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

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

CONVENTIONAL DATA

Blackboard!

Numbers

Tuples

Data types

Lists

Space Model

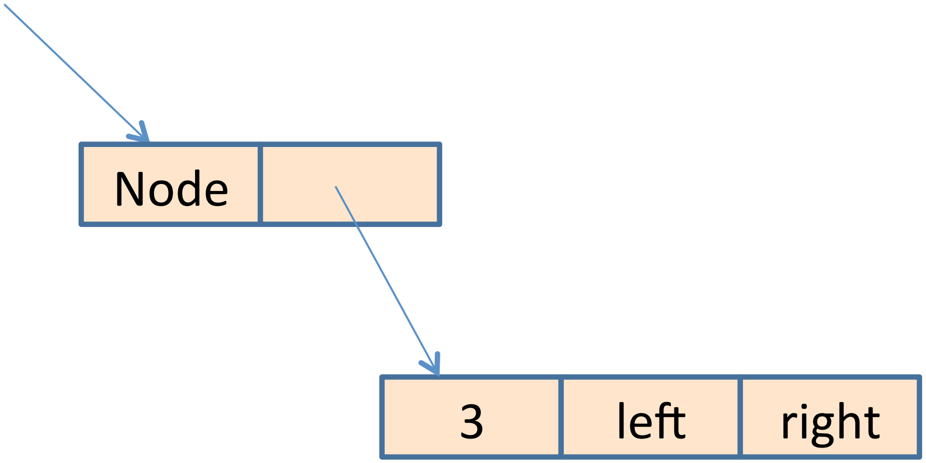
Data type representations:

```
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?

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)
```

