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 functional 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

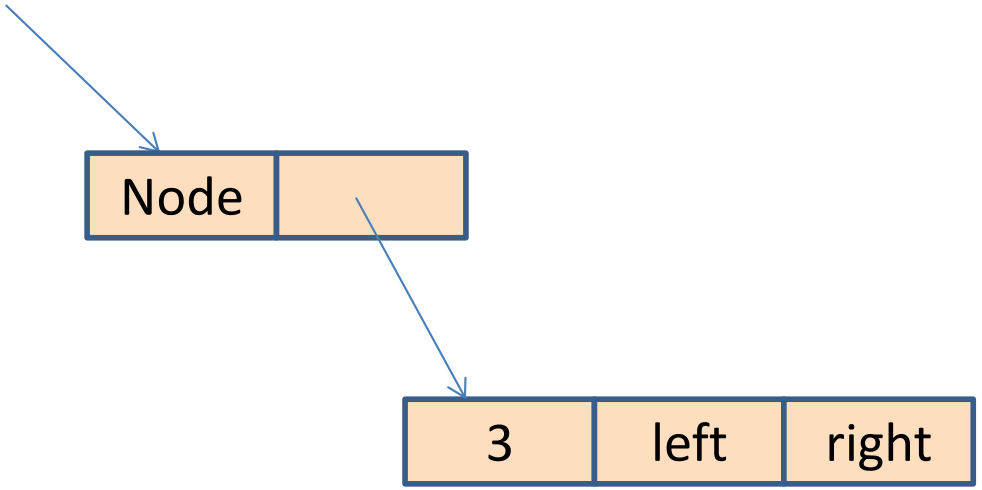
Data type representations:

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

Leaf:

0

Node(i, left, right):



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)
```

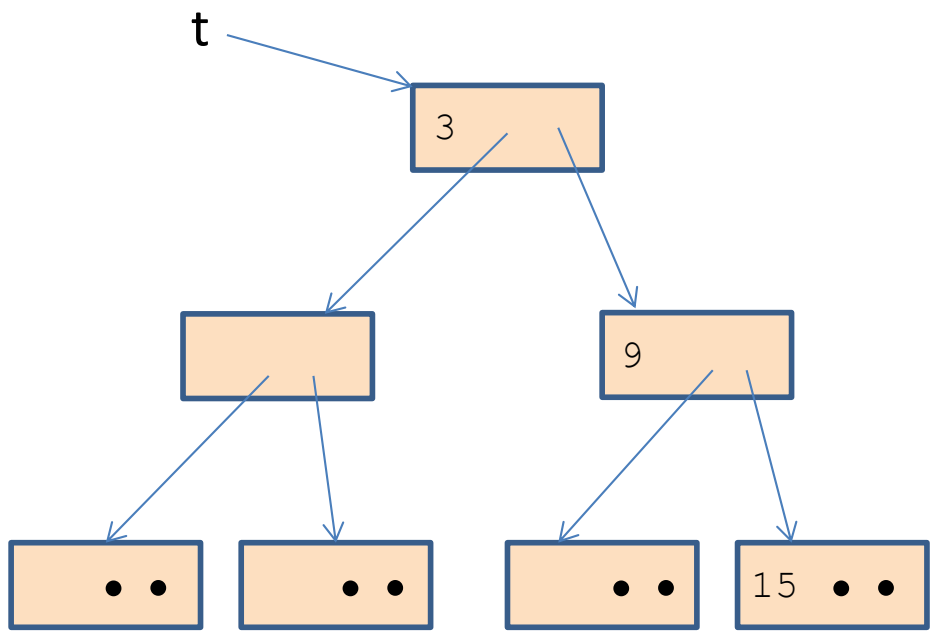
Allocating space

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



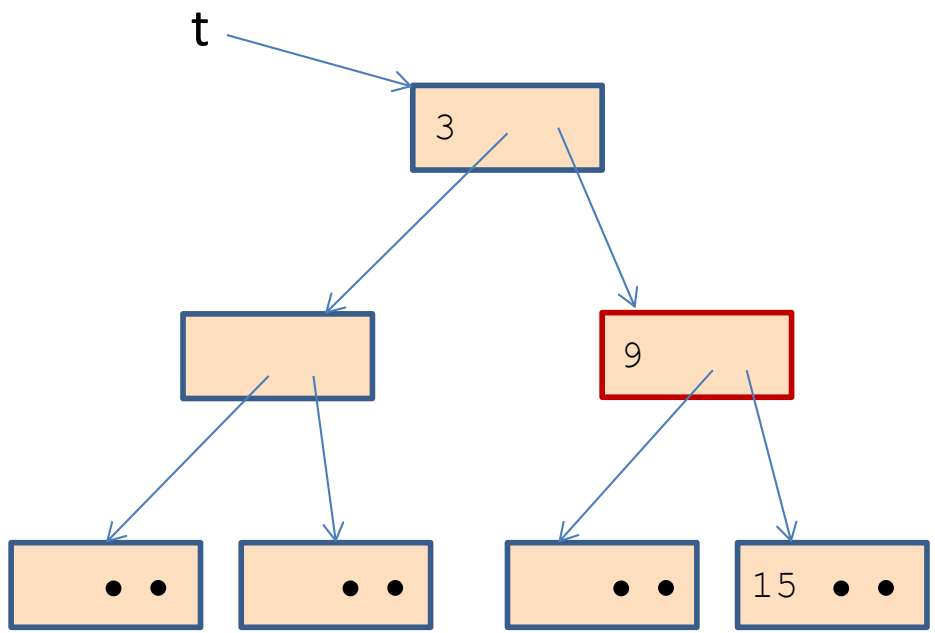
Allocating space

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



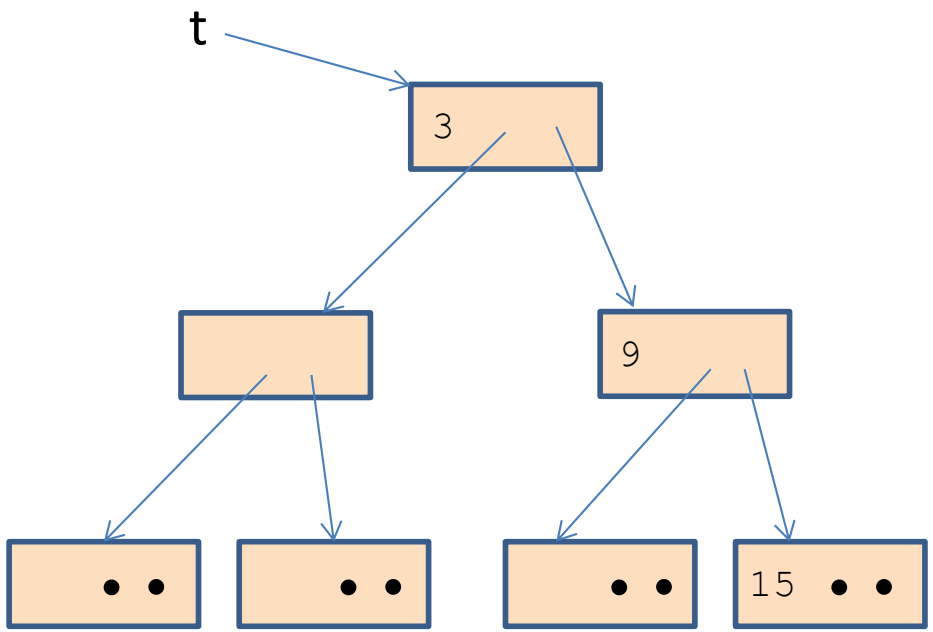
Allocating space

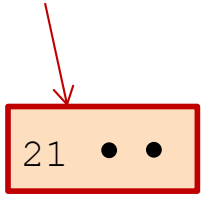
Whenever you use a constructor, space is allocated:

