A low-latency garbage collector for GHC

Ben Gamari Ömer Sinan Ağacan

Well-Typed

Another typical day...

```
$ cd my-program
$ cat MyProgram.hs
{-# LANGUAGE TypeInType #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE MagicHash #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE TypeFamilyDependencies #-}
{-# LANGUAGE DependentHaskell #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE PatternSynonyms #-}
{-# LANGUAGE MachineCognition #-}
```

Another typical day...

```
$ ghc -O2 -threaded -rtsopts MyProgram.hs
$ ./MyProgram +RTS -s

...

  10,349,259,832 bytes allocated in the heap
  22,353,166,880 bytes copied during GC
   4,983,805,744 bytes maximum residency (8 sample(s))
       2,066,896 bytes maximum slop
            4879 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0  8983 colls, 0 par  20.001s   0.315s     0.0042s    0.0252s
  Gen  1   242 colls, 0 par  30.024s   0.153s     0.0925s    1.3753s

...
```

What if this program was. . .

- rendering your cat video?
- drawing your text editor's UI?
- on the critical path in your distributed application?
- driving your car?
- controlling your surgical robot?

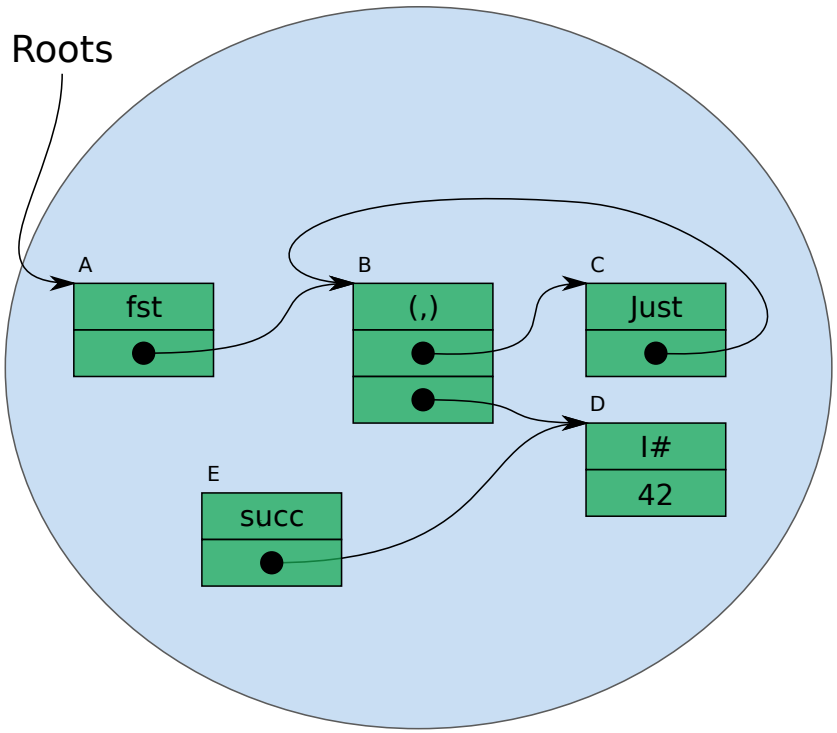
Why?

GHC (in its simplest configuration) has a

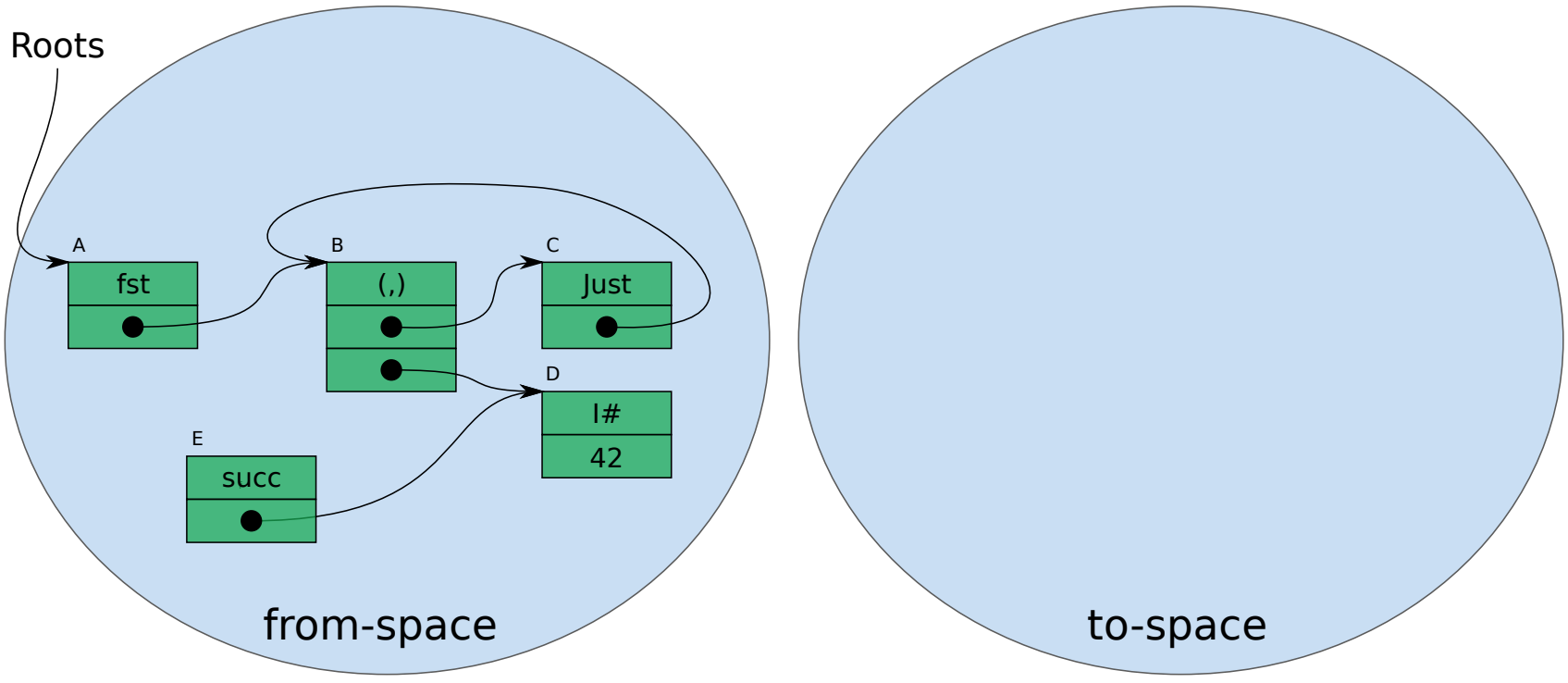
- generational
- two-space moving
- stop-the-world

garbage collector.

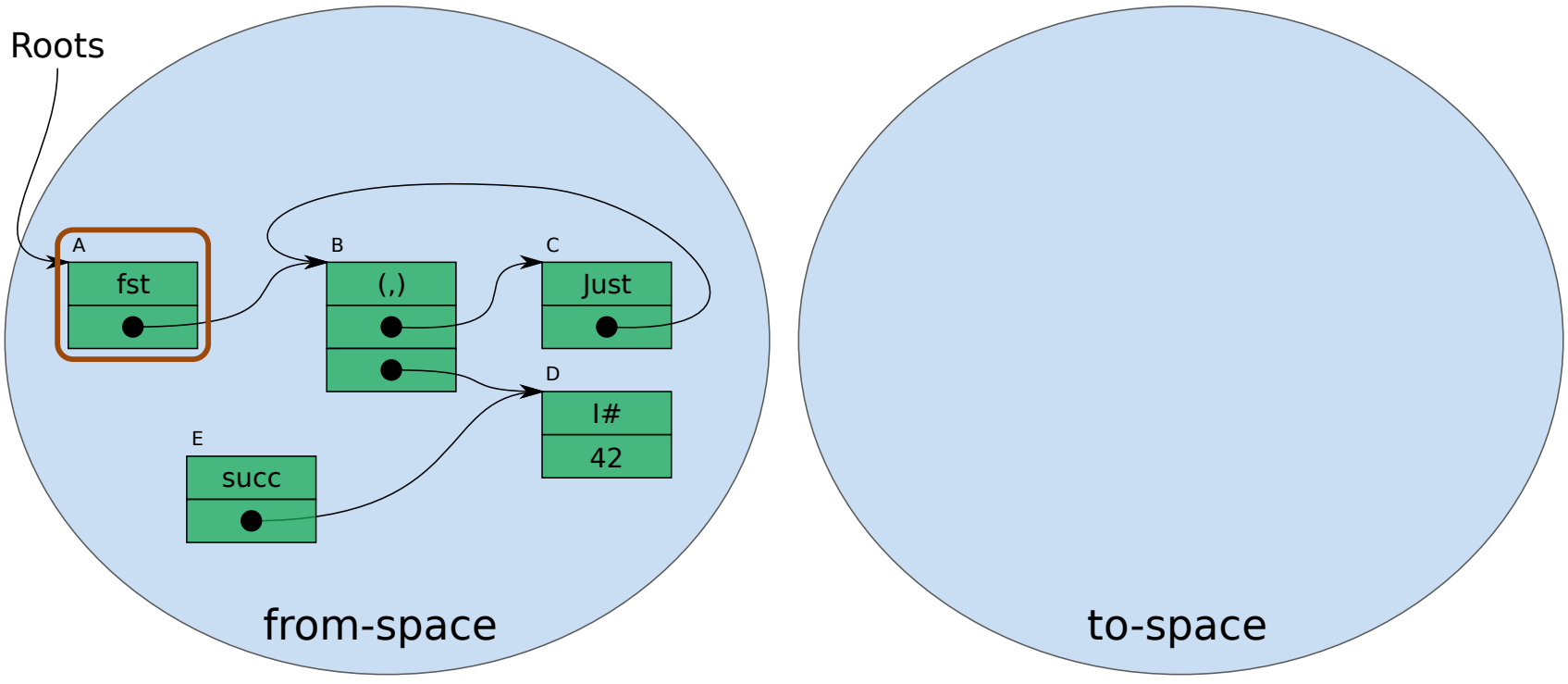
Moving garbage collection



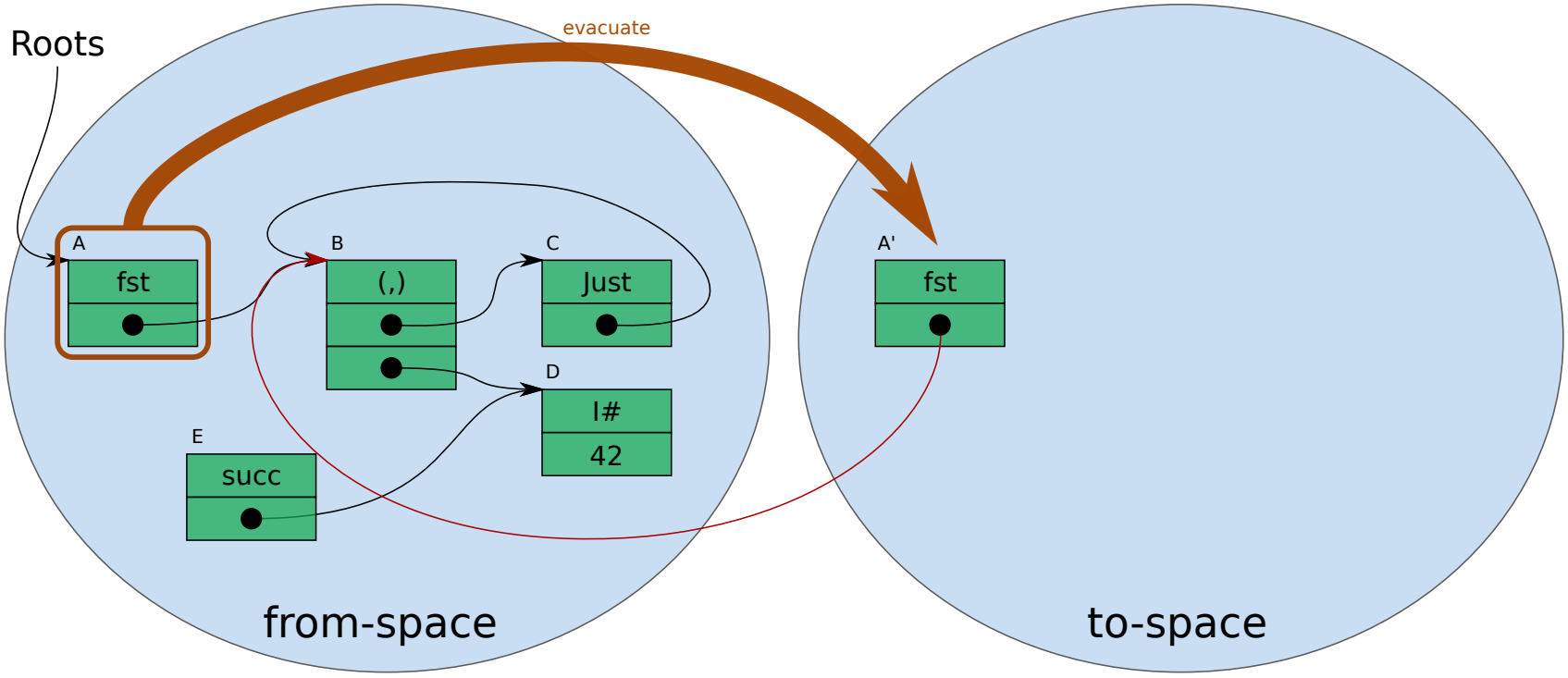
Moving garbage collection



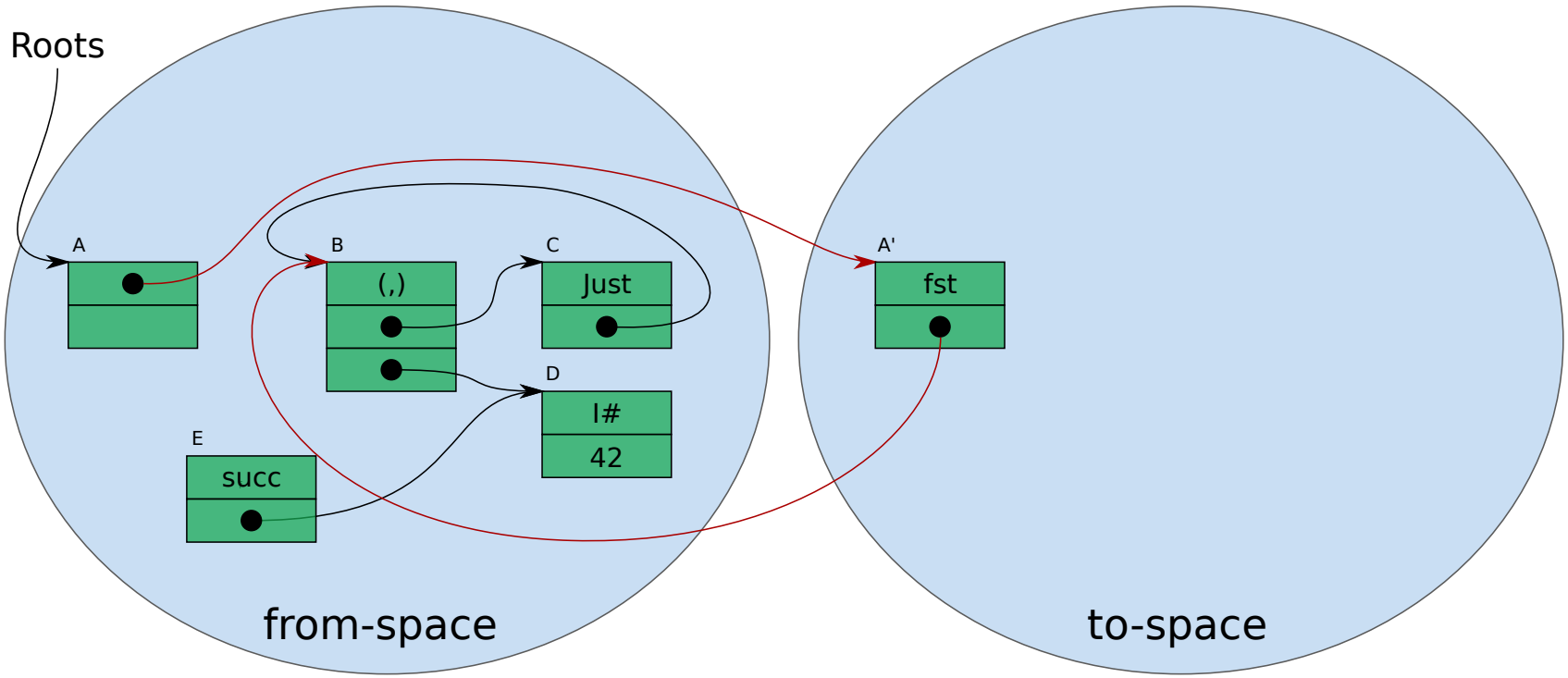
Moving garbage collection: Evacuate roots



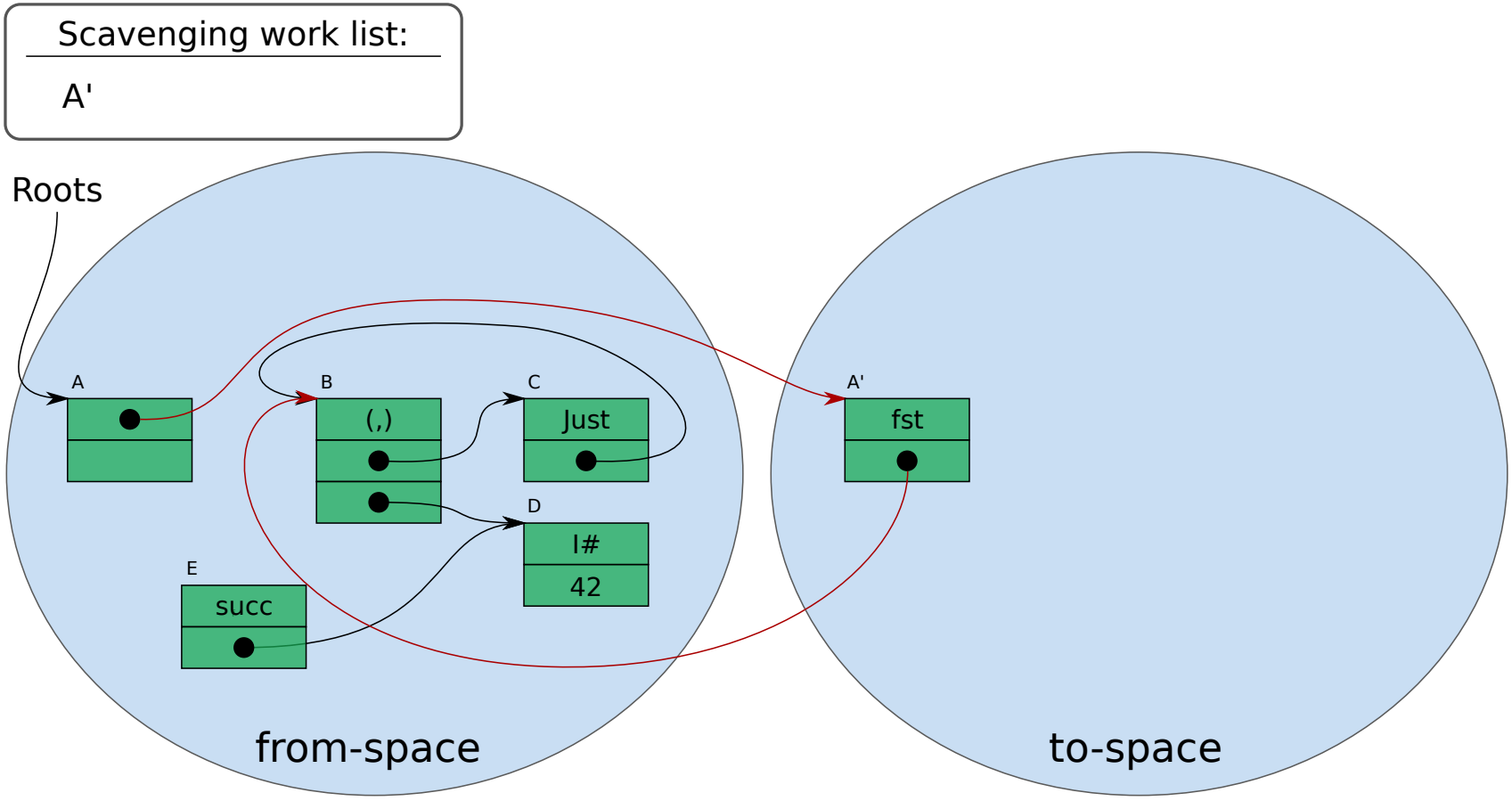
Moving garbage collection: Evacuate roots



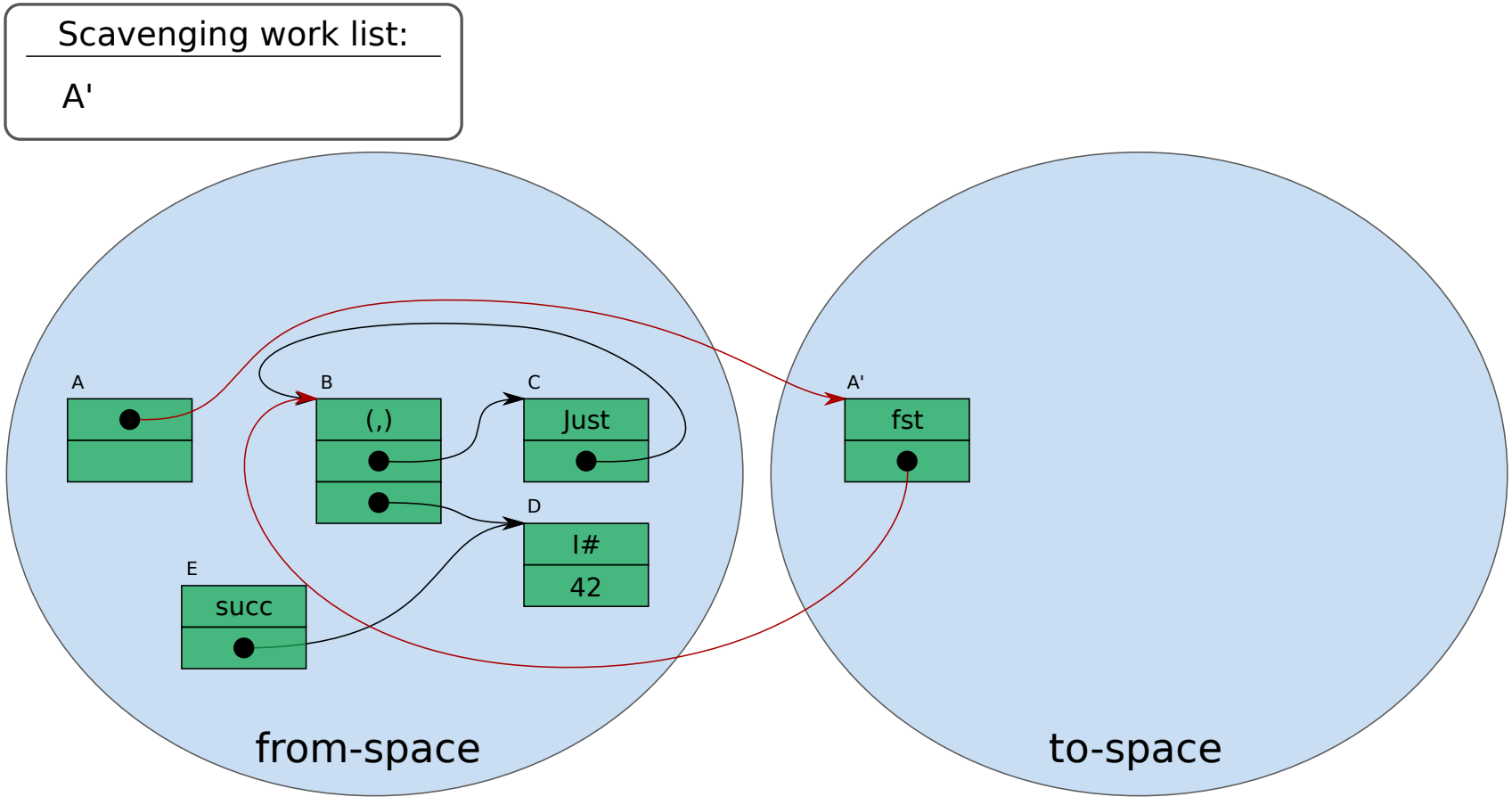
Moving garbage collection: Evacuate roots



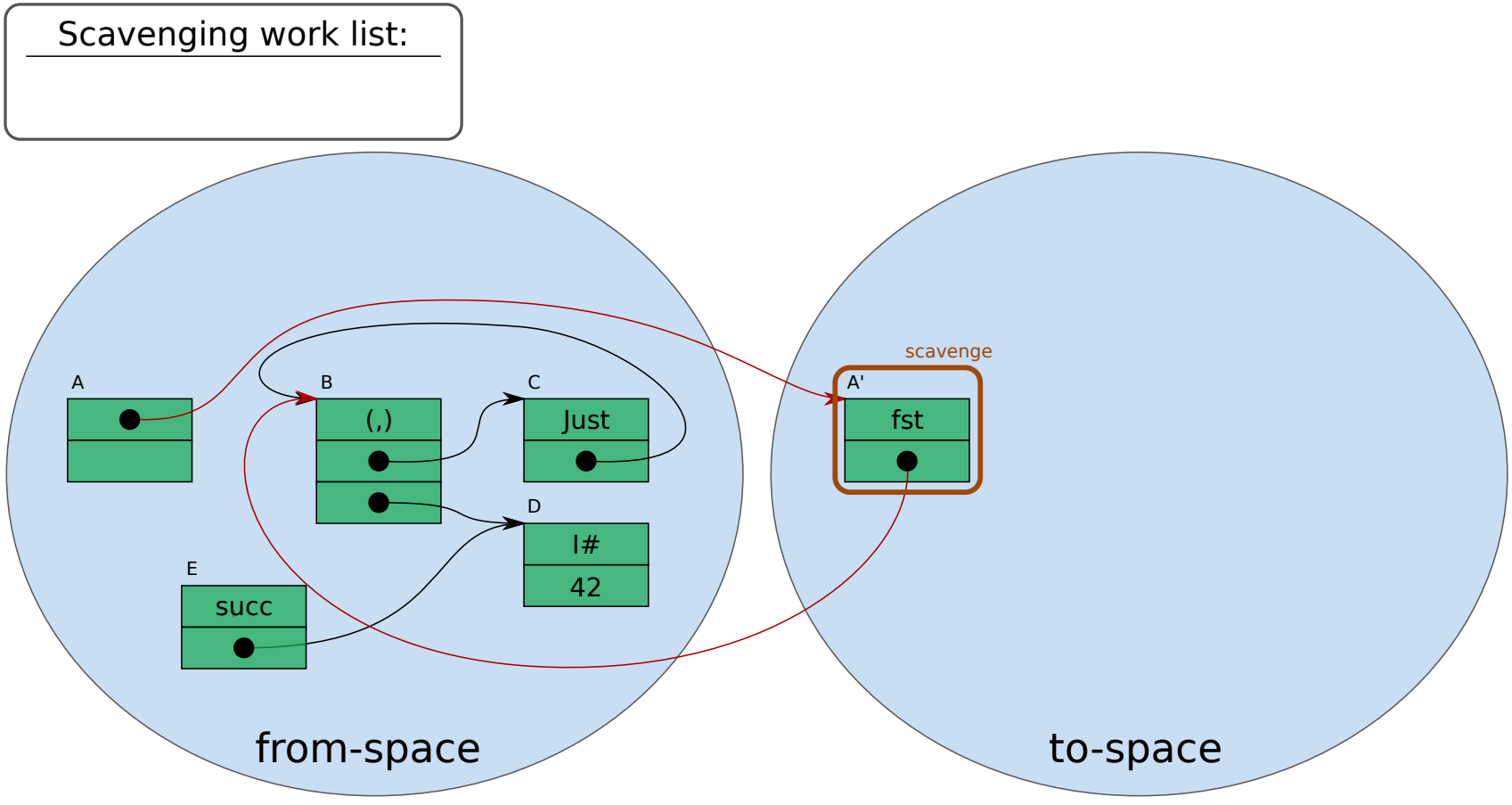
Moving garbage collection: Evacuate roots



