A Functional Space Model

COS 326
David Walker
Princeton University
Data type representations:

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

Leaf: 0

Node(i, left, right):

```
Node
```

```
3  left  right
```
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 at a great rate
      - OCaml garbage collector is optimized with this in mind
      - interesting fact: at the assembly level, the number of writes made by a function program is typically roughly the same as the number of writes by an imperative program
This Time

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 at a great rate
      » OCaml garbage collector is optimized with this in mind
      » interesting fact: at the assembly level, the number of writes made by a function program is typically 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
Allocating space

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)
```
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:

```ocaml```

```
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:

```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:

```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:

```ocaml
insert t 21
```
Allocating space

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
```
Allocating space

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
```
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)
```

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

\( \sim \log n \), if tree with \( n \) nodes is balanced
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"
;;

allocates nothing when arg is Some i

allocates an option when arg is Some i
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)
;;
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)
;;

\[\textit{cadd} \text{ allocates double does not}\]
\[\textit{cadd} \text{ allocates double does not}\]
\[\textit{cadd} \text{ allocates double allocates 2 pairs}\]
let cadd (c1:int*int) (c2:int*int) : int*int =
let (x1,y1) = c1 in
let (x2,y2) = c2 in
(x1+x2, y1+y2)
;;

cadd allocates
double does not

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

extracts components; does not allocate
FUNCTION CLOSURES
Consider the following program:

```ocaml
closures

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);;
```
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);;
```

It’s execution behavior according to the substitution model:

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

```ocaml
define 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);;
```

It’s execution behavior according to the substitution model:

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

It’s 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)
```
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);;
```

It’s execution behavior according to the substitution model:

```ocaml
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);;

execute with substitution

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

```plaintext
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);;
```

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

execute with substitution
```plaintext
==
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 (true, 1, 2);;

execute with substitution
let (b, x, y) = (true, 1, 2) in
if b then
  (fun n -> n + x)
else
  (fun n -> n + y)
;;
Substitution and Compiled Code

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);;

execute with substitution

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

compile

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

==
generate new code block with parameters replaced by arguments

choose_subst:
    mov rb 0xF8[0]
    mov rx 0xFF44
    mov ry 0xFF84[8]
    compare rb 0
    ...
    jmp ret

execute with substitution

choose_subst2:
    compare 1 0
    ...
    jmp ret

main:
    ...
    jmp choose

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 program correctness but it doesn’t help us understand what is going on at the machine-code level

– that’s a good thing! abstraction!!

– you should almost never think about machine code when writing a program. We invented high-level programming languages so you don’t have to.

Still, we need to have a more faithful space model in order to understand how to write efficient algorithms.
Some functions are easy to implement

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

# argument in r1
# return address in r0
add:
  ld r2, r1[0]     # y in r2
  ld r3, r1[4]     # z in r3
  add r4, r2, r3   # sum in r4
  jmp r0

If no functions in ML were nested then compiling ML would be just like compiling C. (Take COS 320 to find out how to do that...)
How do we implement functions?

Let’s remove the nesting and compile them like we compile C.

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

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

```ocaml
let f1 n = n + x;;
```

```ocaml
let f2 n = n + y;;
```
How do we implement functions?

Let’s remove the nesting and compile them like we compile C.

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

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

let f1 n = n + x;;

let f2 n = n + y;;

Darn! *Doesn’t work naively*. Nested functions contain *free variables*. Simple unnesting leaves them undefined.
How do we implement functions?

We can’t define a function like the following using code alone:

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

A **closure** is a pair of some code and an environment:

- Code:
  ```ml
  let f2 (n,env) = 
    n + env.y 
  ;;
  ```
- Environment:
  ```{y = 1}```

A closure is defined as a pair of code and environment: 

code 

environment 

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

```plaintext
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**
  - Create closures
  - Use environment variables instead of free variables
  - Add environment parameter
Closure conversion (also called lambda lifting) converts open, nested functions into closed, top-level functions.

```ml
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_cenv) = c_closure in (* extract code, env *)
let (f_code, f_env) = c_code ((true, 1, 2), c_cenv) in (* call choose code, extract f code, env *)
  (* call f code *)

let f_code (3, f_env) ;;
```
Closure conversion (also called lambda lifting) converts open, nested functions into closed, top-level functions.

```ml
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_env) = c_closure in (* create closure *) let (f_code, f_env) = c_code ((true,1,2), c_env) in (* call choose code, extract f code, env *) (* call f code *)
```
Closure conversion (also called lambda lifting) converts open, nested functions into closed, top-level functions.