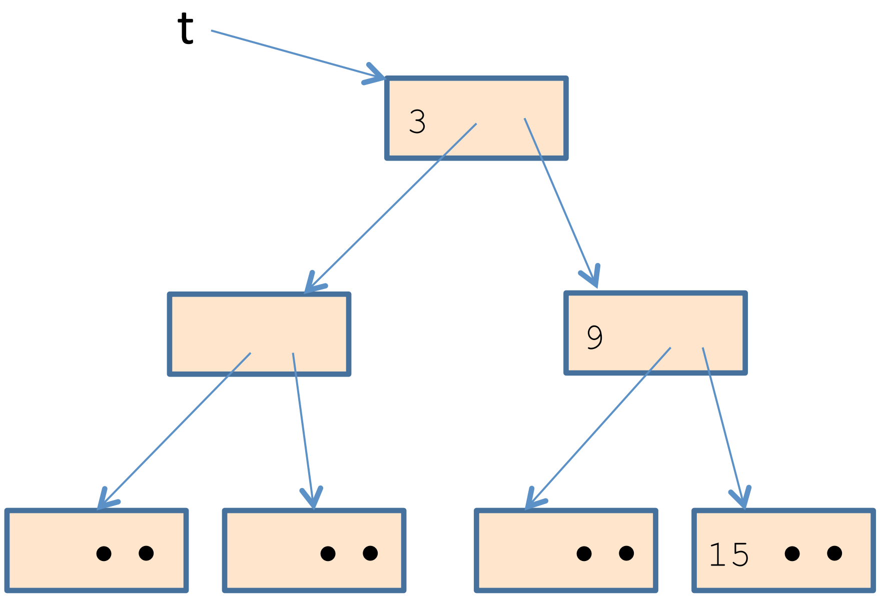
Allocating space

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



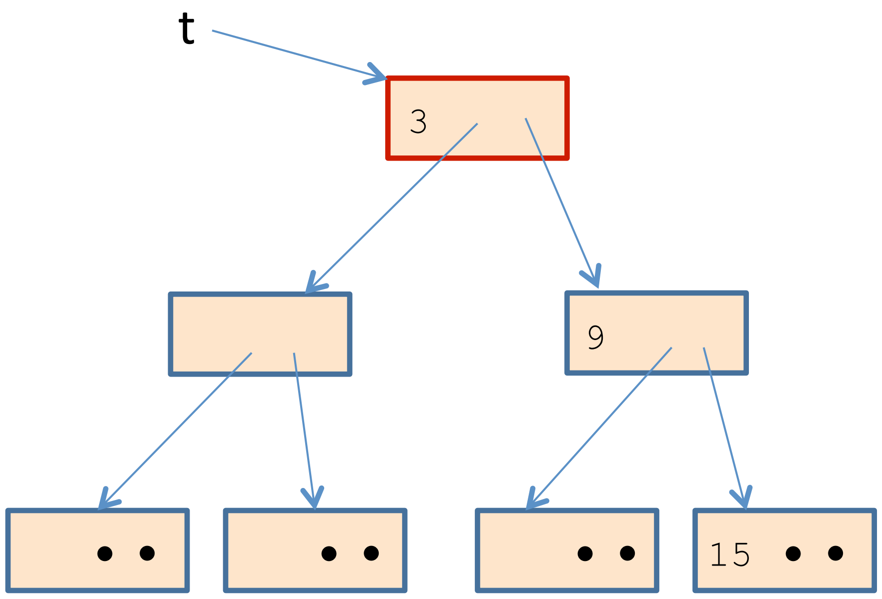
Allocating space

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



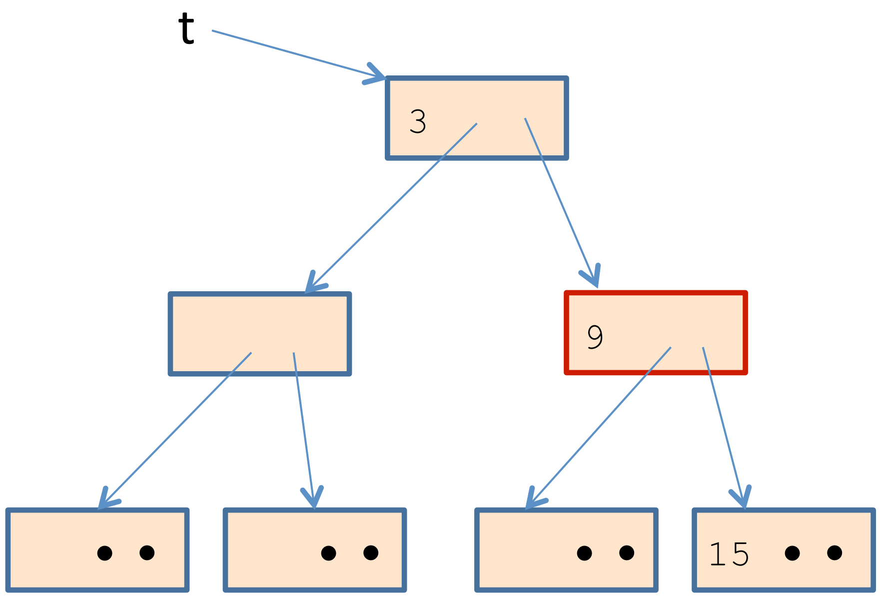
Allocating space

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



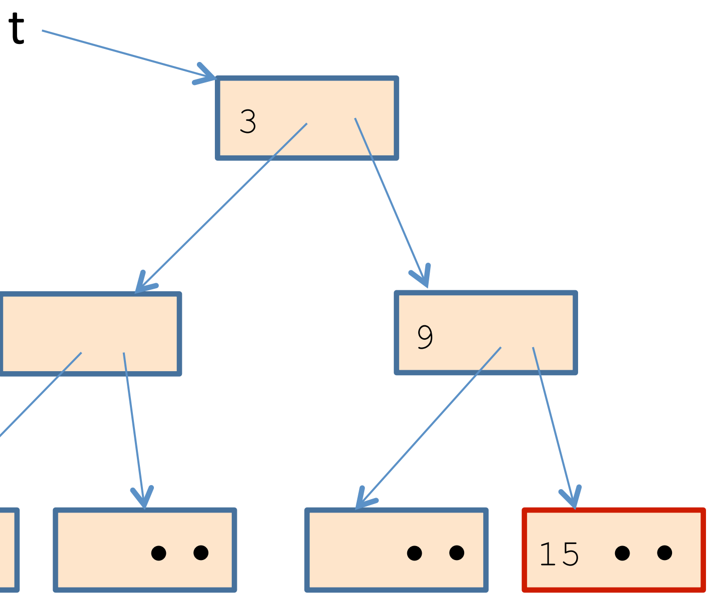
Allocating space

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



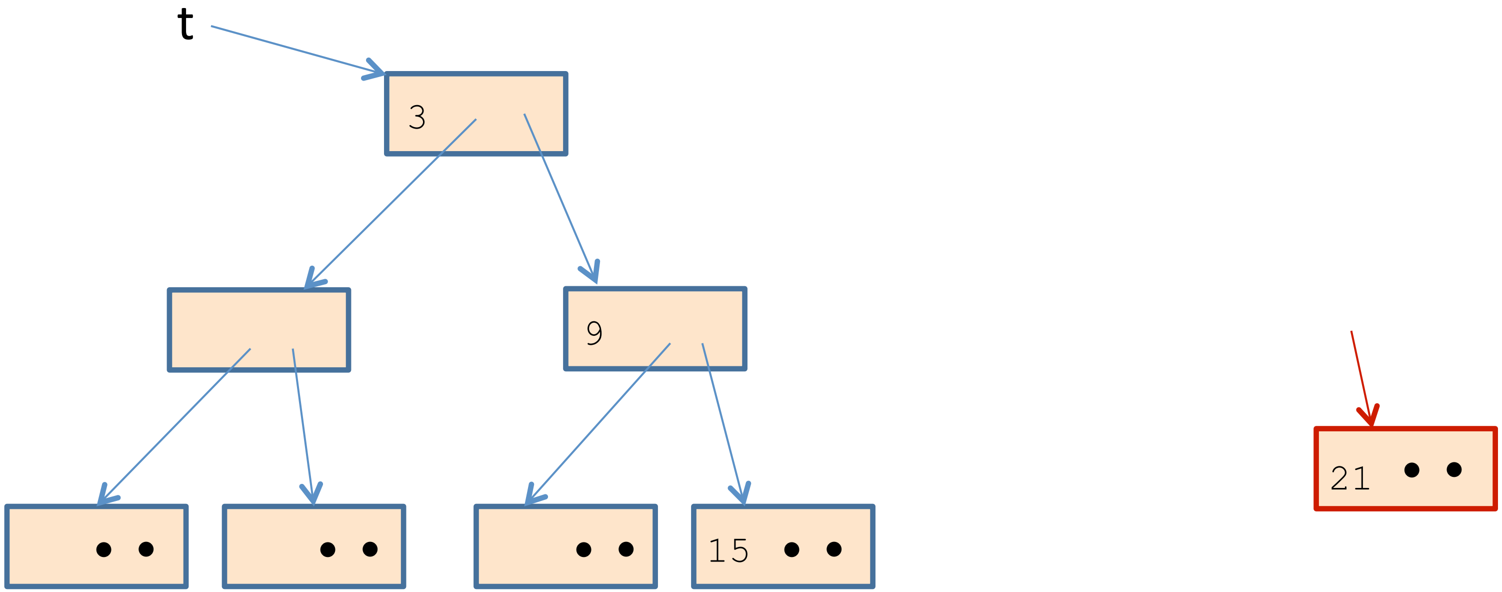
Allocating space

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



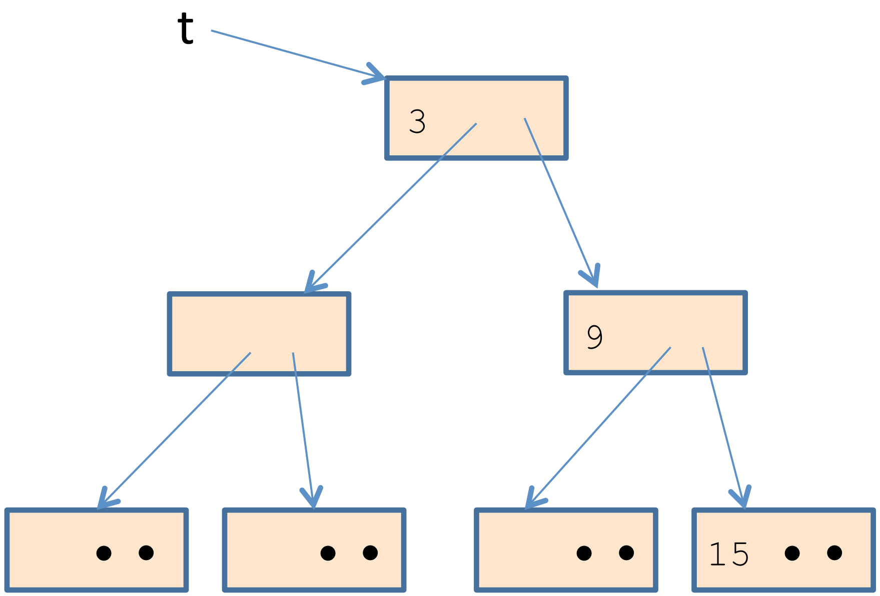
Allocating space

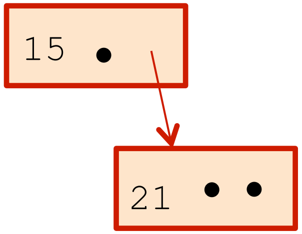
Whenever you use a constructor, space is allocated:

```
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





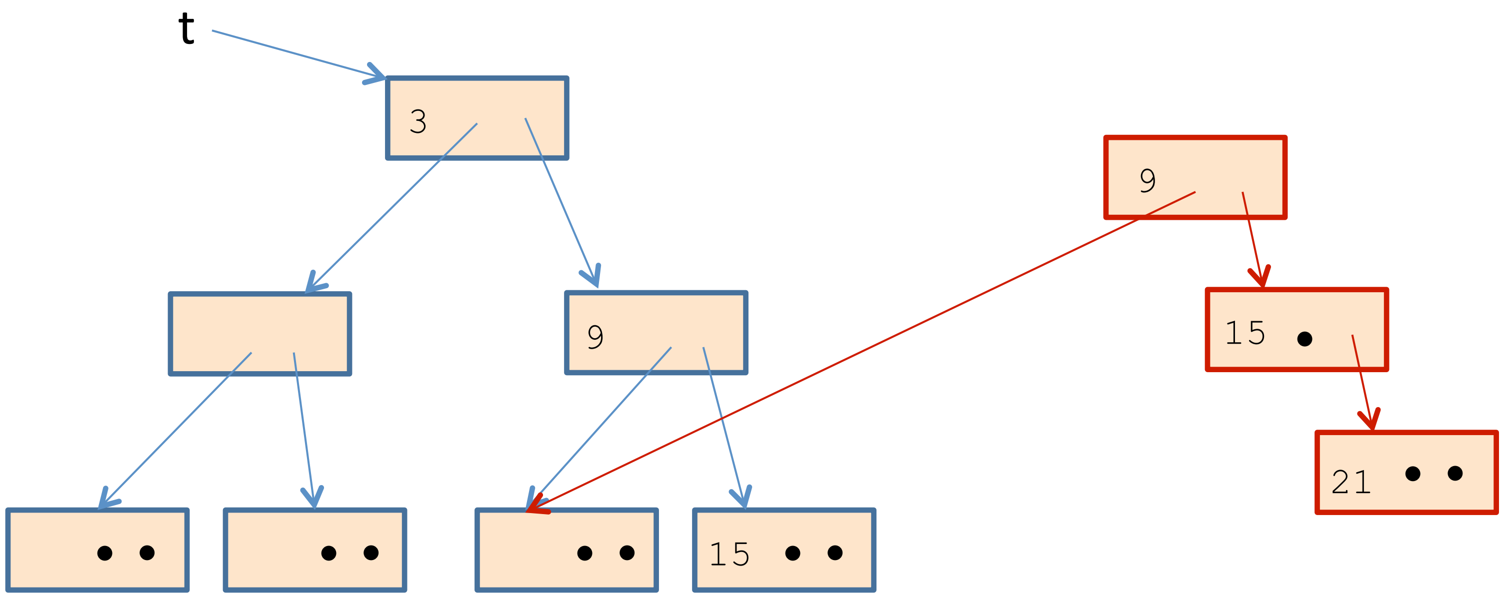
Allocating space

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



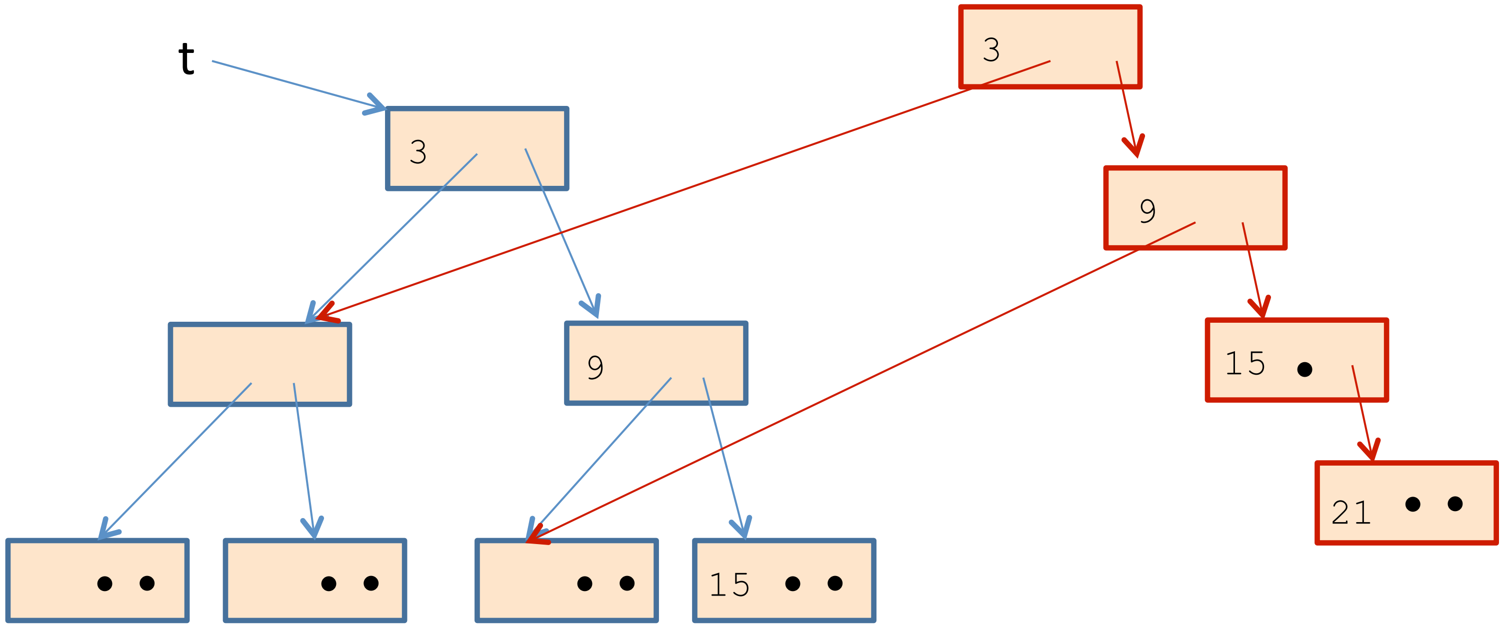
Allocating space

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



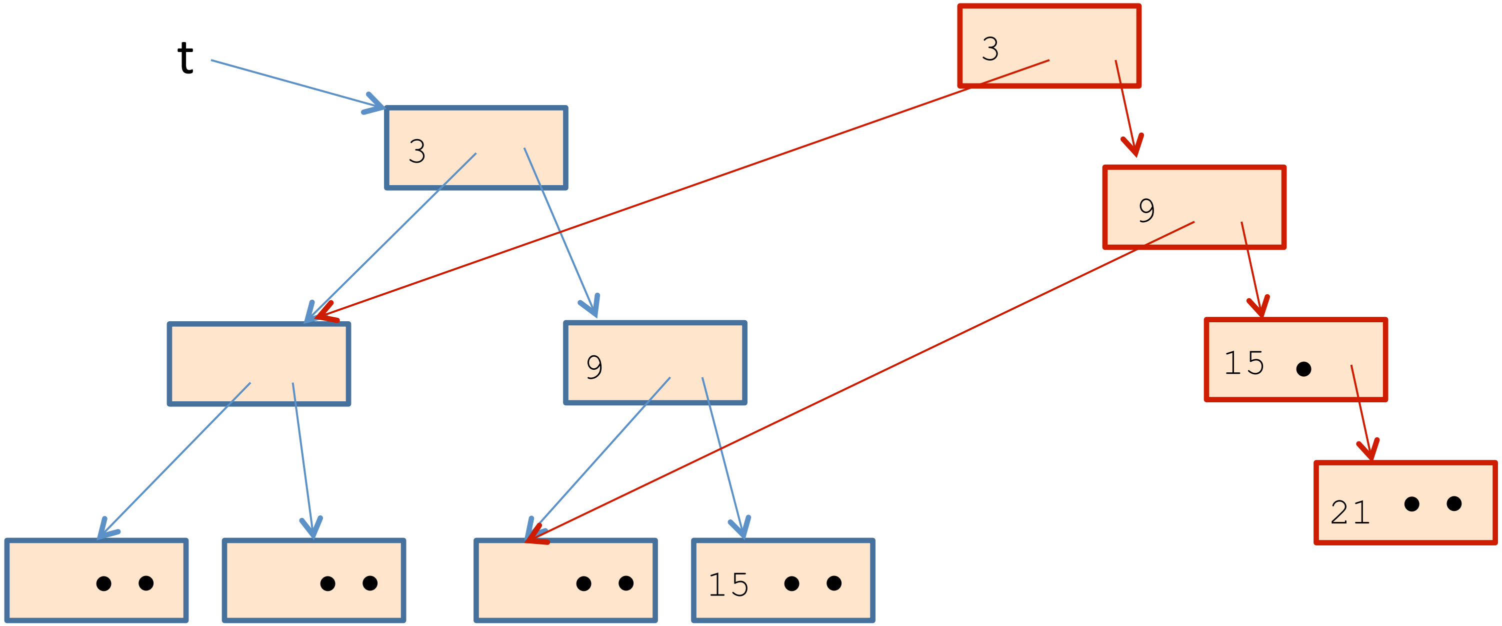
Allocating space

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)
```

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

~ log n, if tree with n nodes is balanced



Compare

```
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”
;;
```

Compare

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

allocates nothing
when arg is Some i

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

allocates an option
when arg is Some i

Compare

```
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
;;
```

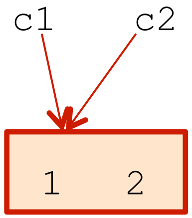
```
let double (c1:int*int) : int*int =
  cadd c1 c1
;;
```

