# A Tour of GHC’s Garbage Collection

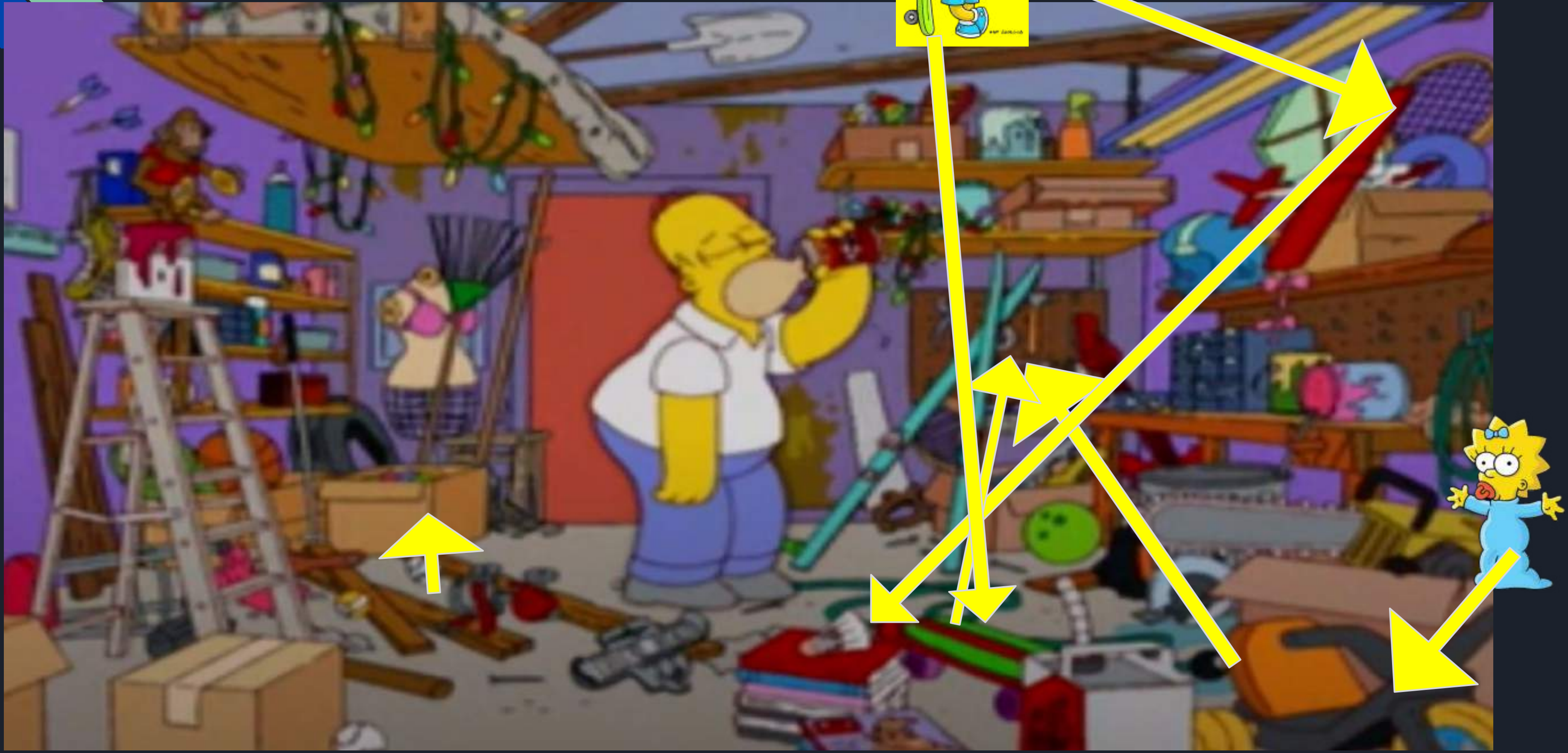
Jost Berthold, FP Syd 2024 07 24

# TOC



**Featuring *heaps* …**
***of slides***
***borrowed from the***
***internet***

# Garbage in the Garage



# Basic algorithms

## —The garage metaphor—

**Reference counting:** Maintain a note on each object in your garage, indicating the current number of references to the object. When an object's reference count goes to zero, throw the object out (it's dead).

**Mark-Sweep:** Put a note on objects you need (roots). Then recursively put a note on anything needed by a live object.
Afterwards, check all objects and throw out objects without notes.

**Mark-Compact:** Put notes on objects you need (as above). Move anything with a note on it to the back of the garage.
Burn everything at the front of the garage (it's all dead).

**Copying:** Move objects you need to a new garage. Then recursively move anything needed by an object in the new garage.
Afterwards, burn down the old garage (any objects in it are dead)!



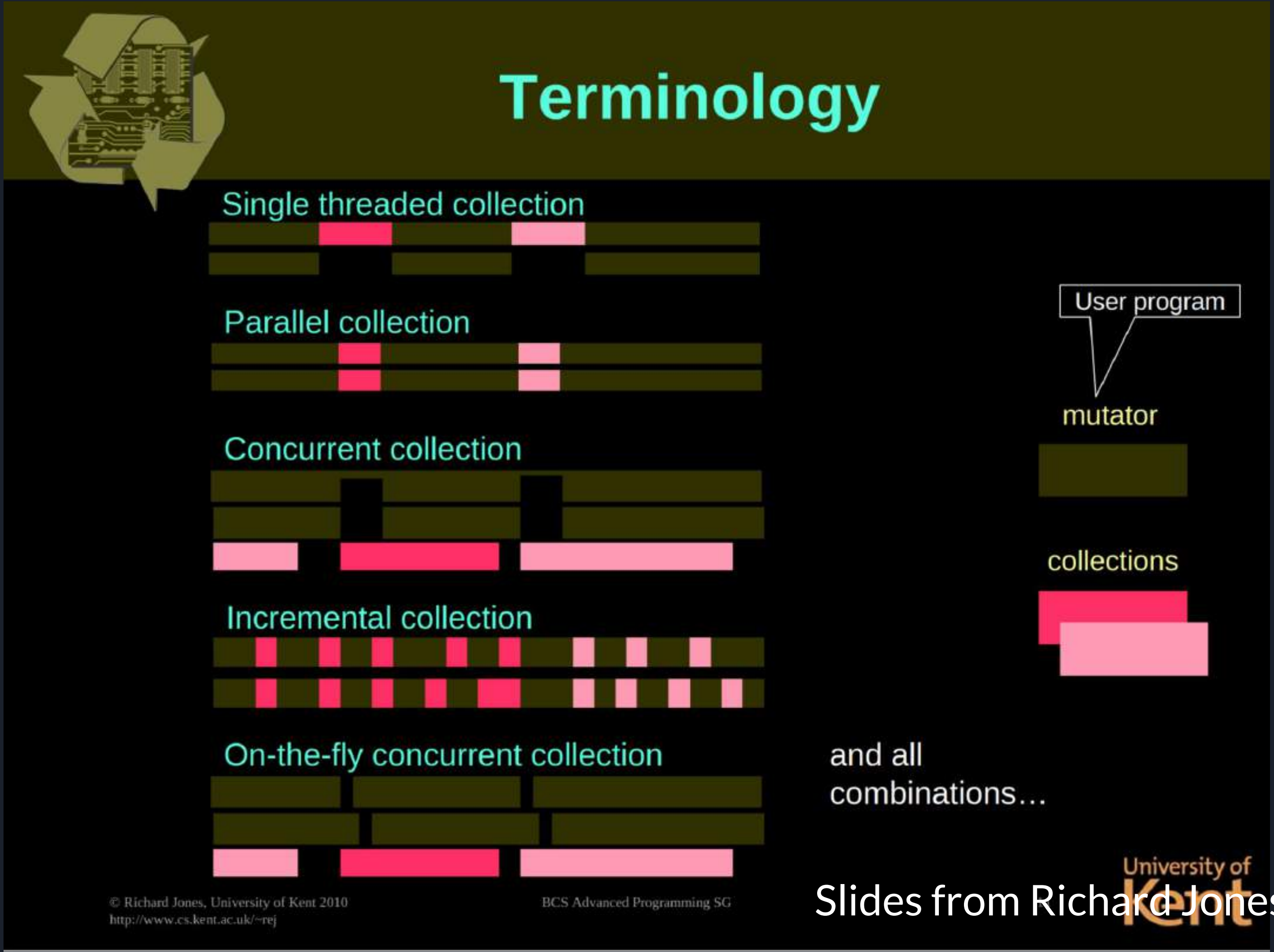
Slides from Richard Jones 2010

# Background: Types of Garbage Collection (GC)

- Identifies items allocated in memory that are not used any more

- Reclaims allocated memory to store new items

- *Reference counting* vs *tracing* strategies
  - Reference counting: identifies and removes *dead* items
  - Tracing: identifies and protects items that are *alive*



