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

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

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

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

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

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

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

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.

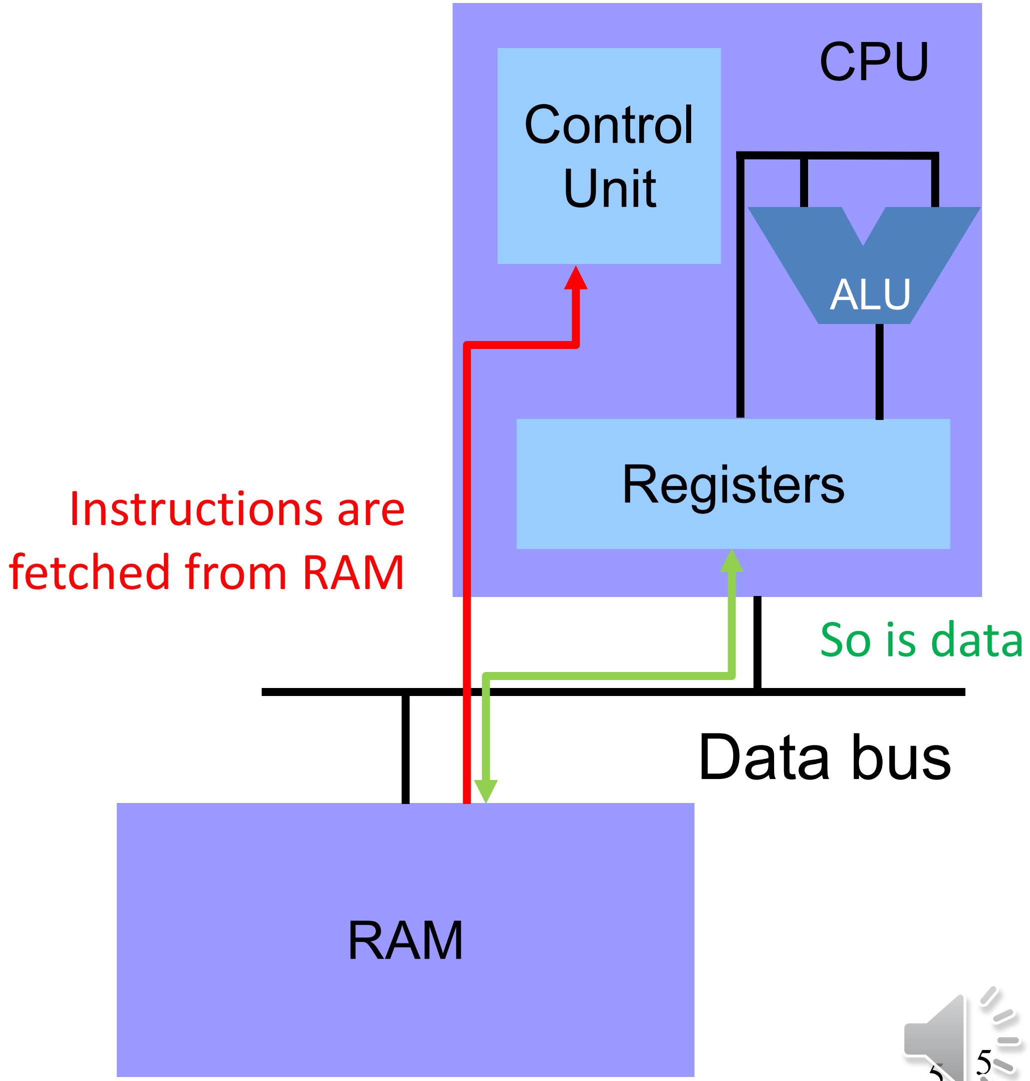
John Von Neumann (1903-1957)

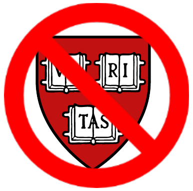
- Scientific achievements
  - Stored program computers
  - Cellular automata
  - Inventor of game theory
  - Nuclear physics

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

4

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)

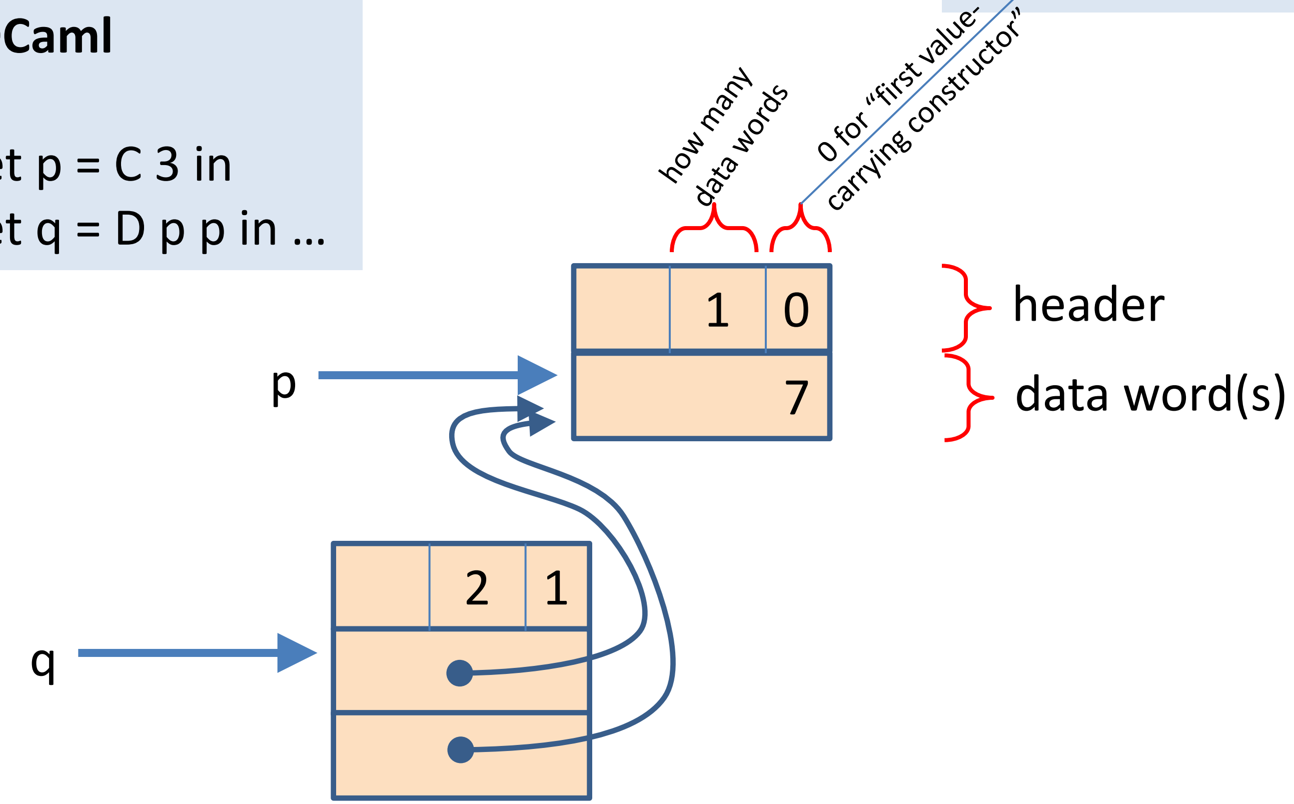
This is similar to how C programs represent NULL as 0

