# Scalable Garbage Collection with Guaranteed MMU

Felix S Klock II (pnkfelix@ccs.neu.edu)
William D Clinger (will@ccs.neu.edu)

# Problem: Garbage Collection

- Naïve GC introduces pauses
- Generational GC introduces (infrequent) pauses
  - Still disruptive and annoying for users
- Real-time / incremental / concurrent can eliminate delays
  - but at significant cost

# Goal:
# Scalability in space and time



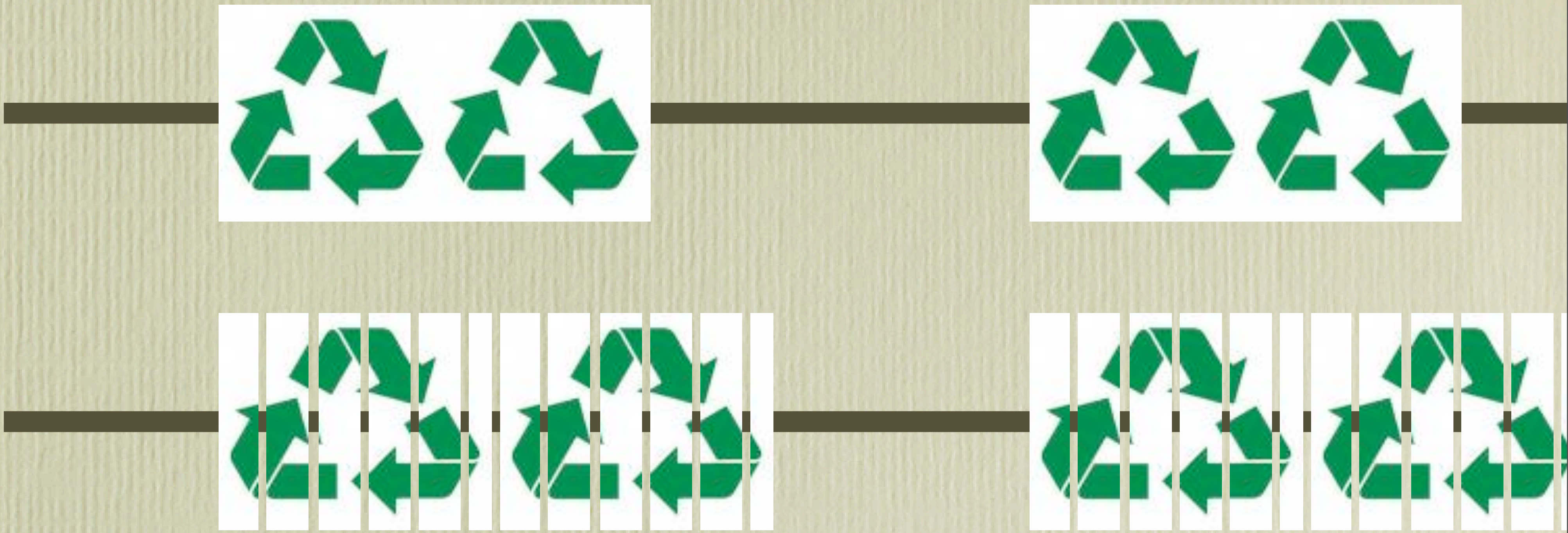
SPACE: GC META DATA + BOUNDING FLOAT.  TIME: PAUSE TIMES, MMU.  (Not worried so much about very fine-grained control flow between collector and mutator, but rather the experience for the end-user in interactive applications.)

# Control Space:
# Metadata & Floating Garbage

META DATA; FLOAT

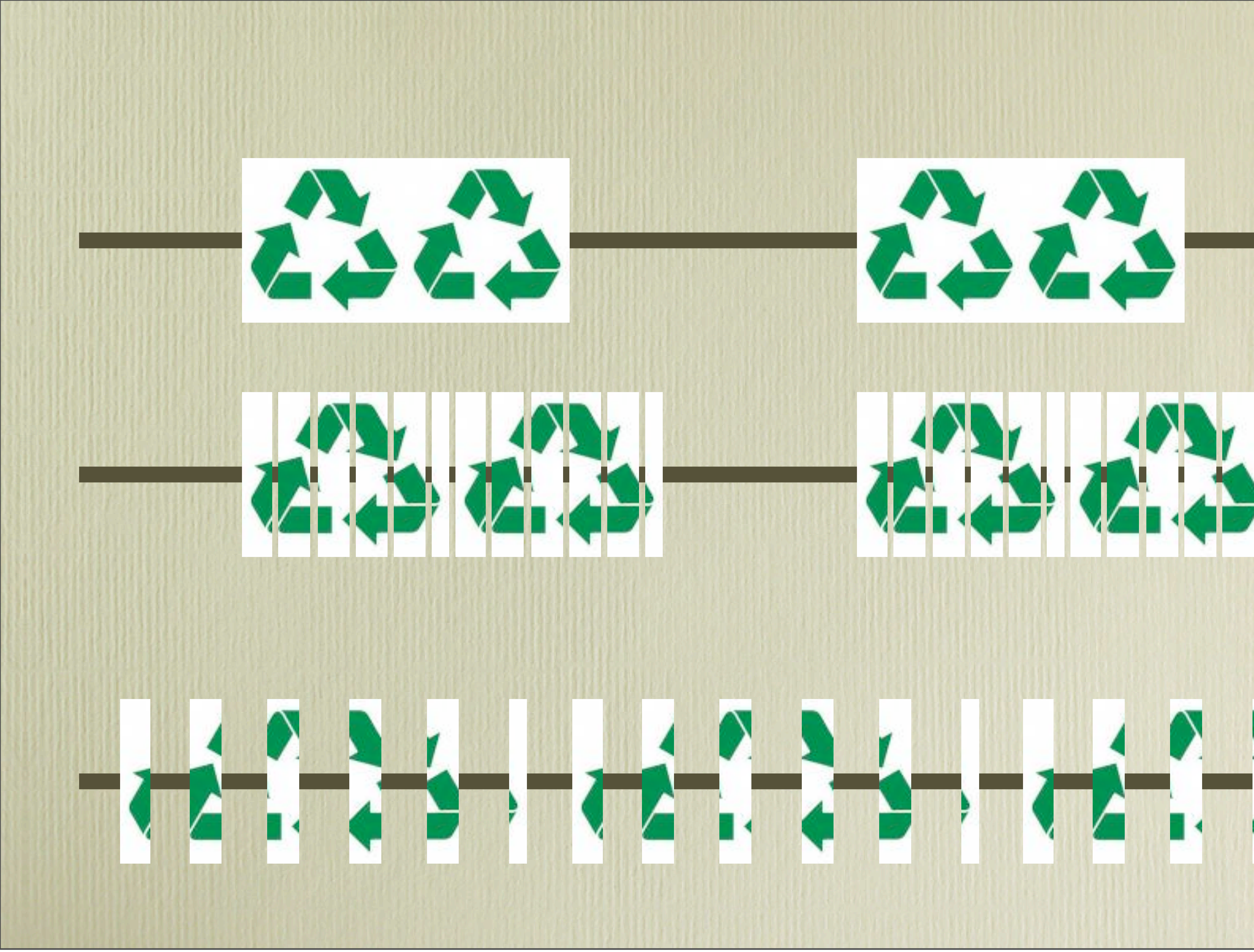
# Control Time:
# Max Pause as metric?

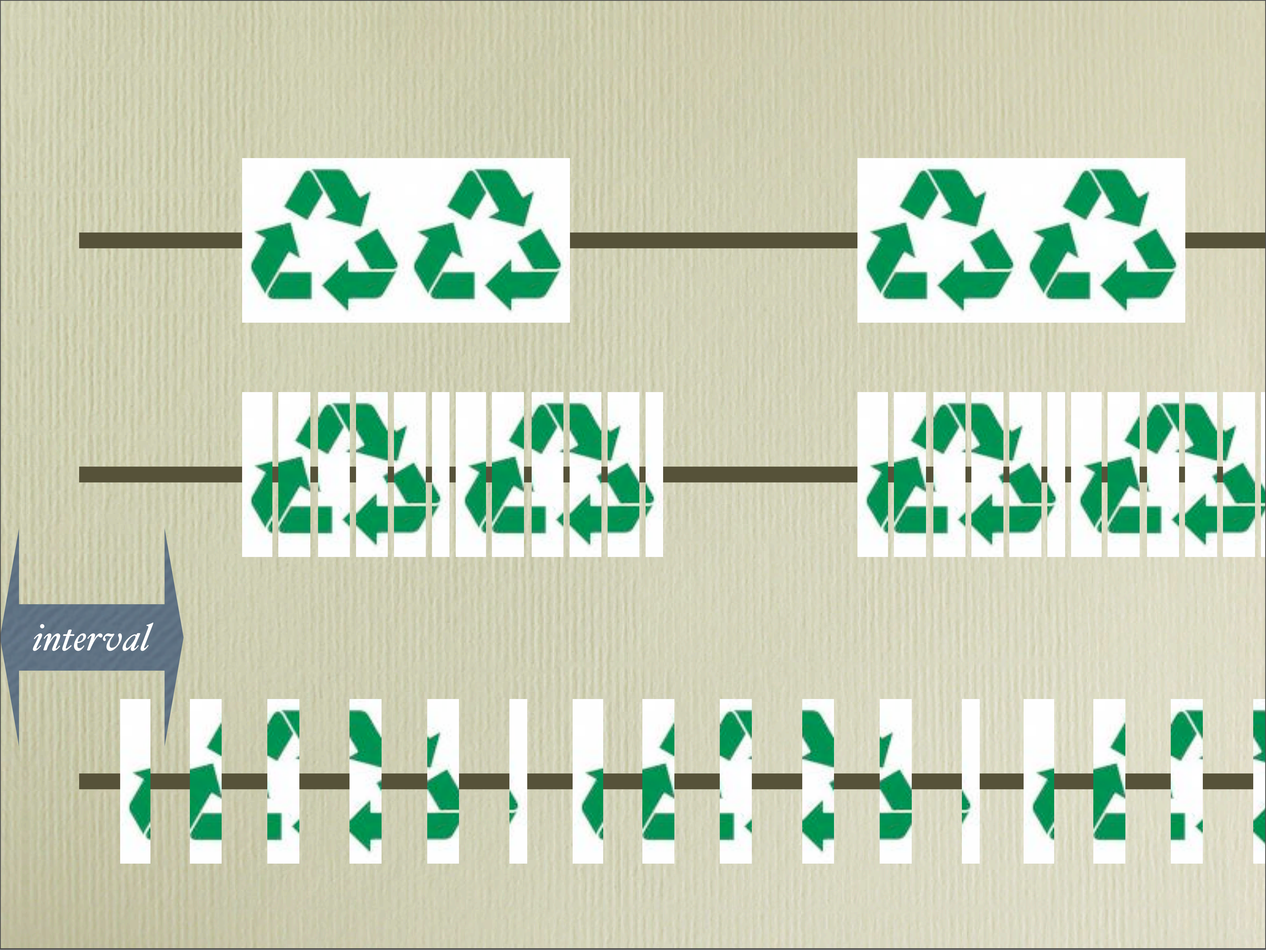
(an artificial illustration)

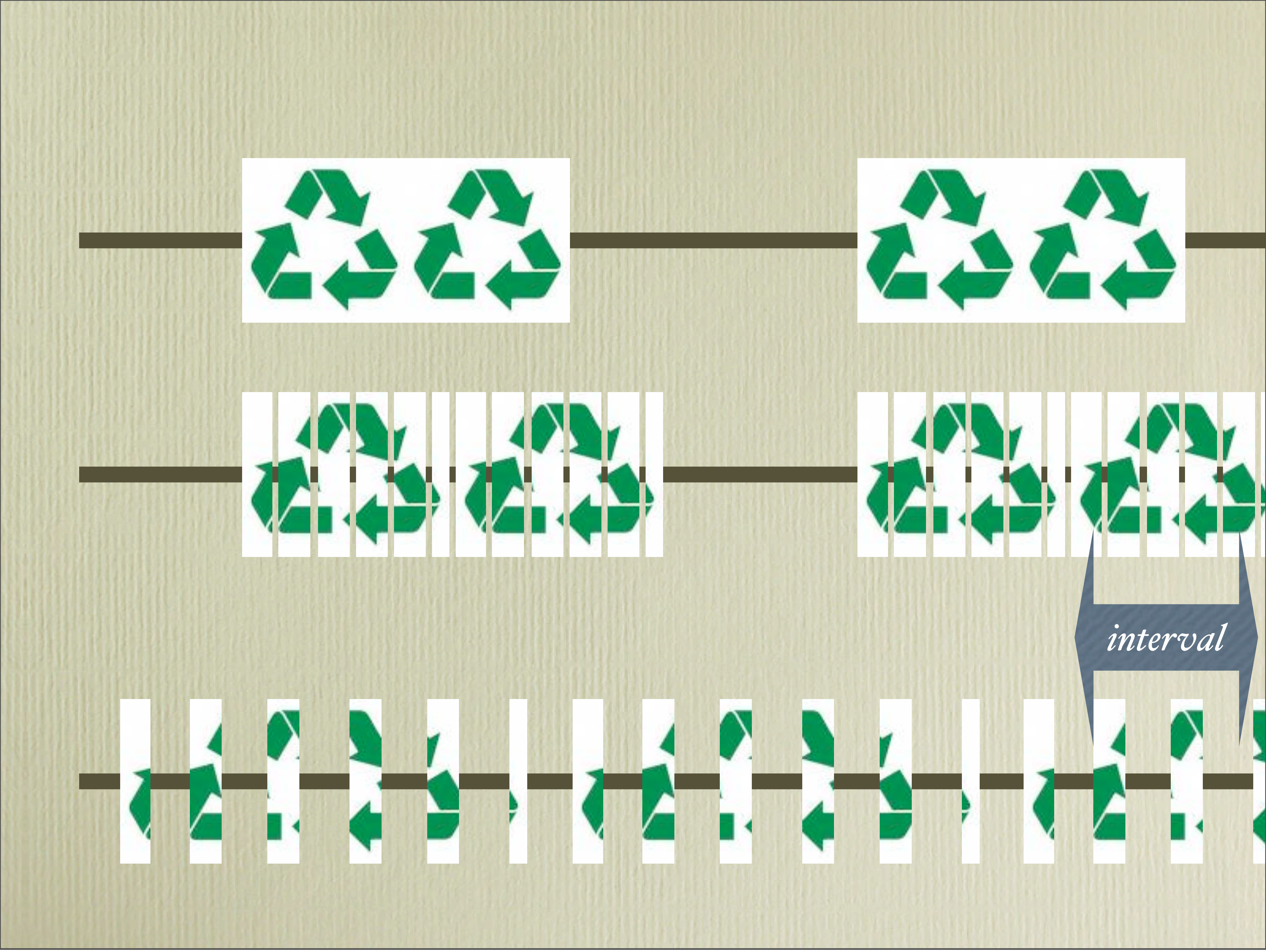


# Minimum Mutator Utilization (MMU)

[Cheng and Blelloch '01]




interval


interval

# Our Scalability Theorem

For *all* mutators, no matter what the mutator does:

1. Max GC pause length is independent of heap size

2. MMU bounded from below, independent of heap size

3. Memory usage is O(*P*), where *P* = peak volume of reachable objects



- Most incremental collectors do not provide theorems like this
- Blelloch and Cheng ’99 is an important exception, but they only provide (1) and (3), not (2).
- We believe they could have proven (2.) after they introduced MMU in 2001, but constant overheads are unclear

# “Simple” Idea

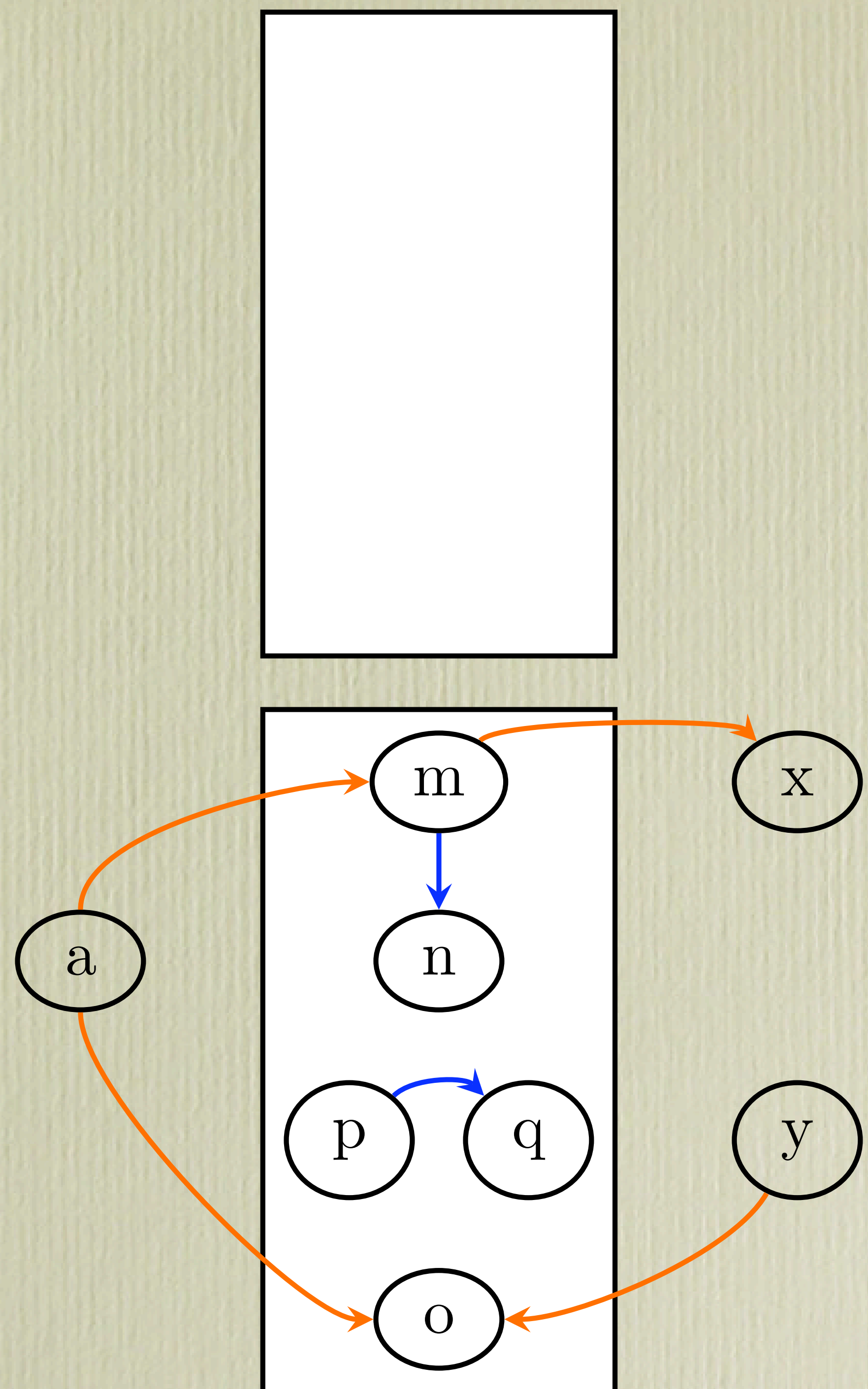
- Partition heap into fixed sized “regions”
- Collect each region independently
- Since regions are bounded in size, can do this in bounded time, right (?)

# “Simple” Idea

- Partition heap into fixed sized “regions”
- Collect each region independently
- Since regions are bounded in size, can do this in bounded time, right (?)

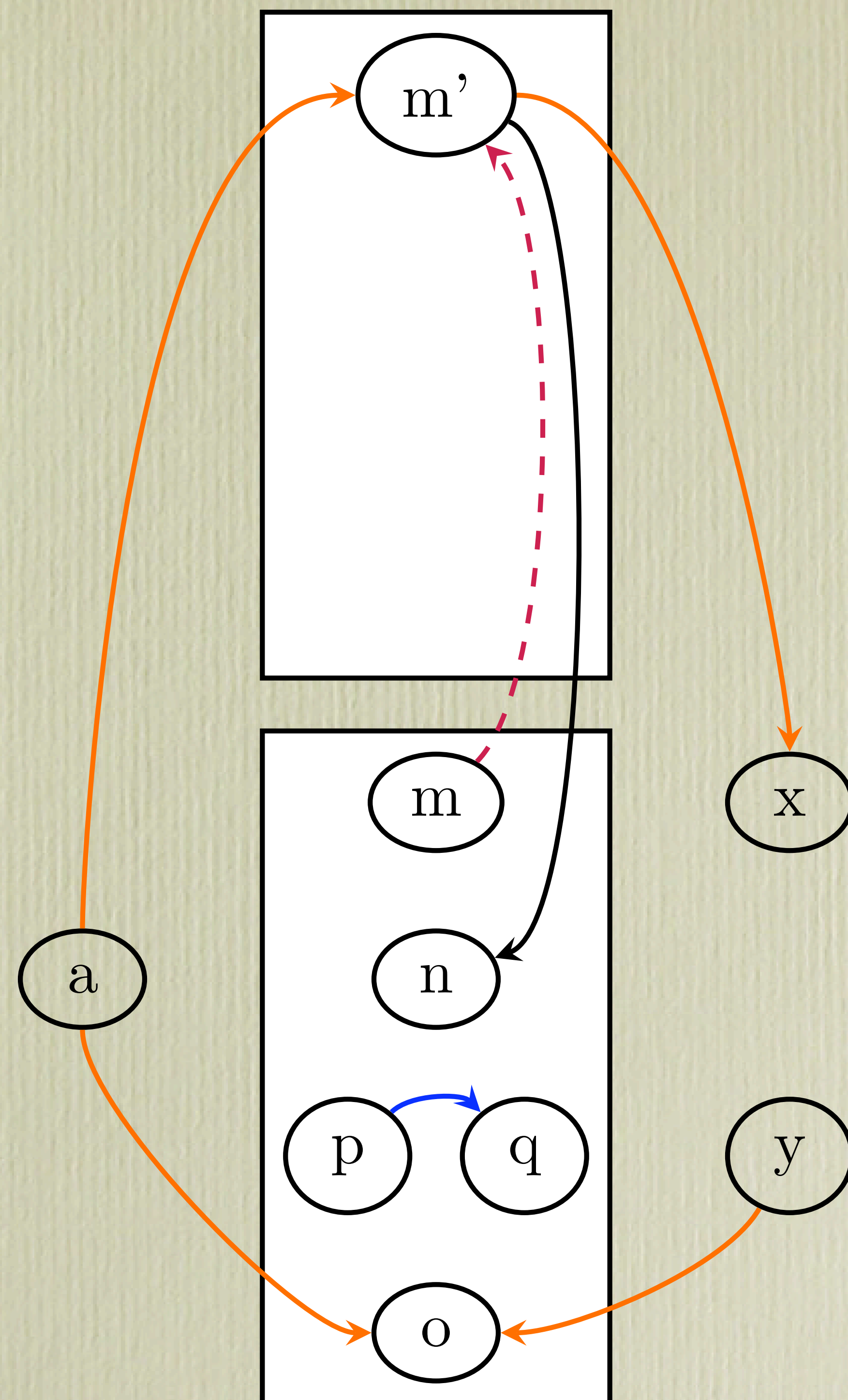
*(yes, but just barely)*

# Regional GC Illustrated (review of Cheney)



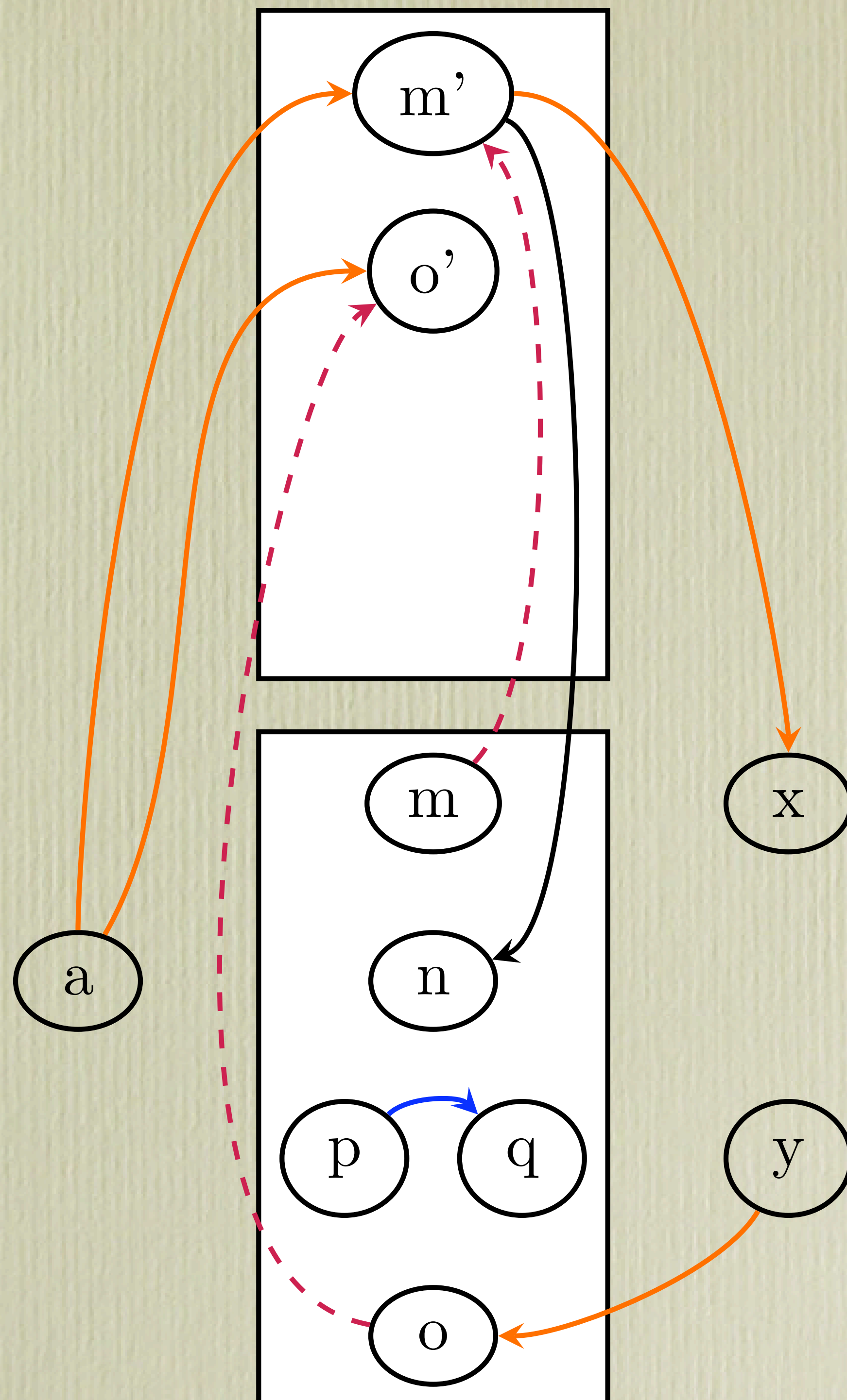



here's a region. The ovals are objects, the arrows are references.
If you care, you might notice that the orange arrows cross regions (as opposed to the yellow ones) We're just going to collect the middle stuff; the {A, X, Y} belong to regions elsewhere.
Start by scanning the object A