```
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





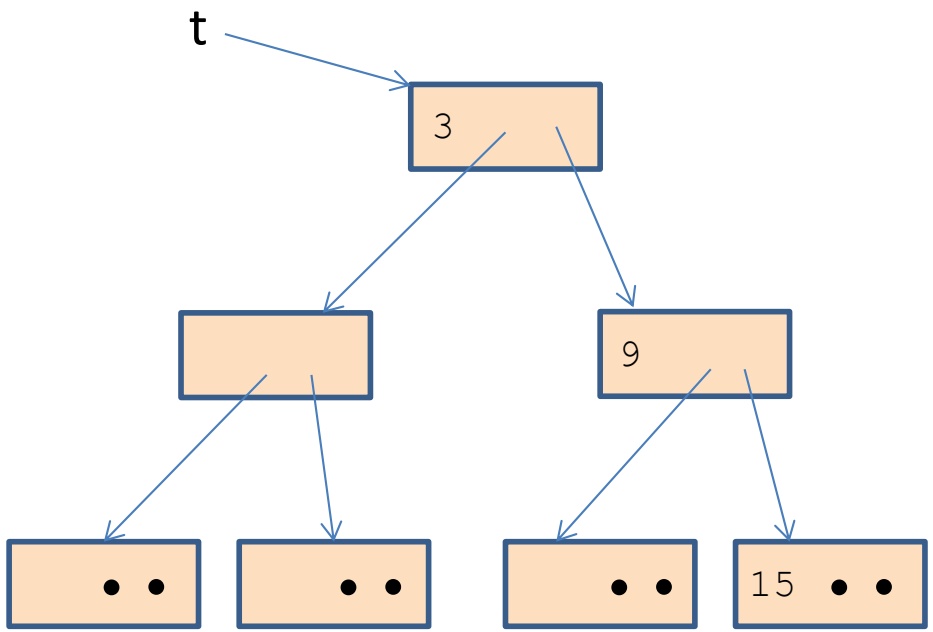
Allocating space

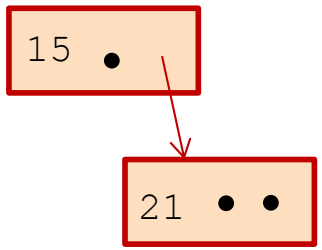
Whenever you use a constructor, space is allocated:

```
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





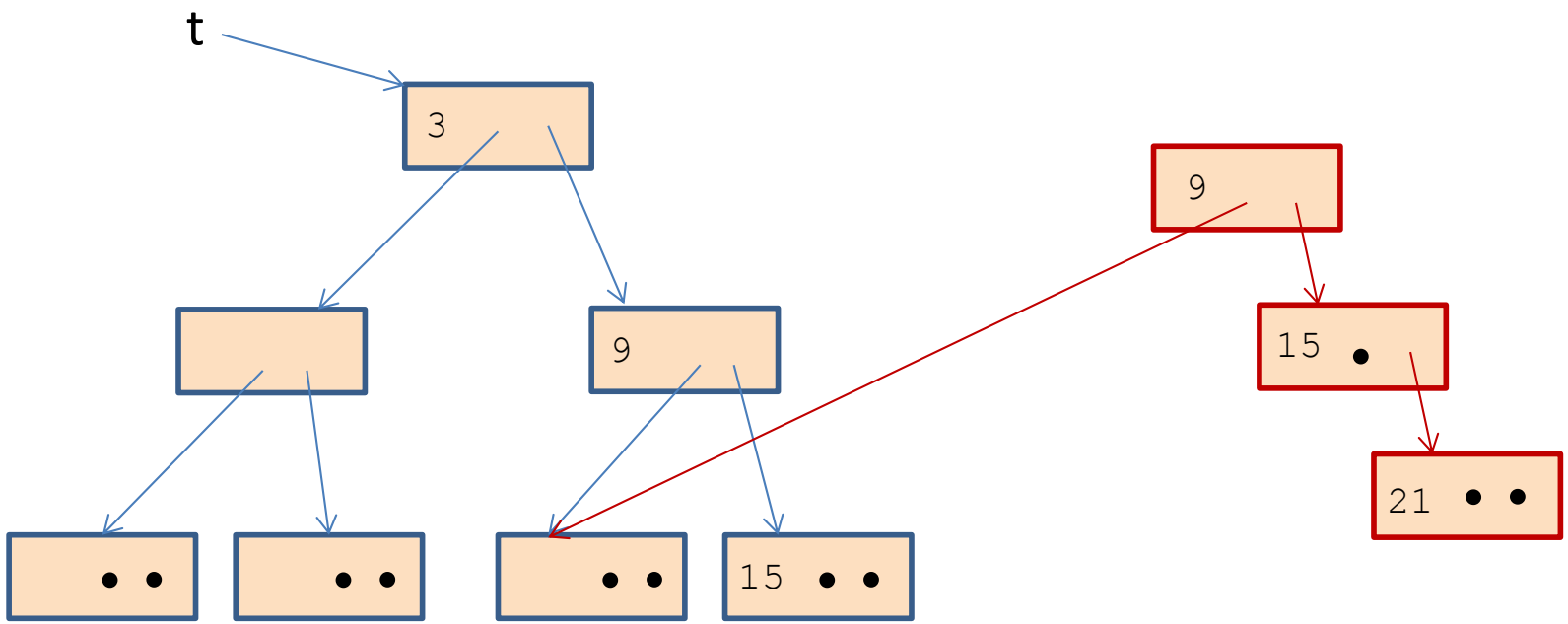