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

Compare

```
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)
;;
```

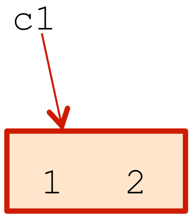
Compare

```
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)
;;
```



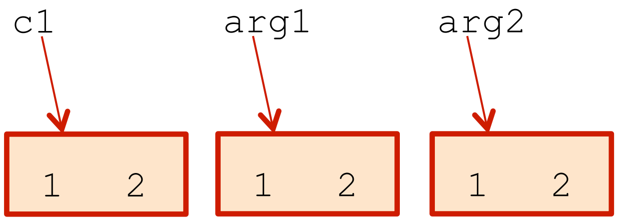
Compare

```
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)
;;
```



Compare

```
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
;;
```

no allocation

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

no allocation

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

allocates 2 pairs

Compare

```
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 (x1,y1) = c1 in
  cadd c1 c1
;;
```

double does not allocate

extracts components: it is a read

FUNCTION CLOSURES

Closures

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);;
```

Closures

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);;
```

It’s execution behavior according to the substitution model:

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

Closures

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);;
```

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)
```

Closures

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);;
```

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)
```

Closures

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);;
```

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)
-->
    (fun n -> n + 1)
```

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);;
```

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);;
```

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
```

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);;
```