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 …
```



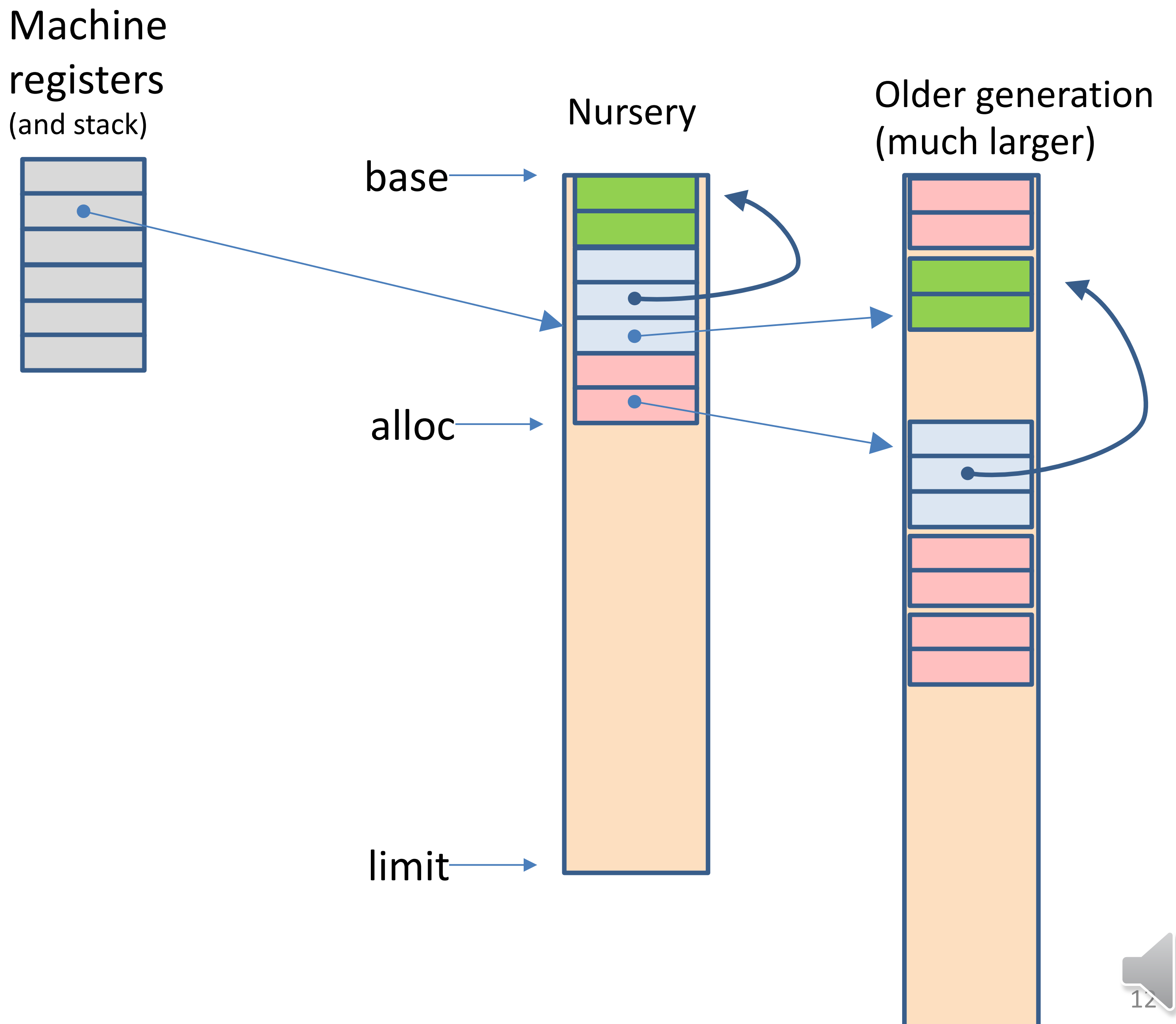
This is similar to how C programs represent malloc'ed struct-pointers

Not malloc/free !

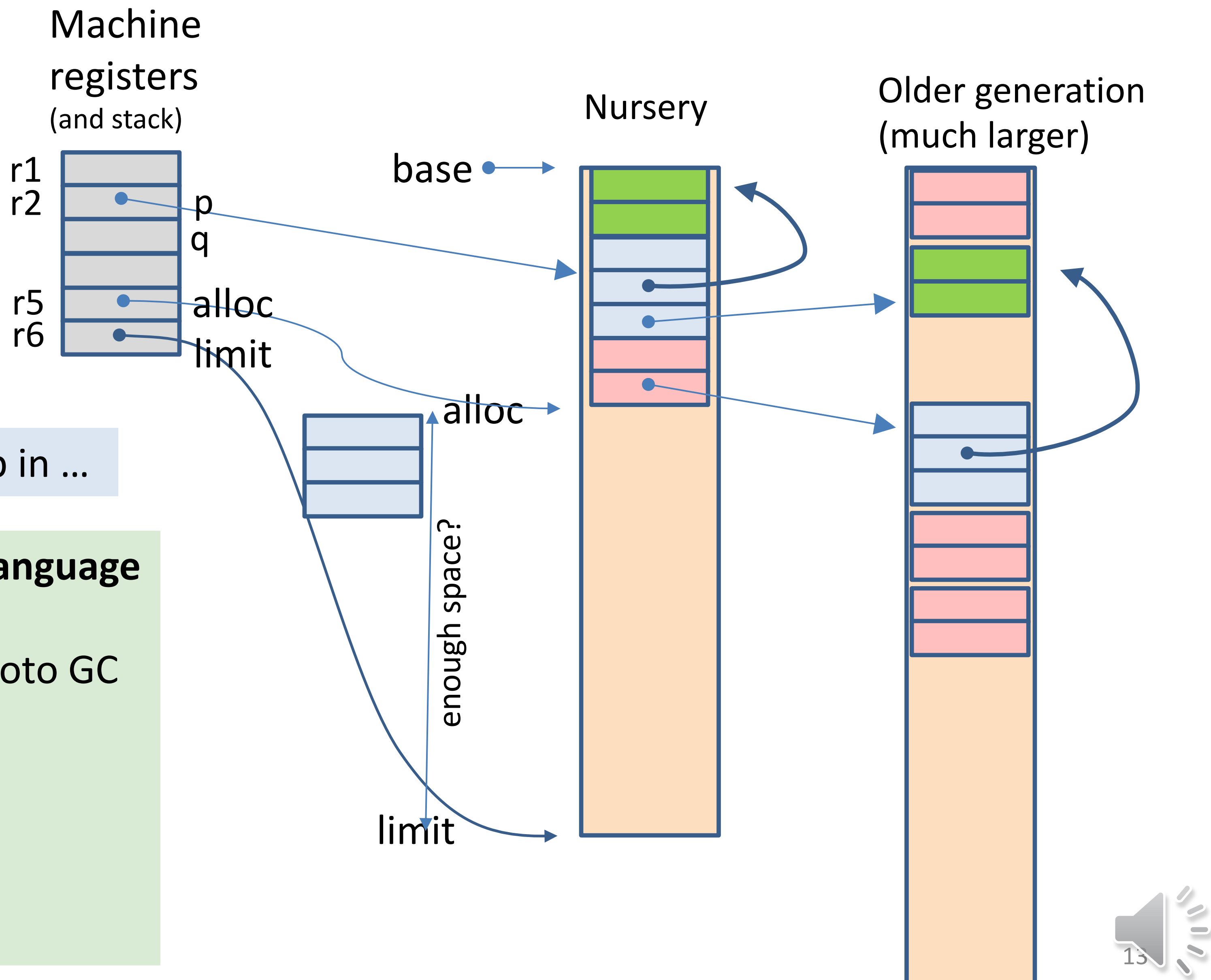
- You may be familiar with how C's malloc/free system works
- Malloc is somewhat expensive:
  - function call
