Continuation-Passing Style

COS 326
David Walker
Princeton University

slides copyright 2017 David Walker
permission granted to reuse these slides for non-commercial educational purposes
Midterm Exam

Wed Oct 25, 2017
In Class (11:00-12:20)
Midterm Week

Be there or be square!
Some Innocuous Code

Let’s try it.

(Go to tail.ml)
Some Other Code

Four functions: Green works on big inputs; Red doesn’t.

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

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

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

let rec sum2 (l:int list) : int =  
  match l with  
    [] -> 0  
  | hd::tail -> hd + sum2 tail
```

```
Four functions: Green works on big inputs; Red doesn’t.

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

let rec sum_to2 (n: int) : int =  
  let rec aux (n:int) (a:int) : int =  
    if n > 0 then  
      aux (n-1) (a+n)  
    else a  
  in  
  aux n 0

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

let rec sum2 (l:int list) : int =  
  match l with  
  [] -> 0  
  | hd::tail -> hd + sum2 tail
```

code that works:

*no computation after recursive function call*
Tail Recursion

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

Not tail-recursive, the substitution model:

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

Not tail-recursive, the substitution model:

```
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
```

```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;;
```
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;;
```
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

```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
```
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
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

```ocaml
code
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 100
Data Needed on Return Saved on Stack

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

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!)
Tail Recursion

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

Tail-recursive:

```ocaml
sum_to2 1000000
```

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

Tail-recursive:

```ocaml
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
```

(* sum of 0..n *)
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
```

```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
;;
```
Tail Recursion

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
```

```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
;;
```
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
;;
```

(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
;;
```

```
stack

aux 10000 0
```
A **tail-recursive function** is a function that does no work after it calls itself recursively.

```ocaml
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)
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
;;
```

```plaintext
aux 9998 19999
```
A **tail-recursive function** is a function that does no work after it calls itself recursively.

(* sum of 0..n *)

```plaintext
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

```plaintext
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
;;
```

stack

```
aux 0 BigNum
```
We used human ingenuity to do the tail-call transform.

Is there a mechanical procedure to transform any recursive function in to a tail-recursive one?

---

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

Can any non-tail-recursive function be transformed into a tail-recursive one? Yes, if we can capture the *differential* between a tail-recursive function and a non-tail-recursive one.

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

Idea: Focus on what happens after the recursive call.
Question

Can any non-tail-recursive function be transformed into a tail-recursive one? Yes, if we can capture the differential between a tail-recursive function and a non-tail-recursive one.

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

Idea: Focus on what happens after the recursive call. Extracting that piece:

```
result of recursive call gets plugged in here
```

How do we capture it?
How do we capture that computation?

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

result of recursive call gets plugged in here
How do we capture that computation?

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

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

```
type cont = int -> int;;
```
How do we capture that computation?

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

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

How do we capture that computation?

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

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 = ??
```

```
type cont = int -> int;;
```

How do we capture that computation?

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

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)
```

(Note: The diagram includes a question and a code snippet demonstrating how to capture a computation in OCaml.)
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]
type \texttt{cont} = \texttt{int} \rightarrow \texttt{int};;

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

let \texttt{sum} (l:int list) : \texttt{int} = \texttt{sum\_cont} \texttt{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));;
type \texttt{cont} = \texttt{int} \rightarrow \texttt{int};

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

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

\[
\begin{align*}
\texttt{sum} \ [1;2] & \rightarrow \\
& \texttt{sum\_cont} \ [1;2] \ (\texttt{fun s -> s}) \\
& \rightarrow \\
& \texttt{sum\_cont} \ [2] \ (\texttt{fun s -> (fun s -> s) (1 + s)})
\end{align*}
\]
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
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 \textit{cont} = \textit{int} -> \textit{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)

\begin{align*}
\text{sum} & \ [1;2] \\
& \rightarrow \sum\_\text{cont} \ [1;2] \ (\text{fun} \ s \rightarrow s) \\
& \rightarrow \sum\_\text{cont} \ [2] \ (\text{fun} \ s \rightarrow (\text{fun} \ s \rightarrow s) \ (1 + s));;
& \rightarrow \sum\_\text{cont} \ [] \ (\text{fun} \ s \rightarrow (\text{fun} \ s \rightarrow (\text{fun} \ s \rightarrow s) \ (1 + s)) \ (2 + s)) \\
& \rightarrow (\text{fun} \ s \rightarrow (\text{fun} \ s \rightarrow (\text{fun} \ s \rightarrow s) \ (1 + s)) \ (2 + s)) \ 0 \\
& \rightarrow (\text{fun} \ s \rightarrow (\text{fun} \ s \rightarrow s) \ (1 + s)) \ (2 + 0)) \\
& \rightarrow (\text{fun} \ s \rightarrow s) \ (1 + (2 + 0))
\end{align*}
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
type `cont = int -> int;;`

