Continuations

COS 326
David Walker
Princeton University
A tail-recursive function does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```ml
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:

```ml
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;;
```

```
sum_to 1000000
-->
1000000 + sum_to 999999
```
A tail-recursive function does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

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

Not tail-recursive, the substitution model:

```plaintext
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;;
```
A tail-recursive function does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

\[
\begin{align*}
\text{sum}_\text{to} & \ 1000000 \\
\rightarrow & \ 1000000 + \text{sum}_\text{to} \ 99999 \\
\rightarrow & \ 1000000 + 99999 + \text{sum}_\text{to} \ 99998 \\
\rightarrow & \ \ldots \\
\rightarrow & \ 1000000 + 99999 + 99998 + \ldots + \text{sum}_\text{to} \ 0 \\
\rightarrow & \ 1000000 + 99999 + 99998 + \ldots + 0
\end{align*}
\]

\[
(* \ \text{sum of 0..n *} *)
\]

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

Not tail-recursive, the substitution model:

```plaintext
(* 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;;
```

```
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 ...
```
Non-tail recursive

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

sum_to 10000
```
Non-tail recursive

```ocaml
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
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

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

sum_to 10000
```
Non-tail recursive

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

sum_to 10000
```
Non-tail recursive

```ocaml
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!

```
9996  
9997  
9998  
9999  
10000
```

every non-tail call puts the data from the calling context on the stack
Memory is partitioned: Stack and Heap

heap space (big!)

stack space (small!)
A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

```
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
;;
```

```
sum_to2 1000000
```

18
A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

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

```ml
(* 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:

```ocaml
(* 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
;;
```

```
sum_to2 1000000
-->
aux 1000000 0
-->
aux 99999 1000000
```
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
;;
```
A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

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

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

```ocaml
(* 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
;;
```

```ocaml
stack
```

```ocaml
aux 10000 0
```
Tail Recursion

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

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

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

```ocaml
(* 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?

```ocaml
let rec 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:

- 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 into CPS
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.
Can any non-tail-recursive function be transformed into a tail-recursive one? Yes!

```ocaml
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.
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.
Step 2: Call the continuation on the base case, passing it the \textit{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.
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)
Step 4: Generate the initial continuation (which does nothing – no leftover work at that start).
type \texttt{cont} = \texttt{int} -> \texttt{int};

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

let sum (l:\texttt{int list}) : \texttt{int} = sum_cont l (fun \texttt{s} -> \texttt{s})

sum [1;2]
type \texttt{cont} = \texttt{int} -> \texttt{int};;

let rec \texttt{sum\_cont} (l:\texttt{int\ list}) (k:\texttt{cont}) : \texttt{int} =
    match l with
    [] -> k 0
    | hd::tail -> \texttt{sum\_cont} tail (fun s -> k (hd + s)) ;;

let \texttt{sum} (l:\texttt{int\ list}) : \texttt{int} = \texttt{sum\_cont} l (fun s -> s)

\texttt{sum} [1;2]

--> \texttt{sum\_cont} [1;2] (fun s -> s)
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));;
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)
type \textit{cont} = int -> int;;

let rec \textit{sum\_cont} \ (l:int\ list) \ (k:\textit{cont}): int =
match \textit{l} with
\[
[] \to k\, 0
\]
| \textit{hd::tail} \to \textit{sum\_cont} \ \textit{tail} \ (fun \ \textit{s} \to k\ (\textit{hd} + \textit{s})) ;;\]

let \textit{sum} \ (l:int\ list) : int = \textit{sum\_cont} \ \textit{l} \ (fun \ \textit{s} \to \textit{s})

\[
\text{sum} \ [1;2] \\
\to \\
\text{sum\_cont} \ [1;2] \ (fun \ \textit{s} \to \textit{s}) \\
\to \\
\text{sum\_cont} \ [2] \ (fun \ \textit{s} \to (fun \ \textit{s} \to \textit{s}) \ (1 + \textit{s})) ;; \\
\to \\
\text{sum\_cont} \ [] \ (fun \ \textit{s} \to (fun \ \textit{s} \to (fun \ \textit{s} \to \textit{s}) \ (1 + \textit{s})) \ (2 + \textit{s})) \\
\to \\
(fun \ \textit{s} \to (fun \ \textit{s} \to (fun \ \textit{s} \to \textit{s}) \ (1 + \textit{s})) \ (2 + \textit{s})) \ 0
\]
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)
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)
type \texttt{cont} = \texttt{int} \rightarrow \texttt{int};

let rec \texttt{sum\_cont} (l:int list) (k:cont): int =
  match l with
  \[\] \rightarrow k 0
  | \texttt{hd::tail} \rightarrow \texttt{sum\_cont} \texttt{tail} (fun s \rightarrow k (\texttt{hd} + s)) ;;

let \texttt{sum} (l:int list) : int = \texttt{sum\_cont} l (\texttt{fun s \rightarrow s})

\texttt{sum} [1;2]
\rightarrow \texttt{sum\_cont} [1;2] (\texttt{fun s \rightarrow s})
\rightarrow \texttt{sum\_cont} [2] (\texttt{fun s \rightarrow (fun s \rightarrow s) (1 + s)}) ;;
\rightarrow \texttt{sum\_cont} [] (\texttt{fun s \rightarrow (fun s \rightarrow (fun s \rightarrow s) (1 + s)) (2 + s)})
\rightarrow (\texttt{fun s \rightarrow (fun s \rightarrow (fun s \rightarrow s) (1 + s)) (2 + s)}) 0
\rightarrow (\texttt{fun s \rightarrow (fun s \rightarrow s) (1 + s)}) (2 + 0)
\rightarrow (\texttt{fun s \rightarrow s}) (1 + (2 + 0))
\rightarrow 1 + (2 + 0)
\rightarrow 3
CORRECTNESS OF A CPS TRANSFORM
Are the two functions the same?

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:

\[
\text{for all } l:\text{int list},
\quad \text{sum\_cont } l \ (\text{fun } x \to x) = \text{sum } l
\]
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
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)
    ==
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)
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)
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
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)
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 [])
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:
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)
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 [])
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!
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))
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)
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)))
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))
                             (eval, since sum total and
                             and sum tail valuable)
== k (hd + (sum 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
== 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
== k (sum (hd::tail))                        and sum tail valuable)

case done!
QED!
Finishing Up

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

\[
\text{for all } l: \text{int list}, \\
\text{for all } k: \text{int}\rightarrow\text{int}, \quad \text{sum\_cont } l \ k == k \ (\text{sum } l)
\]

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

\[
\text{for all } l: \text{int list}, \\
\text{sum2 } l \\
== \text{sum\_cont } l \ (\text{fun } s \rightarrow s) \quad \text{(by eval sum2)} \\
== (\text{fun } s \rightarrow s) \ (\text{sum } l) \quad \text{(by theorem, instantiating } k \text{ with (fun } s \rightarrow s)\text{)} \\
== \text{sum } l \quad \text{(by eval, since sum } l \text{ valuable)}
\]

So, we've show that the function sum2, which is tail-recursive, is functionally equivalent to the non-tail-recursive function sum.
SUMMARY
CPS 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 \textit{specific} theorem we wanted:

$$\text{for all } l: \text{int list}, \; \text{sum\_cont } l \ (\text{fun } s \rightarrow s) = \text{sum } l$$

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

$$\text{for all } l: \text{int list}, \quad \text{for all } k: \text{int}\rightarrow\text{int}, \; \text{sum\_cont } l \ k = k \ (\text{sum } l)$$

This is a common occurrence -- \textit{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.