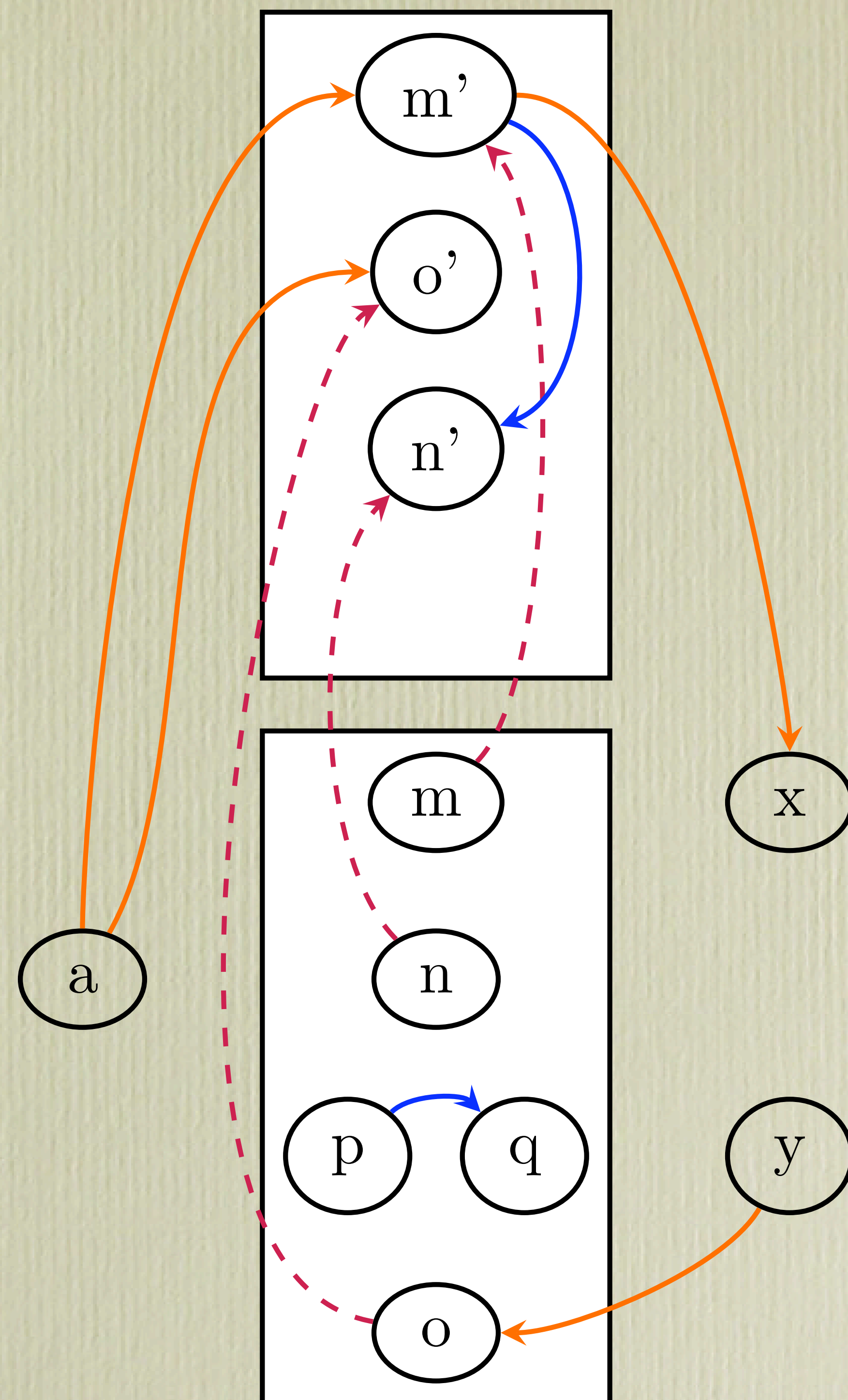



Scanning A implies M is reachable from A; copy M to M' (which includes that reference to N), install a forwarding pointer from M to M', and continue scanning A.



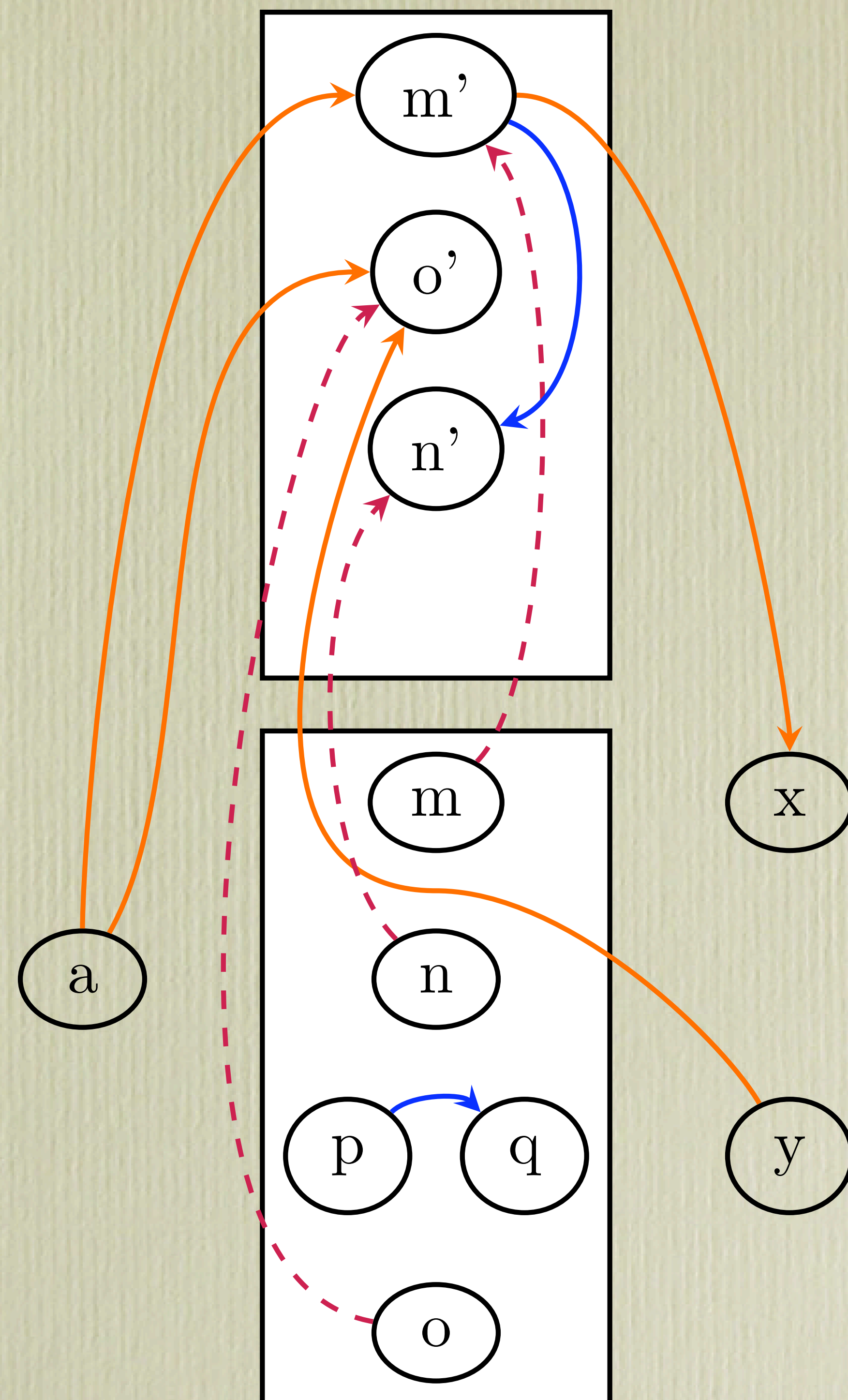



Scanning A also finds a reference to O, so forward that to O'. We're done scanning A, so lets just continue from left-to-right and scan M' next.



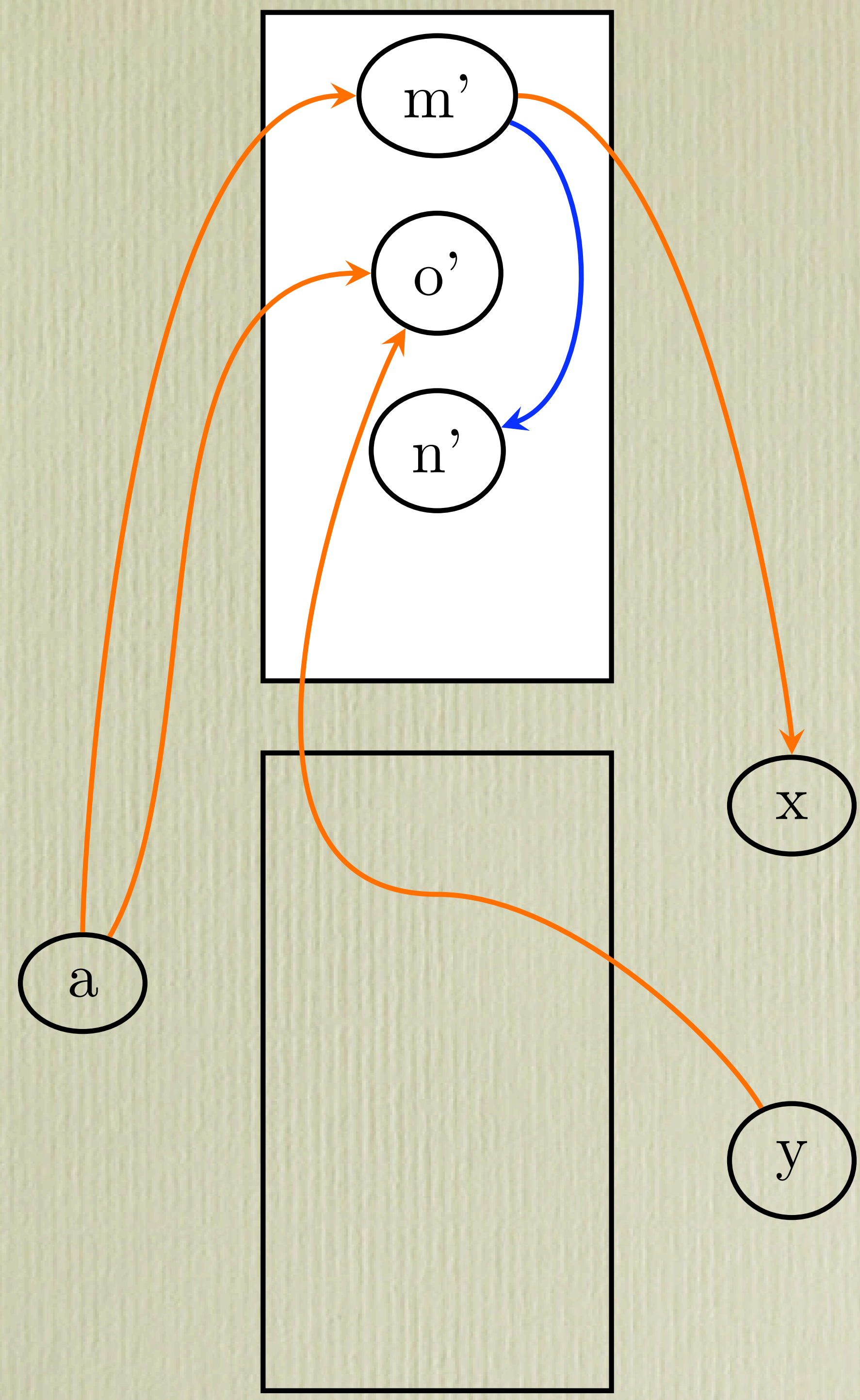



Scanning M' uncovers that reference to N; forward that to N'. Continuing left-to-right means scanning O' and N', which uncovers no more references within the region. So move on to the right side, where we have X and Y...






Scanning Y reveals that reference to O that needs to be updated to O'. Now we've finished scanning everything we need to. Note there are no more references into the region we were collecting; we can reclaim it!






And we are done.

# How to do this efficiently?



I hand-waved about scanning the stuff on the left and right of the region, but in fact that's the name of the game here!

# Goal: Avoid inspecting extraneous state



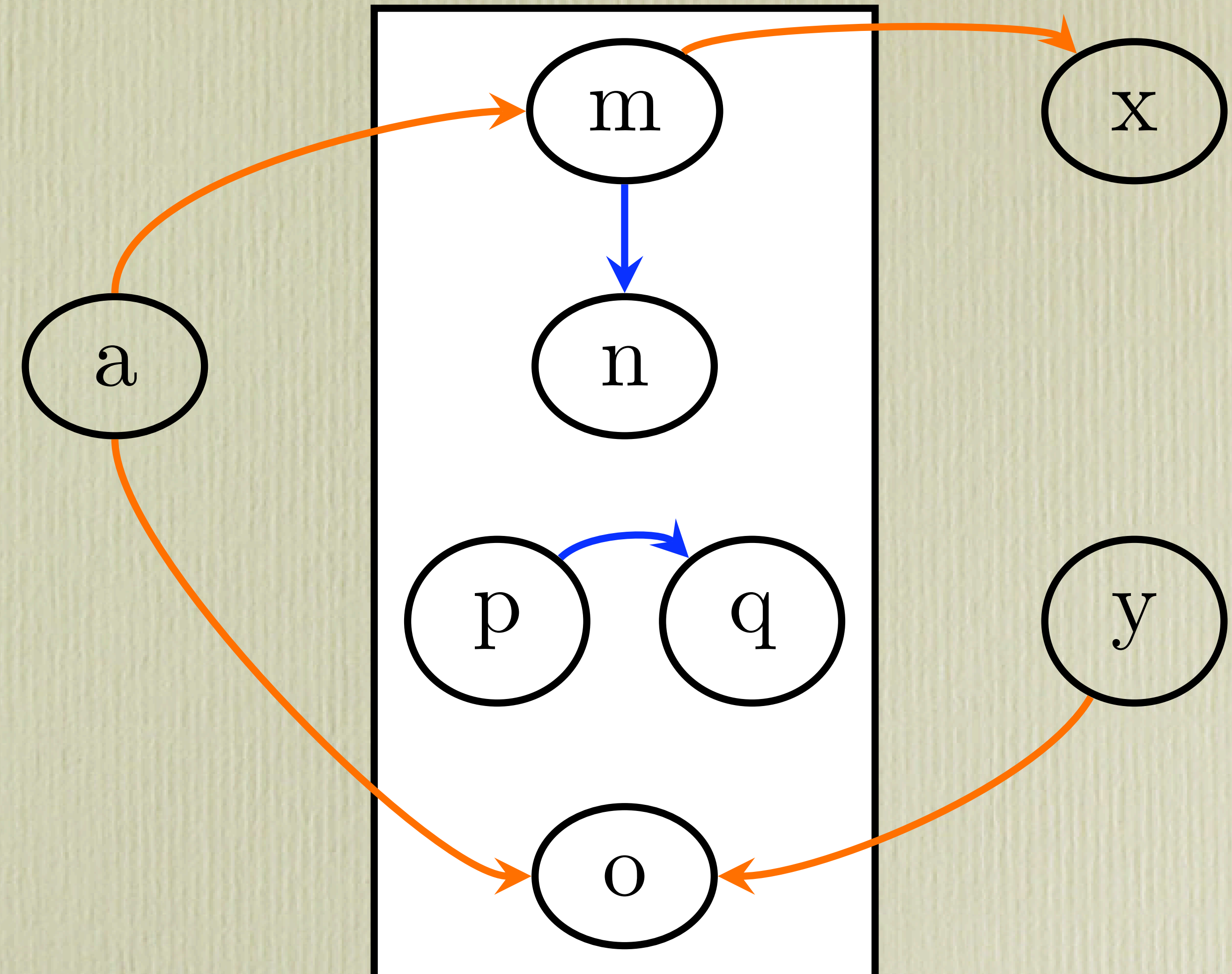
We don't want to scan the whole heap outside of the region, because that would break our pause time bounds.

