A Functional Space Model

COS 326
David Walker
Princeton University
Understanding the space complexity of functional programs

- At least two interesting components:
  - the amount of *live space* at any instant in time
  - the *rate of allocation*
    - a function call may not change the amount of live space by much but may allocate at a substantial rate
    - because functional programs act by generating new data structures and discarding old ones, they often allocate a lot
      » OCaml garbage collector is optimized with this in mind
      » **interesting fact:** at the assembly level, the number of writes by a functional program is roughly the same as the number of writes by an imperative program
Understanding the space complexity of functional programs

- At least two interesting components:
  - the amount of *live space* at any instant in time
  - the *rate of allocation*
    - a function call may not change the amount of live space by much but may allocate at a substantial rate
    - because functional programs act by generating new data structures and discarding old ones, they often allocate a lot
      » OCaml garbage collector is optimized with this in mind
      » *interesting fact*: at the assembly level, the number of writes by a function program is roughly the same as the number of writes by an imperative program

- *What takes up space?*
  - conventional first-order data: tuples, lists, strings, datatypes
  - function representations (closures)
  - the call stack
CONVENTIONAL DATA
Numbers

Tuples

Data types

Lists
Data type representations:

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

Leaf:

```
0
```

Node(i, left, right):

```
Node

3 | left | right
```
In C, you allocate when you call “malloc”

In Java, you allocate when you call “new”

What about ML?
Whenever you *use a constructor*, space is allocated:

```ocaml
define (t:tree) (i:int) =
  match t with
  Leaf -> Node (i, Leaf, Leaf)
| Node (j, left, right) ->
  if i <= j then
    Node (j, insert left i, right)
  else
    Node (j, left, insert right i)
```

Allocating space
Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
  | Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
    if i <= j then
      Node (j, insert left i, right)
    else
      Node (j, left, insert right i)
```

Consider:
```
insert t 21
```
Whenever you use a constructor, space is allocated:

```ocaml
let rec insert (t:tree) (i:int) = 
    match t with 
    Leaf -> Node (i, Leaf, Leaf) 
  | Node (j, left, right) -> 
    if i <= j then 
      Node (j, insert left i, right)
    else
      Node (j, left, insert right i)
```

Consider:

```
insert t 21
```
Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
    if i <= j then
      Node (j, insert left i, right)
    else
      Node (j, left, insert right i)
```

Consider:

```
insert t 21
```
Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
  Leaf -> Node (i, Leaf, Leaf)
| Node (j, left, right) ->
  if i <= j then
    Node (j, insert left i, right)
  else
    Node (j, left, insert right i)
```

Consider:

```
insert t 21
```
Whenever you use a constructor, space is allocated:

let rec insert (t:tree) (i:int) =
  match t with
  Leaf -> Node (i, Leaf, Leaf)
| Node (j, left, right) ->
  if i <= j then
    Node (j, insert left i, right)
  else
    Node (j, left, insert right i)

Consider:

insert t 21
Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
  | Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
    if i <= j then
      Node (j, insert left i, right)
    else
      Node (j, left, insert right i)
```

Consider:

```
insert t 21
```
Whenever you use a constructor, space is allocated:

```
let rec insert (t:tree) (i:int) =
  match t with
    Leaf -> Node (i, Leaf, Leaf)
  | Node (j, left, right) ->
    if i <= j then
      Node (j, insert left i, right)
    else
      Node (j, left, insert right i)
```

Total space allocated is proportional to the height of the tree.

$\sim \log n$, if tree with $n$ nodes is balanced.
Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

let fiddle (t: tree) =
  insert t 21

John McCarthy invented g.c. 1960
The garbage collector reclaims unreachable data structures on the heap.

let fiddle (t: tree) = insert t 21

If t is dead (unreachable),
The garbage collector reclaims unreachable data structures on the heap.

```haskell
let fiddle (t: tree) =
  insert t 21
```

If `t` is dead (unreachable),

Then all these nodes will be reclaimed!
The garbage collector reclaims unreachable data structures on the heap.

let fiddle (t: tree) = insert t 21

Net new space allocated: 1 node

(just like “imperative” version of binary search trees)
But what if you want to keep the old tree?

```haskell
let faddle (t: tree) = (t, insert t 21)
```
Net space allocated