Allocating space

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



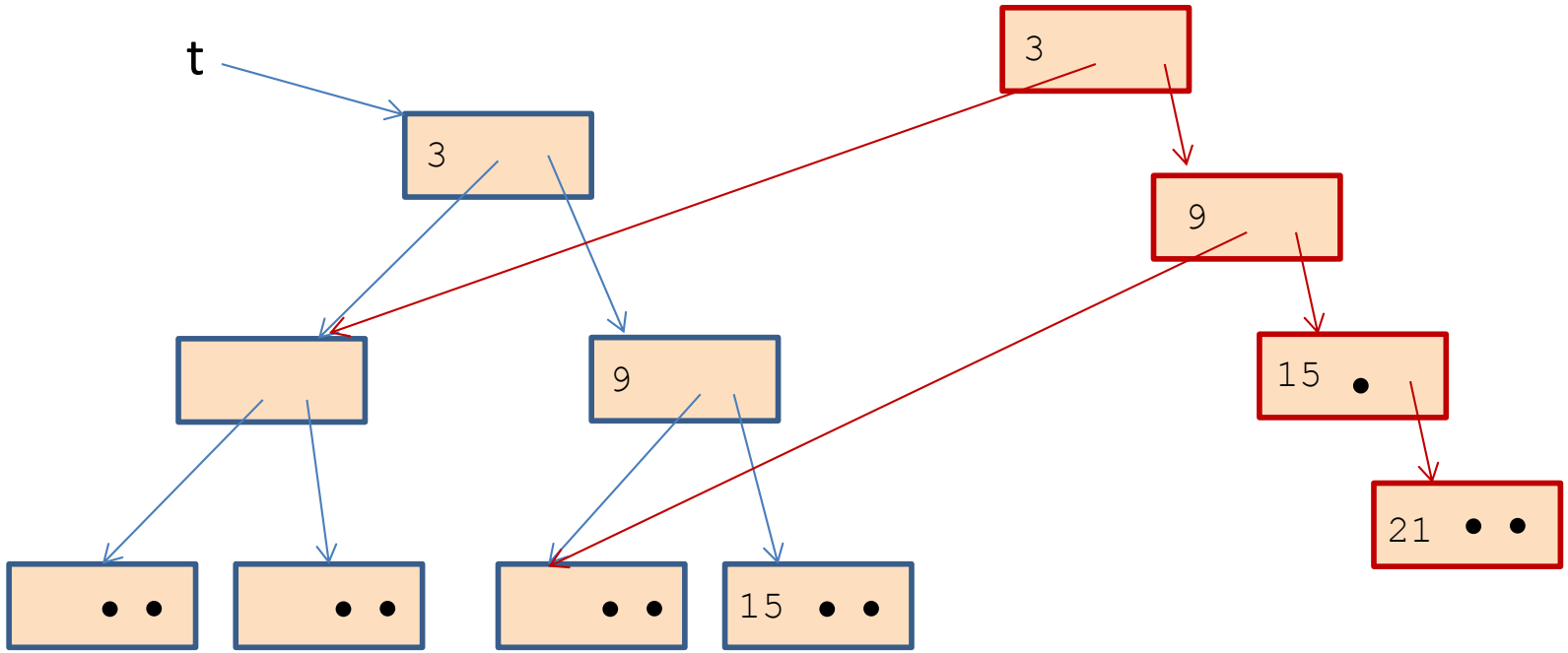
Allocating space

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



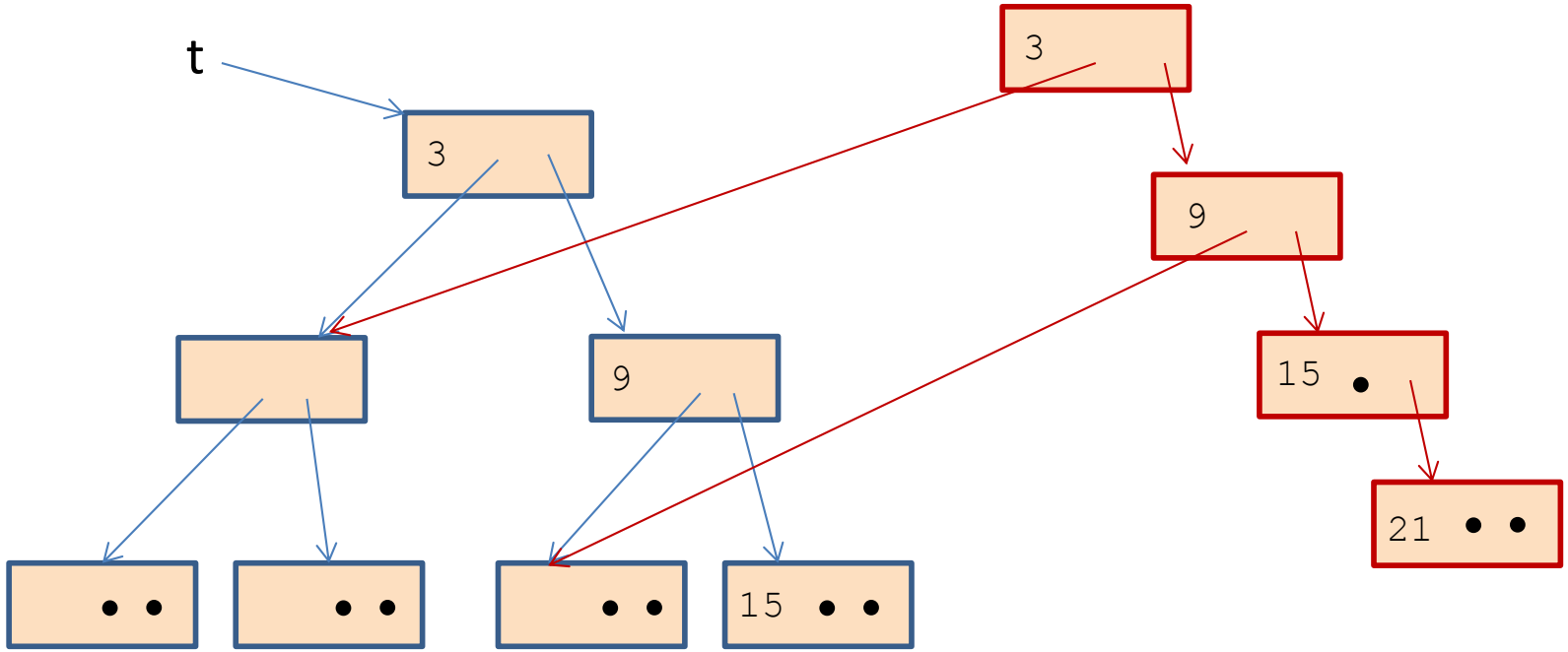
Allocating space

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)
```

