

Interpreter with Continuations

```
(define (eval-expression exp env cont)
  (cases expression exp
    (lit-exp (datum)
      (apply-cont cont datum))
    (var-exp (id)
      (apply-cont cont (apply-env env id)))
    (proc-exp (id body-exp)
      (apply-cont cont
        (closure id body-exp env)))
    ...))

(define (apply-cont cont val)
  (cases continuation cont
    (done-cont () val)
    ...))
```

Continuations and Gotos

```
(define (eval-expression exp env cont)
  (cases exp...
    (app-exp (rator rand)
      (eval-expression
        rator env
        (app-arg-cont rand env cont))))

=>

(define (eval-expression)
  (cases EXP ...
    (app-exp (rator rand)
      (set! EXP rator)
      ;; ENV stays the same
      (set! CONT (app-arg-cont rand ENV CONT))
      (eval-expression)) ; "goto"
```

Continuations and Gotos

```
(define (eval-expression exp env cont)
  (cases exp ...
    (proc-exp (id body-exp)
      (apply-cont cont
        (closure id body-exp env)))

=>

;; registers:
(define EXP ...) (define CONT ...) ...

(define (eval-expression)
  (cases EXP ...
    (proc-exp (id body-exp)
      (set! VAL (closure id body-exp ENV))
      ;; CONT stays the same
      (apply-cont)) ; "goto"
```

Continuations and Gotos

- Registers and gotos explain why the following program never generates a stack overflow:

```
let f = proc(f) proc(n) ((f f) n)
in ((f f) 0)
```

- So, can we compute arbitrarily deep recursions?

```
let f = proc(f)
  proc(n)
    if n then +(1, ((f f) -(n, 1)))
    else 0
in ((f f) 1000000000)
```

No...

Allocation

- We've avoided stack allocation
- But we still have to allocate
 - continuation records
 - closures
 - environment records

Allocation

- Where do we call `malloc` ?

```
(define (eval-expression)
  (cases EXP ...
    (proc-exp (id body-exp)
      (set! VAL (closure id body-exp ENV))
      ;; CONT stays the same
      (apply-cont))
    (app-exp (rator rand)
      (set! EXP rator)
      ;; ENV stays the same
      (set! CONT (app-arg-cont rand ENV CONT))
      (eval-expression))
    ...
```

Allocation

- Where do we call `malloc` ?

```
(define (eval-expression)
  (cases EXP ...
    (proc-exp (id body-exp)
      (set! VAL (closure id body-exp ENV))
      ;; CONT stays the same
      (apply-cont))
    (app-exp (rator rand)
      (set! EXP rator)
      ;; ENV stays the same
      (set! CONT (app-arg-cont rand ENV CONT))
      (eval-expression))
    ...
```

Exposing Allocation

```
(define (closure id body env)
  (let ([v (malloc 4)])
    (mem-set! v 0 closure-tag)
    (mem-set! v 1 id)
    (mem-set! v 2 body)
    (mem-set! v 3 env)
    v))

(define (closure? v)
  (= (mem-ref v 0) closure-tag))

(define (closure->id v)
  (mem-ref v 1))
... 
```

Memory Allocator

```
(define memory (make-vector 200))
(define allocated 0)

(define (malloc size)
  (let ([result allocated])
    (set! allocated (+ allocated size))
    result))

(define (mem-set! a n v)
  (vector-set! memory (+ a n) v))

(define (mem-ref a n)
  (vector-ref memory (+ a n)))
```

Exposing Allocation

- Use of `malloc` explains why the following program runs out of memory:

```
let f = proc(f)
  proc(n)
    if n then +(1, ((f f) -(n, 1)))
    else 0
in ((f f) 1000000000)
```

- Each call to `(f f)` extends the continuation
- Eventually, the continuation fills all memory

Exposing Allocation

- Does the following program run forever?

```
let f = proc(f) proc(n) ((f f) n)
in ((f f) 0)
```

- Each call to `(f f)`
 - creates an extended environment
 - creates a new closure

We need *deallocation*

Deallocation

- Where do we call `free` ?

```
(define (apply-cont)
  (cond ...
    [(app-cont? CONT)
     (let ([rator (app-cont->rator CONT)]
           [old-cont (app-cont->cont CONT)])
       (set! EXP (closure->body rator))
       (set! ENV (extend-env
                    (closure->id rator)
                    VAL
                    (closure->env rator)))
       (set! CONT old-cont))
     (eval-expression)]
    ...
```

Deallocation

- Where do we call `free` ?

```
(define (apply-cont)
  (cond ...
    [(app-cont? CONT)
     (let ([rator (app-cont->rator CONT)]
           [old-cont (app-cont->cont CONT)])
       (set! EXP (closure->body rator))
       (set! ENV (extend-env
                  (closure->id rator)
                  VAL
                  (closure->env rator)))
       (free CONT) ;; unless letcc'd!
       (set! CONT old-cont))
     (eval-expression)]
    ...
```

Deallocation

- Where do we call `free` ?

