A Functional Space Model

COS 326
David Walker
Princeton University
Space

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
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
Space Model

Data type representations:

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

Leaf: 0

Node(i, left, right):

```
Node
3 left right
```
Allocating space

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
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:

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
```
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:

```
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:

```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:

```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 \texttt{Some i}

allocates an option when arg is \texttt{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)
;;
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

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

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:

```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);;
```

It’s execution behavior according to the substitution model:
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)
```
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);;
```

It’s 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)
```
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)
--> 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);;
```ocaml
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);;
```

```assembly
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);;
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

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

0xF8: 0 1 2
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);;

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

execute with substitution

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

choose_subst2:
  compare 1 0
  ...
  jmp ret

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 f1 n = n + x;;
let f2 n = n + y;;
```
How do we implement functions?

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

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:

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

A **closure** is a pair of some code and an environment:
Closure conversion (also called lambda lifting) converts open, nested functions in to 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)
;;
```
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  
;;
```

- Add environment parameter
- Create closures
- Use environment variables instead of free variables
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)
;;

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

```ml
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_env) = c_closure
let f_closure = c_code ((true,1,2), c_env)
let (f_code, f_env) = f_closure
f_code (3, f_env)                                    (* call f code *)
```

- **Closure**: Converts open, nested functions into closed, top-level functions.
- **Add Environment Parameter**: Adds a function to the closure to capture free variables.
- **Create Closures**: Creates closures to encapsulate the environment.
- **Use Environment Variables**: Uses environment variables instead of free variables.
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;;

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

- Add environment parameter
- Create closures
- Use environment variables instead of free variables

```ocaml
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 *)
of (f_closure 3, f_env) in (* 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
```

- **Closure Conversion**
  - Add environment parameter
  - Create closures
  - Use environment variables instead of free variables

```ml
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 *)
  (* call f code *)
```
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) 
;;

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

```plaintext
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 
;;
```

```plaintext
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) in (* 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.

```ocaml
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 _ -> failwith "bad env!"
;;

let f2 (n, env) = 
  match env with 
  F1 _ -> 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 f2_env = {y2:int}
type f2_clos = (int * f2_env -> int) * f2_env

fix l:

  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
```

```ocaml
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

```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 
;;
```

```
fix II:

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
```

“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:

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

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

<table>
<thead>
<tr>
<th>(before)</th>
<th>(after)</th>
</tr>
</thead>
</table>

All function definitions equipped with extra `env` parameter:

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

All free variables obtained from parameters or environment:

```plaintext
x
env.cx
```

All functions values paired with environment:

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

All function calls extract code and environment and call code:

```plaintext
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
Assignmet #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)
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 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  
;;
```
Four functions: Green works on big inputs; Red doesn’t.

```ml
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 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:

```ocaml
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;;
```
Tail Recursion

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

Not tail-recursive, the substitution model:

```
sum_to 1000000
--> 1000000 + sum_to 99999
```

```
(* sum of 0..n *)
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;

let big_int = 1000000;;
sum big_int;;
```
A *tail-recursive function* does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

\[
\text{let rec sum_to (n:int) : int =}
\begin{cases}
  1000000 & \text{if } n > 0 \\
  0 & \text{else}
\end{cases}
\]

 expressive size grows at every recursive call ... lots of adding to do after the call returns"
A tail-recursive function does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
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
```
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;;
```

sum_to 1000000
-->
1000000 + sum_to 99999
-->
1000000 + 99999 + sum_to 99998
--> ...
-->
1000000 + 99999 + 99998 + ... + sum_to 0
-->
1000000 + 99999 + 99998 + ... + 0

(* sum of 0..n *)

recursion finally bottoms out
A *tail-recursive function* does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```plaintext
sum_to 1000000
  -->
  1000000 + sum_to 99999
  -->
  1000000 + 99999 + sum_to 99998
  -->
  ... 
  -->
  1000000 + 99999 + 99998 + ... + sum_to 0
  -->
  1000000 + 99999 + 99998 + ... + 0
  -->
  ... add it all back up ...
```

```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;;
```

**do a long series of additions to get back an int**
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
```
Non-tail recursive

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

sum_to 10000
```
Non-tail recursive

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

sum_to 10000
```
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

```
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

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

sum_to 100
```
Data Needed on Return Saved on Stack

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

not much space left! will run out soon!

the stack

9996
9997
9998
9999
10000

every non-tail call puts the data from the calling context on the stack
Memory is partitioned: Stack and Heap

heap space (big!)

stack space (small!)
Tail Recursion

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

Tail-recursive:

```plaintext
sum_to2 1000000

(* 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:

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

**Tail-recursive:***

```plaintext
sum_to2 1000000
--> aux 1000000 0
--> aux 99999 1000000
--> aux 99998 1999999
```

(* sum of 0..n *)

```plaintext
let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```
A *tail-recursive function* is a function that does no work after it calls itself recursively.

**Tail-recursive:**

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

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

(addition overflow occurred at some point)

constant size expression in the substitution model
A \textit{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
;;

stack

aux 10000 0
Tail Recursion

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

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

```
stack
  aux 9999 10000
```
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
aux 9998 19999
```
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
;;

aux 9997 29998
```
A **tail-recursive function** is a function that does no work after it calls itself recursively.

```ocaml
(* sum of 0..n *)
let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)  (* recursive call *)
    else a
  in
  aux n 0
;;
```

The stack diagram shows the function calls during the execution of the `sum_to2` function.
We used human ingenuity to do the tail-call transform.

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

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

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
;;
```
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?
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.
SUMMARY
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 an important program transformation:

- *closure conversion* makes nested functions with free variables in to pairs of closed code and environment
- next time: *continuation-passing style* transformation