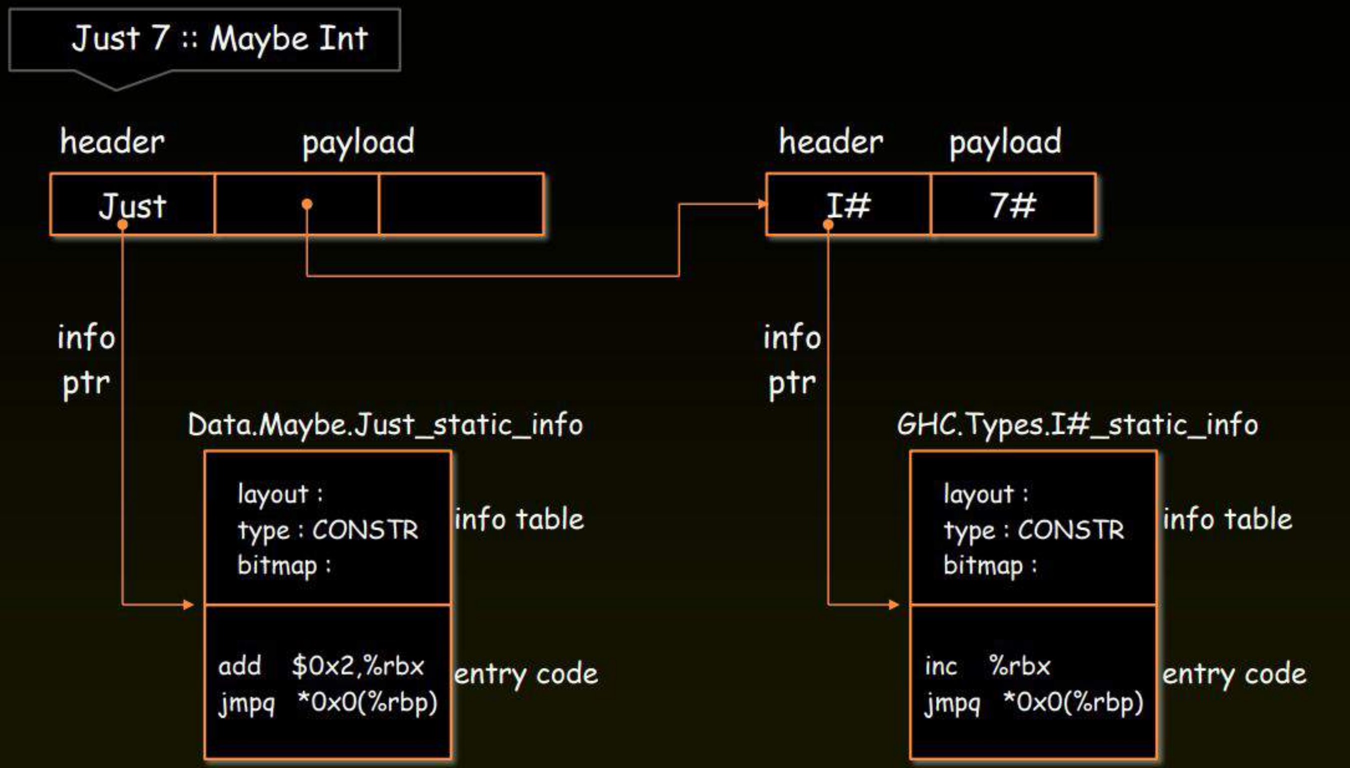

Slides from Richard Jones 2010

# Basic Garbage Collection in GHC

- (Cheney-style) 2-space copying GC
- Heap contains *closures* of evaluated or unevaluated (thunk) data
-

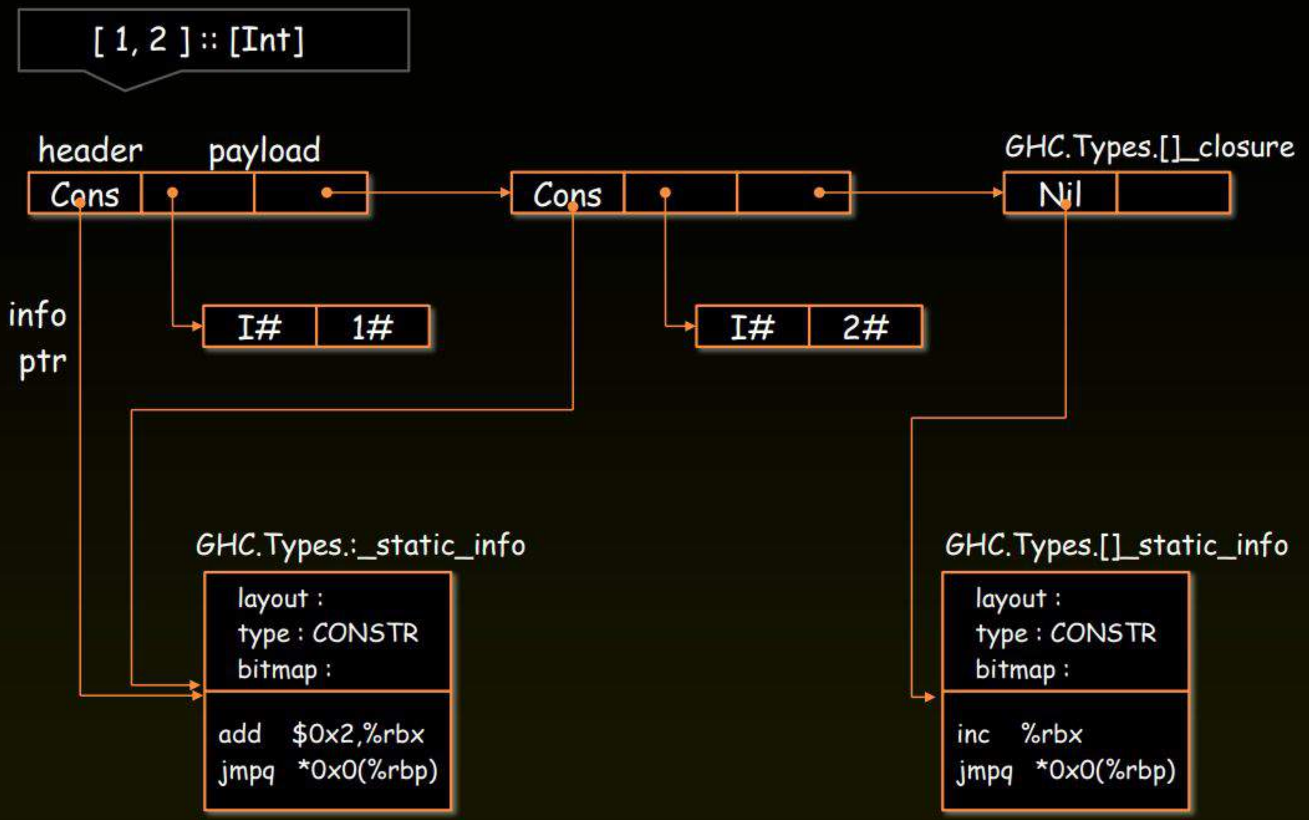
```
./my-program +RTS -H2G -RTS # 2G min heap
./my-program +RTS -M12G -RTS # 12G max heap
./my-program +RTS -m5 -RTS # ensure 5% heap remain available
./my-program +RTS -sstderr -RTS # get a GC summary at the end
```

# Closure examples : Maybe



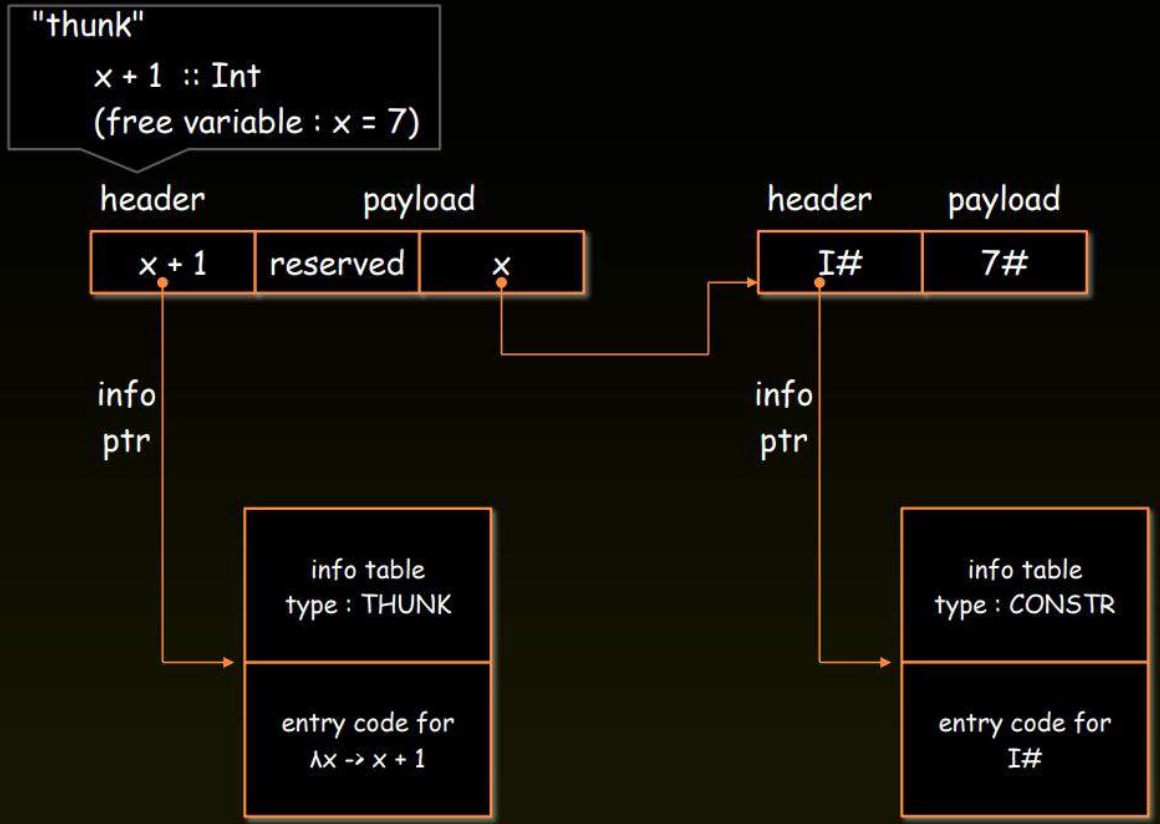

Slides from Takenobu Tani (GHC Illustrated)

# Closure examples : List




Slides from Takenobu Tani (GHC Illustrated)

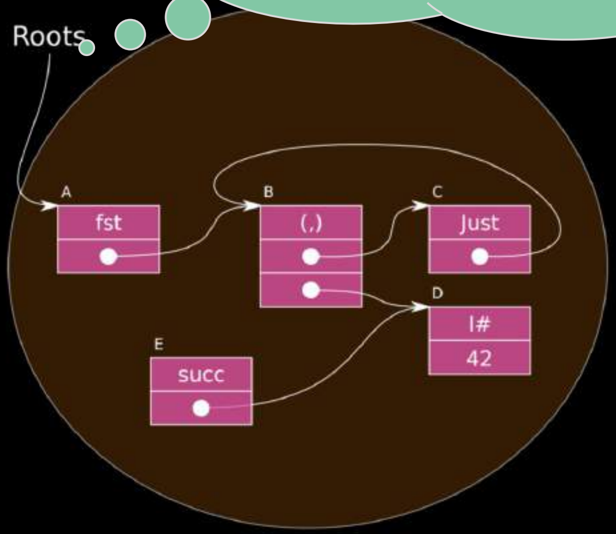
# Closure examples : Thunk




Slides from Takenobu Tani (GHC Illustrated)

