GC continued: Reference counting, two-space collectors

David Bremner

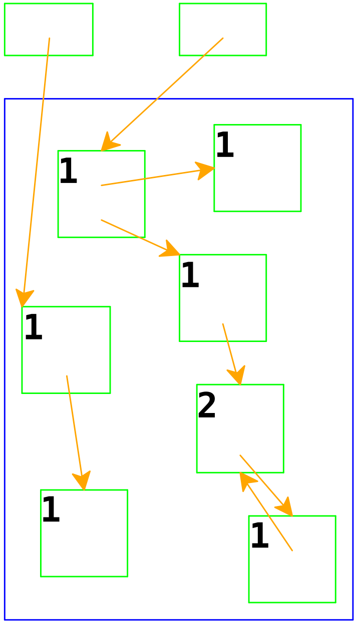
March 31, 2025

Reference Counting

Reference counting: a way to know whether a record has other users

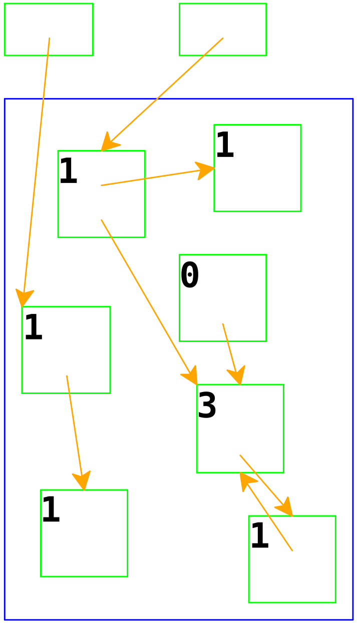
- Attach a count to every record, starting at 0
- When installing a pointer to a record (into a root or another record), increment its count
- When replacing a pointer to a record, decrement its count
- When a count is decremented to 0, decrement counts for other records referenced by the record, then free it

Reference Counting



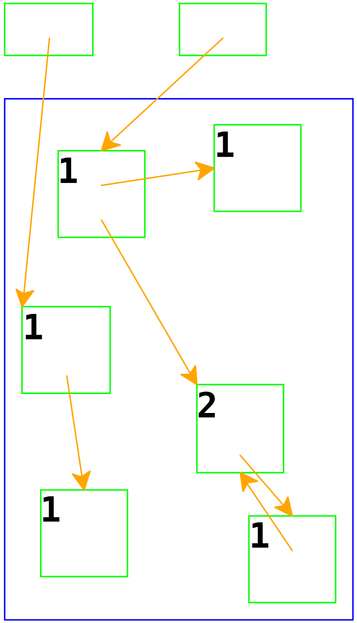
- references outside the main box are *roots*

Reference Counting



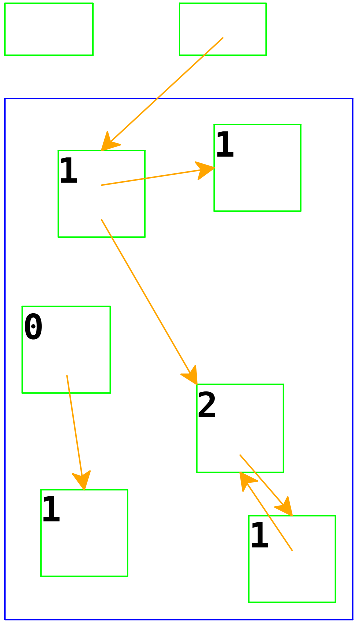
Adjust counts when a pointer is changed...

Reference Counting



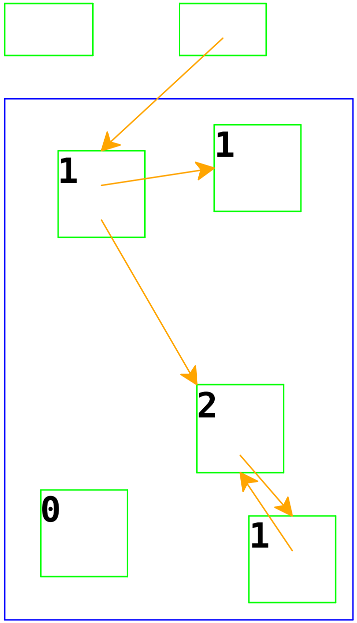
… freeing a record if its count goes to 0