But what if you want to keep the old tree?

```haskell
let faddle (t: tree) = (t, insert t 21)
```

Net new space allocated: $\log(N)$ nodes

but note: “imperative” version would have to copy the old tree, space cost $N$ new nodes!
let check_option (o:int option) : int option =
  match o with
    Some _ -> o
  | None -> failwith "found none"
;;

let check_option (o:int option) : int option =
  match o with
    Some j -> Some j
  | None -> failwith "found none"
;;
let check_option (o:int option) : int option =
  match o with
  | Some _ -> o
  | None -> failwith "found none"
;;

let check_option (o:int option) : int option =
  match o with
  | Some j -> Some j
  | None -> failwith "found none"
;;
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
;;

let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
;;

let double (c1:int*int) : int*int =
  cadd c1 c1
;;

let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
;;
```ocaml
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
;;

let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
;;

let double (c1:int*int) : int*int =
  cadd c1 c1
;;

let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
;;
```
let cadd (c1:int*int) (c2:int*int) : int*int = 
let (x1,y1) = c1 in 
let (x2,y2) = c2 in 
(x1+x2, y1+y2) ;;

let double (c1:int*int) : int*int = 
let c2 = c1 in 
cadd c1 c2 ;;

let double (c1:int*int) : int*int = 
cadd c1 c1 ;;

let double (c1:int*int) : int*int = 
let (x1,y1) = c1 in 
cadd (x1,y1) (x1,y1) ;;
let cadd (c1:int*int) (c2:int*int) : int*int =
  let (x1,y1) = c1 in
  let (x2,y2) = c2 in
  (x1+x2, y1+y2)
;;

let double (c1:int*int) : int*int =
  let c2 = c1 in
  cadd c1 c2
;;

let double (c1:int*int) : int*int =
  cadd c1 c1
;;

let double (c1:int*int) : int*int =
  let (x1,y1) = c1 in
  cadd (x1,y1) (x1,y1)
;;
let cadd (c1:int*int) (c2:int*int) : int*int =
let (x1,y1) = c1 in
let (x2,y2) = c2 in
(x1+x2, y1+y2)
;;

let double (c1:int*int) : int*int =
let c2 = c1 in
cadd c1 c2
;;

let double (c1:int*int) : int*int =
cadd c1 c1
;;

let double (c1:int*int) : int*int =
let (x1,y1) = c1 in
cadd (x1,y1) (x1,y1)
;;

no allocation

no allocation

allocates 2 pairs
(unless the compiler happens to optimize...)
let double (c1:int*int) : int*int =
let (x1,y1) = c1 in
let (x2,y2) = c2 in
(x1+x2, y1+y2);

let cadd (c1:int*int) (c2:int*int) : int*int =
let (x1,y1) = c1 in
let (x2,y2) = c2 in
(x1+x2, y1+y2);

double does not allocate

extracts components: it is a read
FUNCTION CLOSURES
Consider the following program:

```ocaml
let choose (arg : bool * int * int) : int -> int =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
;;

choose (true, 1, 2);;
```
Consider the following program:

```ocaml
let choose (arg:bool * int * int) : int -> int =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
;;
choose (true, 1, 2);;
```

Its execution behavior according to the substitution model:

```ocaml
choose (true, 1, 2)
```
Consider the following program:

```
let choose (arg:bool * int * int) : int -> int =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
;;
choose (true, 1, 2);;
```

Its execution behavior according to the substitution model:

```
choose (true, 1, 2)
-->
let (b, x, y) = (true, 1, 2) in
  if b then (fun n -> n + x)
  else (fun n -> n + y)
```
Consider the following program:

```
let choose (arg: bool * int * int) : int -> int =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
;;
choose (true, 1, 2);;
```

Its execution behavior according to the substitution model:

```
choose (true, 1, 2)
-->
  let (b, x, y) = (true, 1, 2) in
  if b then (fun n -> n + x)
  else (fun n -> n + y)
-->
  if true then (fun n -> n + 1)
  else (fun n -> n + 2)
```
Closures