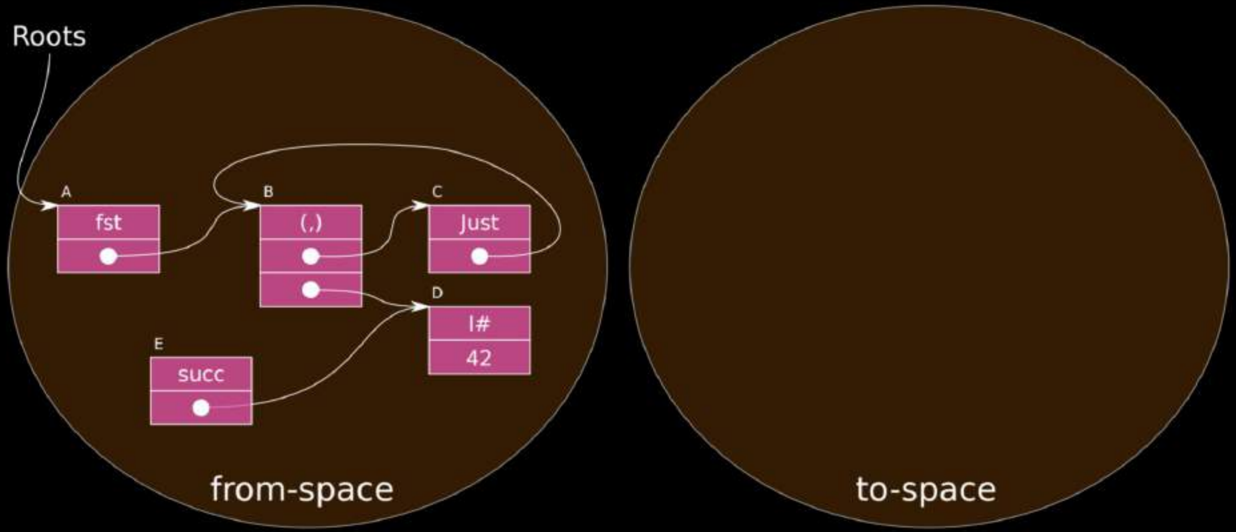
# Moving garbage collection

```
let pair = (Just pair,
42)
in fst pair
```