Reference Counting



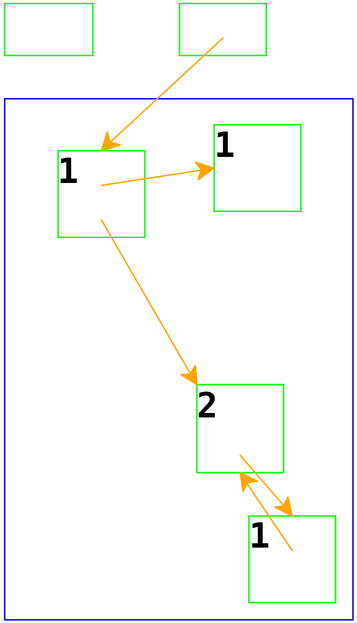
Same if the pointer is in a root

Reference Counting



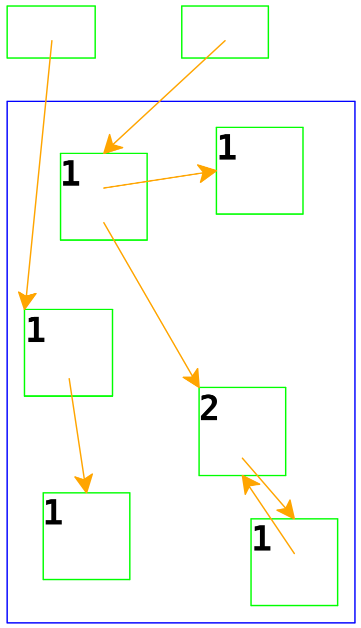
Adjust counts after frees, too...

Reference Counting



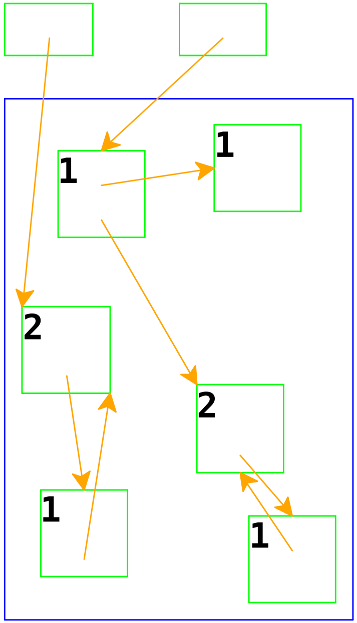
… which can trigger more frees

Reference Counting And Cycles



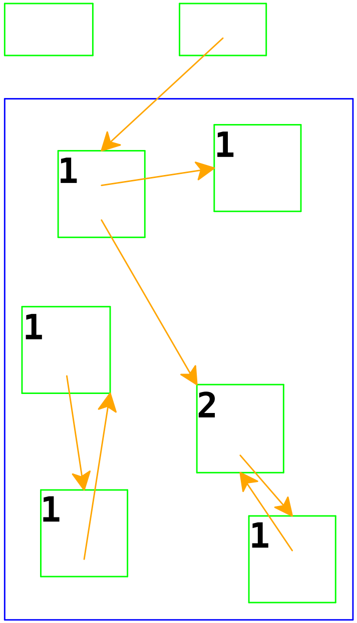
An assignment can create a cycle...

Reference Counting And Cycles



Adding a reference increments a count

Reference Counting And Cycles



Lower-left records are inaccessible, but not deallocated

In general, cycles break reference counting

Pros and Cons of reference counting

Pros

- simple
- tracing pauses are not needed (concurrency is easier).
- predictable destructors

Cons

- Overhead on every reference update
- Ripple out can be expensive
- Space overhead for counters
- Cache effects from updating counters
- Cycles need some special handling, or live forever.

Two-Space Copying Collectors

A two-space copying collector compacts memory as it collects, making allocation easier.