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

```
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);;
```

compile

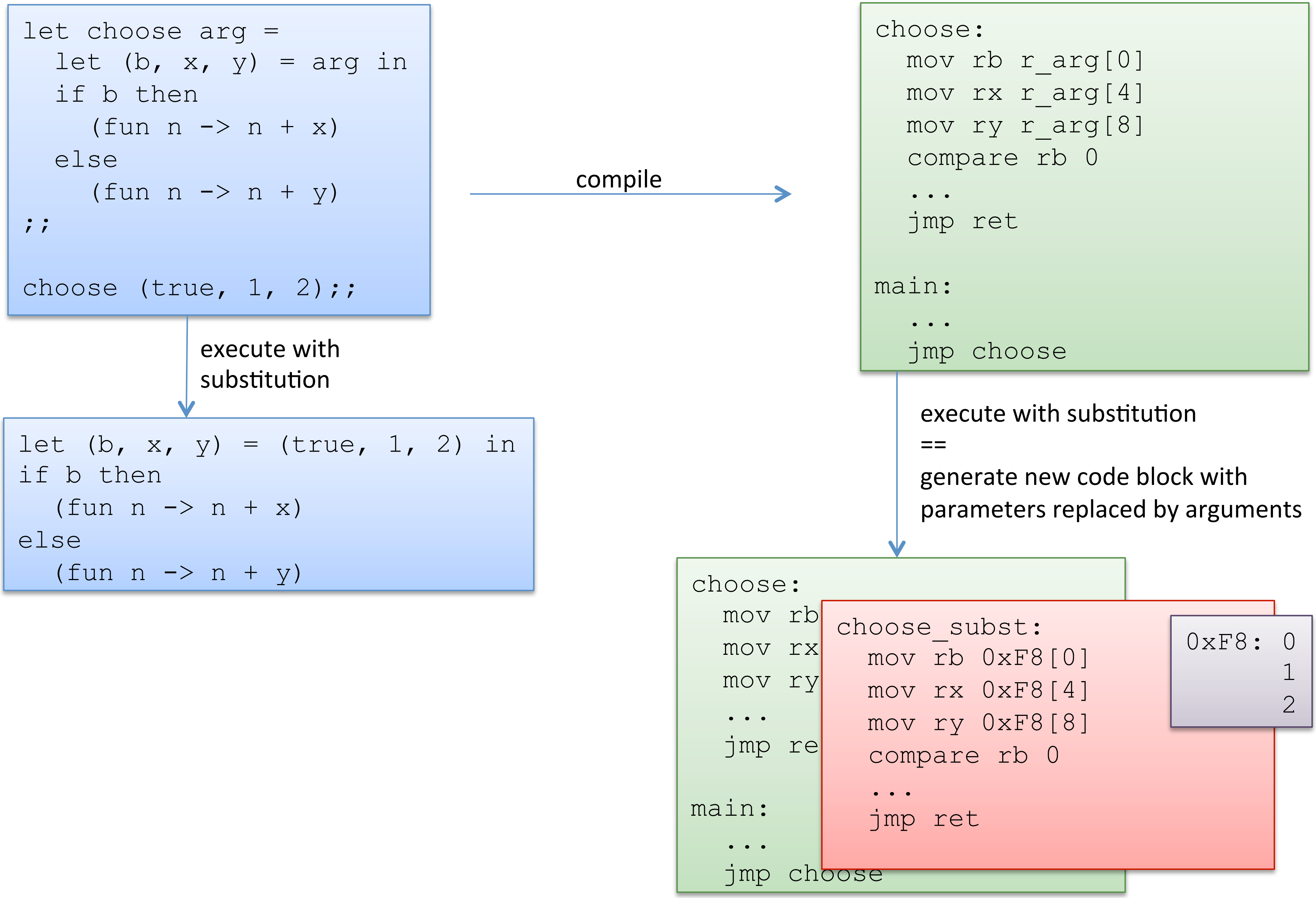
```
choose:
  mov rb r_arg[0]
  mov rx r_arg[4]
  mov ry r_arg[8]
  compare rb 0
  ...
  jmp ret

main:
  ...
  jmp choose
```

execute with substitution

```
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

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);;
```

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

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

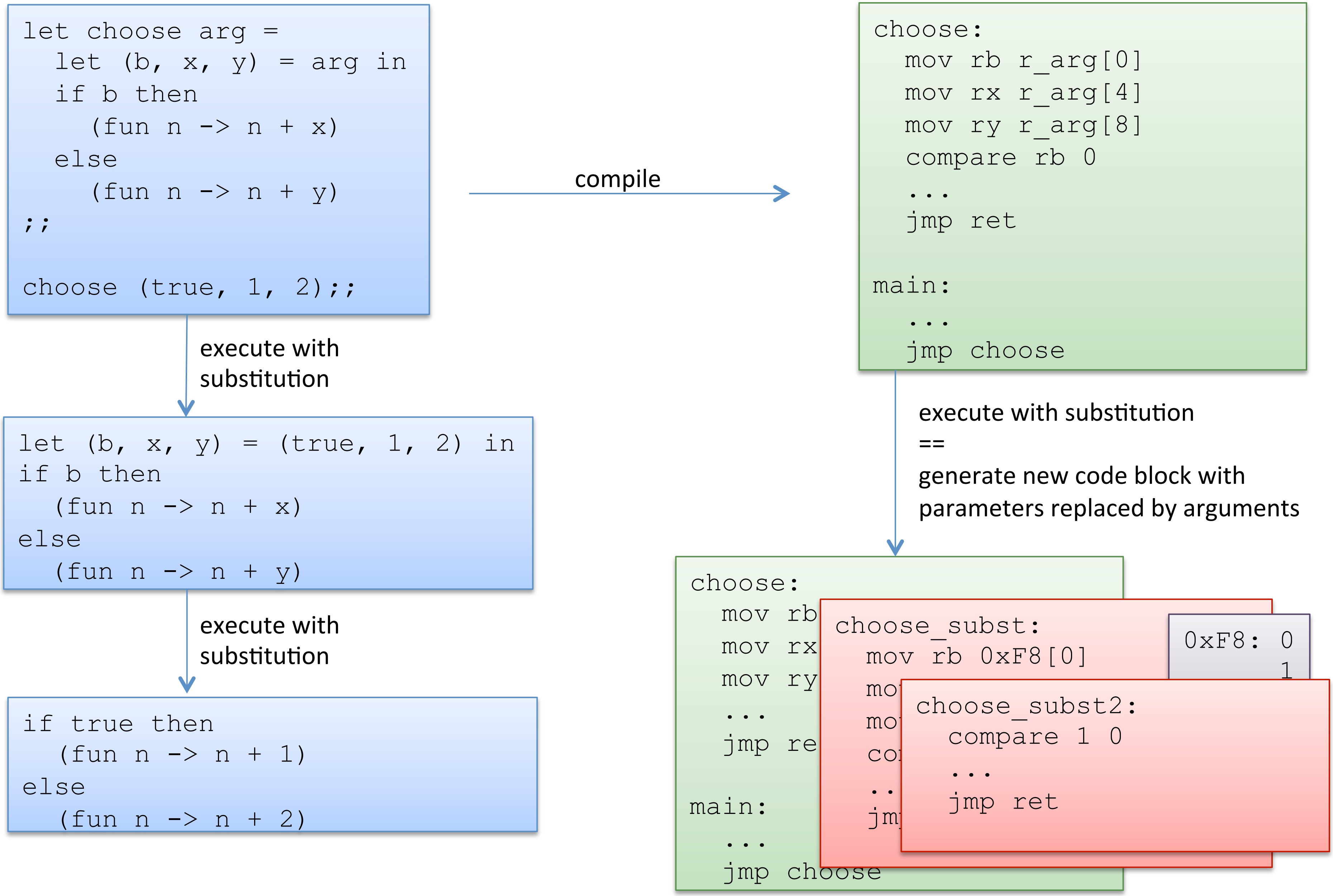
execute with substitution
==
generate new code block with parameters replaced by arguments

```
choose:
  mov rb
  mov rx
  mov ry
  ...
  jmp re

main:
  ...
  jmp choose
```

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

```
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);;
```

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

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

execute with substitution

```
if true then
  (fun n -> n + 1)
else
  (fun n -> n + 2)
```

execute with substitution
==
generate new code block with parameters replaced by arguments

```
choose:
  mov rb
  mov rx
  mov ry
  ...
  jmp re

main:
  ...
  jmp choose
```

```
choose_subst:
  mov rb 0xF8[0]
  mo
  mo
  co
  ..
  jm
```

```
0xF8: 0
      1
```

```
choose_subst2:
  compare 1 0
  ...
  jmp ret
```