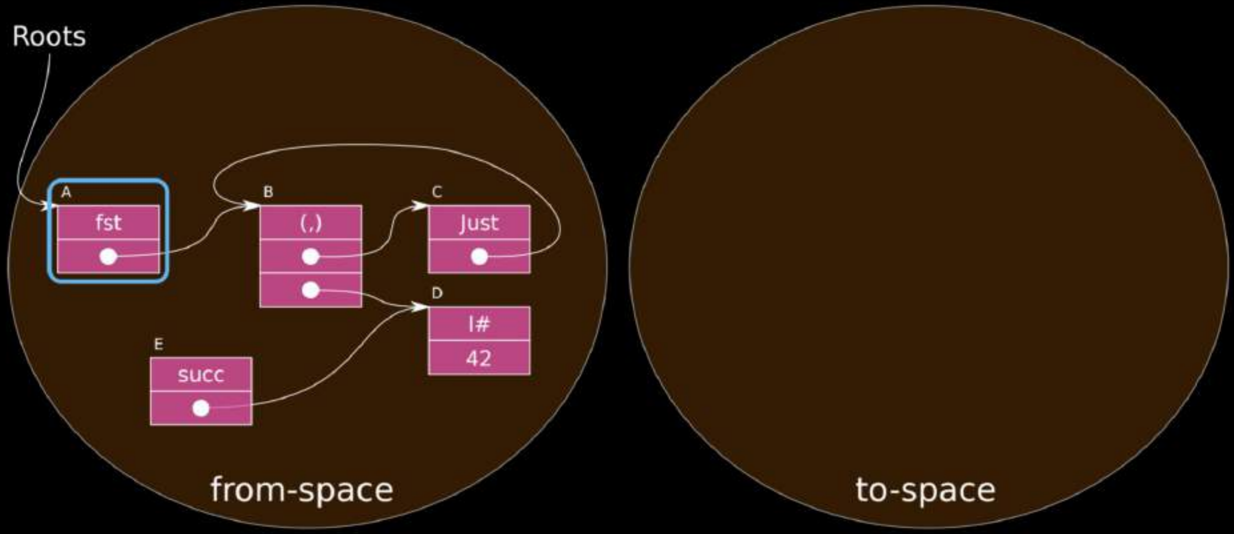

Slides from Ben Gamari's talk 2018

# Moving garbage collection



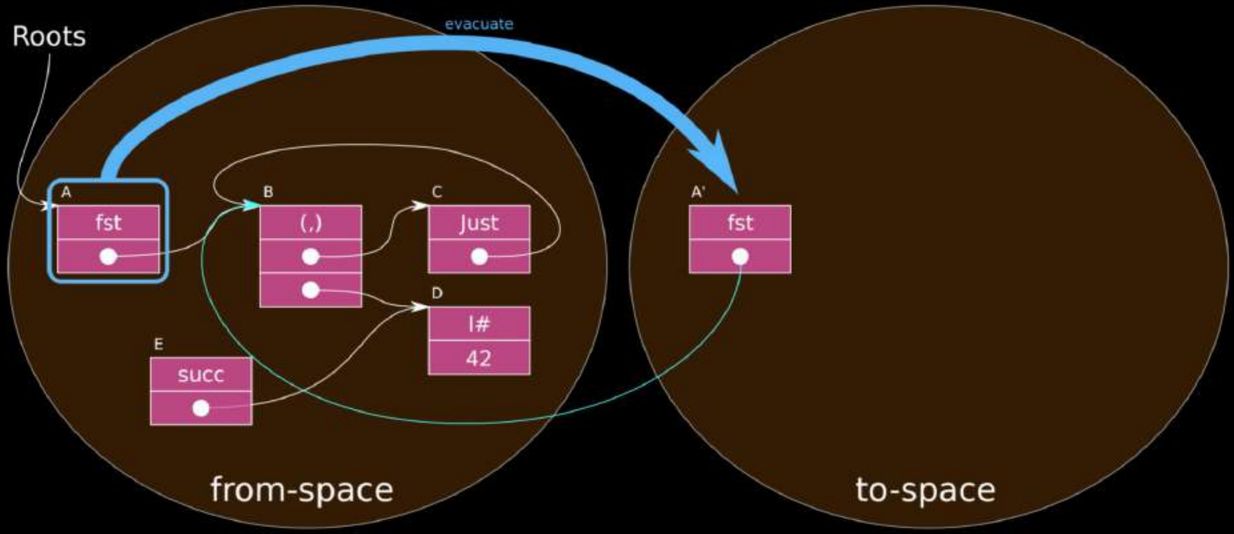

Slides from Ben Gamari’s talk 2018

# Moving garbage collection: Evacuate roots



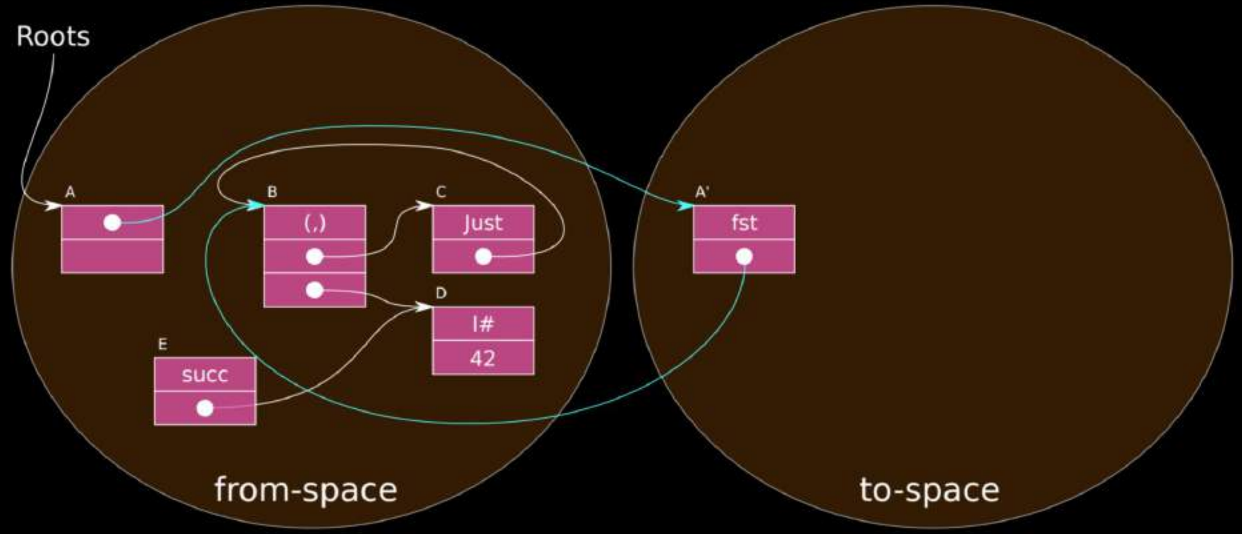

Slides from Ben Gamari’s talk 2018

# Moving garbage collection: Evacuate roots



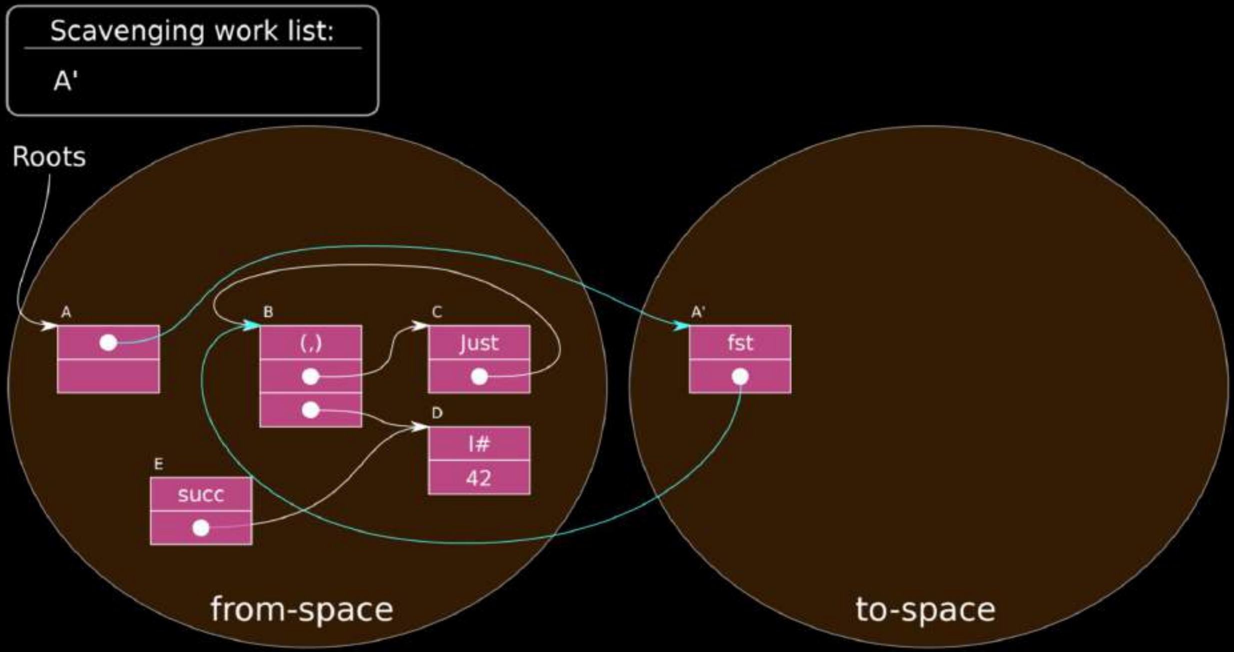

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Evacuate roots