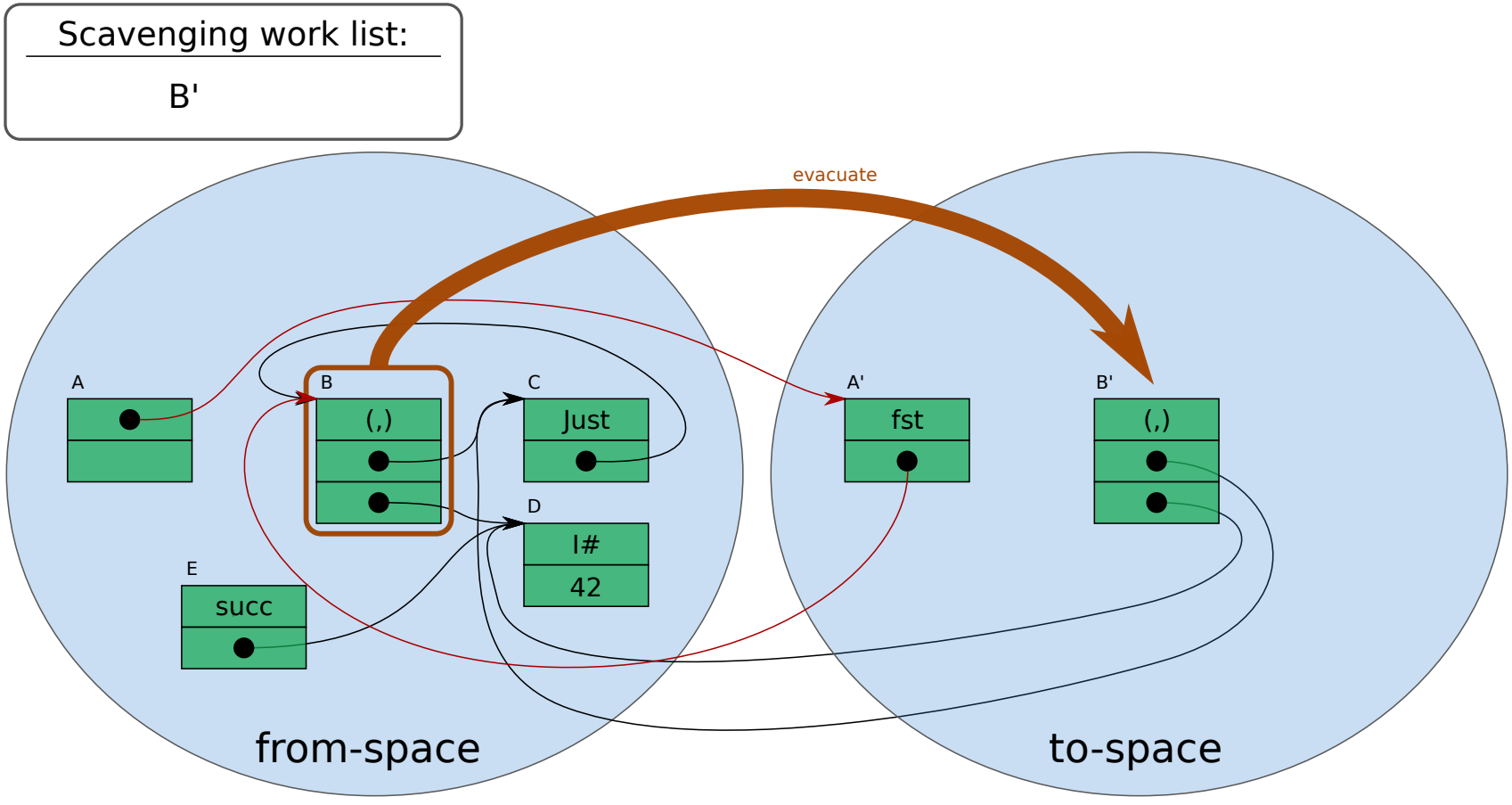
Moving garbage collection



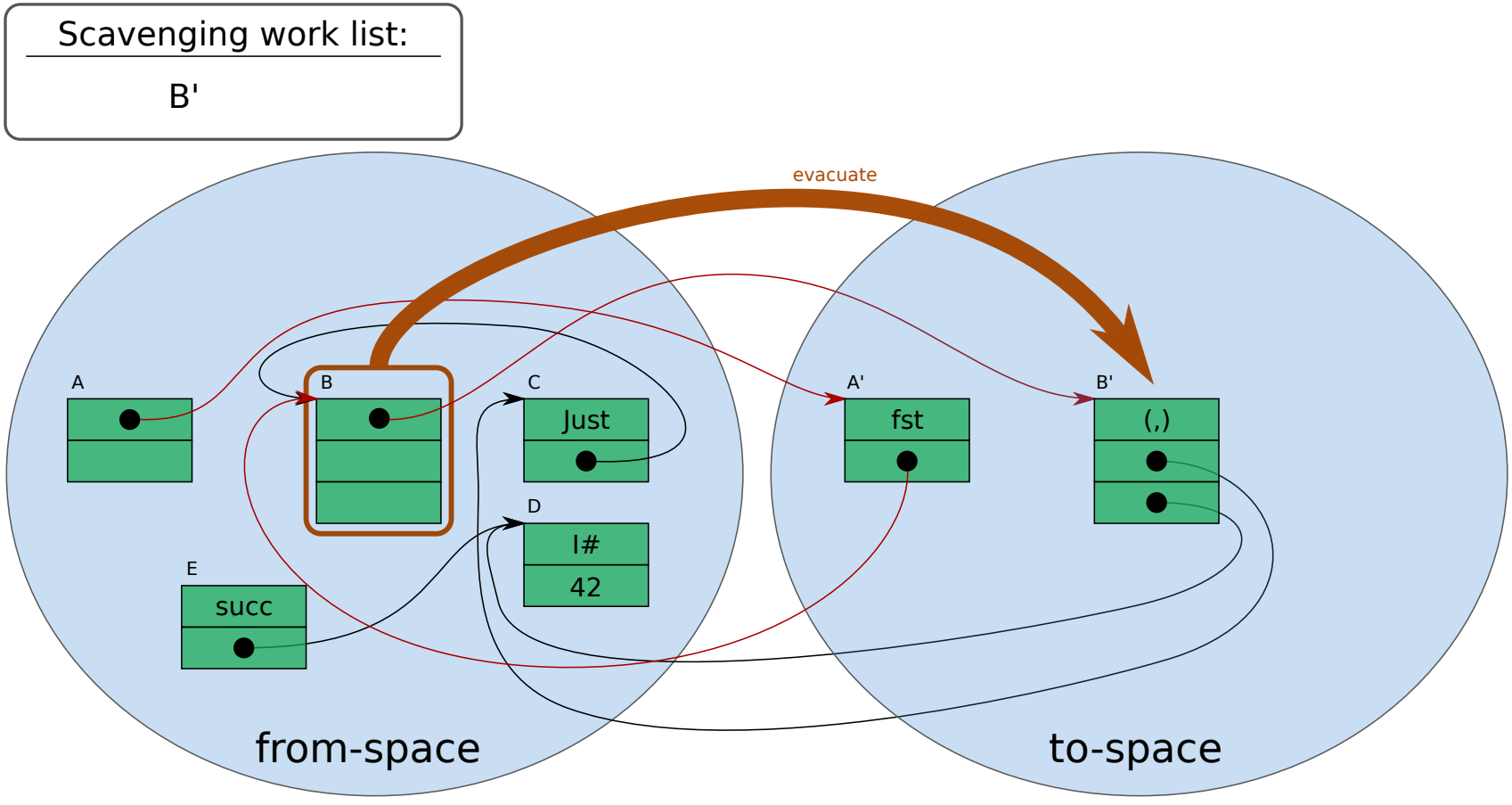
Moving garbage collection: Scavenge A



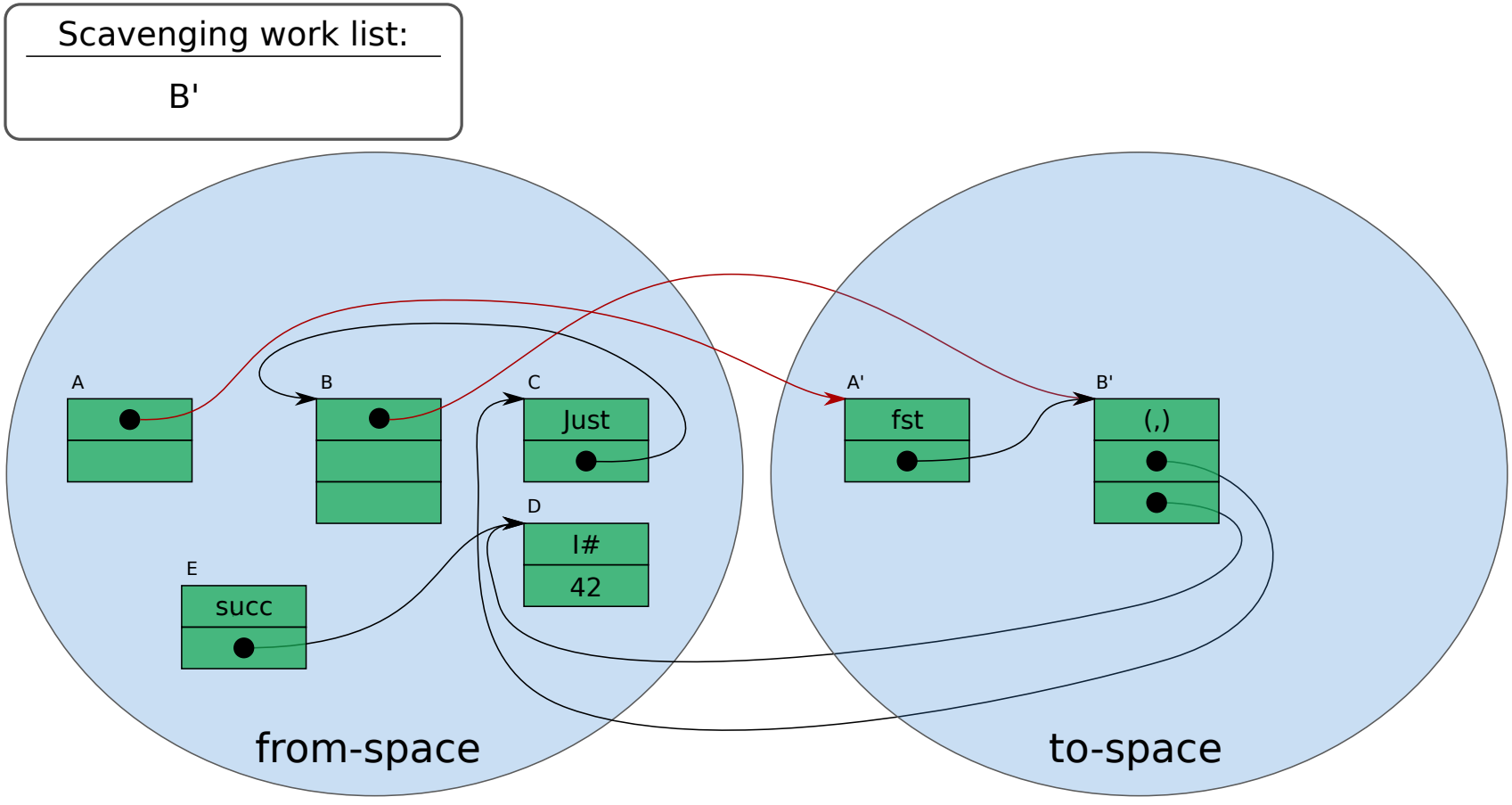
Moving garbage collection: Evacuate B



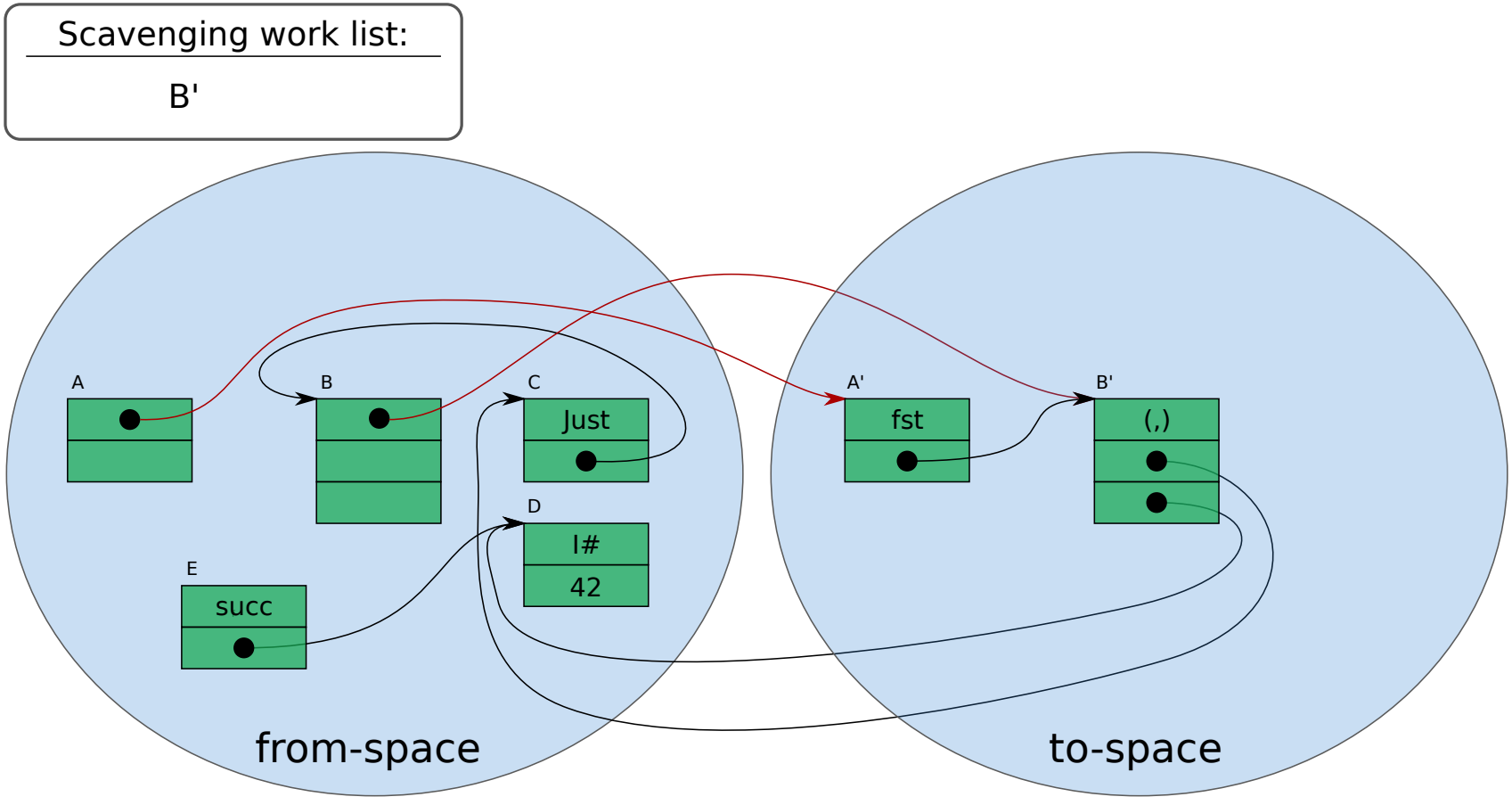
Moving garbage collection: Evacuate B



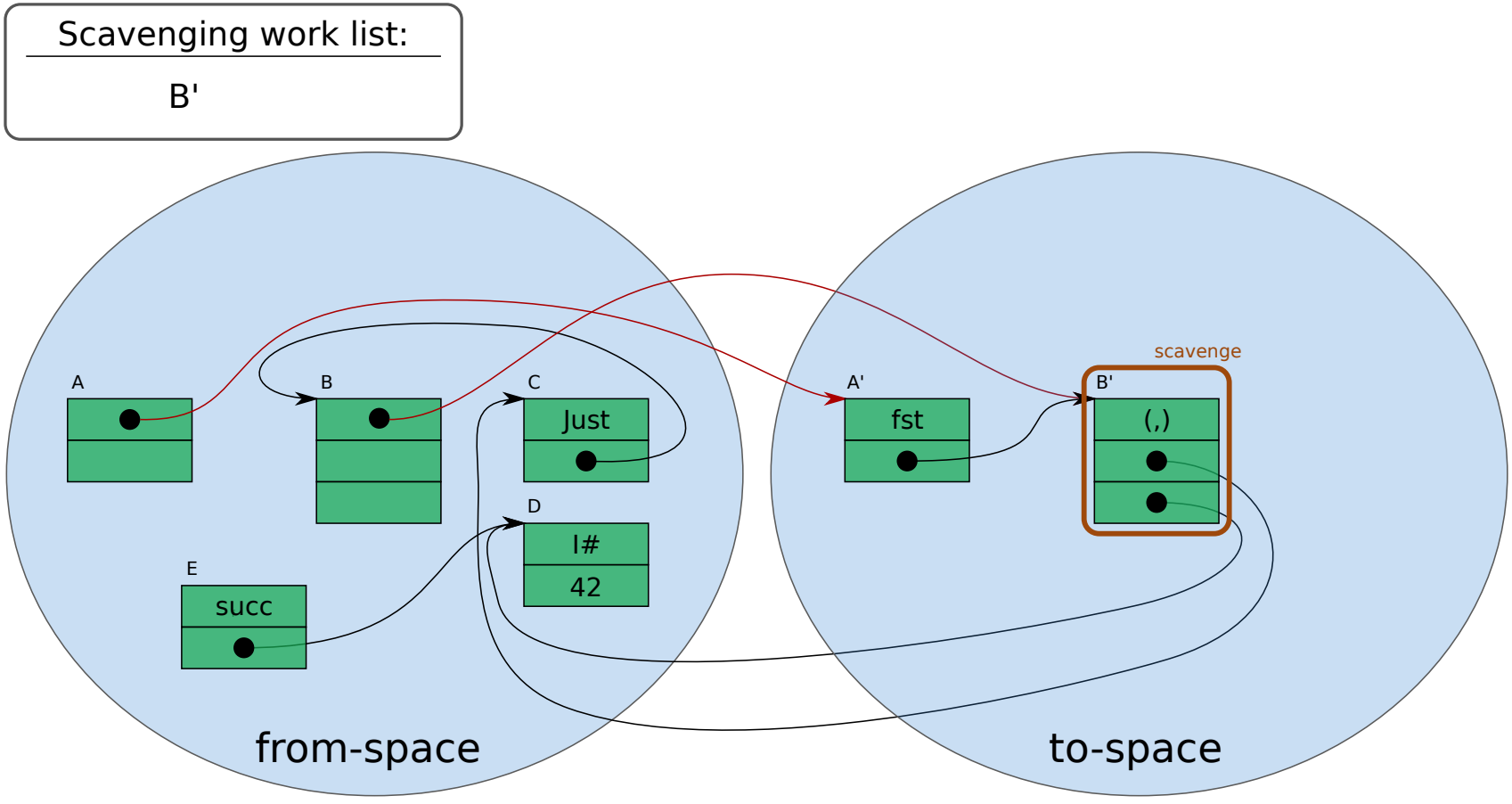
Moving garbage collection: Evacuate B



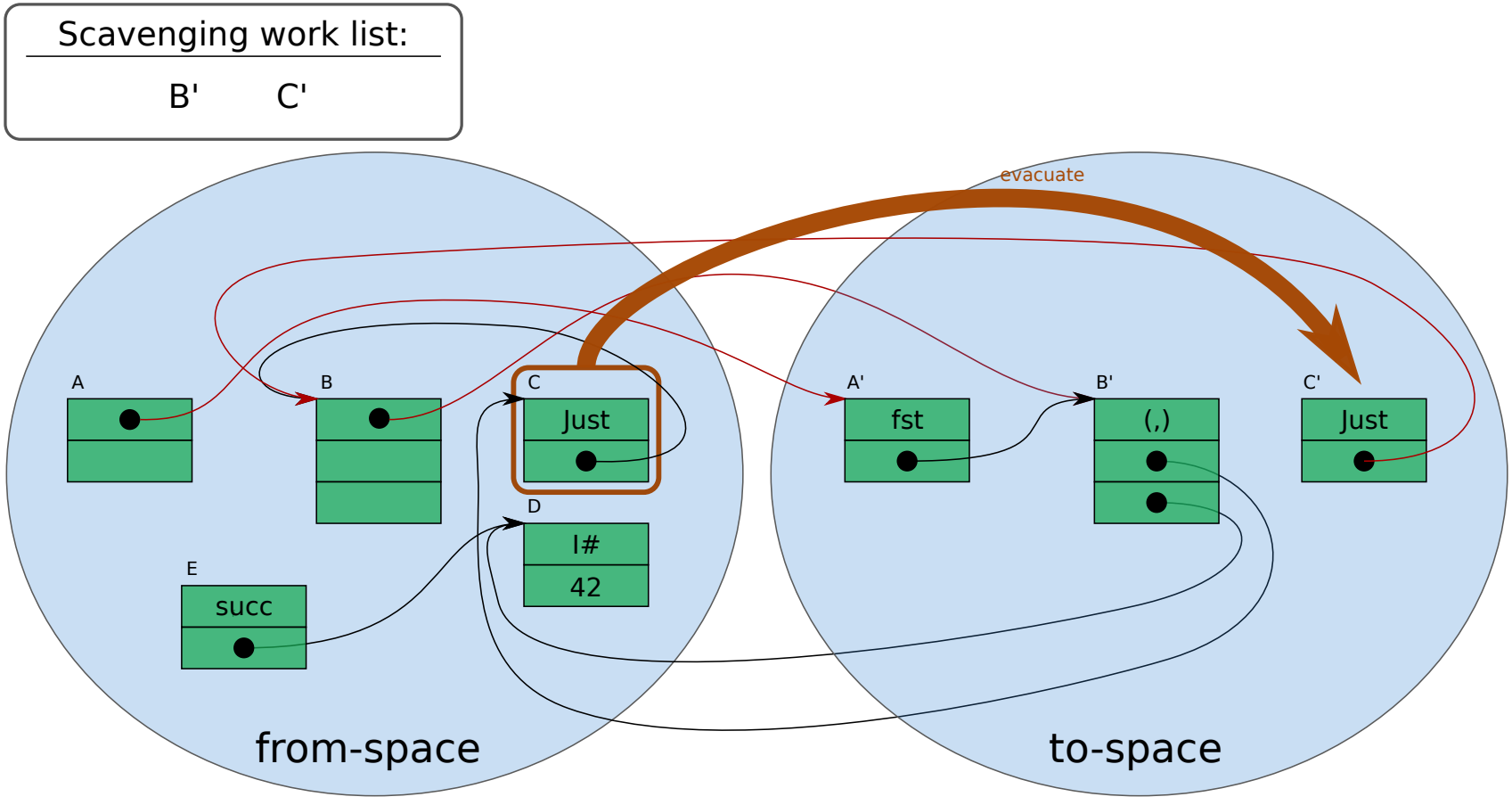
Moving garbage collection



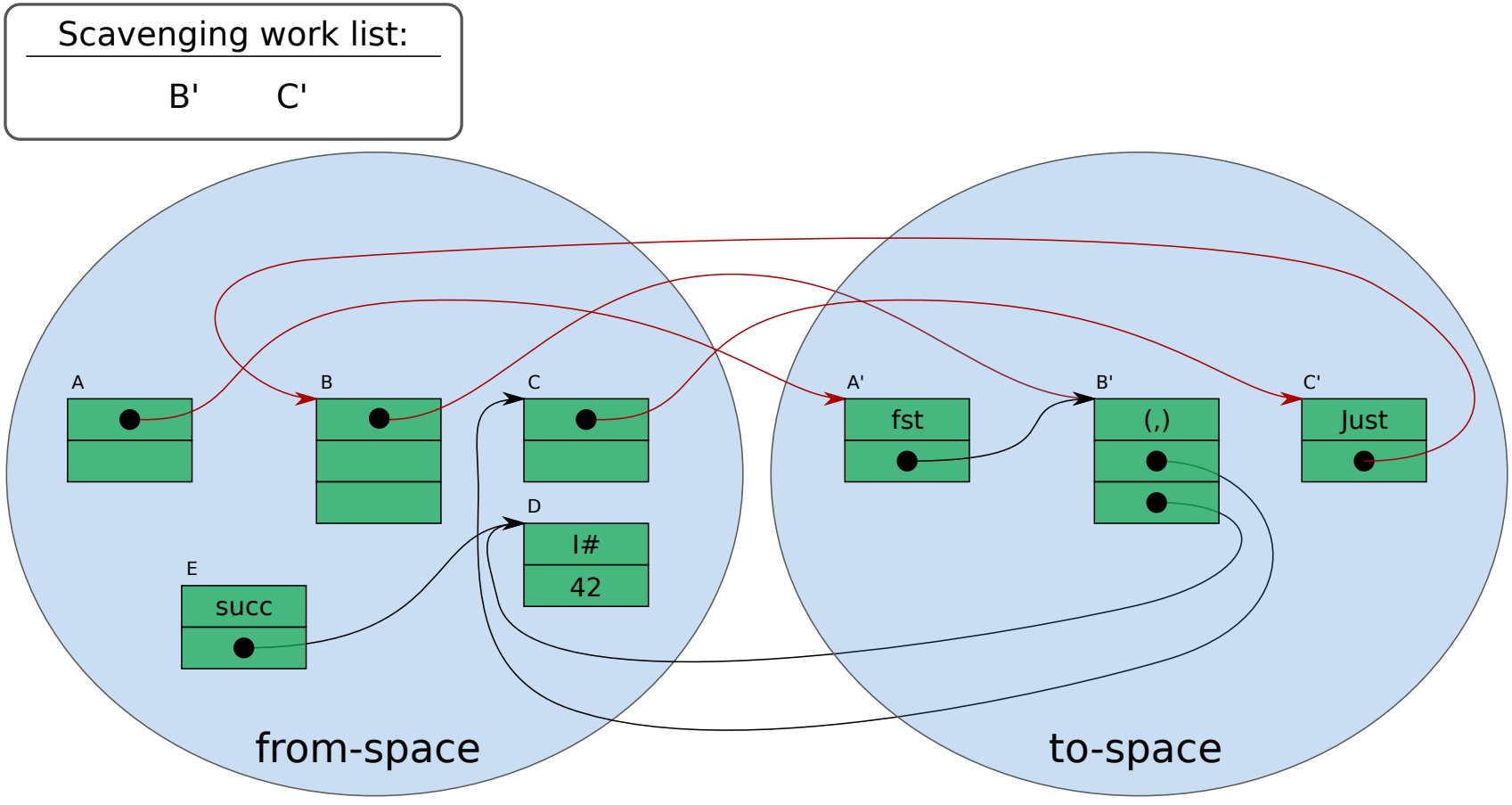
Moving garbage collection: Scavenge B



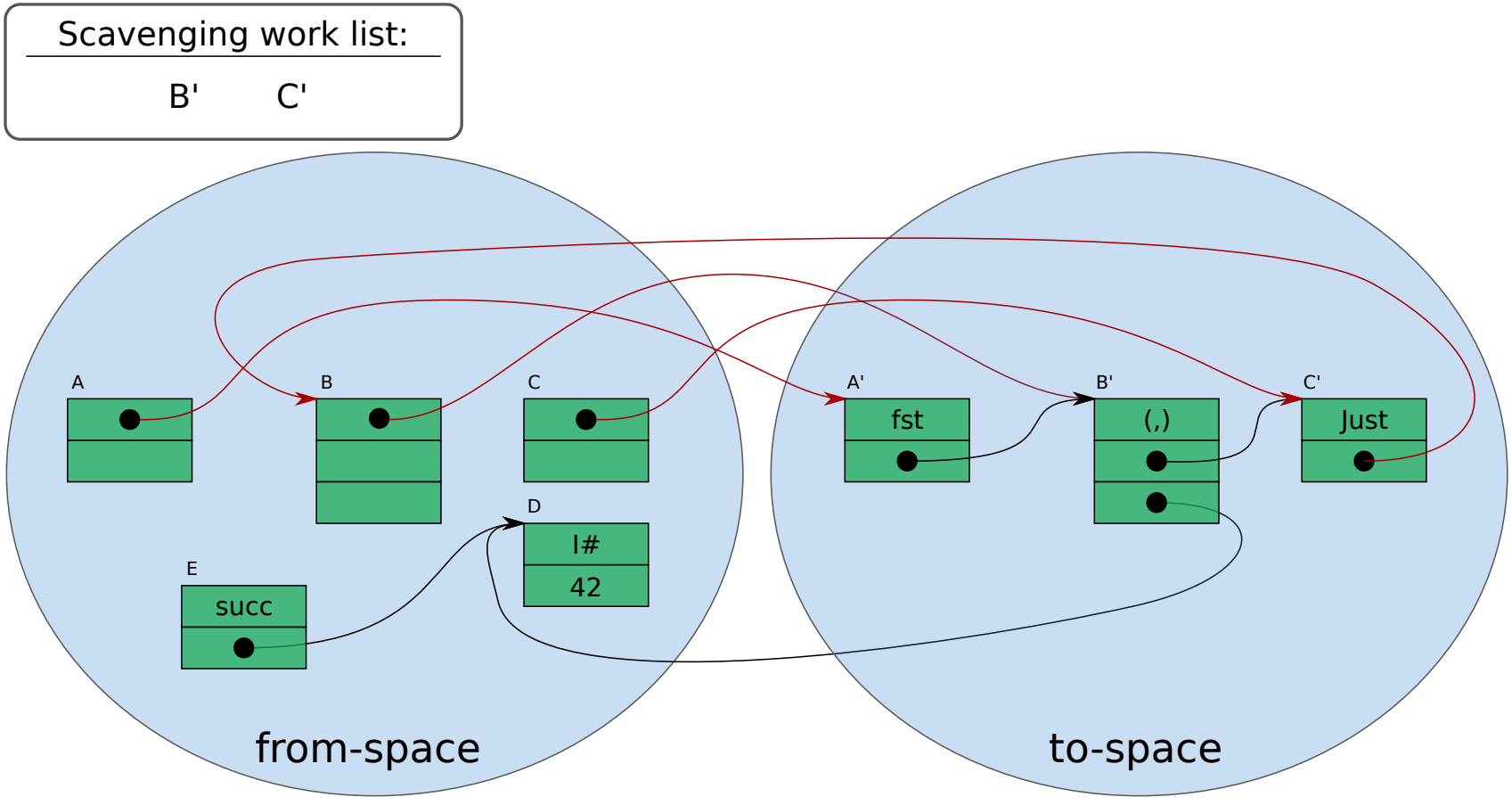
Moving garbage collection: Evacuate C



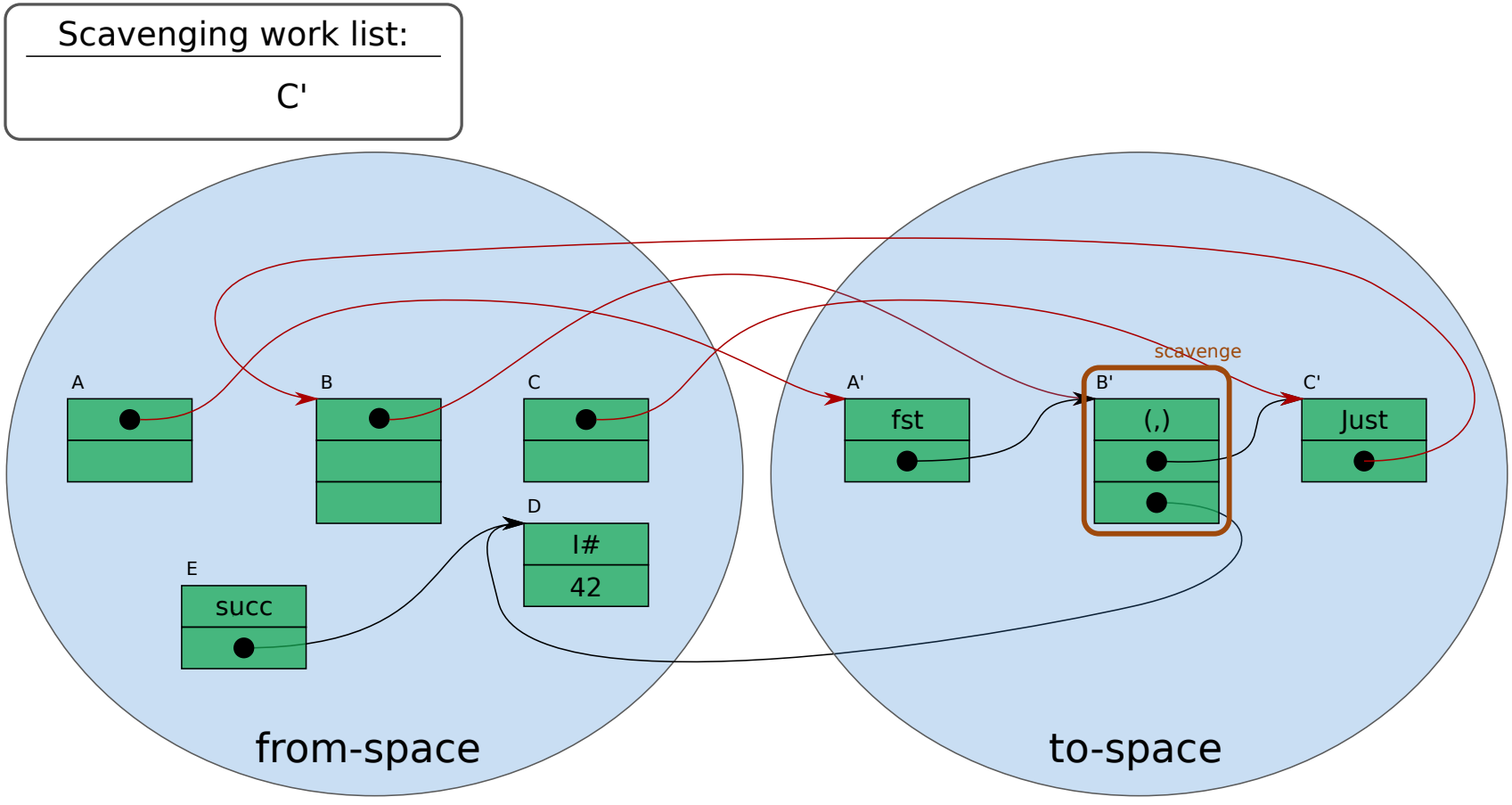