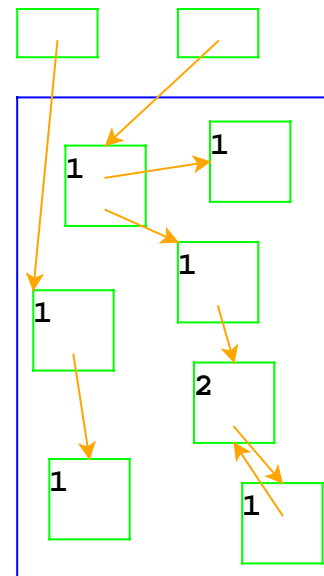
```
(define (apply-cont)
  (cond ...
    [(app-cont? CONT)
     (let ([rator (app-cont->rator CONT)]
           [old-cont (app-cont->cont CONT)])
       (set! EXP (closure->body rator))
       (free ENV) ;; unless in a closure!
       (set! ENV (extend-env
                  (closure->id rator)
                  VAL
                  (closure->env rator)))
       (set! CONT old-cont))
     (eval-expression)]
    ...
```

Reference Counting

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

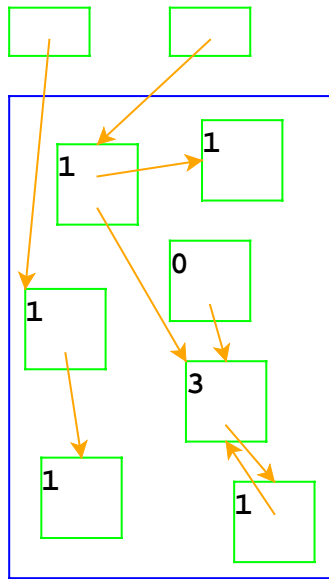
- Attach a count to every record, start at 0
- When installing a pointer to a record (into a register, or another record), increment its count
- When replacing a pointer to a record, decrement its count
- When a count is decremented to 0, decrement counts for other records referenced by the record, then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

Reference Counting



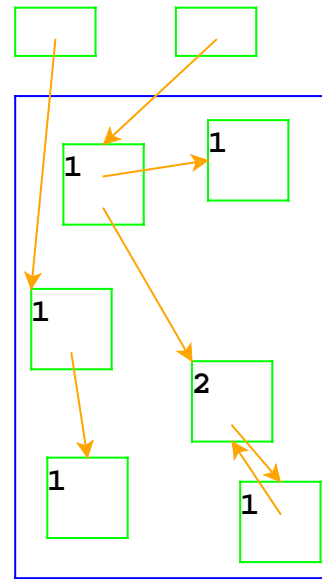
- Top boxes are the registers `ENV`, `CONT`, etc.
- Boxes in the blue area are allocated with `malloc`

Reference Counting



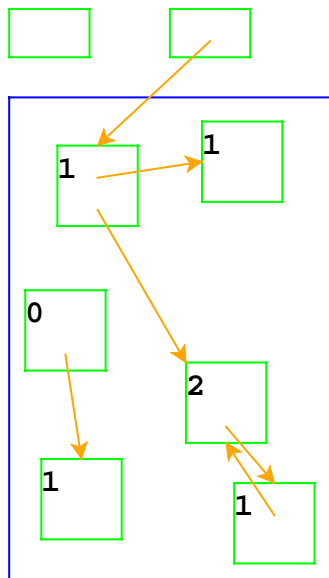
- Adjust counts when a pointer is changed...

Reference Counting



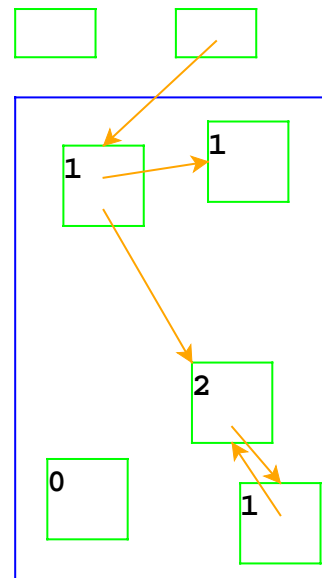
- ... freeing a record if its count goes to 0

Reference Counting



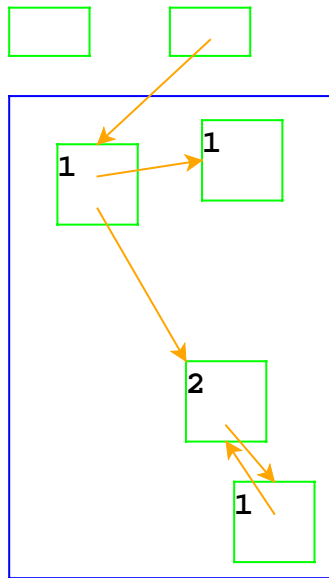
- Same if the pointer is in a register

Reference Counting



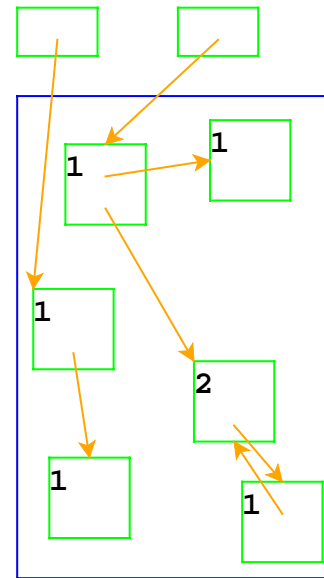
- Adjust counts after frees, too...

