Generalizing your Induction Hypothesis

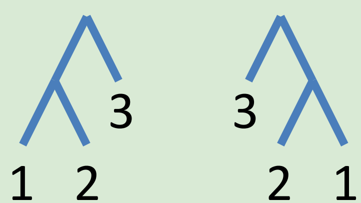
Andrew Appel
COS 326
Princeton University

A PROOF ABOUT TWO TREES

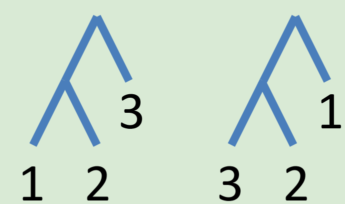
Image credit: pxfuel.com, licensed for free use

Reflection tester

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

mirror 

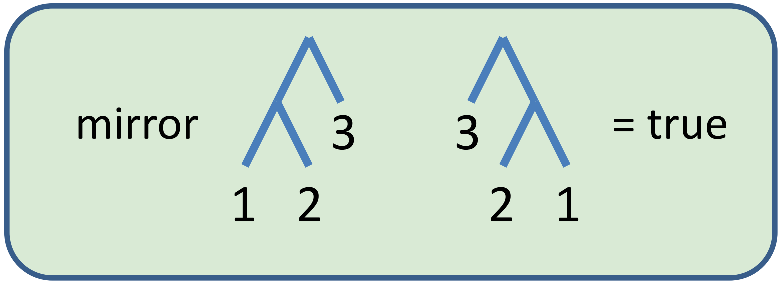
= true

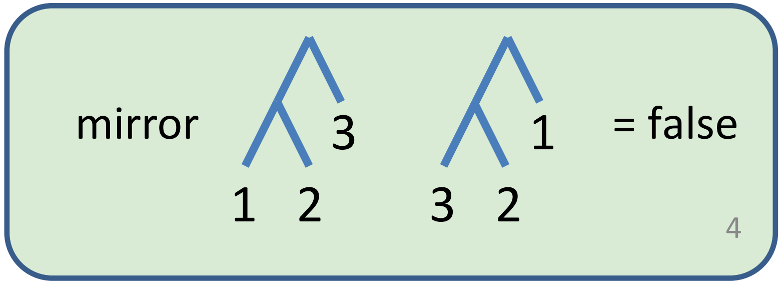
mirror 

= false

Reflection tester

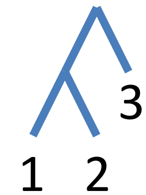
```
type tree = Leaf of int | Node of tree * tree

let rec mirror (t1: tree) (t2: tree) : bool =
  match t1 with
  | Leaf i -> (match t2 with
                | Leaf j -> i=j
                | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                   | Leaf _ -> false
                   | Node (b',a') -> mirror b b' && mirror a a')
```



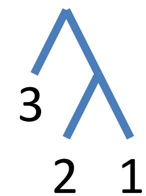


Examples

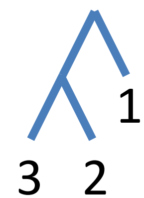
```
let foo = Node(Node(Leaf 1, Leaf 2), Leaf 3)
```




```
let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))
```



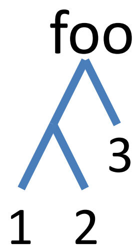

```
let baz = Node(Node(Leaf 3, Leaf 2), Leaf 1)
```

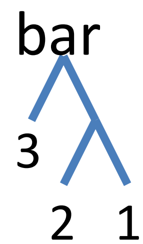


mirror foo bar = true

mirror foo baz = false

Claim!





Theorem: $\forall$ t:tree. mirror t bar = mirror bar t

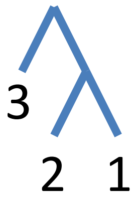
Examples:

mirror foo bar = true = mirror bar foo

mirror foo baz = false = mirror baz foo

Proof attempt 1

```
type tree = Leaf of int | Node of tree * tree
let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))
```



Theorem: ∀ t:tree. mirror t bar = mirror bar t

Proof:

By induction on t.

Case: t = Leaf i

mirror t bar

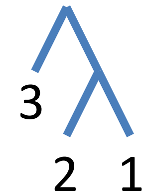
==

•
• *(we hope)*
•

== mirror bar t

Proof attempt 1

type tree = Leaf of int | Node of tree * tree
let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))



Theorem: ∀ t:tree. mirror t bar = mirror bar t

Proof:
By induction on t.
Case: t = Leaf i
mirror t bar
== mirror (Leaf i) bar
== match bar with Leaf j -> i=j | Node(_,_) -> false
== match Node(Leaf 3, Node(Leaf 2, Leaf 1)) with Leaf j -> i=j | Node(_,_) -> false
== false

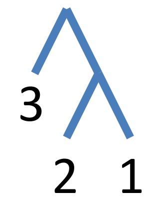
⋮ *(we hope)*

== mirror bar t

Proof attempt 1

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

let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))



Theorem: ∀ t:tree. mirror t bar = mirror bar t

Proof:

By induction on t.

Case: t = Leaf i

mirror t bar

== mirror (Leaf i) bar

== match bar with Leaf j -> i=j | Node(_,_) -> false

== match Node(Leaf 3, Node(Leaf 2, Leaf 1)) with Leaf j -> i=j | Node(_,_) -> false

== false

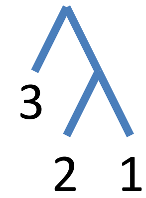
⋮ *(we hope)*

== mirror (Node(Leaf 3,Node(Leaf 2, Leaf 1))) (Leaf i)

== mirror bar t

Proof attempt 1

```
type tree = Leaf of int | Node of tree * tree
let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))
```



Theorem: ∀ t:tree. mirror t bar = mirror bar t

Proof:
By induction on t.
Case: t = Leaf i

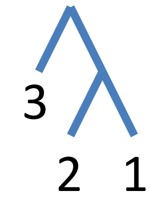
```
mirror t bar
== mirror (Leaf i) bar
== match bar with Leaf j -> i=j | Node(_,_) -> false
== match Node(Leaf 3, Node(Leaf 2, Leaf 1)) with Leaf j -> i=j | Node(_,_) -> false
== false
== false
== mirror (Node(Leaf 3,Node(Leaf 2, Leaf 1))) (Leaf i)
== mirror bar t
```

Done with this case!

Proof attempt 1

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

let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))



Theorem: ∀ t:tree. mirror t bar = mirror bar t

Case: t = Node(a,b)

mirror t bar

== mirror (Node (a,b)) bar

Where a and b satisfy I.H.,
mirror a bar = mirror bar a
mirror b bar = mirror bar b

•
• *(we hope)*
•

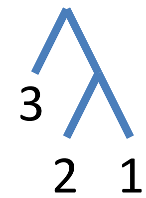
== mirror bar t

```
let rec mirror (t1: tree) (t2: tree) : bool =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                     mirror b b' && mirror a a')
```

Proof attempt 1

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

let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))



Theorem: ∀ t:tree. mirror t bar = mirror bar t

Case: t = Node(a,b)

mirror t bar

== mirror (Node (a,b)) bar

== match bar with Leaf _ -> false | Node(b’,a’) -> mirror b b’ && mirror a a’

•
• *(we hope)*
•

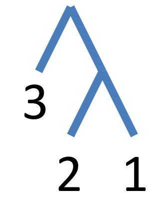
== mirror bar t

```
let rec mirror (t1: tree) (t2: tree) : bool =
 match t1 with
 | Leaf i -> (match t2 with
              | Leaf j -> i=j
              | Node(_,_) -> false)
 | Node(a,b) -> (match t2 with
                 | Leaf _ -> false
                 | Node (b',a') ->
                   mirror b b' && mirror a a')
```

Proof attempt 1

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

let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))



Theorem: ∀ t:tree. mirror t bar = mirror bar t

Case: t = Node(a,b)

mirror t bar

== mirror (Node (a,b)) bar

== match bar with Leaf _ -> false | Node(b’,a’) -> mirror b b’ && mirror a a’

== mirror b (Leaf 3) && mirror a (Node(Leaf 2, Leaf 1))

⋮ *(we hope)*

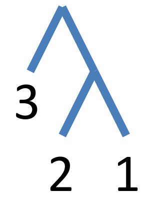
== mirror bar t

```
let rec mirror (t1: tree) (t2: tree) : bool =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                    mirror b b' && mirror a a')
```

Proof attempt 1

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

let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))



Theorem: ∀ t:tree. mirror t bar = mirror bar t

Case: t = Node(a,b)

mirror t bar
== mirror (Node (a,b)) bar
== match bar with Leaf _ -> false | Node(b',a') -> mirror b b' && mirror a a'
== mirror b (Leaf 3) && mirror a (Node(Leaf 2, Leaf 1))

⋮ *(we hope)*

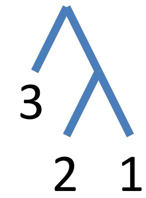
== mirror (Node(Leaf 2, Leaf 1)) a && mirror (Leaf 3) b
== mirror (Node(Leaf 3, Node(_,_))) (Node(a,b))
== mirror bar t

```
let rec mirror t1 t2 =
  match t1 with
  | Leaf i -> (match t2 with
              | Leaf j -> i=j
              | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                 | Leaf _ -> false
                 | Node (b',a') ->
                   mirror b b' &&
                   mirror a a')
```

Proof attempt 1

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

let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))



Theorem: ∀ t:tree. mirror t bar = mirror bar t

Case: t = Node(a,b)

mirror t bar
== mirror (Node (a,b)) bar
== match bar with Leaf _ -> false | Node(b',a') -> mirror b b' && mirror a a'
== mirror b (Leaf 3) && mirror a (Node(Leaf 2, Leaf 1))
== mirror a (Node(Leaf 2, Leaf 1)) && mirror b (Leaf 3)

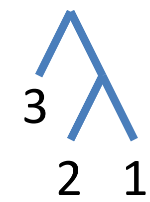
⋮ *(we hope)*

== mirror (Node(Leaf 2, Leaf 1)) a && mirror (Leaf 3) b
== mirror (Node(Leaf 3, Node(_,_))) (Node(a,b))
== mirror bar t

Proof attempt 1

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

let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))



Theorem: ∀ t:tree. mirror t bar = mirror bar t

Case: t = Node(a,b)

mirror t bar

== mirror (Node (a,b)) bar

== match bar with Leaf _ -> false | Node(b',a') -> mirror b b' && mirror a a'

== mirror b (Leaf 3) && mirror a (Node(Leaf 2, Leaf 1))

== mirror a (Node(Leaf 2, Leaf 1)) && mirror b (Leaf 3)

== mirror (Node(Leaf 2, Leaf 1)) a && mirror (Leaf 3) b

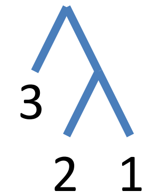
== mirror (Node(Leaf 3, Node(_,_))) (Node(a,b))

== mirror bar t

FAIL!

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

let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))



Theorem: ∀ t:tree, mirror t bar = mirror bar t

Case: t = Node(a,b)

mirror t bar

== mirror (Node (a,b)) bar

== match bar with Leaf _ -> false | Node(b',a') -> mirror b b' && mirror a a'

== mirror b (Leaf 3) && mirror a (Node(Leaf 2, Leaf 1))

== mirror a (Node(Leaf 2, Leaf 1)) && mirror b (Leaf 3)

Induction hyp tells us:
mirror a bar = mirror bar a
mirror b bar = mirror bar b

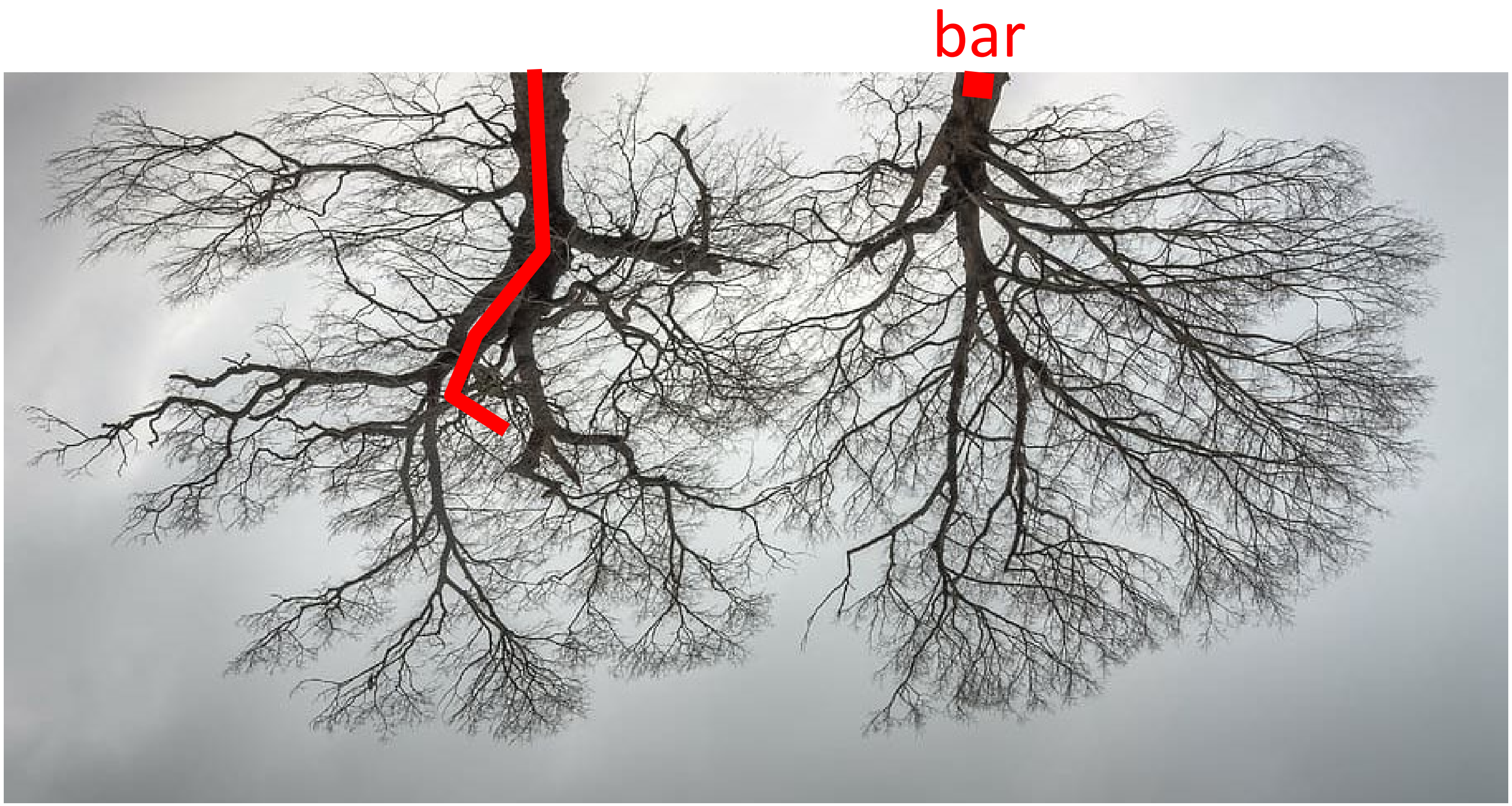
== mirror (Node(Leaf 2, Leaf 1)) a && mirror (Leaf 3) b

== mirror (Node(Leaf 3, Node(_,_))) (Node(a,b))

== mirror bar t

What's the problem?