Moving garbage collection: Evacuate C



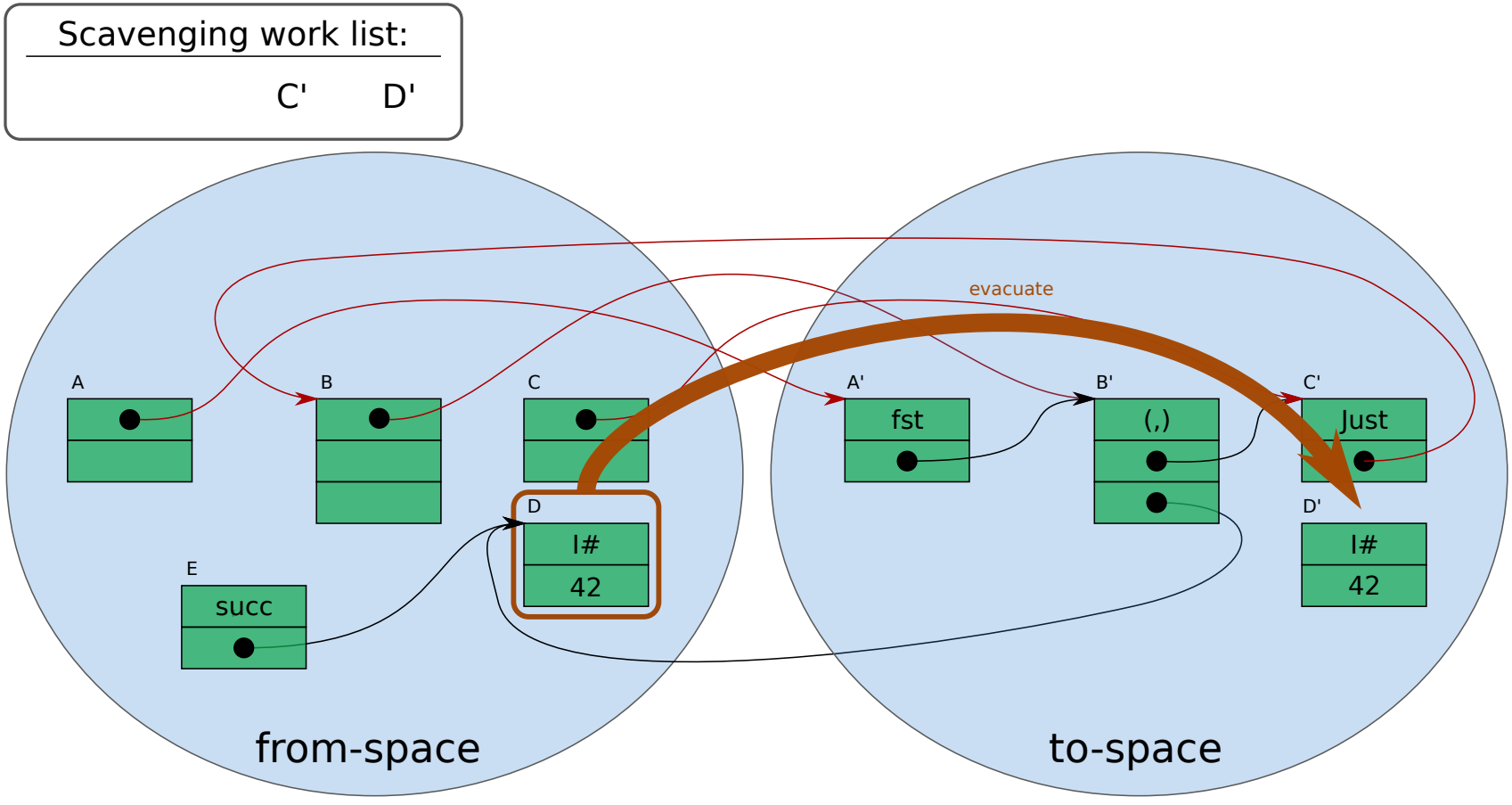
Moving garbage collection: Evacuate C



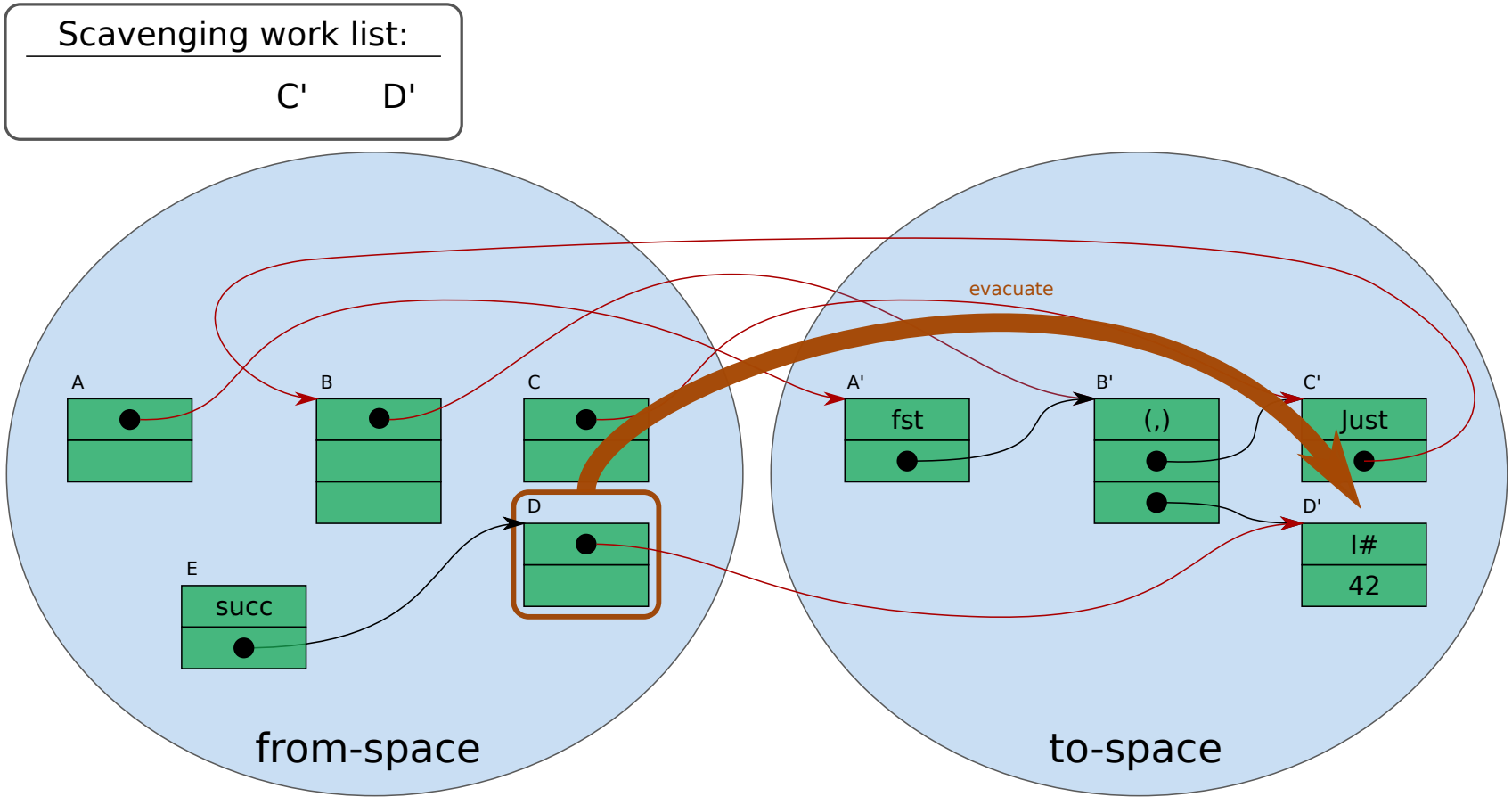
Moving garbage collection: Scavenging B (cont'd)



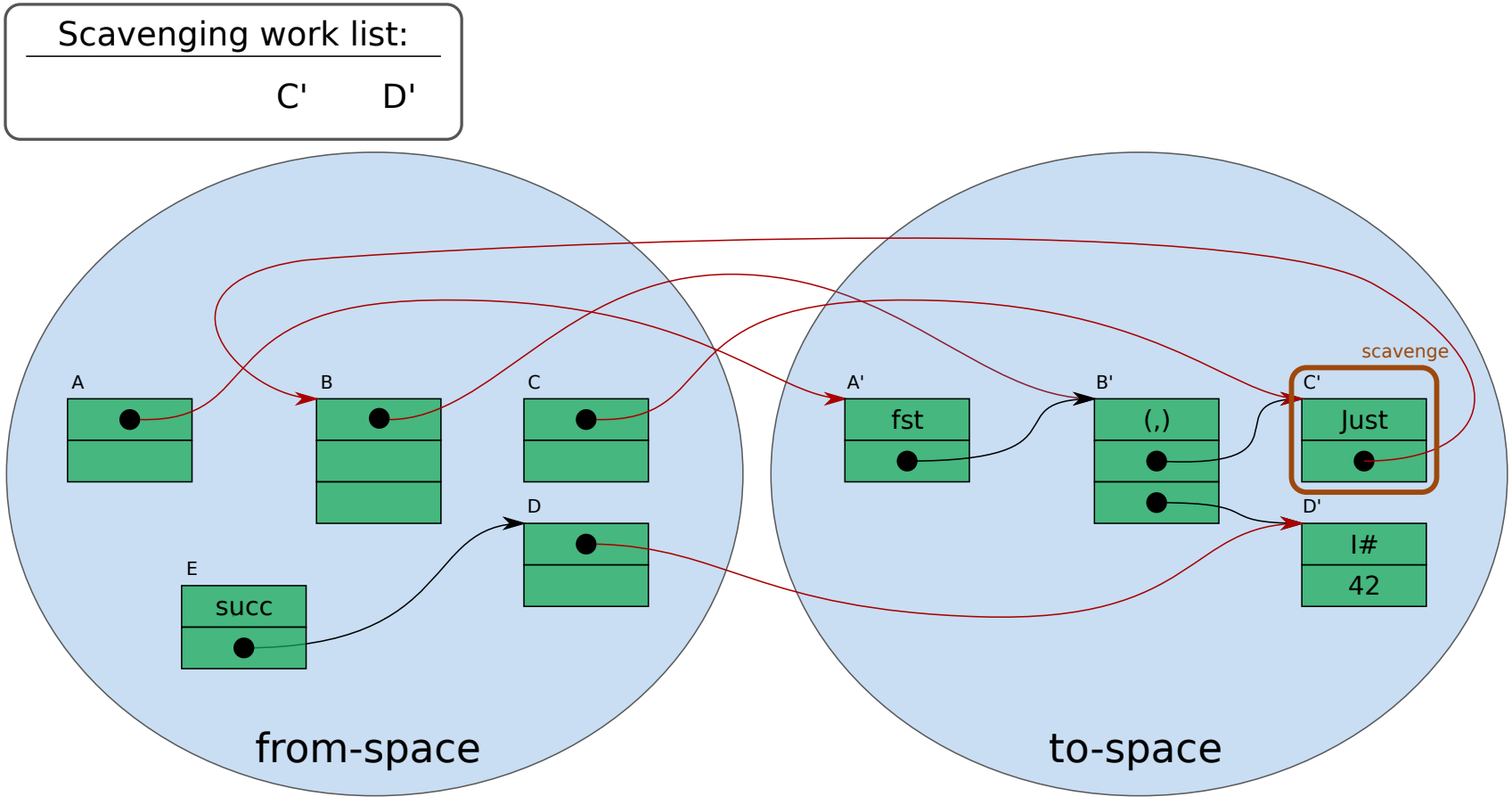
Moving garbage collection: Evacuate D



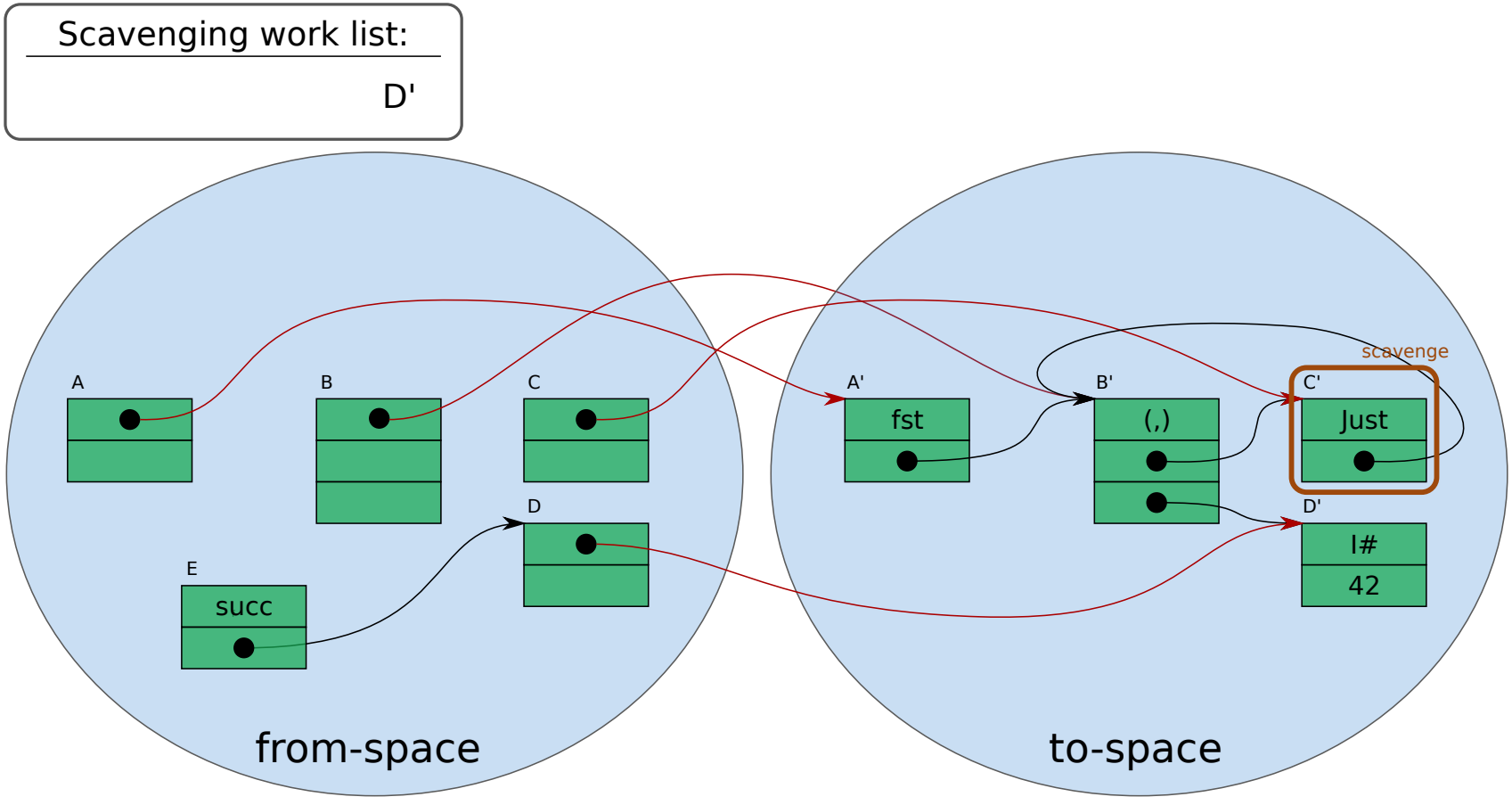
Moving garbage collection: Evacuate D



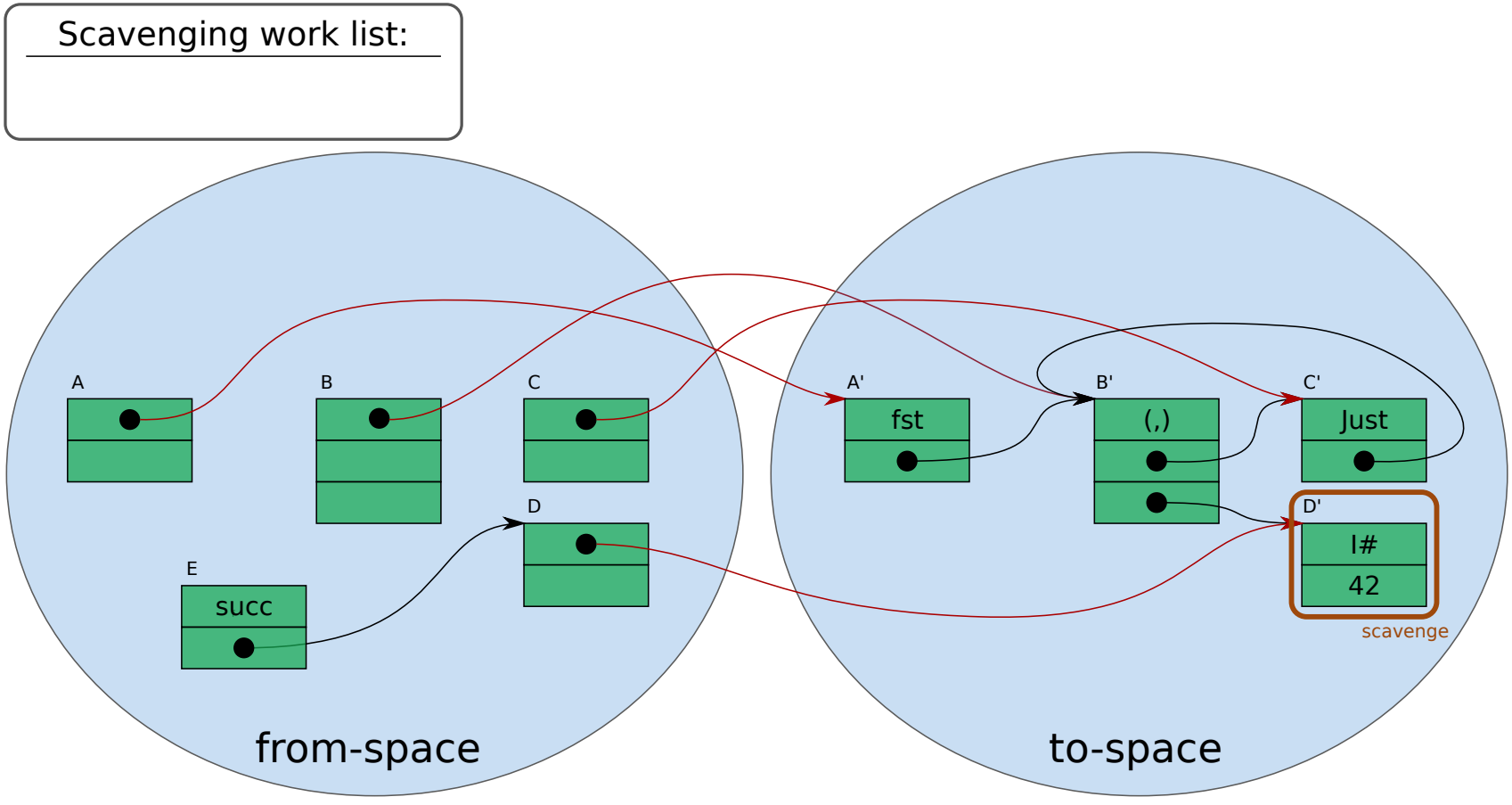
Moving garbage collection: Scavenging C



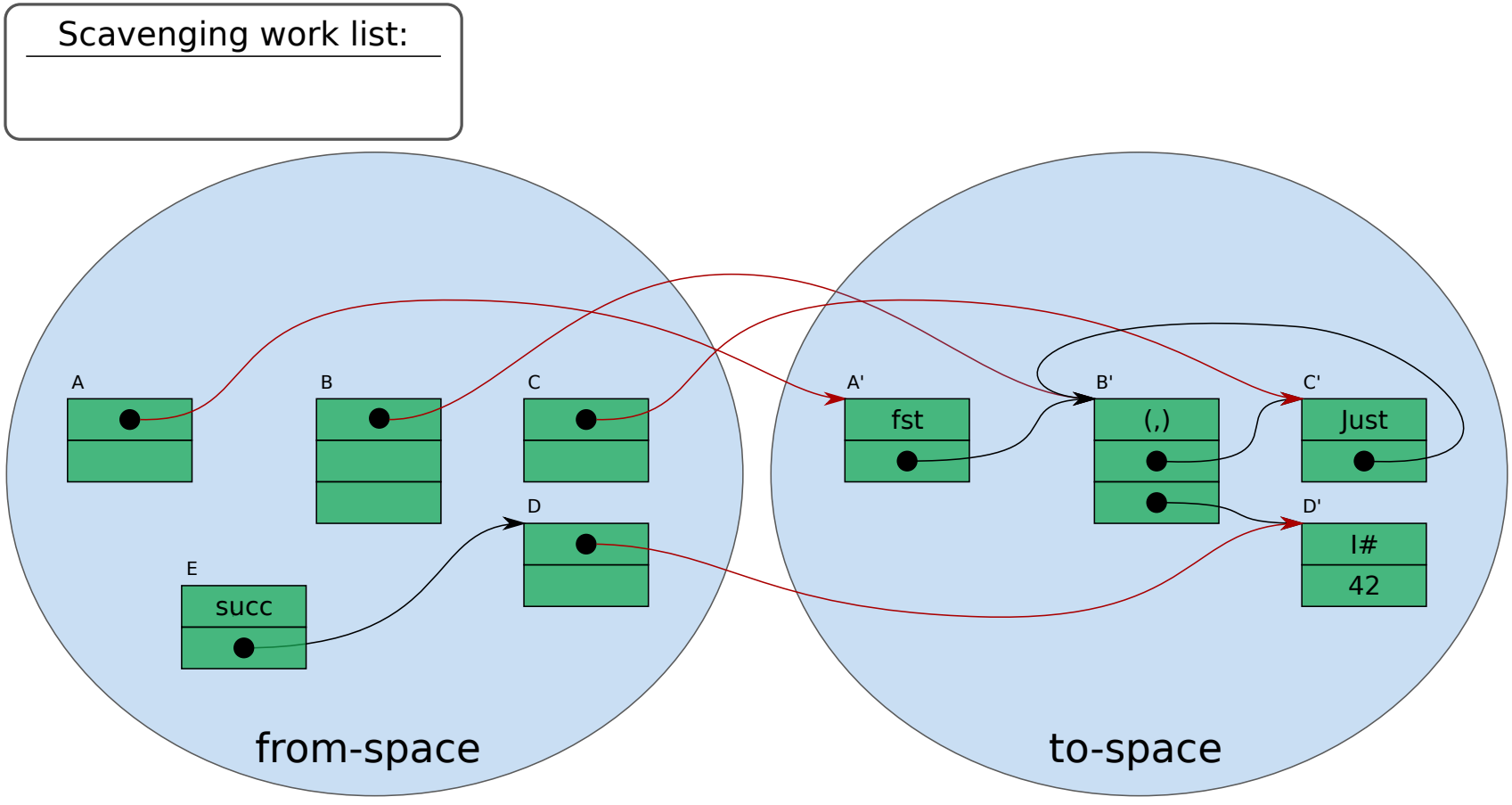
Moving garbage collection: Scavenging C



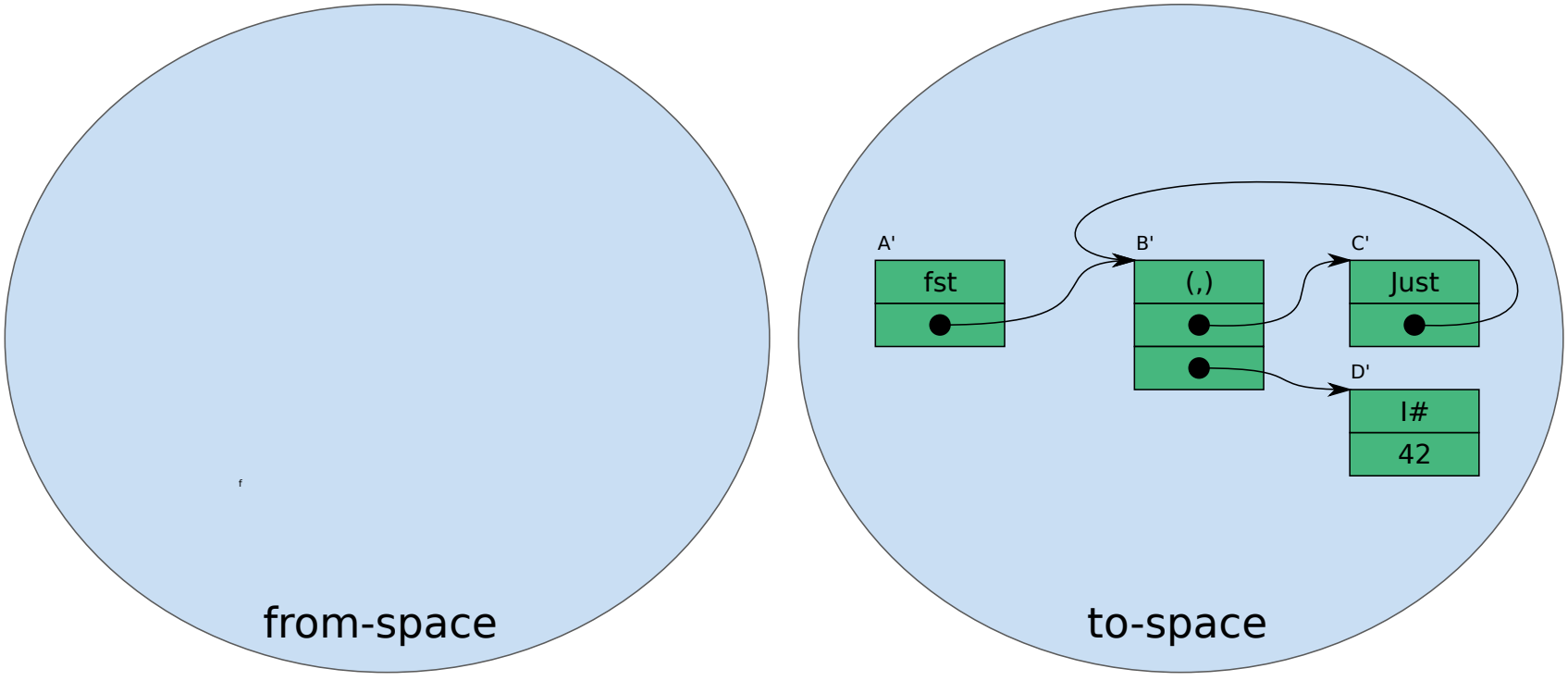
Moving garbage collection: Scavenging D



Moving garbage collection: Finished!



Moving garbage collection: Finished!