Reference Counting



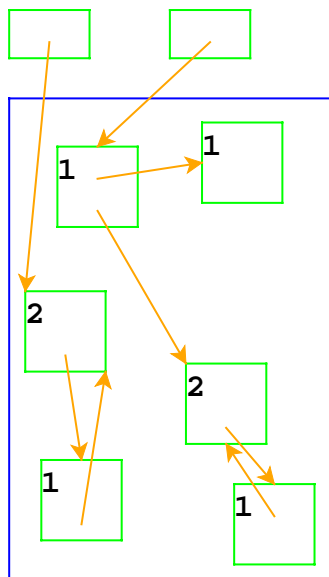
- ... which can trigger more frees

Reference Counting



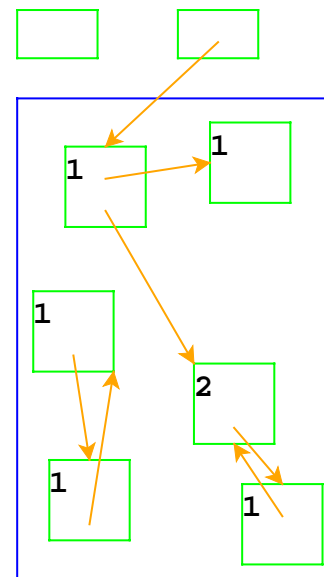
- Another example

Reference Counting



- Adding a reference increments a count

Reference Counting



- Lower-left records are inaccessible, but not deallocated
- In general, cycles break reference counting

Garbage Collection

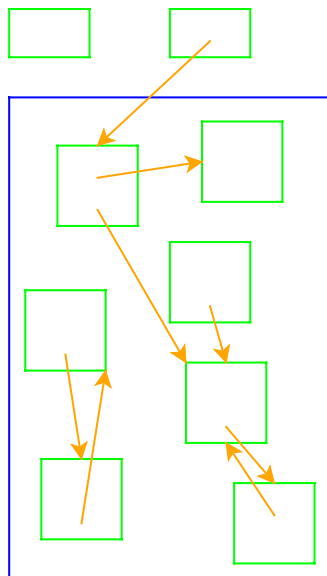
Garbage collection: a way to know whether a record is *accessible*

- A record referenced by a register is *live*
- A record referenced by a live record is also live
- A program can only possibly use live records, because there is no way to get to other records
- A garbage collector frees all records that are not live
- We'll allocate until we run out of memory, then run a garbage collector to get more space

Garbage Collection Algorithm

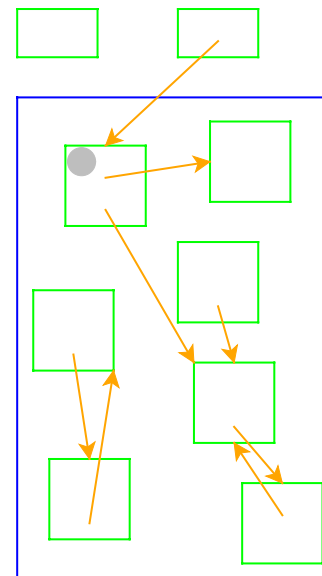
- Color all records *white*
- Color records referenced by registers *gray*
- Repeat until there are no gray records:
 - Pick a gray record, *r*
 - For each white record that *r* points to, make it gray
 - Color *r* *black*
- Deallocate all white records

Garbage Collection



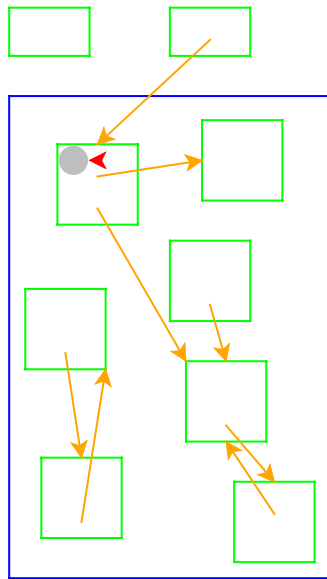
- All records are marked white

Garbage Collection



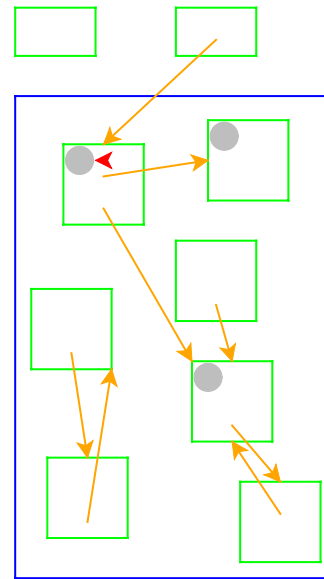
- Mark records referenced by registers as gray

Garbage Collection



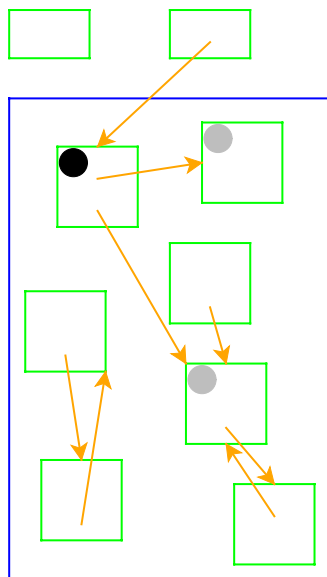
- Need to pick a gray record
- Red arrow indicates the chosen record