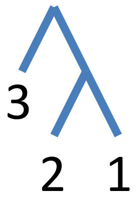
What's the problem?



Solution: prove a more general theorem!

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

let bar = Node(Leaf 3, Node(Leaf 2, Leaf 1))



Theorem: ∀ t:tree, mirror t bar = mirror bar t

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof!

Theorem: $\forall$ t:tree. $\forall$ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: $\forall$ u:tree. mirror t u = mirror u t

Proof!

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume an arbitrary u about which we know nothing (except its type, “tree”)

Proof!

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Need to prove: mirror t u = mirror u t

Proof!

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Leaf i
  Need to prove: ∀ u:tree. mirror t u = mirror u t
  Assume u: tree.
    mirror t u
 == mirror (Leaf i) u

•
•
•

 == mirror u t

```
let rec mirror t1 t2  =
 match t1 with
 | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
 | Node(a,b) -> (match t2 with
                   | Leaf _ -> false
                   | Node (b',a') ->
                      mirror b b' &&
                      mirror a a')
```

Proof!

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Leaf i
  Need to prove: ∀ u:tree. mirror t u = mirror u t
  Assume u: tree.
     mirror t u
 == mirror (Leaf i) u
 == match u with Leaf j -> i=j | Node(_,_) -> false

⋮

 == mirror u t

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                     mirror b b' &&
                     mirror a a')
```

Now, need case analysis on u

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Leaf i
  Need to prove: ∀ u:tree. mirror t u = mirror u t
  Assume u: tree.
      mirror t u
  == mirror (Leaf i) u
  == match u with Leaf j -> i=j | Node(_,_) -> false

⋮

  == mirror u t

```
let rec mirror t1 t2 =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                    mirror b b' &&
                    mirror a a')
```

Case analysis on u: first subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Leaf i
Need to prove: ∀ u:tree. mirror t u = mirror u t
Assume u: tree.
Subcase: u = Leaf j
mirror t u
== mirror (Leaf i) u
== match u with Leaf j -> i=j | Node(_,_) -> false

•
•
•

== mirror u t

```
let rec mirror t1 t2 =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                     mirror b b' &&
                     mirror a a')
```

Case analysis on u: first subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Leaf i
  Need to prove: ∀ u:tree. mirror t u = mirror u t
  Assume u: tree.
  Subcase: u = Leaf j

```
    mirror t u
 == mirror (Leaf i) u
 == match u with Leaf j -> i=j | Node(_,_) -> false
 == (i=j)
    •
    •
    •
 == mirror u t
```

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                    mirror b b' &&
                    mirror a a')
```

Case analysis on u: first subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Leaf i
  Need to prove: ∀ u:tree. mirror t u = mirror u t
  Assume u: tree.
  Subcase: u = Leaf j

```
     mirror t u
  == mirror (Leaf i) u
  == match u with Leaf j -> i=j | Node(_,_) -> false
  == (i=j)
   ⋮
  == mirror (Leaf j) (Leaf i)
  == mirror u t
```

```
let rec mirror t1 t2  =
 match t1 with
  | Leaf i -> (match t2 with
                | Leaf j -> i=j
                | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                    | Leaf _ -> false
                    | Node (b',a') ->
                      mirror b b' &&
                      mirror a a')
```

First subcase done

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Leaf j

```
    mirror t u
== mirror (Leaf i) u
== match u with Leaf j -> i=j | Node(_,_) -> false
== (i=j)
== (j=i)
== mirror (Leaf j) (Leaf i)
== mirror u t
```

Done with Subcase (u=Leaf j).

```
let rec mirror t1 t2 =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                    mirror b b' &&
                    mirror a a')
```

Case analysis on u: **second** subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Node(g,h)

mirror t u

==

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                   | Leaf _ -> false
                   | Node (b',a') ->
                      mirror b b' &&
                      mirror a a')
```

Case analysis on u: second subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Node(g,h)

mirror t u

== mirror (Leaf i) (Node(g,h))

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                    mirror b b' &&
                    mirror a a')
```

Case analysis on u: second subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Node(g,h)

mirror t u

== mirror (Leaf i) (Node(g,h))

== false

•
•
•

== mirror u t

```
let rec mirror t1 t2 =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                    mirror b b' &&
                    mirror a a')
```

Case analysis on u: second subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Node(g,h)

```
    mirror t u
 == mirror (Leaf i) (Node(g,h))
 == false
    •
    •
    •
 == mirror (Node(g,h) (Leaf i)
 == mirror u t
```

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
                | Leaf j -> i=j
                | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                   | Leaf _ -> false
                   | Node (b',a') ->
                     mirror b b' &&
                     mirror a a')
```

Case analysis on u: second subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Node(g,h)

```
    mirror t u
 == mirror (Leaf i) (Node(g,h))
 == false

 == false
 == mirror (Node(g,h) (Leaf i)
 == mirror u t
```

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                     mirror b b' &&
                     mirror a a')
```

Case analysis on u: second subcase done.

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Node(g,h)

```
    mirror t u
 == mirror (Leaf i) (Node(g,h))
 == false
 == mirror (Node(g,h) (Leaf i)
 == mirror u t
```

Done with Subcase (u=Node(g,h)).

Done with Case (t=Leaf i).

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                     mirror b b' &&
                     mirror a a')
```

Case analysis on t: second case

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
                 | Leaf j -> i=j
                 | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                    | Leaf _ -> false
                    | Node (b',a') ->
                       mirror b b' &&
                       mirror a a')
```

Case analysis on t: second case

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Need to prove: mirror t u = mirror u t

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
                   | Leaf j -> i=j
                   | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                          | Leaf _ -> false
                          | Node (b',a') ->
                            mirror b b' &&
                            mirror a a')
```

Case analysis on u: first subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

  Need to prove: ∀ u:tree. mirror t u = mirror u t

  Assume u: tree.

  Subcase: u = Leaf i.

    mirror t u

  ==

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                     mirror b b' &&
                     mirror a a')
```

Case analysis on u: first subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Leaf i.

mirror t u

== mirror (Node(a,b)) (Leaf i)

```
let rec mirror t1 t2  =
 match t1 with
  | Leaf i -> (match t2 with
                 | Leaf j -> i=j
                 | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                        | Leaf _ -> false
                        | Node (b',a') ->
                          mirror b b' &&
                          mirror a a')
```

Case analysis on u: first subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Leaf i.

mirror t u

== mirror (Node(a,b)) (Leaf i)

== false

```
let rec mirror t1 t2 =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                    mirror b b' &&
                    mirror a a')
```

Case analysis on u: first subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Leaf i.

mirror t u

== mirror (Node(a,b)) (Leaf i)

== false

== mirror (Leaf i) (Node(a,b))

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
              | Leaf j -> i=j
              | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                 | Leaf _ -> false
                 | Node (b',a') ->
                   mirror b b' &&
                   mirror a a')
```

Case analysis on u: first subcase done.

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Leaf i.

mirror t u

== mirror (Node(a,b)) (Leaf i)

== false

== mirror (Leaf i) (Node(a,b))

== mirror u t

Done with Subcase (u=Leaf i).

```
let rec mirror t1 t2  =
 match t1 with
 | Leaf i -> (match t2 with
              | Leaf j -> i=j
              | Node(_,_) -> false)
 | Node(a,b) -> (match t2 with
                 | Leaf _ -> false
                 | Node (b',a') ->
                   mirror b b' &&
                   mirror a a')
```

Case analysis on u: second subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

  Need to prove: ∀ u:tree. mirror t u = mirror u t

  Assume u: tree.

  Subcase: u = Node(g,h).

     mirror t u

  ==

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                     mirror b b' &&
                     mirror a a')
```

Case analysis on u: second subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Node(g,h).

mirror t u

== mirror (Node(a,b)) (Node(g,h))

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                    mirror b b' &&
                    mirror a a')
```

Case analysis on u: second subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Node(g,h).

mirror t u

== mirror (Node(a,b)) (Node(g,h))

== mirror b h && mirror a g

```
let rec mirror t1 t2 =
  match t1 with
  | Leaf i -> (match t2 with
                 | Leaf j -> i=j
                 | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                     | Leaf _ -> false
                     | Node (b',a') ->
                       mirror b b' &&
                       mirror a a')
```

Case analysis on u: second subcase

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Node(a,b)
  Need to prove: ∀ u:tree. mirror t u = mirror u t
  Assume u: tree.
  Subcase: u = Node(g,h).
      mirror t u
  == mirror (Node(a,b)) (Node(g,h))
  == mirror b h && mirror a g
  == mirror a b && mirror b h

```
let rec mirror t1 t2 =
  match t1 with
  | Leaf i -> (match t2 with
                | Leaf j -> i=j
                | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                   | Leaf _ -> false
                   | Node (b',a') ->
                     mirror b b' &&
                     mirror a a')
```

What does the induction hypothesis tell us?

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Node(g,h).

mirror t u

== mirror (Node(a,b)) (Node(g,h))

== mirror b h && mirror a g

== mirror a b && mirror b h

Induction hyp tells us:

∀ u:tree. mirror a u = mirror u a

and

∀ u:tree. mirror b u = mirror u b

Why? Because a and b are the immediate subtrees of t

What does the induction hypothesis tell us?

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Node(a,b)
  Need to prove: ∀ u:tree. mirror t u = mirror u t
  Assume u: tree.
  Subcase: u = Node(g,h).
    mirror t u
  == mirror (Node(a,b)) (Node(g,h))
  == mirror b h && mirror a g
  == mirror a b && mirror b h
  == mirror b a && mirror b h

Induction hyp tells us:
∀ u:tree. mirror a u = mirror u a
  and
∀ u:tree. mirror b u = mirror u b

What does the induction hypothesis tell us?

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Node(a,b)
  Need to prove: ∀ u:tree. mirror t u = mirror u t
  Assume u: tree.
  Subcase: u = Node(g,h).
    mirror t u
  == mirror (Node(a,b)) (Node(g,h))
  == mirror b h && mirror a g
  == mirror a b && mirror b h
  == mirror b a && mirror b h
  == mirror b a && mirror h b

Induction hyp tells us:
∀ u:tree. mirror a u = mirror u a
  and
∀ u:tree. mirror b u = mirror u b

Finishing the proof

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Node(a,b)
  Need to prove: ∀ u:tree. mirror t u = mirror u t
  Assume u: tree.
  Subcase: u = Node(g,h).

```
    mirror t u
== mirror (Node(a,b)) (Node(g,h))
== mirror b h && mirror a g
== mirror a g && mirror b h
== mirror g a && mirror b h
== mirror g a && mirror h b
== mirror (Node(g,h)) (Node(a,b))
```

```
let rec mirror t1 t2 =
  match t1 with
  | Leaf i -> (match t2 with
               | Leaf j -> i=j
               | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                  | Leaf _ -> false
                  | Node (b',a') ->
                    mirror b b' &&
                    mirror a a')
```

Finishing the proof.

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Node(g,h).