Moving garbage collection: Why it's great

- Allocation efficiency
- Lack of fragmentation
- Locality
- Simplicity
- Ease of parallelism
- Cost of collection is $O(\text{live data})$

Moving garbage collection: Why it's not so great

- Difficulty of incremental collection
- Even collecting a fraction of the heap requires that we must search the entire heap for references to moved objects

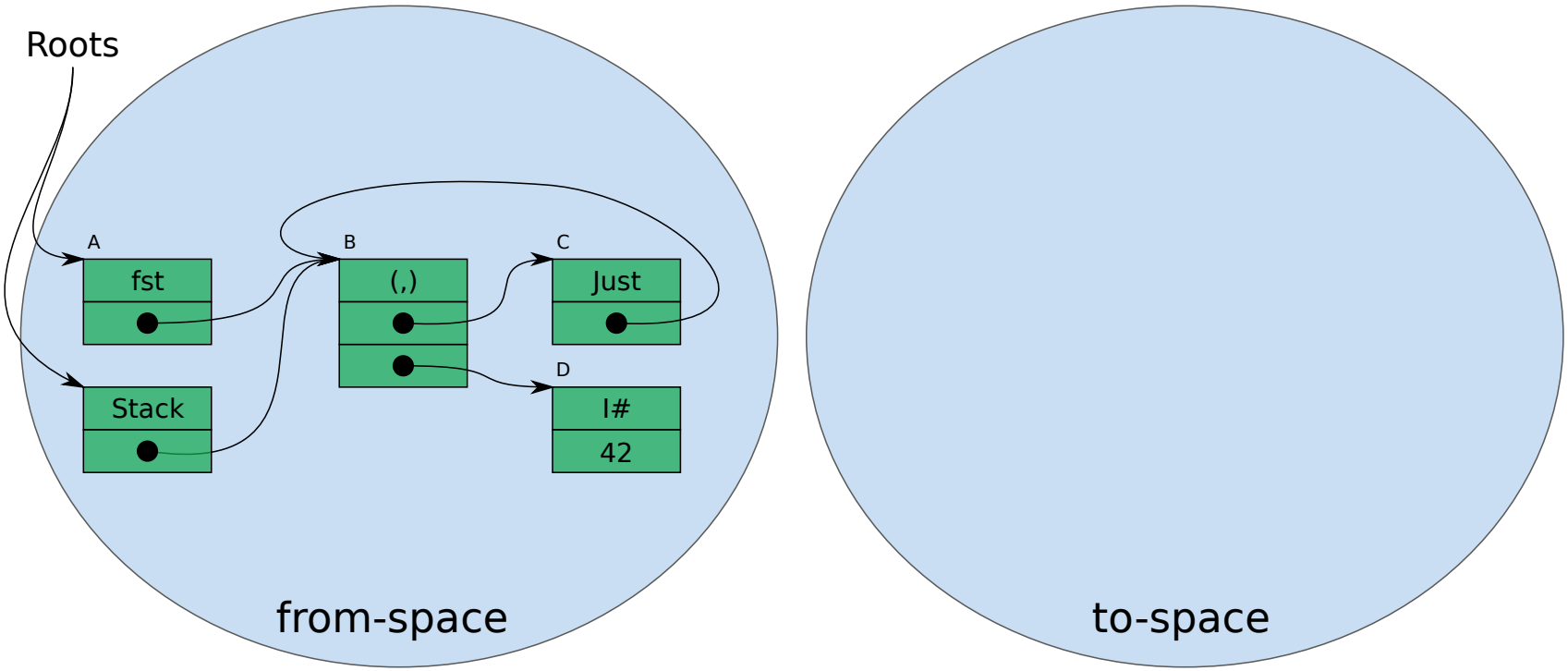
Low-latency collection

Two ways to reduce pause times:

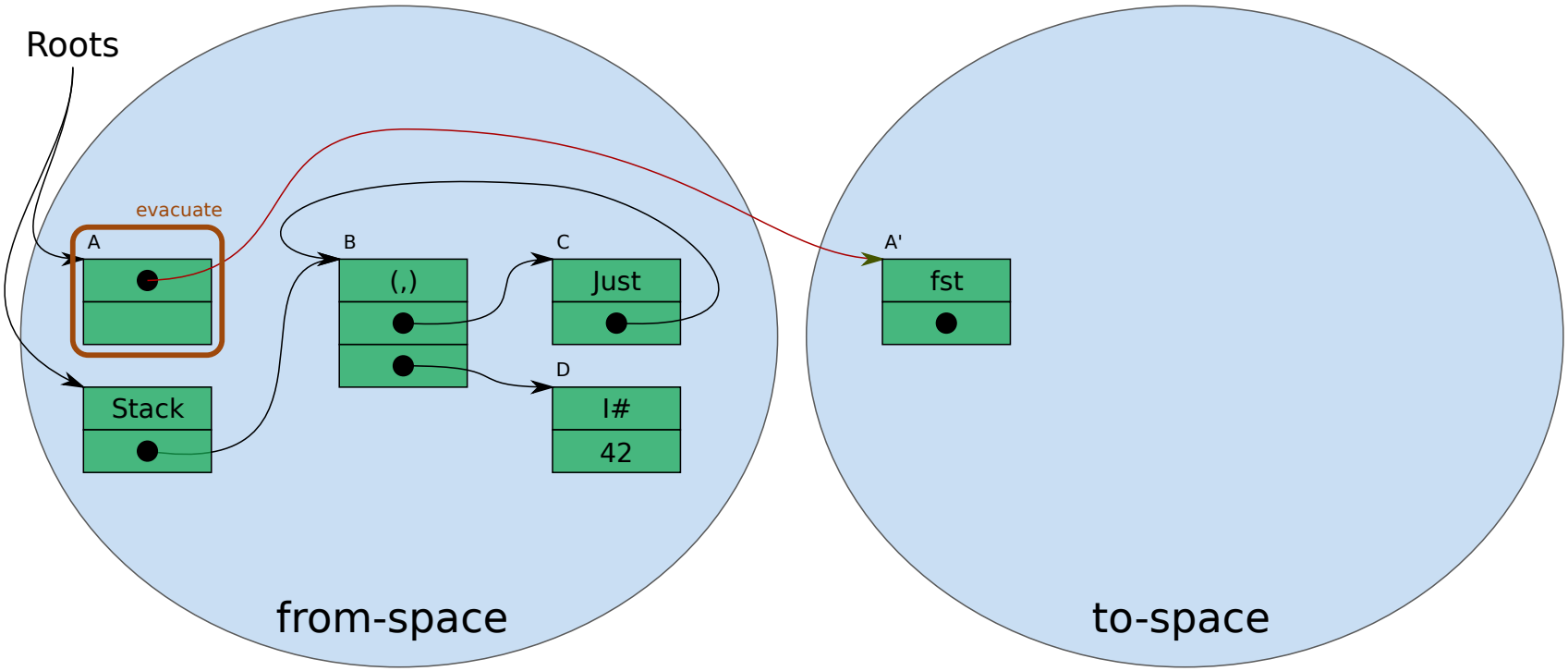
- incremental collection
  - Allow suspension of collection to allow mutator to run.
- concurrent collection
  - Allow collection to proceed while mutator runs.

We choose the latter.

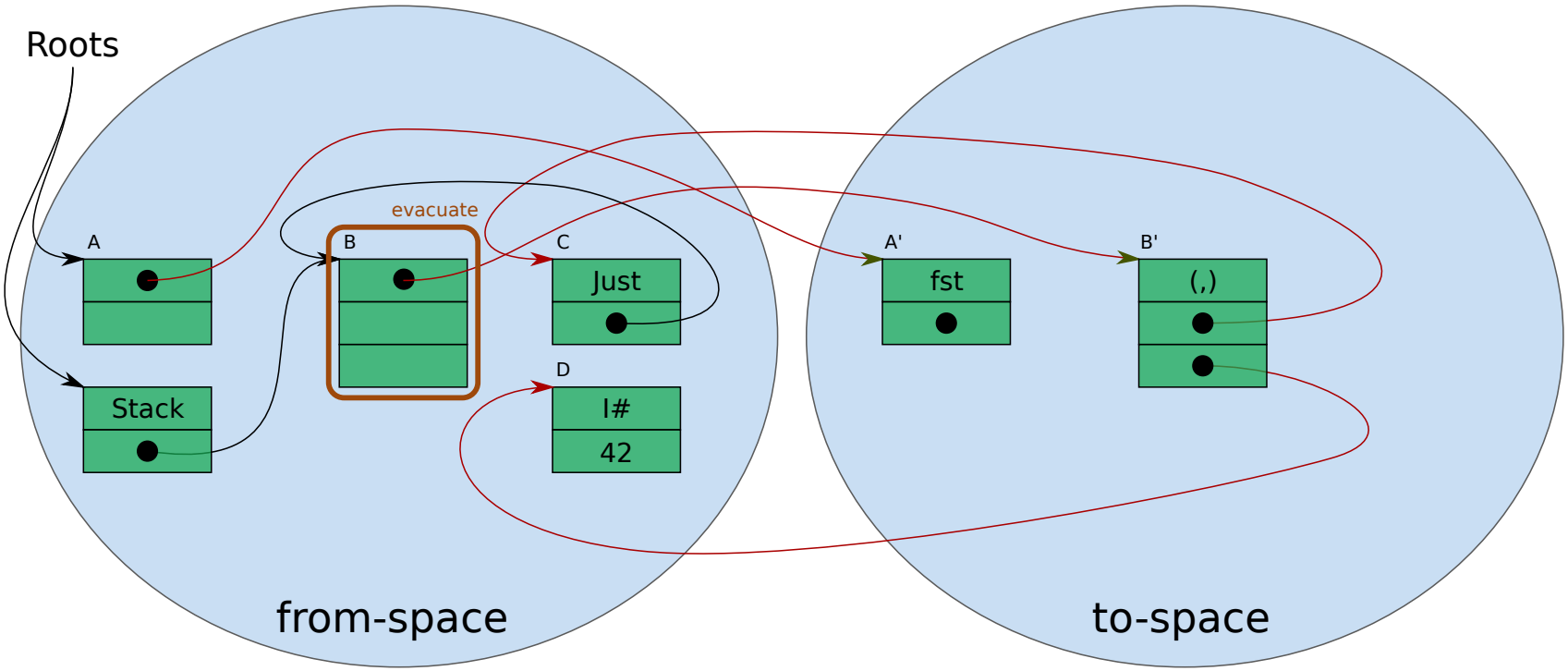
Incremental moving collection



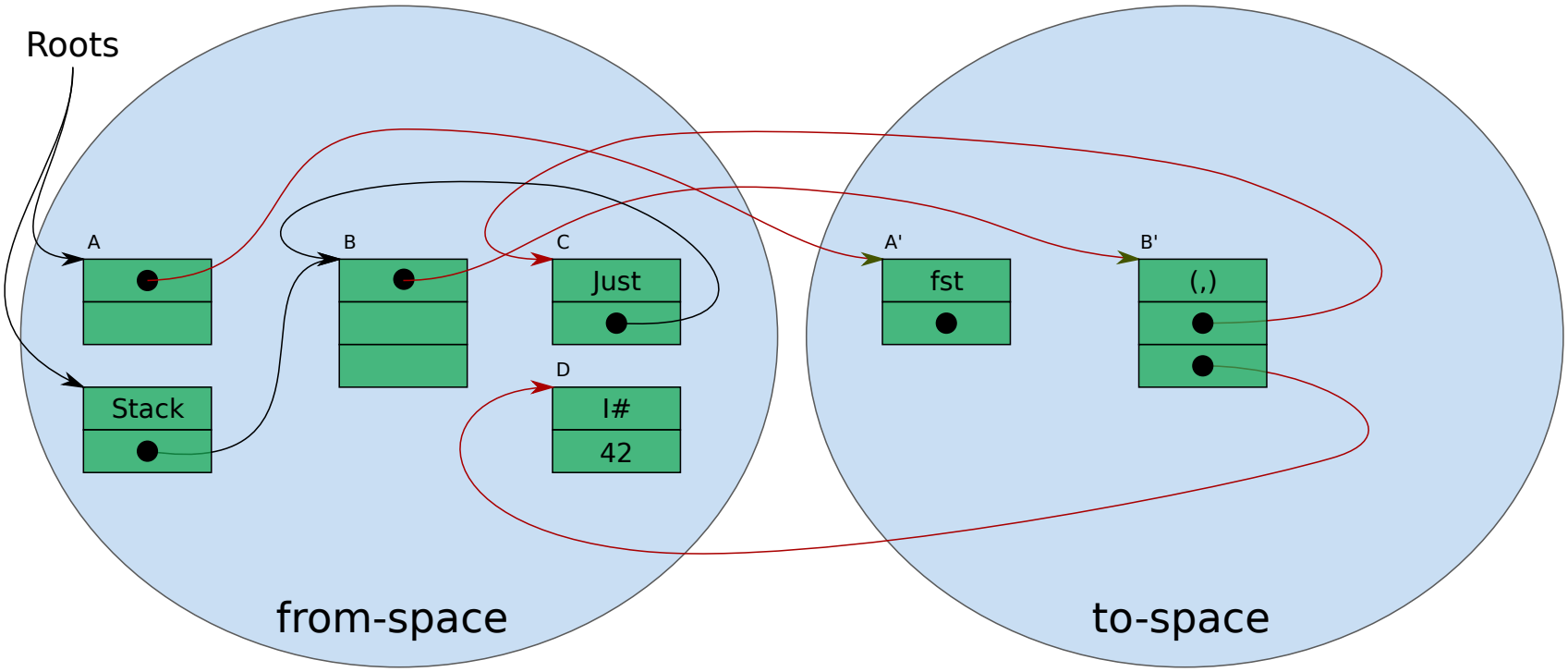
Incremental moving collection: Evacuate A



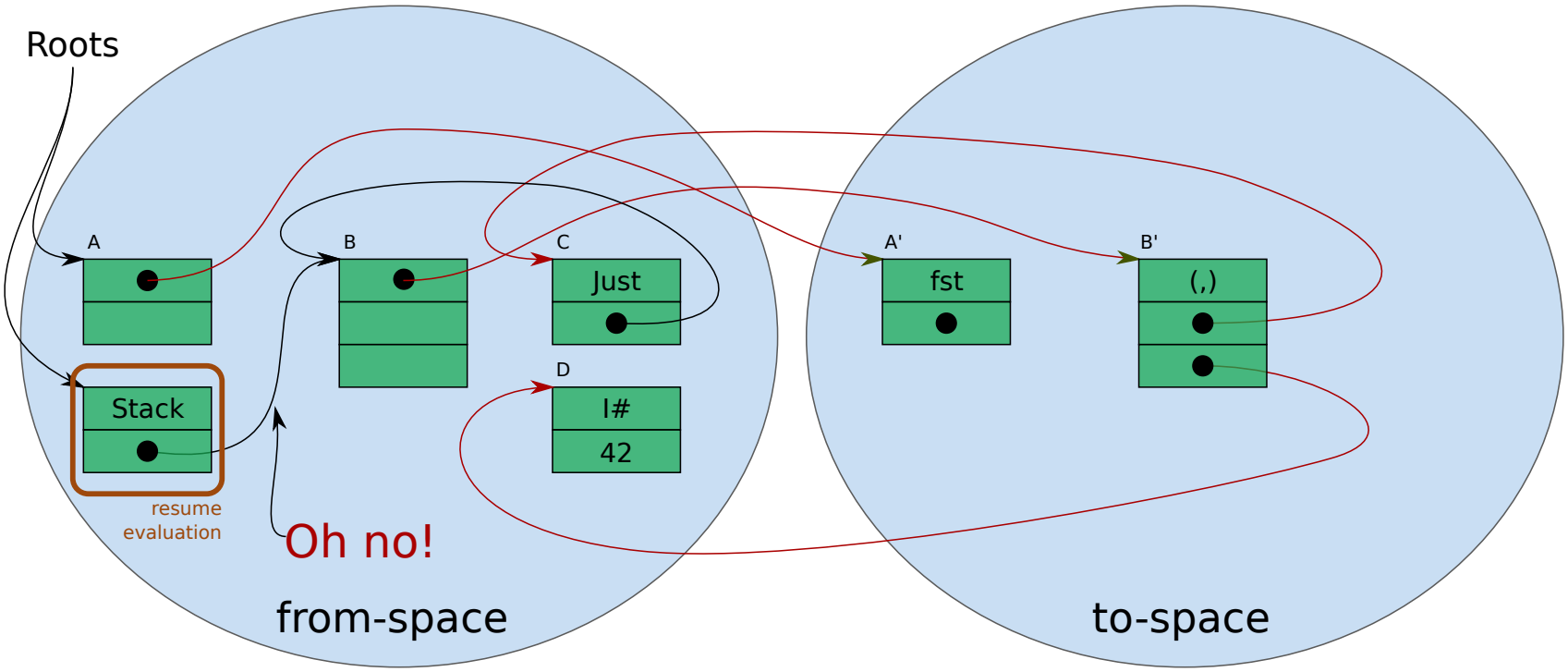
Incremental moving collection: Evacuate B



Incremental moving collection: Suspend collection



Incremental moving collection: Continue mutation



Making incremental moving collection safe

- Ensure that mutator never sees a reference into from-space
- One approach:
  - Have mutator check each pointer it dereferences (a *read barrier*)
  - Costly?

Non-Stop Haskell (2000)

Exploring the Barrier to Entry - Incremental Generational Garbage Collection for Haskell

A.M. Cheadle & A.J. Field
Imperial College London
{amc4,ajf}@doc.ic.ac.uk

S. Marlow & S.L. Peyton Jones
Microsoft Research, Cambridge
{simonmmar,simonpj}@microsoft.com

R.L. While
The University of Western Australia, Perth
lyndon@csse.uwa.edu.au

ABSTRACT

We document the design and implementation of a "production" incremental garbage collector for GHC 6.2. It builds on our earlier work (Non-stop Haskell) that exploited GHC's dynamic dispatch mechanism to hijack object code pointers so that objects in to-space automatically scavenge themselves when the mutator attempts to "enter" them. This paper details various optimisations based on code specialisation that remove the dynamic space, and associated time, overheads that accompanied our earlier scheme. We detail important implementation issues and provide a detailed evaluation of a range of design alternatives in comparison with Non-stop Haskell and GHC's current generational collector. We also show how the same code specialisation techniques can be used to eliminate the write barrier in a generational collector.

Eventually, however, the region(s) containing long-lived objects (the "older" generation(s)) will fill up and it will be necessary to perform a so-called *major* collection.

Major collections are typically expensive operations because the older generations are usually much larger than the young one. Furthermore, collecting an old generation requires the collection of all younger generations so, regardless of the actual number of generations, the entire heap will eventually require collection. Thus, although generational collection ensures a relatively small mean pause time, the pause time distribution has a "heavy tail" due to the infrequent, but expensive, major collections. This renders the technique unsuitable for applications that have real-time response requirements, for example certain interactive or real-time control systems.

The traditional way to reduce the variance of the pause times is to perform the garbage collection incrementally:

But: Pointer tagging (2007)

Faster laziness using dynamic pointer tagging

Simon Marlow
Microsoft Research
simonmar@microsoft.com

Alexey Rodriguez Yakushev
University of Utrecht, The Netherlands
alexey@cs.uu.nl

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Abstract

In the light of evidence that Haskell programs compiled by GHC exhibit large numbers of mispredicted branches on modern processors, we re-examine the "tagless" aspect of the STG-machine that GHC uses as its evaluation model.

We propose two tagging strategies: a simple strategy called semi-tagging that seeks to avoid one common source of unpredictable indirect jumps, and a more complex strategy called dynamic pointer-tagging that uses the spare low bits in a pointer to encode information about the pointed-to object. Both of these strategies have been implemented and exhaustively measured in the context of a production compiler, GHC, and the paper contains detailed descriptions of the implementations. Our measurements demonstrate significant performance improvements (14% for dynamic pointer-tagging with only a 2% increase in code size), and we further demonstrate that much of the improvement can be attributed to the elimination of mispredicted branch instructions.

As part of our investigations we also discovered that one optimisation in the STG-machine, vectored-returns, is no longer worthwhile and we explain why.

1. Introduction

continuation to compute `a+y`, and jumps to the *entry code* for `x`. The first field of every heap closure is its entry code, and jumping to this code is called *entering* the closure. The entry code for an unevaluated closure will evaluate itself and return the value to the continuation; an already-evaluated closure will return immediately.

This scheme is attractive because the code to evaluate a closure is simple and uniform: any closure can be evaluated simply by entering it. But this uniformity comes at the expense of performing indirect jumps, one to enter the closure and another to return to the evaluation site. These indirect jumps are particularly expensive on a modern processor architecture, because they fox the branch-prediction hardware, leading to a stall of 10 or more cycles depending on the length of the pipeline.

If the closure is unevaluated, then we really do have to take an indirect jump to its entry code. However, if it happens to be evaluated already, then a conditional jump might execute much faster. In this paper we describe a number of approaches that exploit this possibility. We have implemented these schemes and show that they deliver substantial performance improvements (10-14%) in GHC, a mature optimising compiler for Haskell. Specifically, our contributions are these:

- We give the first accurate measurements for the dynamic behaviour of `case` expressions in lazy programs, across a range

Pointer tagging by the numbers (2007)

	L1 D-cache		L2 D-cache
Program	accesses	misses	misses
anna	-23.4	-13.3	-28.4
cacheprof	-15.9	-5.8	-8.1
constraints	-12.8	-4.8	-5.4
fulsom	-8.9	-17.2	-53.3
integrate	-7.7	-2.2	+18.6
mandel	-8.4	-1.9	-29.4
simple	-11.8	-3.9	-4.1
sphere	-13.0	-19.5	-73.2
typecheck	-20.4	-8.5	-30.8
wang	-13.4	-2.0	-10.4
(81 more)	...	...	
Min	-31.5	-19.5	-73.2
Max	+0.6	+0.5	+81.6
Geometric Mean	-13.9	-4.5	-20.1

Figure 12. Cache behaviour for pointer-tagging vs. the baseline

Making incremental moving collection safe

Another approach (Ben-Yitzhak 2002)

- Split heap into n chunks
- Track references between chunks
  - Requires cooperation of mutator
- Evacuate/scavenge one chunk at a time
- Scavenge inter-chunk references
- Reduces pause by factor of $1/n$

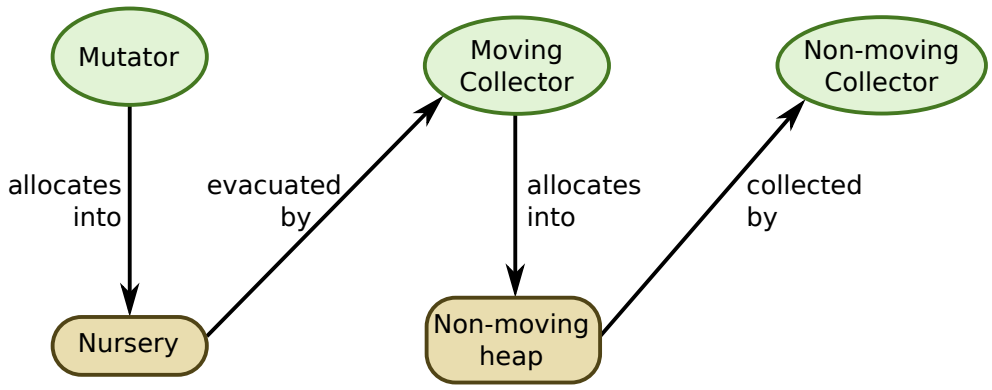
Making incremental moving collection safe

- Moving collection invalidates object references
  - Complicates incremental collection
- Perhaps we can avoid it?

Making incremental moving collection safe

- Moving collection invalidates object references
  - Complicates incremental collection
- Perhaps we can avoid it?
- Yes, we can.
- But can we get the best of worlds?

Hybrid moving/non-moving collector



- Mutator allocates into a moving nursery
- Young generation collector evacuates into non-moving heap
- Non-moving heap collected with mark & sweep
- Mark & sweep amenable to both incremental and concurrent collection

Plan for rest of talk

I will describe our concurrent, moving/non-moving hybrid collector implemented in GHC.

Plan for rest of talk

I will describe our concurrent, moving/non-moving hybrid collector implemented in GHC.

Background	What is a mark & sweep garbage collector?
Snapshotting	How do we safely trace with concurrent mutation?
Allocator	How to provide fresh memory for the minor GC?
Collector	How to find dead objects and free them?
Status	Where are we today?

Mark & Sweep Garbage Collection

Necessary State

Heap A mark flag per object.

Collector A mark queue of references to objects.

Mark & Sweep Garbage Collection

Necessary State

Heap A mark flag per object.

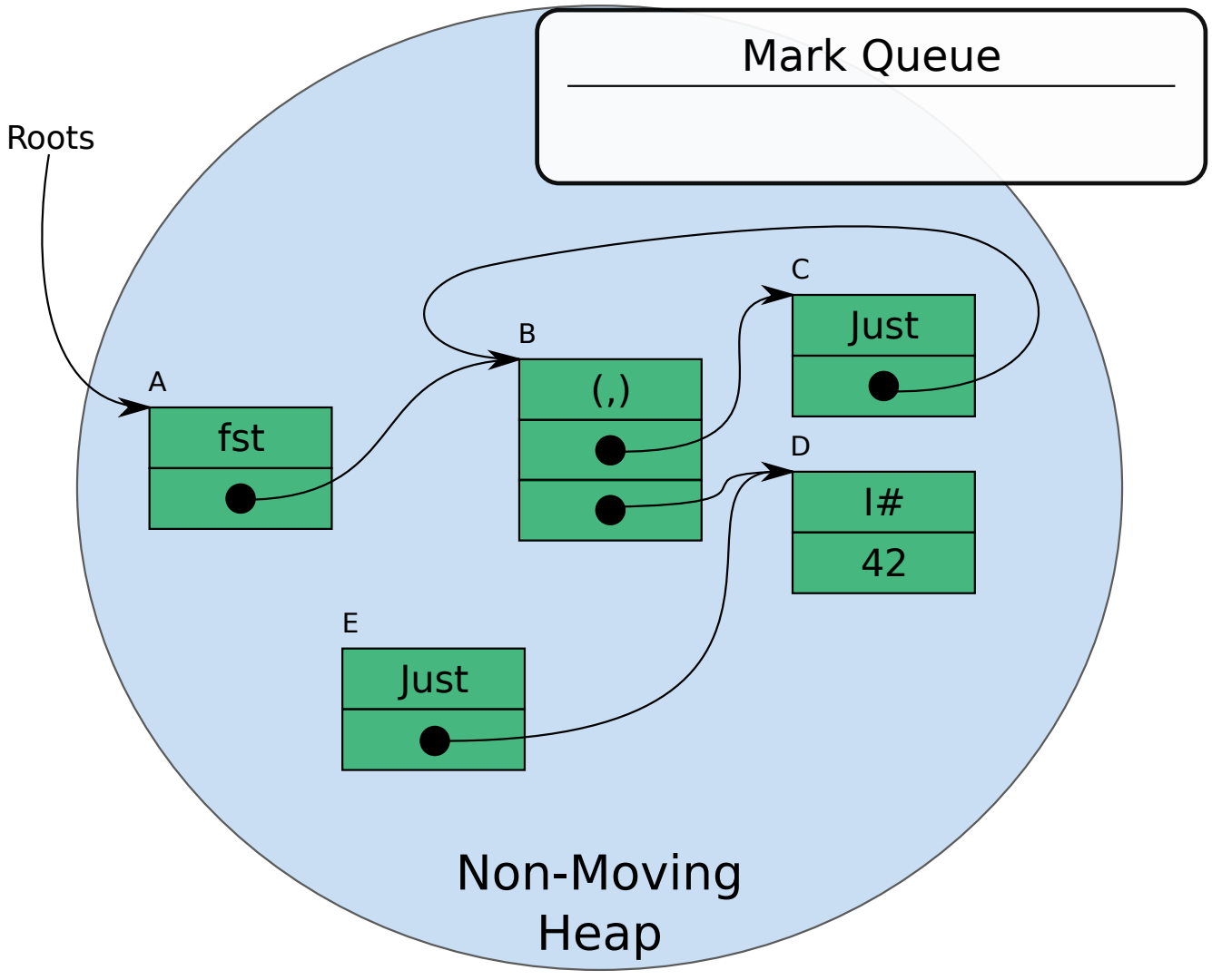
Collector A mark queue of references to objects.

Operations

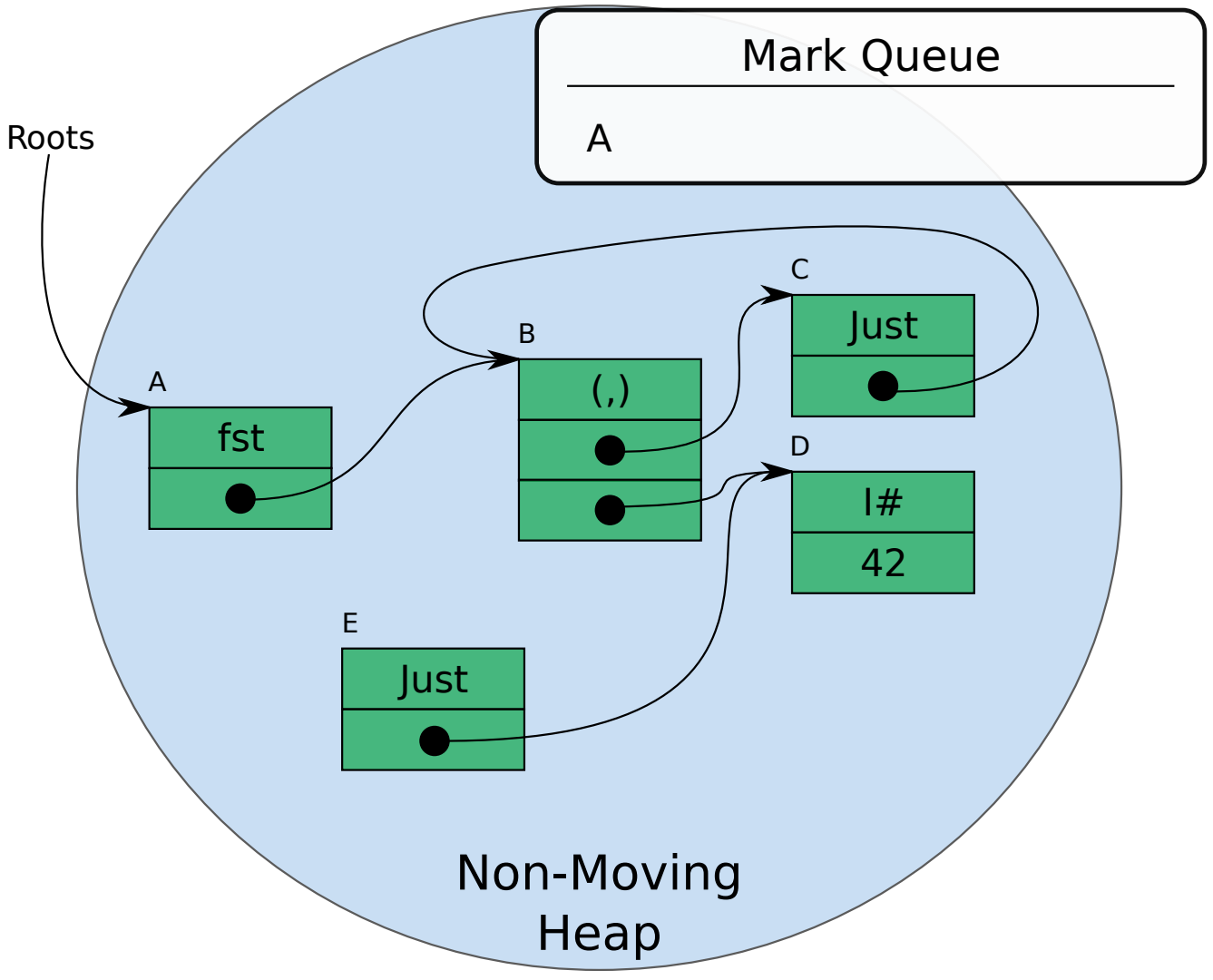
Mark Push an object's pointers to the mark queue, set its mark flag.

Sweep Walk all heap objects, freeing any with un-set mark flags.