  - find right-size block in data structure
  - update data structure, initialize header and footer
- Free is somewhat expensive:
  - function call
  - update data structure
  - test for coalescing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

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

The heap and the nursery



How to allocate a constructed value

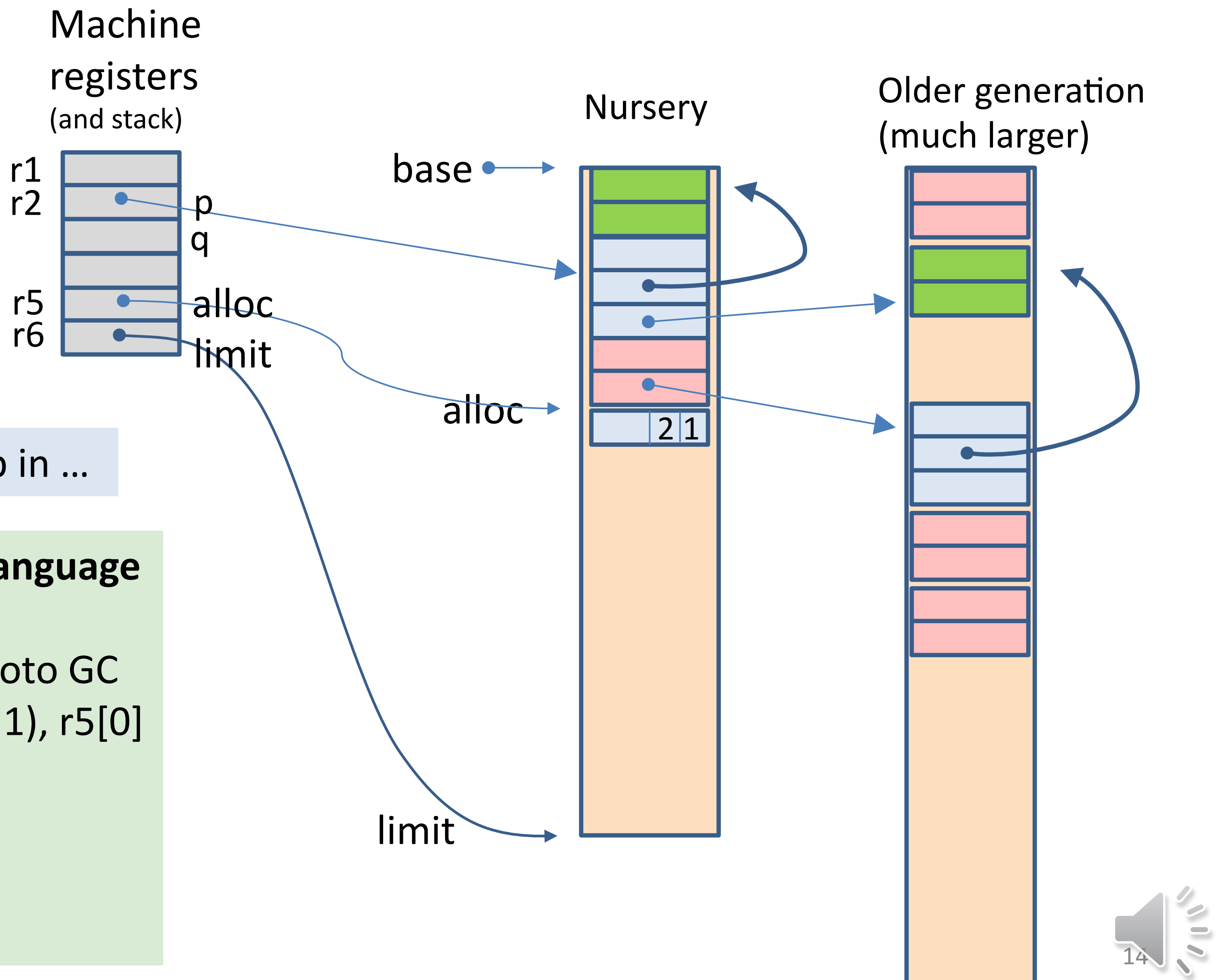


let q = D p p in …

Assembly language

if r5+3>r6 goto GC

How to allocate a constructed value

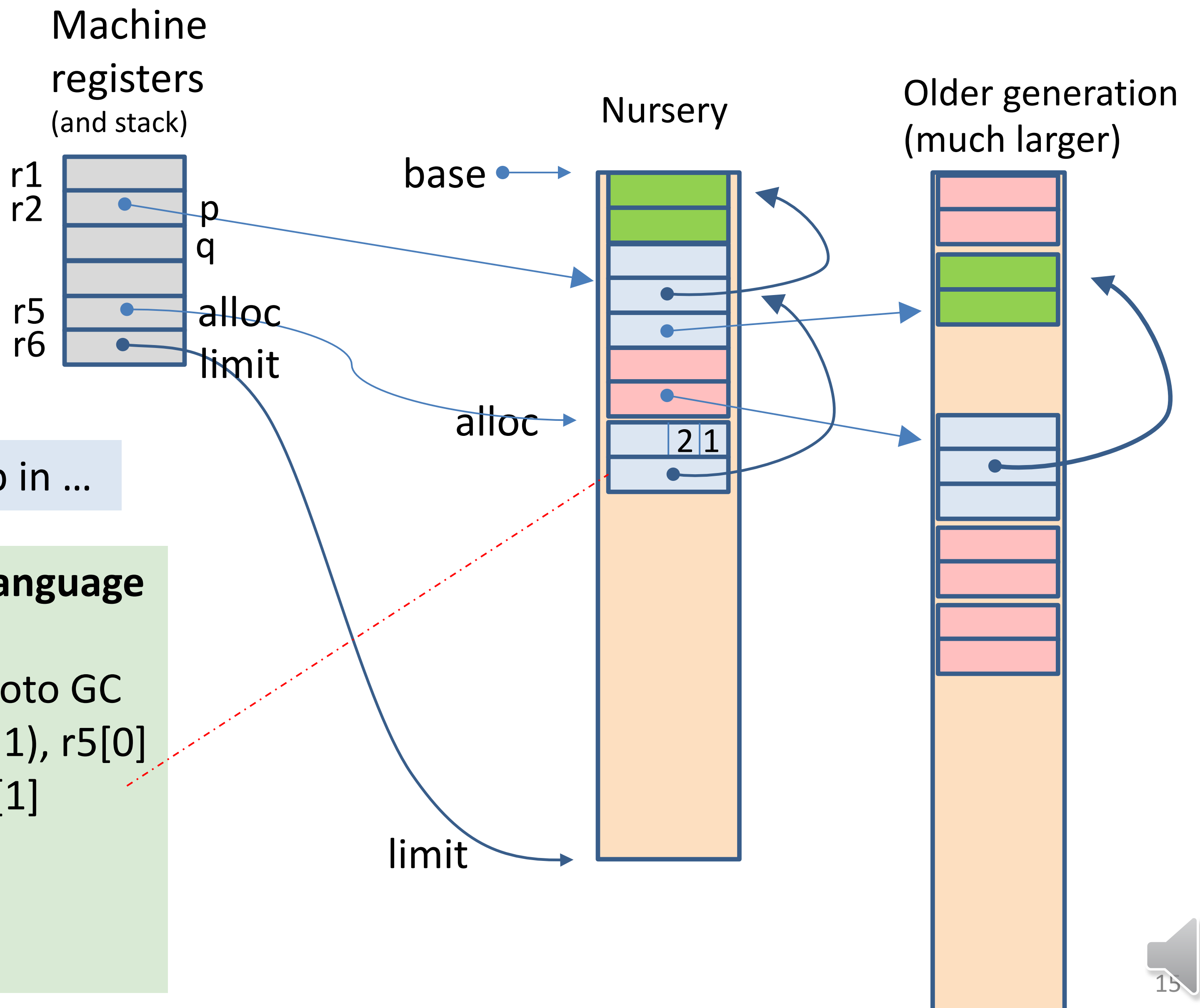


let q = D p p in …

Assembly language

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

How to allocate a constructed value

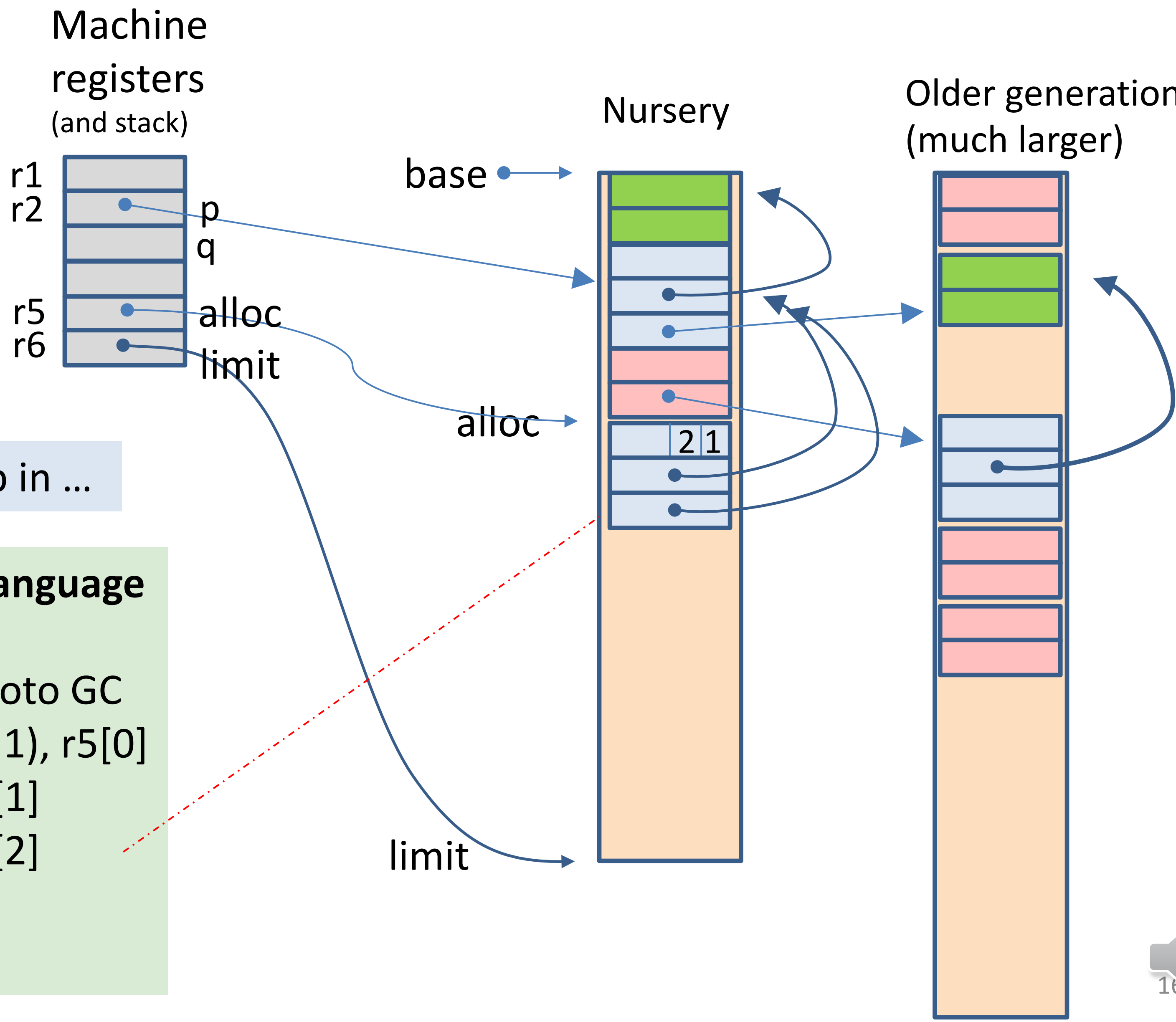


let q = D p p in …