Consider the following program:

```ml
let choose (arg: bool * int * int) : int -> int =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
;;

choose (true, 1, 2);;
```

Its execution behavior according to the substitution model:

```ml
choose (true, 1, 2)
-->
  let (b, x, y) = (true, 1, 2) in
  if b then (fun n -> n + x)
  else (fun n -> n + y)
-->
  if true then (fun n -> n + 1)
  else (fun n -> n + 2)
-->
  (fun n -> n + 1)
```
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
;;

choose (true, 1, 2);;
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
;;
choose (true, 1, 2);;

choose:
  mov rb r_arg[0]
  mov rx r_arg[4]
  mov ry r_arg[8]
  compare rb 0
  ...
  jmp ret
main:
  ...
  jmp choose
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
;;
choose (true, 1, 2);;

choose:
  mov rb r_arg[0]
  mov rx r_arg[4]
  mov ry r_arg[8]
  compare rb 0
  ...
  jmp ret
main:
  ...
  jmp choose
let choose arg =
    let (b, x, y) = arg in
    if b then
        (fun n -> n + x)
    else
        (fun n -> n + y)
;;
choose (true, 1, 2);;

choose:mov rb r_arg[0]
mov rx r_arg[4]
mov ry r_arg[8]
compare rb 0
...
jmp ret
main:
...
jmp choose

execute with substitution

let (b, x, y) = (true, 1, 2) in
if b then
    (fun n -> n + x)
else
    (fun n -> n + y)

execute with substitution

==
generate new code block with parameters replaced by arguments
let choose arg =
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x)
  else
    (fun n -> n + y)
;;
choose (true, 1, 2);;

choose:  
  mov rb r_arg[0]
  mov rx r_arg[4]
  mov ry r_arg[8]
  compare rb 0
  ...
  jmp ret
main:
  ...
  jmp choose

let (b, x, y) = (true, 1, 2) in
if b then
  (fun n -> n + x)
else
  (fun n -> n + y)

choose_subst:
  mov rb 0xF8[0]
  mov rx 0xF8[4]
  mov ry 0xF8[8]
  ...
  jmp ret
  compare rb 0
  ...
main:
  ...
  jmp choose

0xF8: 0
     1
     2
let choose arg =
    let (b, x, y) = arg in
    if b then
        (fun n -> n + x)
    else
        (fun n -> n + y)
;;
choose (true, 1, 2);;

let (b, x, y) = (true, 1, 2) in
if b then
    (fun n -> n + x)
else
    (fun n -> n + y)
;;
choose (true, 1, 2);;

if true then
    (fun n -> n + 1)
else
    (fun n -> n + 2)
What we aren’t going to do

The substitution model of evaluation is *just a model*. It says that we generate new code at each step of a computation. We don’t do that in reality. Too expensive!

The substitution model is a faithful model for reasoning about the relationship between inputs and outputs of a function but it doesn’t tell us much about the resources that are used along the way.

I’m going to tell you a little bit about how ML programs are compiled so you can understand how much space your programs will use. Understanding the space consumption of your programs is an important component in making these programs more efficient.
General tactic: Reduce the problem of compiling ML-like functions to the problem of compiling C-like functions.

Some functions are already C-like:

```ocaml
define add (x:int*int) : int =
  let (y,z) = x in
  y + z
;;
```

```assembly
    # argument in r1
    # return address in r0
    add r4, r2, r3  # sum in r4
    jmp r0
```

```assembly
    ld r2, r1[0]  # y in r2
    ld r3, r1[4]  # z in r3
```
But what about nested, higher-order functions?

```ocaml
let choose arg =  
  let (b, x, y) = arg in  
  if b then  
    (fun n -> n + x)  
  else  
    (fun n -> n + y)  
;;
```

```ocaml
let f1 n = n + x;;
let f2 n = n + y;;
```
But what about nested, higher-order functions?

Given:

```ml
let choose arg = let (b, x, y) = arg in if b then (fun n -> n + x) else (fun n -> n + y) ;;
```

Let's simplify:

```ml
let f1 n = n + x ;;
let f2 n = n + y ;;
```

But what about nested, higher-order functions?

```ml
let choose arg = let (b, x, y) = arg in if b then f1 else f2 ;;
```

Darn! *Doesn’t work naively.* Nested functions contain **free variables**. Simple unnesting leaves them undefined.
But what about nested, higher-order functions?