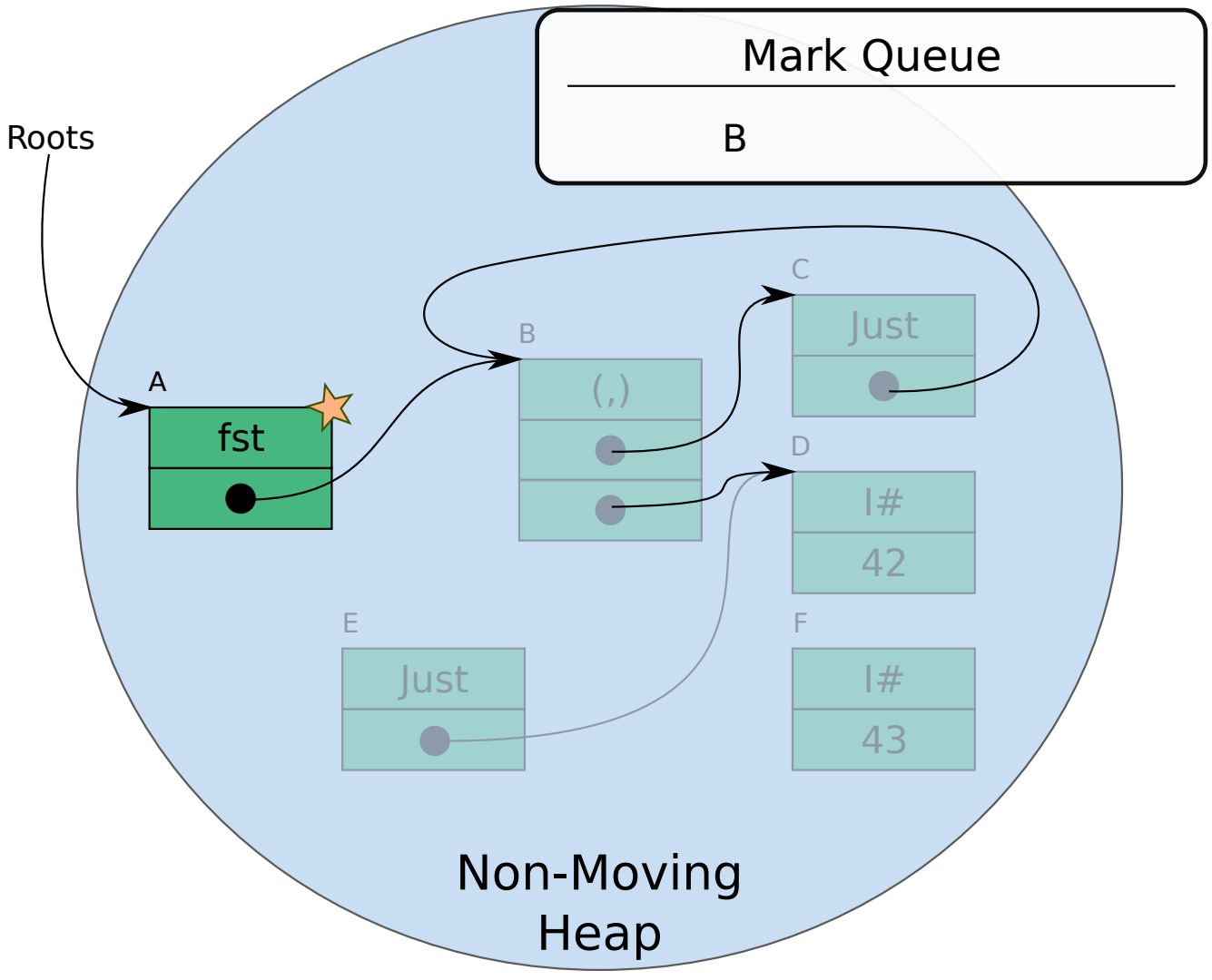
Mark & Sweep Garbage Collection



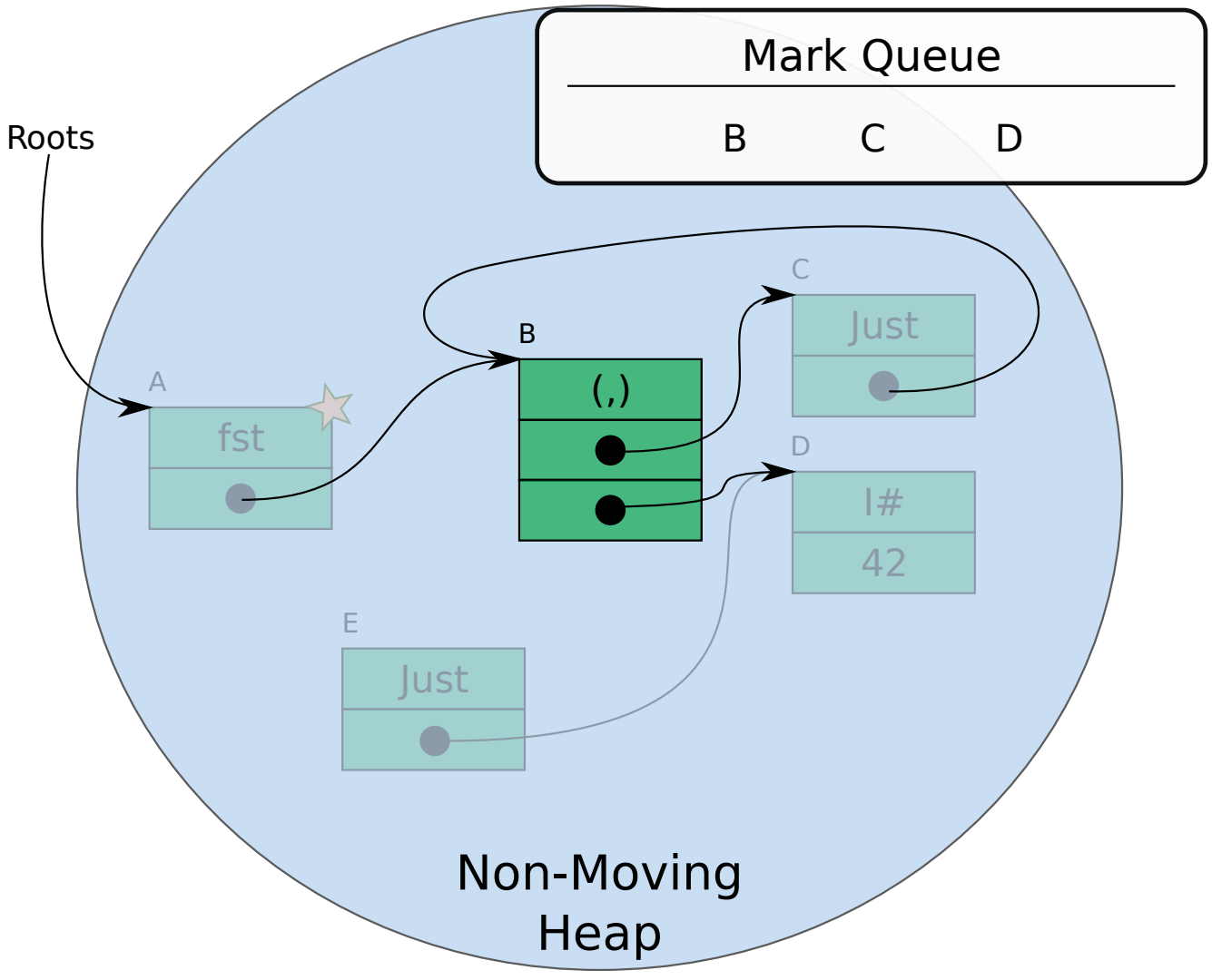
Mark & Sweep Garbage Collection: Collect roots



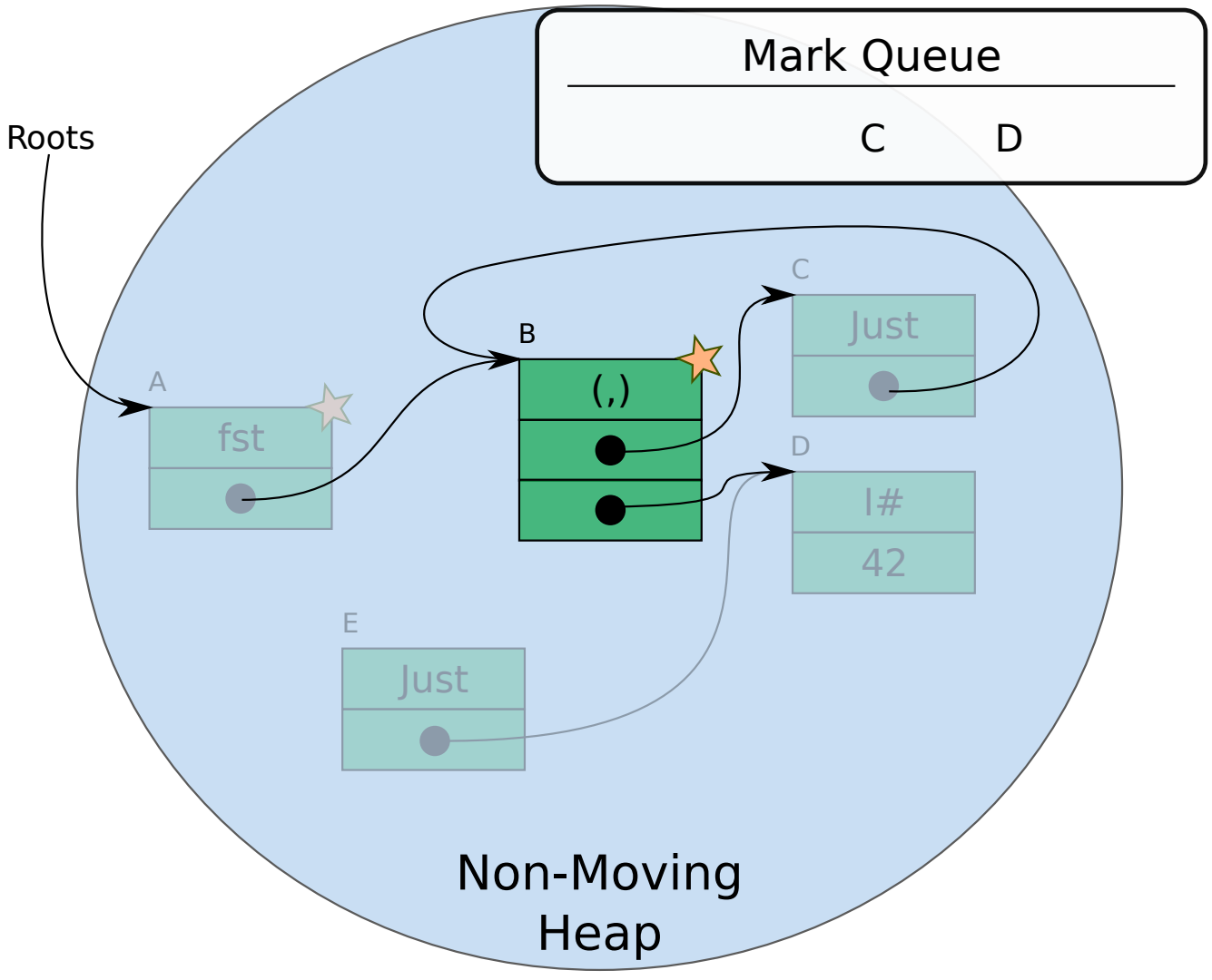
Mark & Sweep Garbage Collection: Marking (A)



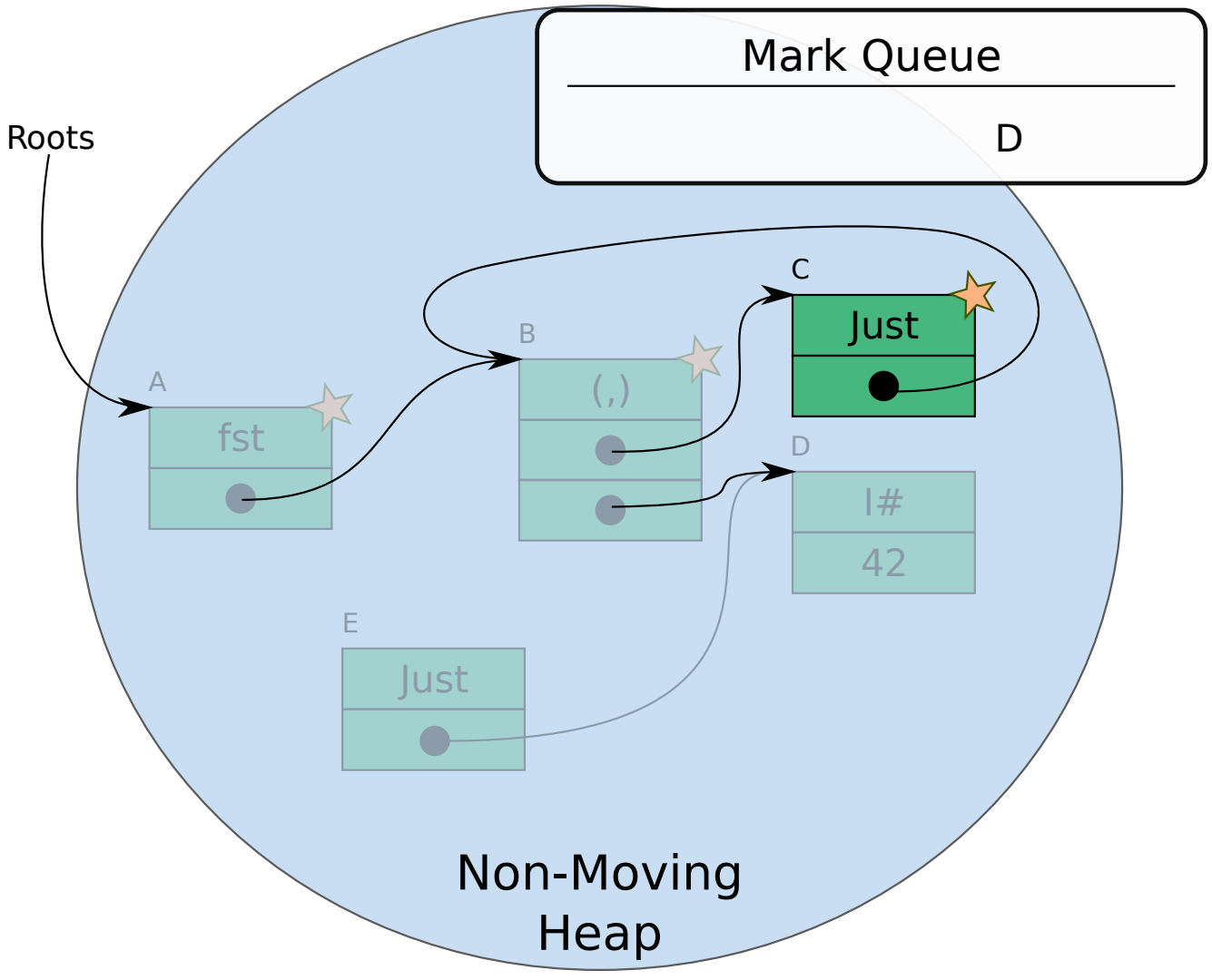
Mark & Sweep Garbage Collection: Marking (B)



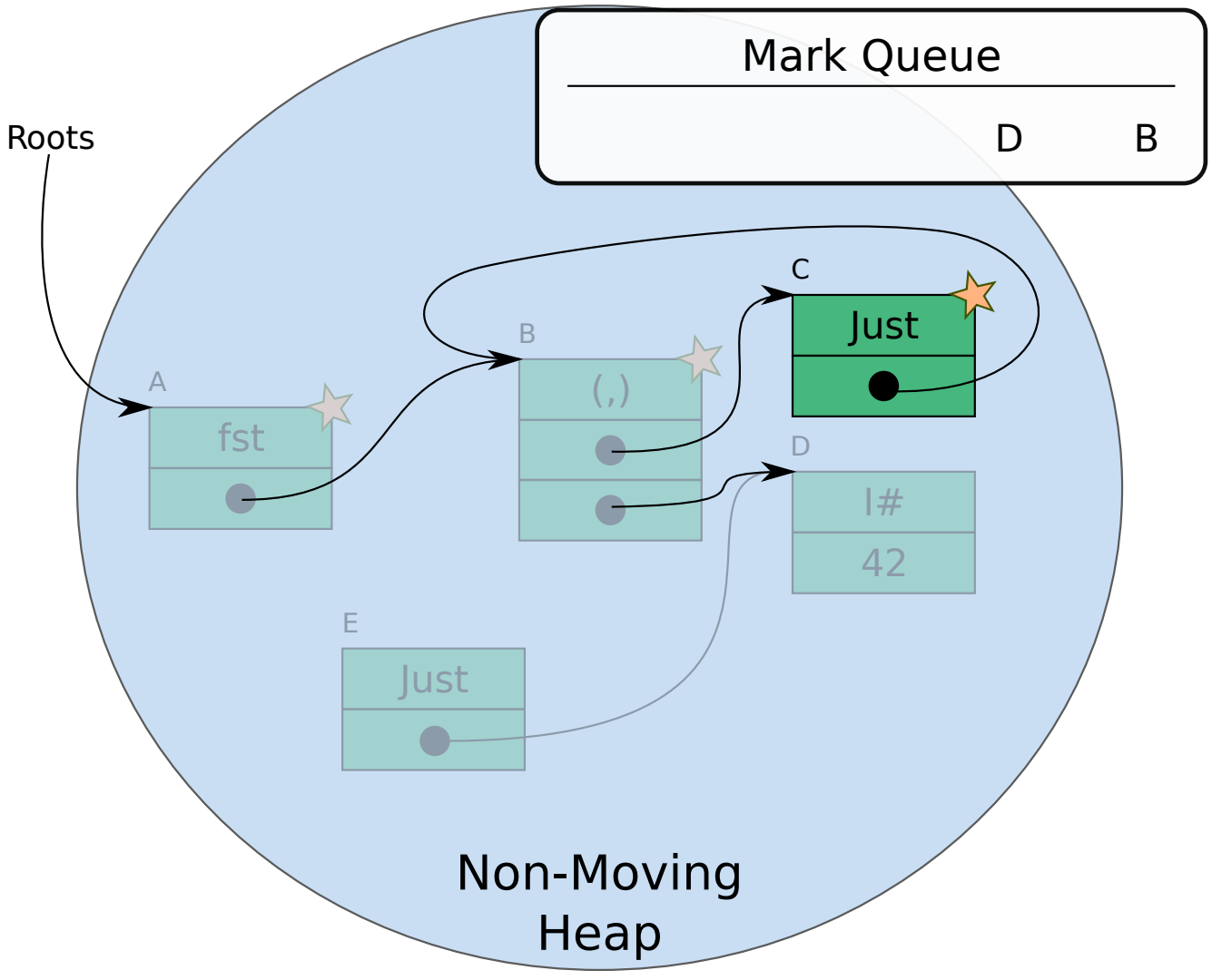
Mark & Sweep Garbage Collection: Marking (B)



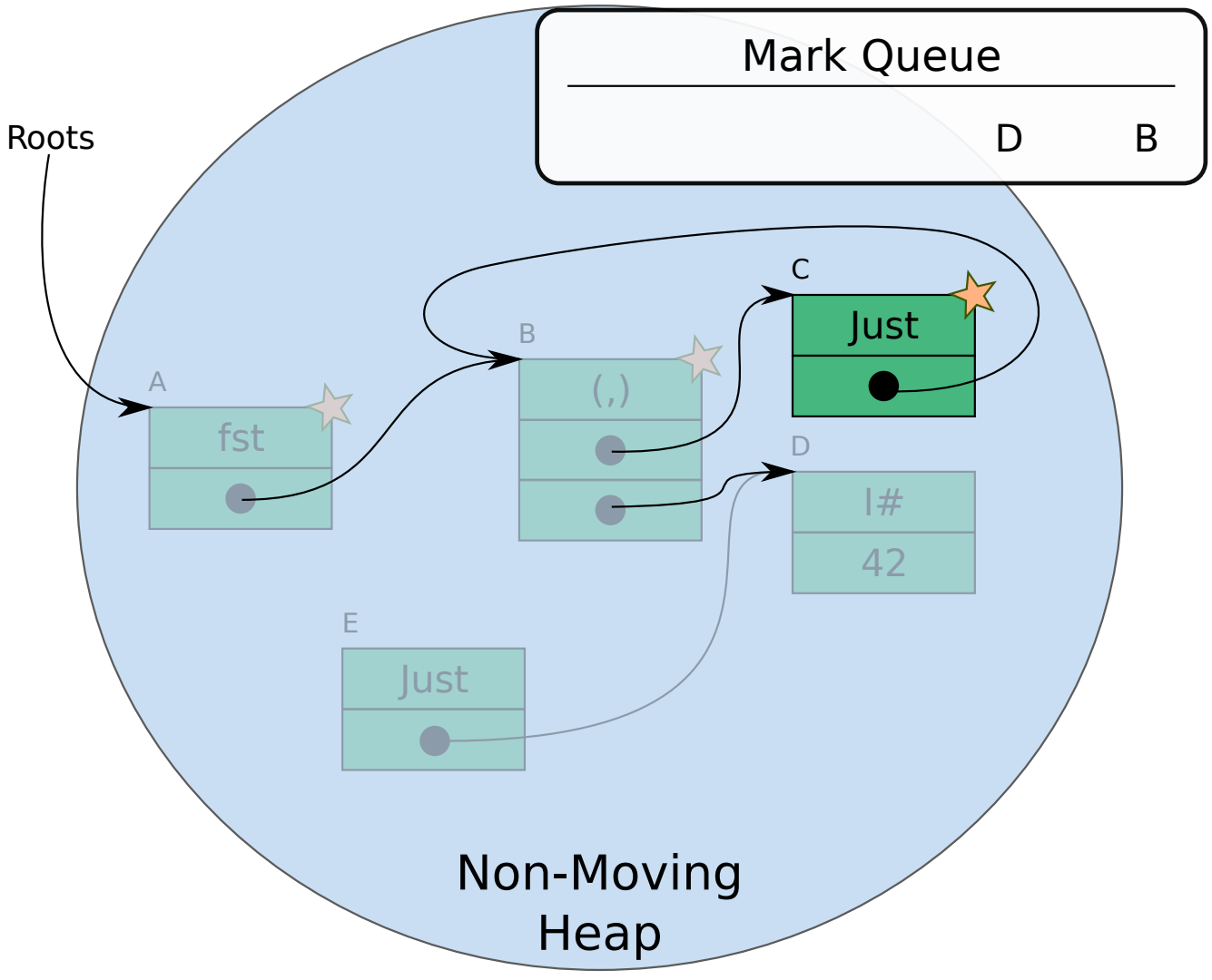
Mark & Sweep Garbage Collection: Marking (C)



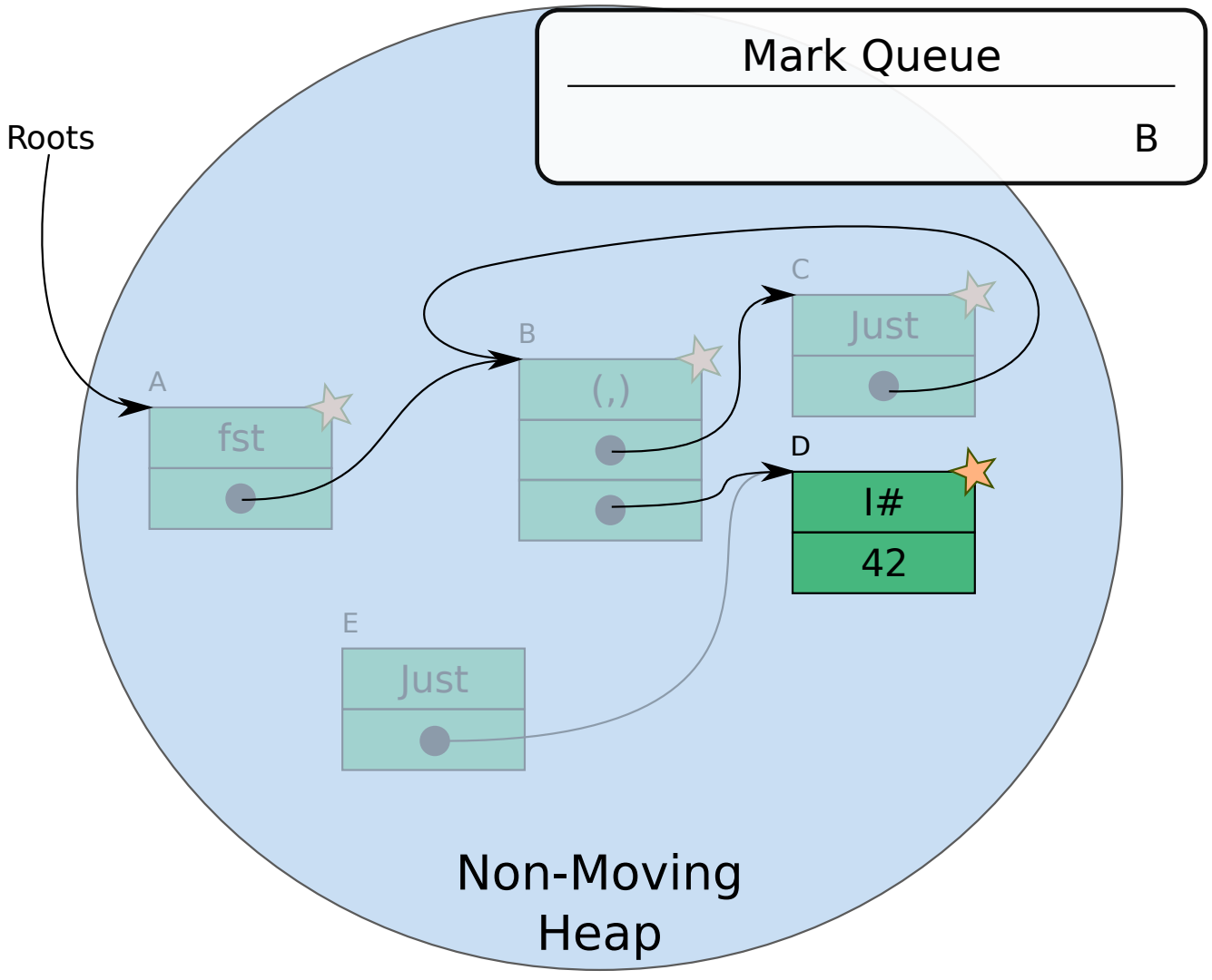
Mark & Sweep Garbage Collection: Marking (C)



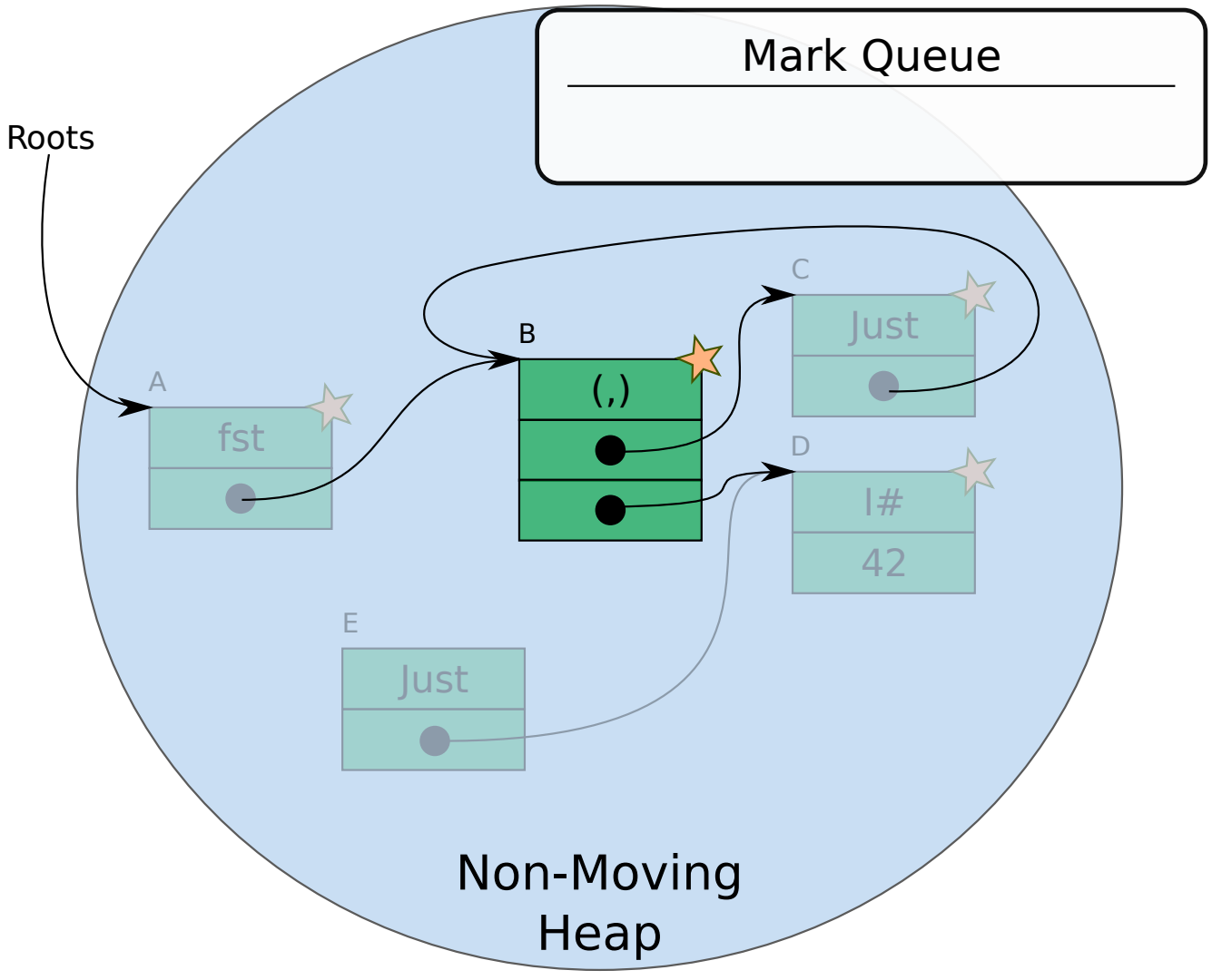
Mark & Sweep Garbage Collection: Marking (C)