Assembly language

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

How to allocate a constructed value

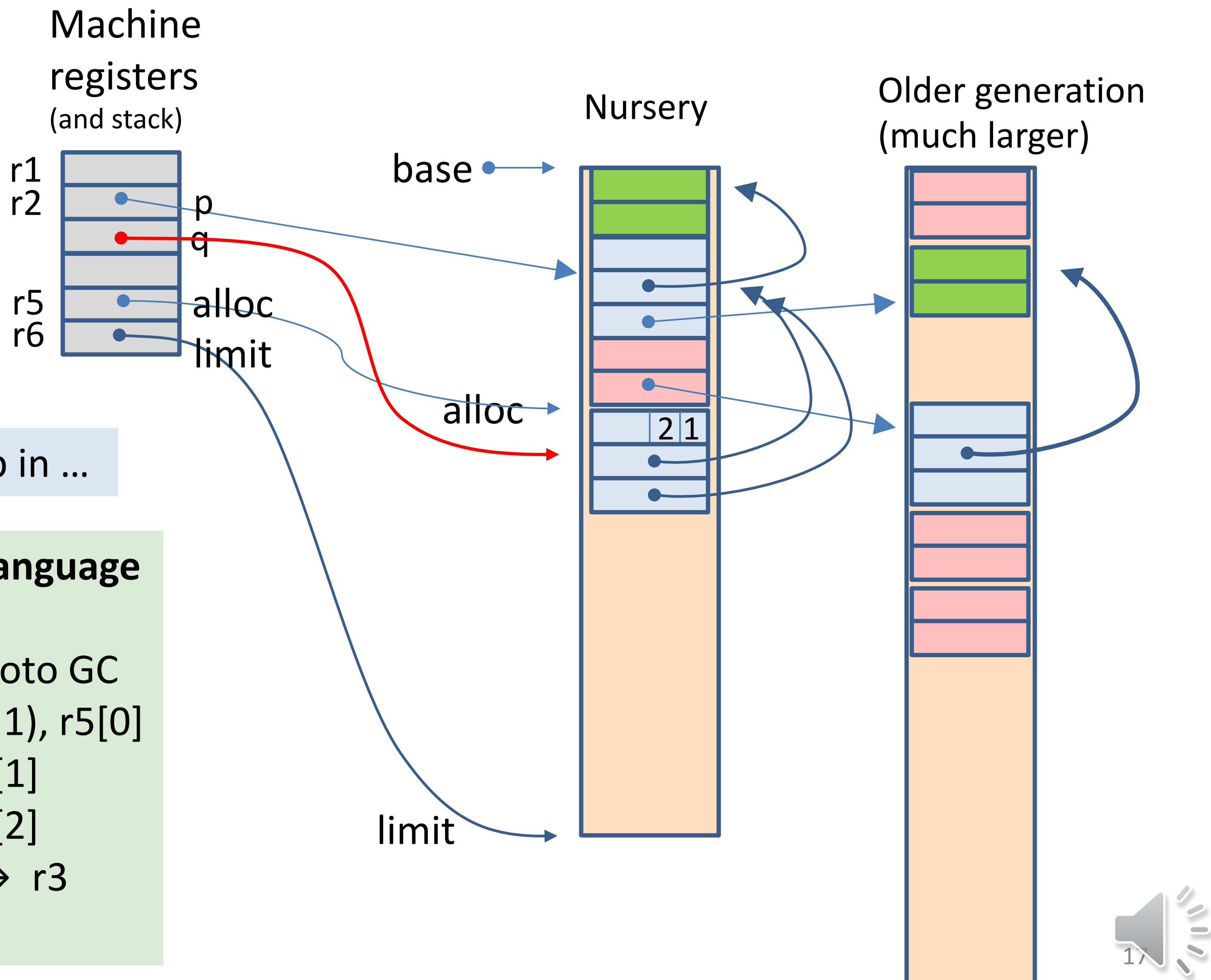


let q = D p p in …

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

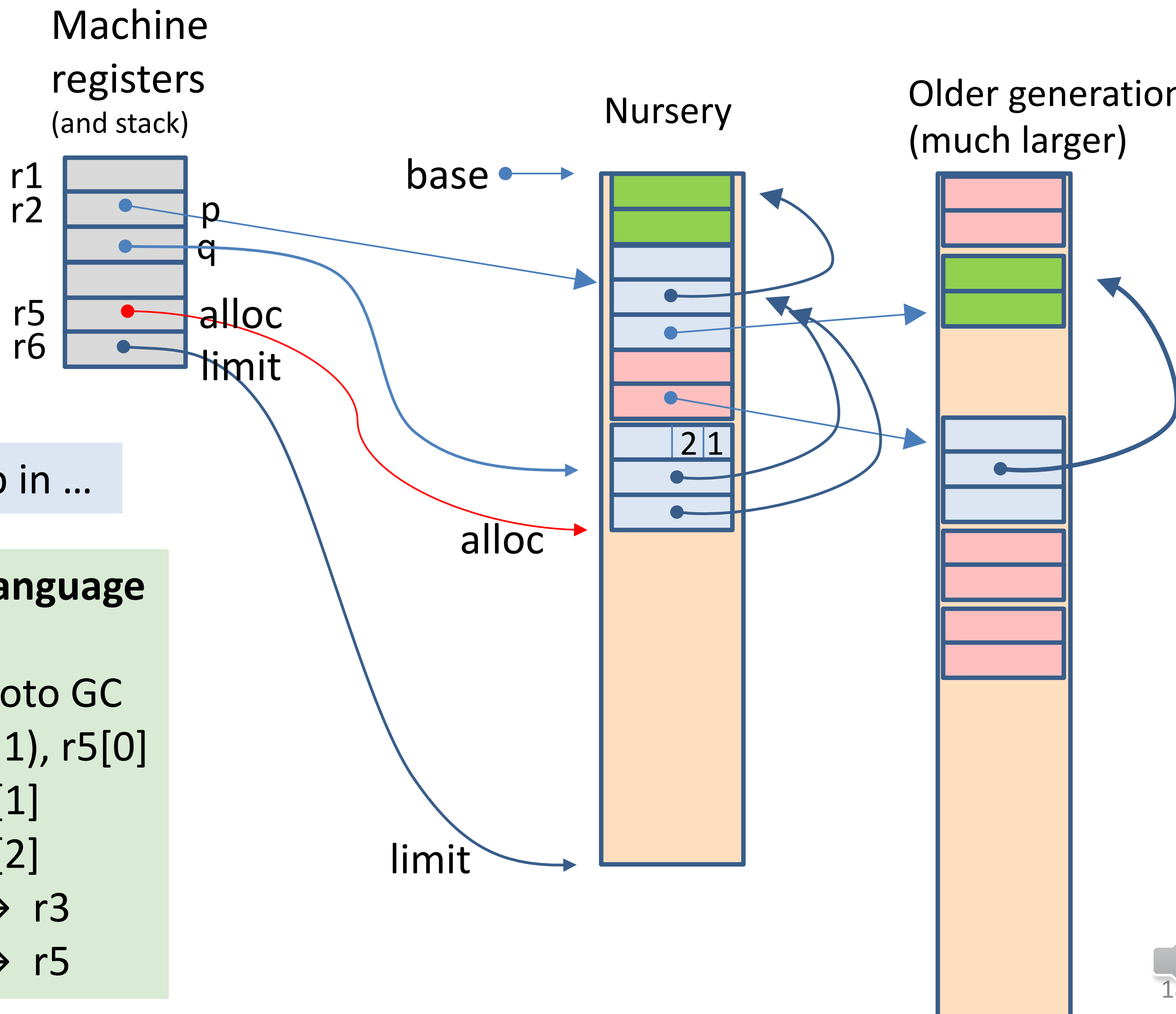


let q = D p p in …

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