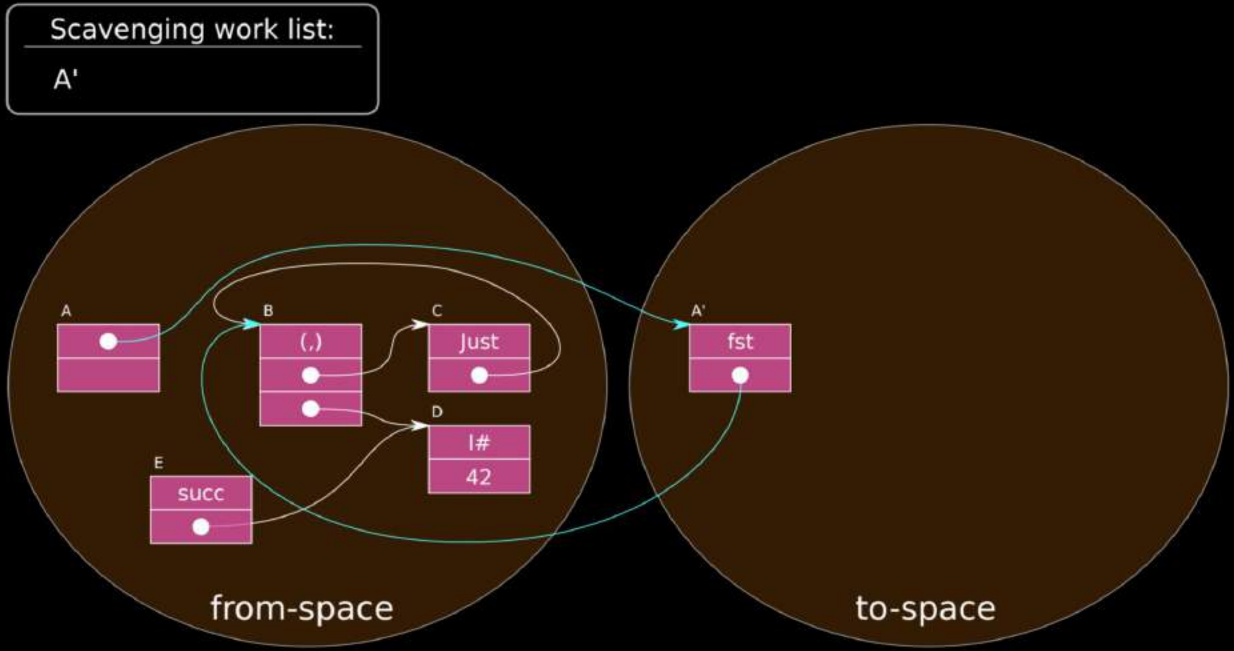

Slides from Ben Gamari’s talk 2018

# Moving garbage collection: Evacuate roots



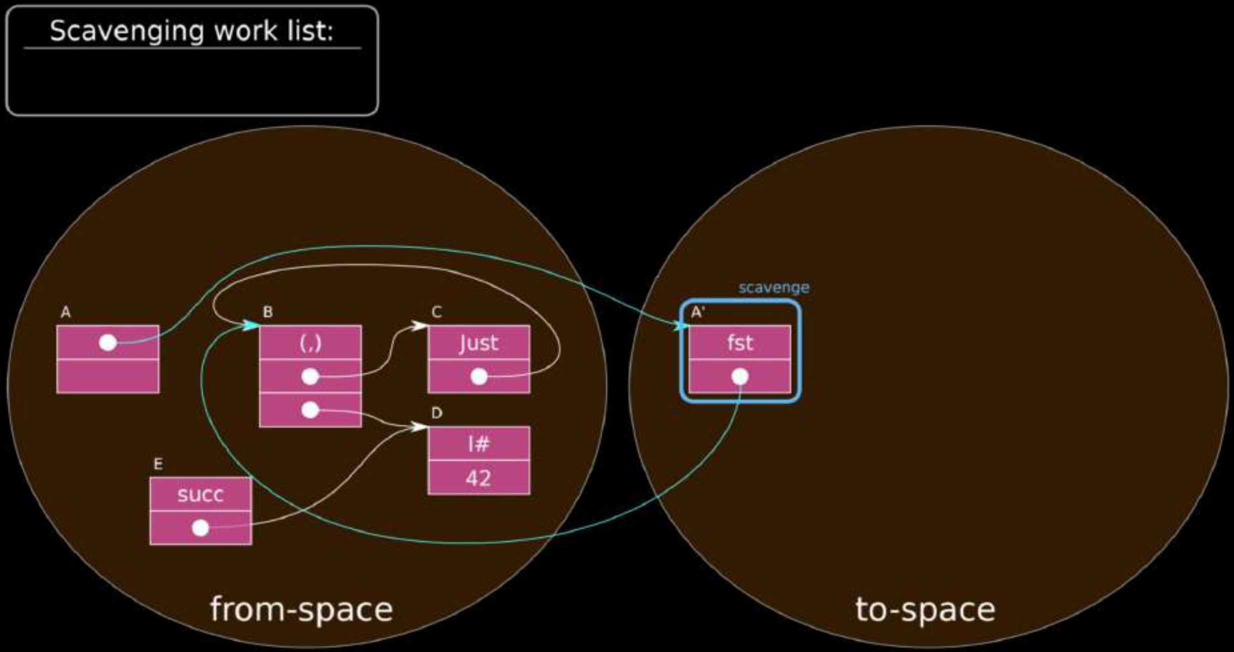

Slides from Ben Gamari's talk 2018

# Moving garbage collection



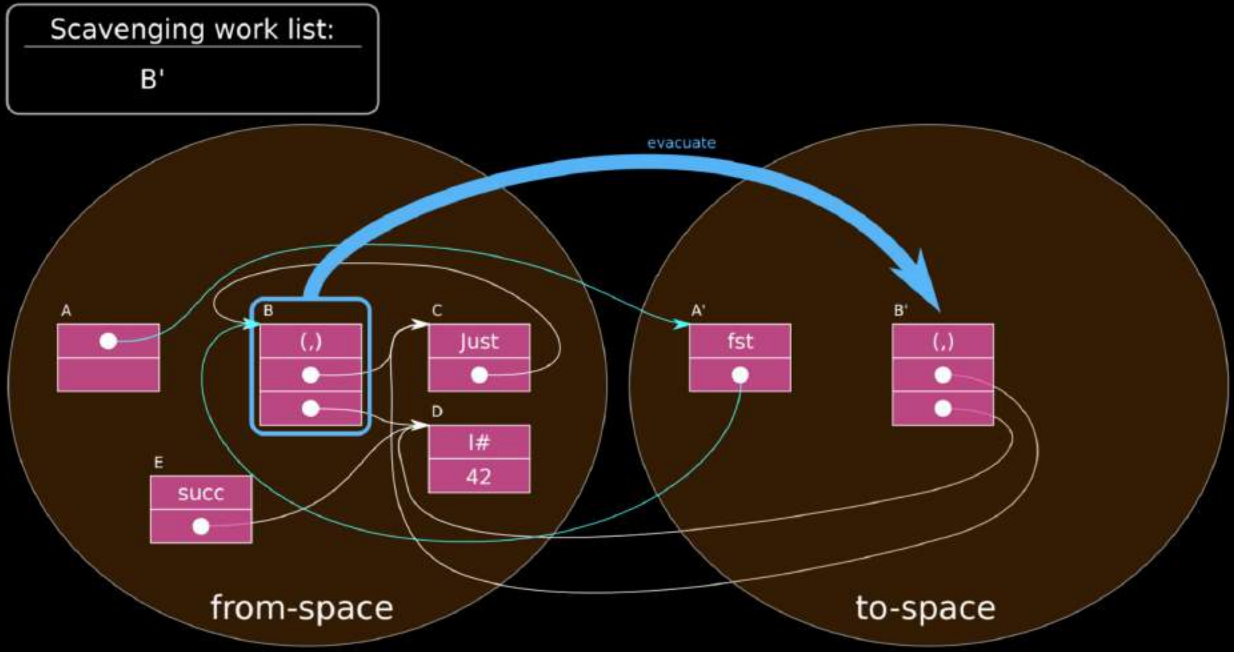

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Scavenge A



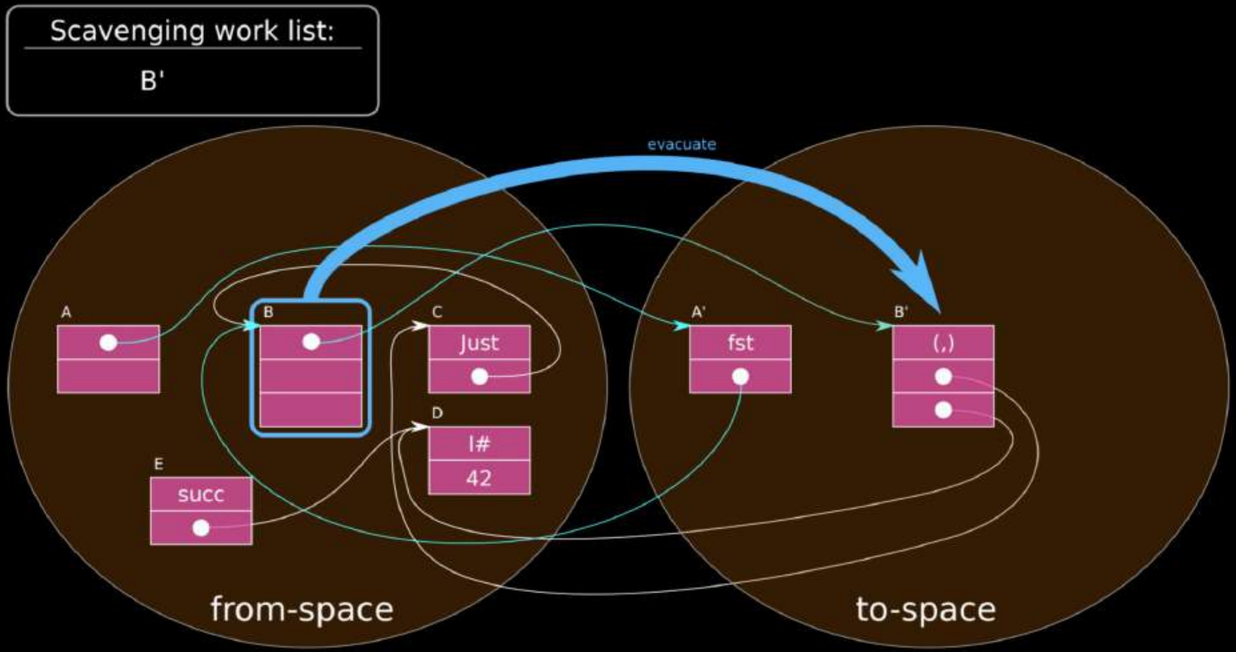

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Evacuate B