Allocator

- Partitions memory into to-space and from-space
- Allocates only in to-space

Collector

- Starts by swapping to-space and from-space
- Coloring gray $\rightarrow$ copy from from-space to to-space
- Choosing a gray record $\rightarrow$ walk once though the new to-space, update pointers

Allocator fast-path

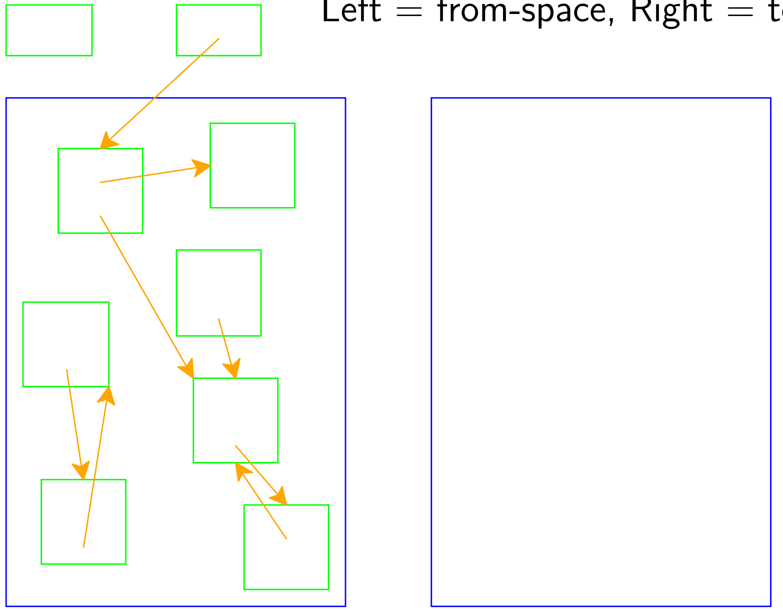
```
(define (malloc n some-roots more-roots)
  (define addr (heap-ref (alloc-ptr)))
  (cond
    [(<= (+ addr n) (space-limit))
     (heap-set! (alloc-ptr) (+ addr n))
     addr]
    [else
      ; ⋮
     ]))

(define (gc/alloc n)
  (define addr (heap-ref (alloc-ptr)))
  (unless (<= (+ addr n) (space-limit))
    (error 'gc/alloc "no space"))
  (heap-set! (alloc-ptr) (+ addr n)) addr)
```

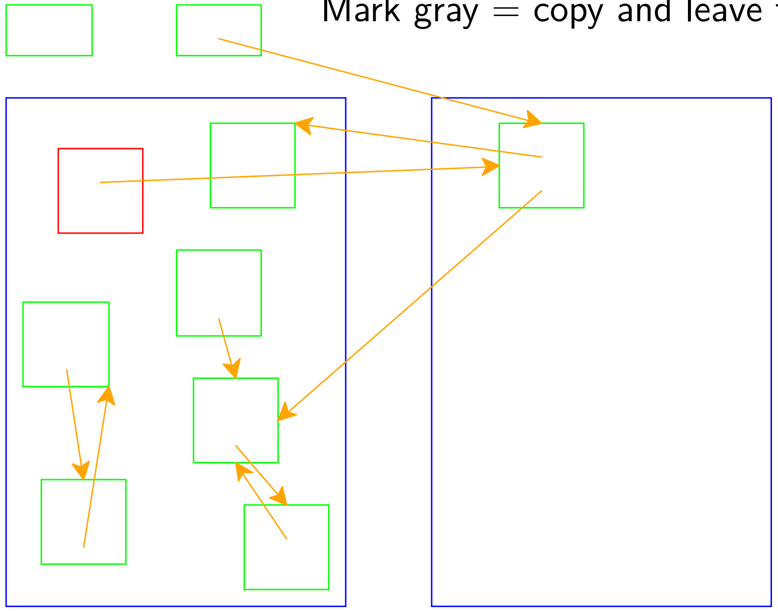
Allocator slow path

```
(collect-garbage some-roots more-roots)
(define next (heap-ref (alloc-ptr)))
(unless (<= (+ next n) (space-limit))
  (error 'alloc "no space"))
(heap-set! (alloc-ptr) (+ next n))
;; check for remaining forward info
(unless (or (at-from-space? some-roots)
            (at-from-space? more-roots))
  (free-from-space))
next
```

