Continuing CPS

COS 326
Andrew W. Appel
Princeton University

slides copyright 2013-2015 David Walker and Andrew W. Appel
Last Time

We saw that code like this could take up a lot of stack space:

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

We saw that code like this could take up a lot of stack space:

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

work to do after the recursive call returns

== not a tail-recursive function
Last Time

We saw that code like this could take up a lot of stack space:

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

Every recursive call requires we allocate a “stack frame” to store the return address + data used after the call.
Last Time

But we can capture the computation that happens after the call:

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

fun result -> n + result
```

A function that explains “what to do next” is called a *continuation*. 
 Conversion to Continuation-Passing Style:

let rec sum_to_cont (n:int) (k:int -> int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun result -> k (n + result))
  else
    0
Many functions can be made tail-rec easily:

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

sum_to 100
```

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

sum_to_opt 100
```

not only did we make the function tail-recursive, but we didn’t add any stack or linked-list of closures!
Many functions can be made tail-rec easily:

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

sum_to 100
```

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

sum_to_opt 100
```

not only did we make the function tail-recursive, but we didn’t add any stack or linked-list of closures!
ANOTHER EXAMPLE
Challenge: CPS Convert the incr function

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

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

Hint: It is a little easier to put the continuations in the order in which they are called.
```
Challenge: CPS Convert the incr function

```ml
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)
;;
```

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

```ocaml
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)
;;
```

called A-Normal Form (intermediate computations given names; no function calls as args to other function calls)
Challenge: CPS Convert the incr function

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

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

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

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

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

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

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

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

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

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

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

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

let g input =
  f3 (f2 (f1 input))

let g input k =
  f1 input (fun x1 ->
    f2 x1 (fun x2 ->
      f3 x2 k))

Direct Style

let g input =
  let x1 = f1 input in
  let x2 = f2 x1 in
  f3 x2

A-normal Form

let g input =
  let x1 = f1 input in
  let x2 = f2 x1 in
  f3 x2

CPS converted
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
```

CPS is pretty tricky. Let's try to prove a theorem:

