Continuing CPS

COS 326
David Walker
Princeton University

Last Time

Tail-recursive functions

- the recursive call is the last thing they do in a function

Continuation-passing style

- Any function can be made tail-recursive by passing it an extra argument – *a continuation*
  - Bottle up the stuff you might do after returning from a function and make it into a "continuation"
  - Many OS interfaces use continuations too: they are called “call backs” in that context

An Example

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

stuff that happens after the recursive call

Call continuation as last thing you do

add continuation argument

```
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)
```

last thing sum_cont does is to call this

Do your last thing:
hd +
after summing tail.
Then do k!

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)
==
```

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

Attempting a Proof

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

Proof: By induction on the structure of the list l.

case l = []
  ...

case: hd::tail
  IH: sum_cont tail (fun s -> s) == sum tail

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

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

Attempting a Proof

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

Proof: By induction on the structure of the list l.

case l = []
  ...

case: hd::tail
  IH: sum_cont tail (fun s -> s) == sum tail

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

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

Need to Generalize the Theorem and IH

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

Proof: By induction on the structure of the list l.

case l = []
  ...

case: hd::tail
  IH: sum_cont tail (fun s -> s) == sum tail

   sum_cont (hd::tail) (fun s -> s)
== sum_cont tail (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)
```

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)
  == 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)
```

But what we wanted was:

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

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)
```

But what we wanted was:

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

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

Theorem 2:

```
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)
```

WHAT JUST HAPPENED? GENERALIZING A THEOREM

sum vs sum_cont

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

Theorem we tried to prove directly:

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

sum vs sum_cont

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

Theorem we tried to prove directly:

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

It didn't work because sum_cont does not call itself recursively using (fun s -> s).

To reason about recursive calls, we need to use the induction hypothesis, but we aren't allowed to here.

sum vs sum_cont

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

Theorem we tried to prove directly:

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

It didn't work because sum_cont does not call itself recursively using (fun s -> s).

To reason about recursive calls, we need to use the induction hypothesis, but we aren't allowed to here.

Need to come up with IH that characterizes recursive calls

sum vs sum_cont

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

Theorem we tried to prove directly:

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

New Theorem Attempt #1:

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

key idea: replace one specific value (the id function in this case) with *all* possible values

sum vs sum_cont

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

Theorem we tried to prove directly:

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

specific

New Theorem Attempt #1:

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

general

key idea: replace one specific value (the id function in this case) with *all* possible values

sum vs sum_cont

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

Theorem we tried to prove directly:

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

New Theorem Attempt #1:

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

But the theorem is false! :-(
counter-example, choose:
k = (fun x -> x + 1)

sum vs sum_cont

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

Theorem we tried to prove directly:

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

New Theorem Attempt #2:

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

Success!

A Possible Proof Strategy

Look at the recursive calls made within your function(s).

- If the arguments (other than the one you are doing induction on) are unchanged, you may have success with simple induction

```
let rec f (l:int list) (x: ...) (y:...) : int =
  match l with
    [] ->  ...
  | hd::tail -> ... (f tail x y)
```

- If they are different, you may have to search for a more general theorem that allows you to conclude something useful about those recursive calls.

```
let rec f (l:int list) (x: ...) (y:...) : int =
  match l with
    [] ->  ...
  | hd::tail -> ... (f tail (complex1) (complex2))
```

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Theorem:

```
for all e:exp,
  eval1 e == eval2 e 0
```

Another Example

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

Theorem:

```
for all e:exp,
  eval1 e == eval2 e 0
```

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
     Int i -> i
   | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Theorem:
for all e:exp,
  eval1 e == eval2 e 0

What is going to go wrong if we try induction on the structure of e directly?

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Theorem:
`for all e:exp,`
`  eval1 e == eval2 e 0`

In the case when e is Add(e1, e2), we will need to reason that `eval2 e1 (eval2 e2 0) == ???` involving eval1

But we won't be able to use IH. We'll have no way to reason about `eval2 e1 (...)` when (...) is not 0.

Another Example

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

Theorem:

```
for all e:exp,
  eval1 e == eval2 e 0
```

Suggestions?

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Theorem:
for all e:exp,
  eval1 e == eval2 e 0

Suggestions?

We will need to reason about **eval2 e1 (...)**
and to relate it to **eval1 e1** somehow.
What is the relationship?

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Strategy: Introduce a new Lemma:

```
for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n
```

we replaced a *specific* value (0) with something more *general* – any integer n!

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Strategy: Introduce a new Lemma:

for all e:exp, for all n:int

  (eval1 e) + n == eval2 e n

Proof: By induction on the structure of e.