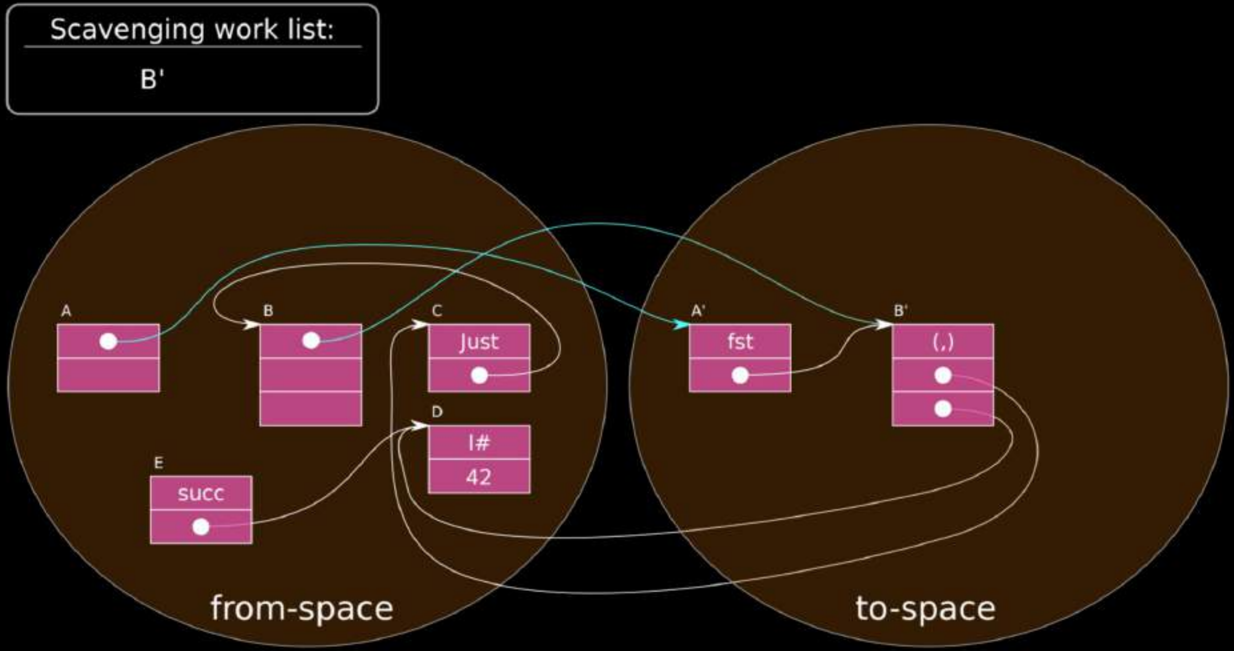

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Evacuate B



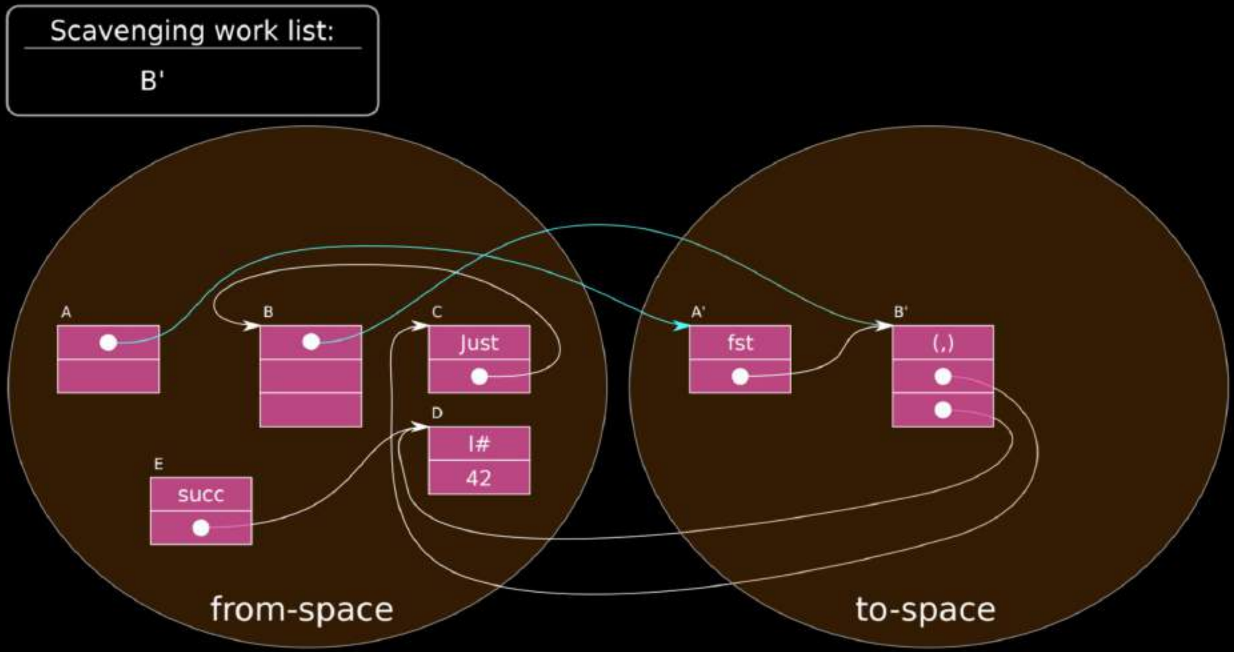

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Evacuate B



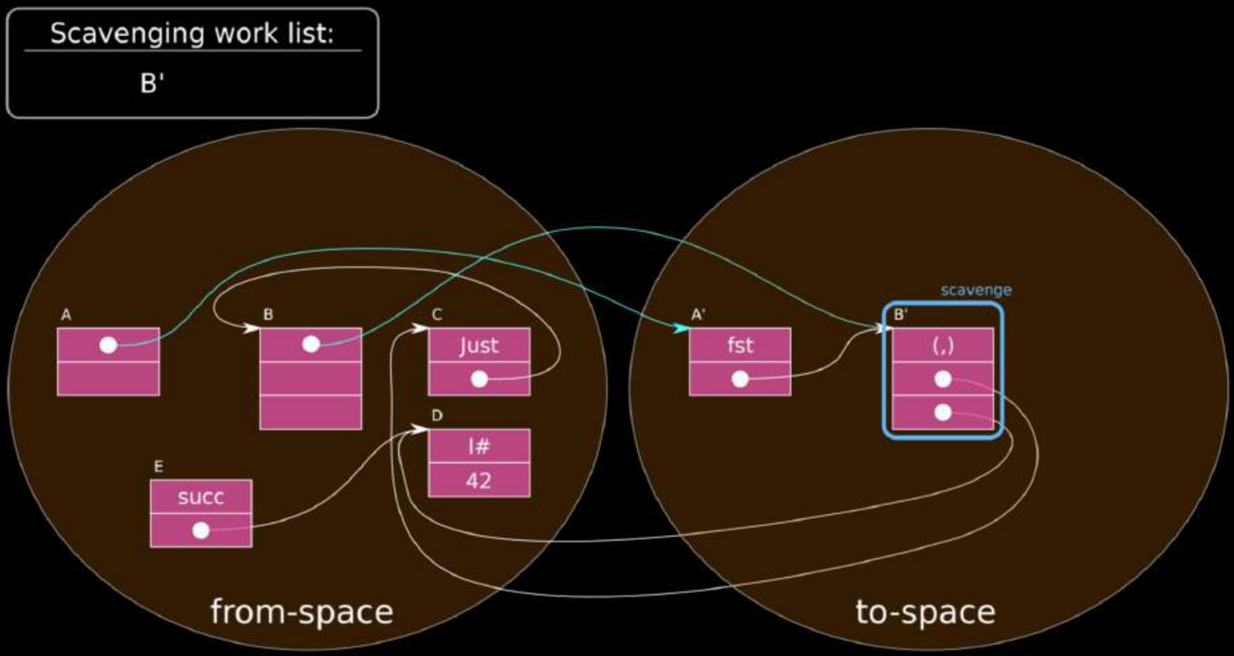

Slides from Ben Gamari's talk 2018

# Moving garbage collection



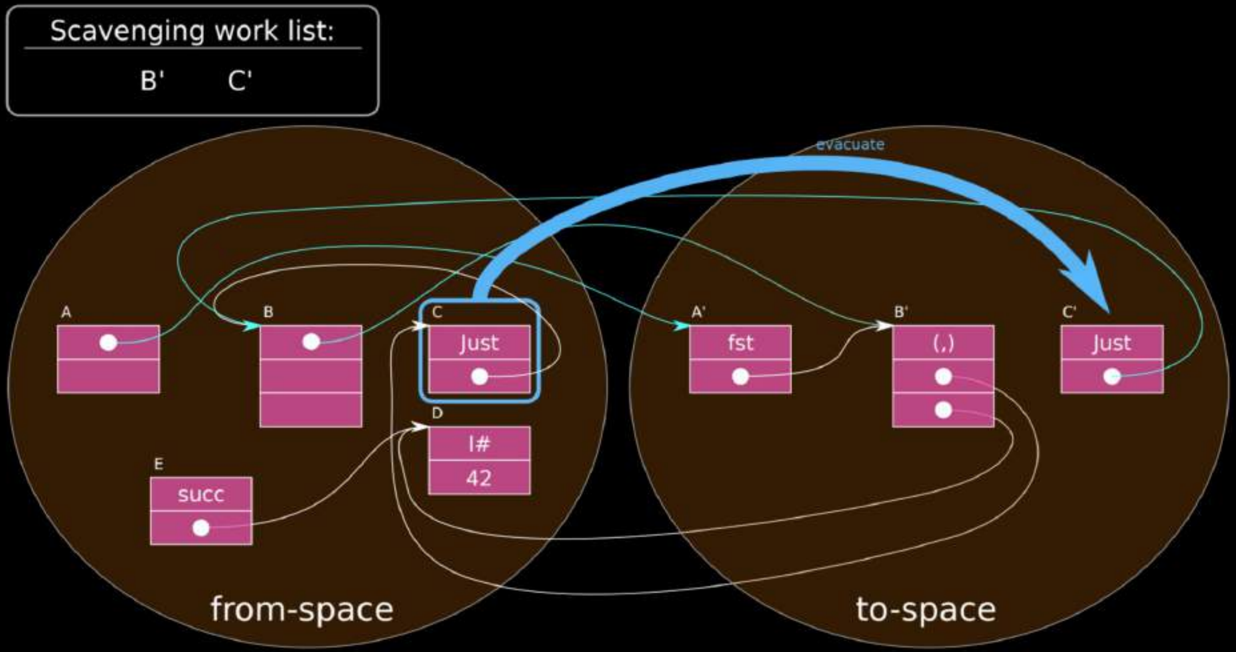

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Scavenge B