```ml
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, ())  (* create closure *)
let (c_code, c_cenv) = c_closure  (* extract code, env *)
let (f_code, f_env) = c_code ((true,1,2), c_env)  (* call choose code, extract f code, env *)
  f_code (3, f_env)  (* call f code *)
;;
```
Closure conversion (also called lambda lifting) 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)
;;

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

```
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
;;

let c_closure = (choose, ()) in (* create closure *)
let (c_code, c_cenv) = c_closure in (* extract code, env *)
let (f_code, f_env) = c_code ((true,1,2), c_cenv) in (* call choose code, extract f code, env *)
  f_code (3, f_env) in (* call f code *).
```
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
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
```
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

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

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

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

type f1_env = {x1:int; y1:int} 

type f1_clos = (int * f1_env -> int) * f1_env

type f1_clos = (int * f1_env -> int) * f1_env

type f2_env = {y2:int}

type f2_clos = (int * f2_env -> int) * f2_env

fix I:
  type env = F1 of f1_env | F2 of f2_env
  type f1_clos = (int * env -> int) * env
  type f2_clos = (int * env -> int) * env
```
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
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:**

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

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

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

```
fix II:
```
```
Aside: Existential Types

map has a *universal* polymorphic type:

\[
\text{map : ('a -> 'b) -> 'a list -> 'b list}
\]

"for all types 'a and for all types 'b, ..."

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

\[
\text{exists 'a. ((int * 'a) -> int) * 'a}
\]

"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

All function definitions equipped with extra env parameter:

(before)

let f arg = ...

(after)

let f_code (arg, env) = ...

All free variables obtained from environment:

x

env.cx

All functions values paired with environment:

f

(f_code, {v1=v1; ...; vn=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
+ the cost of the data referred to by function free variables
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 =
  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
;;
```
Four functions: Green works on big inputs; Red doesn’t.

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

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

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
;;
```

code that works: no computation after recursive function call
A *tail-recursive function* is a function that does no work after it calls itself recursively.

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

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

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

Tail-recursive:

```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
;;
```

(sum of 0..n)

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

(addition overflow occurred at some point)

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

**Not tail-recursive:**

```plaintext
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 *)

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

let big_int = 1000000;;

sum big_int;;
```

expression grows at every recursive call
Memory is partitioned: Stack and Heap

heap space (big!)

stack space (small!)
Data Needed on Return Saved on Stack

sum_to 1000000
--> ...
--> 1000000 + 99999 + 99998 + 99997 + ... + ...
--> ...

every non-tail call puts the data from the calling context on the stack

not much space left! will run out soon!

the stack
Can any non-tail-recursive function be transformed into a tail-recursive one?

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

```ocaml
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*!
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.

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

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

How do we capture it?
Question

How do we capture that computation?

```
hd + [ ]
```

```
fun s -> hd + [s]
```
How do we capture that computation?

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

define 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 -> ???) ;;
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 -> k (hd + s)) ;;

fun s -> hd + s
How do we capture that computation?

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

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 = ??
```
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 `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 cont = int \rightarrow int;;

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

let sum (l:int list) : int = sum_cont l (fun s \rightarrow s)

sum [1;2]
\rightarrow
  sum_cont [1;2] (fun s \rightarrow 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 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))
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
type cont = int -> int;;

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

let sum (l:int list) : int = sum_cont l (fun s -> s)
type cont = int -> int;;

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

let sum (l:int list) : int = sum_cont l (fun s -> s)
type cont = int -> int;;

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

let sum (l:int list) : int = sum_cont l (fun s -> s)
type 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?
CPS:

– short for *Continuation-Passing Style*
– Every function takes a continuation as an argument that expresses "what to do next"
– CPS functions only call other functions as the last thing they do
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:

\[
\text{type } \text{cont} = \text{int} \rightarrow \text{int};;
\]

\[
\text{let rec sum_cont } (l:\text{int list}) \ (k:\text{cont}) : \text{int} = \\
\text{match } l \text{ with} \\
\quad [] \rightarrow k \ 0 \\
\quad \text{hd}::\text{tail} \rightarrow \text{sum_cont tail } \ (f\ u \ s \rightarrow k \ (\text{hd} + s)) ;;
\]

\[
\text{let sum2 } (l:\text{int list}) : \text{int} = \text{sum_cont } l \ (f\ u \ s \rightarrow s)
\]

\[
\text{let rec sum } (l:\text{int list}) : \text{int} = \\
\text{match } l \text{ with} \\
\quad [] \rightarrow 0 \\
\quad \text{hd}::\text{tail} \rightarrow \text{hd} + \text{sum tail} ;;
\]

\[
\text{for all } l:\text{int list}, \\
\text{sum_cont } l \ (f\ u \ x \rightarrow x) = \text{sum } l
\]
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, \( \text{sum\_cont} \ l \ (fun \ s \rightarrow s) \ == \ \text{sum} \ l \)