# Remembered Set?

( [Lieberman and Hewitt '83] ),
[Ungar '84]



We might first take inspiration from generational collectors. The original Lieberman and Hewitt paper on GenGC was actually trying to solve the very problem we're talking about. So how about REMEMBERING the objects that have region-crossing references?



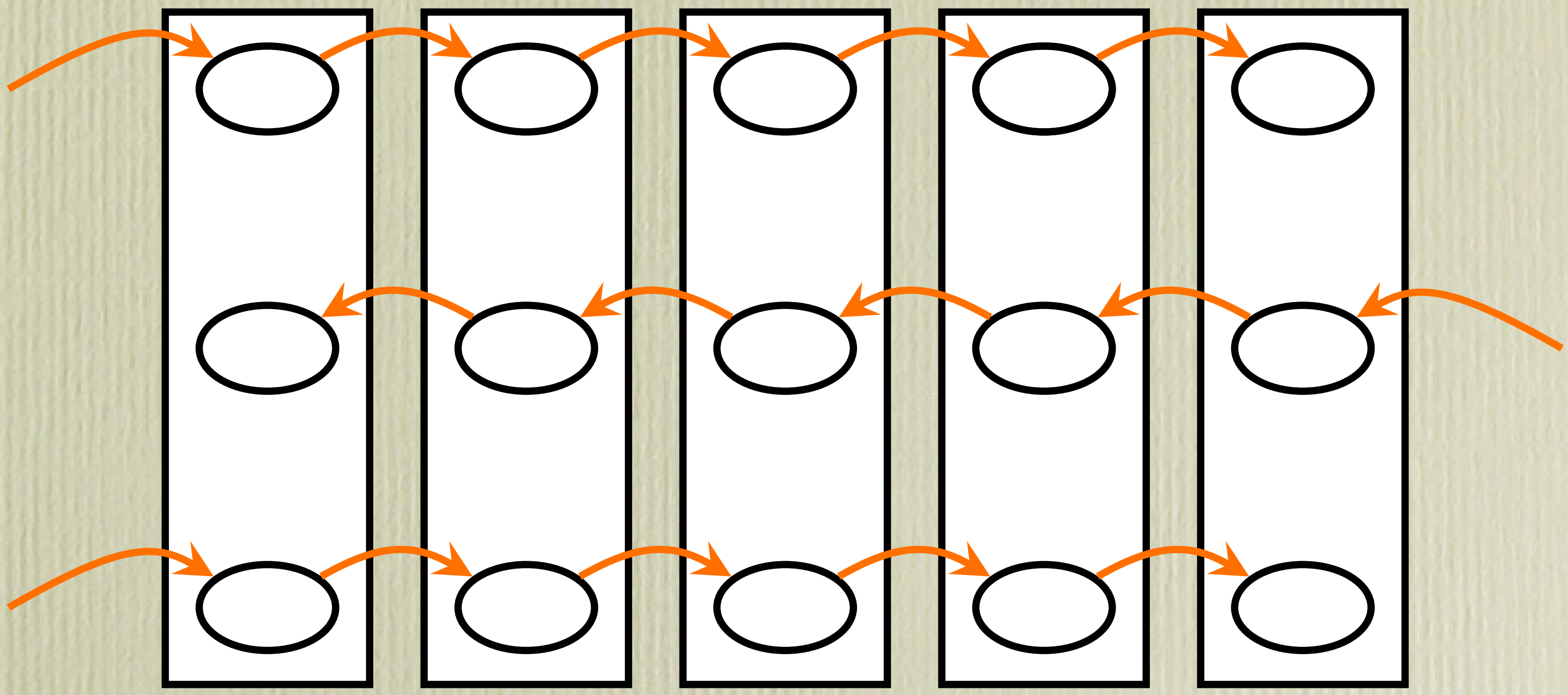

# Remembered Set $\supseteq \{ a, m, y \}$



So we could try to maintain a remembered set that captures all of the objects with region-xing ref's (and maybe others too; its allowed to be imprecise).

# Problem with Remembered Set

# Remembered set can grow proportional to size of heap



scan time generally not proportional to region size

# Remembered Set holds junk irrelevant to current GC

(still useful; still used)

Try maintaining state per-region ?

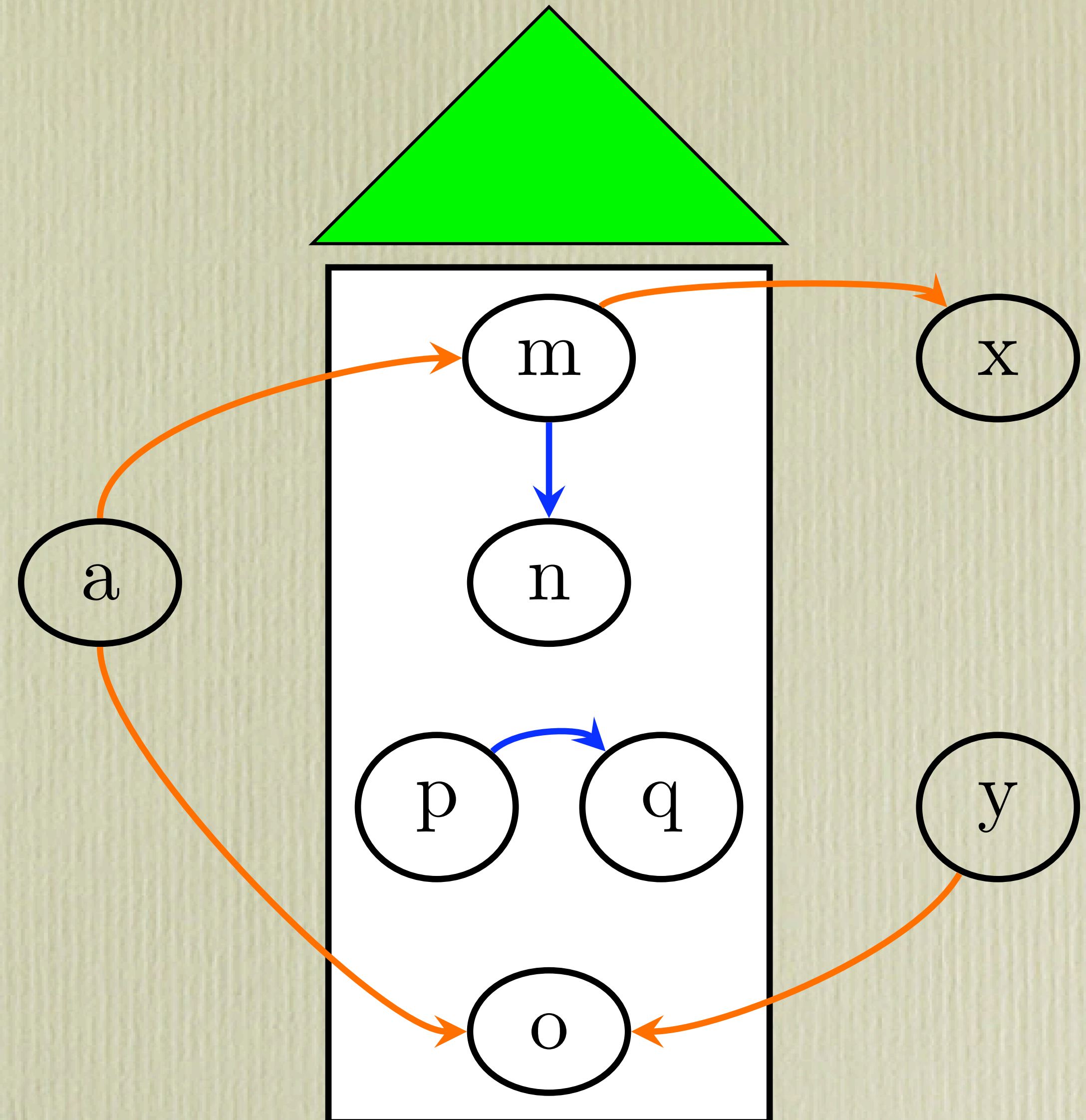
(Garbage-First [Detlefs '04])

(breathe)

PATTER: “Don’t have a slogan.  Realized workshop deadline was approaching, so I’ll resort to just summarizing the approach here, and later we will refine our view of the approach after the whole picture has been developed.”

*Summarize* as deadline approaches; *Refine* after whole picture has been developed

# Summary Sets




This summary set $\supseteq$

{&a[1],&a[3],&y[0]}

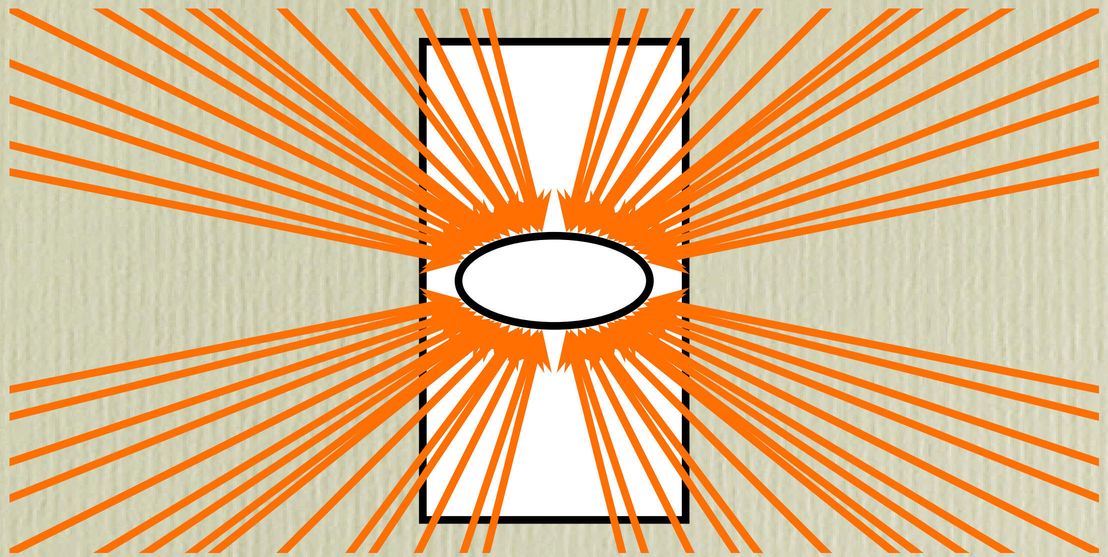


Apologies for the C notation. Its there to STRESS that we're talking tracking locations, not objects, in the summary sets. The paper discusses this more.
Note also that M is not in this summary set; its slot will be present in the summary set for some OTHER region

# Summary Set Structure

- Summary set structure focuses attention away from irrelevant objects
- Does it work?
  - Popular objects / regions
  - Space cost

# #1: The Popularity Problem



- Many locations may point to one object
  - (or group of objects co-located in same region)
- Implies LARGE summary for that region



This is a real-world issue; consider the symbol table for a compiler, or instances of the singleton pattern, interned objects, ... etc

# #2: The Space Problem

- Maintaining *precise* summary sets for every region at all times is unrealistic
  - (too much overhead in time)
- Maintain *imprecise* summary sets ?
  - Entails unacceptable worst case space overhead
  - i.e. O(N^2) where N is heap size
  - Garbage-First [Detlefs '04] had this problem

# Insight from Lake Woebegon

{Keillor '85}

# Solving both problems: Insight #1

- Not *all* regions can be more popular than *average*
  - Generalizes nicely via a pigeon-hole argument
- Therefore, we may be able to *skip* collection of popular regions entirely!

# Applying Popularity Insight

- Do not maintain summary sets at all times
- Instead construct afresh on "just-in-time" basis
- Wave off collection of popular regions
  - i.e. those with summary set ≥ $S \cdot R$
  - (don't bother finishing their summaries)

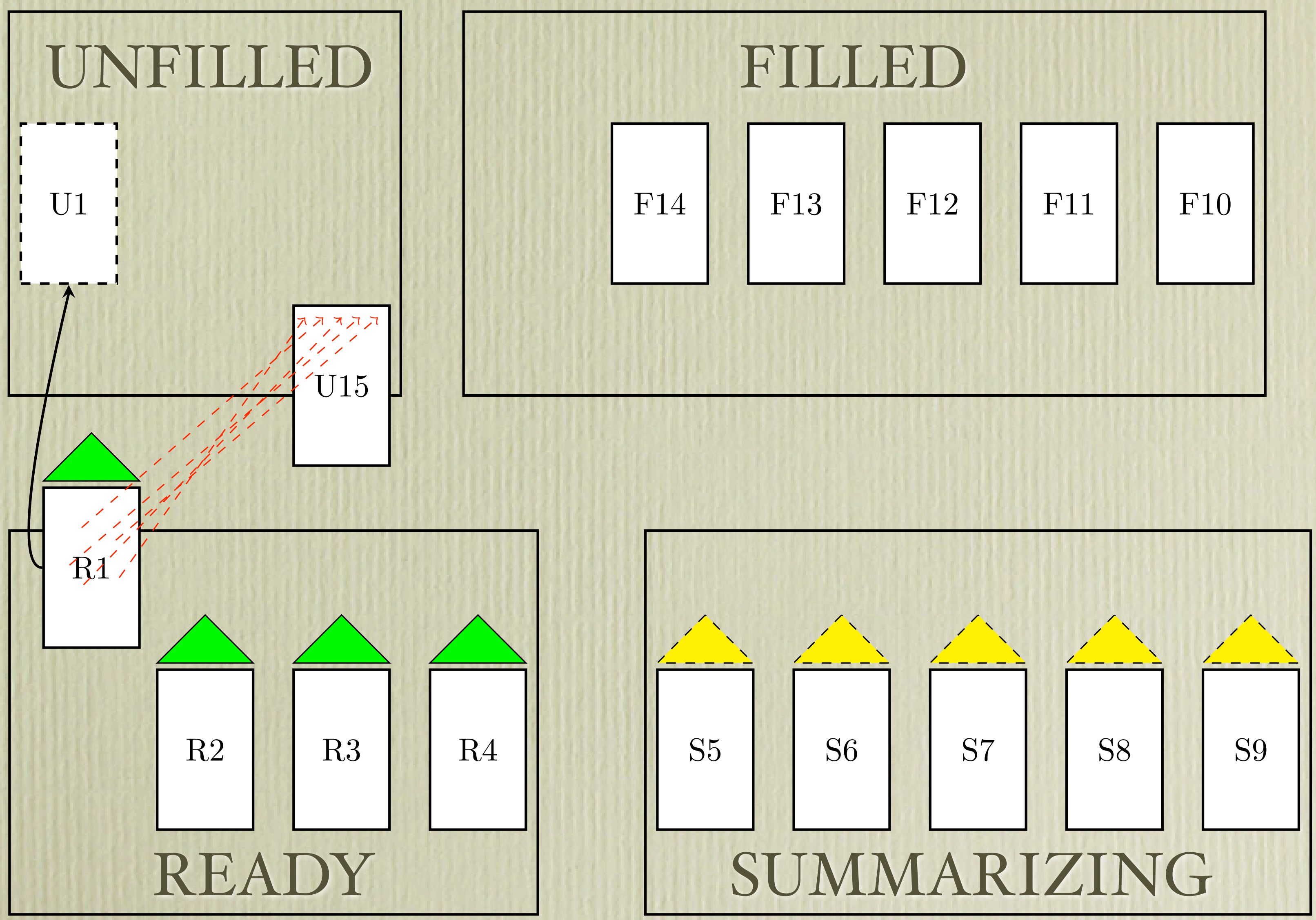
# New problem

- Constructing one summary generally requires scanning whole heap
- Not necessarily time between collection of region *r* to construct summary for region *r'*