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

~ log n, if tree with n nodes is balanced

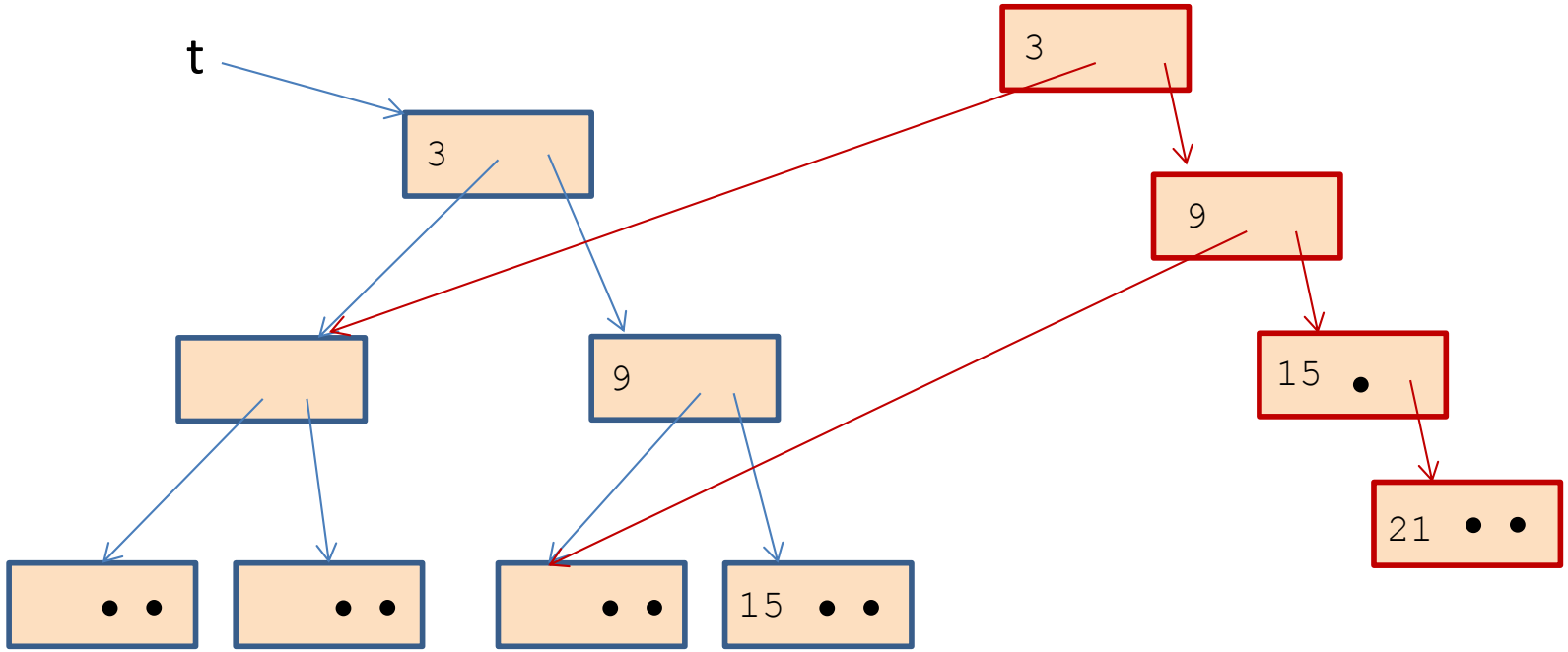


Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

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

John McCarthy invented g.c. 1960

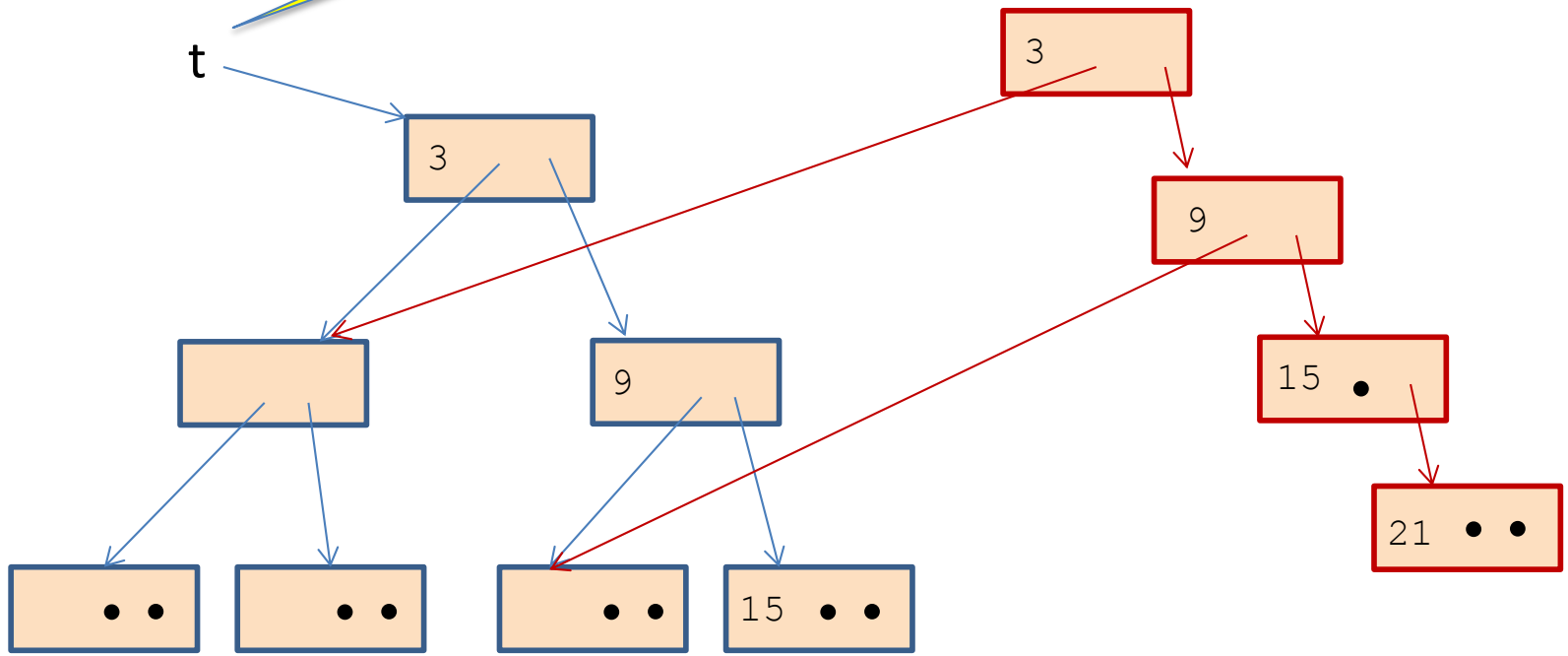


Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

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

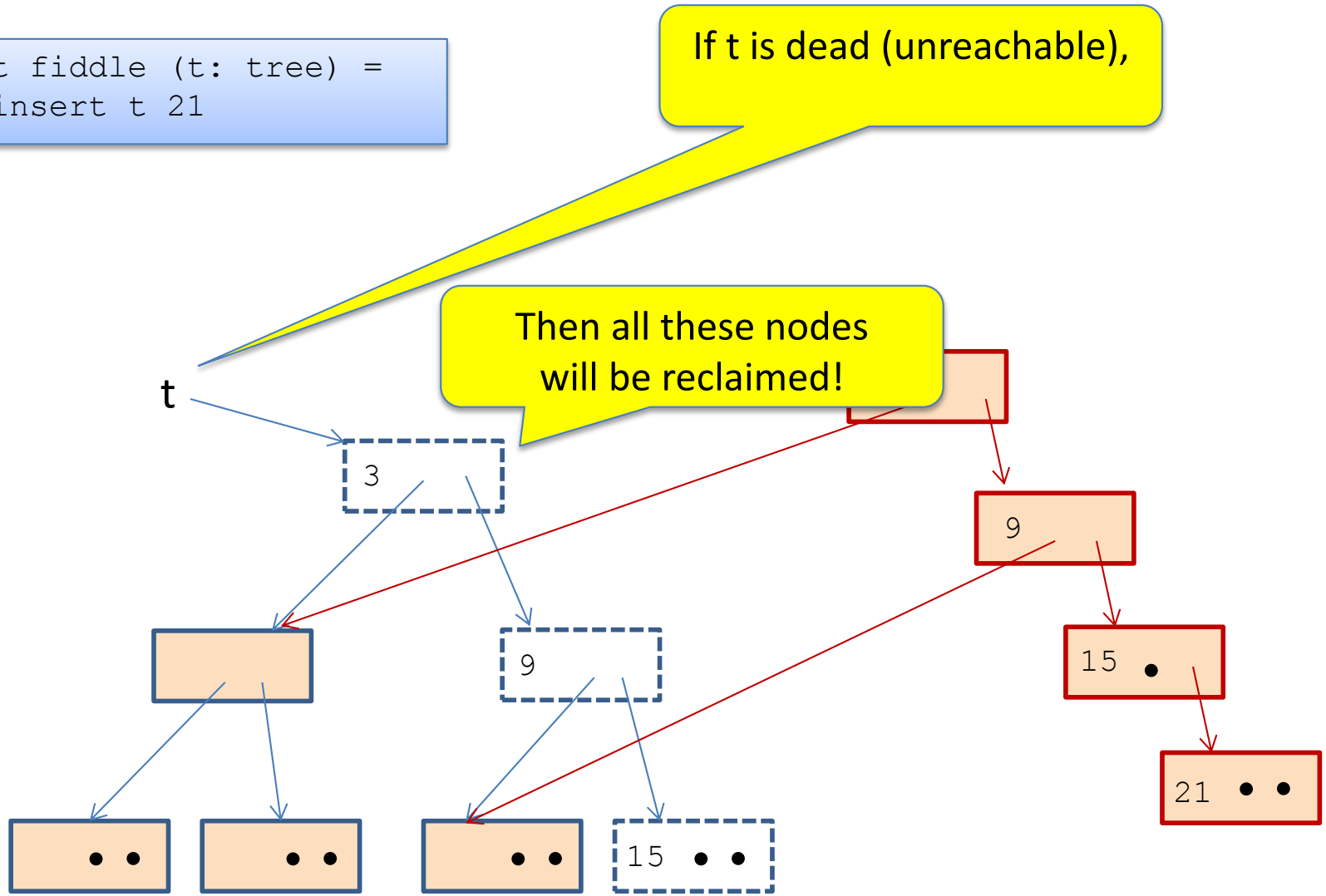
If t is dead (unreachable),



Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

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



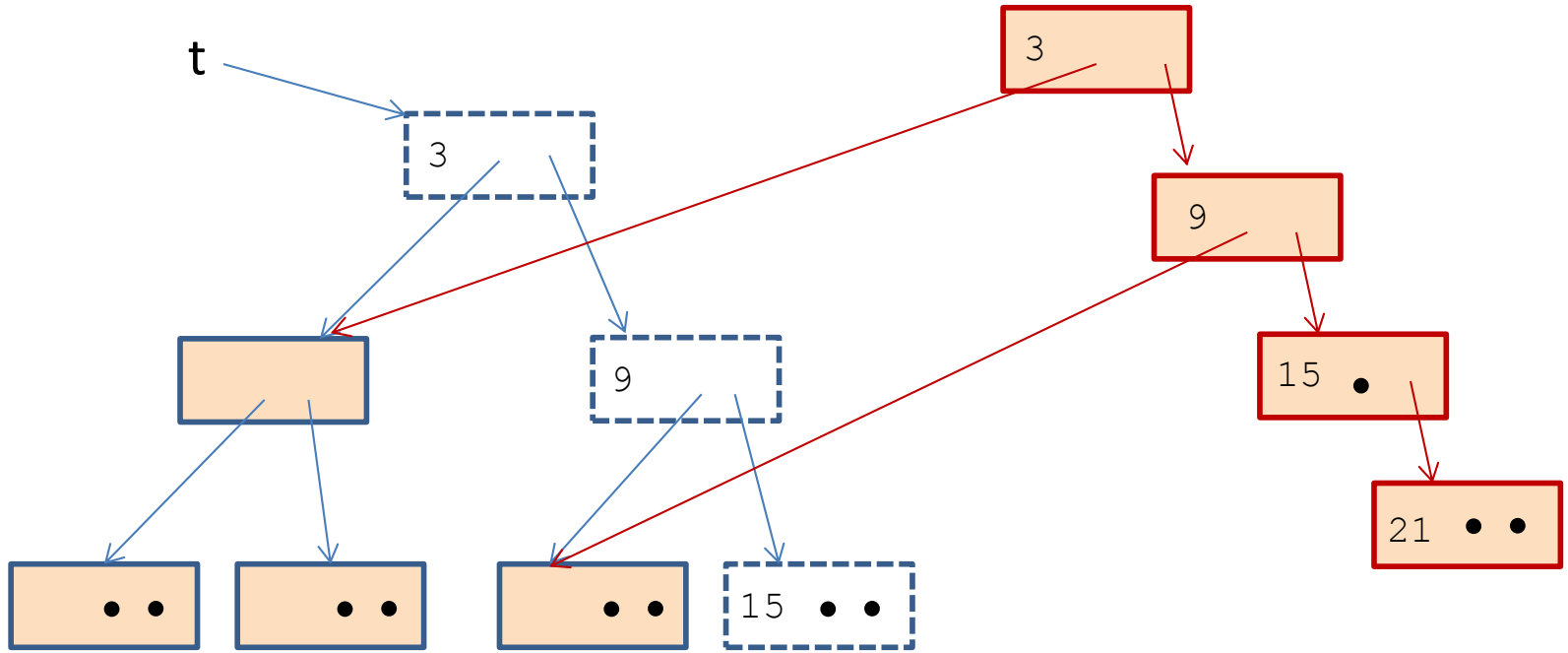
Net space allocated

The garbage collector reclaims unreachable data structures on the heap.

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

Net new space allocated:
1 node

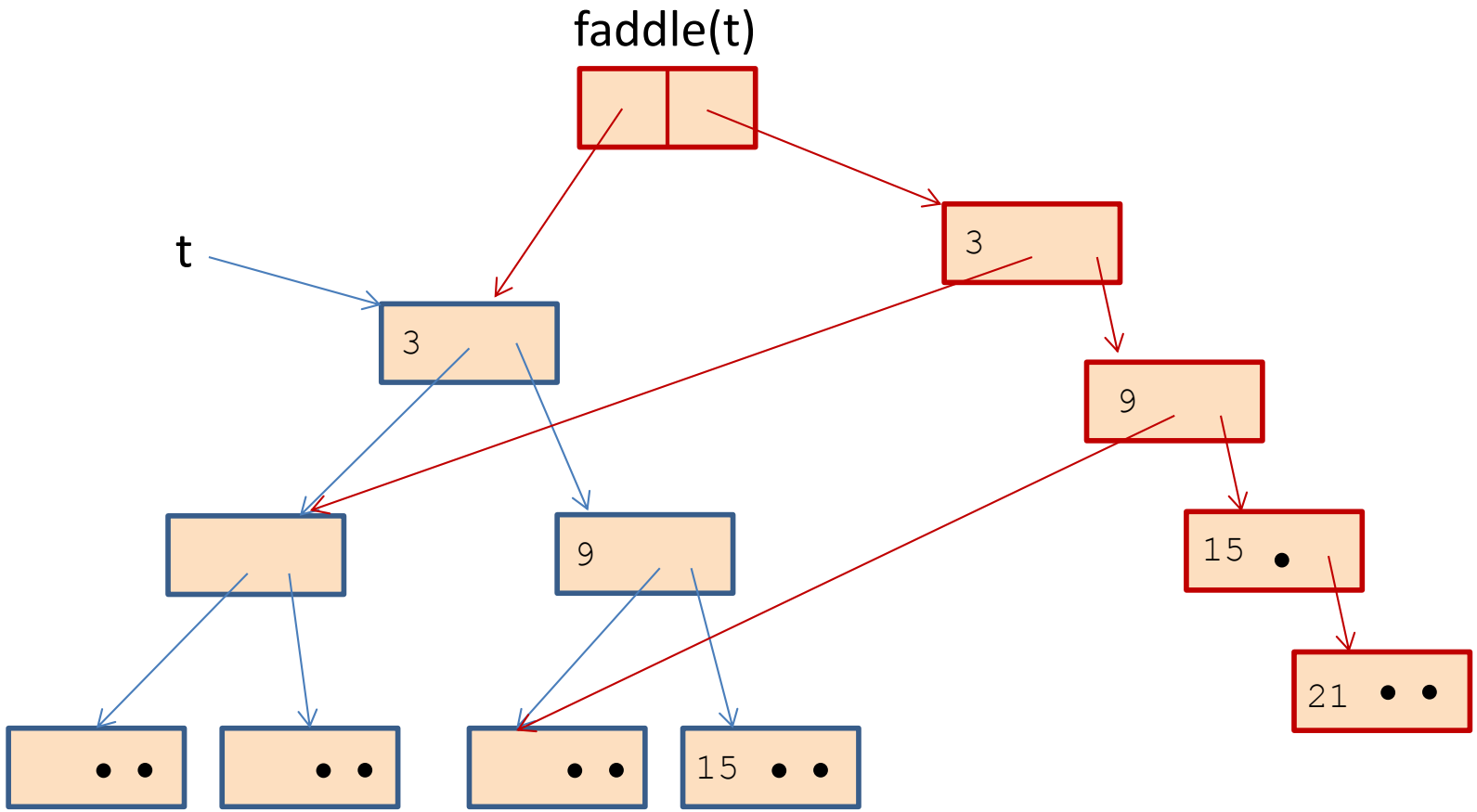
(just like “imperative” version of binary search trees)



Net space allocated

But what if you want to keep the old tree?