Left = from-space, Right = to-space



Mark gray = copy and leave forward address

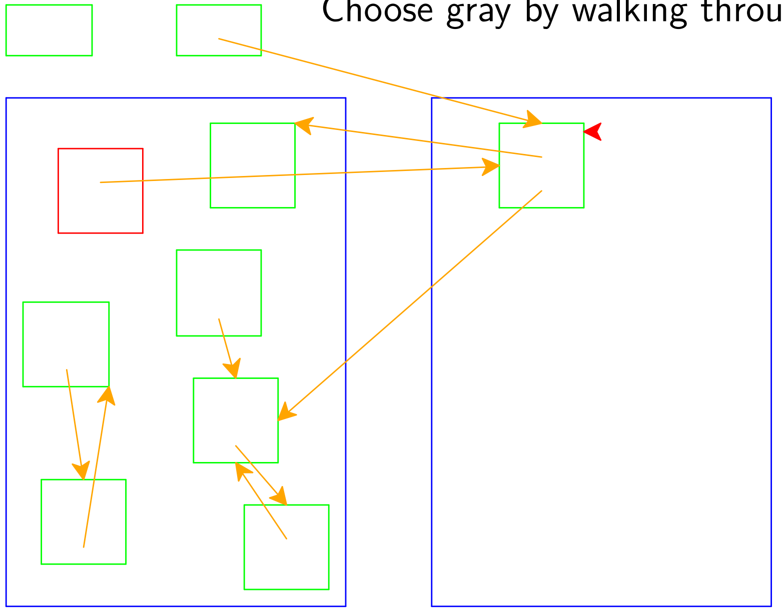


Copy and forward

```
(case (heap-ref loc)
  [(flat) (define new-addr (gc/alloc 2))
          (heap-set! new-addr 'flat)
          (heap-set! (+ new-addr 1)
                     (heap-ref (+ loc 1)))
          (heap-set! loc 'frwd)
          (heap-set! (+ loc 1) new-addr)
          new-addr]
   ; ⋮

  [(frwd) (heap-ref (+ loc 1))]
  [else (error 'forward/loc "wrong tag at ~a" loc)])
```