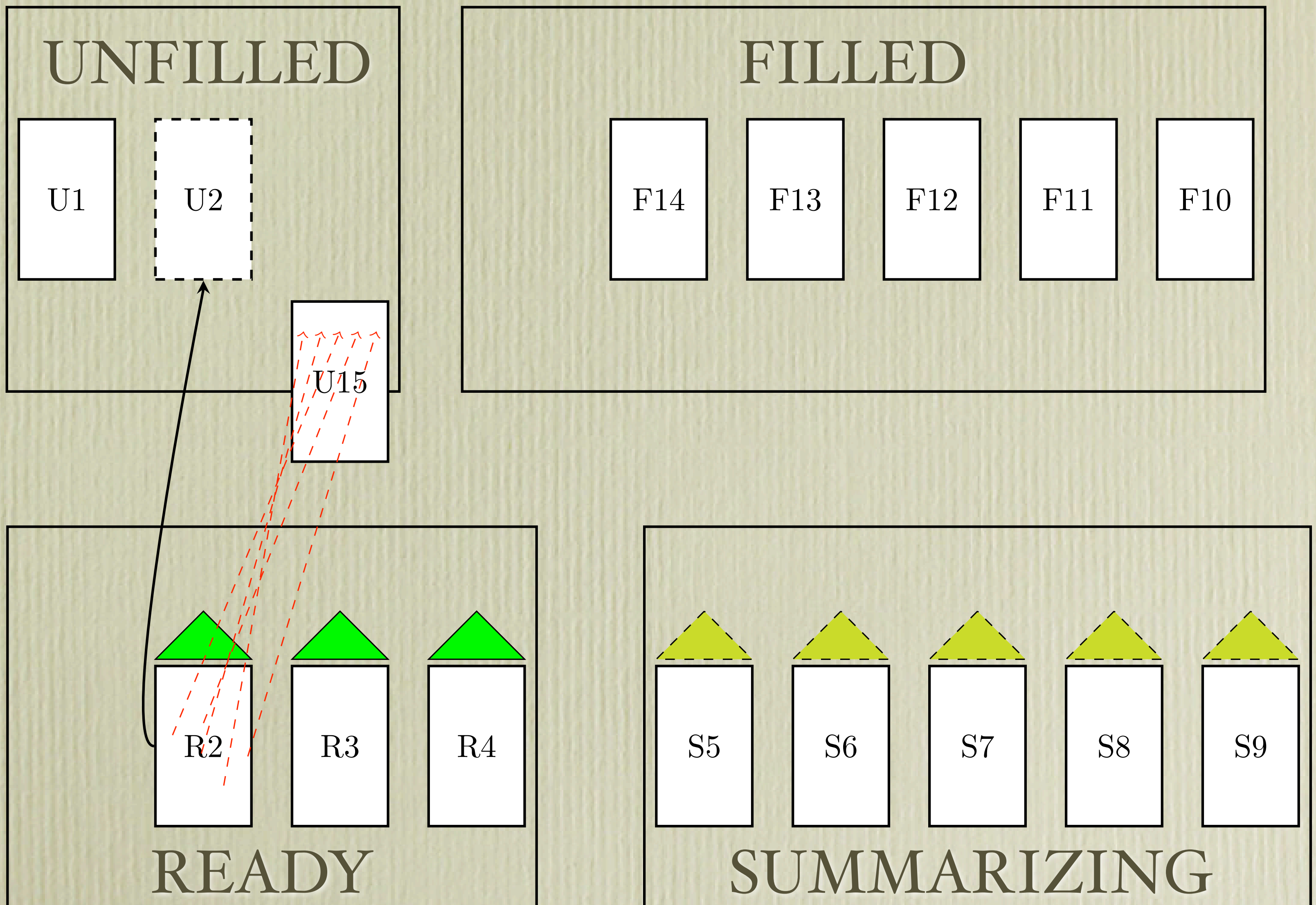
# Insight #2

- Amortize the effort!
- Construct the summary sets for many regions at once during one incremental scan