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



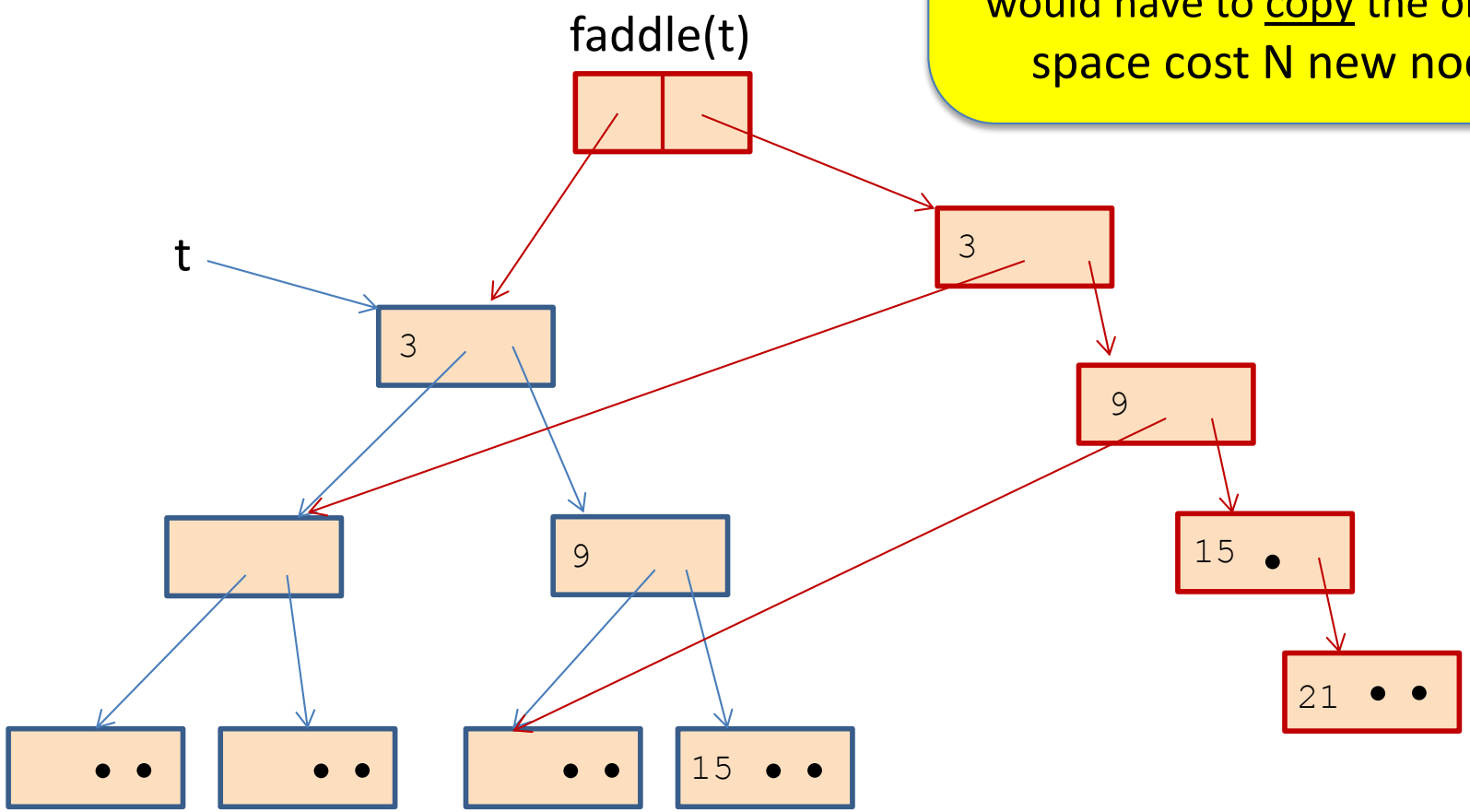
Net space allocated

But what if you want to keep the old tree?

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

Net new space allocated:
log(N) nodes

but note: "imperative" version would have to copy the old tree,
space cost N new nodes!



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
;;
```

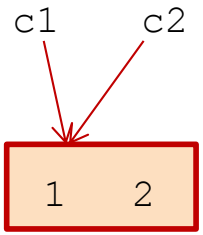
```
let double (c1:int*int) : int*int =
  cadd c1 c1
;;
```

```
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)
;;
```

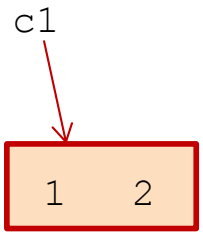
Compare

```
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)
;;
```



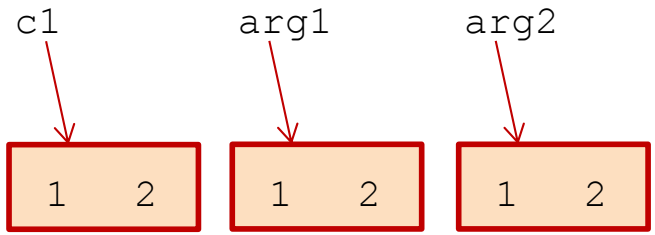
Compare

```
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
(unless the compiler happens to optimize…)

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);;
```

Its 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);;
```

Its execution behavior according to the substitution model:

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

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);;
```

Its execution behavior according to the substitution model:

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

Closures

Consider the following program:

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

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

Its execution behavior according to the substitution model:

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


```
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 the relationship between inputs and outputs of a function but it doesn't tell us much about the resources that are used along the way.

I'm going to tell you a little bit about how ML programs are compiled so you can understand how much space your programs will use. Understanding the space consumption of your programs is an important component in making these programs more efficient.

Compiling functions

General tactic: Reduce the problem of compiling ML-like functions to the problem of compiling C-like functions.

Some functions are already C-like:

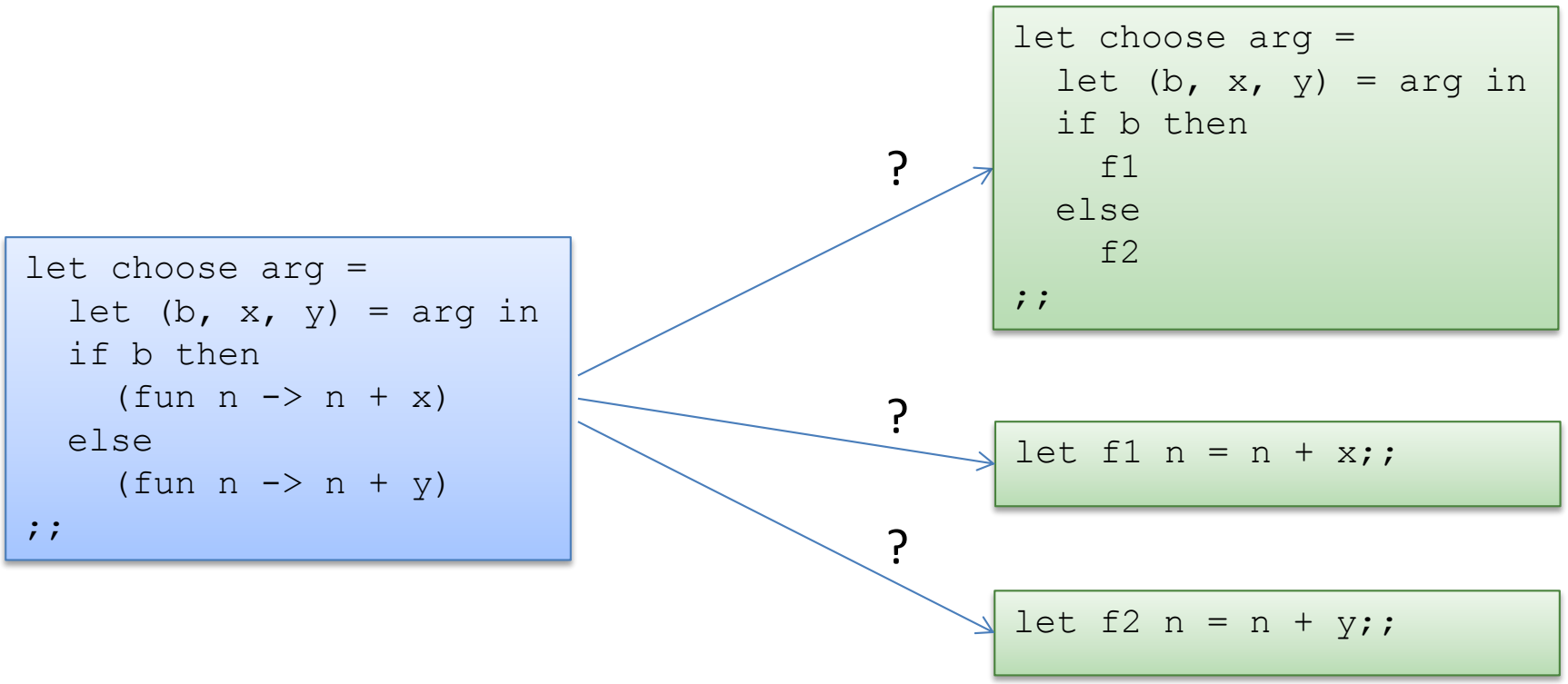
```
let add (x:int*int) : int =
  let (y,z) = x in
  y + z
;;
```