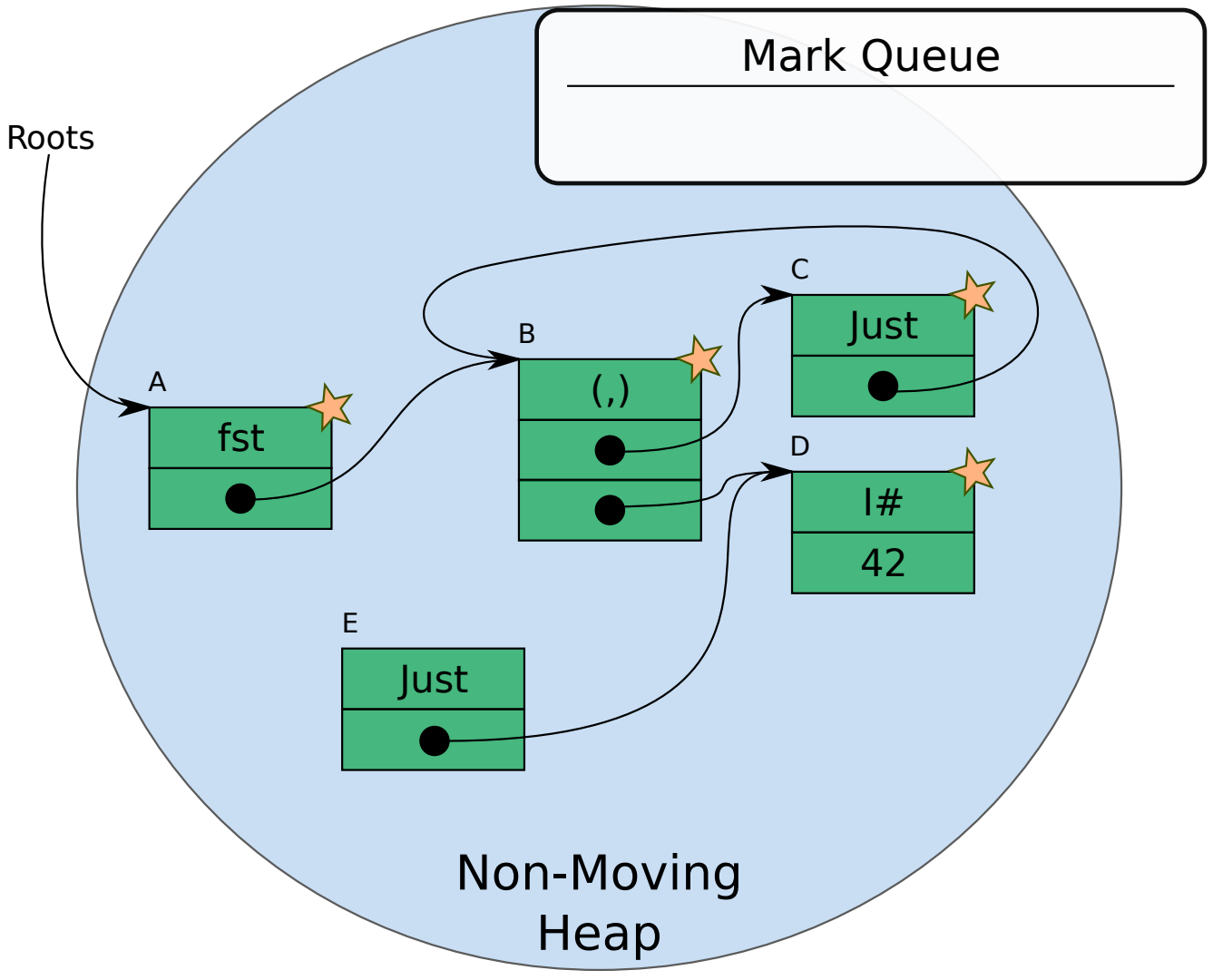
Mark & Sweep Garbage Collection: Marking (D)



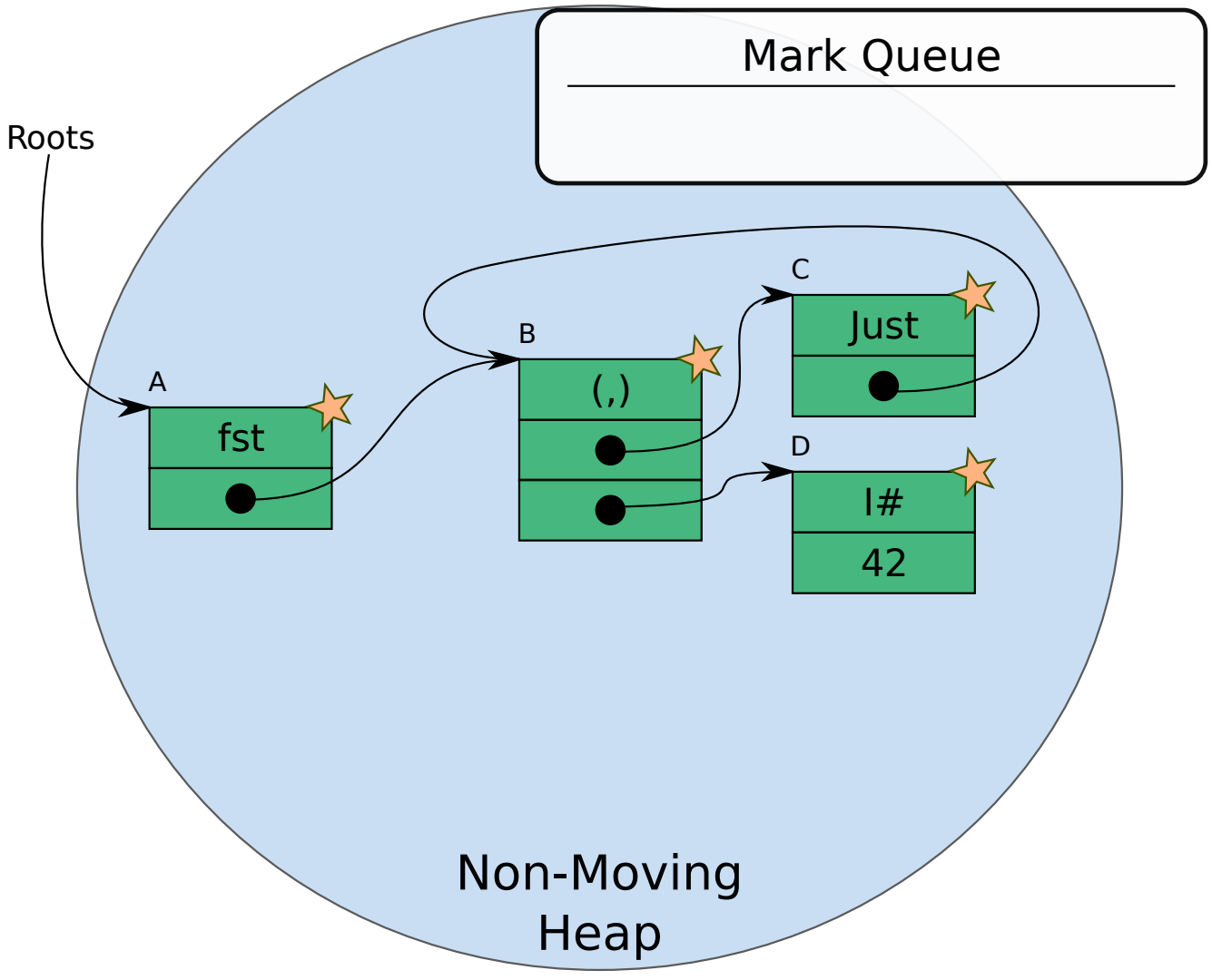
Mark & Sweep Garbage Collection: Marking (B)



Mark & Sweep Garbage Collection: Marking



Mark & Sweep Garbage Collection: Sweep

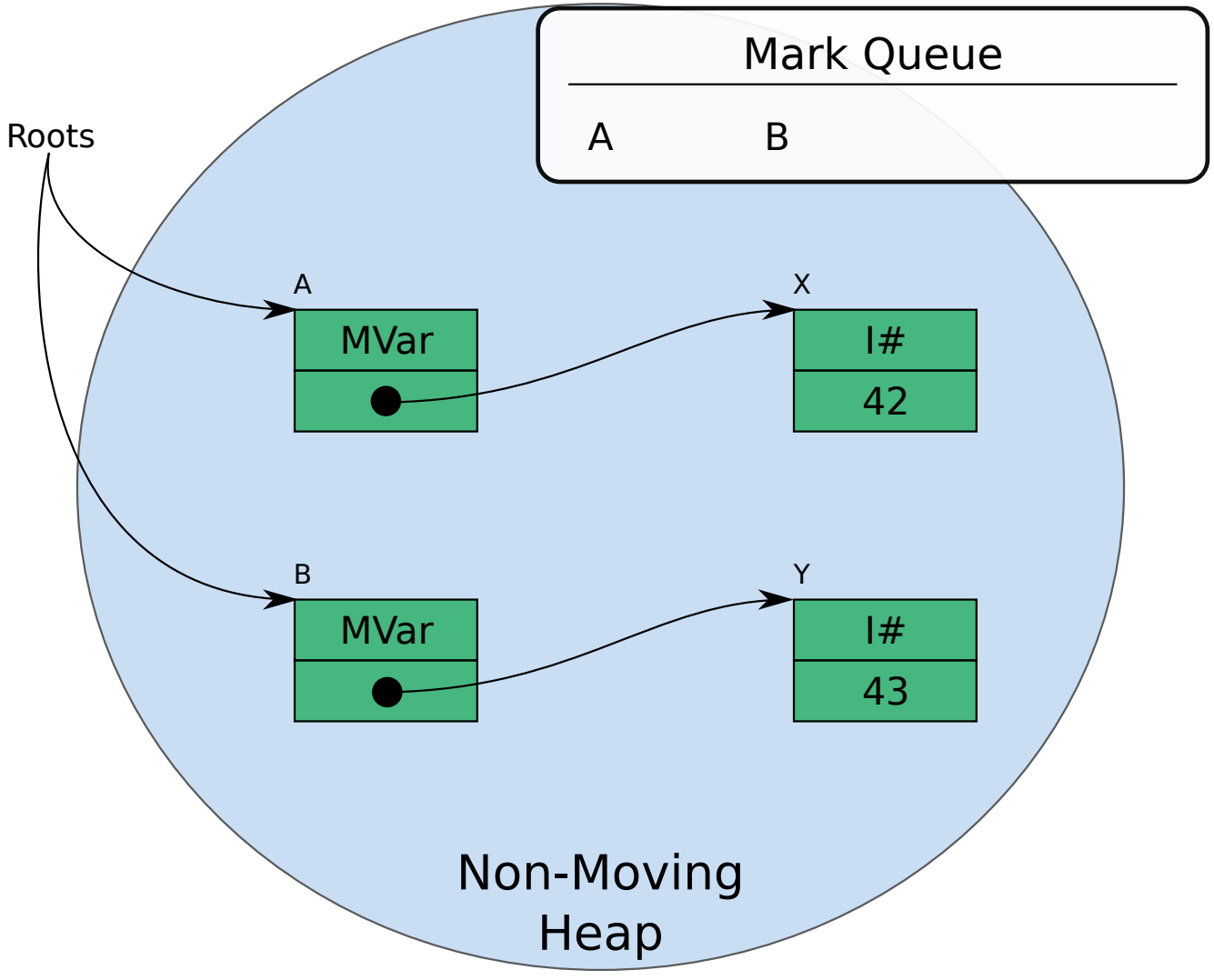


Concurrency: The problem of mutation

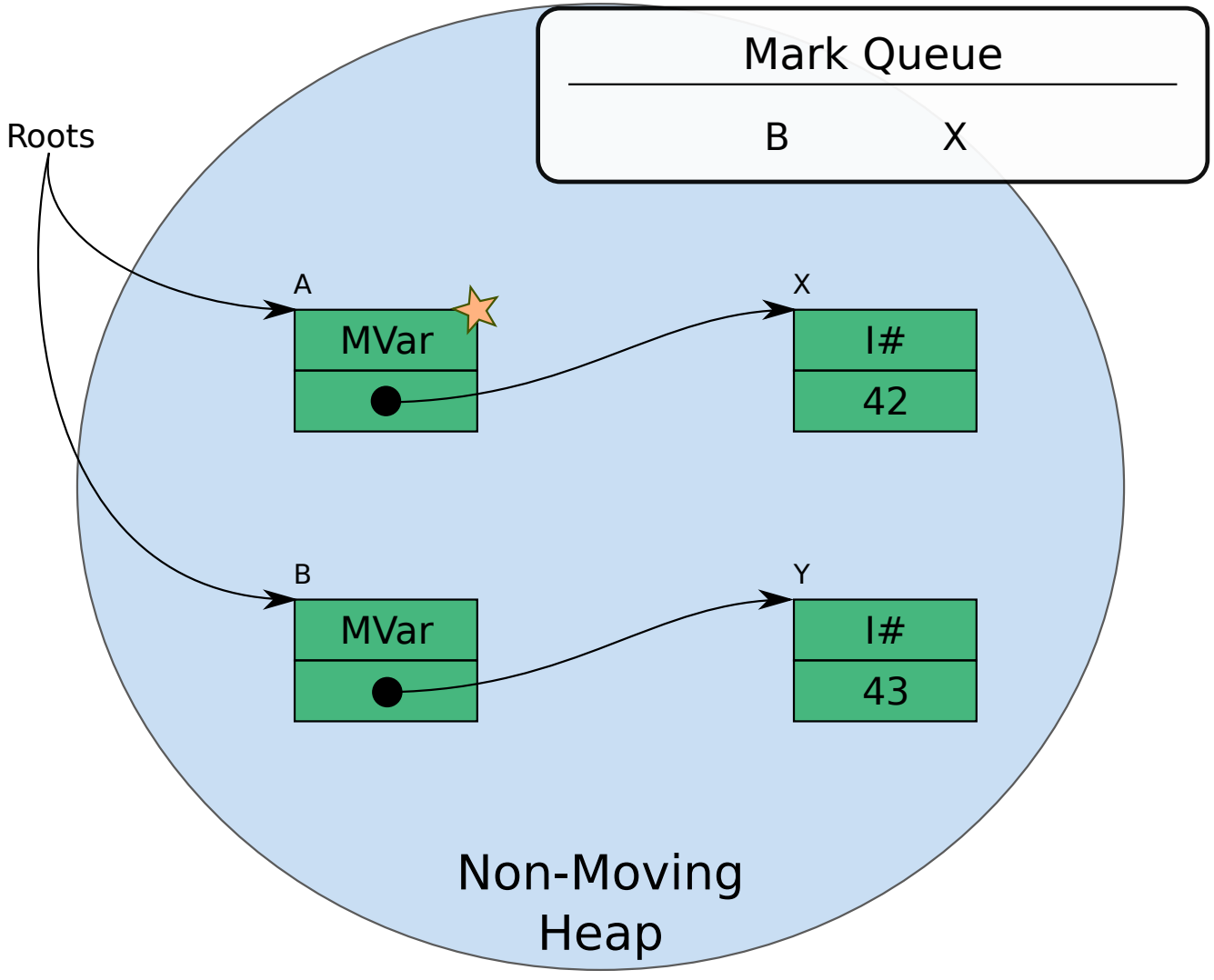
While non-moving collection does not invalidate references, this alone is not sufficient for safe concurrent mutation.

Let's see why...