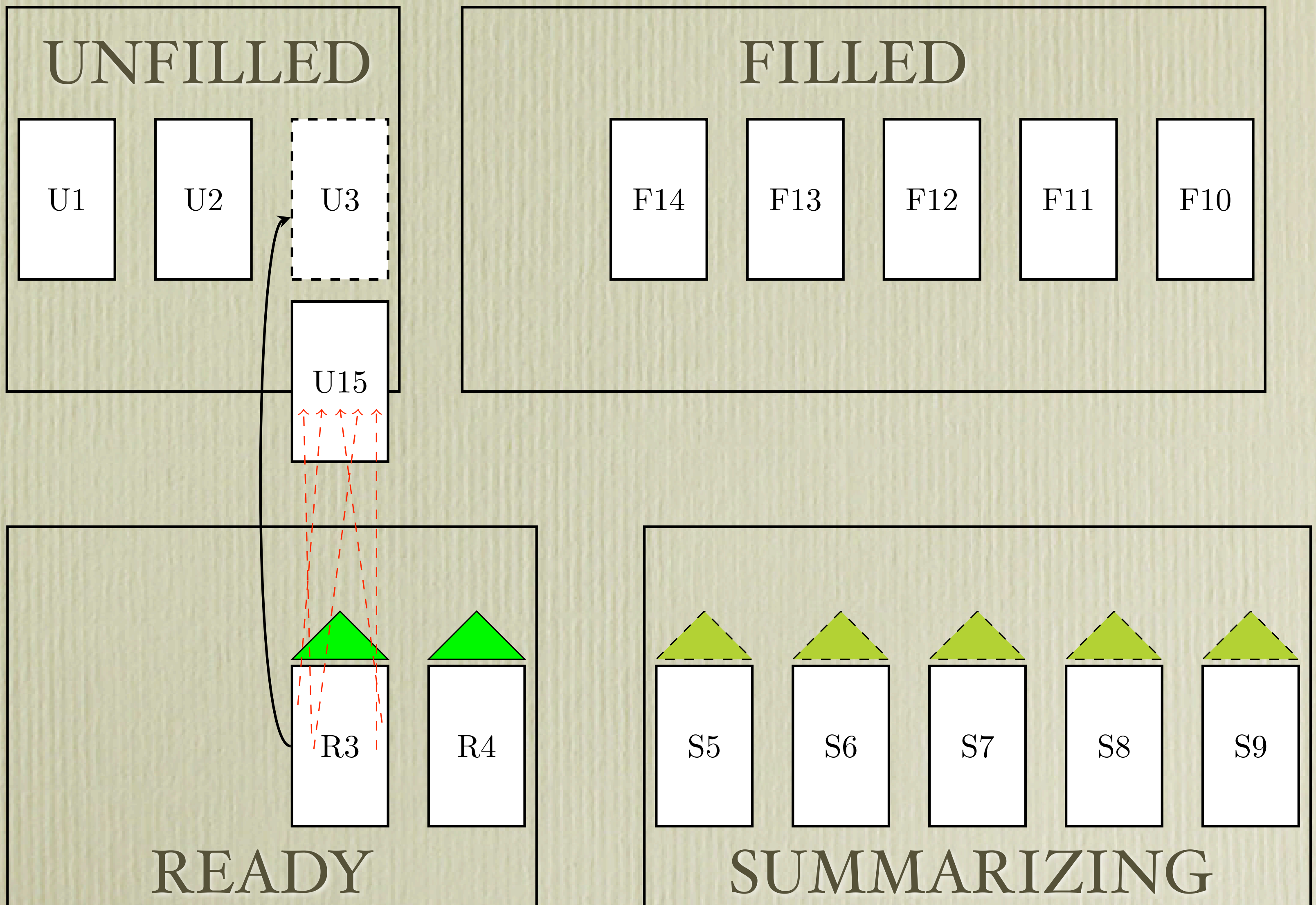
# The Basic Idea



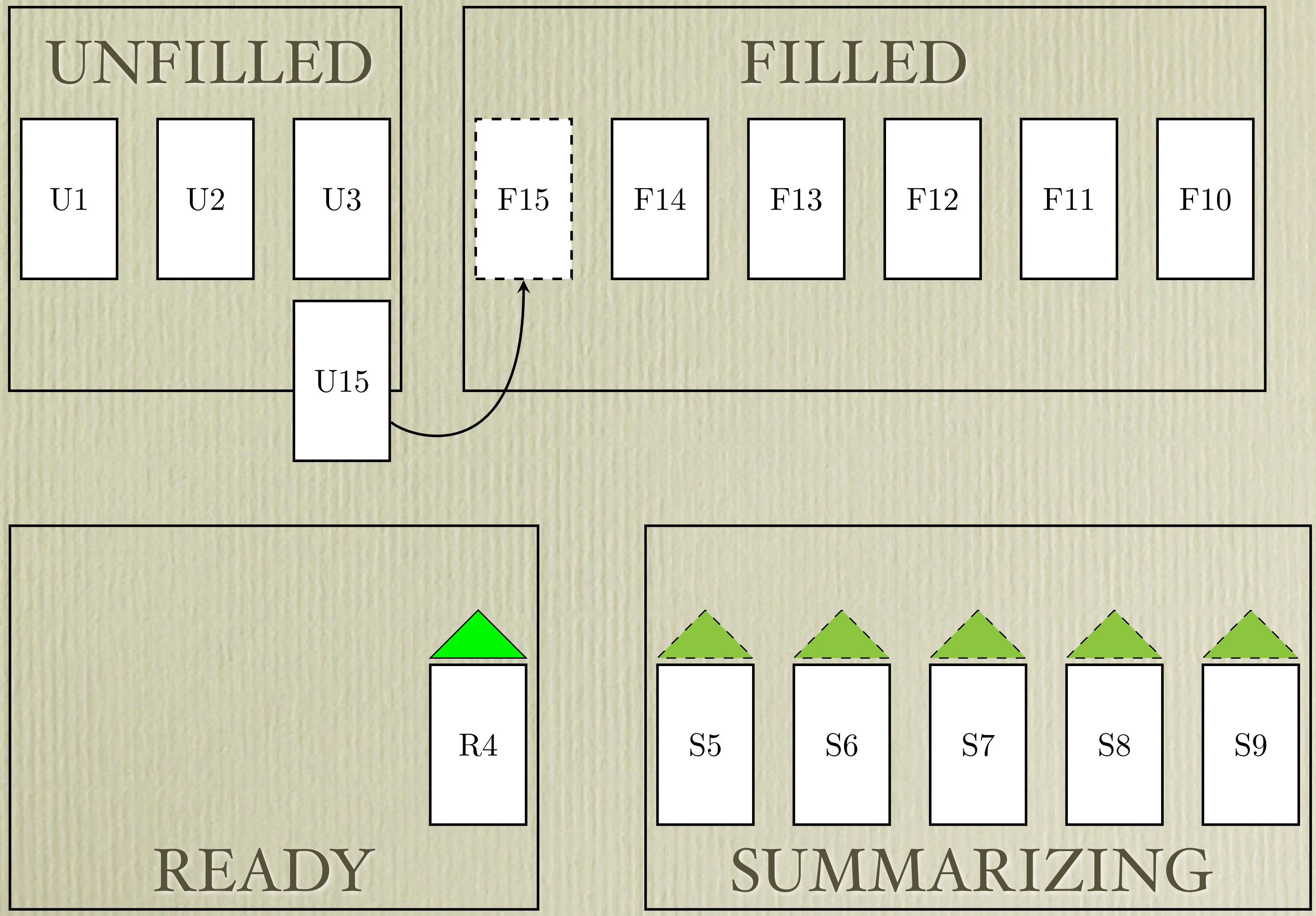
# The Basic Idea



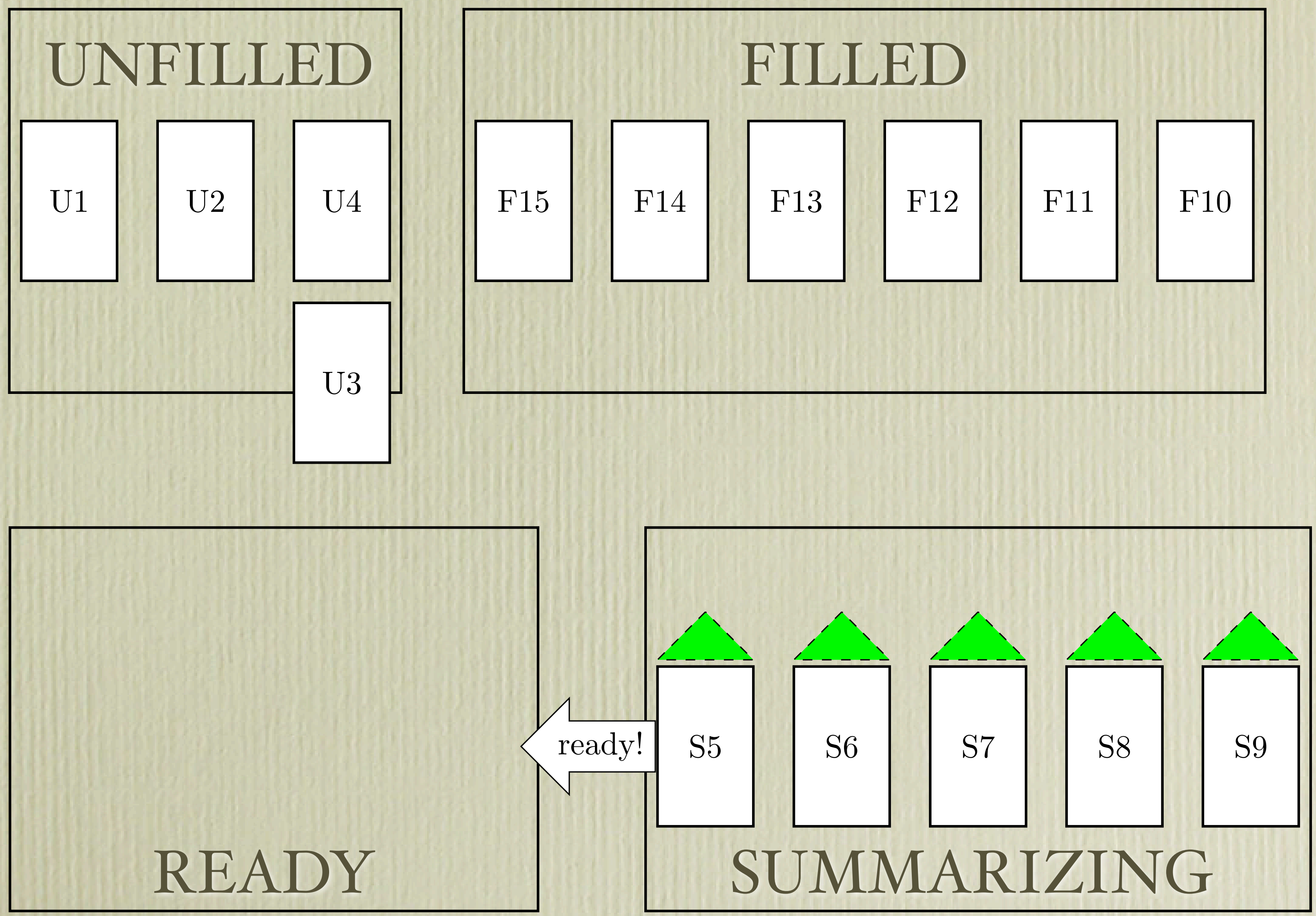
# The Basic Idea



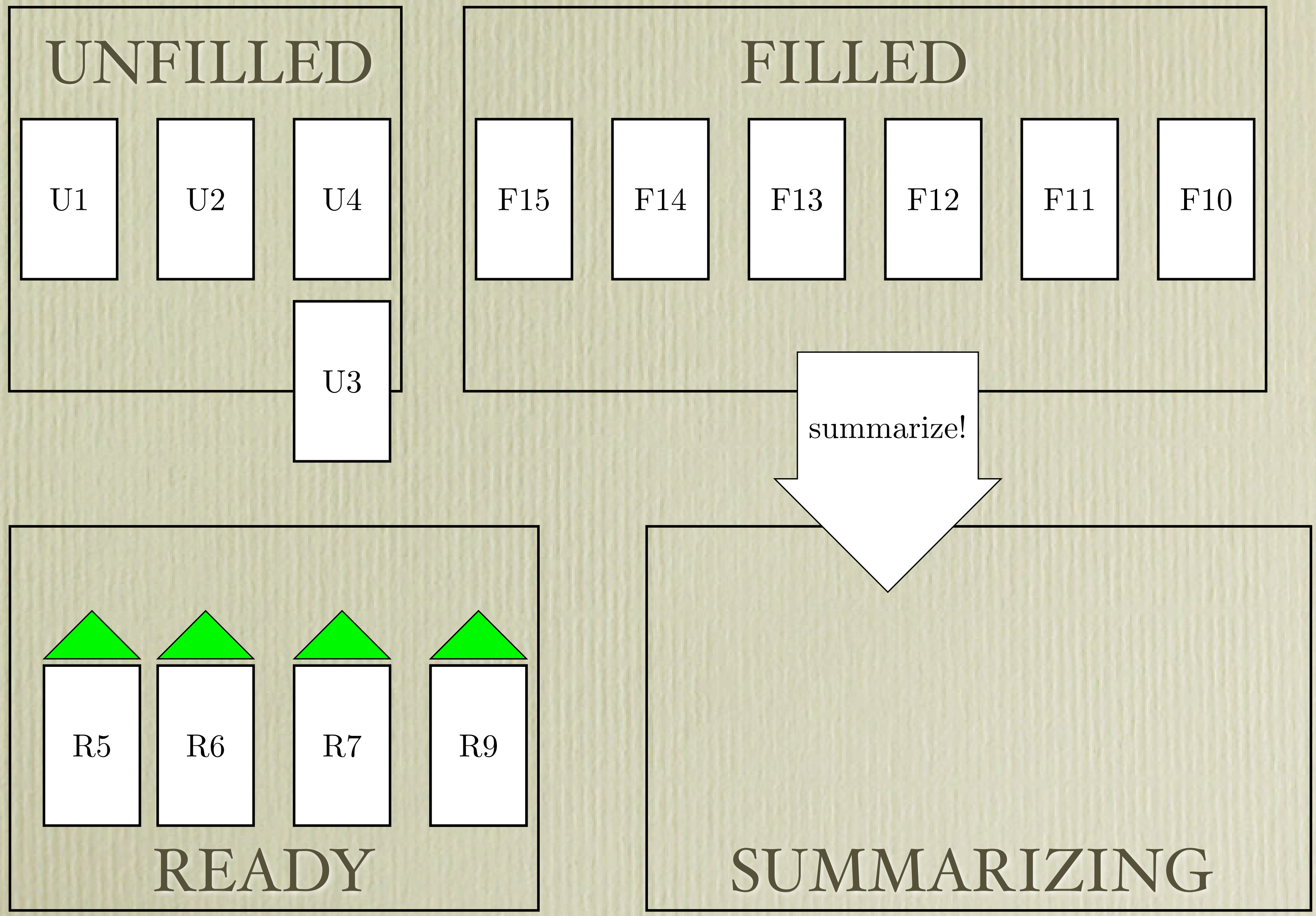
# The Basic Idea



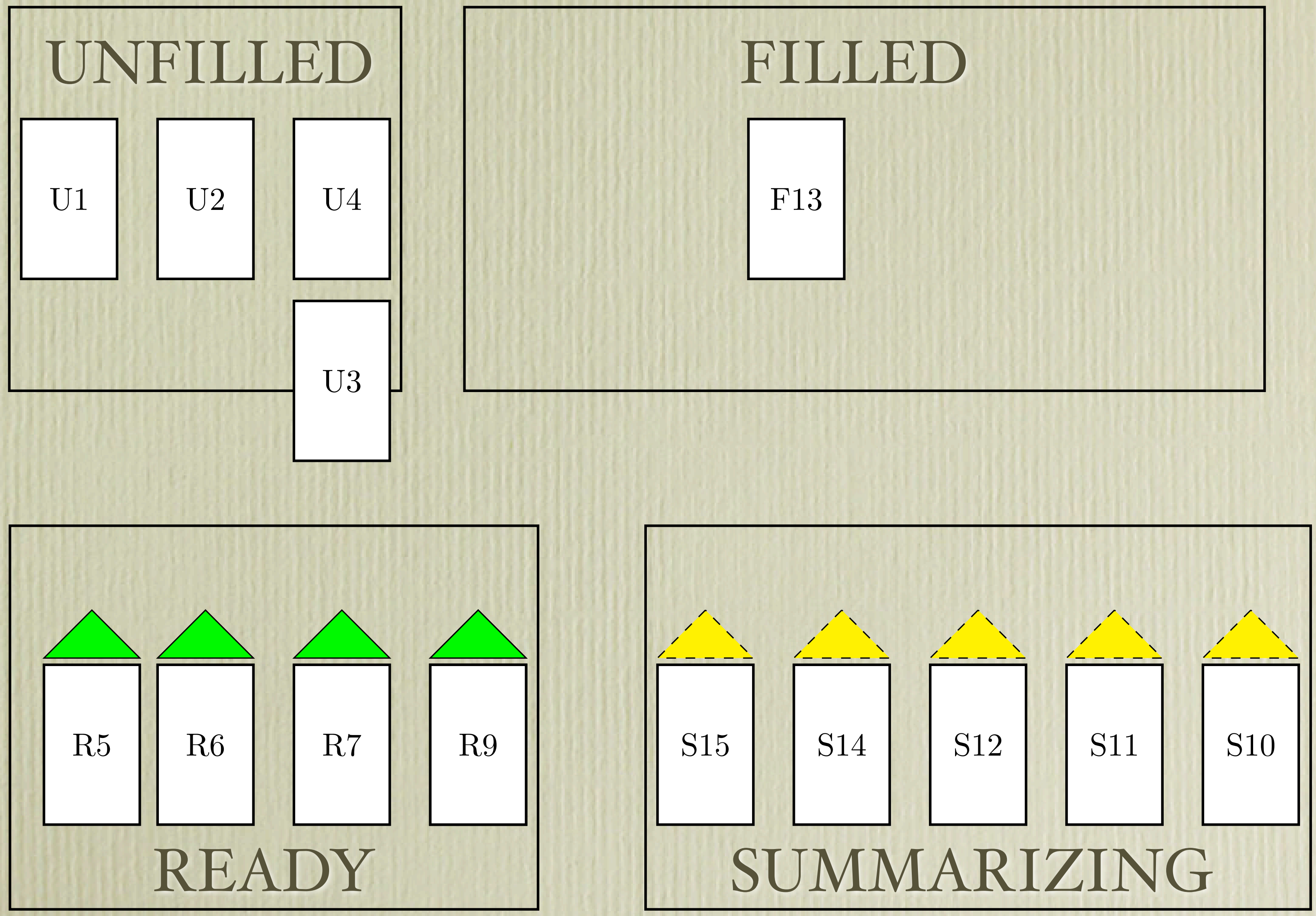
# The Basic Idea



# The Basic Idea

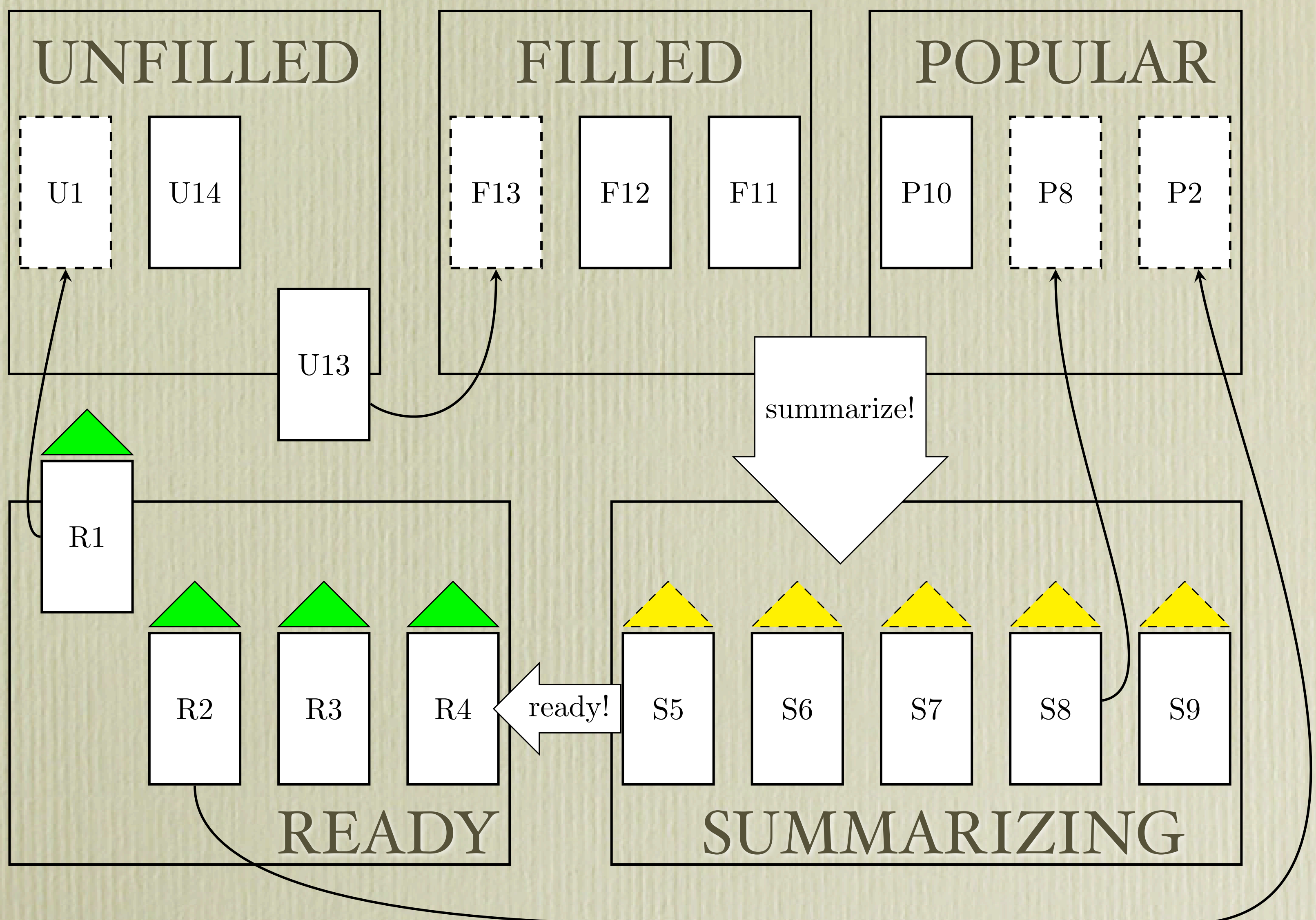


# The Basic Idea



# What about skipping the popular regions?

# The Real Picture






NOTE: You have to choose the right policy parameters here for the theorem to hold true!