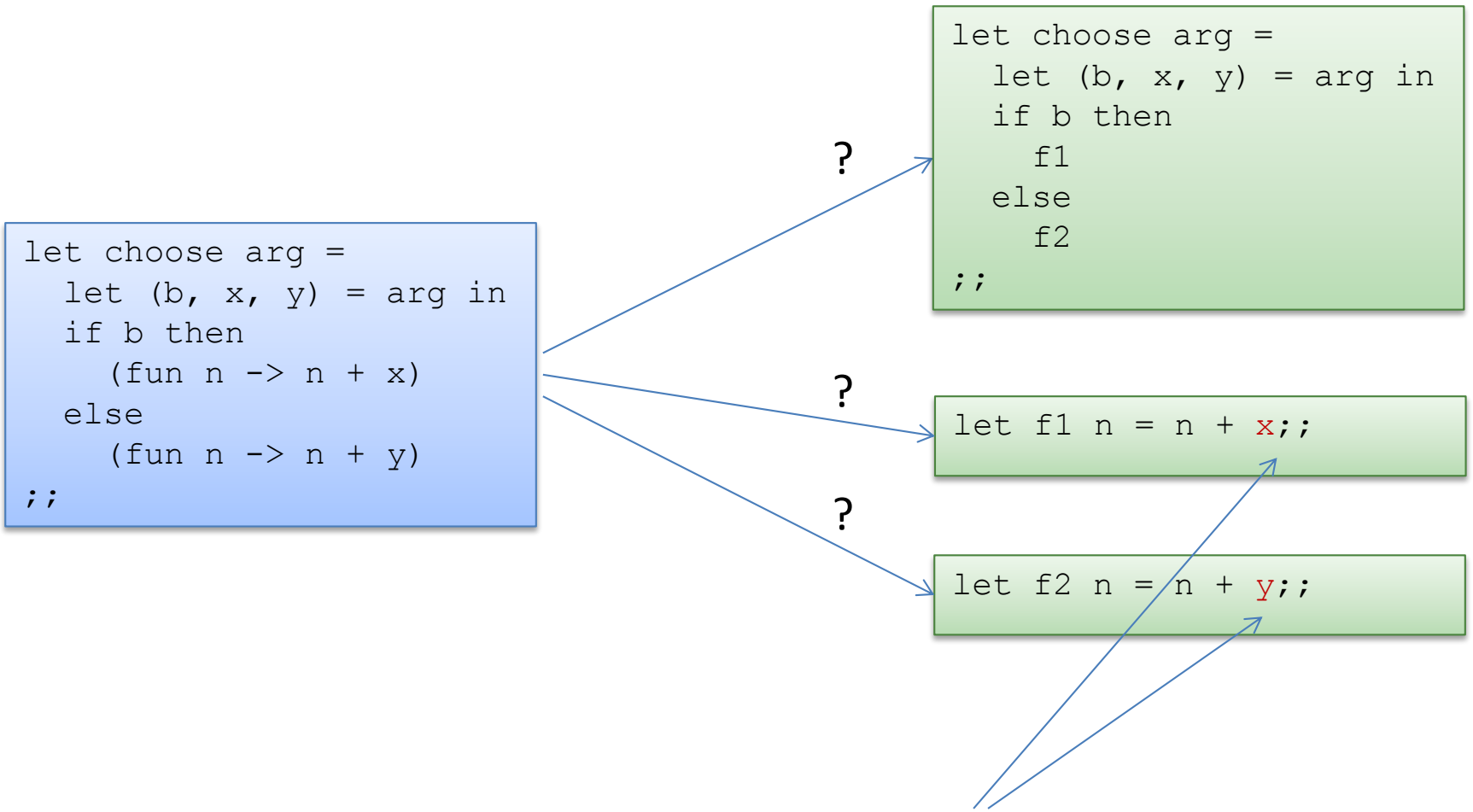
```
# argument in r1
# return address in r0

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

But what about nested, higher-order functions?



But what about nested, higher-order functions?



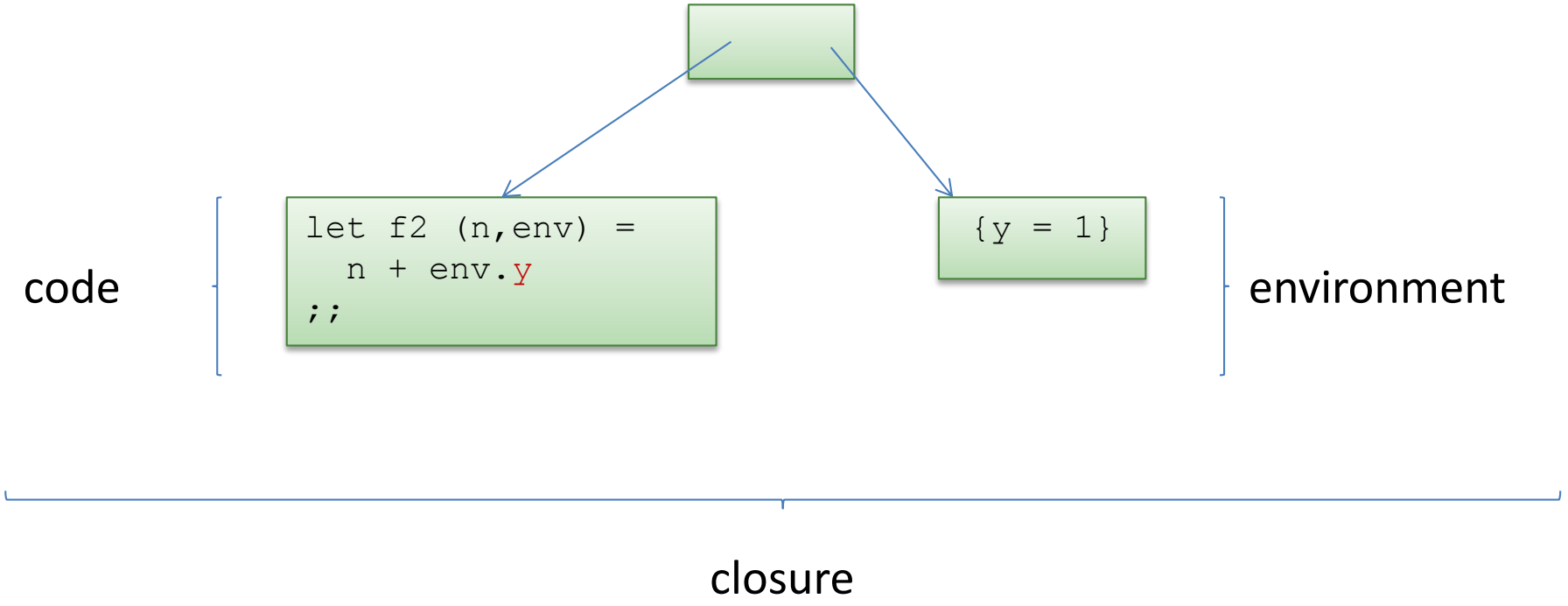
Darn! *Doesn't work naively*. Nested functions contain *free variables*.
Simple unnesting leaves them undefined.

But what about nested, higher-order functions?

We can't execute a function like the following:

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

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



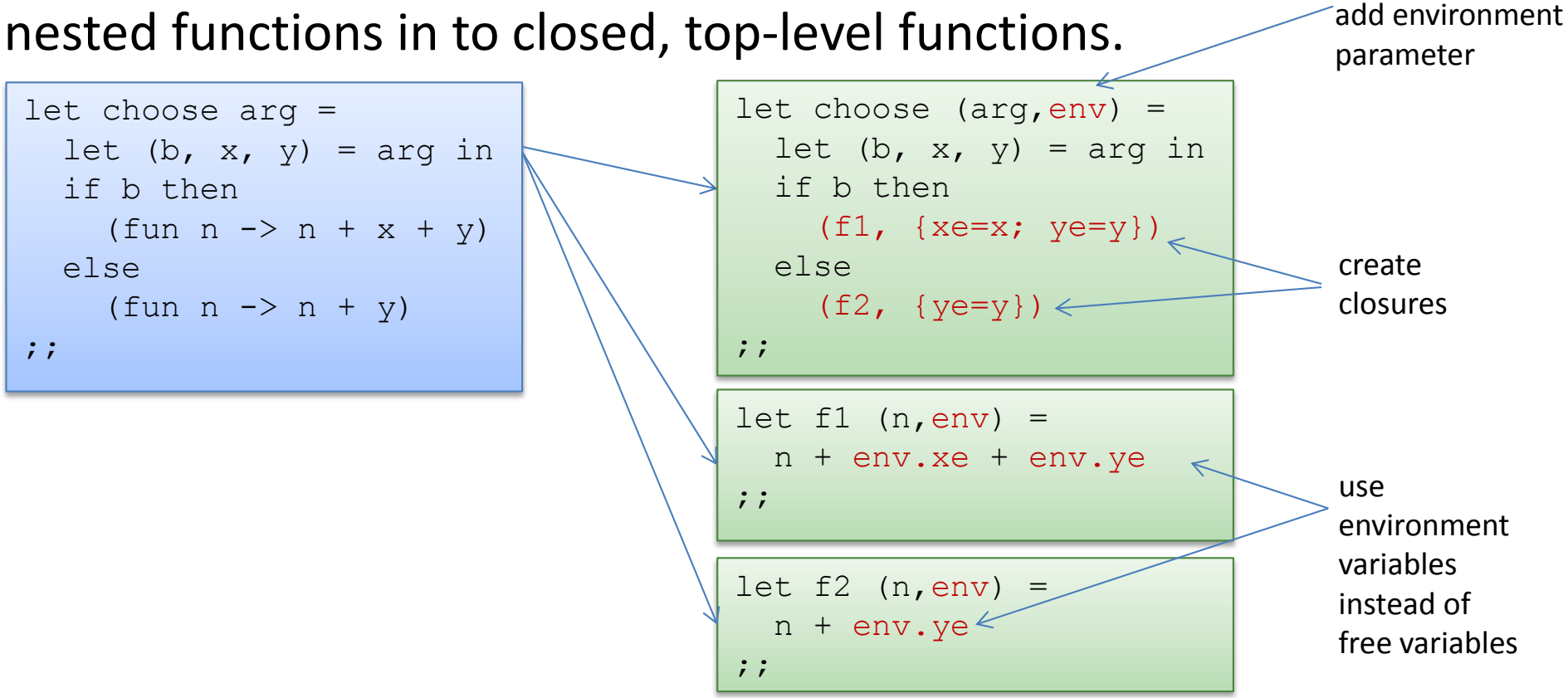
Closure Conversion

Closure conversion converts open, nested functions into closed, top-level functions.

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

Closure Conversion

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



Closure Conversion

Closure conversion 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
;;
```