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Strategy: Introduce a new Lemma:

```
for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n
```

Proof: By induction on the structure of e.

```
case: e = int i
```

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Strategy: Introduce a new Lemma:

```
for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n
```

Proof: By induction on the structure of e.

```
case: e = int i
   eval1 (Int i) + n(LHS)
```

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

```
Strategy: Introduce a new Lemma:
for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n
Proof: By induction on the structure of e.
case: e = int i
   eval1 (Int i) + n(LHS)
== i + n                         (by eval of eval1)
== eval2 (Int i) n               (by reverse eval of eval2)
```

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Strategy: Introduce a new Lemma:

```
for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n
```

Proof: By induction on the structure of e.

```
case: e = Add(e1, e2)
```

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Strategy: Introduce a new Lemma:

```
for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n
```

Proof: By induction on the structure of e.

```
case: e = Add(e1, e2)
   eval2 (Add(e1, e2)) n          (RHS)
```

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
     Int i -> i
   | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Strategy: Introduce a new Lemma:

```
for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n
```

Proof: By induction on the structure of e.

```
case: e = Add(e1, e2)
   eval2 (Add(e1, e2)) n            (RHS)
== eval2 e1 (eval2 e2 n)            (eval of eval2)
```

Another Example

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

Strategy: Introduce a new Lemma:

```
for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n
```

Proof: By induction on the structure of e.

```
case: e = Add(e1, e2)
   eval2 (Add(e1, e2)) n              (RHS)
== eval2 e1 (eval2 e2 n)              (eval of eval2)
== eval2 e1 (eval1 e2 + n)            (by IH)
```

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
     Int i -> i + n
   | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
     Int i -> i
   | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Strategy: Introduce a new Lemma:

```
for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n
```

Proof: By induction on the structure of e.

```
case: e = Add(e1, e2)
   eval2 (Add(e1, e2)) n           (RHS)
== eval2 e1 (eval2 e2 n)           (eval of eval2)
== eval2 e1 (eval1 e2 + n)         (by IH)
== eval1 e1 + (eval1 e2 + n)       (by IH)
== (eval1 e1 + eval1 e2) + n       (associativity of +)
== eval1 (Add (e1, e2)) + n        (by eval in reverse)
```

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
     Int i -> i
   | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Strategy: Introduce a new Lemma:

```
for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n
```

Proof: By induction on the structure of e.

```
case: e = Add(e1, e2)
   eval2 (Add(e1, e2)) n           (RHS)
== eval2 e1 (eval2 e2 n)           (eval of eval2)
== eval2 e1 (eval1 e2 + n)         (by IH)
== eval1 e1 + (eval1 e2 + n)       (by IH)
== (eval1 e1 + eval1 e2) + n       (associativity of +)
== eval1 (Add (e1, e2)) + n        (by eval in reverse)
```

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Back to the Theorem:

for all e:exp,
  eval1 e == eval2 e 0

Proof:

Lemma:

for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n

Proof: Done!

Another Example

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)
```

Back to the Theorem:

for all e:exp,
  eval1 e == eval2 e 0

Proof:

```
Pick any e.
   eval2 e 0          (RHS)
== eval1 e + 0        (by Lemma, using 0 for n)
== eval1 e            (by math)
```

Lemma:

for all e:exp, for all n:int
  (eval1 e) + n == eval2 e n

Proof: Done!

Quick Question

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

Is eval2 tail recursive?

Quick Question

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

Is eval2 tail recursive?

No! Lot's of stuff happens after the first recursive call to eval2!

Quick Question

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (__________)
```

continuation of **eval2 e2 n**

Quick Question

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (__________)
```

```
let rec eval2 (e:exp) (n:int) (k: int -> int) : int =
  match e with
      Int i -> k (i + n)
    | Add (e1, e2) -> eval2 e2 n (fun m -> eval2 e1 m k)
```

Quick Question

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

```
let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (__________)
```

continuation of **eval2 e1 is whatever eval2 does when it returns**

```
let rec eval2 (e:exp) (n:int) (k: int -> int) : int =
  match e with
      Int i -> k (i + n)
    | Add (e1, e2) -> eval2 e2 n (fun m -> eval2 e1 m k)
```

Summary

Tail-recursive programs:

- do not do any computation after they make a recursive call
- conversion to CPS is one way to make any computation tail-recursive
  - bottle up the stuff you do after the call into a continuation

Proving programs correct can be arbitrarily hard:

- the difficult part comes in finding auxiliary lemmas to prove.
- these lemmas must be:
  - *strong enough* to imply the theorem you want
  - *weak enough* that they remain true and can be proven
  - insight is needed to find the right middle ground

Challenge: CPS Convert the incr function

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

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

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

Challenge: CPS Convert the incr function

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

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

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

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

Challenge: CPS Convert the incr function

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

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

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

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

called *A-Normal Form* (intermediate computations given names; no function calls as args to other function calls)

Challenge: CPS Convert the incr function

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

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

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

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

Challenge: CPS Convert the incr function

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

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

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

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

Challenge: CPS Convert the incr function

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

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

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

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

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

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

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

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

In general

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

Direct Style

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

A-normal Form

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

CPS converted