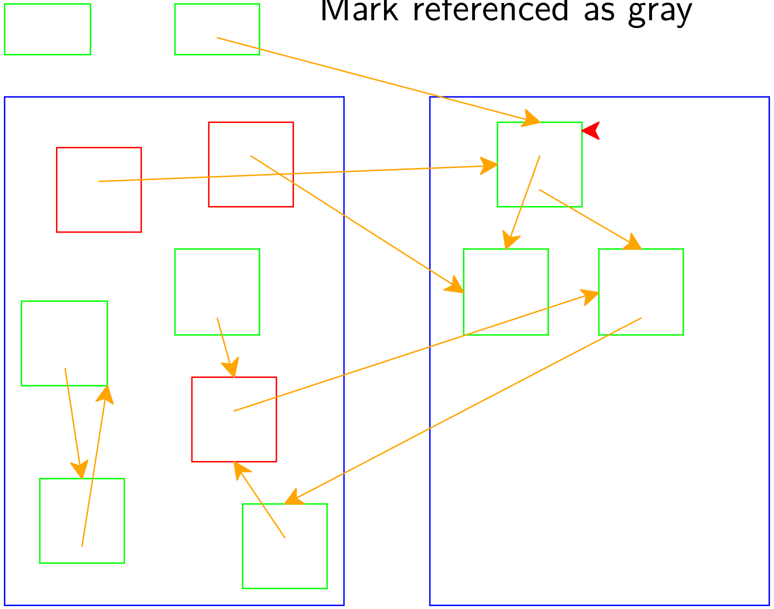
Choose gray by walking through to-space



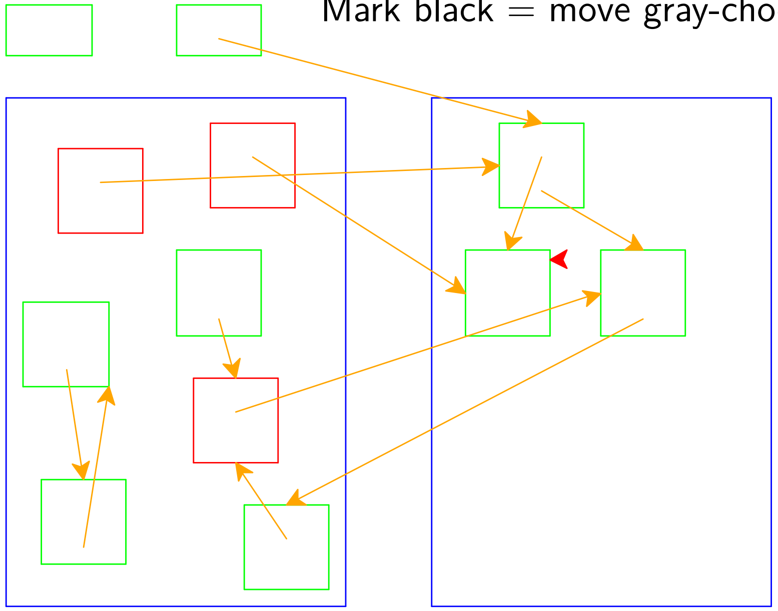
Walking to-space

```
(define (forward/ref loc)
  (cond
    [(= loc (heap-ref (alloc-ptr))) (void)]
    [else
     (case (heap-ref loc)
       [(flat) (forward/ref (+ loc 2))]
       [(cons)
        (gc:set-first! loc (forward/loc
                             (heap-ref (+ loc 1))))
        (gc:set-rest! loc (forward/loc
                            (heap-ref (+ loc 2))))
        (forward/ref (+ loc 3))]
        ; ⋮
       [else (error 'forward/ref "wrong tag at ~a"
               loc)])]))
```