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

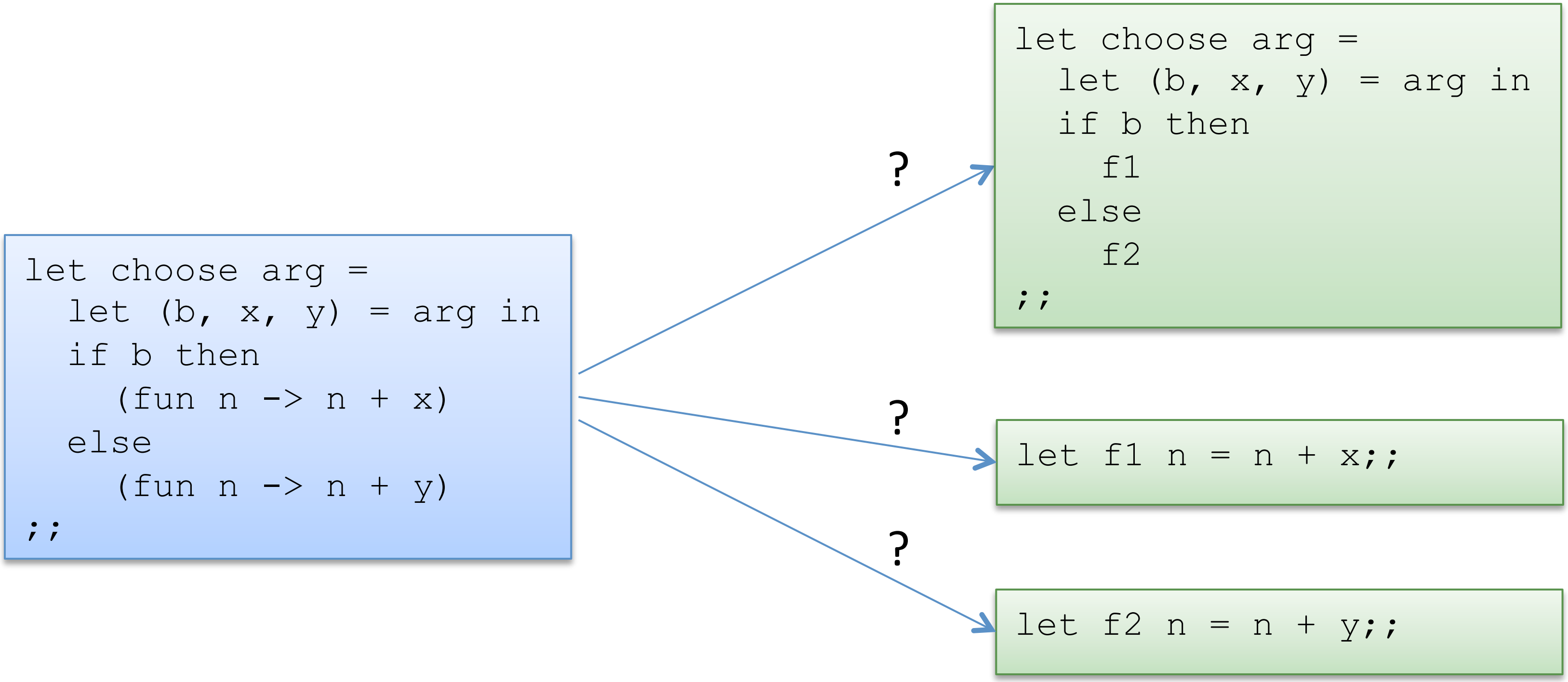
```
let add (x:int*int) : int =
  let (y,z) = x in
  y + z
;;
```

```
# 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.



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

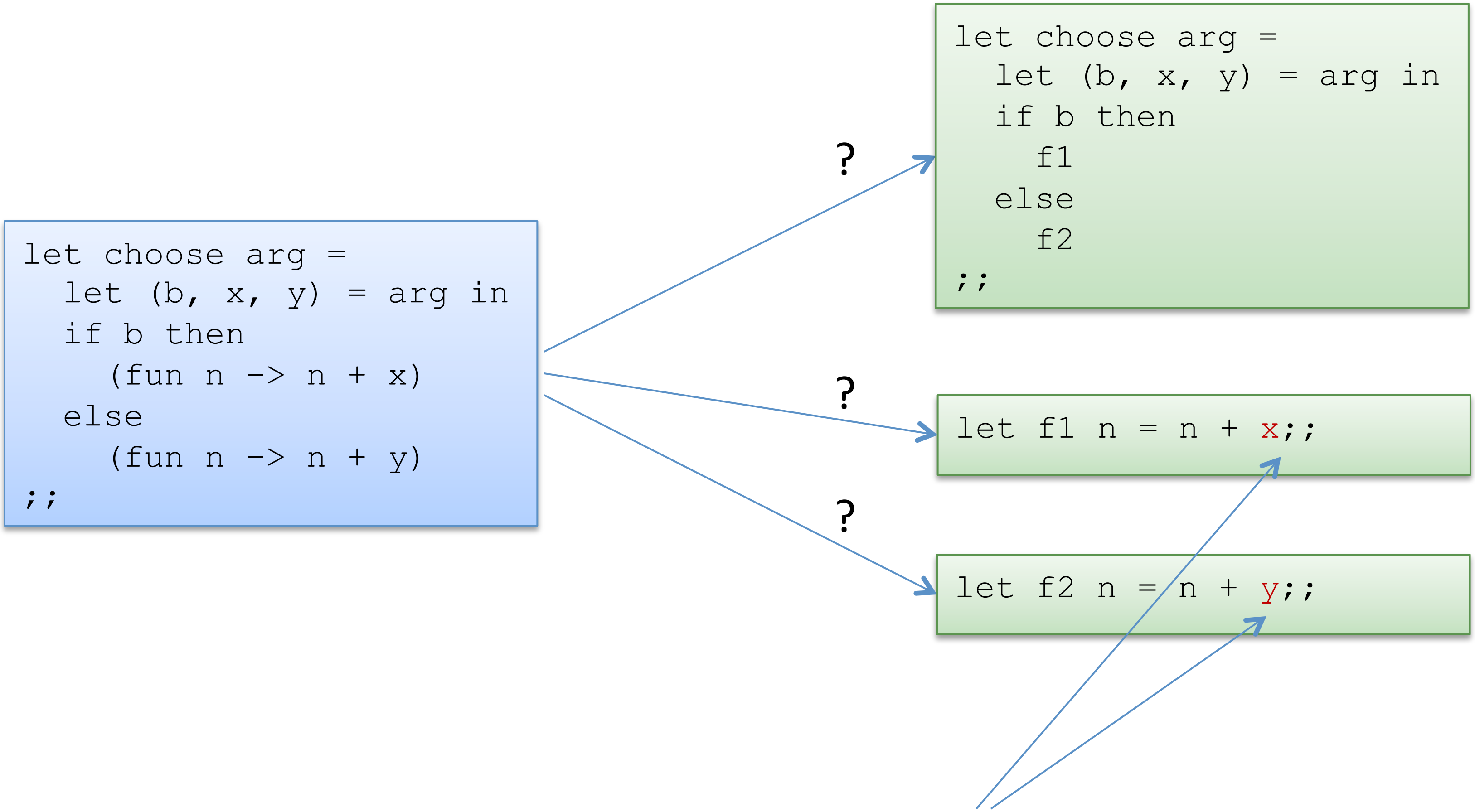
```
let choose arg =
  let (b, x, y) = arg in
  if b then
    f1
  else
    f2
;;
```

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

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

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;;
```