Garbage Collection



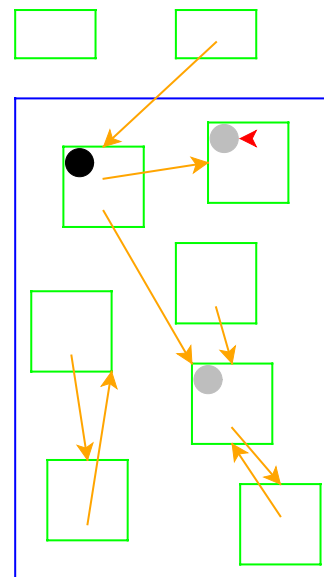
- Mark white records referenced by chosen record as gray

Garbage Collection



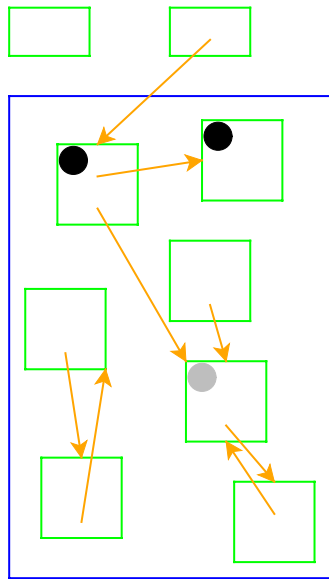
- Mark chosen record black

Garbage Collection



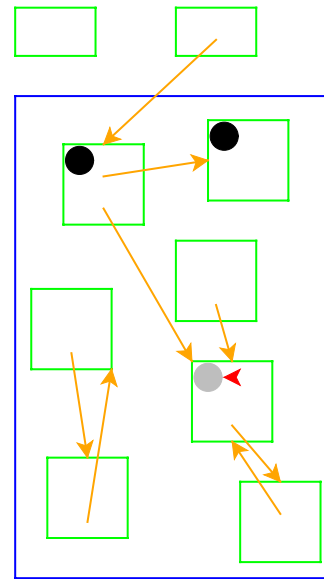
- Start again: pick a gray record

Garbage Collection



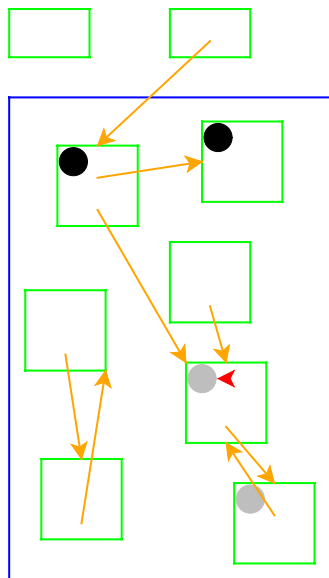
- No referenced records; mark black

Garbage Collection



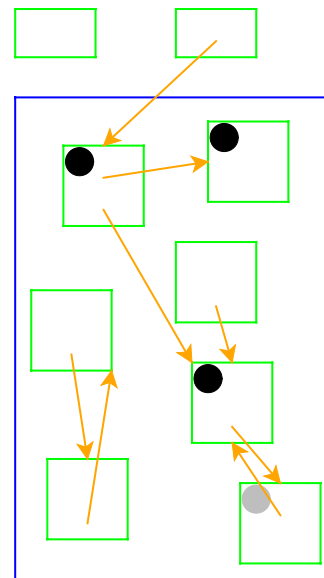
- Start again: pick a gray record

Garbage Collection



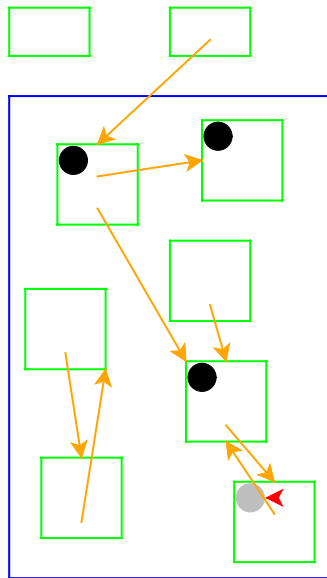
- Mark white records referenced by chosen record as gray

Garbage Collection



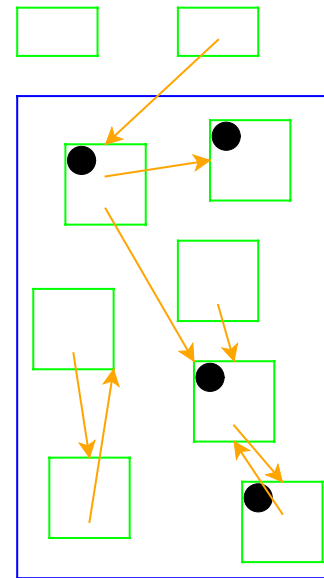
- Mark chosen record black

Garbage Collection



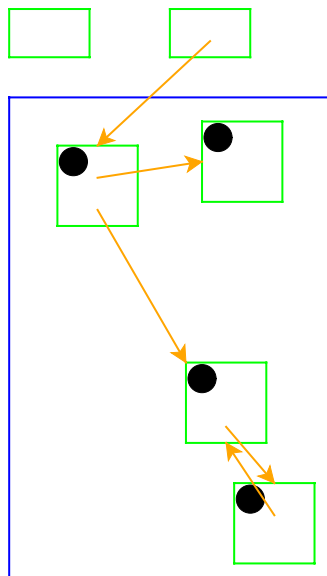
- Start again: pick a gray record

Garbage Collection



- No referenced white records; mark black

Garbage Collection



- No more gray records; deallocate white records
- Cycles **do not** break garbage collection