We can’t execute a function like the following:

```plaintext
let f2 n = n + y;;
```

But we can execute a *closure* which is a pair of some code and an environment:

```plaintext
let f2 (n, env) = n + env.y;;
{y = 1}
```
Closure Conversion

Closure conversion (also called lambda lifting) converts open, nested functions into closed, top-level functions.

```ocaml
let choose arg = 
  let (b, x, y) = arg in 
  if b then 
    (fun n -> n + x + y)
  else 
    (fun n -> n + y)
;;
```
Closure conversion (also called lambda lifting) converts open, nested functions into closed, top-level functions.

```ocaml
let choose arg = let (b, x, y) = arg in if b then (fun n -> n + x + y) else (fun n -> n + y);;

let choose (arg, env) = let (b, x, y) = arg in if b then (f1, {xe=x; ye=y}) else (f2, {ye=y});;

let f1 (n, env) = n + env.xe + env.ye;;

let f2 (n, env) = n + env.ye;;
```

- **Closure Conversion**
  - Add environment parameter
  - Create closures
  - Use environment variables instead of free variables
Closure conversion converts open, nested functions into closed, top-level functions.

```ocaml
let choose arg = 
  let (b, x, y) = arg in
  if b then
    (fun n -> n + x + y)
  else
    (fun n -> n + y)

let choose (arg, env) = 
  let (b, x, y) = arg in
  if b then
    (f1, {xe=x; ye=y})
  else
    (f2, {ye=y})

let f1 (n, env) = 
  n + env.xe + env.ye

let f2 (n, env) = 
  n + env.ye

(choose (true, 1, 2)) 3
```

```ocaml
let c_closure = (choose, ())               in (* create closure *)
let (c_code, c_cenv) = c_closure in (* extract code, env *)
let f_closure = c_code ((true, 1, 2), c_env) in (* call choose code, extract f code, env *)
let (f_code, f_env) = f_closure in (* extract code, env *)
let f_code (3, f_env) = f_closure in (* call f code *)
```
Closure conversion converts open, nested functions into closed, top-level functions.

```ocaml
let choose arg = let (b, x, y) = arg in if b then (fun n -> n + x + y) else (fun n -> n + y);;

let choose (arg, env) = let (b, x, y) = arg in if b then (f1, {xe=x; ye=y}) else (f2, {ye=y});;

let f1 (n, env) = n + env.xe + env.ye;;
let f2 (n, env) = n + env.ye;;

(choose (true, 1, 2)) 3
```

```ocaml
let c_closure = (choose, ()) in (* create closure *)
let (c_code, c_cenv) = c_closure in (* extract code, env *)
let f_closure = c_code ((true, 1, 2), c_cenv) in (* call choose code, extract f code, env *)
let (f_code, f_env) = f_closure in (* extract code, env *)
f_code (3, f_env) in (* call f code *)
```
Closure conversion converts open, nested functions into closed, top-level functions.

```ocaml
let choose arg = let (b, x, y) = arg in
  if b then
    (fun n -> n + x + y)
  else
    (fun n -> n + y);

let choose (arg, env) = let (b, x, y) = arg in
  if b then
    (f1, {xe=x; ye=y})
  else
    (f2, {ye=y});

let f1 (n, env) = n + env.xe + env.ye;
let f2 (n, env) = n + env.ye;

(choose (true, 1, 2)) 3
```

```
let c_closure = (choose, ());
let (c_code, c_cenv) = c_closure
let f_closure = c_code ((true, 1, 2), c_cenv)
let (f_code, f_env) = f_closure
f_code (3, f_env)
```
Closure conversion converts open, nested functions into closed, top-level functions.

```
let choose arg =
    let (b, x, y) = arg in
    if b then
        (fun n -> n + x + y)
    else
        (fun n -> n + y)
;;

let choose (arg, env) =
    let (b, x, y) = arg in
    if b then
        (f1, {xe=x; ye=y})
    else
        (f2, {ye=y})
;;

let f1 (n, env) =
    n + env.xe + env.ye
;;

let f2 (n, env) =
    n + env.ye
;;

(choose (true, 1, 2)) 3
```

```
let c_closure = (choose, ()) in (* create closure *)
let (c_code, c_env) = c_closure in (* extract code, env *)
let f_closure = c_code ((true, 1, 2), c_env) in (* call choose code, extract f code, env *)
let (f_code, f_env) = f_closure in (* extract code, env *)
f_code (3, f_env) (* call f code *)
```
Even though the original, non-closure-converted code was well-typed, the closure-converted code isn’t—because the environments are different.

```
let choose (arg, env) = 
    let (b, x, y) = arg in 
    if b then 
        (f1, F1 {xe=x; ye=y}) 
    else 
        (f2, F2 {ye=y}) 
;;

let f1 (n, env) = 
    n + env.xe + env.ye 
;;

let f2 (n, env) = 
    n + env.ye 
;;
```

```
type f1_env = {x1:int; y1:int} 
type f1_clos = (int * f1_env -> int) * f1_env 

type f2_env = {y2:int} 
type f2_clos = (int * f2_env -> int) * f2_env 
```
Even though the original, non-closure-converted code was well-typed, the closure-converted code isn’t because the environments are different.

```ml
let choose (arg,env) = 
  let (b, x, y) = arg in 
  if b then 
    (f1, F1 {xe=x; ye=y}) 
  else 
    (f2, F2 {ye=y}) 
;;

let f1 (n,env) = 
  n + env.xe + env.ye 
;;

let f2 (n,env) = 
  n + env.ye 
;;
```