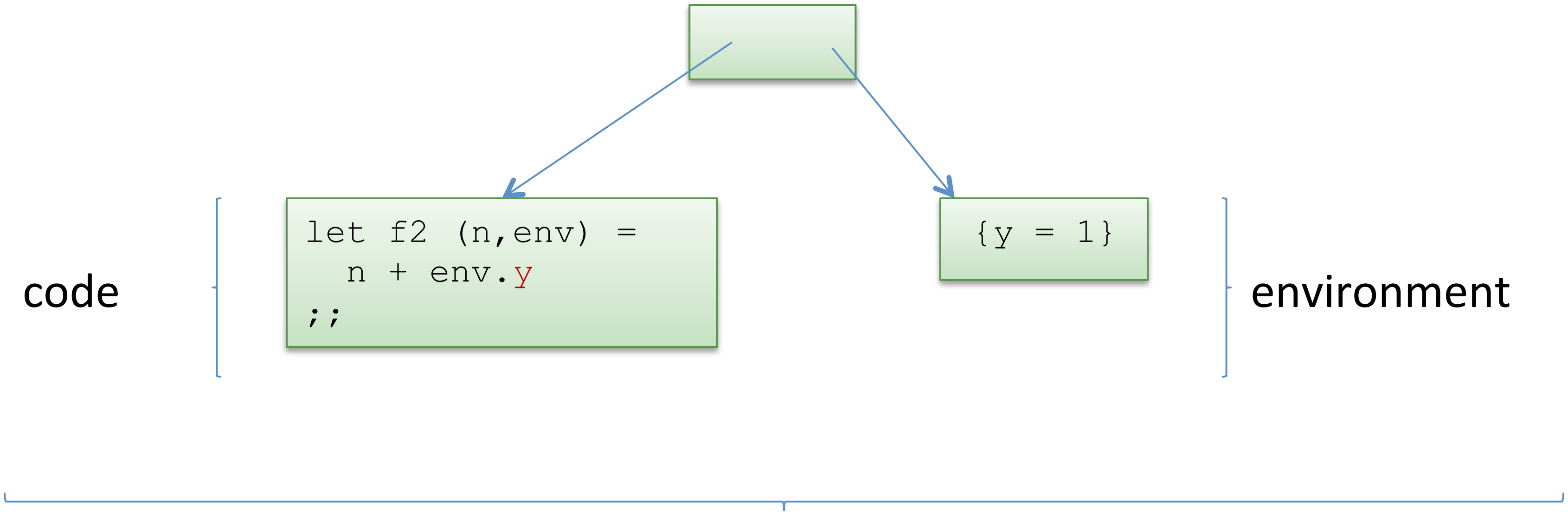
Darn! *Doesn’t work naively*. Nested functions contain *free variables*.
Simple unnesting leaves them undefined.

How do we implement functions?

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

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

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



code

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

```
{y = 1}
```

environment

closure

Closure Conversion

Closure conversion (also called lambda lifting) converts open, nested functions in to 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

Closure conversion (also called lambda lifting) converts open, nested functions in to 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)
;;
```

add environment parameter

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

create closures

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

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

use environment variables instead of free variables

Closure Conversion

Closure conversion (also called lambda lifting) converts open, nested functions in to 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})
;;
```

add environment parameter

create closures

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

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

use environment variables instead of free variables

```
(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_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 *)
;;
```

Closure Conversion

Closure conversion (also called lambda lifting) converts open, nested functions in to 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)
;;
```

add environment parameter

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

create closures

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

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

use environment variables instead of free variables

```
(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_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 *)
```

Closure Conversion

Closure conversion (also called lambda lifting) converts open, nested functions in to 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)
;;
```

add environment parameter

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

create closures

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

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

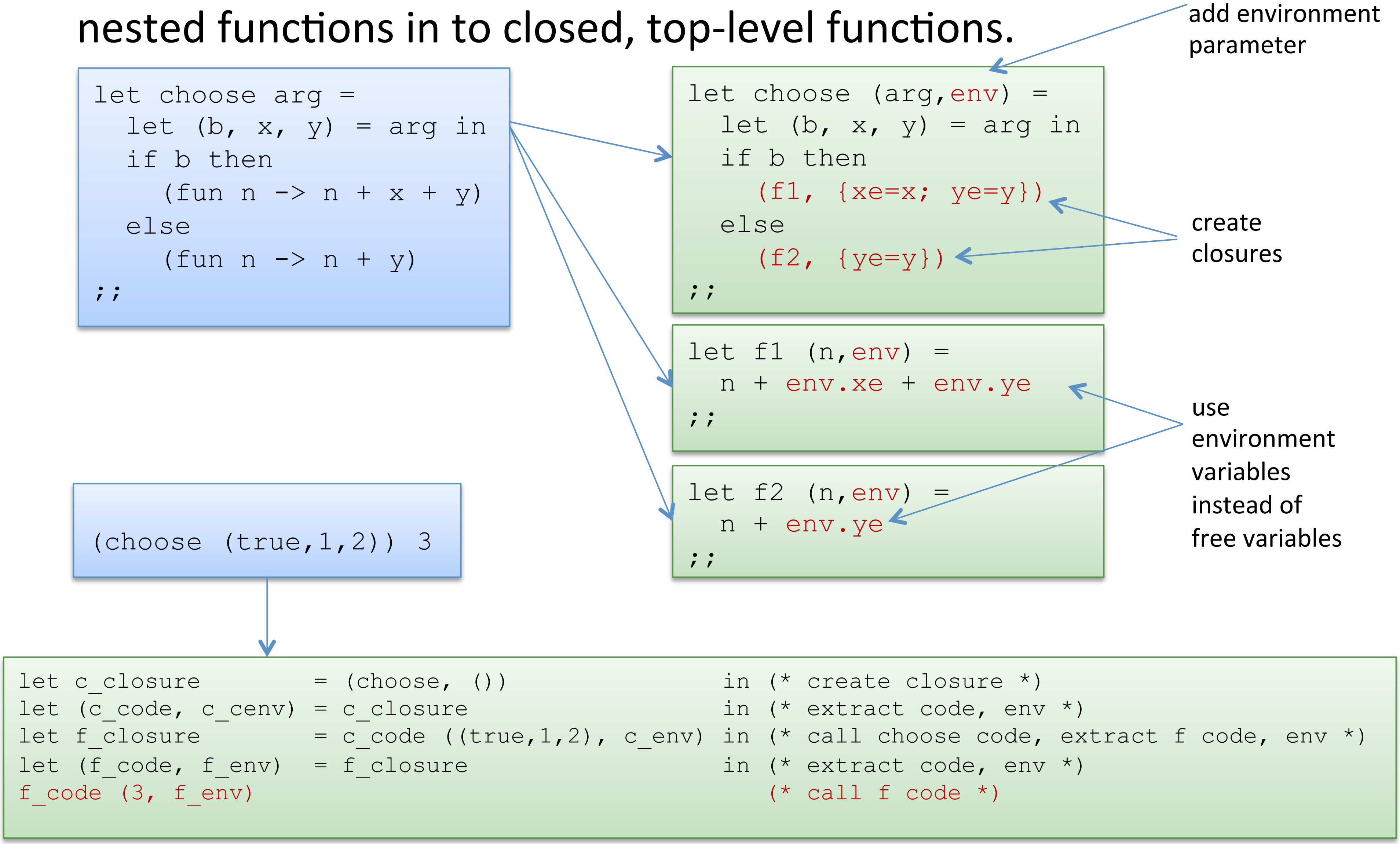
use environment variables instead of free variables

```
(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_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 *)
```

Closure Conversion

Closure conversion (also called lambda lifting) converts open, nested functions in to 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)
;;
```

add environment parameter

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

create closures

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

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

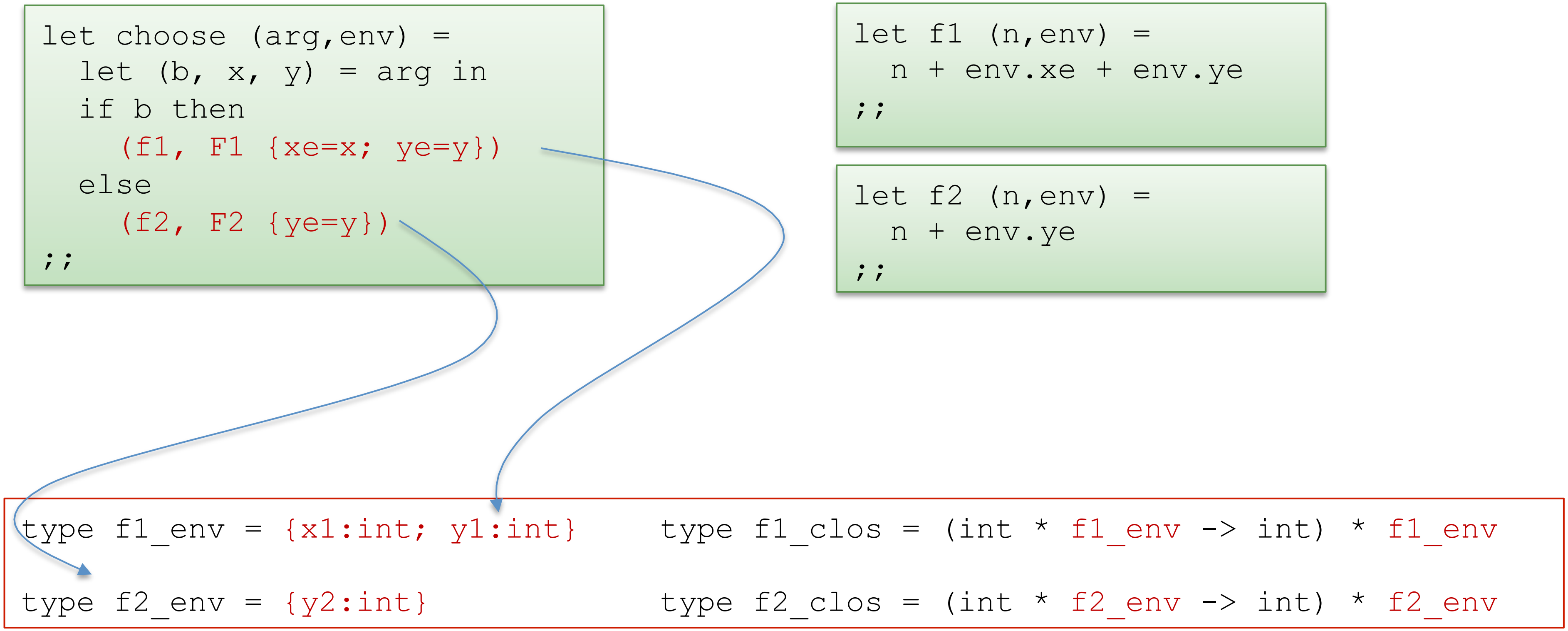
use environment variables instead of free variables