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

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

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

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

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

case l = []

   must prove: for all k:int->int, sum_cont [] k == k (sum [])
for all l:int list,
   for all k:int->int, sum_cont l k == k (sum l)

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

case l = []

   must prove: for all k:int->int, sum_cont [] k == k (sum [])

   pick an arbitrary k:
for all l:int list,
    for all k:int->int, sum_cont l k == k (sum l)

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

case l = []

    must prove: for all k:int->int, sum_cont [] k == k (sum [])

    pick an arbitrary k:

        sum_cont [] k
Need to Generalize the Theorem and IH

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

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

case l = []

  must prove: for all k:int -> int, sum_cont [] k == k (sum [])

  pick an arbitrary k:

    sum_cont [] k
  == match [] with [] -> k 0 | hd::tail -> ... (eval)
  == k 0 (eval)
for all l:int list,
   for all k:int->int, sum_cont l k == k (sum l)

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

case l = []

must prove: for all k:int->int, sum_cont [] k == k (sum [])

pick an arbitrary k:

   sum_cont [] k
== match [] with [] -> k 0 | hd::tail -> ...  (eval)
  == k 0  
     (eval)

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

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

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

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

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

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

Pick an arbitrary \(k\),
\[
\text{sum} \_\text{cont} \ (\text{hd}::\text{tail}) \ k
\]
\[
= \text{sum} \_\text{cont} \ \text{tail} \ (\text{fun } s \rightarrow k \ (\text{hd} + x)) \quad \text{(eval)}
\]
\[
= (\text{fun } s \rightarrow k \ (\text{hd} + s)) \ (\text{sum} \ \text{tail}) \quad \text{(IH with IH quantifier } k' \text{ replaced with } (\text{fun } x \rightarrow k \ (\text{hd}+x))
\]
Need to Generalize the Theorem and IH

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

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

case l = [] ===> done!

case l = hd::tail

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

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 x -> k (hd+x)) (eval, since sum total and and sum tail valuable)
Need to Generalize the Theorem and IH

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

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

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

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

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

Must prove: \( \text{for all } k : \text{int} \rightarrow \text{int}, \ \text{sum$_{cont}$} (\text{hd}::\text{tail}) \ k = \ k \ \text{(sum (hd}::\text{tail}) \}\)

Pick an arbitrary \( k \),
\[
\text{sum$_{cont}$} (\text{hd}::\text{tail}) \ k
= \text{sum$_{cont}$} \text{tail} (\text{fun } s \rightarrow k \ (\text{hd} + x)) \quad \text{(eval)}
= (\text{fun } s \rightarrow k \ (\text{hd} + s)) \ \text{(sum tail)} \quad \text{(IH with IH quantifier } k' \text{ replaced with (fun } x \rightarrow k \ (\text{hd}+x)) \text{)}
= k \ (\text{hd} + \text{(sum tail)}) \quad \text{(eval, since sum total and \ and sum tail valuable)}
= k \ \text{(sum (hd:tail))} \quad \text{(eval sum, reverse)}
\]
case done!
QED!
Finishing Up

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

$$\text{for all } l:\text{int list,}$$
$$\text{for all } k:\text{int->int, } \text{sum}_\text{cont} l \ k =\ k \ (\text{sum} \ l)$$

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

$$\text{for all } l:\text{int list,}$$
$$\text{sum}_2 l$$
$$= \text{sum}_\text{cont} l \ (\text{fun } s -> s) \quad \text{(by eval sum2)}$$
$$= (\text{fun } s -> s) \ (\text{sum} \ l) \quad \text{(by theorem, instantiating } k \text{ with } (\text{fun } s -> s)$$
$$= \text{sum} \ l$$

So, we've show that the function sum2, which is tail-recursive, is functionally equivalent to the non-tail-recursive function sum.
We tried to prove the *specific* theorem we wanted:

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

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 \((\text{fun } s \to s)\) like our IH required. So we had to *prove a more general theorem* about *all* continuations.

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

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 into tail-recursive ones that use no stack space
  • the stack gets moved into 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

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)
END
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) -> ...
  ;;
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)

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)