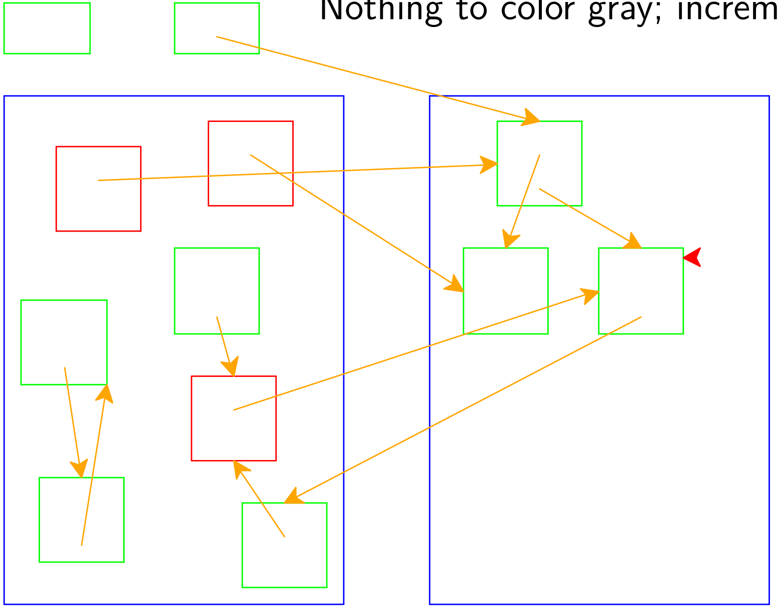
Mark referenced as gray



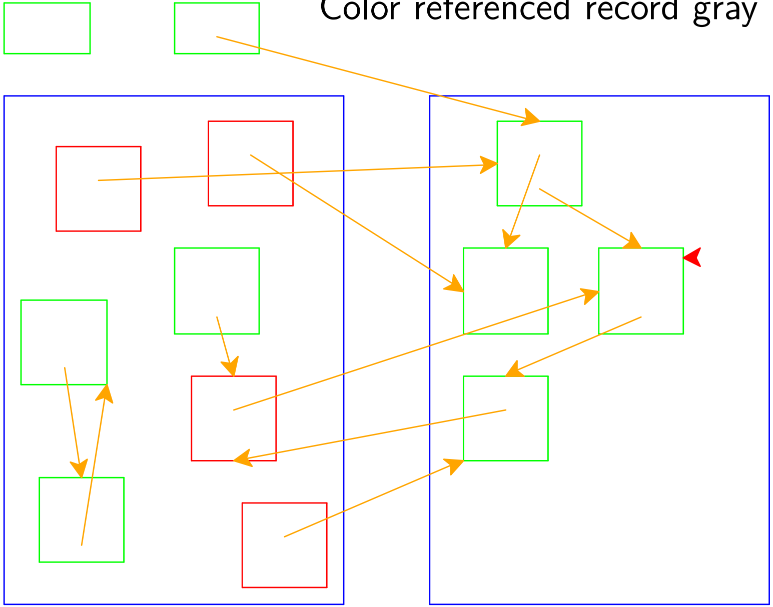
Mark black = move gray-choosing arrow



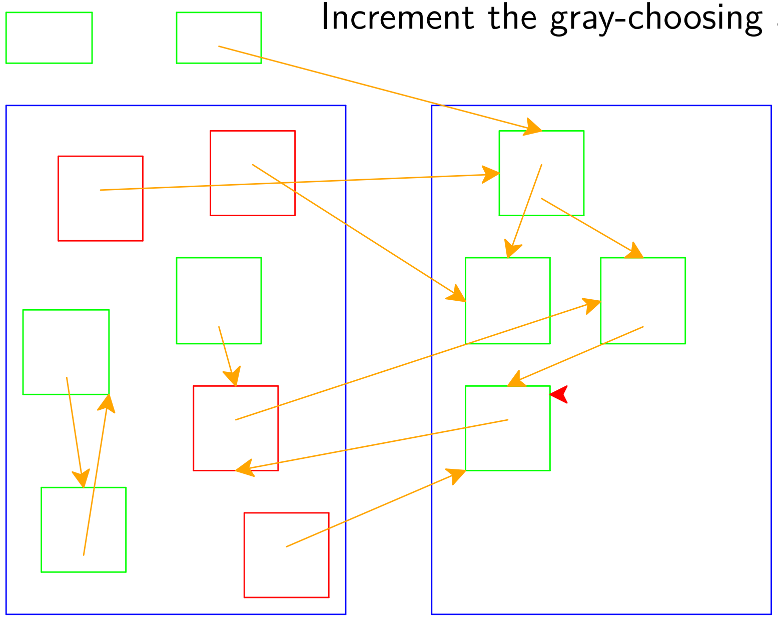
Nothing to color gray; increment the arrow