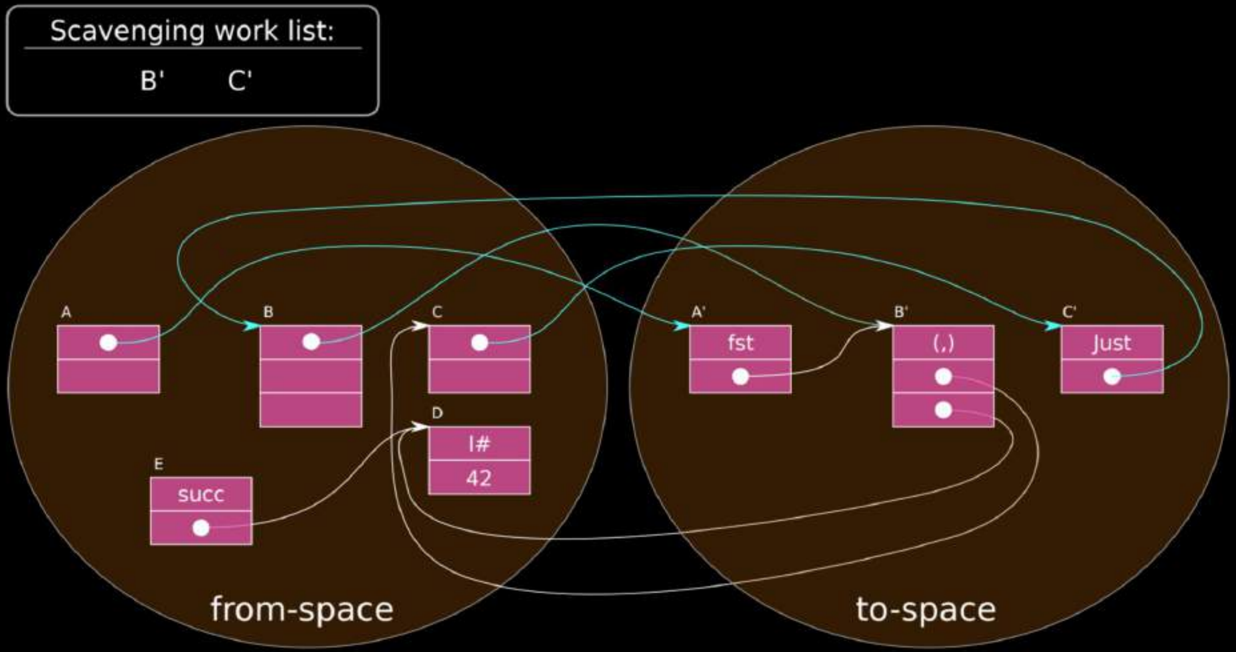

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Evacuate C



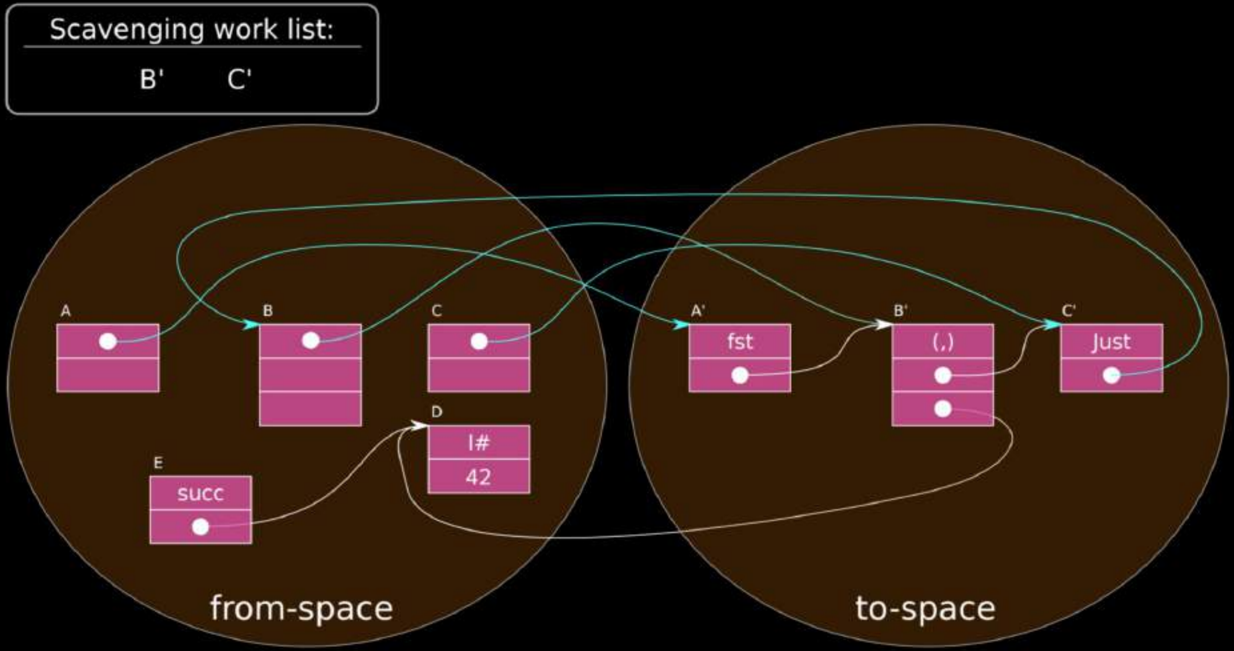

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Evacuate C



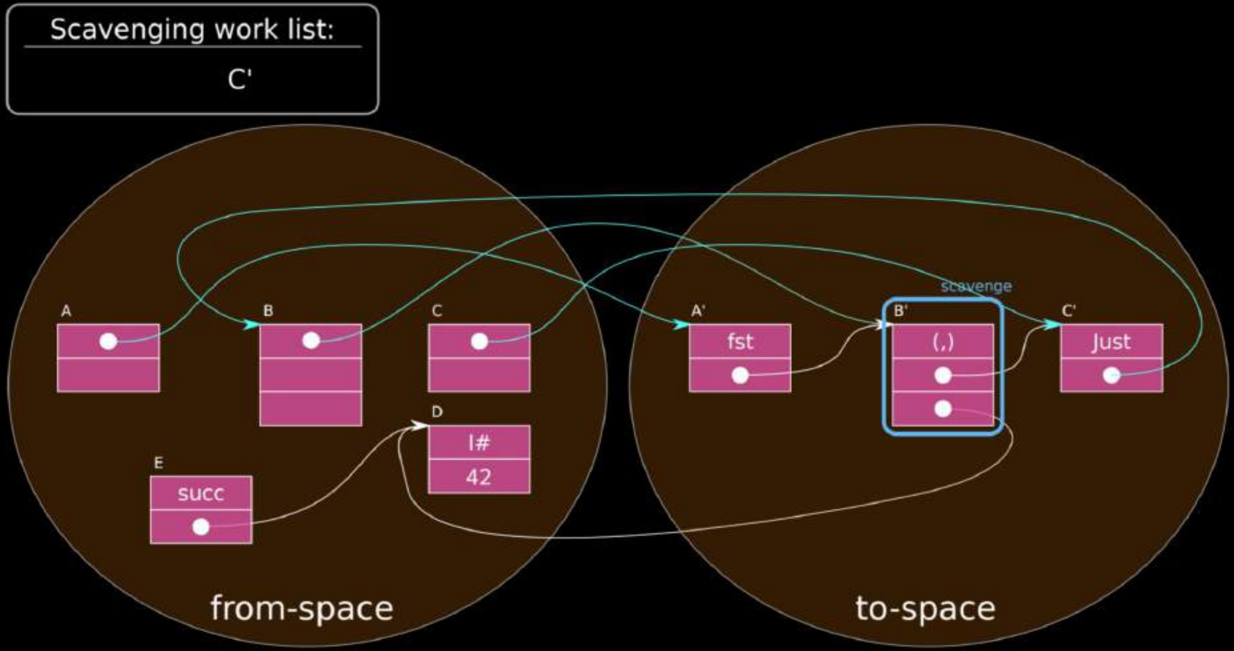

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Evacuate C



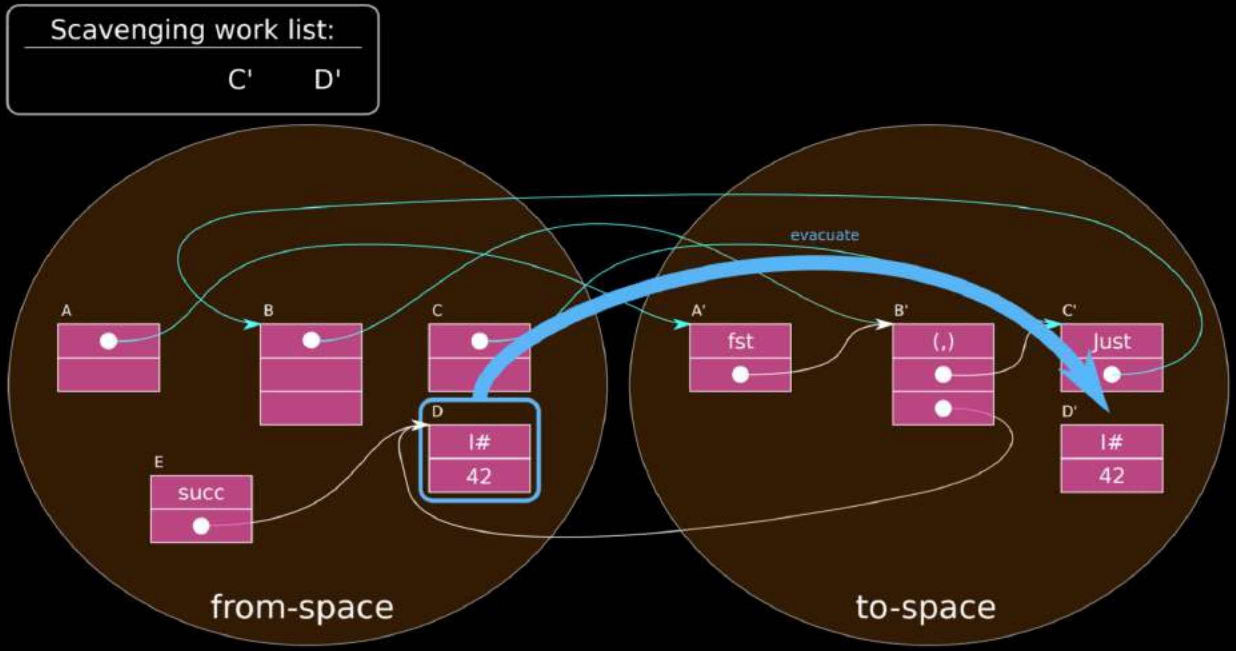

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Scavenging B (cont'd)