```
    mirror t u
 == mirror (Node(a,b)) (Node(g,h))
 == mirror b h && mirror a g
 == mirror a g && mirror b h
 == mirror g a && mirror b h
 == mirror g a && mirror h b
 == mirror (Node(g,h)) (Node(a,b))
 == mirror u t
```

Done with Subcase (u=Node(g,h)),

Done with Case (t=Node(a,b))

```
let rec mirror t1 t2  =
  match t1 with
  | Leaf i -> (match t2 with
              | Leaf j -> i=j
              | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                 | Leaf _ -> false
                 | Node (b',a') ->
                   mirror b b' &&
                   mirror a a')
```

Finishing the proof.

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:
By induction on t.
Case: t = Node(a,b)
  Need to prove: ∀ u:tree. mirror t u = mirror u t
  Assume u: tree.
  Subcase: u = Node(g,h).
      mirror t u
  == mirror (Node(a,b)) (Node(g,h))
  == mirror b h && mirror a g
  == mirror a g && mirror b h
  == mirror g a && mirror b h
  == mirror g a && mirror h b
  == mirror (Node(g,h)) (Node(a,b))
  == mirror u t
Done with Subcase (u=Node(g,h)),
Done with Case (t=Node(a,b)
QED

```
let rec mirror t1 t2 =
  match t1 with
  | Leaf i -> (match t2 with
              | Leaf j -> i=j
              | Node(_,_) -> false)
  | Node(a,b) -> (match t2 with
                 | Leaf _ -> false
                 | Node (b',a') ->
                   mirror b b' &&
                   mirror a a')
```

Summary of the proof

Theorem: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof:

By induction on t.

Case: t = Leaf i

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Leaf j

mirror t u == . . . == mirror u t

Subcase: u = Node(g,h)

mirror t u == . . . == mirror u t

Case: t = Node(a,b)

Need to prove: ∀ u:tree. mirror t u = mirror u t

Assume u: tree.

Subcase: u = Leaf j

mirror t u == . . . == mirror u t

Subcase: u = Node(g,h)

mirror t u == . . . == mirror u t

QED

Our original proof goal

Theorem 1: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof . . . QED

Theorem 2: ∀ t:tree. mirror t bar = mirror bar t

Proof.

Assume t:tree.

Must prove: mirror t bar = mirror bar t.

Our original proof goal

Theorem 1: ∀ t:tree. ∀ u:tree. mirror t u = mirror u t

Proof . . . QED

Theorem 2: ∀ t:tree. mirror t bar = mirror bar t

Proof.
Assume t:tree.
  Must prove: mirror t bar = mirror bar t.
  Apply Theorem 1, instantiating variable t with t, instantiating u with bar.
QED.

Moral of the story:

WHEN PROVING BY INDUCTION, SOMETIMES YOU MUST GENERALIZE THE THEOREM

(OR ELSE THE INDUCTION HYPOTHESIS WON’T FIT)

Another example

```
let rec same (i: int) (j: int) : bool =
  if i=0 then j=0
  else j>0 && same (i-1) (j-1)
```

Claim: $\forall$ x:nat. same x 3 = same 3 x

Remark: x:nat means that x≥0

Examples:

same 3 3 = true = same 3 3

same 4 3 = false = same 3 4

Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem: $\forall$ x:nat.   same x 3 = same 3 x

Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem: ∀ x:nat.   same x 3 = same 3 x

By induction on x.

Case: x=0

        same x 3

  ==

        ⋮

  == same 3 x

Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem: ∀ x:nat.   same x 3 = same 3 x

By induction on x.

Case: x=0

same x 3

== same 0 3

== if 0=0 then 3=0 else …

⋮

== same 3 x

Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem: ∀ x:nat.   same x 3 = same 3 x

By induction on x.

Case: x=0

      same x 3

  == same 0 3

  == if 0=0 then 3=0 else …

  == 3=0

  == false

  ⋮

  == same 3 x

Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem: ∀ x:nat.   same x 3 = same 3 x

By induction on x.

Case: x=0

```
      same x 3
  == same 0 3
  == if 0=0 then 3=0 else …
  == 3=0
  == false
        ⋮
  == if 3=0 then 0=0 else 0>0 && same (3-1) (0-1)
  == same 3 x
```

Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem: ∀ x:nat.   same x 3 = same 3 x

By induction on x.

Case: x=0

```
      same x 3
  == same 0 3
  == if 0=0 then 3=0 else …
  == 3=0
  == false
  == false && same (3-1) (0-1)
  == 0>0 && same (3-1) (0-1)
  == if 3=0 then 0=0 else 0>0 && same (3-1) (0-1)
  == same 3 x
```

Done with Case: x=0.

Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem: ∀ x:nat.   same x 3 = same 3 x
By induction on x.

Where a satisfies I.H.,
same a 3 = same 3 a

Case: x=a+1, where a:nat

      same x 3
  == same (a+1) 3

⋮

  == same 3 x

Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem: ∀ x:nat.   same x 3 = same 3 x

By induction on x.

Case: x=a+1, where a:nat

```
     same x 3
  == same (a+1) 3
  == if (a+1)=0 then 3=0 else 3>0 && same (a+1-1) (3-1)
       ⋮
  == same 3 x
```

Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem: ∀ x:nat.   same x 3 = same 3 x

By induction on x.

Case: x=a+1, where a:nat

```
      same x 3
  == same (a+1) 3
  == if (a+1)=0 then 3=0 else 3>0 && same (a+1-1) (3-1)
  == 3>0 && same a 2
  == same a 2

        ⋮

  == same 3 x
```

Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem: ∀ x:nat.   same x 3 = same 3 x

By induction on x.

Case: x=a+1, where a:nat

```
      same x 3
  == same (a+1) 3
  == if (a+1)=0 then 3=0 else 3>0 && same (a+1-1) (3-1)
  == 3>0 && same a 2
  == same a 2
        ⋮
  == same 2 a
  == a+1>0 && same 2 a
  == if 3=0 then (a+1)=0 else a+1>0 && same (3-1) (a+1-1)
  == same 3 x
```

Now prove this!

let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)

Theorem: ∀ x:nat.   same x 3 = same 3 x

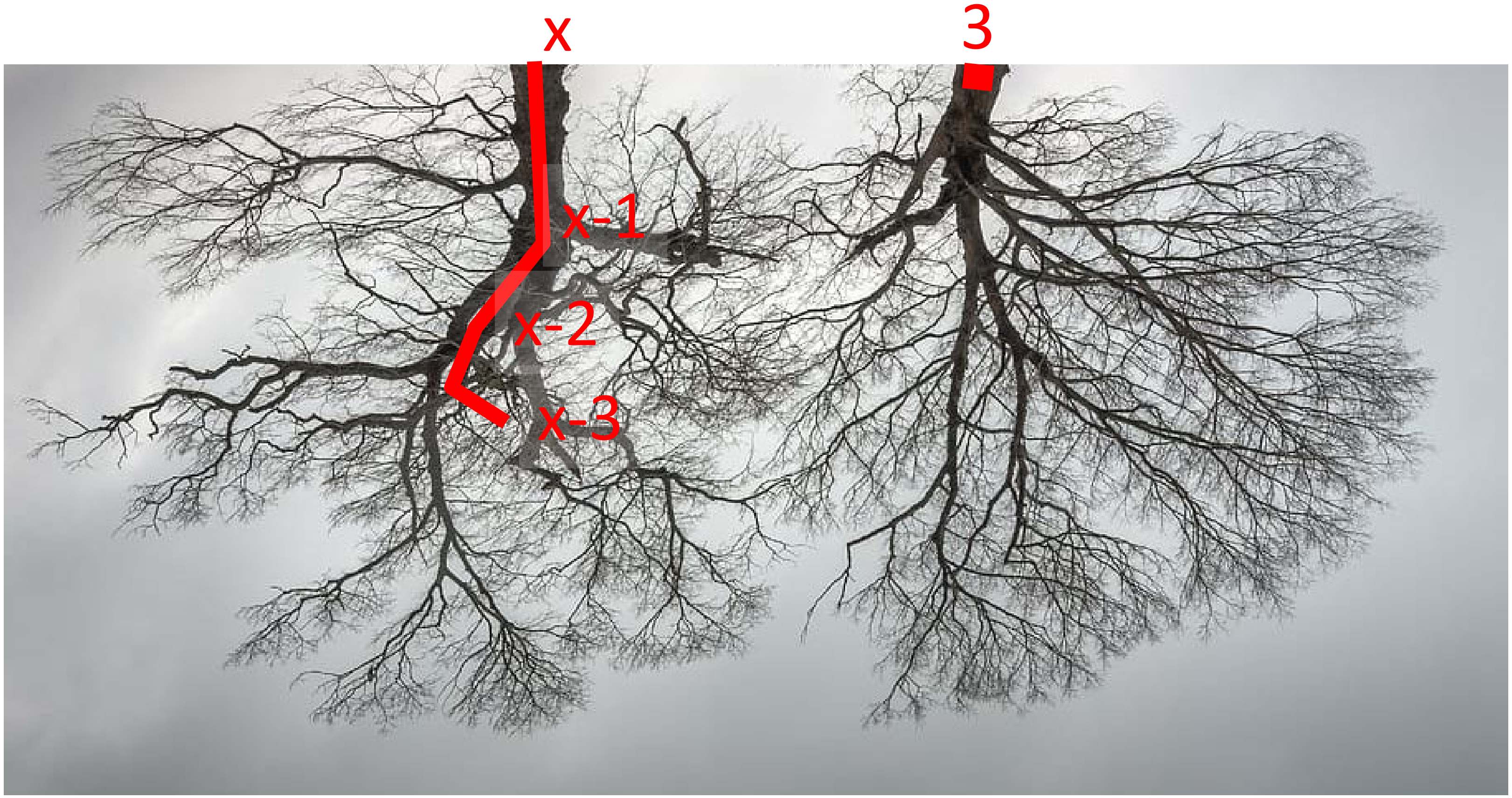
By induction on x.

Case: x=a+1, where a:nat

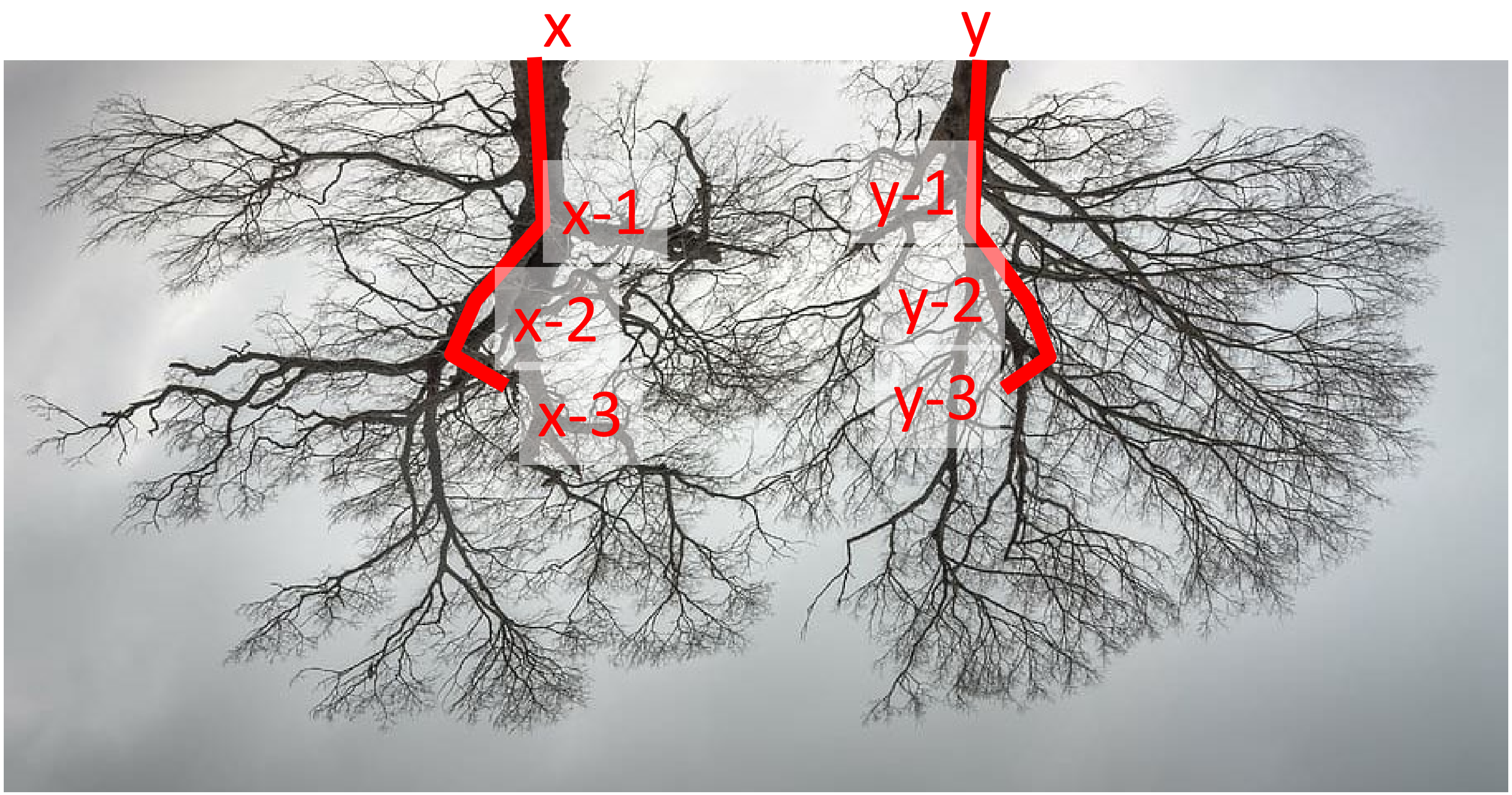
       same x 3
   == same (a+1) 3
   == if (a+1)=0 then 3=0 else 3>0 && same (a+1-1) (3-1)
   == 3>0 && same a 2
   == same a 2
          ⋮
   == same 2 a
   == a+1>0 && same 2 a
   == if 3=0 then (a+1)=0 else a+1>0 && same (3-1) (a+1-1)
   == same 3 x

Induction hyp tells us:
same a 3 = same 3 a

What's the problem?



What's the problem?



Now prove this!