Solution 0:  Don’t bother to typecheck after closure conversion.

After all, the source program was well typed (checked by the source-language ML typechecker), and the compiler (with its closure conversion algorithm) cannot possibly have produced a program with the wrong behavior.

That is, consider the post-closure-converted language to be an untyped language.

This is the traditional solution, and it’s not stupid. But can we do better?
One Extra Note: Typing

Even though the original, non-closure-converted code was well-typed, the closure-converted code isn’t because the environments are different

```ocaml
define choose (arg, env) =
define (b, x, y) = arg in
  if b then
    (f1, F1 {x1=x; y2=y})
  else
    (f2, F2 {y2=y})

let f1 (n, env) =
define match env with
  F1 e -> n + e.x1 + e.y2
  | F2 _ -> failwith "bad env!"

let f2 (n, env) =
define match env with
  F1 _ -> failwith "bad env!"
  | F2 e -> n + e.y2

let f1_env = {x1:int; y1:int} type f1_clos = (int * f1_env -> int) * f1_env
let f2_env = {y2:int} type f2_clos = (int * f2_env -> int) * f2_env

let env = F1 of f1_env | F2 of f2_env
let f1_clos = (int * env -> int) * env
let f2_clos = (int * env -> int) * env
```
Even though the original, non-closure-converted code was well-typed, the closure-converted code isn’t because the environments are different

```ocaml
let choose (arg, env) = 
  let (b, x, y) = arg in 
  if b then 
    (f1, {xe=x; ye=y}) 
  else 
    (f2, {ye=y}) ;;

let f1 (n, env) = 
  n + env.xe + env.ye 
;;

let f2 (n, env) = 
  n + env.ye 
;;

type f1_env = {xe:int; ye:int} 
type f1_clos = (int * f1_env -> int) * f1_env 

type f2_env = {xe:int} 
type f2_clos = (int * f2_env -> int) * f2_env 
```

fix II:
One Extra Note: Typing

Even though the original, non-closure-converted code was well-typed, the closure-converted code isn’t because the environments are different.

```
let choose (arg, env) =
  let (b, x, y) = arg in
  if b then
    (f1, {xe=x; ye=y})
  else
    (f2, {ye=y})

let f1 (n, env) =
  n + env.xe + env.ye

let f2 (n, env) =
  n + env.ye
```

```
type f1_env = {xe:int; ye:int}
type f1_clos = (int * f1_env -> int) * f1_env

type f2_env = {xe:int}
type f2_clos = (int * f2_env -> int) * f2_env
```

```
fix II:

type f1_env = {xe:int; ye:int}
type f2_env = {xe:int}
type f1_clos = env.(int * env -> int) * env
type f2_clos = env.(int * env -> int) * env
```