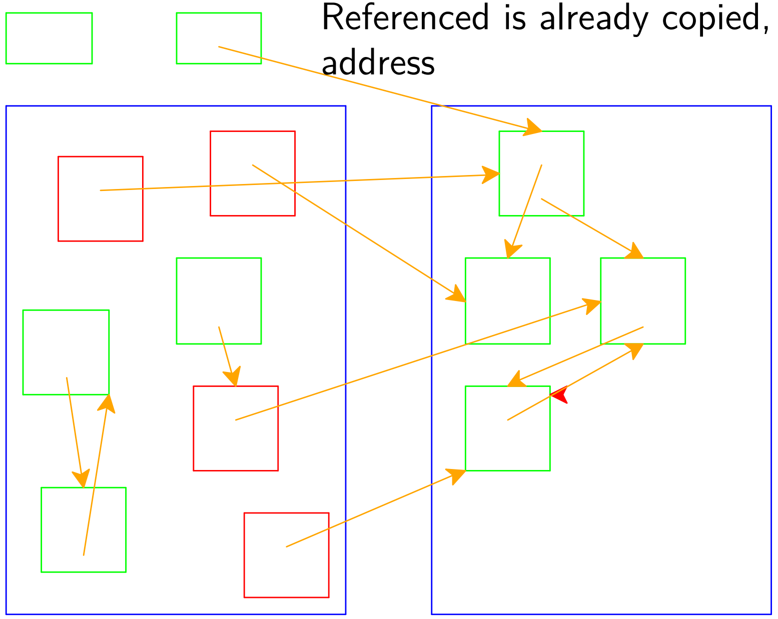
Color referenced record gray



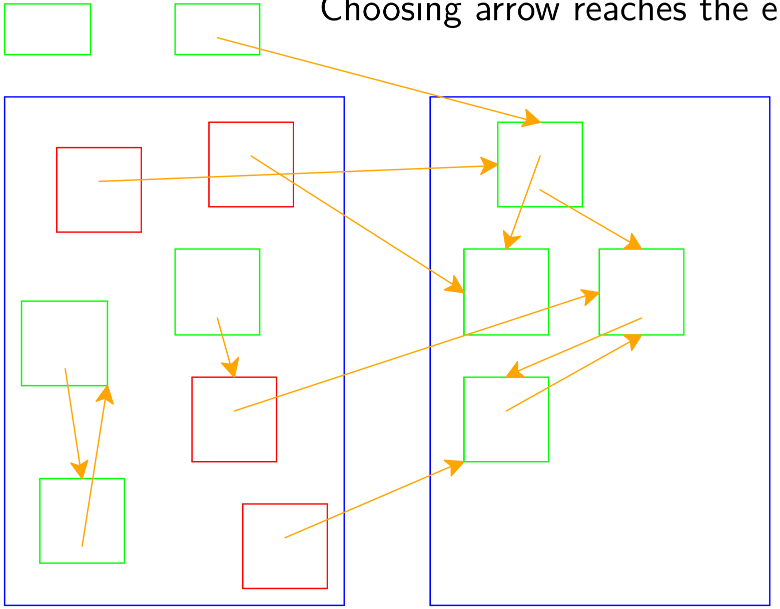
Increment the gray-choosing arrow



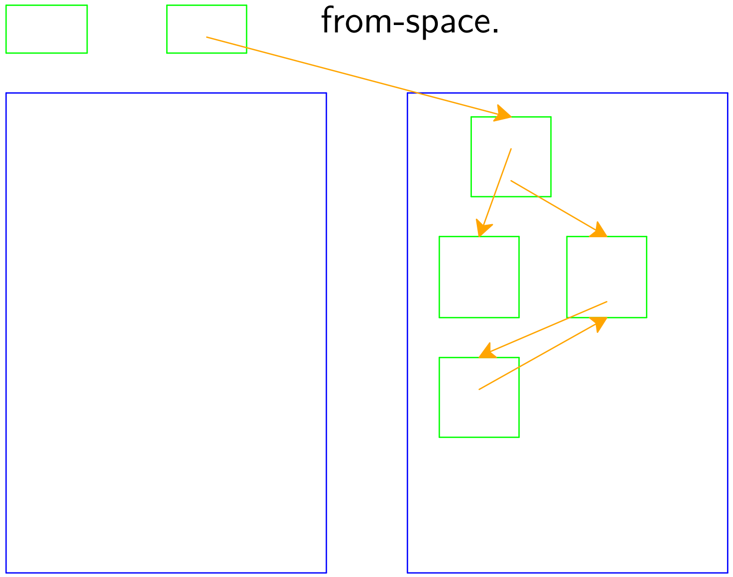
Referenced is already copied, use forwarding address



Choosing arrow reaches the end of to-space: done



Next collection: Left = to-space; Right = from-space.



Fib, again

fib

```
(allocator-setup "copying.rkt" 160)
(define (fib n)
  (cond
    [(<= n 1) 1]
    [else (+ (fib (- n 1)) (fib (- n 2)))]))

(fib 5)
```

Two-Space Numeric Example

memory 26-byte (13 bytes for each space), 2 registers

tags 1: integer, 2: pointer, 3: (integer, pointer), 99: moved

```
      Register 1: 7                Register 2: 0

Addr:       00 01 02 03 04 05 06 07 08 09 10 11 12
From:       01 75 02 00 03 02 10 03 02 02 03 01 04
      Register 1: 7                Register 2: 0

Addr:       00 01 02 03 04 05 06 07 08 09 10 11 12
From:      |01 75|02 00|03 02 10|03 02 02|03 01 04|
      Register 1: 0                Register 2: 0

Addr:       00 01 02 03 04 05 06 07 08 09 10 11 12
```

Acknowledgements

- Lecture 19 based in part on slides by Vincent St. Amour.
- Copying collector from Master's Thesis of Yixi Zhang
  https://github.com/yixizhang/racket-gc/