```
let rec same (i: int) (j: int) : bool = if i=0 then j=0 else j>0 && same (i-1) (j-1)
```

Theorem 3: ∀ x:nat.   same x three = same three x

First, prove a more general theorem:

Theorem 4: ∀ x:nat. ∀ y:nat. same x y = same y x

Exercise

- Finish the proof yourself!

It looks just like the proof about

$\forall$ t:tree. $\forall$ u:tree. mirror t u = mirror u t

Conclusion:

WALK DOWN BOTH TREES TOGETHER, IN YOUR PROOF;

DON’T STAY AT THE ROOT OF ONE OF THE TREES.

How OCaml is compiled to a von Neumann machine

Speaker: Andrew Appel

COS 326

Princeton University

Two models for OCaml

Interpreter

```
let rec eval (e:exp) : exp =
  match e with
  | Int_e i -> Int_e i
  | Op_e(e1,op,e2) ->
        eval_op (eval e1) op (eval e2)
  | Let_e(x,e1,e2) ->
        eval (substitute (eval e1) x e2)
  | Var_e x -> raise (UnboundVariable x)
  | Fun_e (x,e) -> Fun_e (x,e)
  | FunCall_e (e1,e2) ->
      (match eval e1
       | Fun_e (x,e) ->
            eval (Let_e (x,e2,e))
       | _ -> raise TypeError)
  | LetRec_e (x,e1,e2) ->
     (Rec_e (f,x,e)) as f_val ->
       let v = eval e2 in
       substitute f_val f
                (substitute v x e)
```

Operational semantics

$$\frac{i \in Z}{i \dashrightarrow i}$$

$$\frac{e1 \dashrightarrow v1 \qquad e2 \dashrightarrow v2 \qquad \text{eval\_op}\,(v1, op, v2) == v}{e1\ op\ e2 \dashrightarrow v}$$

$$\frac{e1 \dashrightarrow v1 \qquad e2\,[v1/x] \dashrightarrow v2}{\text{let } x = e1 \text{ in } e2 \dashrightarrow v2}$$

$$\frac{}{\lambda x.e \dashrightarrow \lambda x.e}$$

$$\frac{e1 \dashrightarrow \lambda x.e \qquad e2 \dashrightarrow v2 \qquad e[v2/x] \dashrightarrow v}{e1\ e2 \dashrightarrow v}$$

$$\frac{e1 \dashrightarrow \text{rec } f\ x = e \qquad e2 \dashrightarrow v2 \qquad e[\text{rec } f\ x = e/f][v2/x] \dashrightarrow v3}{e1\ e2 \dashrightarrow v3}$$

Another model of computation

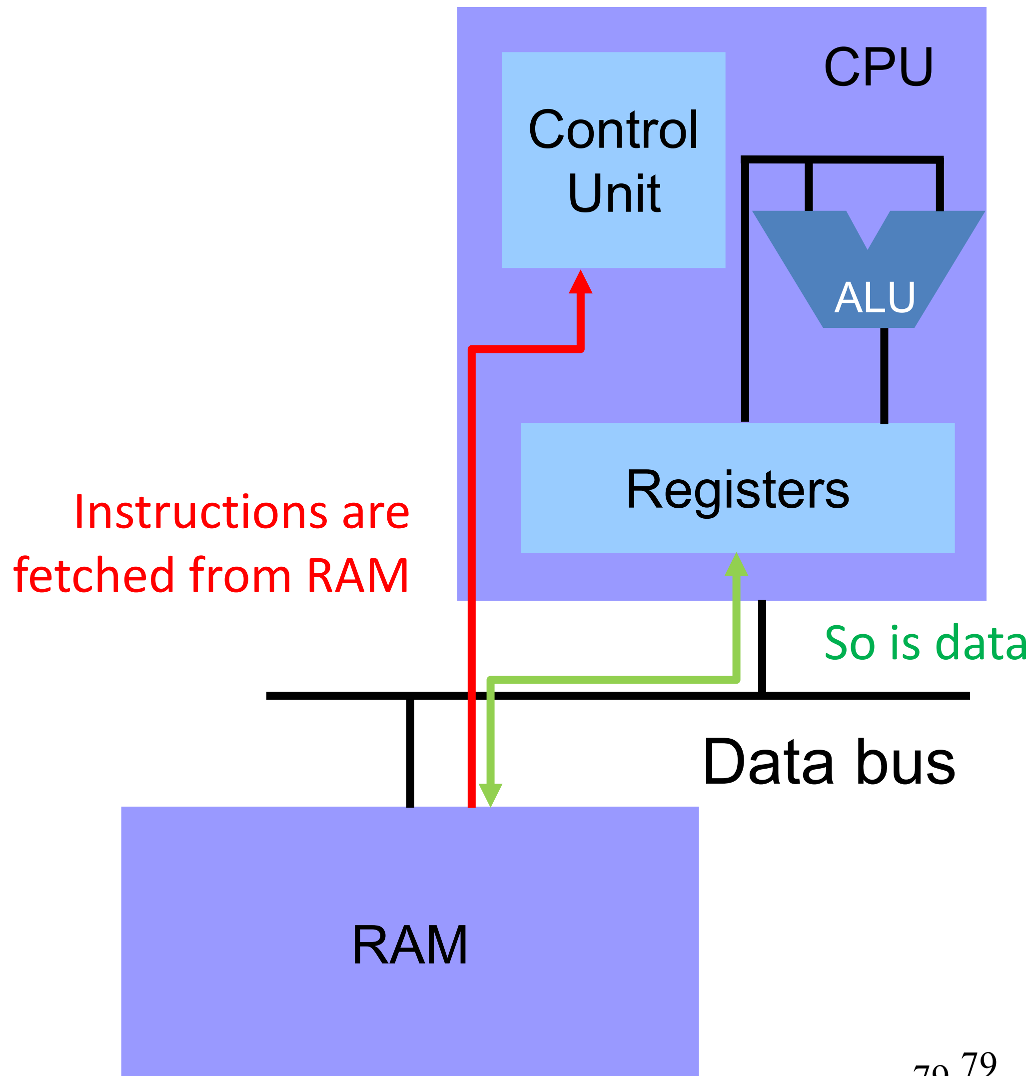
com·put·er
/kəmˈpyo͞odər/
noun
1. an electronic device for storing and processing data, typically in binary form, according to instructions given to it in a variable program.

John Von Neumann (1903-1957)

- Scientific achievements
  - Stored program computers
  - Cellular automata
  - Inventor of game theory
  - Nuclear physics

- Princeton Univ. & Princeton I.A.S. 1930-1957
- Known for "Von Neumann architecture" (1950)
  - In which programs are just data in the memory

Von Neumann Architecture



How OCaml is compiled to machine language

- Variables
- Integers
- Constant constructors
- Value-carrying constructors
- Pattern-matching
- Let x = exp in exp
- Function definition
- Function call
- Tail call

```
type t =
  A | B
  | C of int | D of t*t
```

Variables

Variables are kept in registers,
just as in the translation of C programs
to assembly language

OCaml	**Assembly language**
let x = 3 in …	move 3, r2

When you do a function call, variables whose values will still be needed after the call, will be stored into the stack frame, just as in the translation of C programs to assembly language

If you have more active variables in your function than your machine has registers, some variables will be kept in the stack frame instead of registers, j.a.i.t.t.o.C.p.t.a.l

Integers

The garbage collector needs to distinguish integers from pointers. OCaml does that by using the last bit of the word:
(Word-aligned) pointers end in 00 (binary)
Integers end in 1 (binary)

OCaml

```
let x = 3 in …
```

Assembly language

```
move  7, r2
```

There was a little fib on the previous slide

So, integer N is really stored as 2N+1

And, on a 64-bit-word machine, you really only get 63-bit integers

Constant constructors

```
type t =
 A | B
 | C of int | D of t*t
```

A is represented as 1 (the first odd number)
B is represented as 3 (the second odd number)

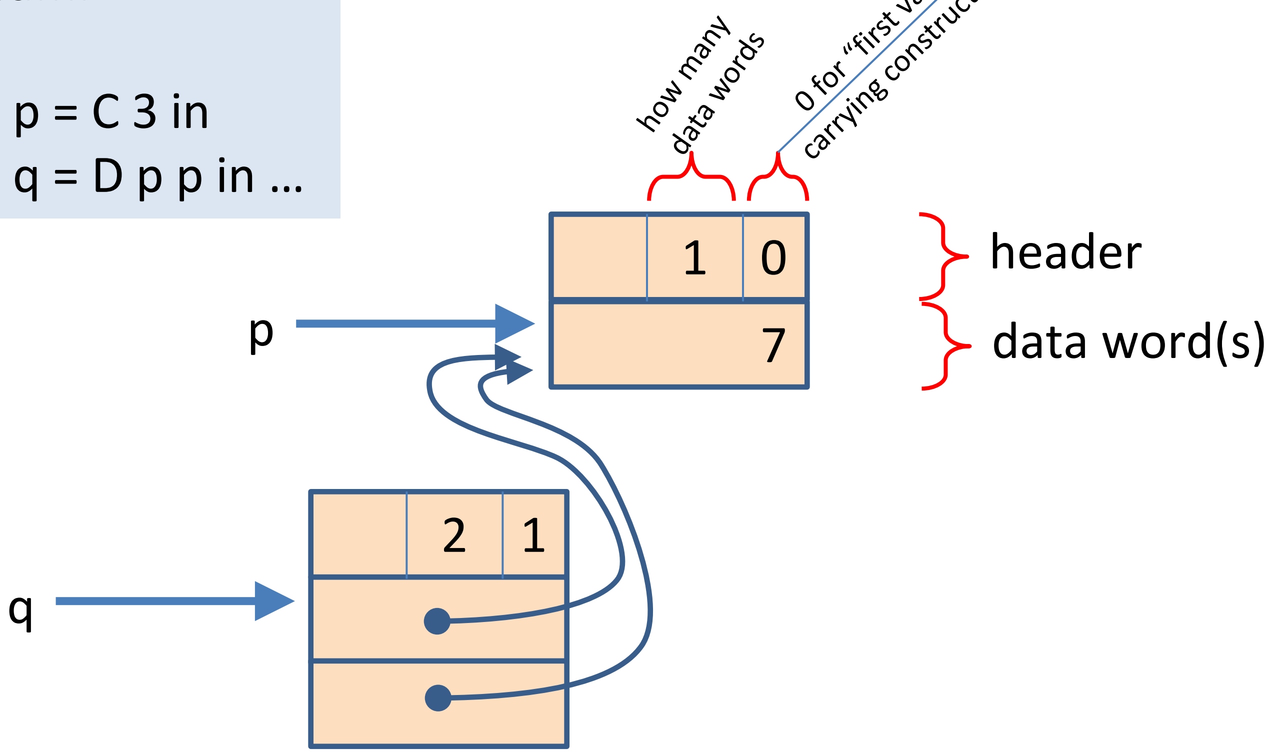
This is similar to how C programs represent NULL as 0

Value-carrying constructors