“From System F to Typed Assembly Language,”
-- Morrisett, Walker et al.
Aside: Existential Types

map has a *universal* polymorphic type:

\[
\text{map : ('a -> 'b) -> 'a list -> 'b list} \quad \text{"for all types 'a and for all types 'b, ..."}
\]

when we closure-convert a function that has type \(\text{int} \to \text{int}\), we get a function with *existential* polymorphic type:

\[
\exists \text{'a. ((int * 'a) -> int) * 'a} \quad \text{"there exists some type 'a such that, ..."}
\]

In OCaml, we can approximate existential types using datatypes (a data type allows you to say "there exists a type 'a drawn from one of the following finite number of options." In Haskell, you've got the real thing.
**Closure Conversion: Summary**

*(before)*

All function definitions equipped with extra env parameter:

```
let f arg = ...
```

*(after)*

```
let f_code (arg, env) = ...  
```

All free variables obtained from parameters or environment:

```
x
```

```
env.cx
```

All functions values paired with environment:

```
f
```

```
(f_code, {cx1=v1; ...; cxn=vn})
```

All function calls extract code and environment and call code:

```
f e
```

```
let (f_code, f_env) = f in
f_code (e, f_env)
```
The space cost of a closure
= the cost of the pair of code and environment pointers (2 words)
+ the cost of the data referred to by function free variables
  (1 word for each free variable)
Assignment #4

An environment-based interpreter:

• Instead of substitution, build up environment.
  • just a list of variable-value pairs

• When you reach a free variable, look in environment for its value.

• To evaluate a recursive function, create a closure data structure
  • pair current environment with recursive code

• To evaluate a function call, extract environment and code from closure, pass environment and argument to code
TAIL CALLS AND CONTINUATIONS
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
        n + sum_to (n-1)
    else 0
in
    sum_to

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
```

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

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

Not tail-recursive, the substitution model:

```plaintext
sum_to 1000000

(* sum of 0..n *)

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

let big_int = 1000000;;

sum big_int;;
```
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
```

```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:

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

let big_int = 1000000;
sum big_int;;
```

expression size grows at every recursive call ...

lots of adding to do after the call returns”
Tail Recursion

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

Not tail-recursive, the substitution model:

```
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:

\[
\text{sum\_to 1000000}
\rightarrow
1000000 + \text{sum\_to 99999}
\rightarrow
1000000 + 99999 + \text{sum\_to 99998}
\rightarrow
\ldots
\rightarrow
1000000 + 99999 + 99998 + \ldots + \text{sum\_to 0}
\rightarrow
1000000 + 99999 + 99998 + \ldots + 0
\]

(* sum of 0..n *)

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

let big_int = 1000000;;

sum big_int;;

recursion finally bottoms out
Tail Recursion

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

Not tail-recursive, the substitution model:

```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

```plaintext
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

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

sum_to 10000
```

Stack:
```
0
.
.
.
9998 +
9999 +
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
```
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
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:***

```plaintext
sum_to2 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
;;
```
A **tail-recursive function** is a function that does no work after it calls itself recursively.

Tail-recursive:

```
sum_to2 1000000
```

```
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
```

```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
```

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

(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.

\[
\begin{align*}
(* \text{ sum of } 0..n *) \\
\text{let sum_to2 } (n: \text{ int}) : \text{ int } = \\
\quad \text{let rec aux } (n: \text{ int})(a: \text{ int}) : \text{ int } = \\
\quad \quad \text{if } n > 0 \text{ then} \\
\quad \quad \quad \text{aux } (n-1) \ (a+n) \\
\quad \quad \text{else } a \\
\quad \text{in} \\
\quad \text{aux } n \ 0 \\
;;
\end{align*}
\]
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
;;

aux 9999 10000
```

stack

aux 9999 10000
A **tail-recursive function** is a function that does no work after it calls itself recursively.

