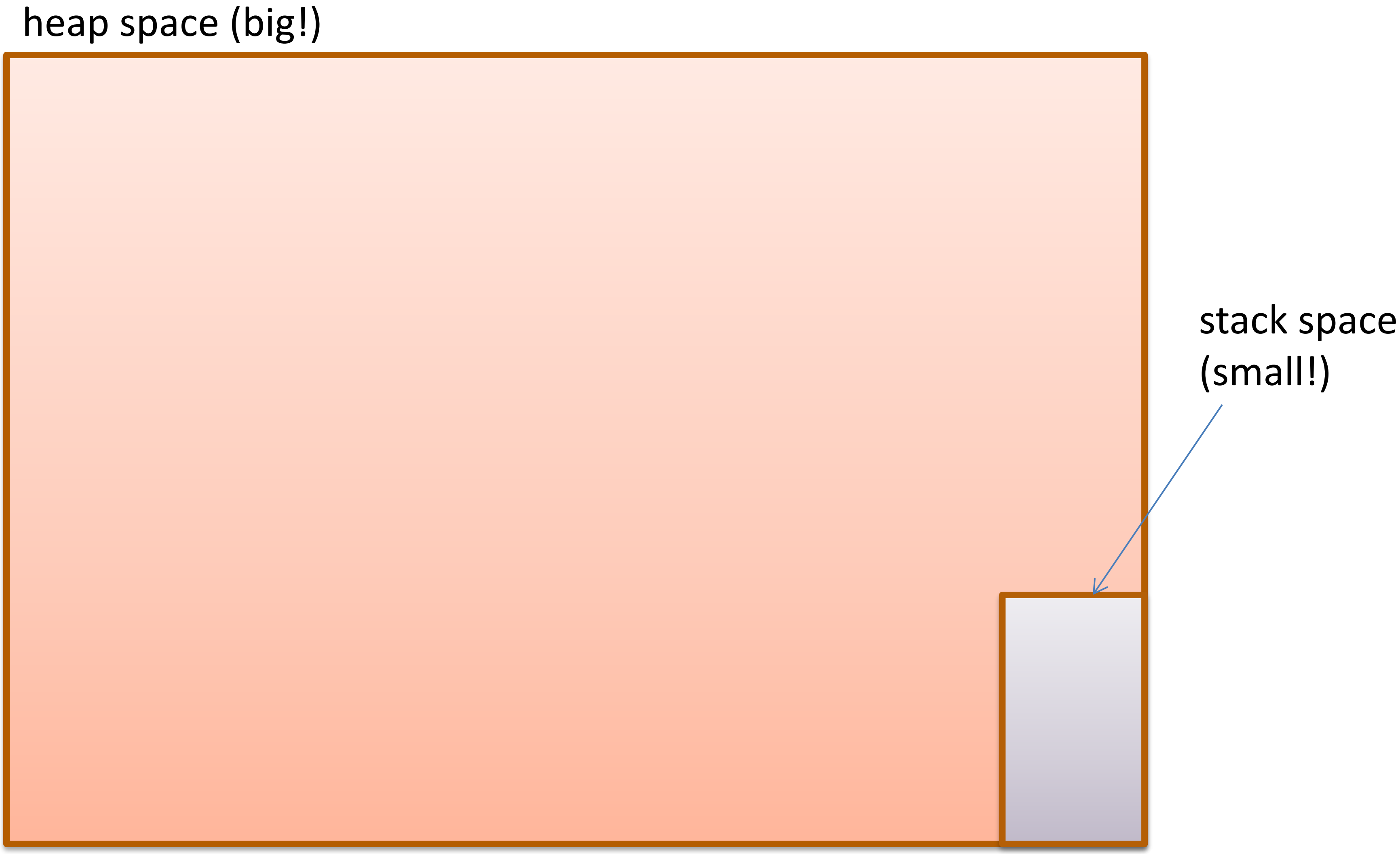
9996
9997
9998
9999
10000

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
;;
```

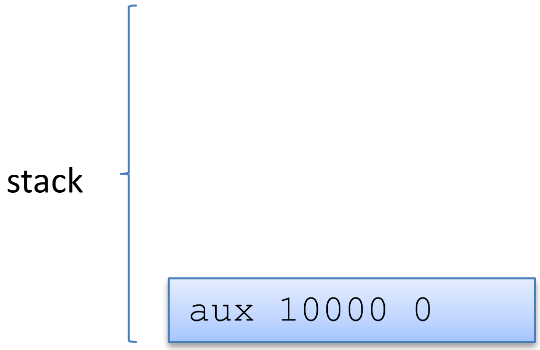
constant size expression
in the substitution model

(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
;;
```

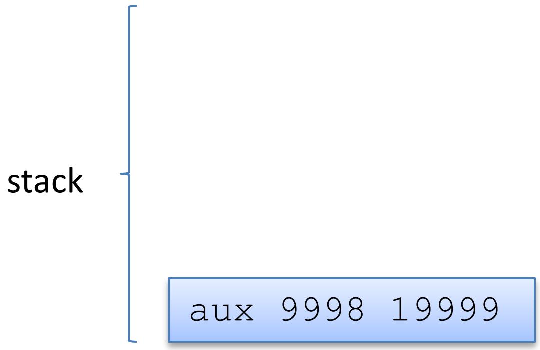
stack

```
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
;;
```



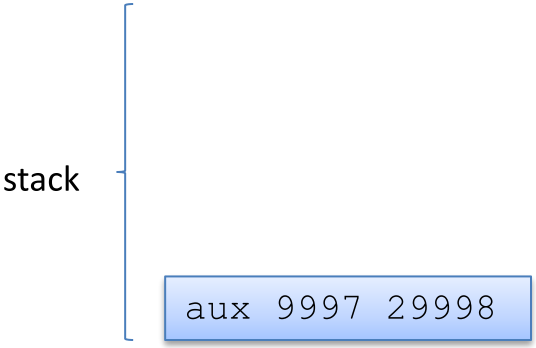
stack

`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
;;
```



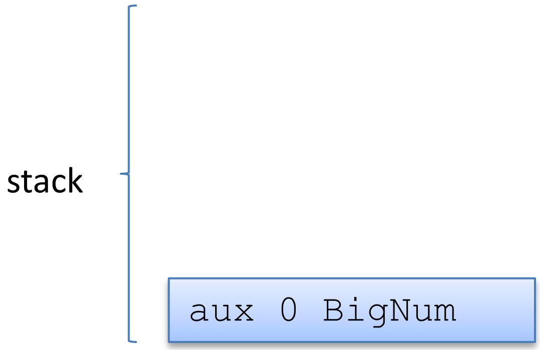
stack

`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
;;
```



Question

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

Is there a mechanical procedure to transform *any* recursive function into 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
;;
```

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*!

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!

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

Idea: Instead of returning to do some work, add an argument called a *continuation*. That continuation "does the rest of the work." Instead of returning to do work, call the continuation to do it.

Question

```
type cont = int -> int

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

needs
to be fixed up

Step 1: Add the continuation. Think of this as
"the work you have left to do" and always call it last to finish up that work

Question

```
type cont = int -> int

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

Step 2: Call the continuation on the base case, passing it the *result* of the current computation

Trust that the continuation is going to do the rest of the work that you've saved for later when you've finished fixing up your function.

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))
;;
```

To do after summing the tail:
add hd to the result (s) and then do continuation k

Step 3: On recursive calls, pass a new continuation that does the leftover work you were supposed to do after this call (plus the work of k)

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) = sum_cont l (fun s -> s)
```

Step 4: Generate the initial continuation (which does nothing – no leftover work at that start).

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
```

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.