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

let `sum` (l:int list) : int = `sum_cont` l (fun s -> s)

`sum` [1;2]
  -->
  `sum_cont` [1;2] (fun s -> s)
  -->
  `sum_cont` [2] (fun s -> (fun s -> s) (1 + s));;
  -->
  `sum_cont` [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
  -->
  ...
  -->
  3

*Where did the stack space go?*
function inside expression

each function is a closure; points to the closure inside it

a stack of closures on the heap

```
sum_cont []
   (fun s3 ->
     (fun s2 ->
       (fun s1 -> s1) (hd1 + s2)
     ) (hd2 + s3)
   )
```
function inside function inside function inside expression

⇒ a stack of closures on the heap

sum_cont []
  (fun s3 ->
   (fun s2 ->
     (fun s1 -> s1) (hd1 + s2)
   ) (hd2 + s3)
  )

stack

heap

(fun s3 ->
  (fun s2 ->
    (fun s1 -> s1) (hd1 + s2)
  ) (hd2 + s3)
)
function inside function inside function inside function inside expression

a stack of closures on the heap

$$\text{sum_cont}$$

$$\text{fun } s \text{ env } \rightarrow\text{ env.k (env.h2 + s)}$$

$$\text{hd2} = 2$$

$$\text{fun } s \text{ env } \rightarrow s$$

$$\text{hd1} = 1$$

$$\text{sum_cont} \ [\ [ \ (\text{fun } s3 \rightarrow\text{ (fun } s2 \rightarrow\text{ (fun } s1 \rightarrow s1) \text{ (hd1 + s2)} \text{ ) (hd2 + s3)} \text{ ) } ] ]$$

stack

heap
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)) ;;
Continuation-passing style

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

fun s env -> s

n = 100
k =
Continuation-passing style

let rec sum_to_cont (n:int) (k:int->int) : int =
    if n > 0 then
        sum_to_cont (n-1) (fun s -> k (n+s))
    else
        k 0

sum_to_cont 100 (fun s -> s)
Continuation-passing style

```ocaml
let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    k 0

sum_to_cont 100 (fun s -> s)
```
Continuation-passing style

let rec sum_to_cont (n:int) (k:int->int) : int =
  if n > 0 then
    sum_to_cont (n-1) (fun s -> k (n+s))
  else
    k 0 ;;

sum_to 100 (fun s -> s)
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;
sum_to 100
but how do you really implement that?

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

sum_to 100
Back to stacks

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;
sum_to 100

but how do you really implement that?

there is two bits of information here:
(1) some state (n=100) we had to remember
(2) some code we have to run later
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;
sum_to 100

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

fun s stack ->
  return (stack.n + s)
sum_to_cont 98 k3

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

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

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

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

n = 100

n = 99

k =

k =

sum_to 98

with the stack

with the heap

state

sum_to 98
\[ \text{sum_to_cont}_{98\, k3} \]

\[ \text{fun } s \text{ env } \rightarrow \text{env} . k \left( \text{env} . n + s \right) \]

\[ n = 99 \]

\[ \text{fun } s \text{ env } \rightarrow \text{env} . k \left( \text{env} . n + s \right) \]

\[ n = 100 \]

\[ \text{fun } s \text{ env } \rightarrow s \]

\[ \text{with the stack} \]

\[ \text{CPS} \]
Why CPS?

Continuation-passing style is *inevitable*.

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

- If you explicitly CPS-convert your code, “*what to do next*” is stored on the heap
- If you don’t, it’s stored on the stack

If you take a conventional compilers class, the continuation will be called a *return address* (but you’ll know what it really is!)