```haskell
(* 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

```plaintext
aux 9998 19999
```
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

```plaintext
aux 9997 29998
```
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

```
av0 BigNum
```
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?

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

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

not only is sum2 tail-recursive but it reimplements an algorithm that took linear space (on the stack) using an algorithm that executes in constant space!
CONTINUATION-PASSING STYLE CPS!
CPS:

- 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, if we can capture the differential between a tail-recursive function and a non-tail-recursive one.

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

Extracting that piece:

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

```haskell
define a function `fun`:
```
Question

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?

```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))

type cont = int -> int;
```

Question
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 = ??
```
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)
```

**Type**

```ocaml
type cont = int -> int
```
type \texttt{cont} = \texttt{int} \rightarrow \texttt{int};;

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

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

\texttt{sum} [1;2]
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)
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} \to \texttt{int};;

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

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

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

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

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

\texttt{sum \ [1;2]}
\to
\texttt{sum\_cont \ [1;2] (fun s -> s)}
\to
\texttt{sum\_cont \ [2] (fun s -> (fun s -> s) (1 + s))};;
\to
\texttt{sum\_cont \ [ ] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))}
\to
(fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s)) 0
\to
(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)

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))
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 \textit{cont} = \textit{int} \rightarrow \textit{int};;

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

\text{sum} \ [1;2]
\rightarrow
\text{sum\_cont} \ [1;2] \ (\text{fun} \ \textit{s} \rightarrow \ \textit{s})
\rightarrow
\text{sum\_cont} \ [2] \ (\text{fun} \ \textit{s} \rightarrow \ (\text{fun} \ \textit{s} \rightarrow \ \textit{s}) \ (1 + \textit{s})) ;;
\rightarrow
\text{sum\_cont} \ [] \ (\text{fun} \ \textit{s} \rightarrow \ (\text{fun} \ \textit{s} \rightarrow \ (\text{fun} \ \textit{s} \rightarrow \ \textit{s}) \ (1 + \textit{s})) \ (2 + \textit{s}))
\rightarrow
\ldots
\rightarrow
3

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

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

a stack of closures on the heap
function inside function inside function inside function inside expression

a stack of closures on the heap

1 2

stack

sum_cont

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)
  )

heap

1

2

func'ion inside func'ion inside func'ion inside expression

expression

a stack of closures on the heap
Continuation-passing style

```ocaml
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)) ;;
```

![Diagram of stack and heap with continuation passing style](image)
Continuation-passing style

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

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

\[
\text{sum_to_cont \ k2}
\]

\[
\begin{array}{c}
n = 100 \\
k = \end{array}
\]
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)
```
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 100 (fun s -> s)
```

Diagram:
- Stack:
  - `sum_to_cont 98 k3`
  - `fun s env -> env.k (env.n + s)`
    - `n = 99`
    - `k =`
  - `fun s env -> env.k (env.n + s)`
    - `n = 100`
    - `k =`
  - `fun s env -> s`

- Heap:
  - `n = 100`
  - `k =`
  - `fun s env -> s`
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?
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
Back to stacks

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

with reality added

code we have to run next

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

k =

n = 100

k =

fun s env ->
  s

with the stack

with the heap
n = 100

return_address

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

state

sum_to 98

with the stack

sum_to_cont 98 k3

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

n = 99

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

k =

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

n = 100

k =

fun s env -> s

with the heap
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:

```ocaml
let request_url url = (fun html -> process html) (get_url url)

let request_url http://www.stuff.com/i.html =
  (fun html -> process html)
```

continuation
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*
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
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)
;;
```

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

Hint 2: you will need two continuations
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:

```
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
  ;;
```

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

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

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

case \( l = [] \)

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

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 (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!
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))
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 + 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)

  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 + x)) (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 + x))
      (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))
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 + x))  (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))
== k (sum (hd:tail))

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

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

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

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

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
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, *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
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) ;;

(see solution after the next slide)
Solution:

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 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)
;;

first continuation: Node (i+j, ___________ , incr right i)

second continuation: Node (i+j, left_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 i left, incr i right) ;;

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

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) ;;

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) ->
        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) ;;