```
type t =
  A | B
  | C of int | D of t*t
```

OCaml

```
let p = C 3 in
let q = D p p in …
```



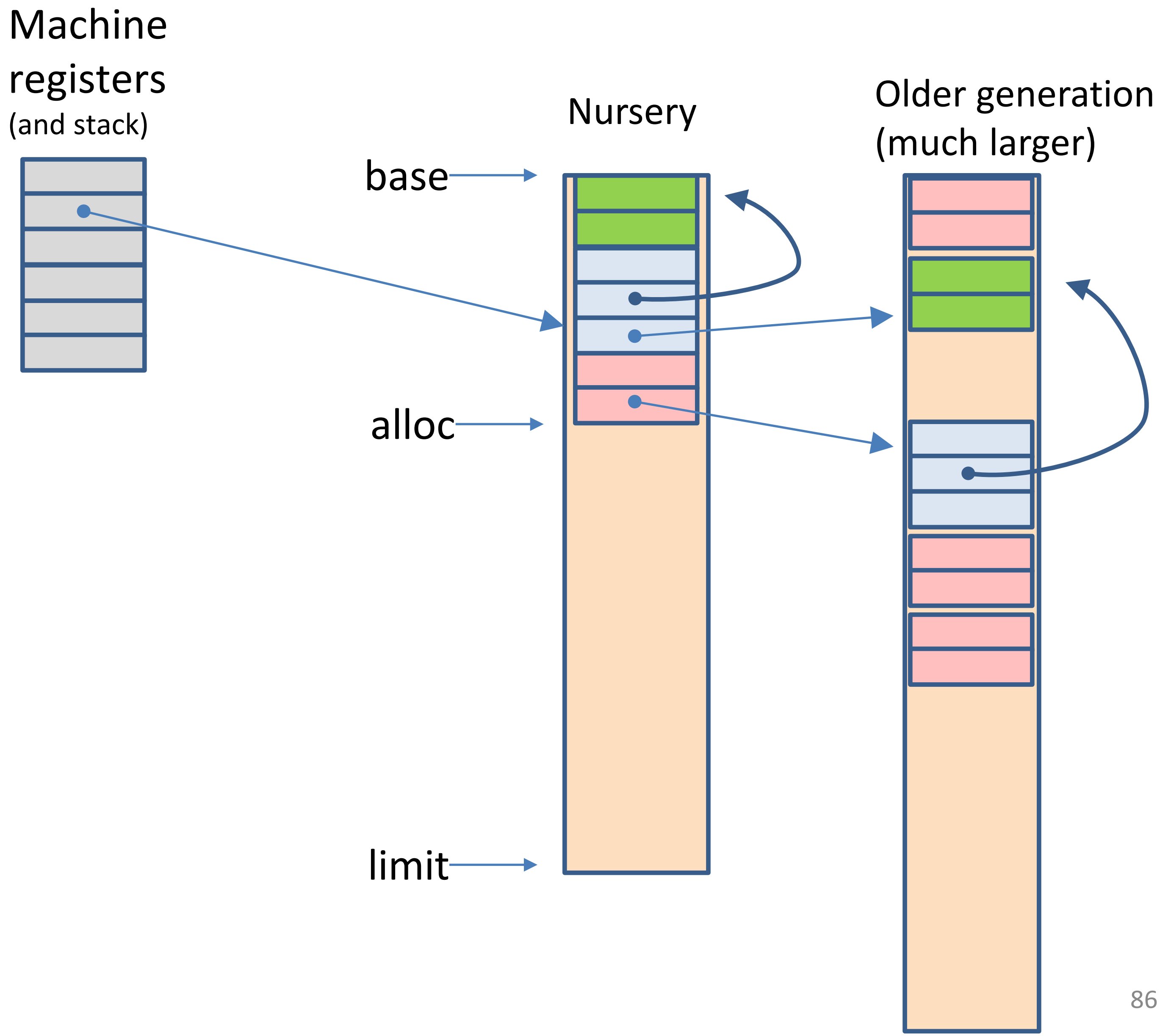
This is similar to how C programs represent malloc'ed struct-pointers

Not malloc/free !

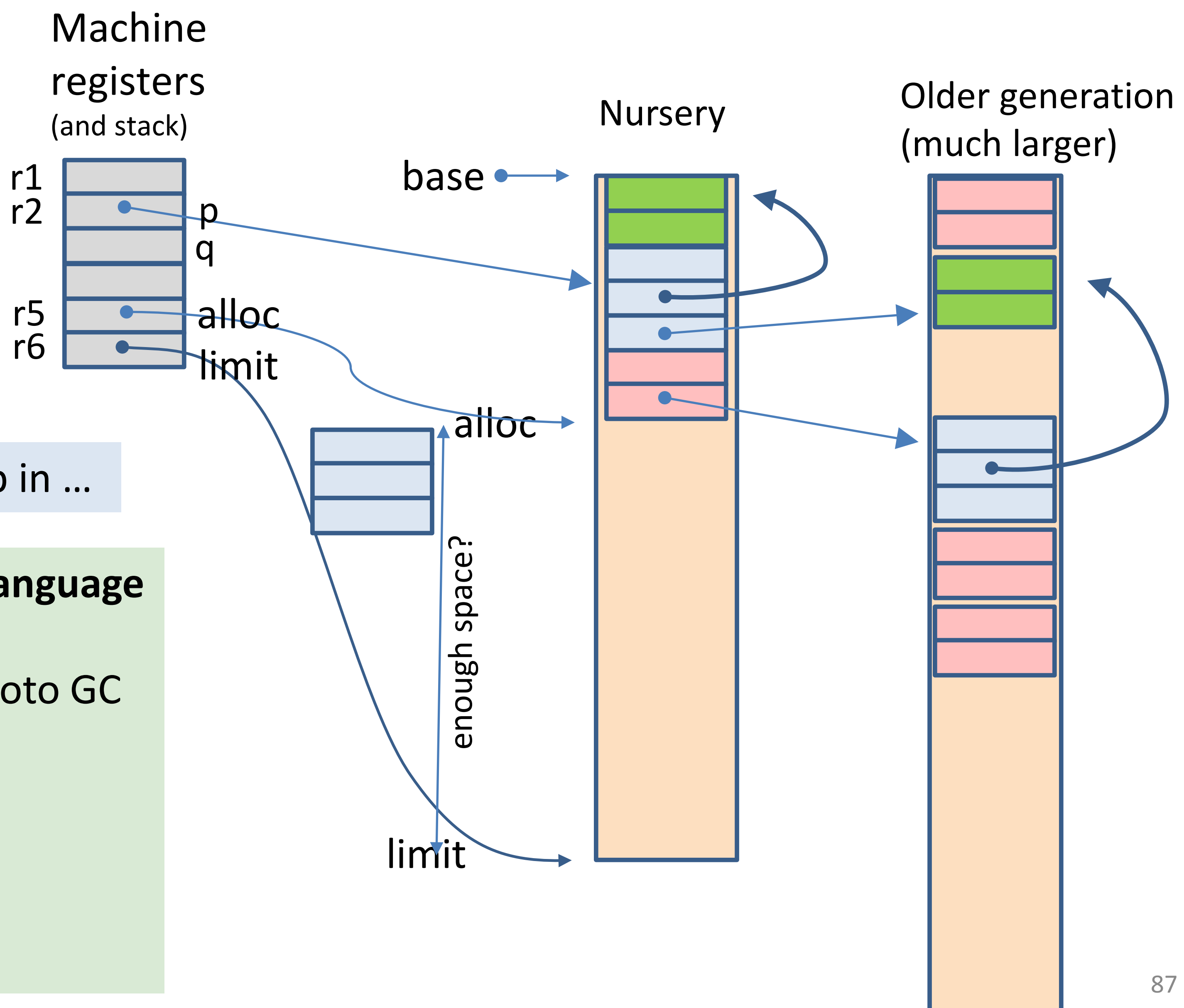
- You may be familiar with how C's malloc/free system works
- Malloc is somewhat expensive:
  - function call
  - find right-size block in data structure
  - update data structure, initialize header and footer
- Free is somewhat expensive:
  - function call
  - update data structure
  - test for coalescing (?)

- OCaml (and other functional languages) have a different system

The heap and the nursery



How to allocate a constructed value

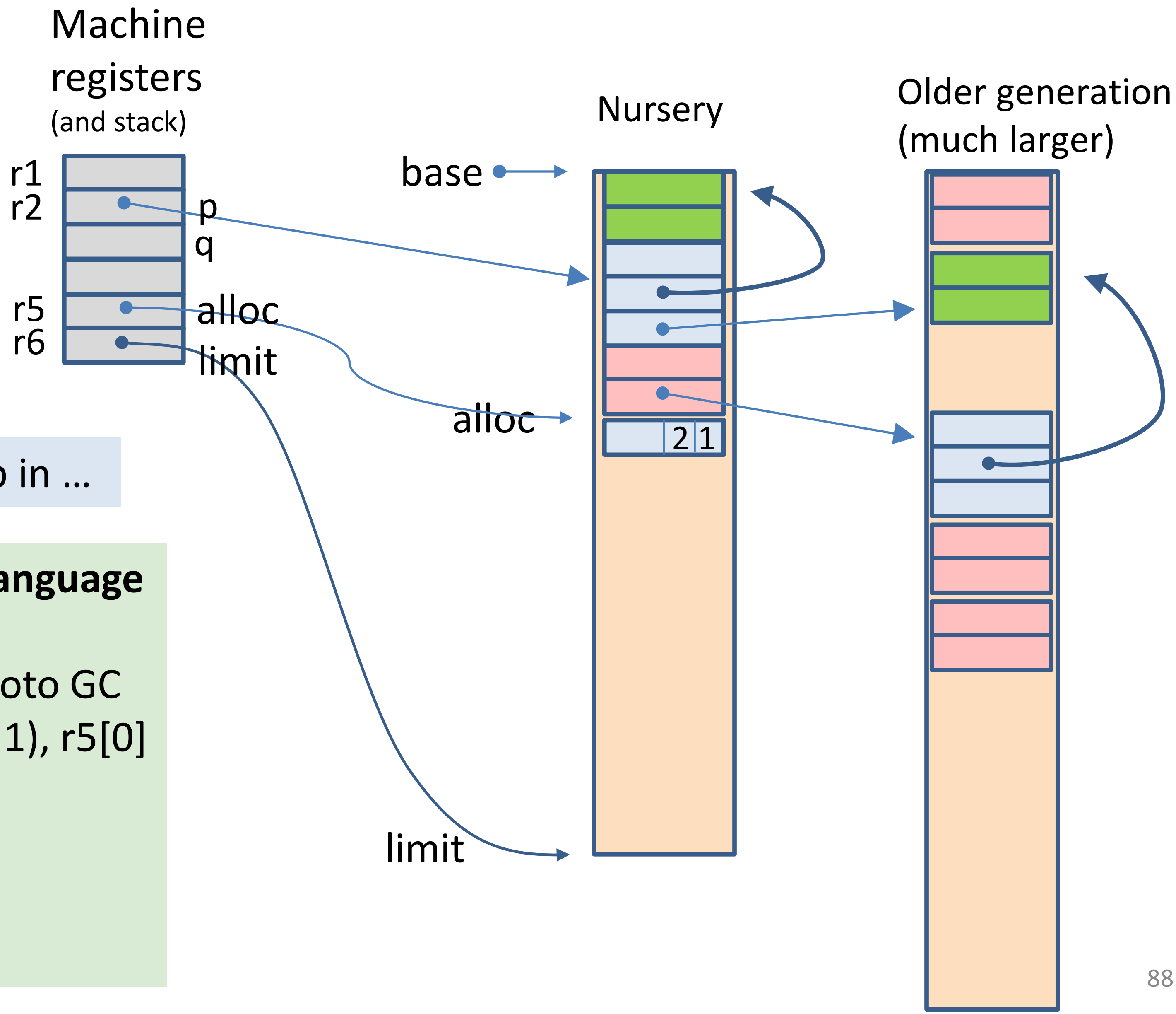


let q = D p p in …

Assembly language

if r5+3>r6 goto GC

How to allocate a constructed value

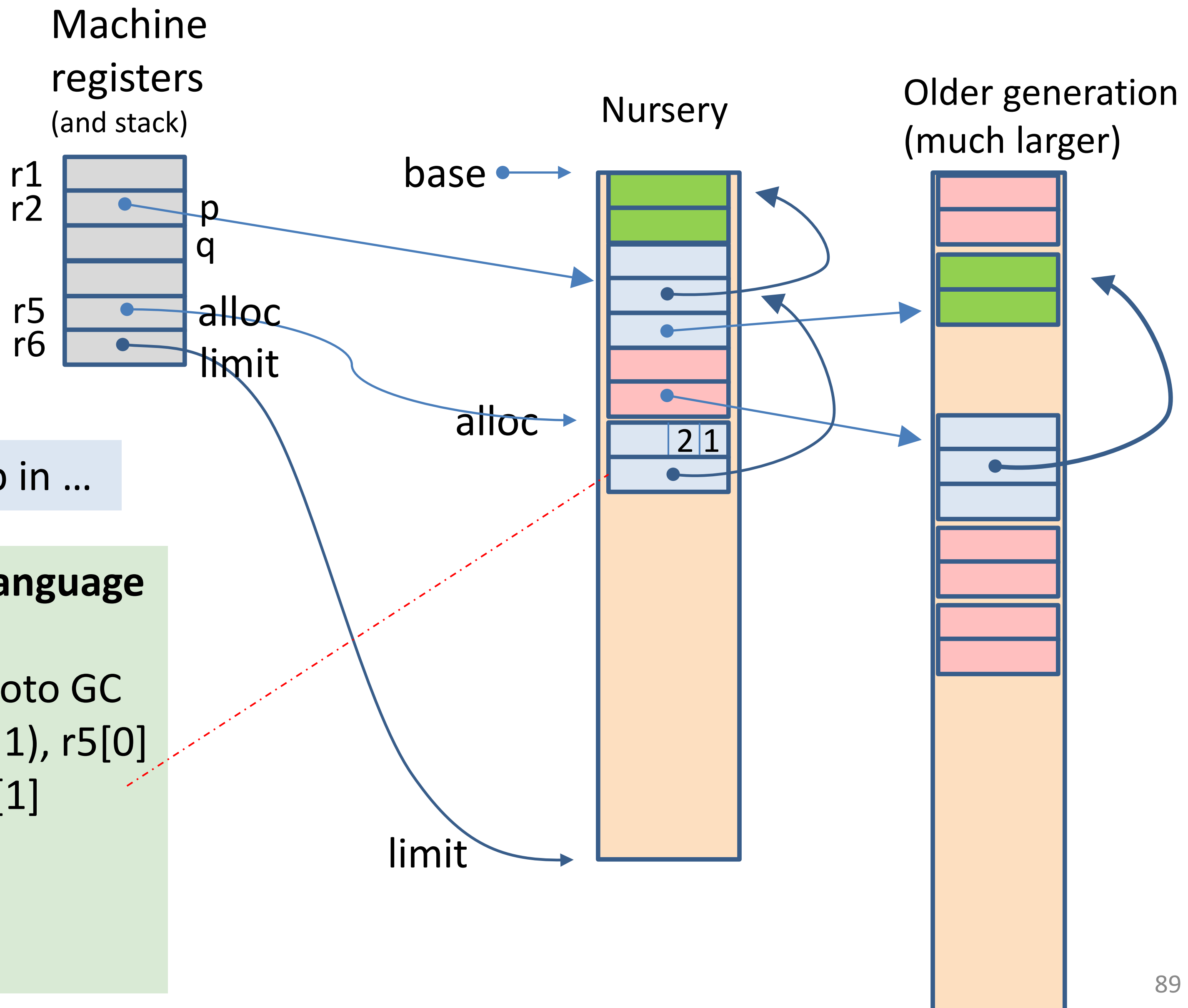


let q = D p p in …

Assembly language