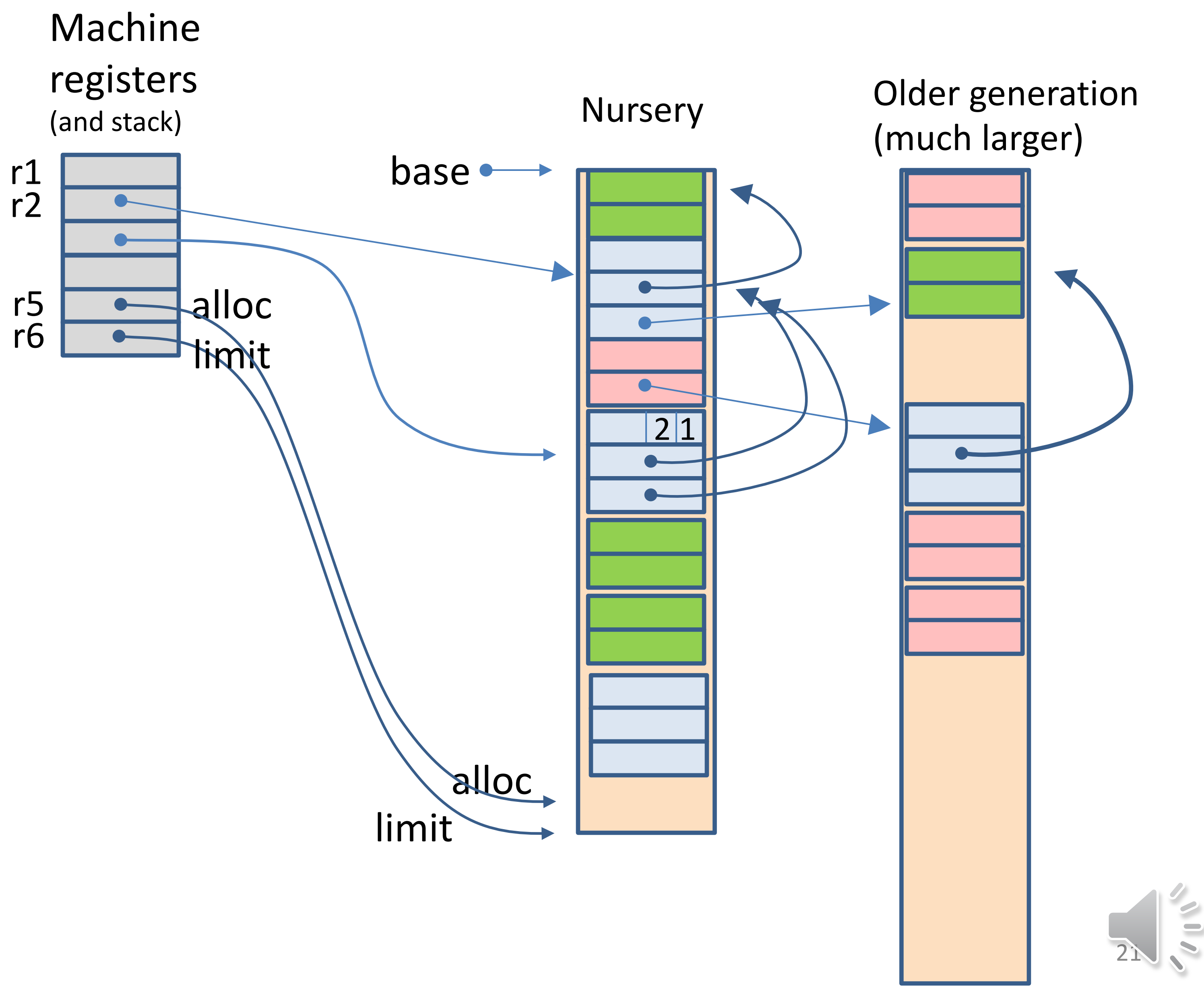
if r5+3>r6 goto GC	test for space available	2 instructions
store (0|2|1), r5[0]	store the header word	
store r2, r5[1]	store first field	initialize the fields
store r2, r5[2]	store second field	
add r5+1 → r3	assign the result (q)	2 instructions
add r5+3 → r5	adjust the "alloc" pointer	

What happens

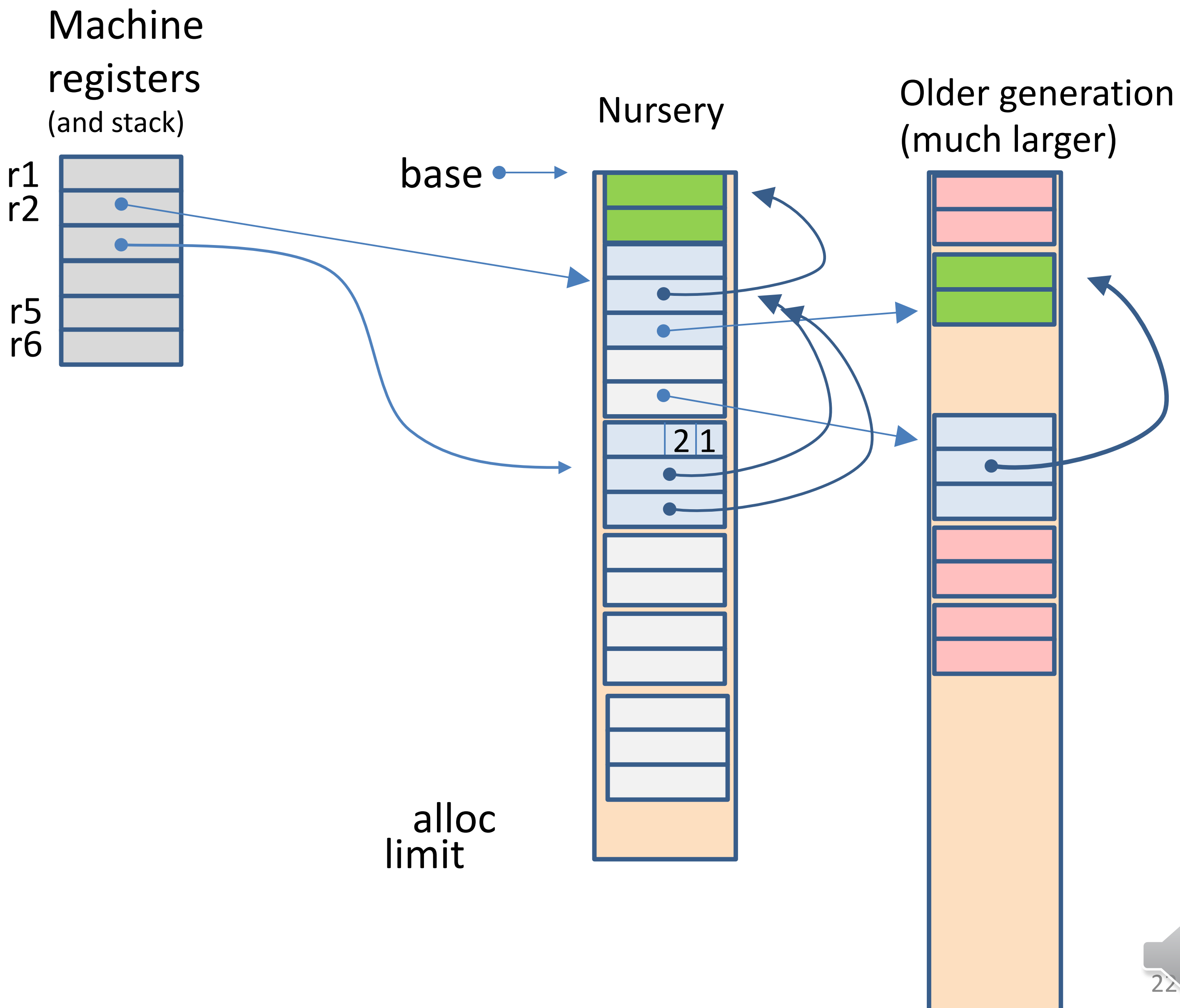