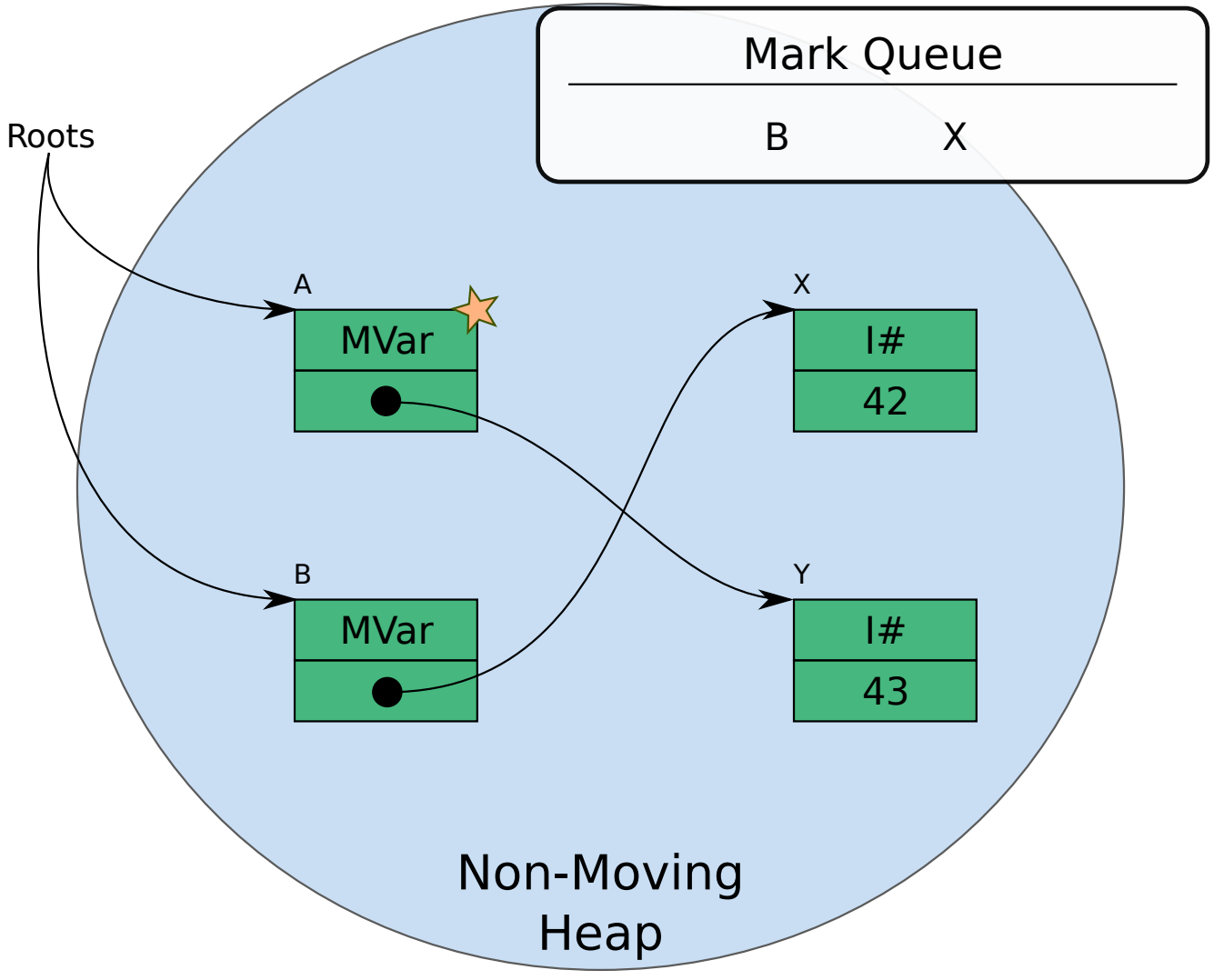
The problem of mutation



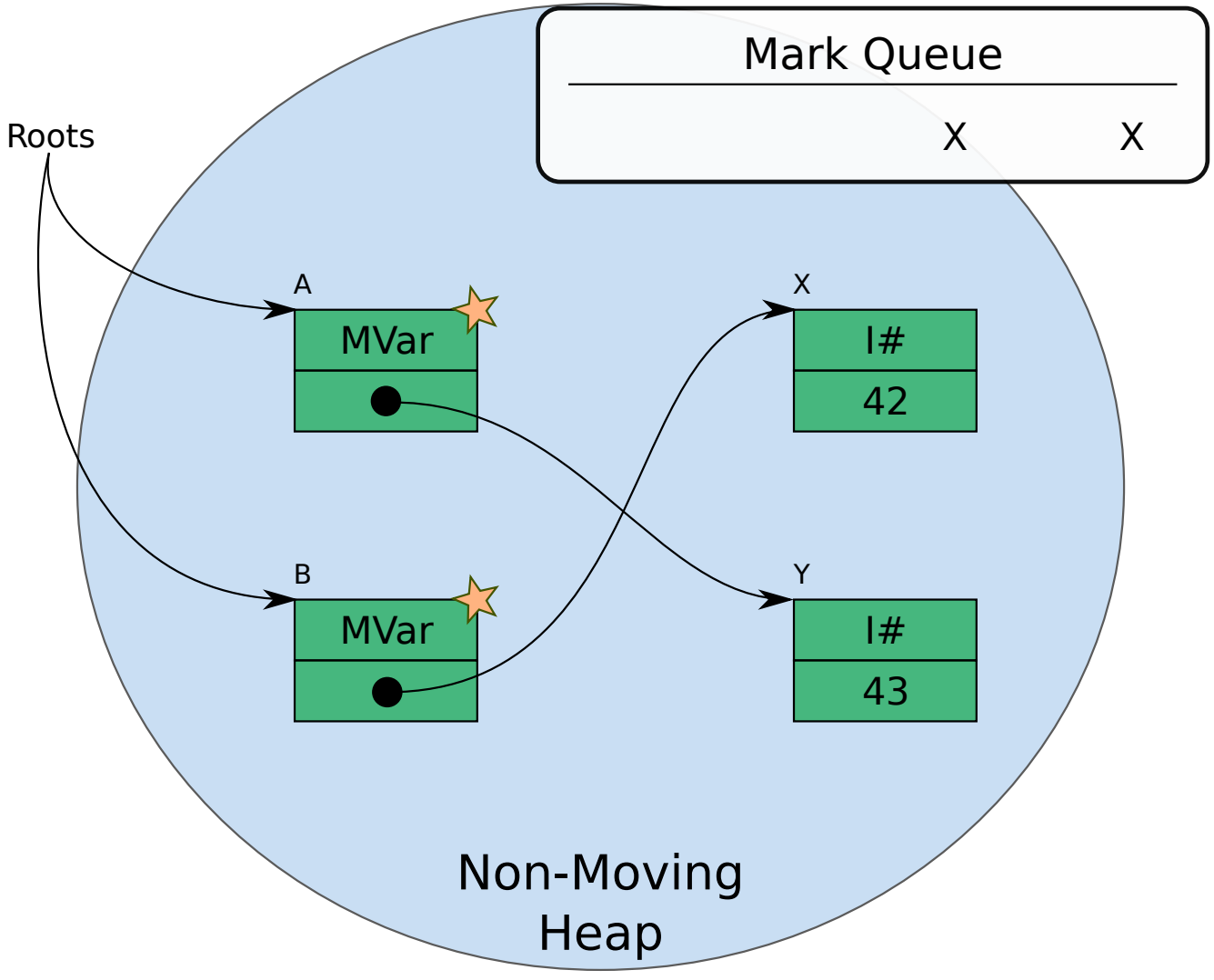
The problem of mutation



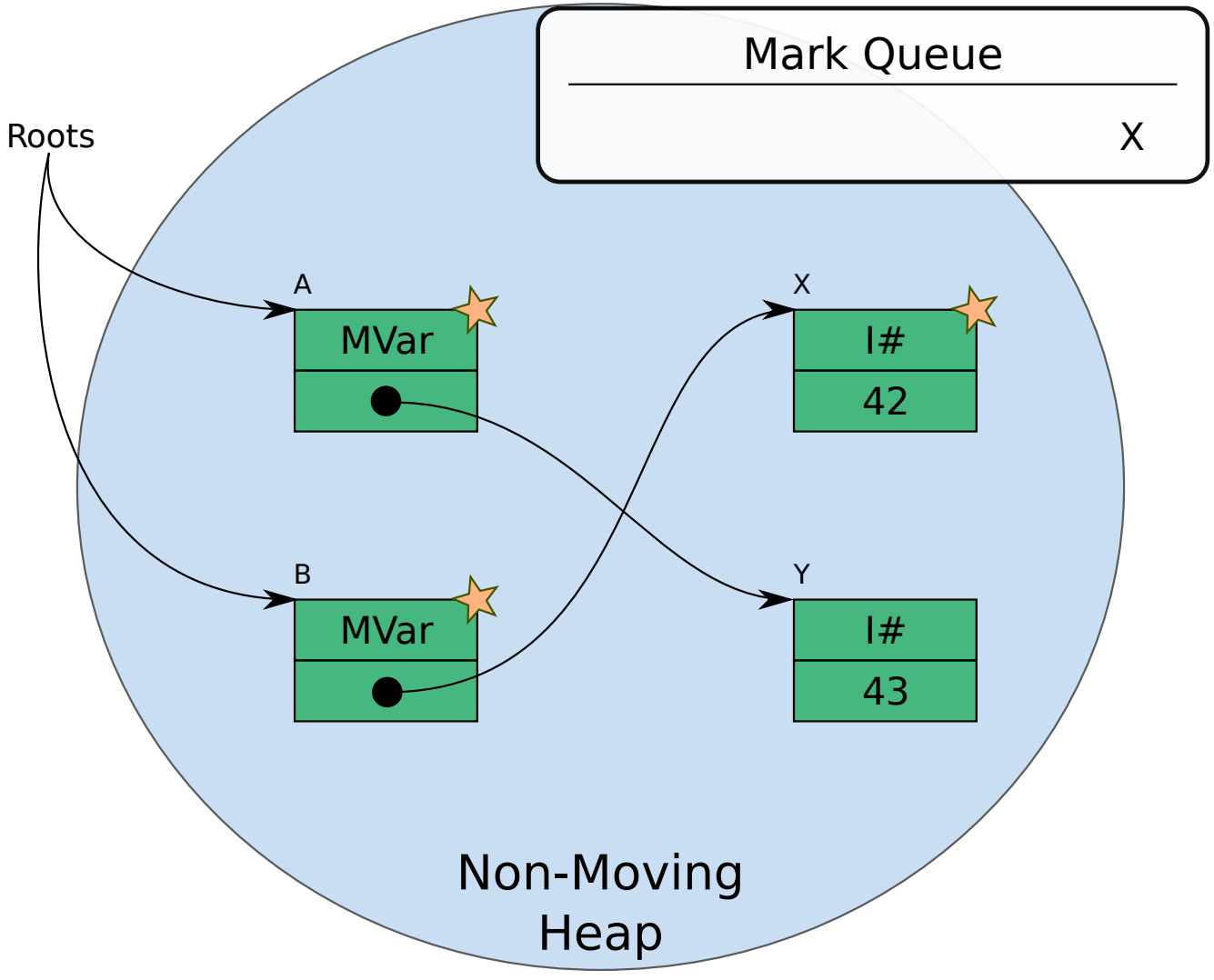
The problem of mutation



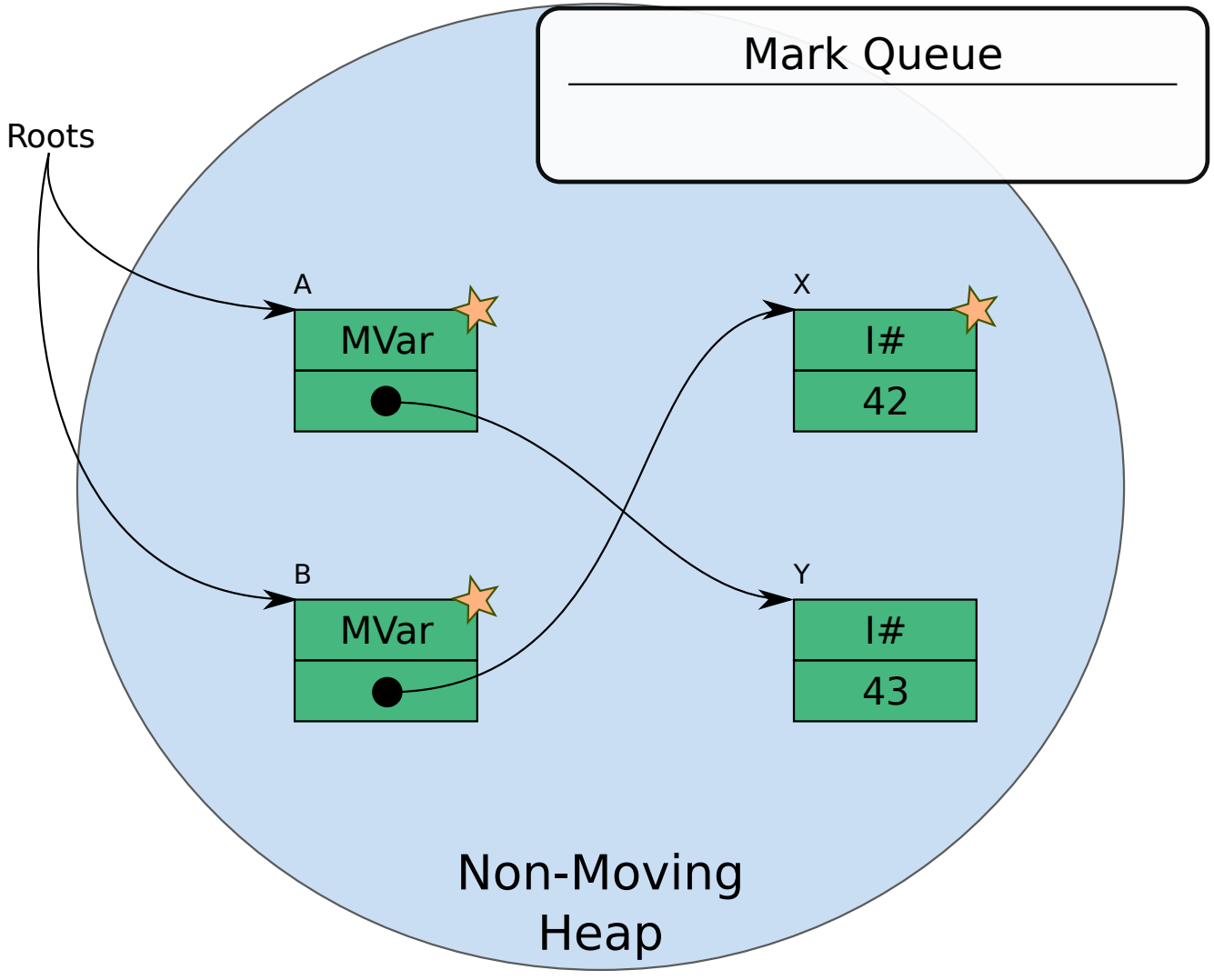
The problem of mutation



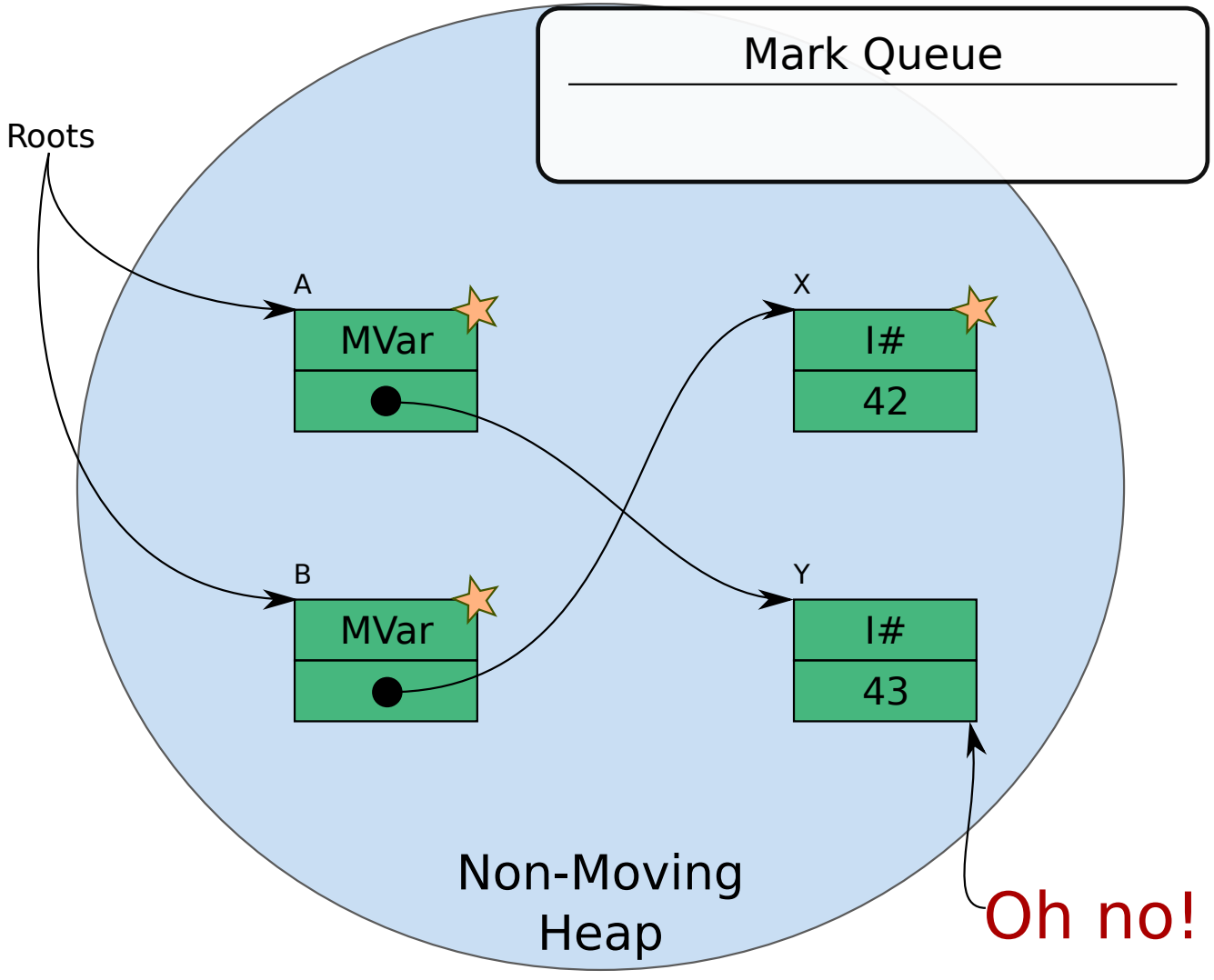
The problem of mutation



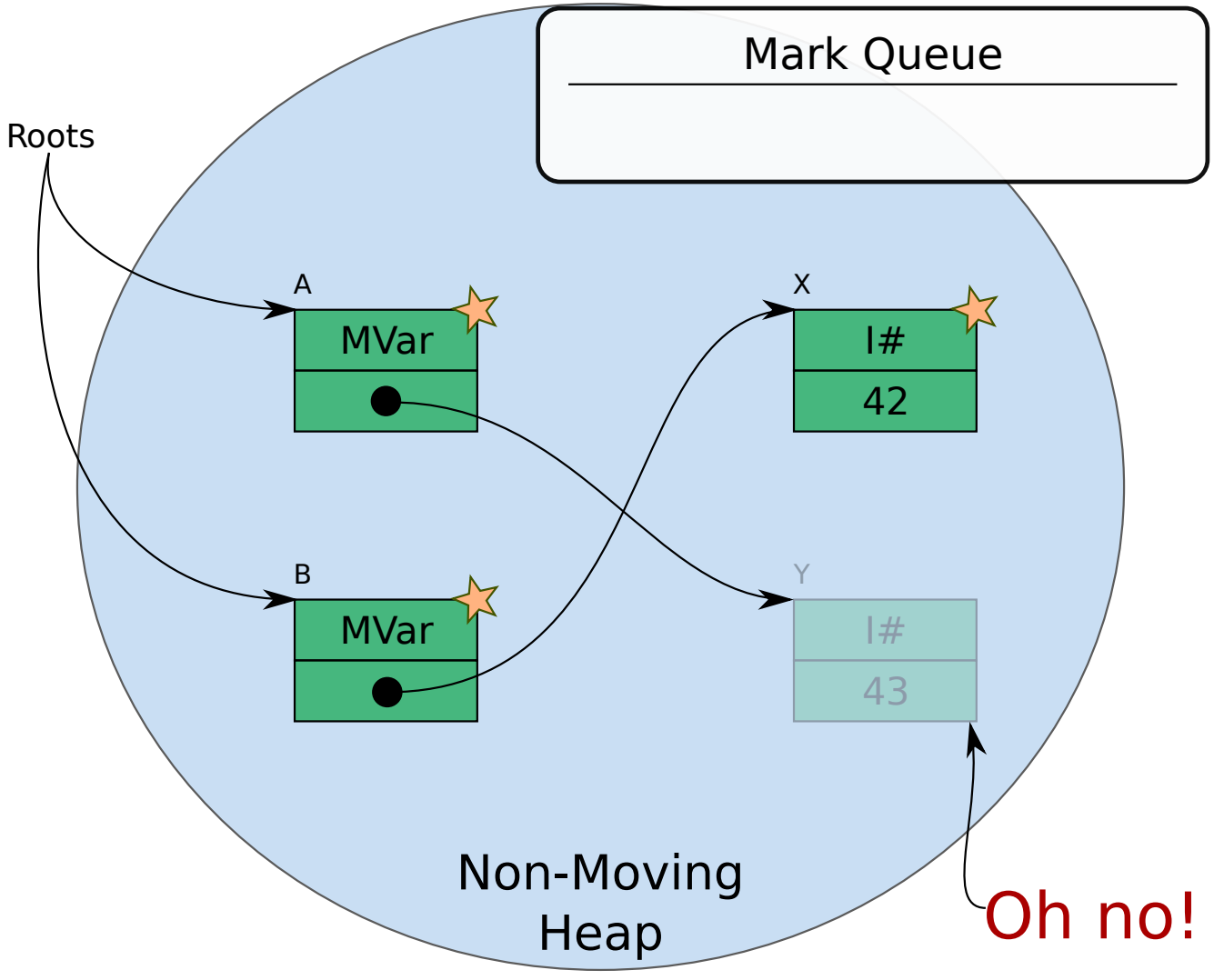
The problem of mutation



The problem of mutation



The problem of mutation



Snapshot-at-the-beginning

Solution:

- Collect with respect to the state of heap at start of mark (time t_0).

Snapshot-at-the-beginning

Solution:

- Collect with respect to the state of heap at start of mark (time t_0).

Concretely, the collector must ensure this property (henceforth the *snapshot invariant*):

> *The collector must mark all objects reachable at* t_0.

N.B. it is also safe to mark objects that were *dead* at t_0.

Snapshot-at-the-beginning

The snapshot invariant:

> *The collector must mark all objects reachable at t_0.*

Consequently,

1. All objects live at t_0 will be retained.
2. Many objects dead at t_0 will be freed.
3. All objects allocated after t_0 will be retained.

Snapshot-at-the-beginning

The snapshot invariant:

> *The collector must mark all objects reachable at t_0.*

Consequently,

1. All objects live at t_0 will be retained.
2. Many objects dead at t_0 will be freed.
3. All objects allocated after t_0 will be retained.

How to accomplish this?

Snapshot-at-the-beginning

The snapshot invariant:

> *The collector must mark all objects reachable at t_0.*

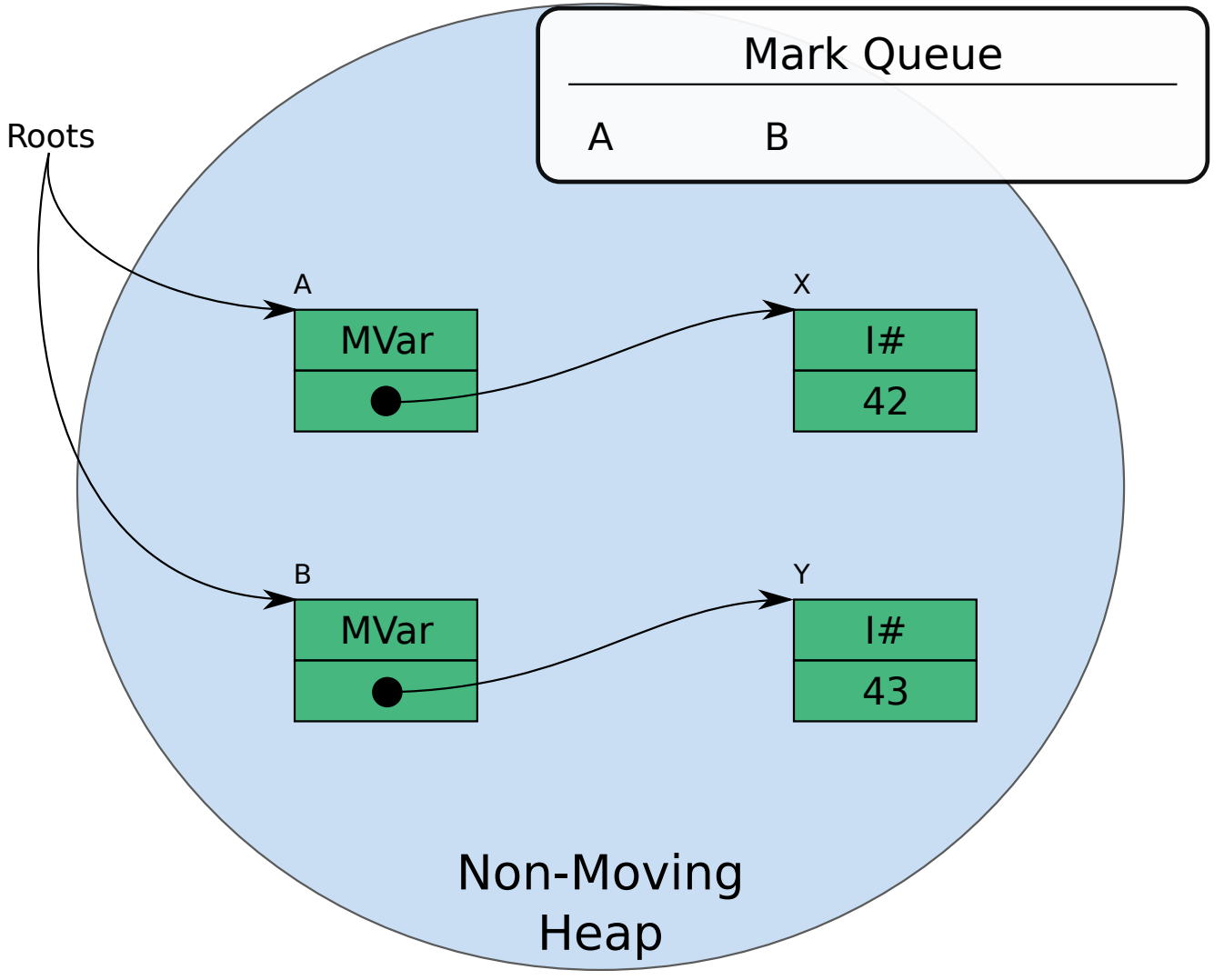
Consequently,

1. All objects live at t_0 will be retained.
2. Many objects dead at t_0 will be freed.
3. All objects allocated after t_0 will be retained.

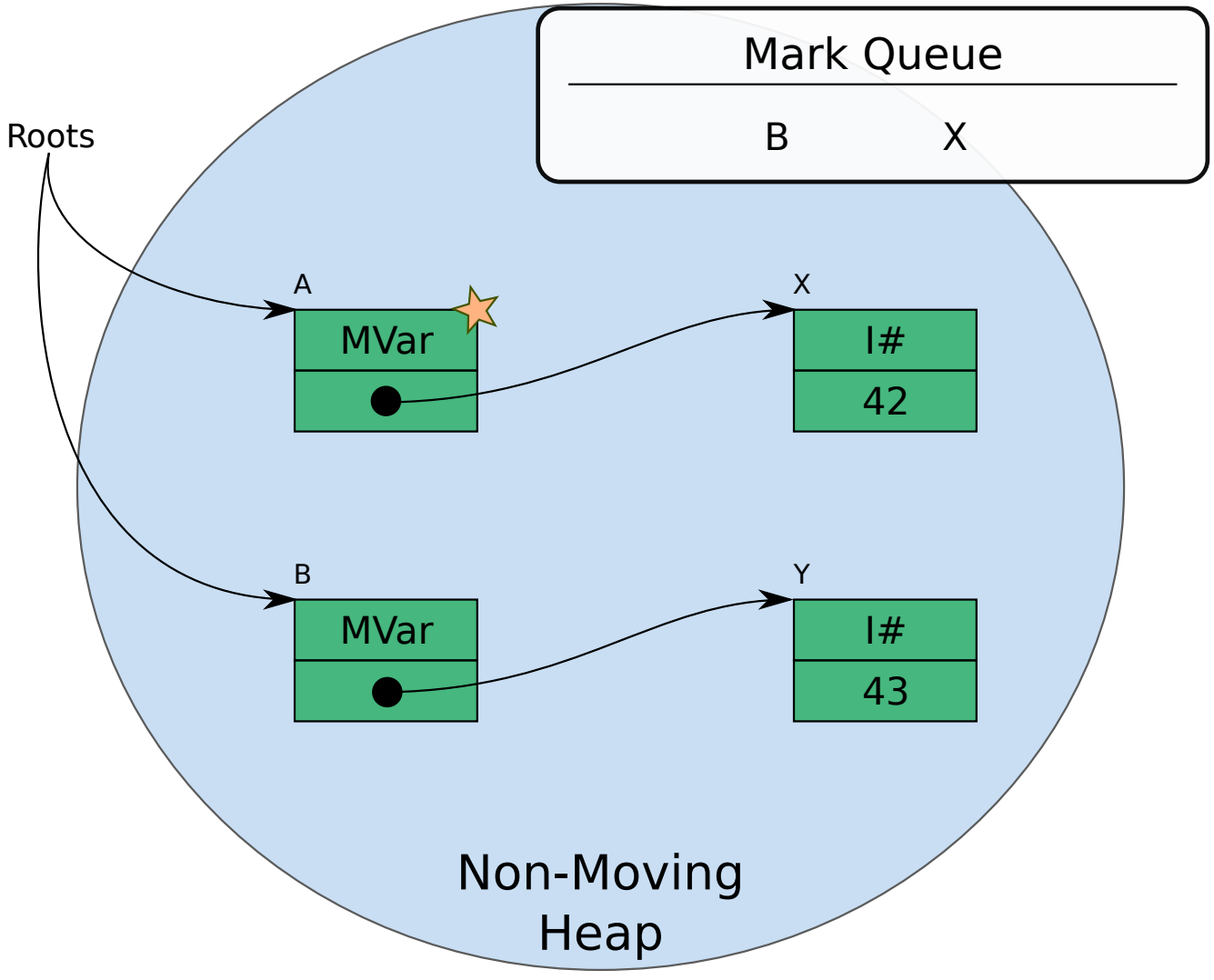
How to accomplish this?

A write barrier.