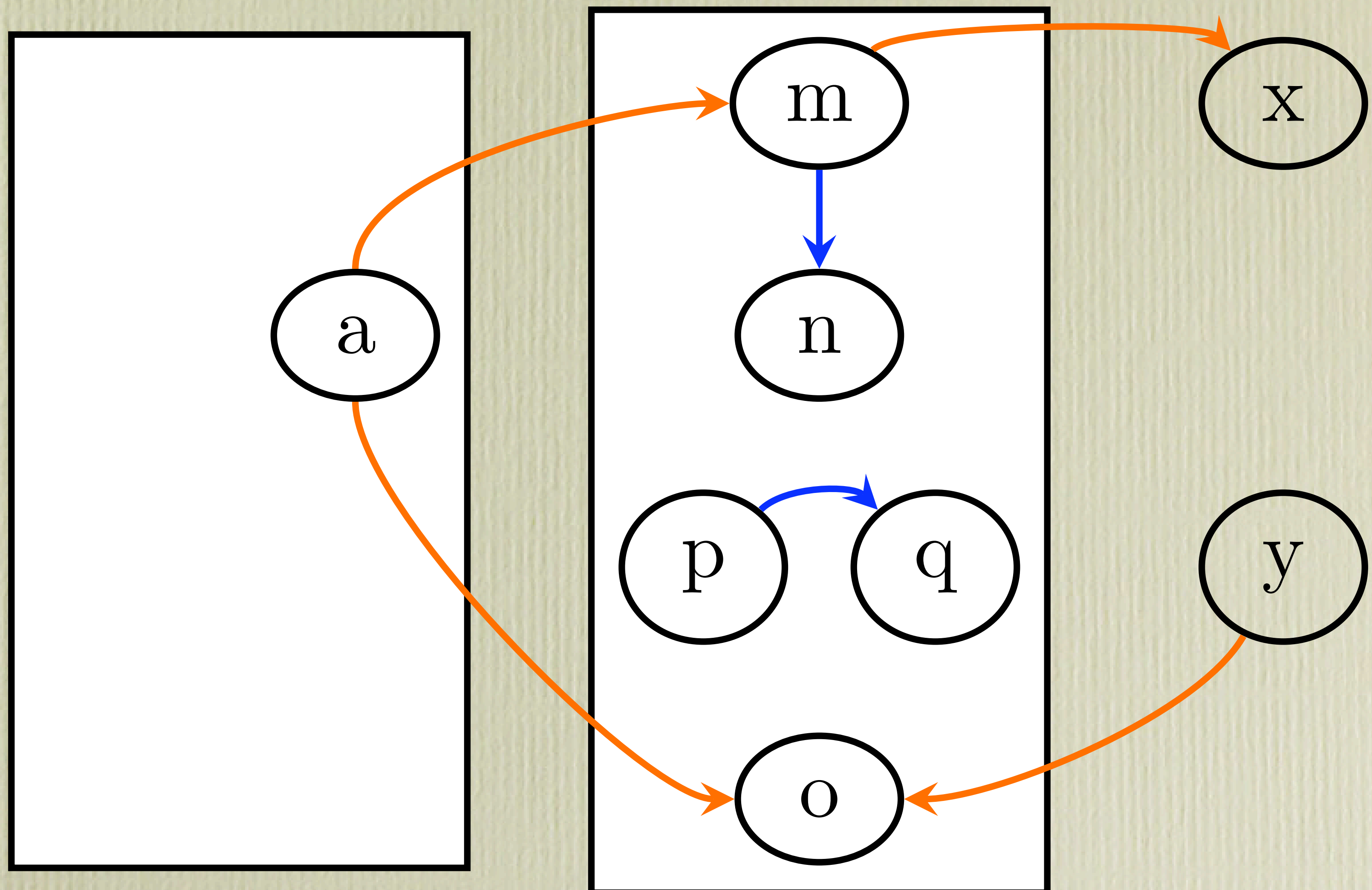
# Last Problem & Insight #3

- How to collect region crossing cycles?
  - How to ensure amount of floating garbage remains in O(Reachable State) ?
- Use Snapshot-at-the-Beginning (SATB) [Yuasa'90] to refine remembered set and summary sets
  - (also ensures popular regions won't hold onto other regions' state forever!)

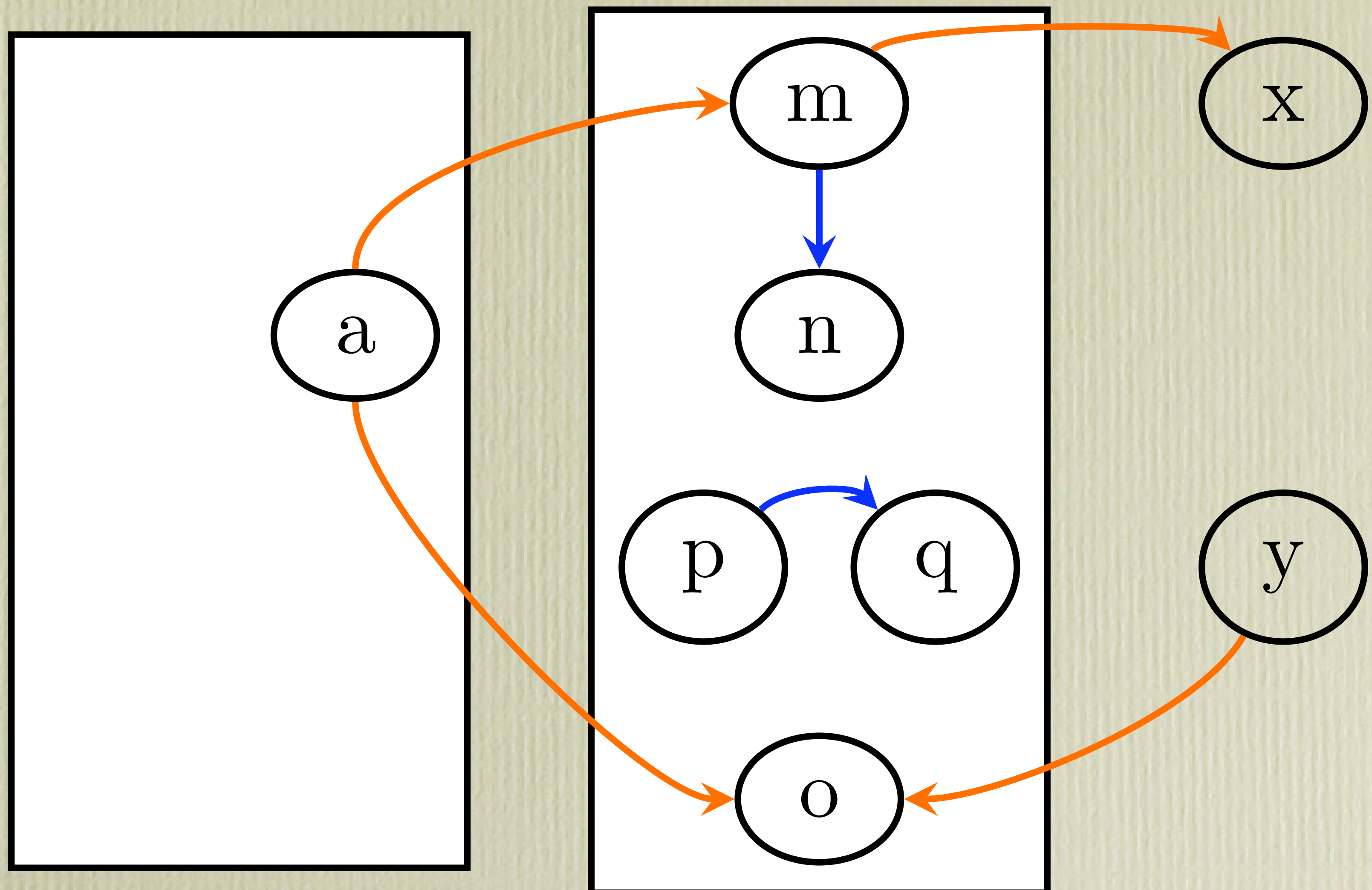


the paper discusses this more

# SATB Refinement Illustrated

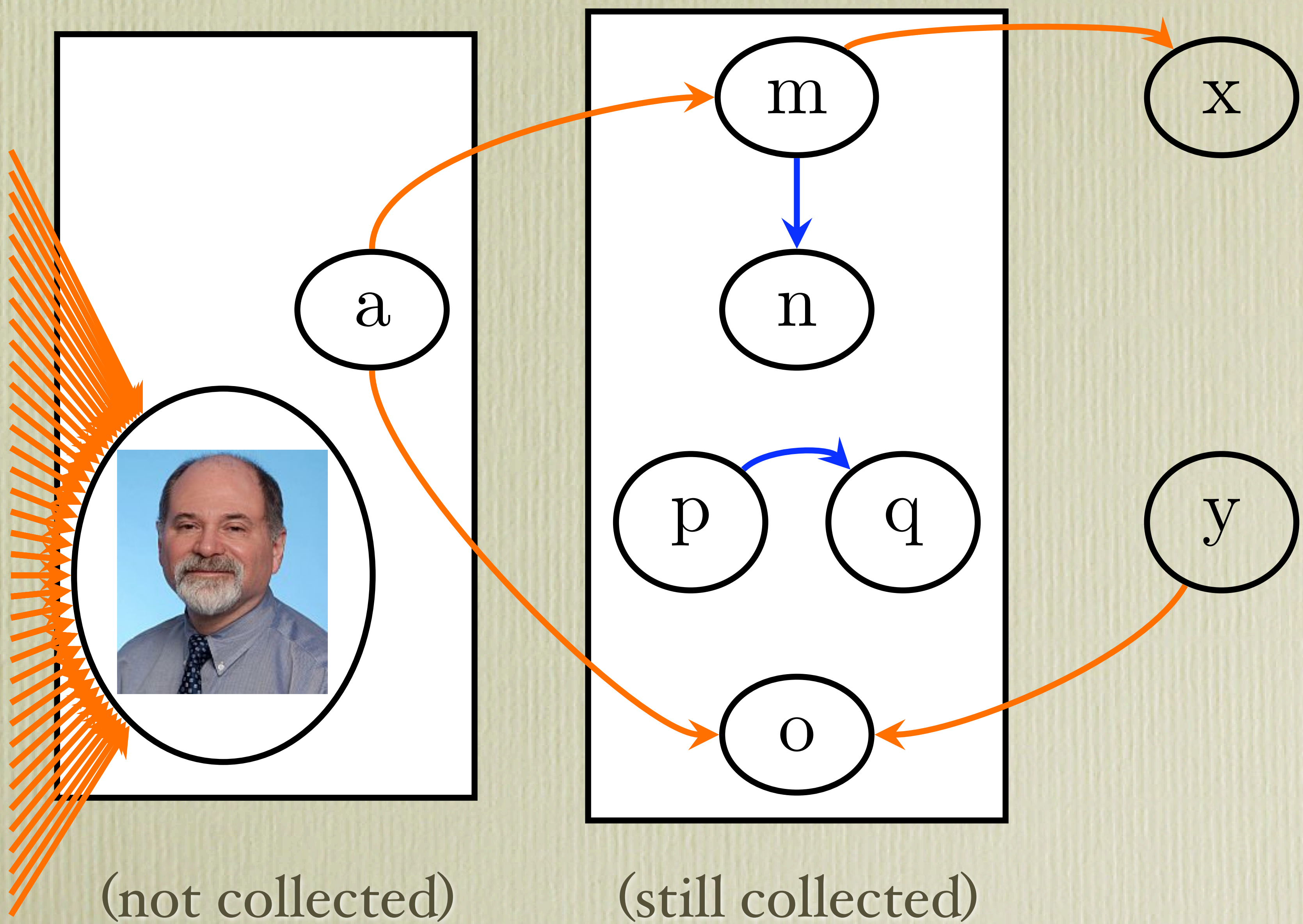


# SATB Refinement Illustrated

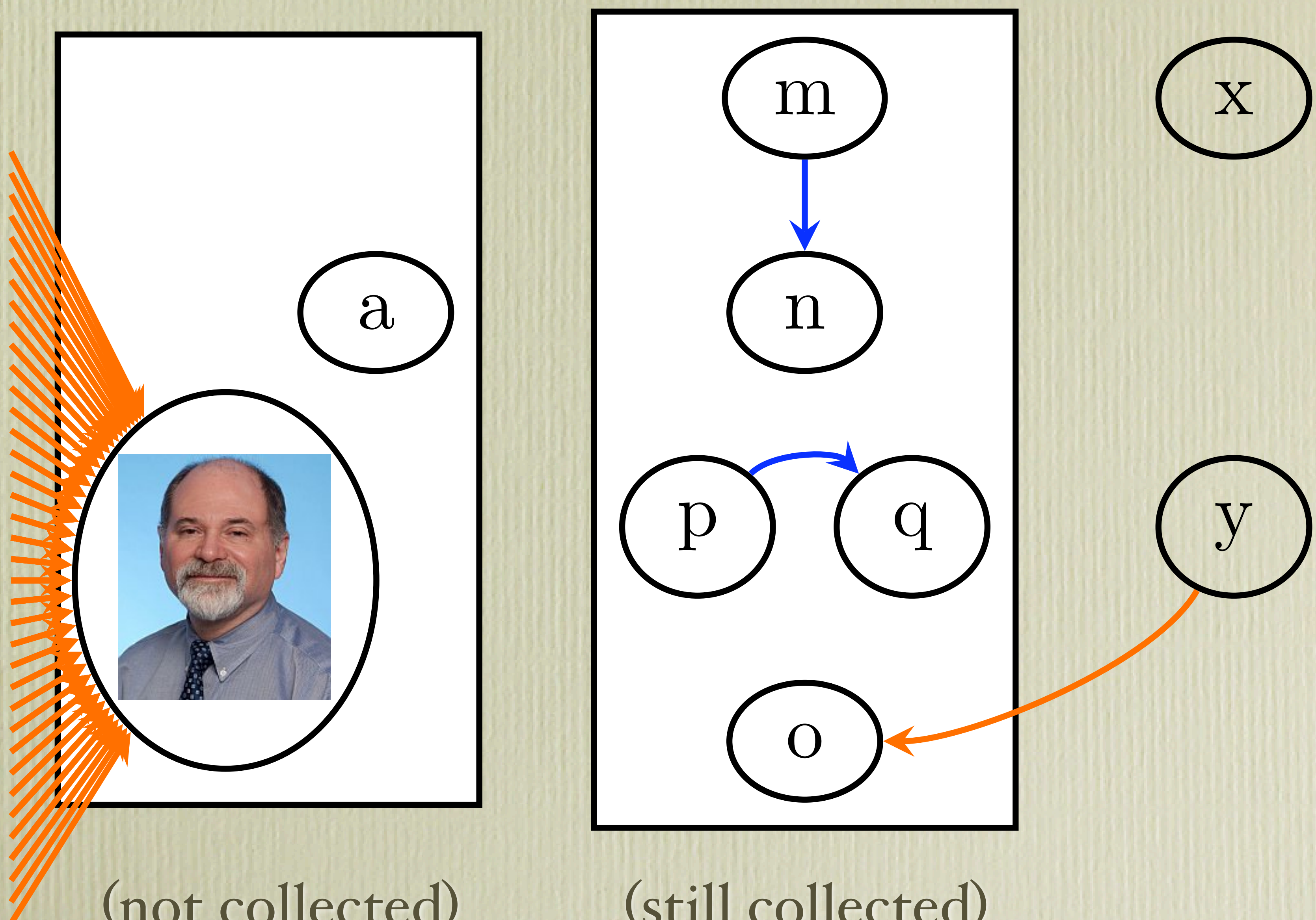



what if (a) were unreachable and in a region with popular objects?