The idea of a *continuation* is much more general!
Your compiler can put all the continuations in the heap so you don’t have to (and you don’t run out of stack space)!

Other pros:

• light-weight concurrent threads

Some cons:

• hardware architectures optimized to use a stack
• need tight integration with a good garbage collector

see

Empirical and Analytic Study of Stack versus Heap Cost for Languages with Closures. Shao & Appel
Call-backs: Another use of continuations

Call-backs:

```plaintext
request_url : url -> (html -> 'a) -> 'a

request_url "http://www.s.com/i.html" (fun html -> process html)
```
Overall Summary

We developed techniques for reasoning about the space costs of functional programs

- the cost of *manipulating data types* like tuples and trees
- the cost of allocating and *using function closures*
- the cost of *tail-recursive* and non-tail-recursive *functions*

We also talked about some important program transformations:

- *closure conversion* makes nested functions with free variables in to pairs of closed code and environment
- the *continuation-passing style* (CPS) transformation turns non-tail-recursive functions in to tail-recursive ones that use no stack space
  - the stack gets moved in to the function closure
- since stack space is often small compared with heap space, it is often necessary to use *continuations and tail recursion*
  - but full CPS-converted programs are unreadable: use judgement
type tree = Leaf | Node of int * tree * tree ;;

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

Hint 1: introduce one let expression for each function call:
        let x = incr left i in ...

Hint 2: you will need two continuations
CPS Convert the incr function

```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) ;;

type cont = tree -> tree ;;

let rec incr_cps (t:tree) (i:int) (k:cont) : tree =
  match t with
  Leaf -> k Leaf
  | Node (j,left,right) -> ...
  ;;
```
type tree = Leaf | Node of int * tree * tree ;;

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

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

first continuation: fun left_done -> Node (i+j, left_done, incr right i)

second continuation: fun right_done -> k (Node (i+j, left_done, right_done))
type tree = Leaf | Node of int * tree * tree;;

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

fun left_done ->
  let k2 =
    (fun right_done ->
      k (Node (i+j, left_done, right_done))
    )
  in
  incr right i k2
type tree = Leaf | Node of int * tree * tree;;

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


let rec incr_cps (t:tree) (i:int) (k:cont) : tree =
    match t with
    Leaf -> k Leaf
    | Node (j,left,right) ->
        let k1 = (fun left_done ->
                let k2 = (fun right_done ->
                        k (Node (i+j, left_done, right_done)))
                in
                incr_cps right i k2
        )
        in
        incr_cps left i k1 ;;

let incr_tail (t:tree) (i:int) : tree = incr_cps t i (fun t -> t);;
CORRECTNESS OF A CPS TRANSFORM
Are the two functions the same?

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)
Need to Generalize the Theorem and IH

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

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

case \( l = [] \)

...  

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

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

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

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

\[
\text{sum\_cont} \ \text{tail} \ (\text{fn} \ s' \rightarrow \text{hd} + s') \) == \text{sum} \ \text{tail}
\]

... but that's not even true
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
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
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)) (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)) (eval, since sum total and and sum tail valuable)
  == k (hd + (sum tail)) (eval sum, reverse)
  == k (sum (hd::tail))

  case done!
  QED!
Ok, now what we have is a proof of this theorem:

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

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

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

So, we've show that the function sum2, which is tail-recursive, is functionally equivalent to the non-tail-recursive function sum.
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:

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

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.

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

This is a common occurrence -- *generalizing the induction hypothesis* -- and it requires human ingenuity. It's why proving theorems is hard. It's also why writing programs is hard -- you have to make the proofs and programs work more generally, around every iteration of a loop.
Overall Summary

We developed techniques for reasoning about the space costs of functional programs

– the cost of *manipulating data types* like tuples and trees
– the cost of allocating and using *function closures*
– the cost of *tail-recursive* and non-tail-recursive *functions*

We also talked about some important program transformations:

– *closure conversion* makes nested functions with free variables into pairs of closed code and environment

– the *continuation-passing style* (CPS) transformation turns non-tail-recursive functions in to tail-recursive ones that use no stack space
  • the stack gets moved in to the function closure

– since stack space is often small compared with heap space, it is often necessary to use *continuations and tail recursion*
  • but full CPS-converted programs are unreadable: use judgement