Two-Space Copying Collectors

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

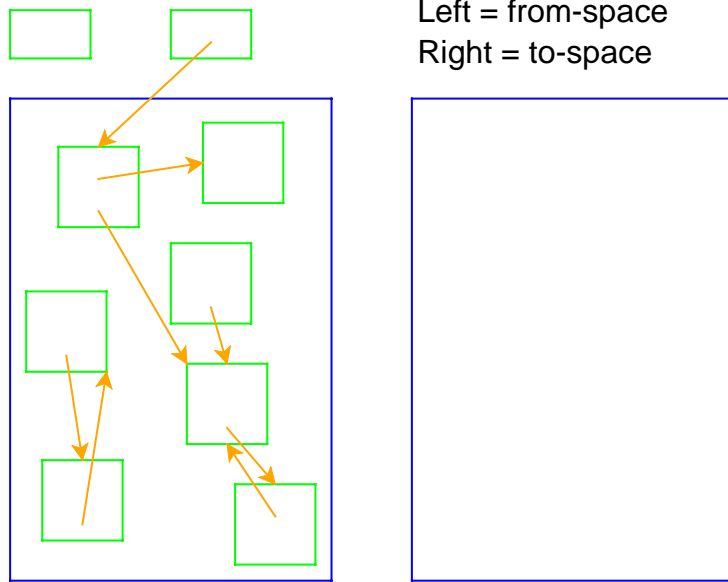
Allocator:

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

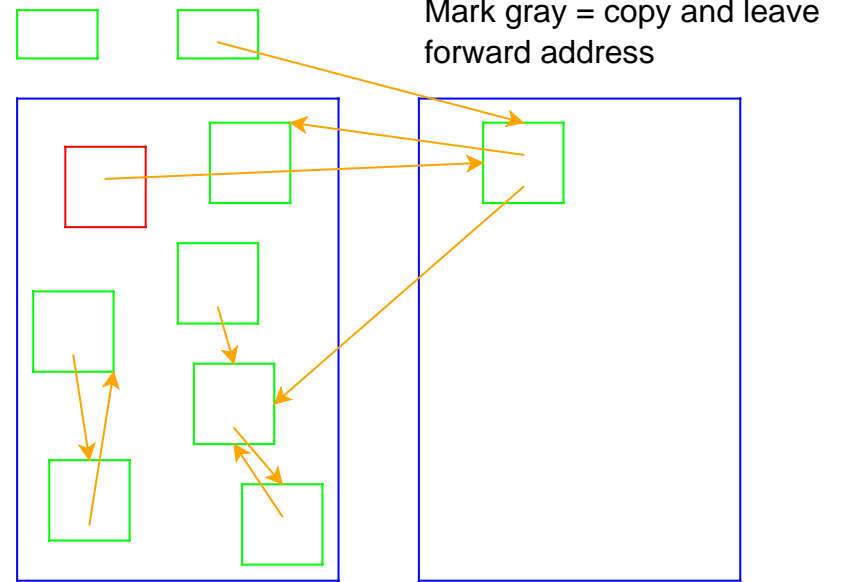
Collector:

- Starts by swapping **to-space** and **from-space**
- Coloring gray => copy from **from-space** to **to-space**
- Choosing a gray record => walk once through the new **to-space**, update pointers