# Before Refinement



# After Refinement




(not collected) (still collected)

# Prototype implementation in Larceny

# Evaluation:
# One Difficult Benchmark

# The Queue Benchmark

- Repeatedly:
  - allocate list of one million elements
  - store each list into circular buffer of size *k*
- List elements drawn from *p* popular objects
  - (for *p* of zero,  list elements are small integers)
- Regular steady-state behavior
  - approximating pause time: “MaxVary”



MaxVary: calculated by subtracting the average time to create a million element list from the longest time to create one of those lists.  Reasonably approximates pause time under certain assumptions

# Comparison against: Larceny, other Schemes, and JVM

# queue 160 MB

| System | Elapsed | GC Time | GC Pause | MaxVary | MaxRSIZE |
|---|---|---|---|---|---|
| Larceny *R* | 192 sec | 170 sec | 0.07 sec | 0.60 sec | 386 MB |
| Gambit | 63 sec | 44 sec | | 0.52 sec | 493 MB |
| Ypsilon | 265 sec | ≥53 sec | 0.64 sec | (?) | 711 MB |
| SunJVM *G* | 175 sec | ? | | 0.78 sec | 333 MB |
| Larceny *G* | 109 sec | 88 sec | 0.80 sec | 0.88 sec | 555 MB |
| SunJVM *P* | 275 sec | ? | | 0.91 sec | 511 MB |
| Larceny *S* | 76 sec | 55 sec | 0.90 sec | 0.94 sec | 518 MB |
| Chicken | 87 sec | 36 sec | | 1. sec | 490 MB |
| PLT | 227 sec | 211 sec | | 1. sec | 617 MB |
| Ikarus | 264 sec | 242 sec | | 2.25 sec | 1055 MB |
| SunJVM *I* | 409 sec | ? | | 3.41 sec | 530 MB |



Gambit: Stop&Copy; Ypsilon: MostlyConcurrent; Chicken: Cheney-on-the-MTA; PLT: Generatational; Ikarus: Generational. Suffixes legend:: Regional, Generational, Parallel, Stopandcopy, IncrementalMarkSweep

# queue 800 MB

| System | Elapsed | GC Time | GC Pause | Max Vary | MaxRSIZE |
|---|---|---|---|---|---|
| Larceny *R* | 212 sec | 187 sec | 0.11 sec | 0.7 sec | 1808 MB |
| Ypsilon | 24971 sec | ≥24818 s | 2.4 sec | (?) | 2067 MB |
| Gambit | 68 sec | 47 sec | | 2.5 sec | 2363 MB |
| Chicken | 118 sec | 62 sec | | 4. sec | 1955 MB |
| SunJVM *P* | 311 sec | ? | | 4.2 sec | 1973 MB |
| Larceny *G* | 149 sec | 128 sec | 4.2 sec | 4.3 sec | 2073 MB |
| Larceny *S* | 119 sec | 95 sec | 4.5 sec | 4.5 sec | 2058 MB |
| SunJVM *G* | 212 sec | ? | | 4.9 sec | 1497 MB |
| PLT | 286 sec | 273 sec | | 5. sec | 2109 MB |
| Ikarus | 419 sec | 371 sec | | 11.6 sec | 2575 MB |
| SunJVM *I* | 457 sec | ? | | 15.8 sec | 2083 MB |



Gambit: Stop&Copy; Ypsilon: MostlyConcurrent; Chicken: Cheney-on-the-MTA; PLT: Generatational; Ikarus: Generational. Suffixes legend:: Regional, Generational, Parallel, Stopandcopy, IncrementalMarkSweep

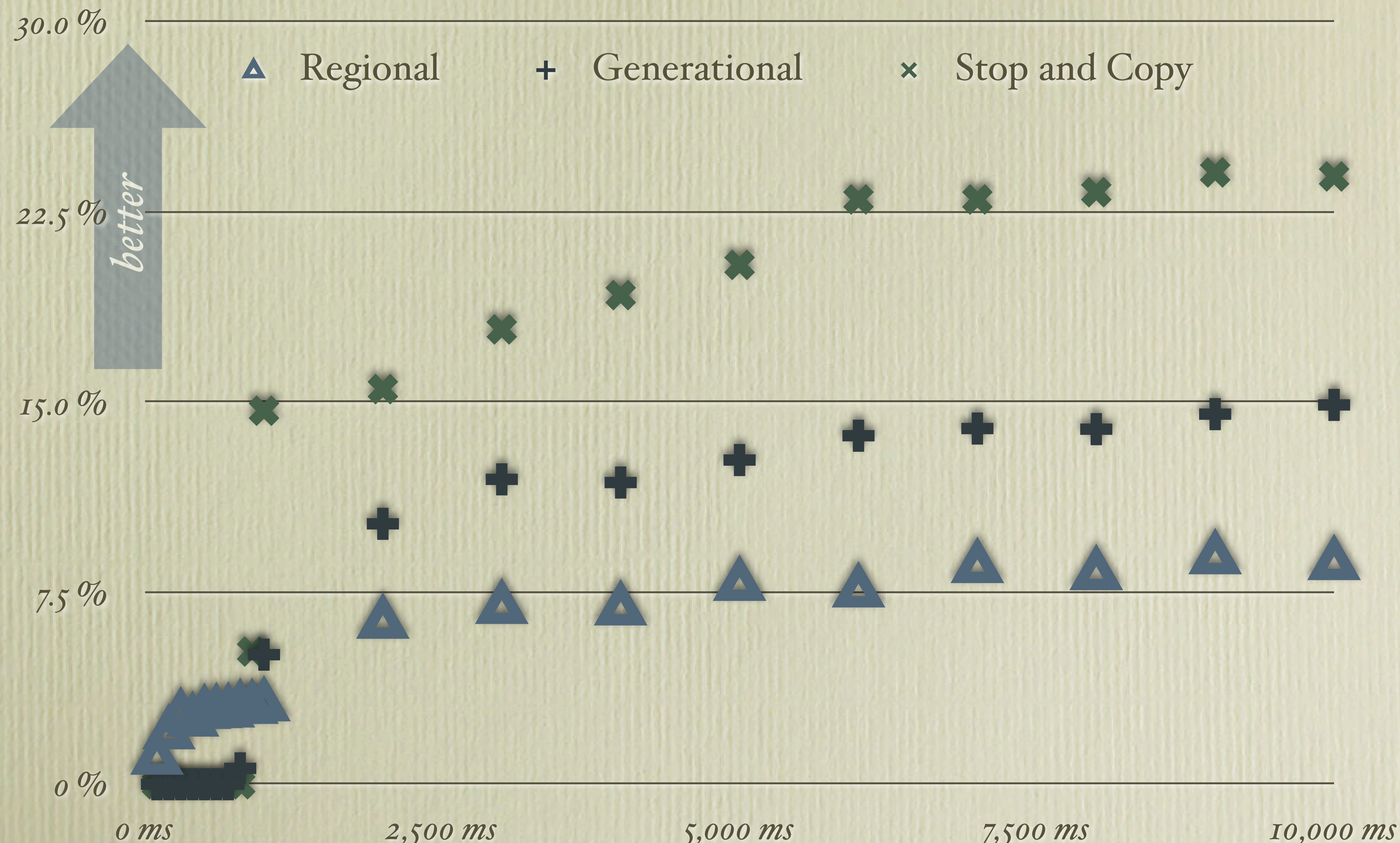
# queue 800 MB + 50 pop. obj.'s

| System | Elapsed | GC Time | GC Pause | Max Vary | MaxRSIZE |
|---|---|---|---|---|---|
| Larceny *R* | 618 sec | 592 sec | 0.35 sec | 2.9 sec | 1865 MB |
| Gambit | 72 sec | 51 sec | | 2.7 sec | 2363 MB |
| Ypsilon | 28366 sec | ≥28212 s | 2.89 sec | (?) | 1772 MB |
| SunJVM *P* | 314 sec | ? | | 4.1 sec | 1918 MB |
| Larceny *G* | 162 sec | 141 sec | 4.5 sec | 4.6 sec | 2064 MB |
| Larceny *S* | 120 sec | 96 sec | 4.8 sec | 4.8 sec | 2060 MB |
| Chicken | 127 sec | 69 sec | | 5. sec | 1955 MB |
| SunJVM *G* | 216 sec | ? | | 5.0 sec | 1497 MB |
| PLT | 339 sec | 320 sec | | 5. sec | 2089 MB |
| Ikarus | 427 sec | 409 sec | | 10.7 sec | 2588 MB |
| SunJVM *I* | 479 sec | ? | | 18.1 sec | 2083 MB |



Gambit: Stop&Copy; Ypsilon: MostlyConcurrent; Chicken: Cheney-on-the-MTA; PLT: Generatational; Ikarus: Generational. Suffixes legend:: Regional, Generational, Parallel, Stopandcopy, IncrementalMarkSweep

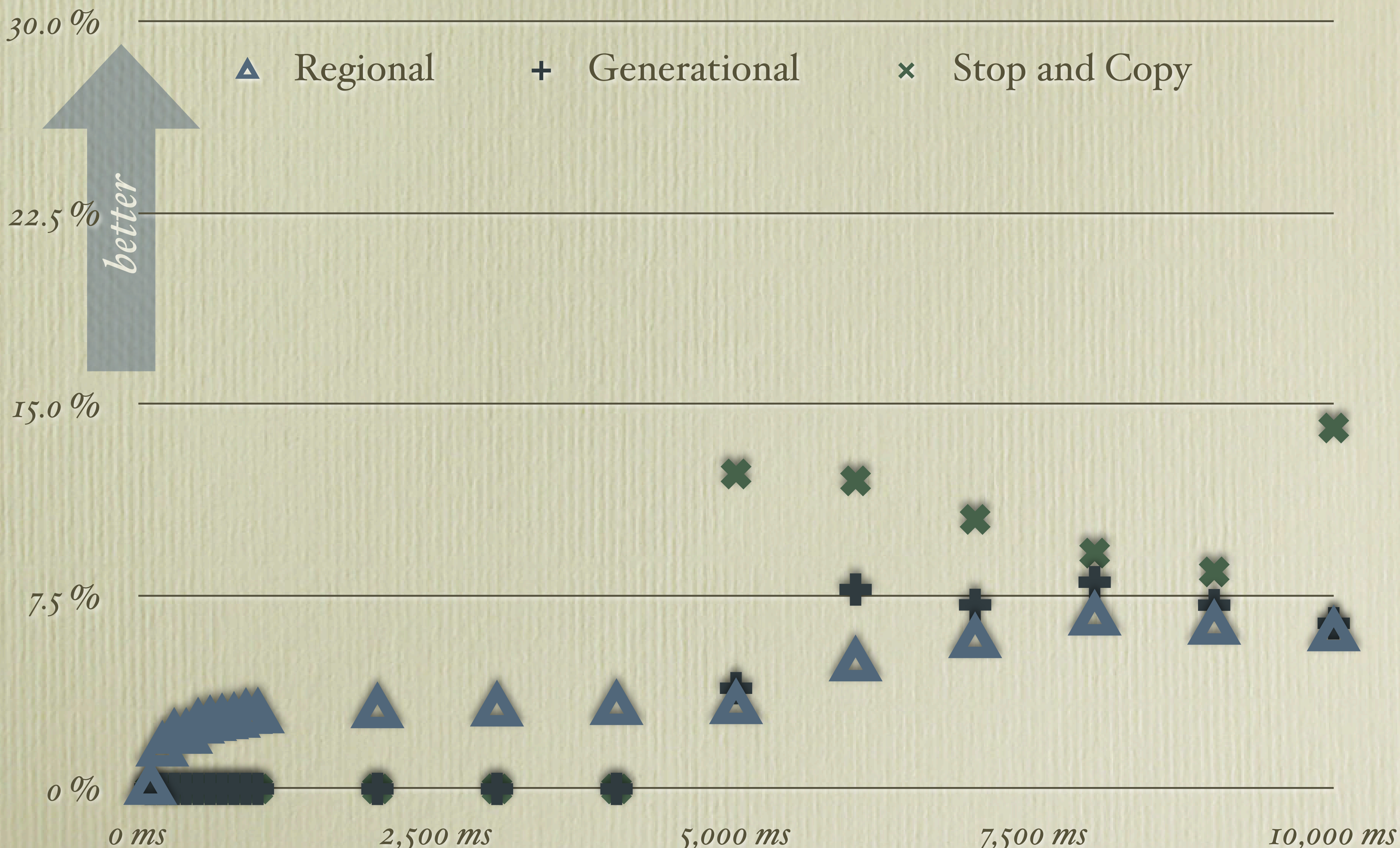
# Observed MMU, 160 MB live






Point out that the blur of triangles is a bunch of entries for the Regional Collector!

# Observed MMU, 800 MB live



# Related Work (lots)

- "Windows" of MarkCopy [Sachindran & Moss '03]
- Parallel Incremental Compaction [Ben-Yitzhak et al '02]
- Older-First GC [Stefanovic et al. '02] [Hansen and Clinger '02]
- Metronome [Bacon et al 2003]



Windows are much like our Summary Sets; Parallel Incr Compaction also had something much like our Summary Set Structure. (The specific goals of those systems differed from our own.)
For OLDER FIRST, the connection is that our round robin selection of regions to collect is very much like an older-first stategy.
For METRONOME, they do provide guarantees on MMU and SPACE USAGE, but only when given a priori models of the particular mutator behavior (!). We have an explicit goal of not requiring such knowledge in the collector.

# Future Work

- SATB Marking and Summarization could be performed concurrently with the mutator
- Regional copying GC could itself be parallelized
- More thorough description of the implementation technologies (“what fun hacks did Felix need?”)

# Conclusion

- Described and prototyped a *regional* collector
  - novel, elegant solutions for popularity and float
- Proved that regional collector is *scalable*
- Compared performance for near worst case benchmark

# *Thanks*

(a slide I did not think to include at actual workshop, but WISH I HAD, as it completes the "slogan story" appropriately at the end of the design presentation)

incrementally

*Summarize* as deadline approaches;

*Refine* after ~~whole picture~~ has been developed

snapshot

(a slide I did not include at the end of presentation at actual workshop, but wish I had.)