```
for all l:int list,
    sum_cont l (fun x -> x) == sum l
```
Theorem: For all \( l : \text{int list} \), \( \text{sum\_cont} \ l \ (\text{fun} \ s \to s) == \text{sum} \ l \)

Proof: By induction on the structure of the list \( l \).

case \( l = [] \)
  ...

case: \( \text{hd} :: \text{tail} \)
  IH: \( \text{sum\_cont} \ \text{tail} \ (\text{fun} \ s \to s) == \text{sum} \ \text{tail} \)
Theorem: 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)
  ==
Theorem: 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)
Theorem: 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)
Theorem: For all l:int list, \( \text{sum\_cont}\ l \ (\text{fun} \ s \to s) = \text{sum}\ l \)

Proof: By induction on the structure of the list \( l \).

\text{case}\ l = []

... 

\text{case}: \text{hd}::\text{tail}

\text{IH}: \text{sum\_cont}\ \text{tail} \ (\text{fun} \ s \to s) = \text{sum}\ \text{tail}

\begin{align*}
\text{sum\_cont}\ (\text{hd}::\text{tail}) \ (\text{fun} \ s \to s) & = \text{sum\_cont}\ \text{tail} \ (\text{fn} \ s' \to (\text{fn} \ s \to s) \ (\text{hd} + s')) \quad \text{(eval)} \\
& = \text{sum\_cont}\ \text{tail} \ (\text{fn} \ s' \to \text{hd} + s') \quad \text{(eval)} \\
& = \text{darn!}
\end{align*}

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

\text{sum\_cont}\ \text{tail} \ (\text{fn} \ s' \to \text{hd} + s') = \text{sum}\ \text{tail}

... but that's not even true

not the identity continuation (fun s -> s) like the IH requires
for all \( l: \text{int} \) list, 
for all \( k: \text{int} \to \text{int} \), \( \text{sum}_{\text{cont}} \ l \ k = k \ (\text{sum} \ l) \)
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 \( l: \text{int list}, \)
\[
\text{for all } k: \text{int->int}, \; \text{sum\_cont } l \; k = k \; (\text{sum } l)
\]

Proof: By induction on the structure of the list \( l.\)

case \( l = []\)

must prove: for all \( k: \text{int->int}, \; \text{sum\_cont } [] \; k = k \; (\text{sum } [])

pick an arbitrary \( k:\)

\[
\text{sum\_cont } [] \; k
\]
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
  == match [] with [] -> k 0 | hd::tail -> ...                  (eval)
  == k 0

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

Proof: By induction on the structure of the list \( l \).

case \( l = [] \)

must prove: \[ \text{for all } k : \text{int}\rightarrow\text{int}, \sum\_\text{cont} \ [] \ k = k (\sum \ []) \]

pick an arbitrary \( k \):

\[ \sum\_\text{cont} \ [] \ k \]

== match \( [] \) with \( [] \) \rightarrow \ k 0 \mid \text{hd}::\text{tail} \rightarrow \ldots \] (eval)

== \( k 0 \) (eval)

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

Proof: By induction on the structure of the list \( l \).

case \( l = [] \)

must prove: for all \( k: \text{int} \rightarrow \text{int} \), \( \text{sum\_cont} \; [] \; k = k \cdot (\text{sum} \; []) \)

pick an arbitrary \( k \):

\[
\text{sum\_cont} \; [] \; k
\]
\[
= \text{match} \; [] \; \text{with} \; [] \rightarrow k \; 0 \mid \text{hd}::\text{tail} \rightarrow \ldots \quad \text{(eval)}
\]
\[
= k \; 0 \quad \text{(eval)}
\]
\[
= k \; (0) \quad \text{(eval, reverse)}
\]
\[
= k \; (\text{match} \; [] \; \text{with} \; [] \rightarrow 0 \mid \text{hd}::\text{tail} \rightarrow \ldots) \quad \text{(eval, reverse)}
\]
\[
= k \; (\text{sum} \; [])
\]

case done!
for all \(l:\text{int list,}
\begin{align*}
  \text{for all } k:\text{int->int, } \text{sum}_{\text{cont}} l \ k &= k \ (\text{sum } l) 
\end{align*}
\)

Proof: By induction on the structure of the list \(l\).

case \(l = []\) ===> done!

case \(l = \text{hd::tail}\)

IH: \(\text{for all } k':\text{int->int, } \text{sum}_{\text{cont}} \text{tail} \ k' = k' \ (\text{sum tail})\)

MP: \(\text{for all } k:\text{int->int, } \text{sum}_{\text{cont}} (\text{hd::tail}) \ k = k \ (\text{sum (hd::tail)})\)
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)

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

  Pick an arbitrary k,

    sum_cont (hd::tail) 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 = [] ===> done!

case l = hd::tail

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

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

   Pick an arbitrary k,

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

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

      Pick an arbitrary k,

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

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

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

  Pick an arbitrary k,

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

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

  == k (hd + (sum tail)) (eval, since sum total 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)

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

   Pick an arbitrary k,

   sum_cont (hd::tail) k
   == sum_cont tail (fun s -> k (hd + x)) (eval)
   == (fun s -> k (hd + s)) (sum tail) (IH)
   == k (hd + (sum tail)) (eval)
   == 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 \( l: \text{int list} \),
    for all \( k: \text{int} \rightarrow \text{int} \), \( \text{sum\_cont} \ l \ k \ = \ k \ (\text{sum} \ l) \)

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

for all \( l: \text{int list} \),
    \( \text{sum2} \ l \)
    \( = \ \text{sum\_cont} \ l \ (\lambda s. s) \) \quad \text{(by eval sum2)}
    \( = \ (\lambda s. s) \ (\text{sum} \ l) \) \quad \text{(by theorem, instantiating } k \text{ with } (\lambda s. s) \)
    \( = \ \text{sum} \ l \)

So, we've show that the function \( \text{sum2} \), which is tail-recursive, is functionally equivalent to the non-tail-recursive function \( \text{sum} \).
Summary of the CPS Proof

We tried to prove the 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\text{)} like our IH required. So we had to \textit{prove a more general theorem} about \textit{all} continuations.

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

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.