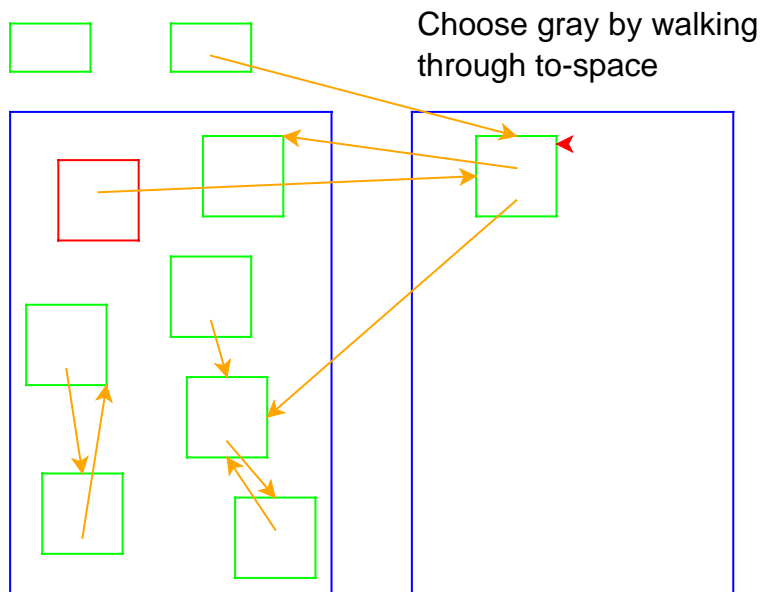
Two-Space Collection



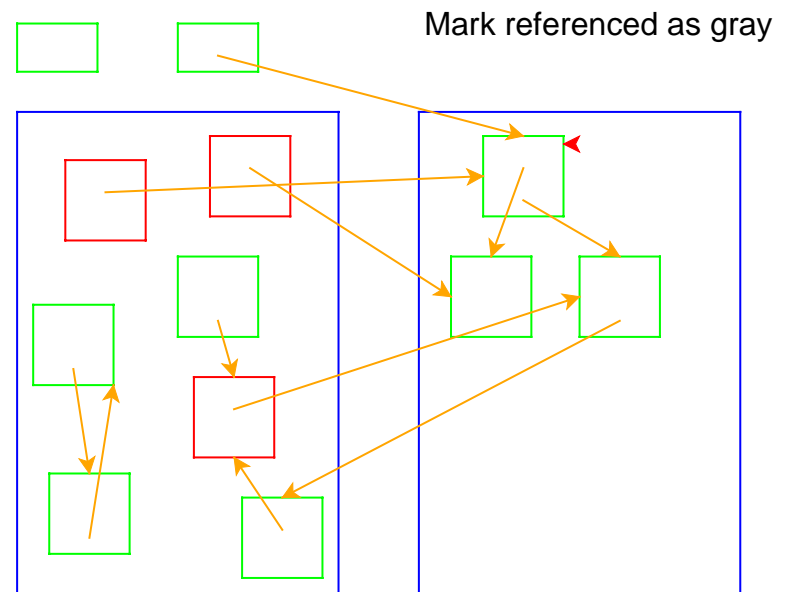
Two-Space Collection



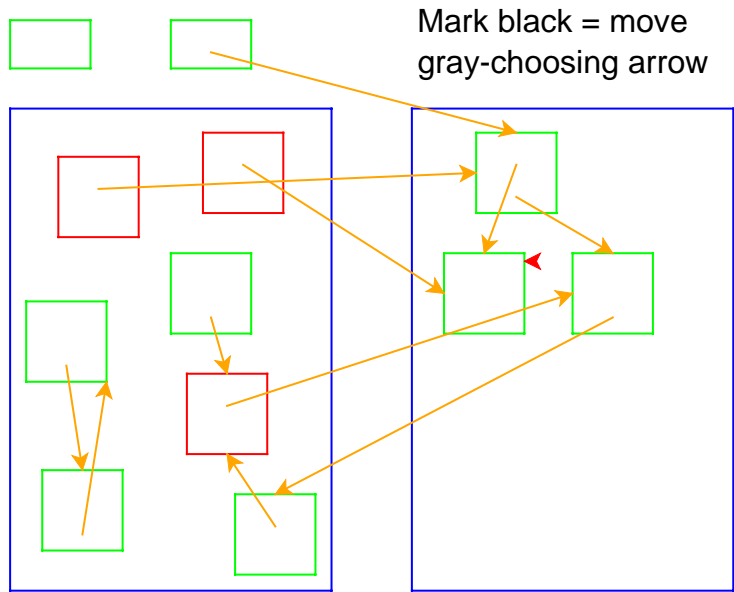
Two-Space Collection



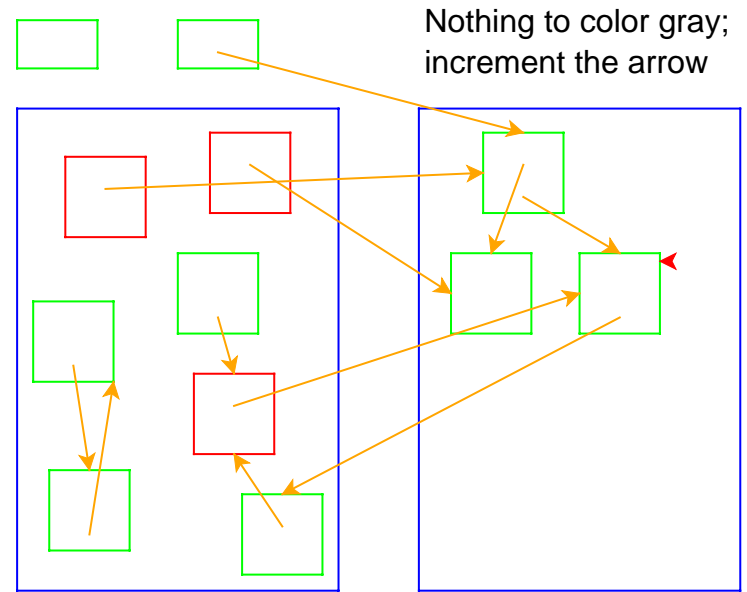
Two-Space Collection



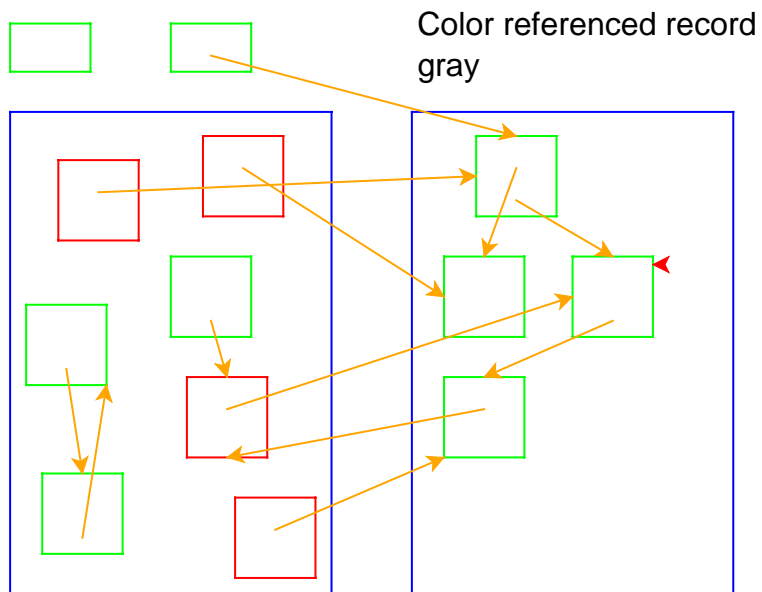
Two-Space Collection



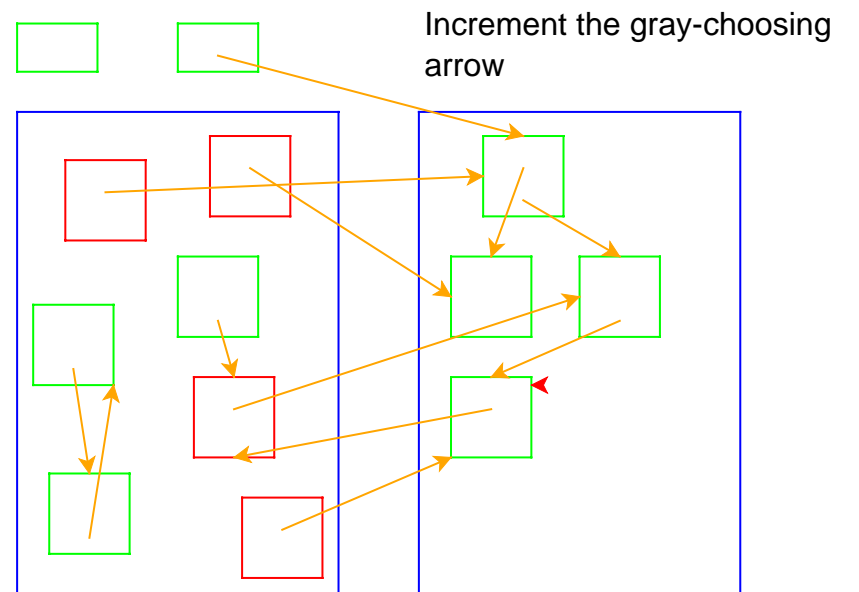
Two-Space Collection



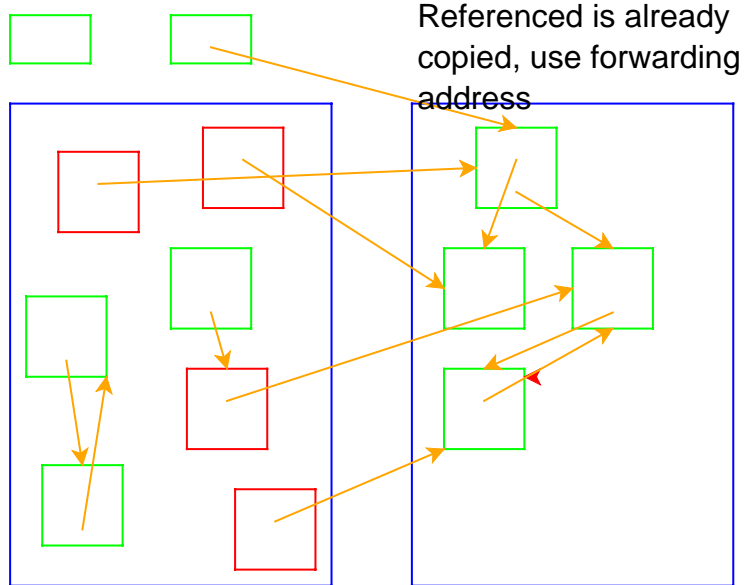
Two-Space Collection



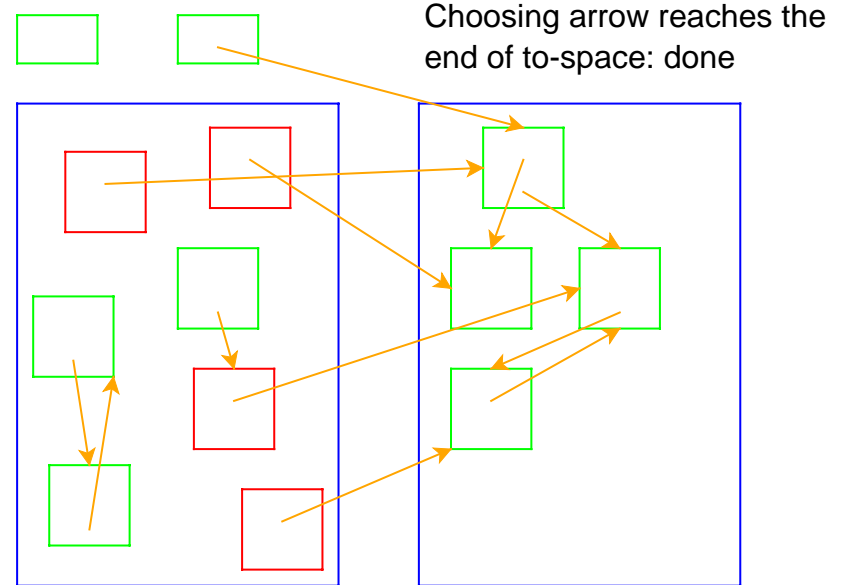
Two-Space Collection



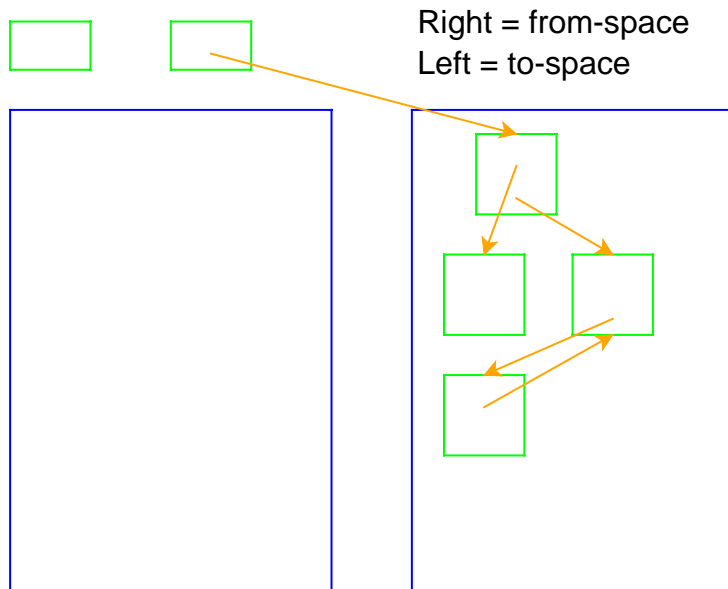
Two-Space Collection



Two-Space Collection



Two-Space Collection



Two-Space Collection on Vectors

- Everything is a number:
 - Some numbers are immediate integers
 - Some numbers are pointers
- An allocated record in memory starts with a tag, followed by a sequence of pointers and immediate integers
 - The tag describes the shape

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
 - Tag 1: one integer
 - Tag 2: one pointer
 - Tag 3: one integer, then one pointer

Register 1: 7 Register 2: 0

From: 1 75 2 0 3 2 10 3 2 2 3 1 4

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