```
if r5+3>r6 goto GC
store  (0|2|1), r5[0]
```

How to allocate a constructed value

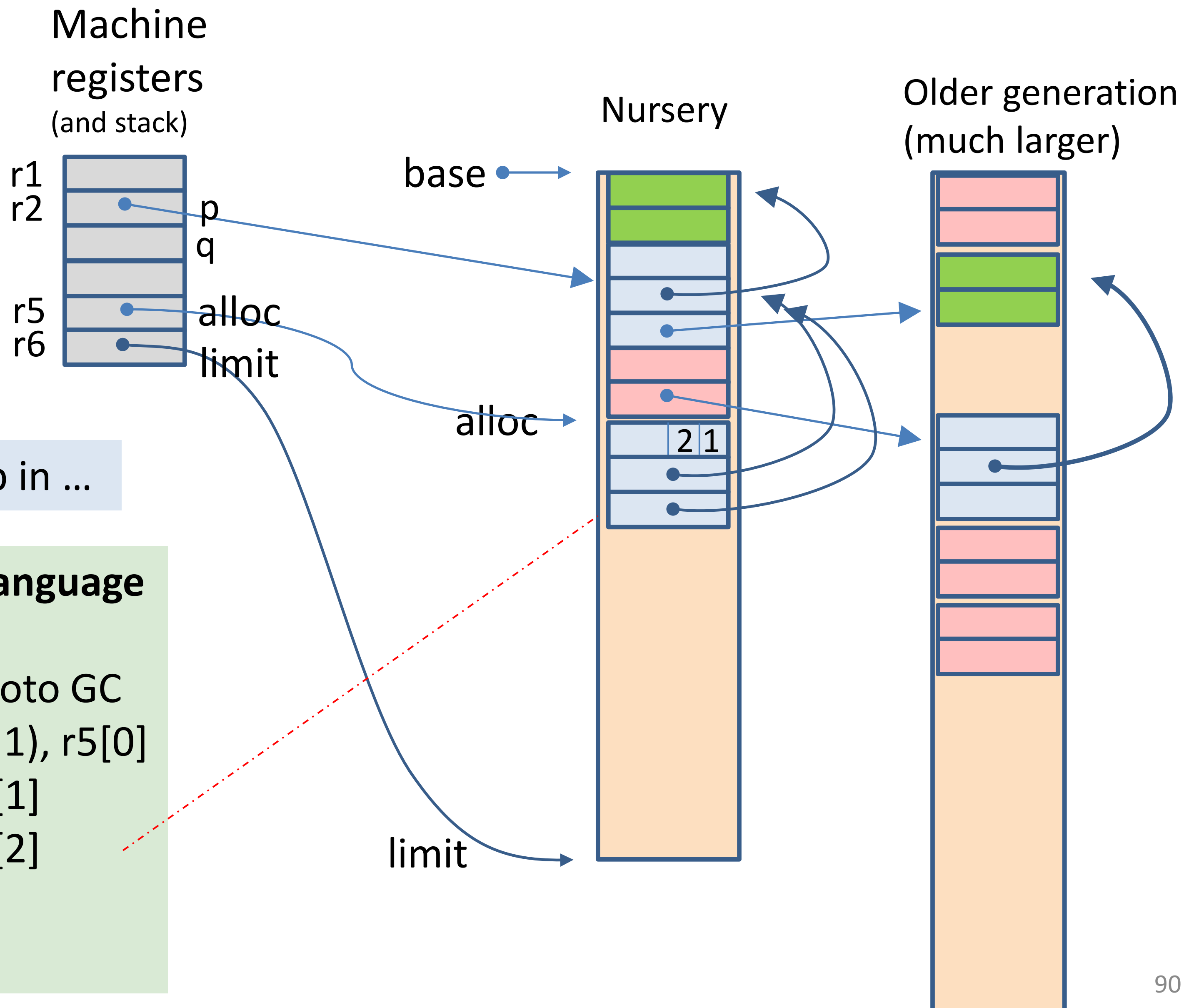


let q = D p p in …

Assembly language

if r5+3>r6 goto GC
store (0|2|1), r5[0]
store r2, r5[1]

How to allocate a constructed value

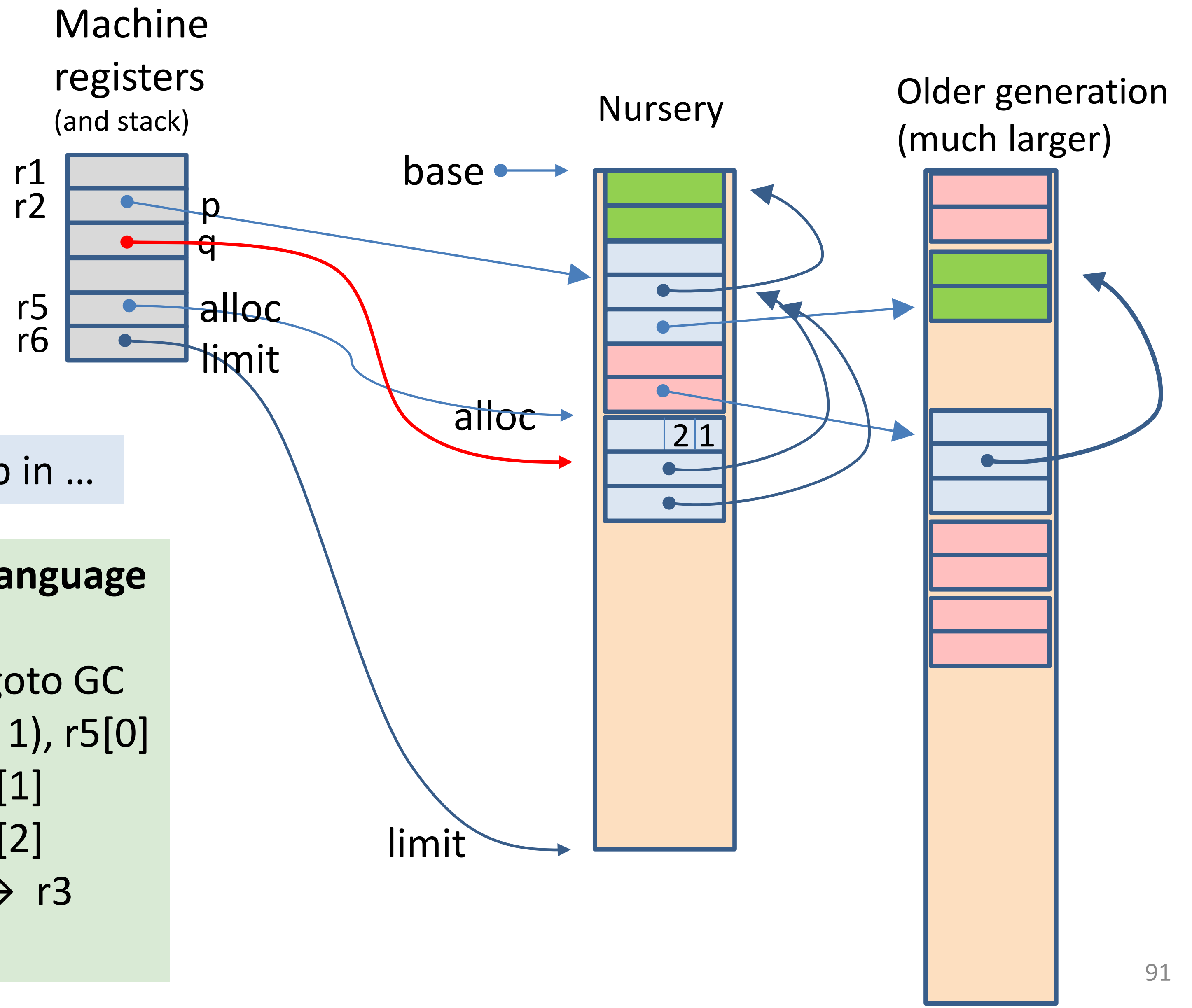


let q = D p p in …

Assembly language

```
if r5+3>r6 goto GC
store  (0|2|1), r5[0]
store r2, r5[1]
store r2, r5[2]
```

How to allocate a constructed value

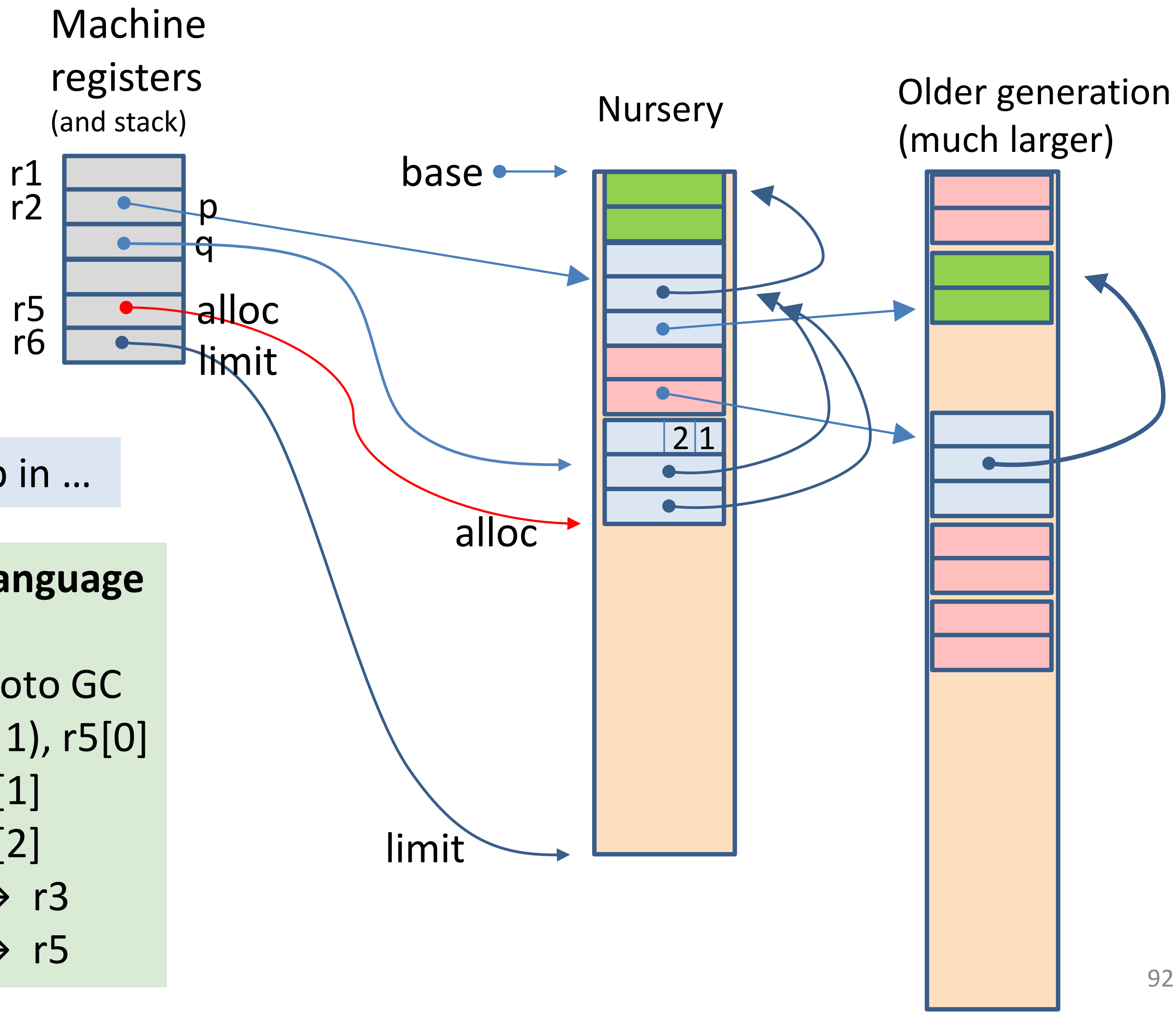


let q = D p p in …

Assembly language

if r5+3>r6 goto GC
store (0|2|1), r5[0]
store r2, r5[1]
store r2, r5[2]
add r5+1 → r3

How to allocate a constructed value



let q = D p p in …

Assembly language

if r5+3>r6 goto GC
store (0|2|1), r5[0]
store r2, r5[1]
store r2, r5[2]
add r5+1 → r3
add r5+3 → r5

How to allocate a constructed value