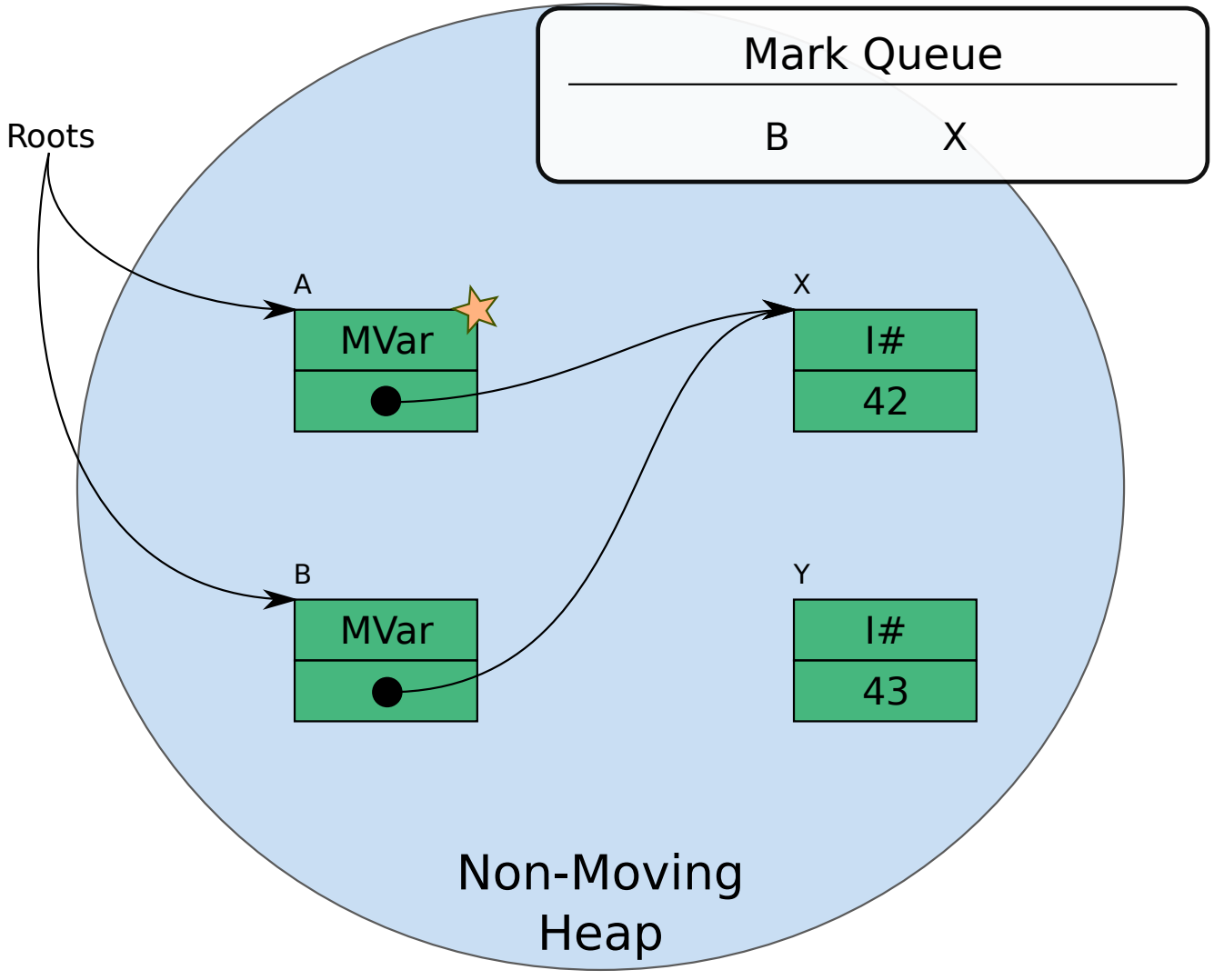
Maintaining the snapshot invariant



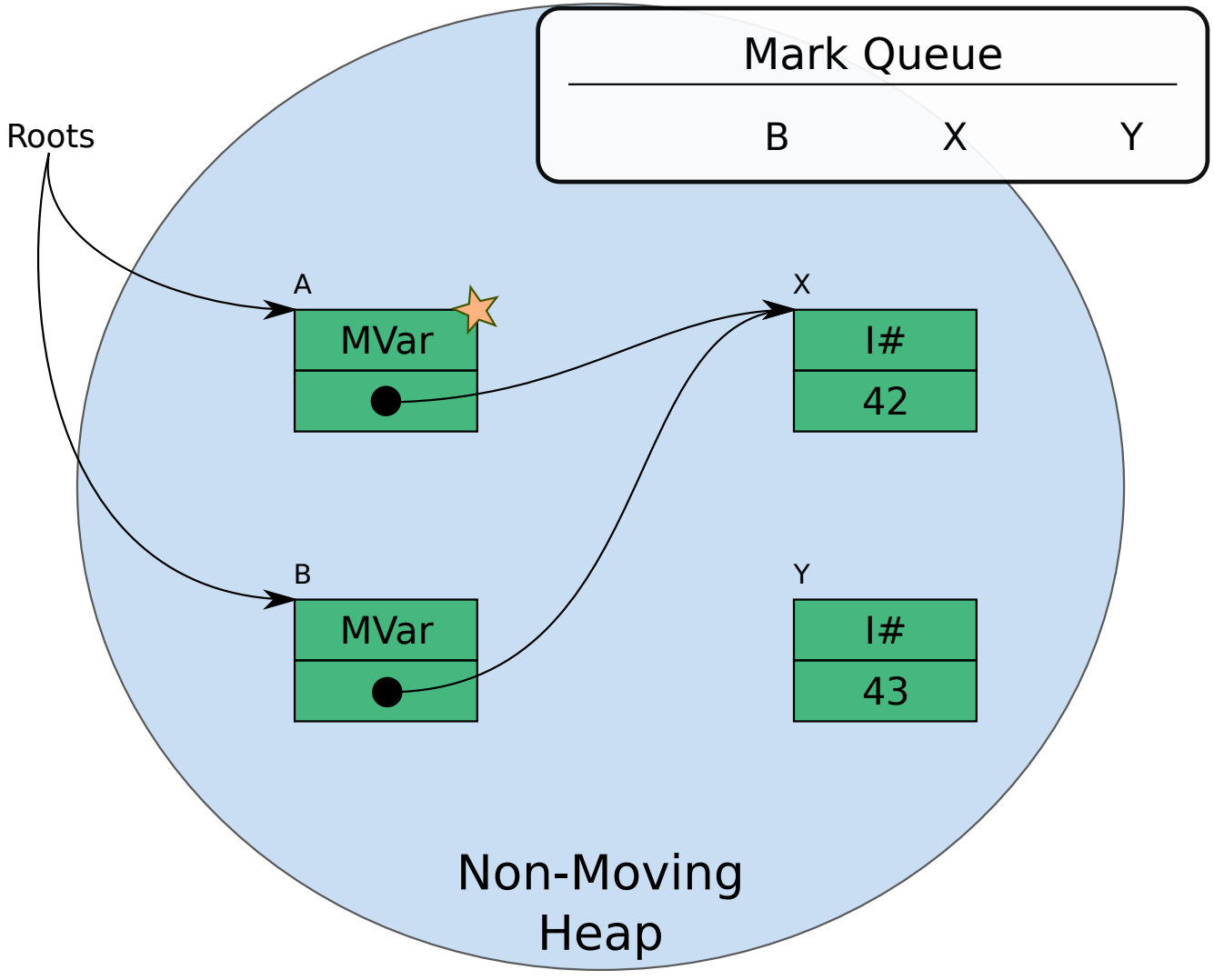
Maintaining the snapshot invariant



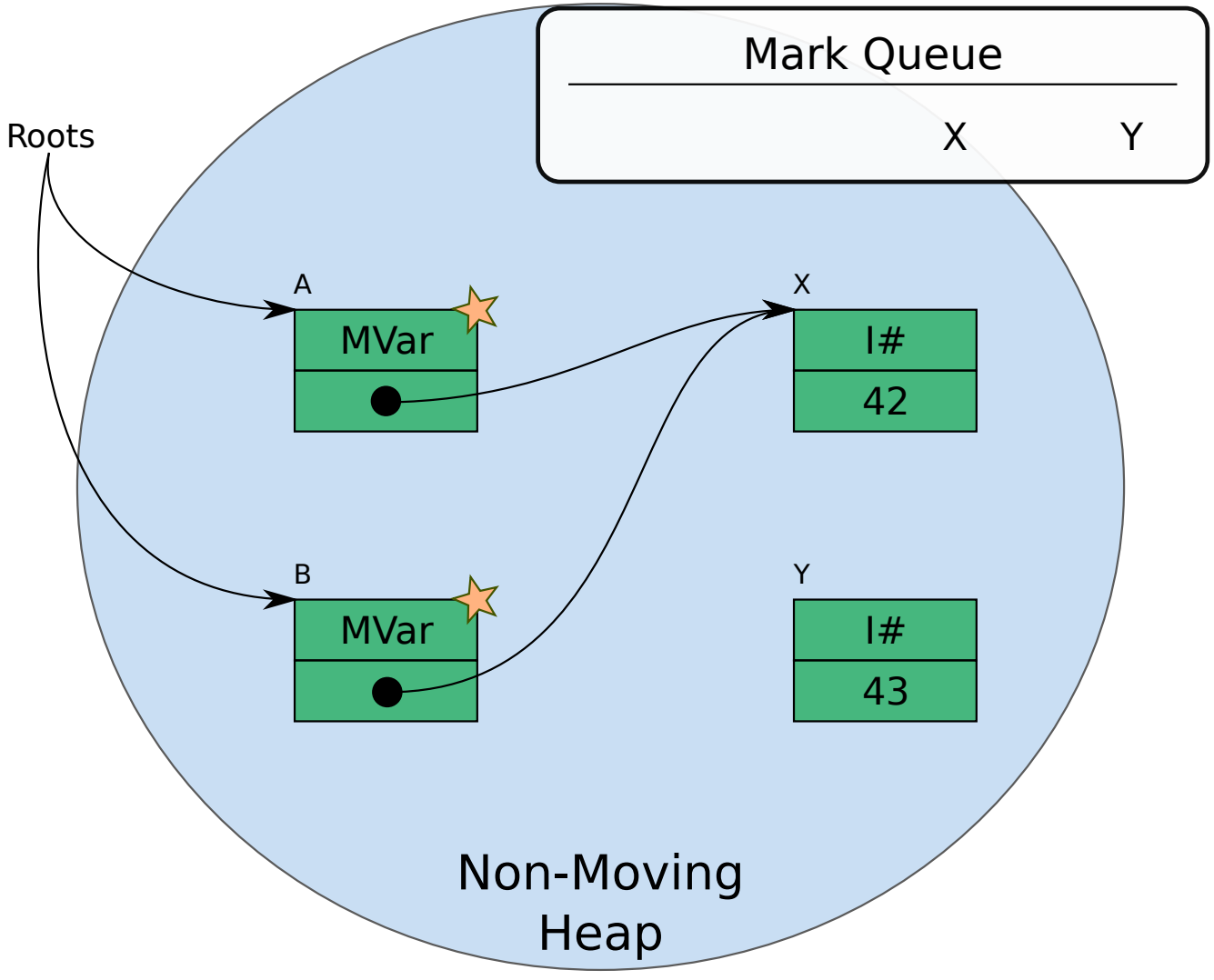
Maintaining the snapshot invariant



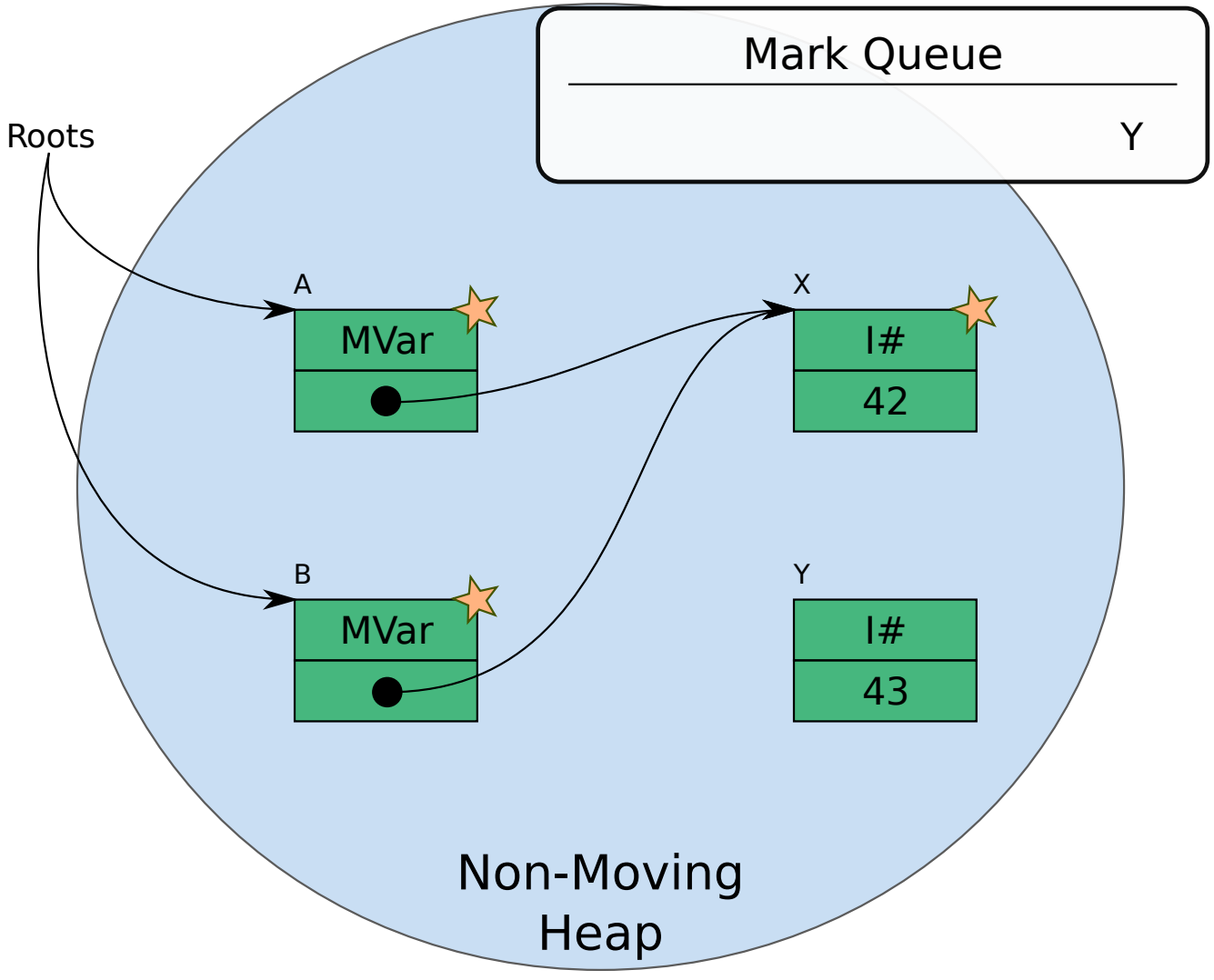
Maintaining the snapshot invariant



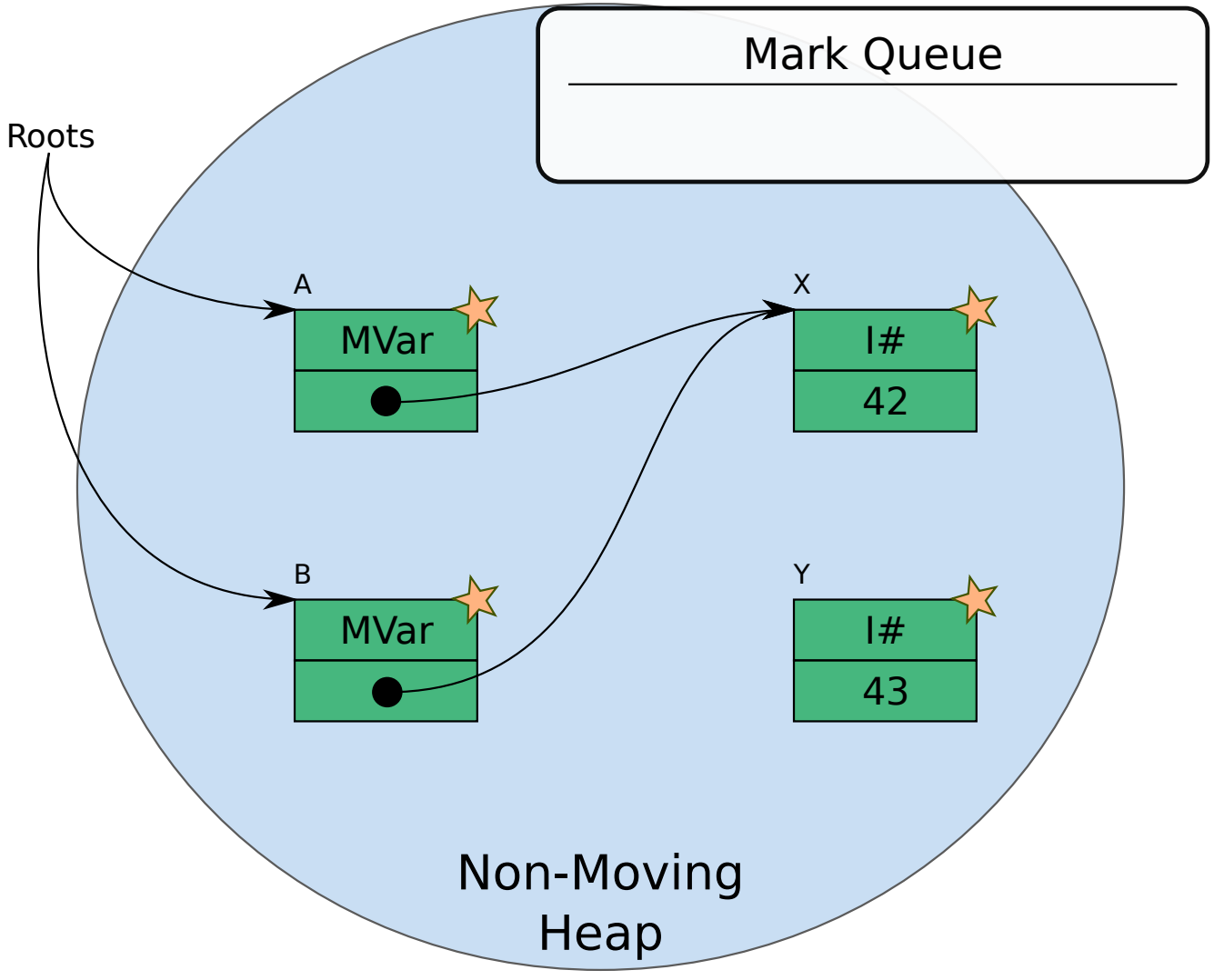
Maintaining the snapshot invariant



Maintaining the snapshot invariant



Maintaining the snapshot invariant



Allocator requirements

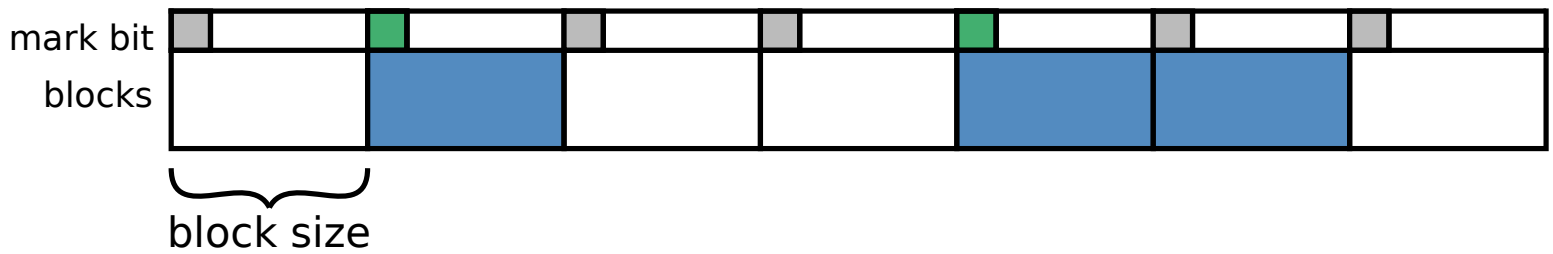
- Recall: Non-moving allocator serves nursery collector
- Relatively cheap allocation
- Fragmentation-resistant
- Efficient encoding of mark flags
- Snapshot construction must be cheap
- Accommodate concurrent collection and allocation
- Accommodate parallel collection

Allocator requirements

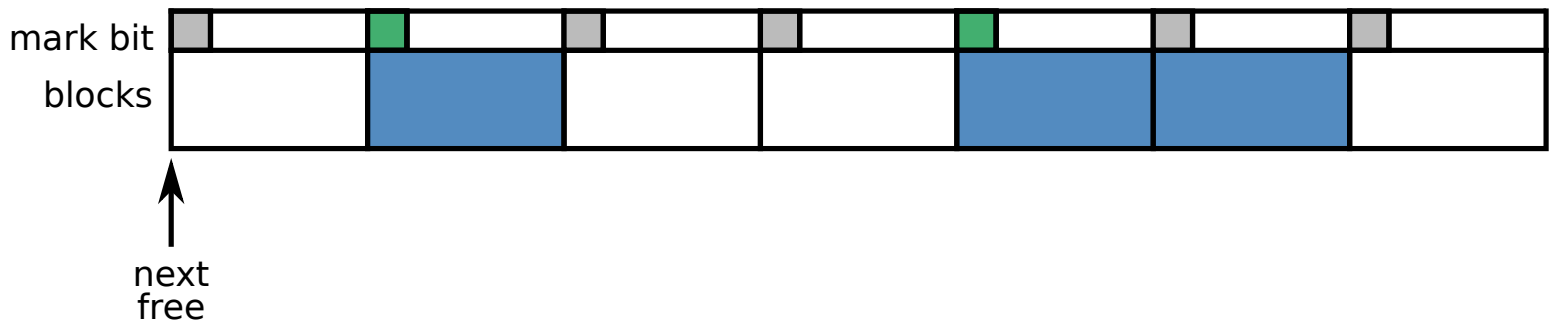
- Recall: Non-moving allocator serves nursery collector
- Relatively cheap allocation
- Fragmentation-resistant
- Efficient encoding of mark flags
- Snapshot construction must be cheap
- Accommodate concurrent collection and allocation
- Accommodate parallel collection

For simplicity: Let's first consider an allocator for a single object size.

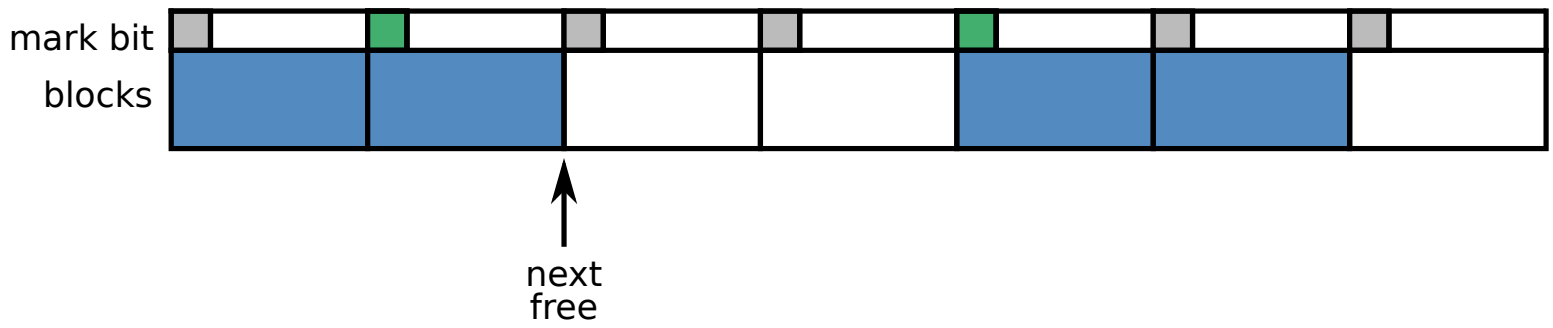
Allocator structure: The segment



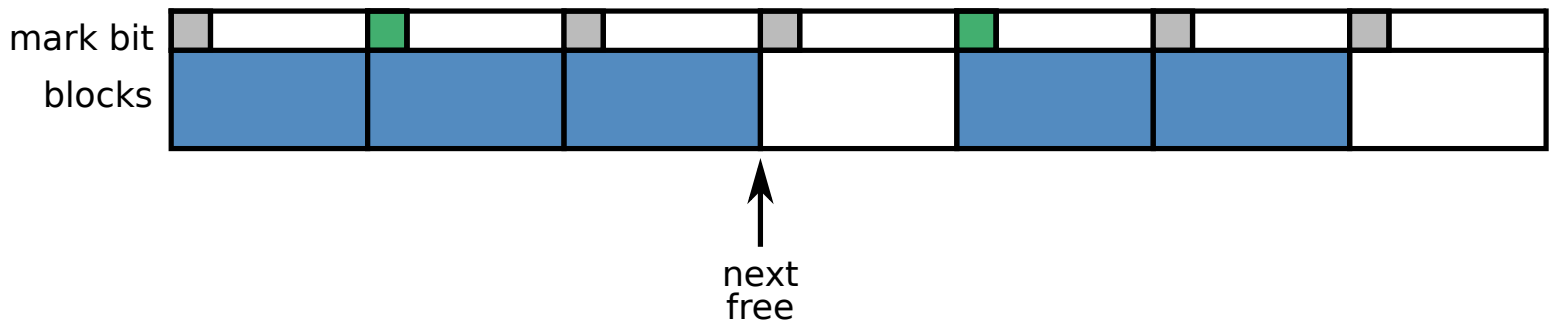
Allocator structure: The segment



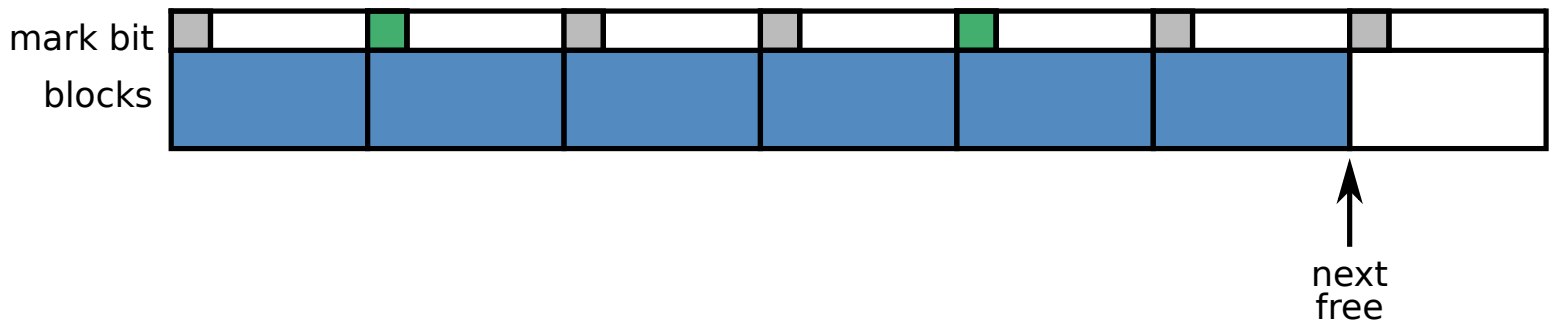
Allocator structure: The segment



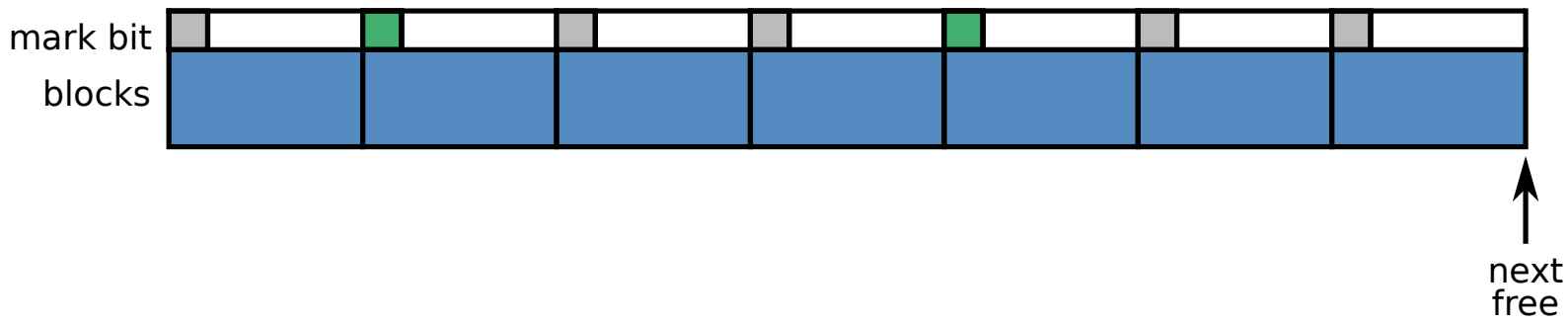
Allocator structure: The segment



Allocator structure: The segment



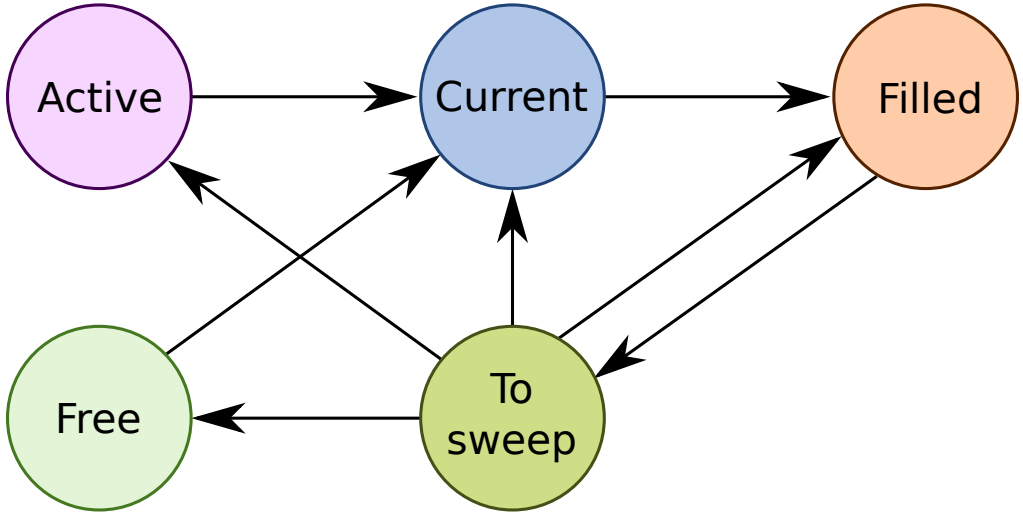
Allocator structure: The segment



Allocator structure: The segment

- Note that we did not set the mark bit during allocation
- Each capability has a *current* segment which serves its allocations.

Allocator structure: Segment states



Free segment A segment containing no live objects.

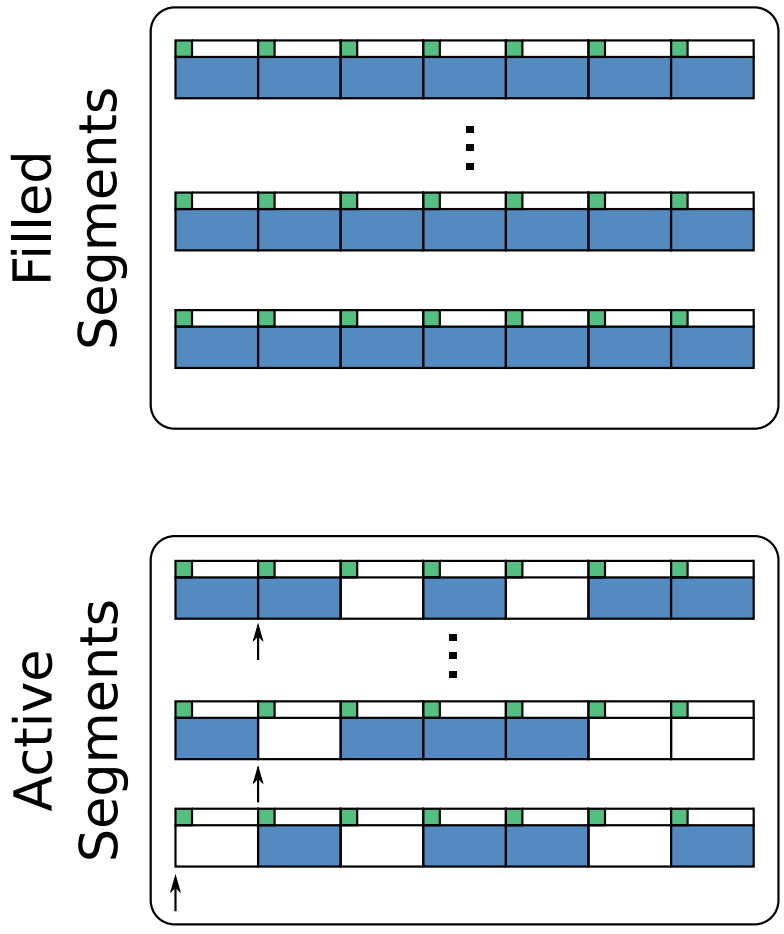
Current segment A segment being used by a capability as an allocation target.

Filled segment A segment having no unallocated blocks.

To-sweep segment A filled segment that will be swept during the current major GC cycle.

Active segment A segment containing at least one live object.

Allocator structure: Segment lists



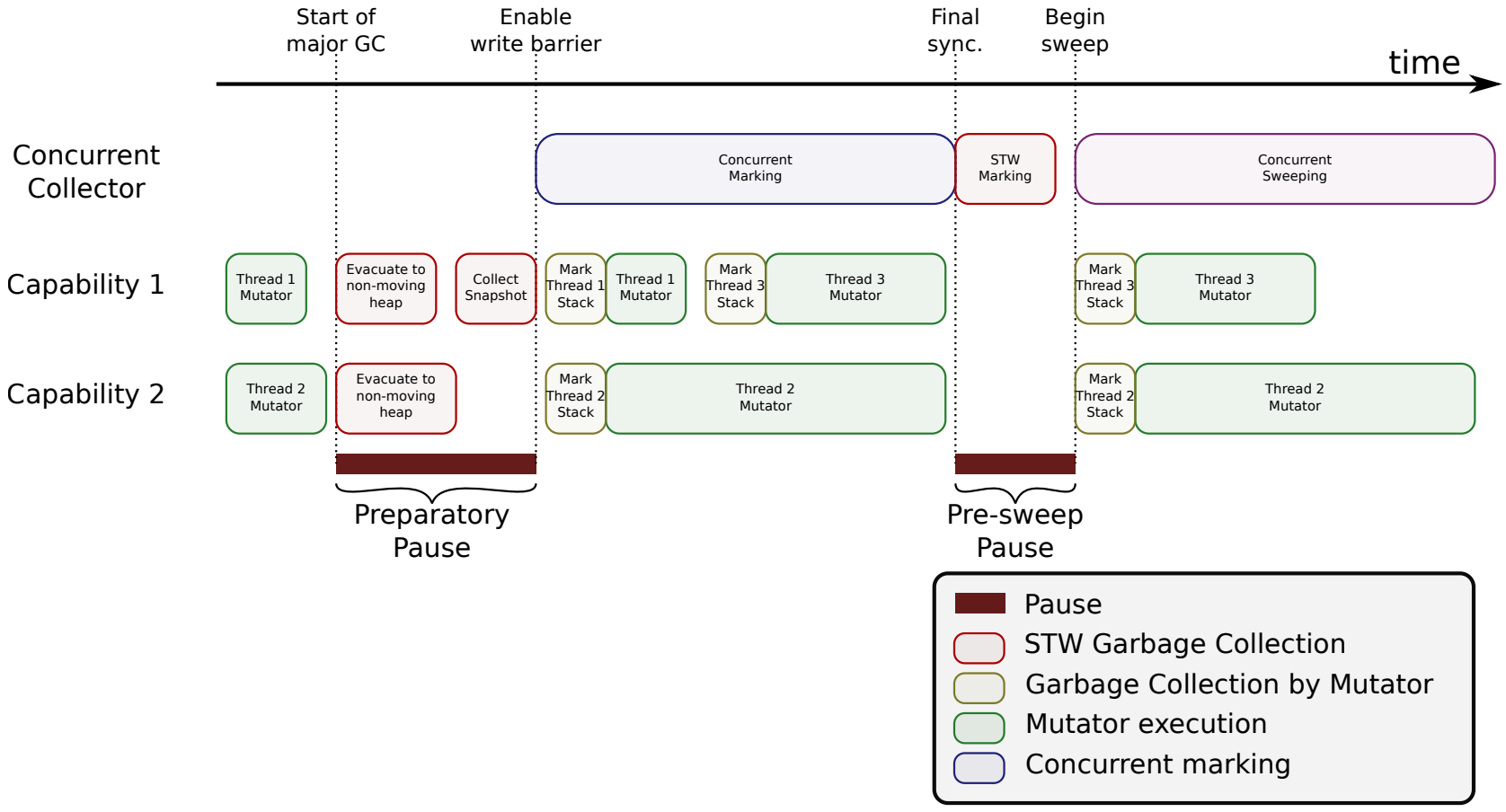
Extending the allocator to multiple object sizes

- Maintain a family of these allocators with logarithmically-spaced block sizes.
- Nicely fragmentation-resistant

Mark/sweep design

- Heavily borrows from Ueno 2016.
- Conservative: Only reclaim values in filled segments
- Write barrier entries accumulated in capability-local *update remembered set*
- Data-race freedom achieved by way of ownership:
  - All mark flags owned by collector
  - Free segments owned by allocator
  - Current segments owned by mutator
  - Active segments owned by collector

The lifecycle of a GC

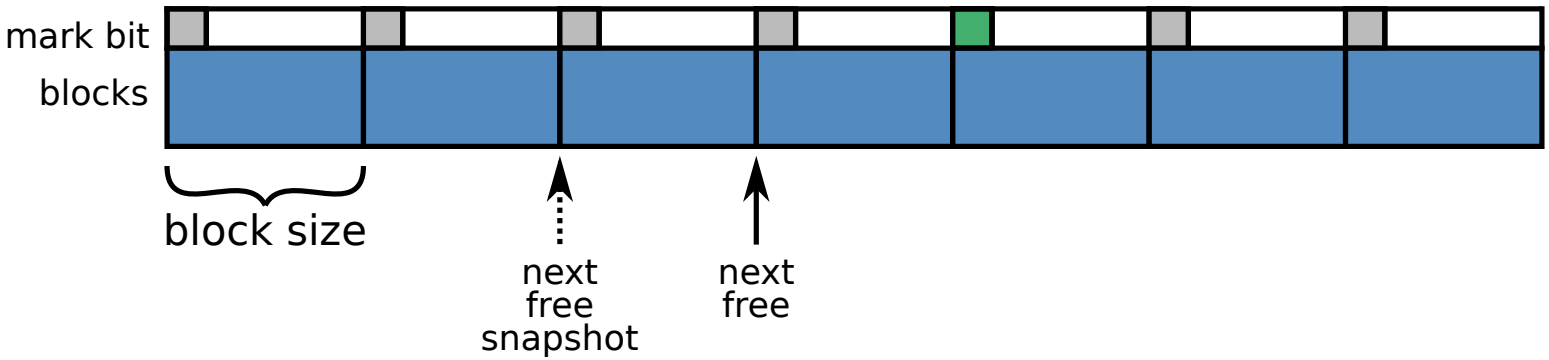


Determining object allocation state

- Recall: the snapshot invariant only requires that we mark objects which were reachable at t_0
- How do we know whether an object was allocated at t_0?

Determining object allocation state

- Recall: the snapshot invariant only requires that we mark objects which were reachable at t_0
- How do we know whether an object was allocated at t_0?



- Record the value of each segment's `next_free` field when the snapshot is taken.
- During collection we can conclude that objects above the snapshot needn't be marked.

Non-moving collection: Preparation Phase

Preparation:

1. Stop the world
2. Evacuate all live data to non-moving haep
3. Collect snapshot
4. Collect root set, push to mark queue
5. Start concurrent mark
6. Resume mutators

Marking:

1. Mark until mark queue is empty
2. Stop the world
3. Request update remembered sets from mutators
4. Mark objects reachable from update remembered set
5. Resume mutators

Sweeping:

1. Sweep unmarked objects

Non-moving collection: Marking

```
void mark_closure(MarkQueue queue, Closure *p) {
    Segment seg = get_segment(p);
    Int i = get_block_index(p);

    // Is mark flag already set?
    if (seg.mark_flag[i])
        return;

    if (i > seg.next_free_snapshot)
        // Either not yet allocated or allocated since
        // t_0, no need to mark.
        return;

    for (Closure *p in p.pointers())
        queue.push(p);

    seg.set_mark_flag(i);
}
```

Details abound

- Concurrent, incremental marking of stacks
- Weak pointers
- Large objects
- Handling of cycles
- Constant Applicative Forms
- Selector optimization
- Indirection shortcutting
- Collection scheduling policy
- Various tricks to help mark efficiency
  - Incremental handling of arrays
  - Closure prefetch

All beyond the scope of this talk but happy to discuss offline.

What the new collector won't do...

Concurrent collection isn't a silver bullet:

- It probably won't make your program run faster
- It won't make your program scale more effectively across cores (but this may be future work)
- It may reduce your program's memory footprint, but not by as much as you might expect
- It is not provide hard realtime latency guarantees
- It does not mow your lawn (yet)

Performance

Preliminary measurements:

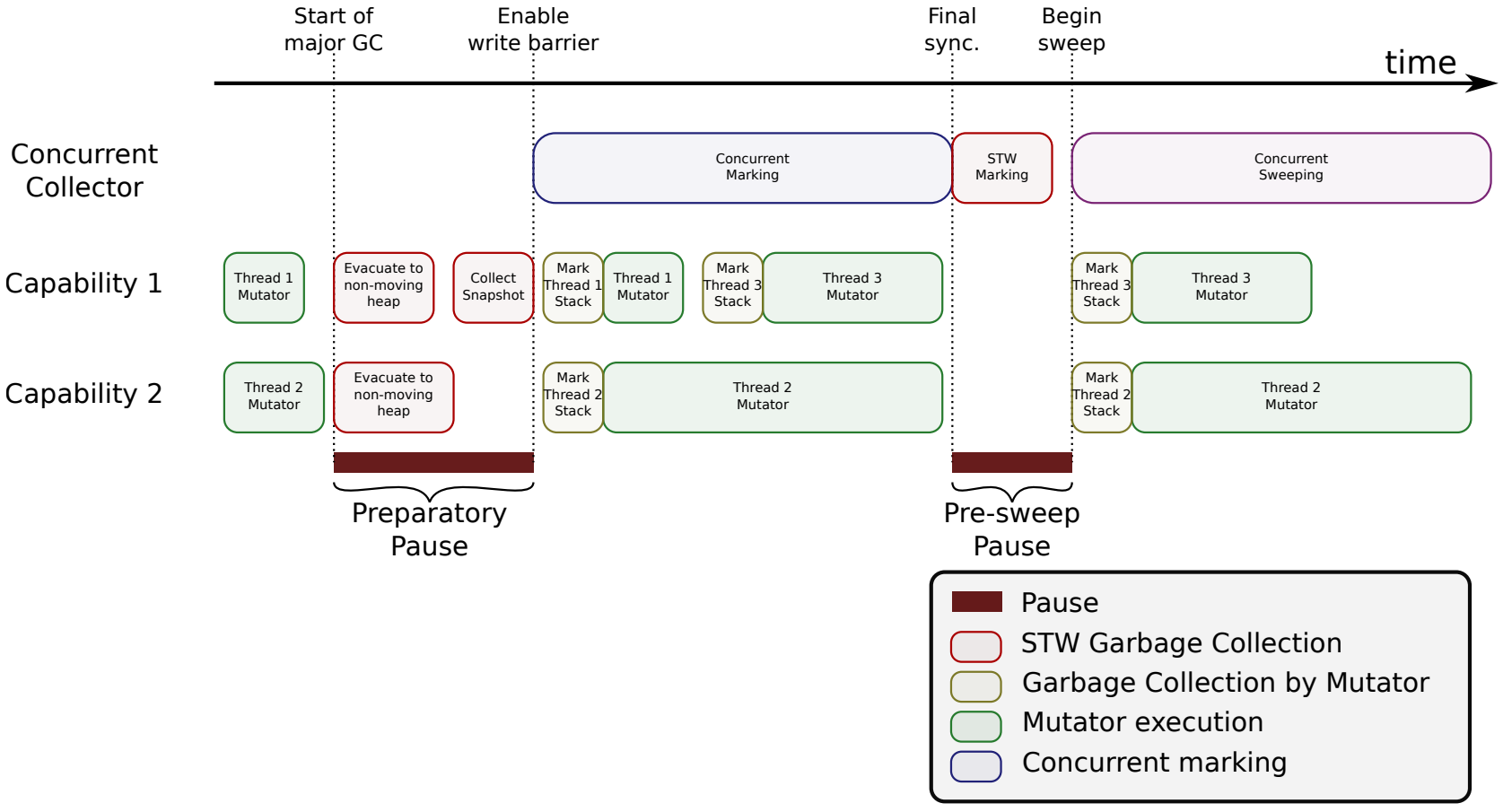
- Major GC pause times reduced by factor of between 5 and 50
- Major pauses generally comparable to minor collection pauses
- Mutator throughput generally regresses by around 10%, more work to be done

What work remains?

- Things currently "mostly work"
  - ~20 failing testsuite tests
- Hopefully attain testsuite correctness in the coming weeks
- Characterisation and optimisation follows
- Still some things missing:
  - Cost-center profiler support
  - Support for compact normal forms
  - STM not adequately tested
  - Selector optimisation, indirection shortcutting currently disabled
  - RTS shutdown is a living nightmare
- Plan to merge for GHC 8.10

Summary

Questions?



Email: ben@well-typed.com

Future work

- Teaching mutator to allocate directly into non-moving heap
  - Would require changes in code generation
- Further shrink pause times
  - Ueno 2016 proposes a collector which allows a concer synchronization