WHEN THE NURSERY FILLS UP . . .

GARBAGE COLLECTION!

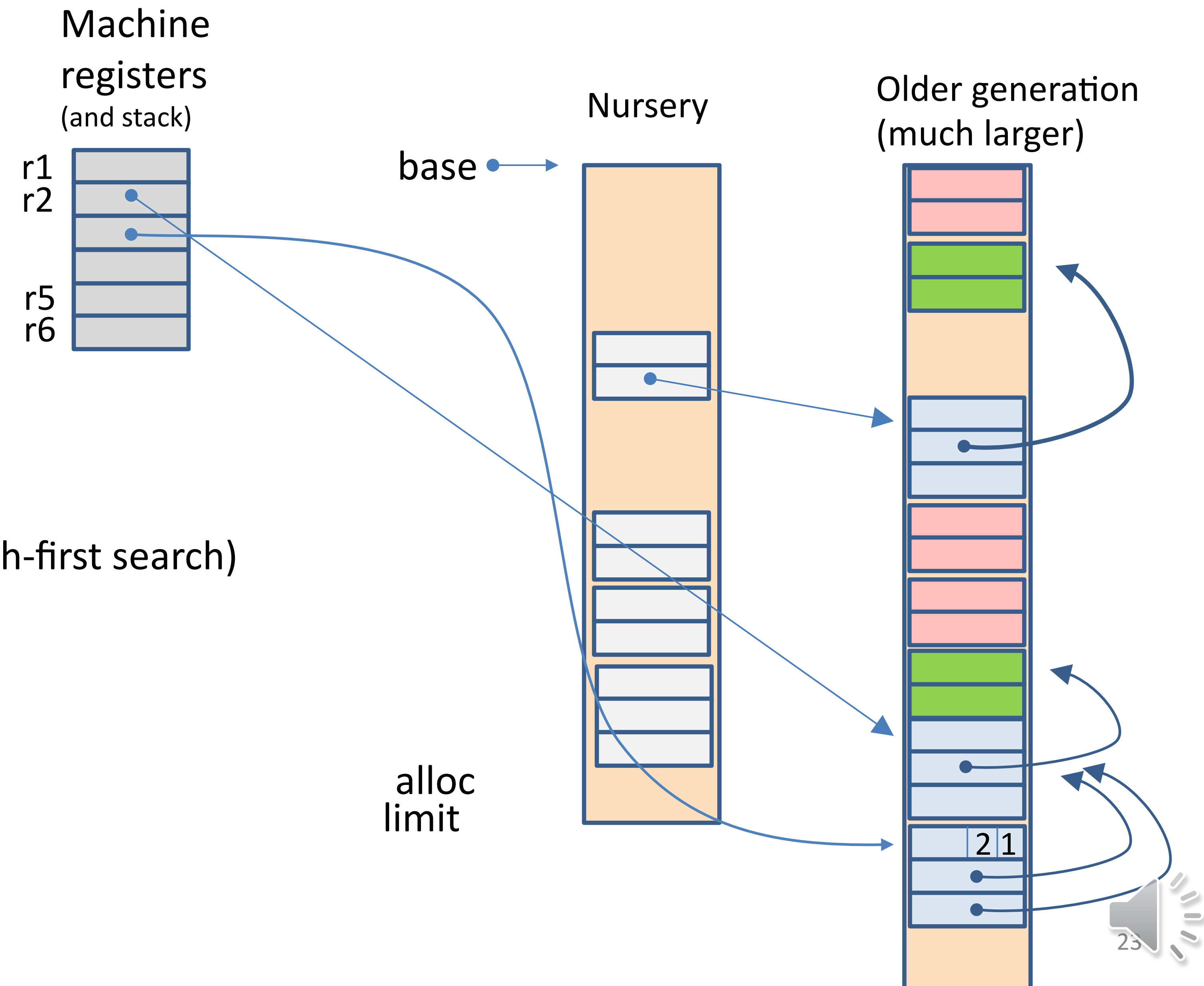
The nursery is full



Only these records are reachable

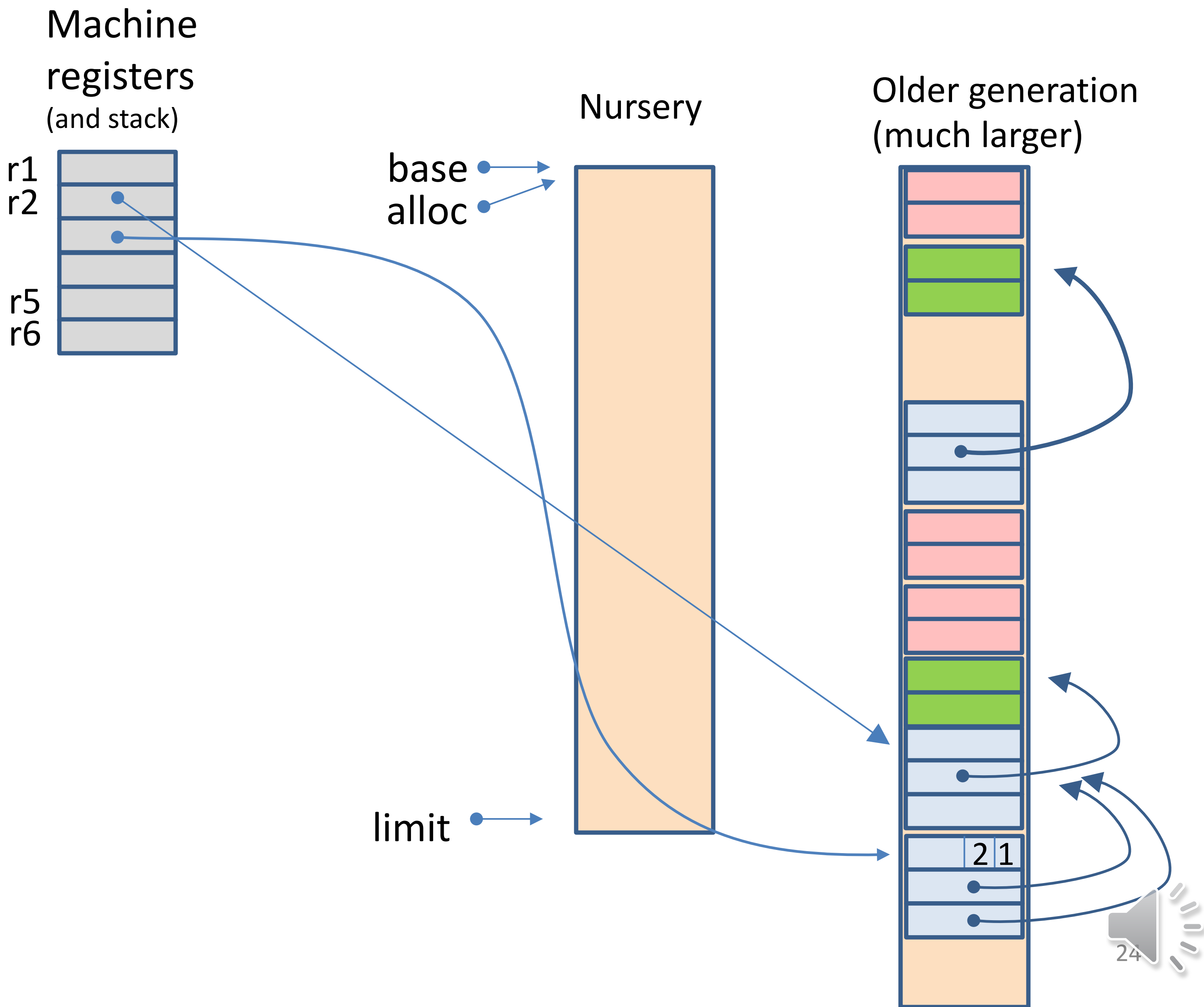


Move reachable records to older generation



(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:

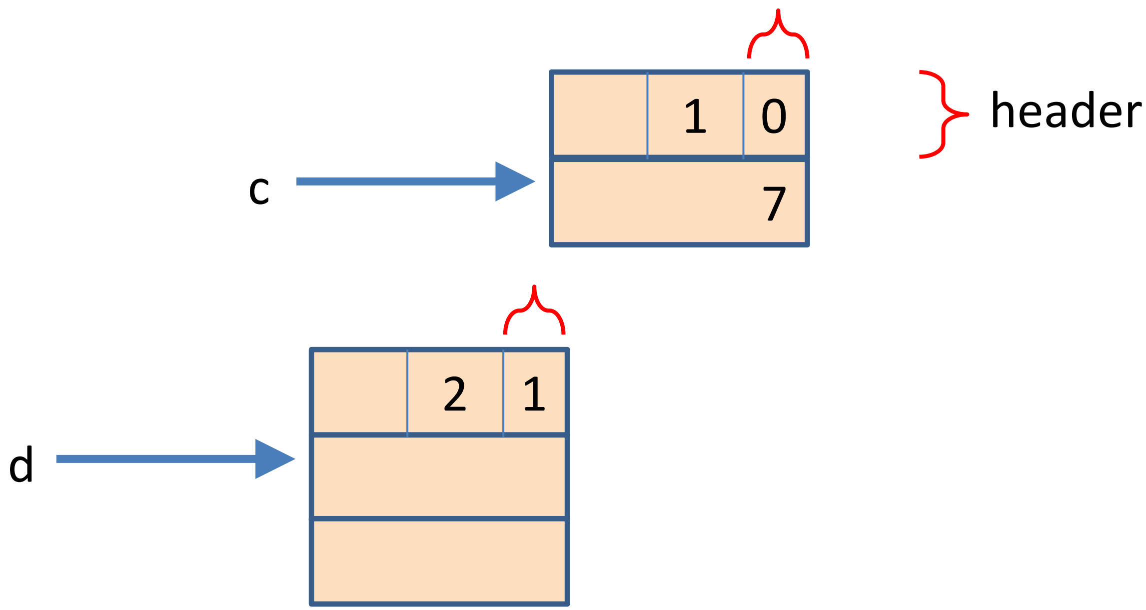
Pattern-matching