```
type t =
  A | B
  | C of int | D of t*t
```

```
let q = D p p in ...
```

Assembly language

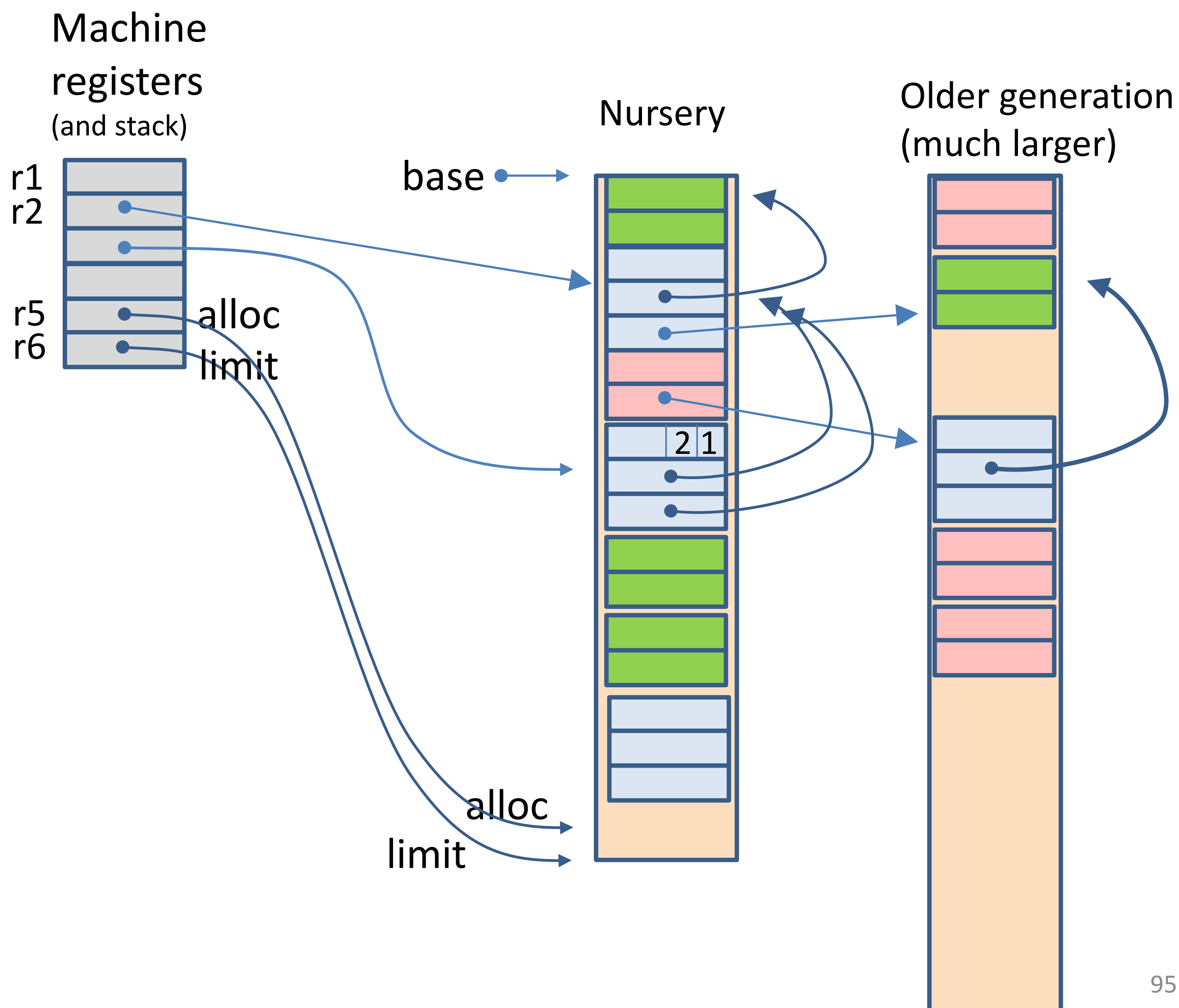
if r5+3>r6 goto GC	test for space available	2 instructions
store (0|2|1), r5[0]	store the header word	
store r2, r5[1]	store first field	initialize the fields
store r2, r5[2]	store second field	
add r5+1 → r3	assign the result (q)	2 instructions
add r5+3 → r5	adjust the "alloc" pointer	

What happens

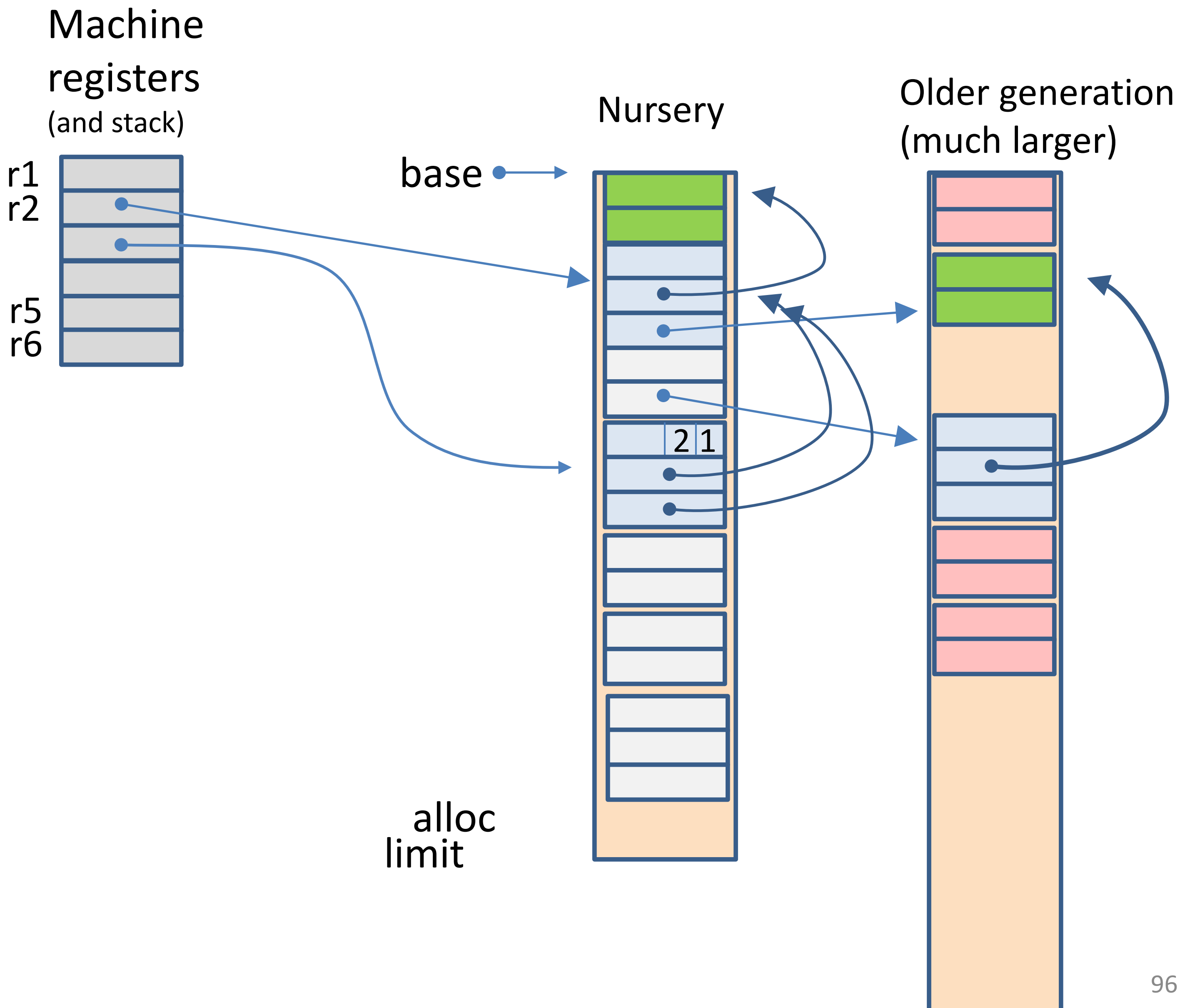
WHEN THE NURSERY FILLS UP . . .

GARBAGE COLLECTION!

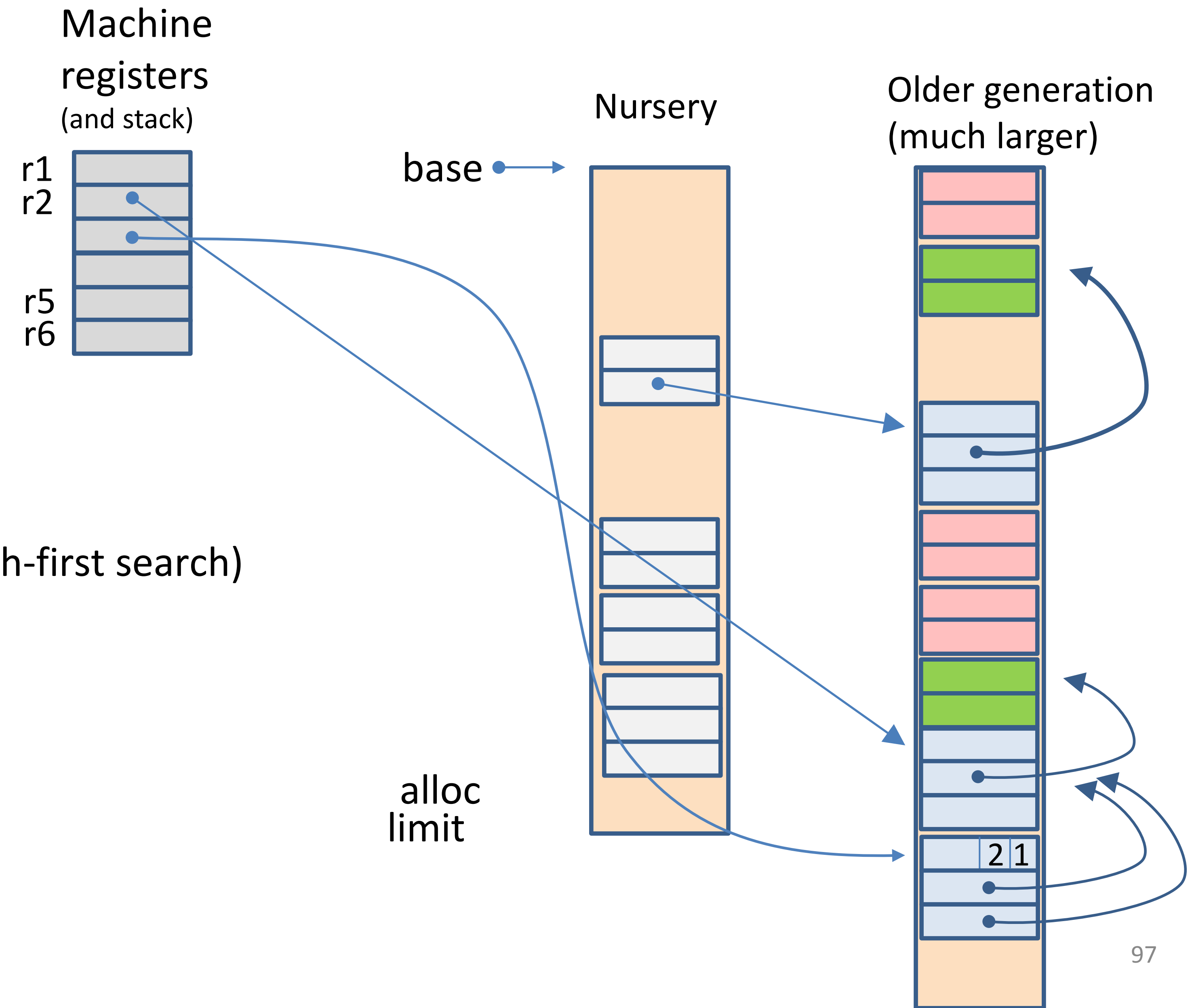
The nursery is full



Only these records are reachable

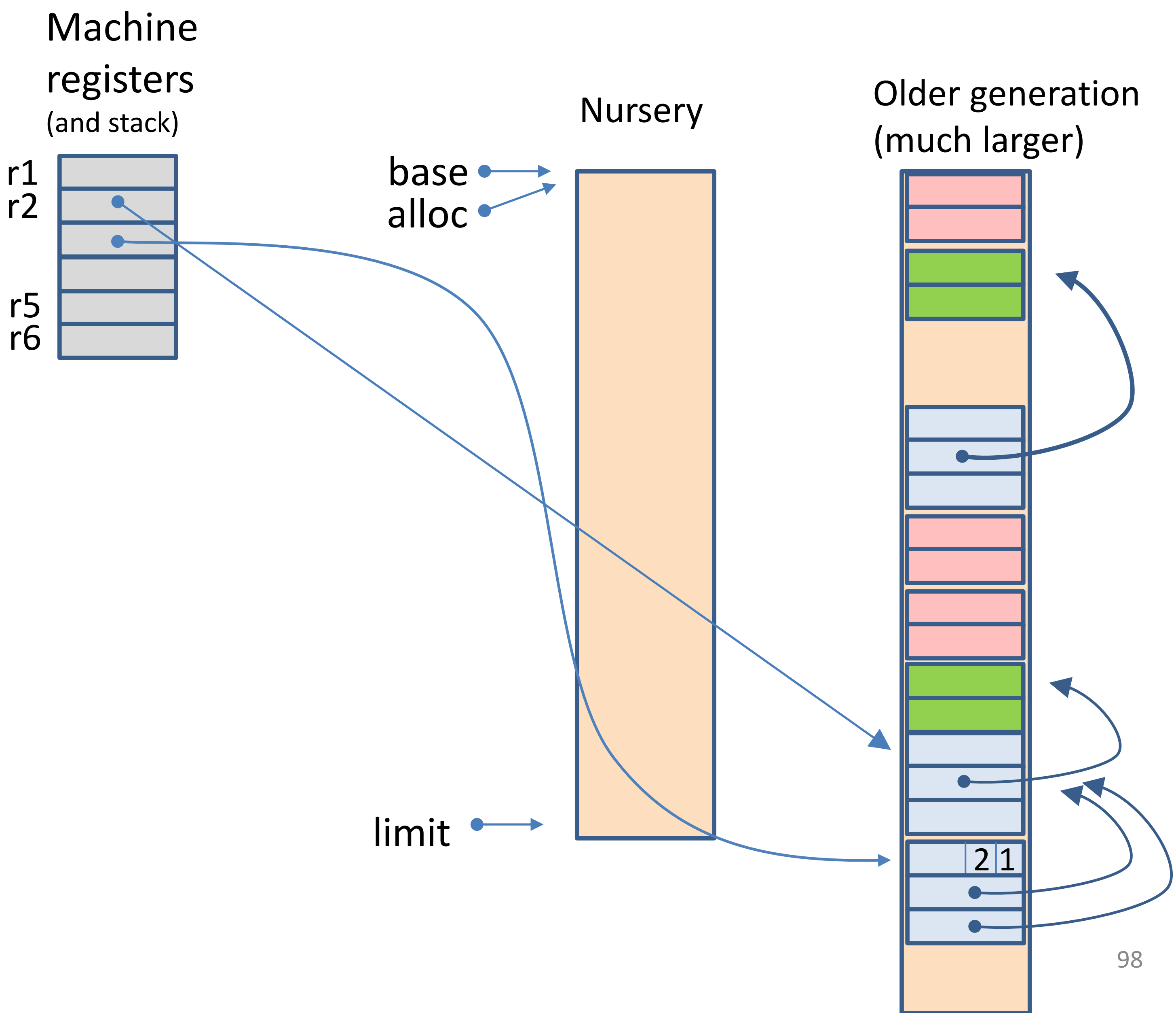


Move reachable records to older generation



(by breadth-first search)

Reset “alloc” pointer of Nursery



How OCaml is compiled to machine language

- ✓ Variables
- ✓ Integers
- ✓ Constant constructors
- ✓ Value-carrying constructors
- Pattern-matching
- Let x = exp in exp
- Function definition
- Function call
- Tail call

```
type t =
  A | B
  | C of int | D of t*t
```

Pattern-matching

```
match x with
| A ->   exp1
| B ->   exp2
| C i -> exp3(i)
| D(i,j) -> exp4 i j
```

```
type t =
  A | B
 | C of int | D of t*t
```

Assembly language
(suppose x is in register r2)

andb   r2,1 → r3
if r3=0 goto Boxed
 handle cases A,B
goto Done
Boxed:
 handle cases C,D
Done:

First, test whether the constructed value is "unboxed" (constant constructor) or "boxed" (value-carrying constructor)

Pattern-matching

```
match x with
| A ->   exp1
| B ->   exp2
| C i -> exp3(i)
| D(i,j) -> exp4 i j
```

```
type t =
  A | B
 | C of int | D of t*t
```

Assembly language
(suppose x is in register r2)

andb   r2,1 → r3
if r3=0 goto Boxed
 (if r2=1 then exp1 else exp2)
goto Done
Boxed:
 handle cases C,D
Done:

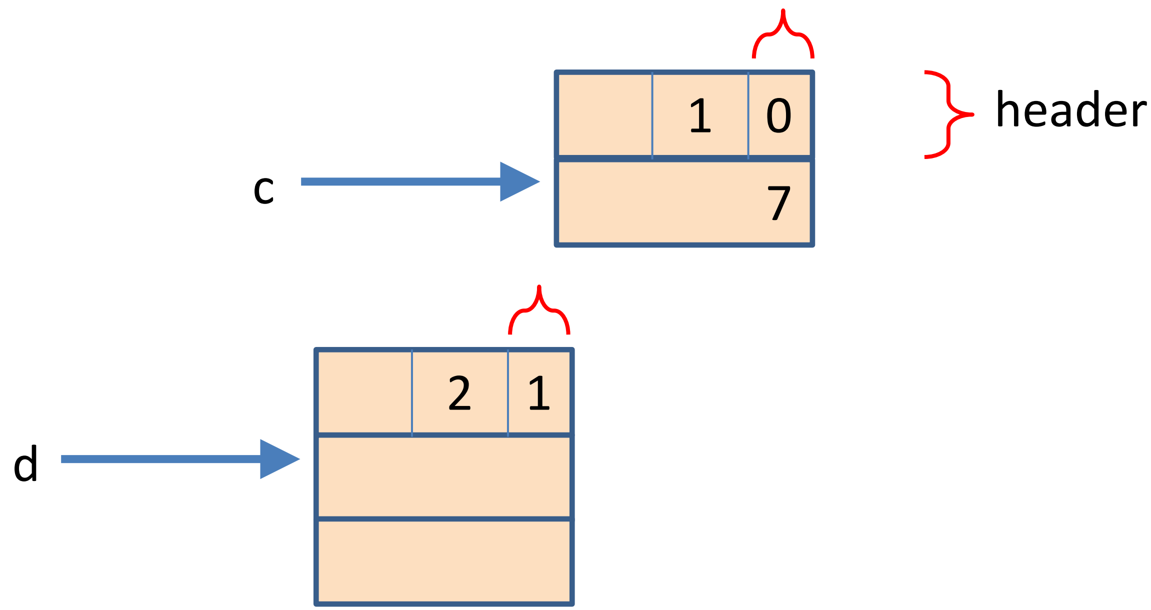
Pattern-matching