 - Tag 1: one integer
 - Tag 2: one pointer
 - Tag 3: one integer, then one pointer

Register 1: 7 Register 2: 0

From: 1 75 2 0 3 2 10 3 2 2 3 1 4

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

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
 - Tag 1: one integer
 - Tag 2: one pointer
 - Tag 3: one integer, then one pointer

Register 1: 7 Register 2: 0

From: 1 75 2 0 3 2 10 3 2 2 3 1 4

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

 ^ ^ ^ ^ ^

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
 - Tag 1: one integer
 - Tag 2: one pointer
 - Tag 3: one integer, then one pointer

Register 1: 7 Register 2: 0

From: 1 75 2 0 3 2 10 3 2 2 3 1 4

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

 ^ ^ ^ ^ ^

To: 0 0 0 0 0 0 0 0 0 0 0 0 0

 ^

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
 - Tag 1: one integer
 - Tag 2: one pointer
 - Tag 3: one integer, then one pointer

	Register 1: 0						Register 2: 0						
From:	1	75	2	0	3	2	10	99	0	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	3	2	2	0	0	0	0	0	0	0	0	0	0
	^												

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
 - Tag 1: one integer
 - Tag 2: one pointer
 - Tag 3: one integer, then one pointer

	Register 1: 0						Register 2: 3						
From:	99	3	2	0	3	2	10	99	0	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	3	2	2	1	75	0	0	0	0	0	0	0	0
	^												

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
 - Tag 1: one integer
 - Tag 2: one pointer
 - Tag 3: one integer, then one pointer

	Register 1: 0						Register 2: 3						
From:	99	3	99	5	3	2	10	99	0	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	3	2	5	1	75	2	0	0	0	0	0	0	0
				^									

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
 - Tag 1: one integer
 - Tag 2: one pointer
 - Tag 3: one integer, then one pointer

	Register 1: 0						Register 2: 3						
From:	99	3	99	5	3	2	10	99	0	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	3	2	5	1	75	2	0	0	0	0	0	0	0
						^							

Two-Space Vector Example

- 26-byte memory (13 bytes for each space), 2 registers
 - Tag 1: one integer
 - Tag 2: one pointer
 - Tag 3: one integer, then one pointer

	Register 1: 0						Register 2: 3						
From:	99	3	99	5	3	2	10	99	0	2	3	1	4
Addr:	00	01	02	03	04	05	06	07	08	09	10	11	12
	^		^		^			^			^		
To:	3	2	5	1	75	2	3	0	0	0	0	0	0
								^					