```
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:



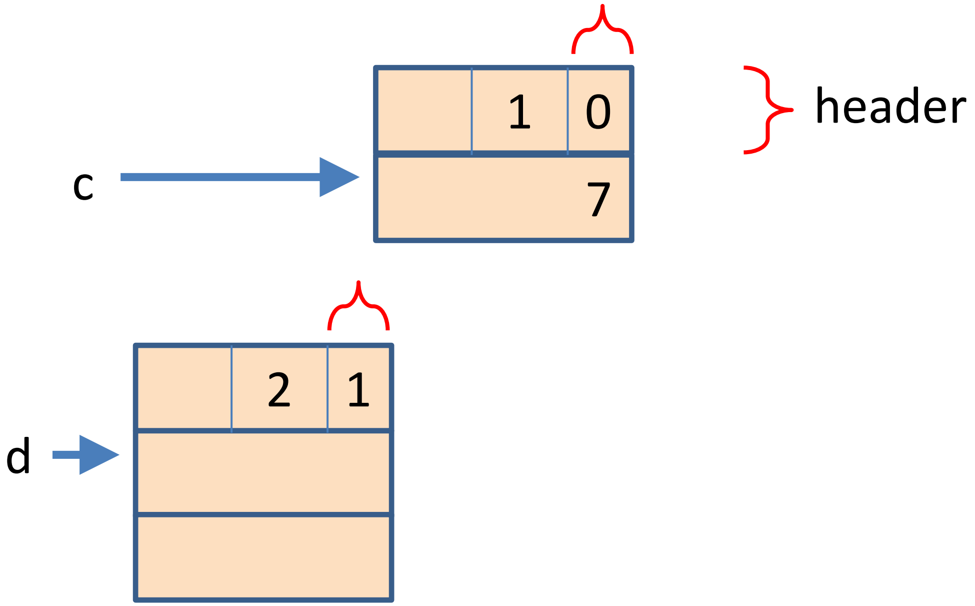
Pattern-matching

```
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 ...

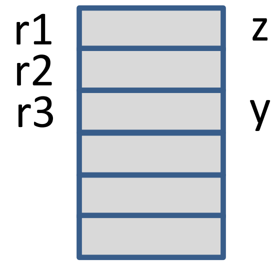
let x = y + z in ...

Almost as simple as,

~~**Assembly language**~~

~~add r3+r1 → r4~~

Machine registers (and stack)



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:

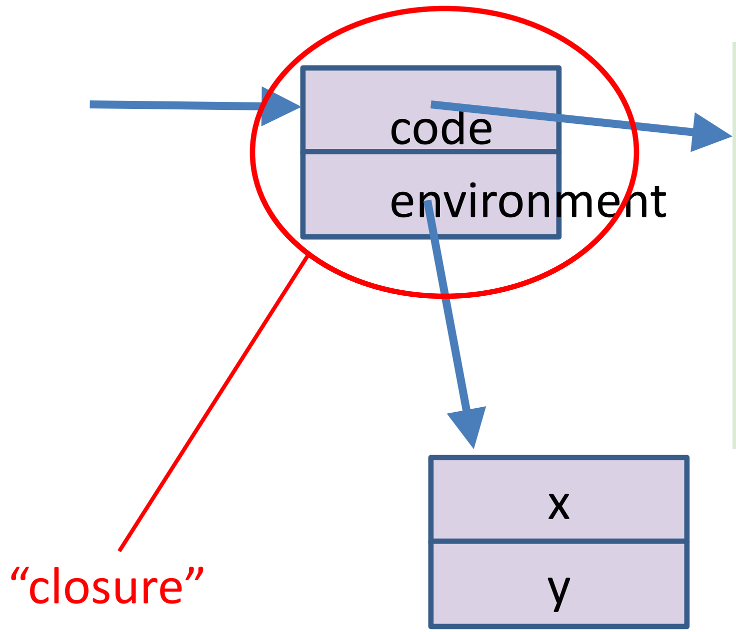
um, how do I know the values of x and y?

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

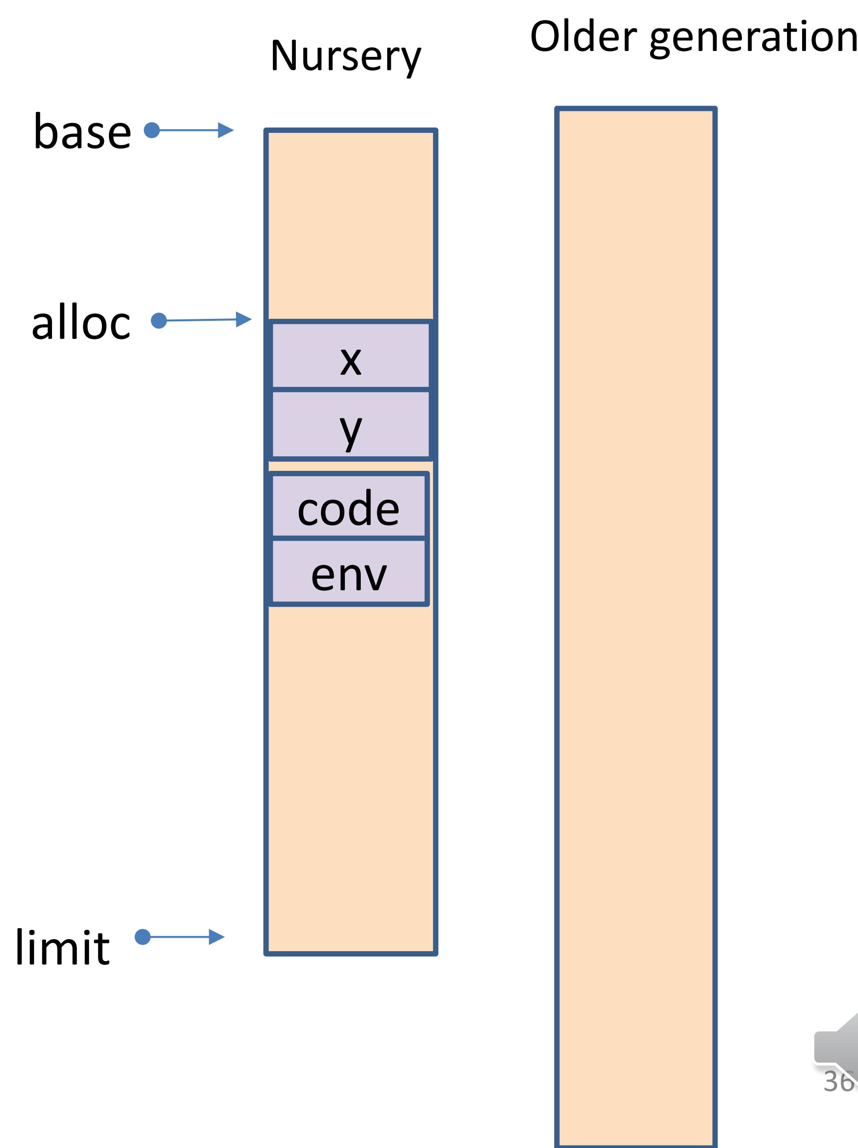
(fun w -> x+w+y)

Evaluating “fun … -> …”

is like constructing two records on the heap

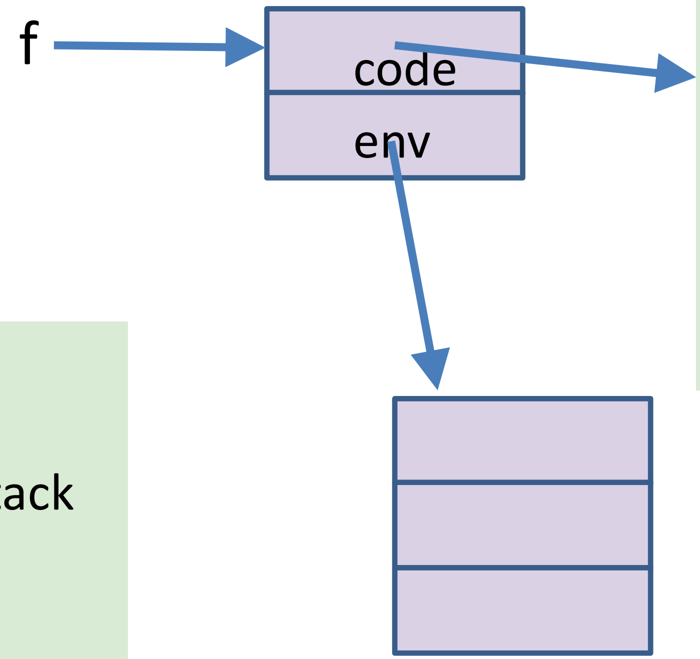
. . .

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

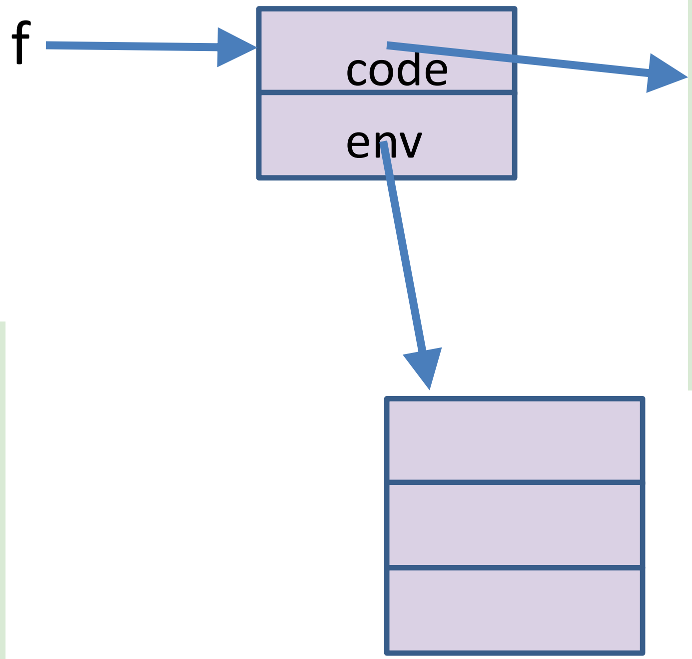
ret

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