```
(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_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 *)
```

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, 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

```
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 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

```
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 = 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
;;
```

"*From System F to Typed Assembly Language*,"
-- Morrisett, Walker et al.

```
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 = exists env.(int * env -> int) * env
type f2_clos = exists env.(int * env -> int) * env
```

Aside: Existential Types

map has a *universal* polymorphic type:

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 int -> int, we get a function with *existential* polymorphic type:

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

(before)	(after)

All function definitions equipped with extra env parameter:

`let f arg = ...`	`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 Closures

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

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

Some Innocuous Code

```
(* 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;;
```

Let’s try it.

(Go to tail.ml)

Some Other Code

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

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

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

Some Other Code

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

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

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

code that works:

no computation after recursive function call

Tail Recursion

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

Not tail-recursive, the substitution model:

```
    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;;
```

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
-->
    1000000 + 99999 + sum_to 99998
```

```
(* 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;;
```

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:

```
    sum_to 1000000
-->
    1000000 + sum_to 99999
-->
    1000000 + 99999 + sum_to 99998
-->
    ...
-->
    1000000 + 99999 + 99998 + ... + sum_to 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;;
```

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

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

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

let big_int = 1000000;;

sum big_int;;
```

recursion
finally bottoms out

Tail Recursion

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

Not tail-recursive, the substitution model:

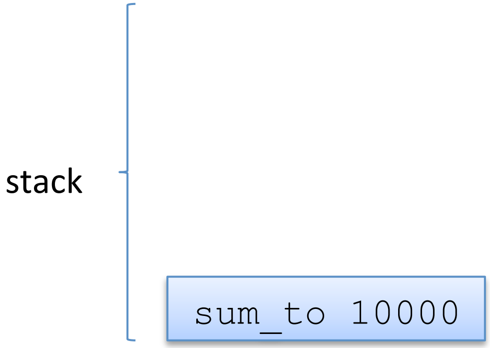
```
    sum_to 1000000
-->
    1000000 + sum_to 99999
-->
    1000000 + 99999 + sum_to 99998
-->
    ...
-->
    1000000 + 99999 + 99998 + ... + sum_to 0
-->
    1000000 + 99999 + 99998 + ... + 0
-->
    ... add it all back up ...
```

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

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

let big_int = 1000000;;

sum big_int;;
```

do a long series
of additions to get
back an int

Non-tail recursive

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

sum_to 10000
```

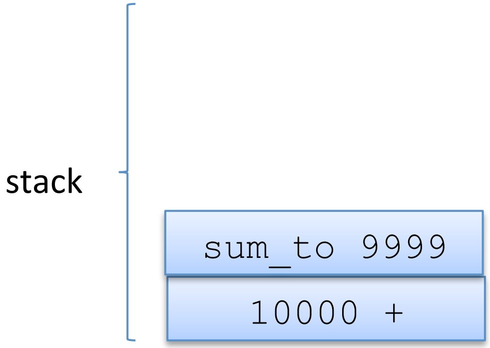


stack

`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
```



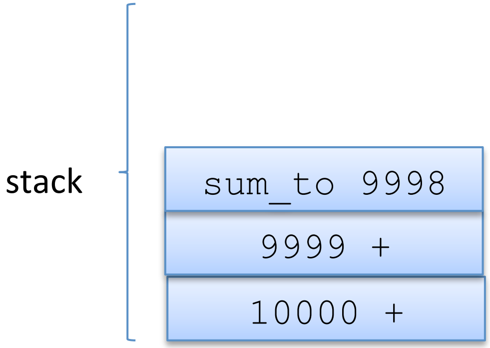
stack

sum_to 9999

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
```



stack

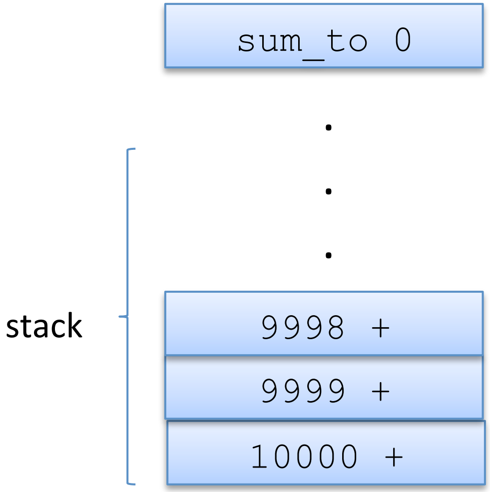
sum_to 9998

9999 +

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
```



sum_to 0

.

.

.

stack

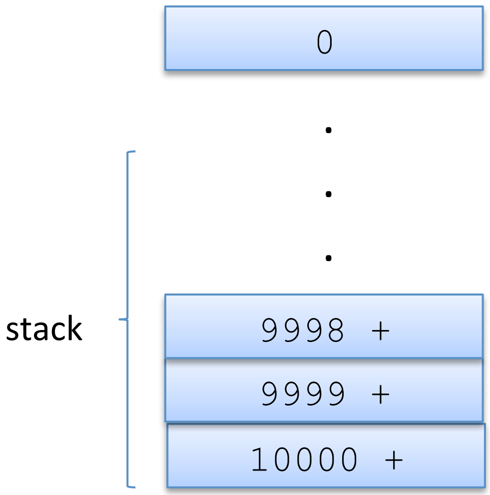
9998 +

9999 +

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
```



0

.
.
.

stack

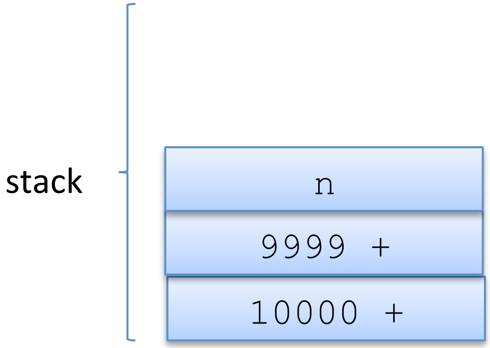
9998 +

9999 +

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
```



stack

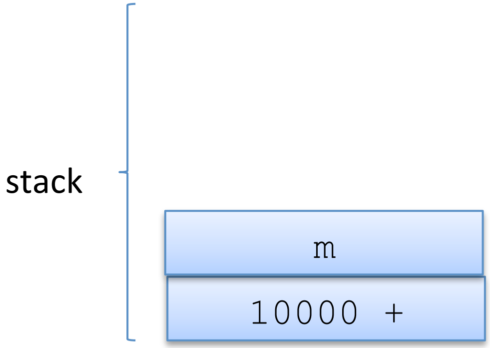
n

9999 +

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
```