```
match x with
| A ->   exp1
| B ->   exp2
| C i -> exp3(i)
| D(i,j) -> exp4 i j
```

```
type t =
 A | B
 | C of int | D of t*t
```

Assembly language
(suppose x is in register r2)

andb   r2,1 → r3
if r3=0 goto Boxed
 handle cases A,B
goto Done
Boxed:
 load r2[-1] → r3
 andb 127,r3 → r3
  (if r3=0 then C else D)
Done:



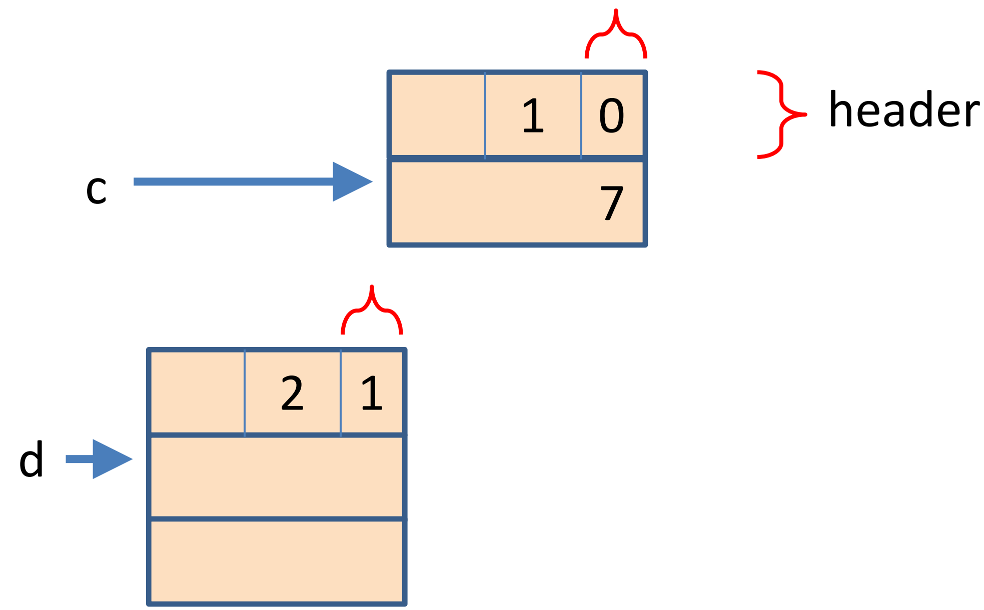
Pattern-matching

```
match x with
| A ->   exp1
| B ->   exp2
| C i -> exp3(i)
| D(i,j) -> exp i j
```

```
type t =
  A | B
  | C of int | D of t*t
```

Assembly language
(suppose x is in register r2)

D case:
load r2[0] → r4   (fetch i)
load r2[1] → r5   (fetch j)



Summary of Pattern-matching

```
match x with
| A -> exp1
| B -> exp2
| C i -> exp3(i)
| D (i,j) -> exp4 i j
```

Conditional branches
(or switch-statement)

Memory loads

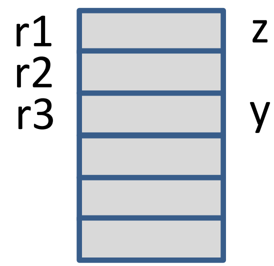
How OCaml is compiled to machine language

- ✓ Variables
- ✓ Integers
- ✓ Constant constructors
- ✓ Value-carrying constructors
- ✓ Pattern-matching
- Let x = exp in exp
- Function definition
- Function call
- Tail call

let x = y + z in ...

let x = y + z in ...

Machine registers
(and stack)



Almost as simple as,

~~**Assembly language**~~

~~add r3+r1 → r4~~

But remember, in order to make integers distinguishable from pointers,
OCaml represents integers with low-order-bit 1,
which is to say, r3=2y+1 r1=2z+1
and we need to compute r4=2(y+z)+1

Assembly language

add r3+r1 → r4
sub r4-1 → r4

Function definitions

```
fun x ->  x+1
```

More or less, a function is translated as a label in assembly language, which stands for an address in machine language, where some machine instructions implement the function:

Assembly language

```
f:
add  r0+2 → r0
ret
```

But there is one important difference from the way C functions are compiled!

Function definitions

```
(fun w -> x+w+y)
```

Free variables!   (in this case, x and y)

Assembly language

f:

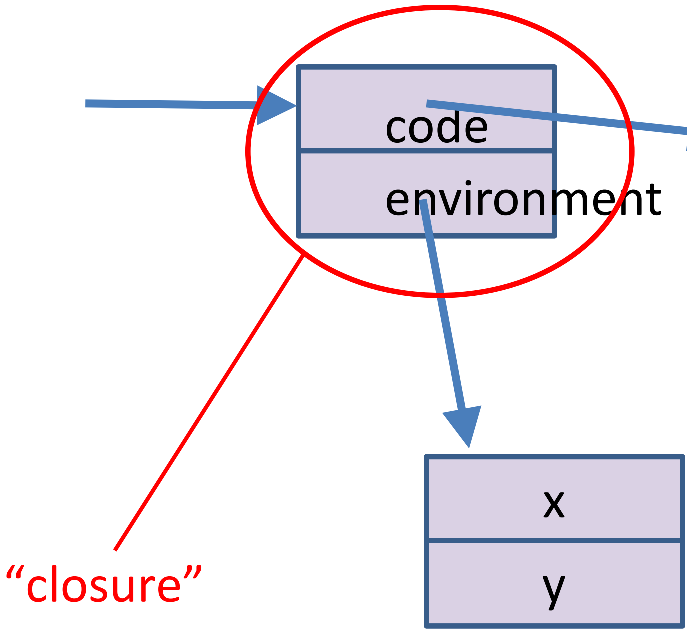
um, how do I know the values of x and y?

ret

Function definitions

(fun w -> x+w+y)

Free variables!   (in this case, x and y)



Assembly language

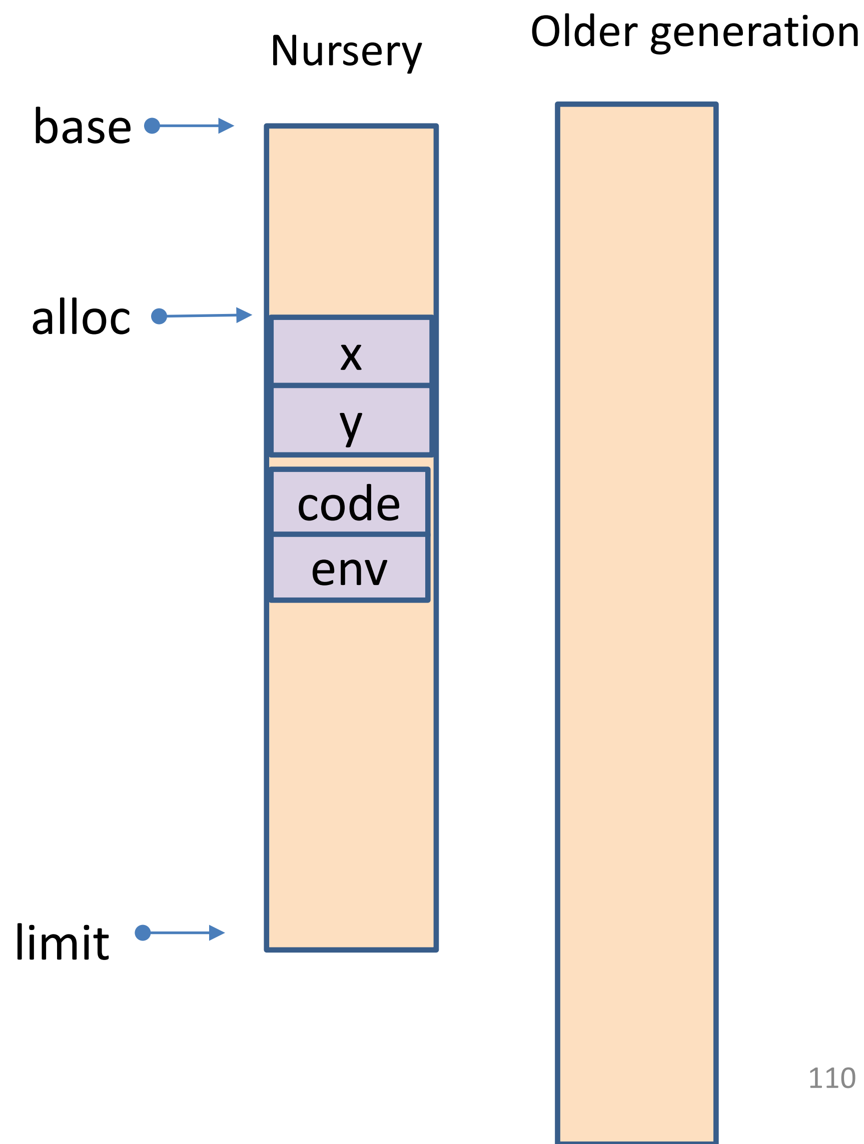
f_code:

get x and y from environment-pointer

ret

Function definitions

(fun w -> x+w+y)



Evaluating “fun … -> …”

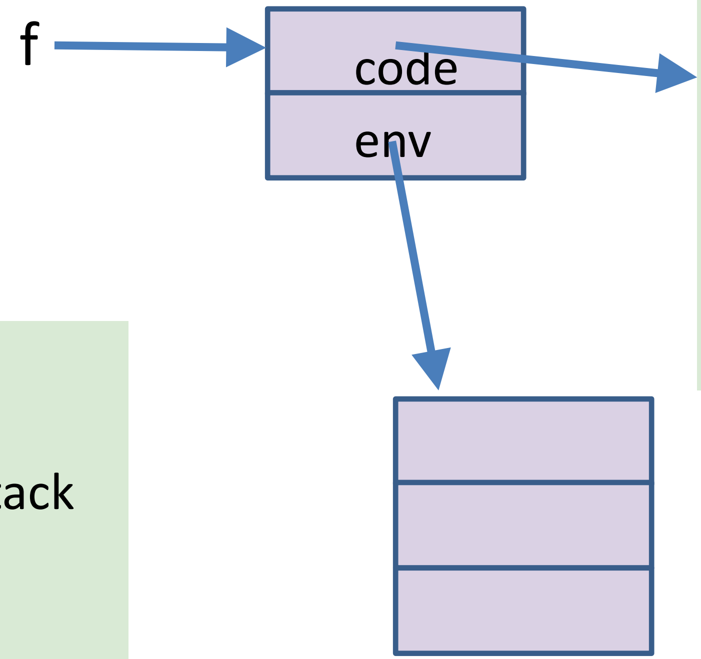
is like constructing two records on the heap

. . .

and will be garbage-collected when no longer in use

Function call

let y = f(x) in …



Assembly language

f_code:

get free vars from env

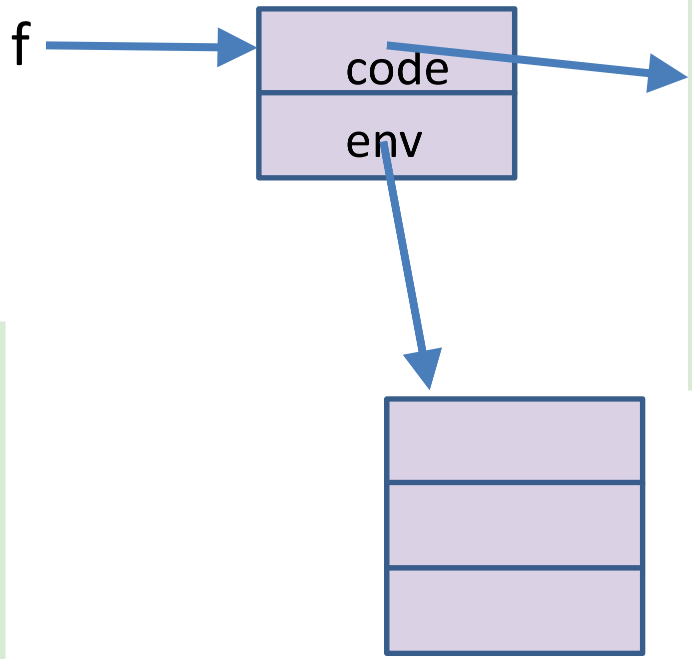
ret

Assembly language

push saved locals on stack
move x → r1 # arg
load f[1] → r2 # env
load f[0] → r3 # code
call r3
pop saved locals from stack

Tail call

f(x)



Assembly language

f_code:

get free vars from env

ret

Assembly language

move x → r1 # arg
load f[1] → r2 # env
load f[0] → r3 # code
jmp r3

Conclusion

- Each feature of the OCaml language is implemented in a few instructions of machine language
- Some of these features work just like their counterparts in C,
- What's different:
  - garbage collection, instead of malloc/free
  - function closures
  - distinguishing integers from pointers, by low-order bit