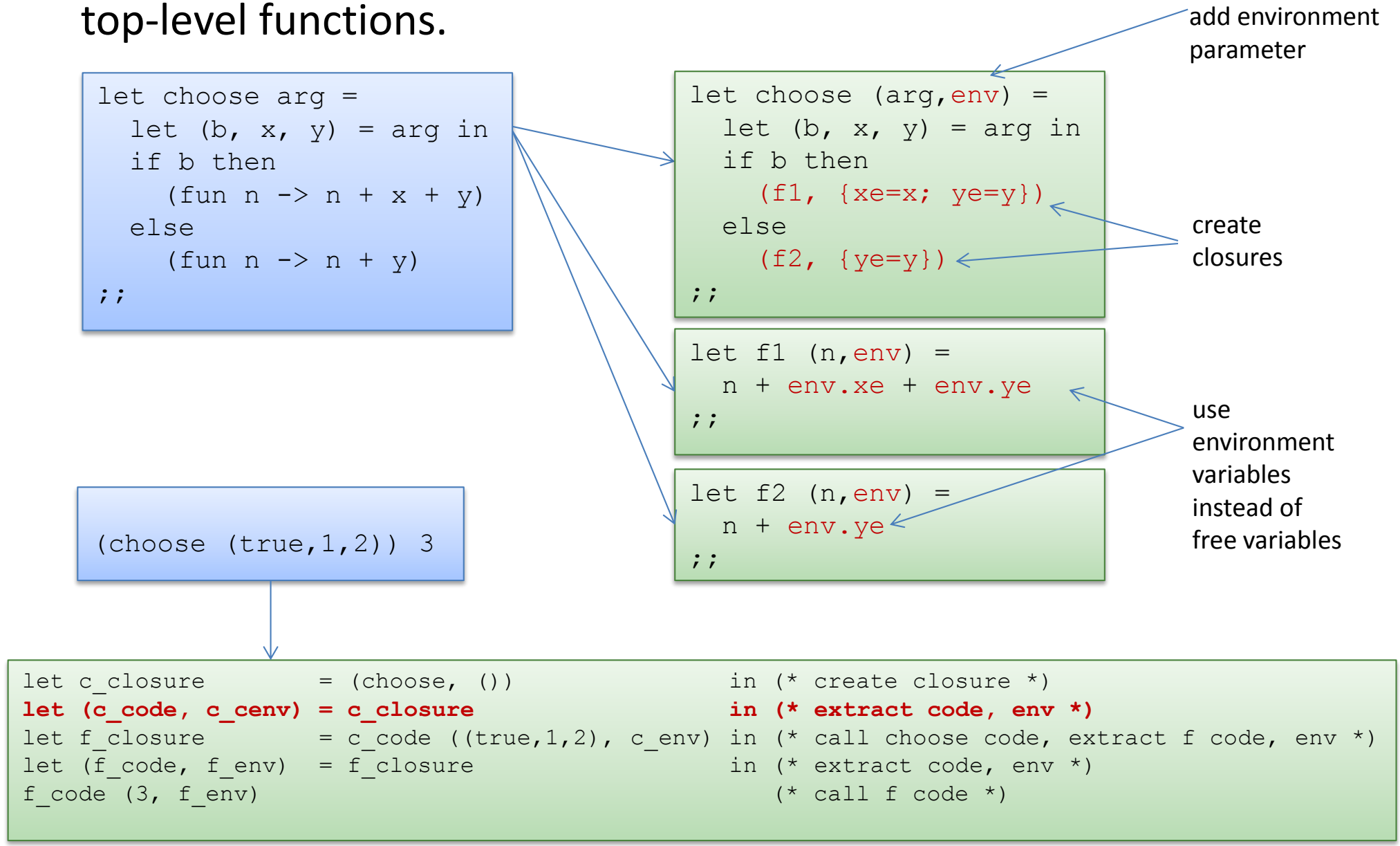
use environment variables instead of free variables

```
(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 converts open, nested functions in to closed, top-level functions.



Closure Conversion

Closure conversion 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 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 *)
```

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 f2_env = {y2:int}
```

```
type f1_clos = (int * f1_env -> int) * f1_env
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 {xe=x; ye=y})
  else
    (f2, F2 {ye=y})
;;
```

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

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

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

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

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

This is the traditional solution, and it's not stupid. But can we do better?

One Extra Note: Typing

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

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

∃ '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 (2 words)

+ the cost of the data referred to by function free variables

(1 word for each free variable)