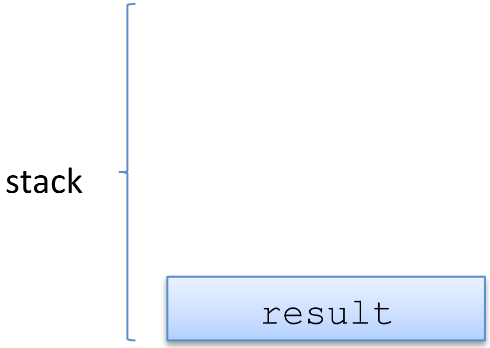
stack

m

10000 +

Non-tail recursive

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

sum_to 100
```

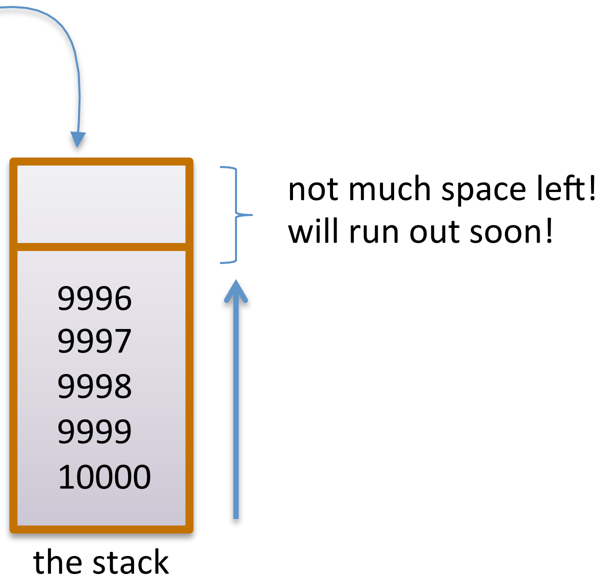


stack

result

Data Needed on Return Saved on Stack

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



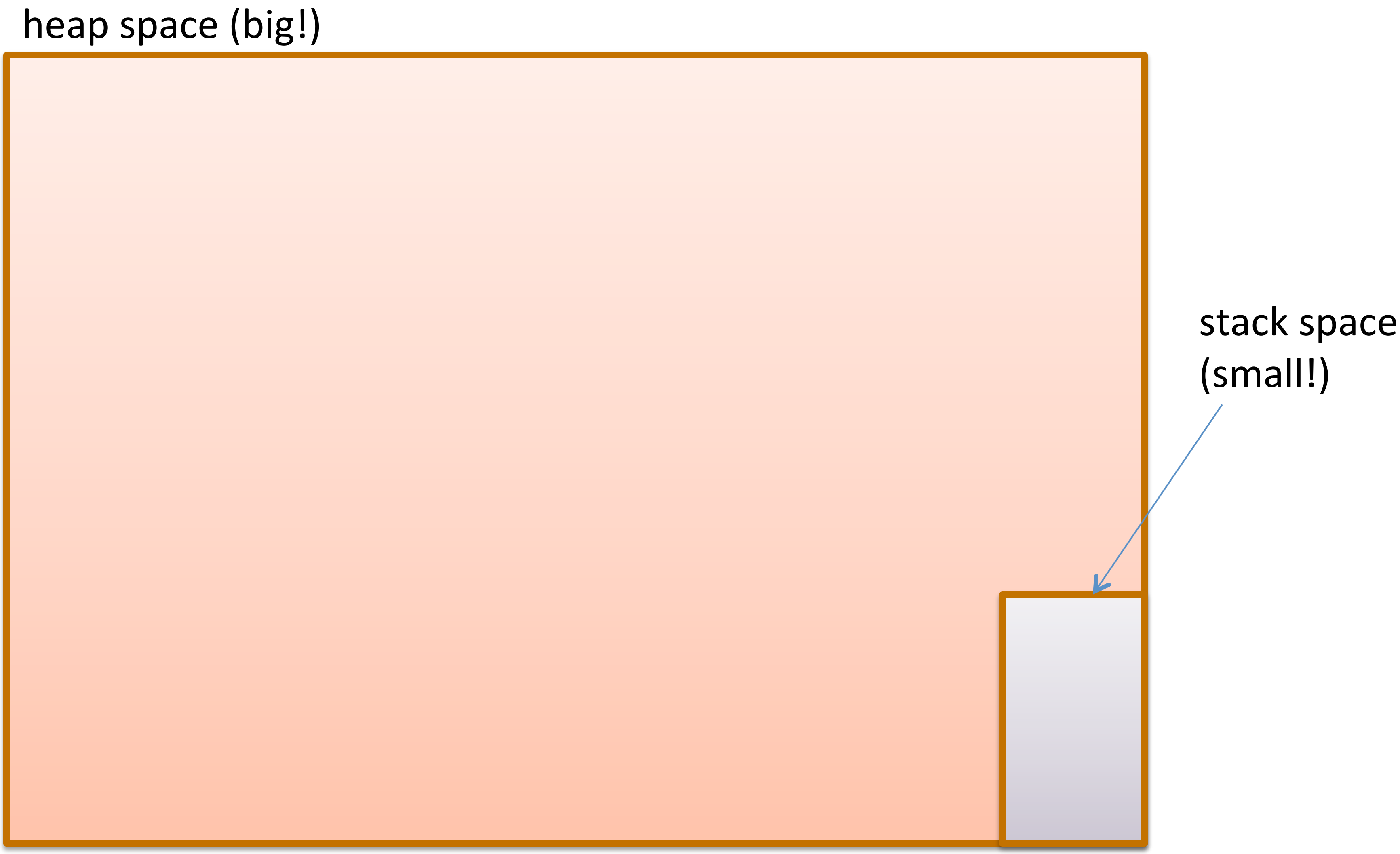
9996
9997
9998
9999
10000

the stack

not much space left!
will run out soon!

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

Memory is partitioned: Stack and Heap



Tail Recursion

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

Tail-recursive:

```
   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:

```
    sum_to2 1000000
-->
    aux 1000000 0
```

```
(* 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
```

```
(* 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
;;
```

Tail Recursion

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

Tail-recursive:

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

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

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

constant size expression
in the substitution model

(addition overflow occurred
at some point)

Tail Recursion

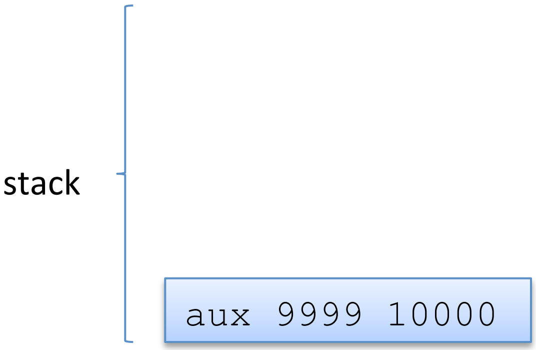
A *tail-recursive function* is a function that does no work after it calls itself recursively.

```
(* 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.

```
(* 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.

```
(* 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
;;
```

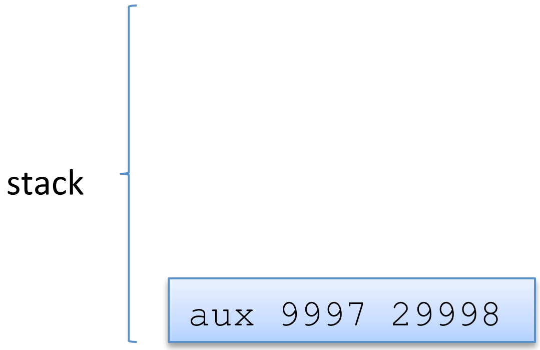
stack

`aux 9998 19999`

Tail Recursion

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
;;
```

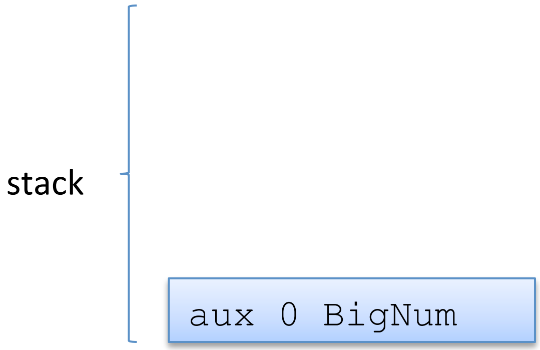

stack

```
aux 9997 29998
```

Tail Recursion

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
;;
```



Question

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?

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*!

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

human ingenuity

```
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
;;
```

CONTINUATION-PASSING STYLE

CPS!

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 in to 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