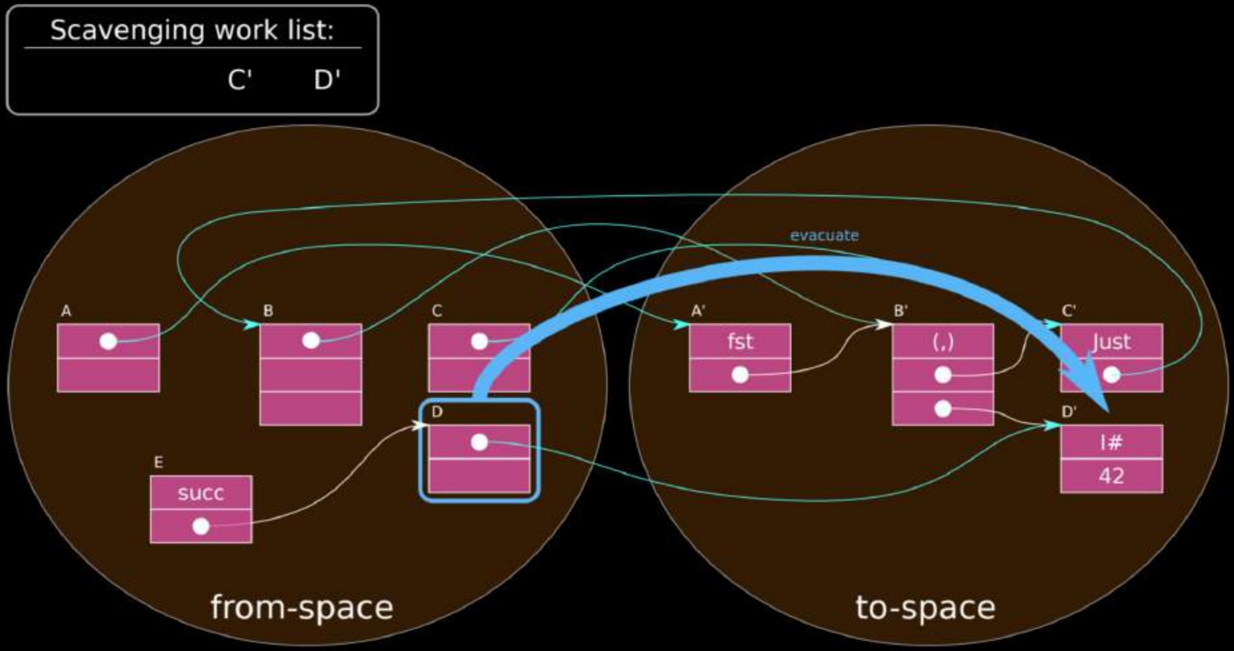

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Evacuate D



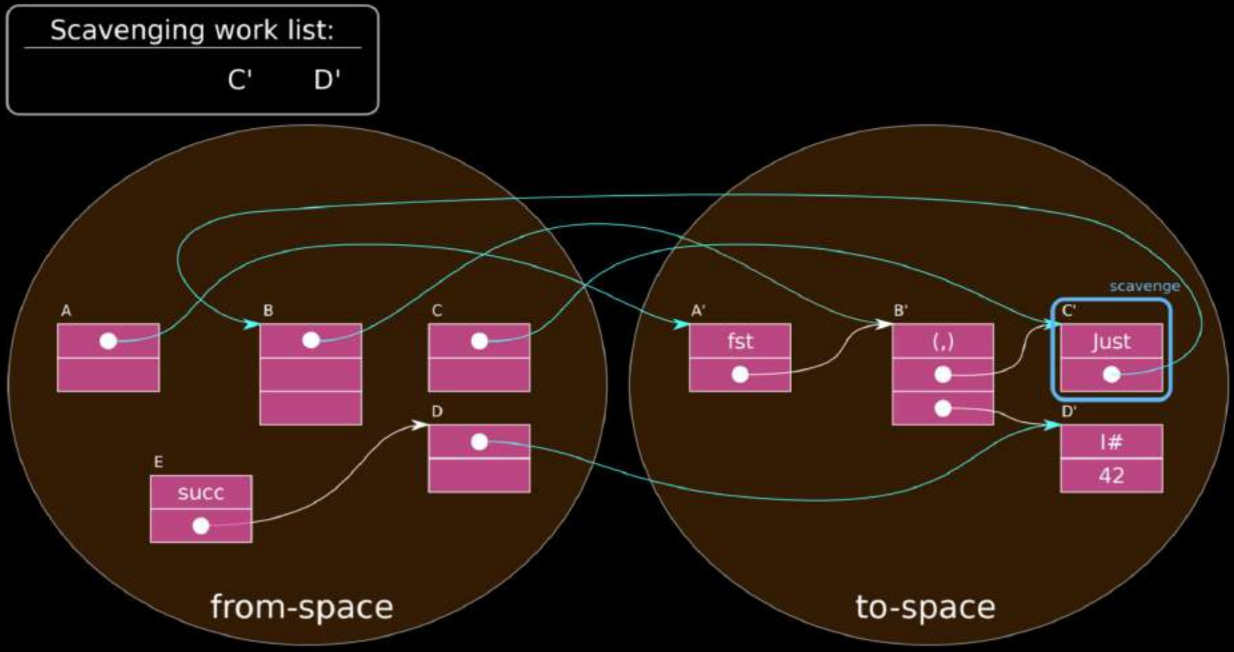

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Evacuate D



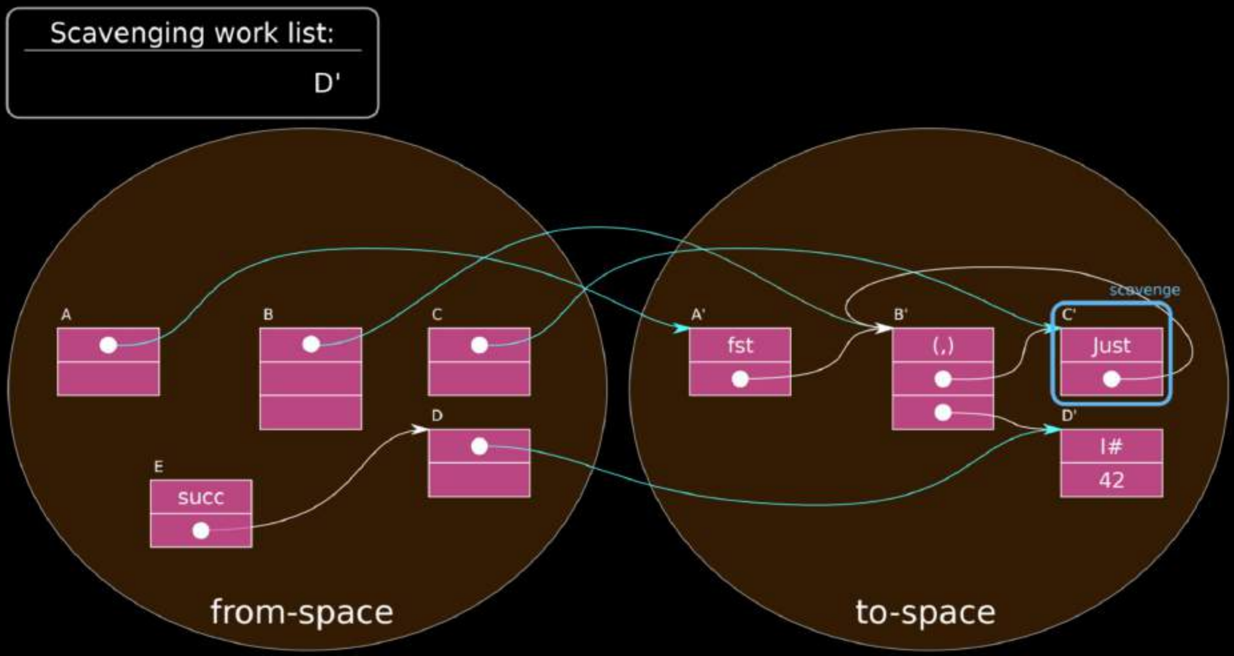

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Scavenging C



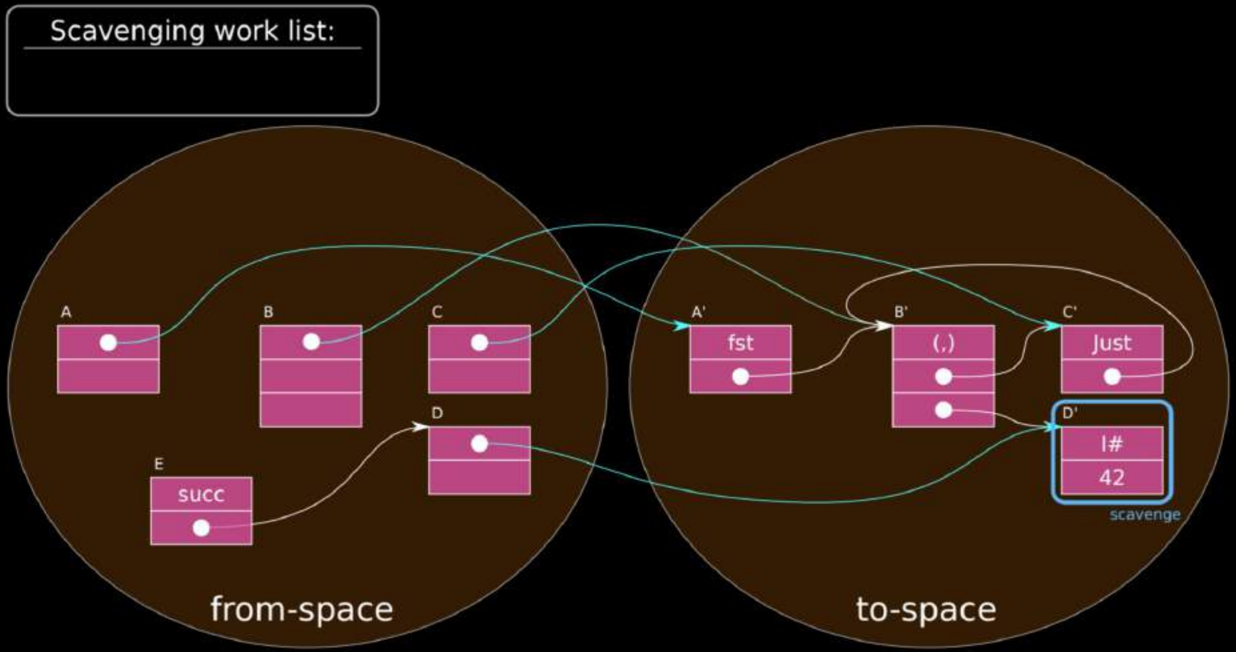

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Scavenging C



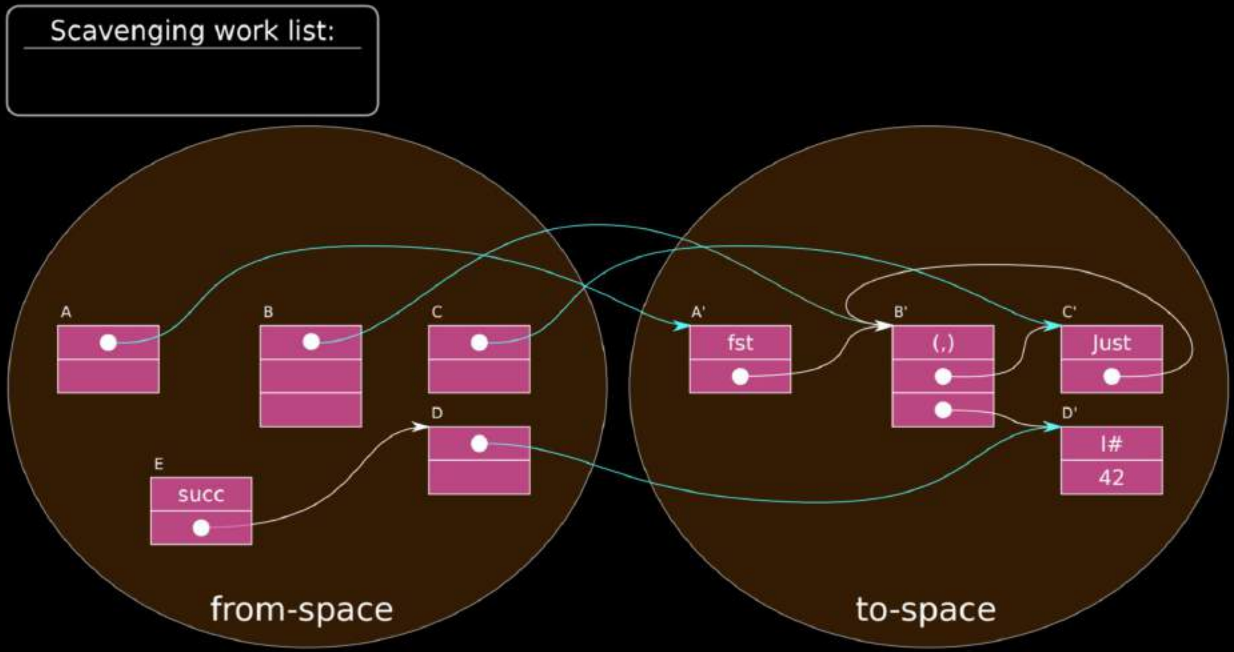

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Scavenging D



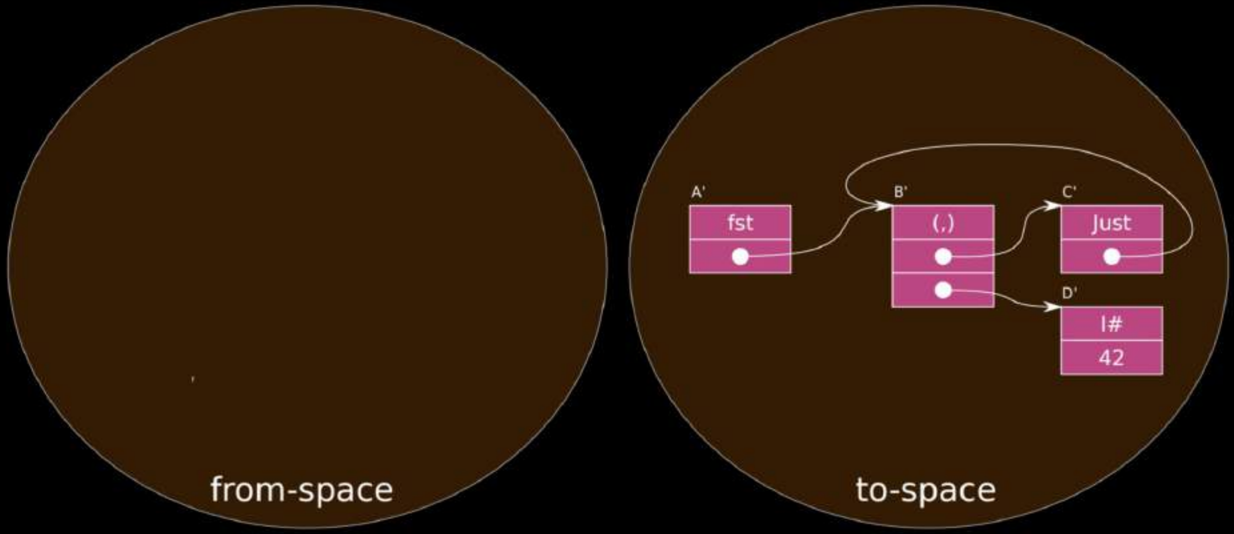

Slides from Ben Gamari's talk 2018

# Moving garbage collection: Finished!




Slides from Ben Gamari's talk 2018

# Moving garbage collection: Finished!




Slides from Ben Gamari's talk 2018

# Actual (Generational) GHC Garbage Collector

- Several “generations” of items
- Generation 0 is collected often
- Heap objects get *promoted* to higher generations
- Generation 1 is collected when gen.0 GC does not reclaim enough space

- In practice: usually 2 generations, Gen 0 split into 2 steps.

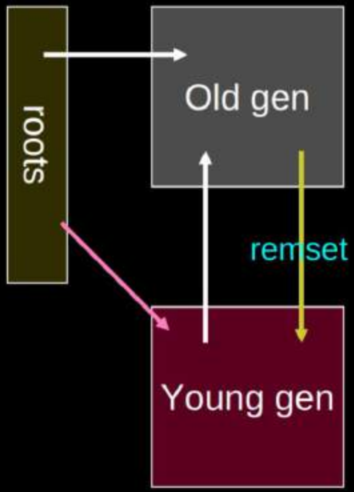
# Generational GC

**Weak generational hypothesis**

- "Most objects die young" [Ungar, 1984]
- Common for 80-95% objects to die before a further MB of allocation.

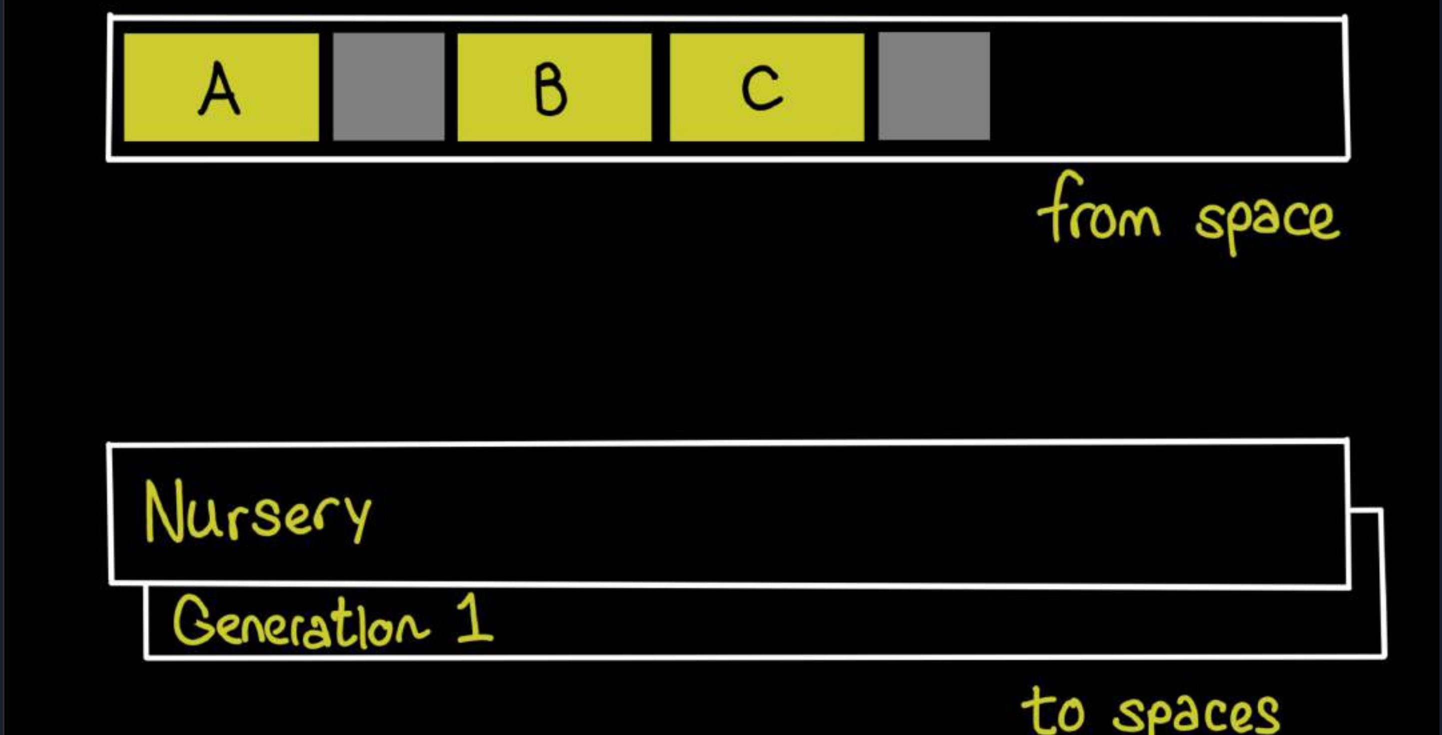
**Strategy**:

- Segregate objects *by age* into **generations (regions of the heap).**
- Collect different generations at different frequencies.
  - so need to "remember" pointers that cross generations.
- Concentrate on the nursery generation to reduce pause times.
  - full heap collection pauses 5-50x longer than nursery collections.

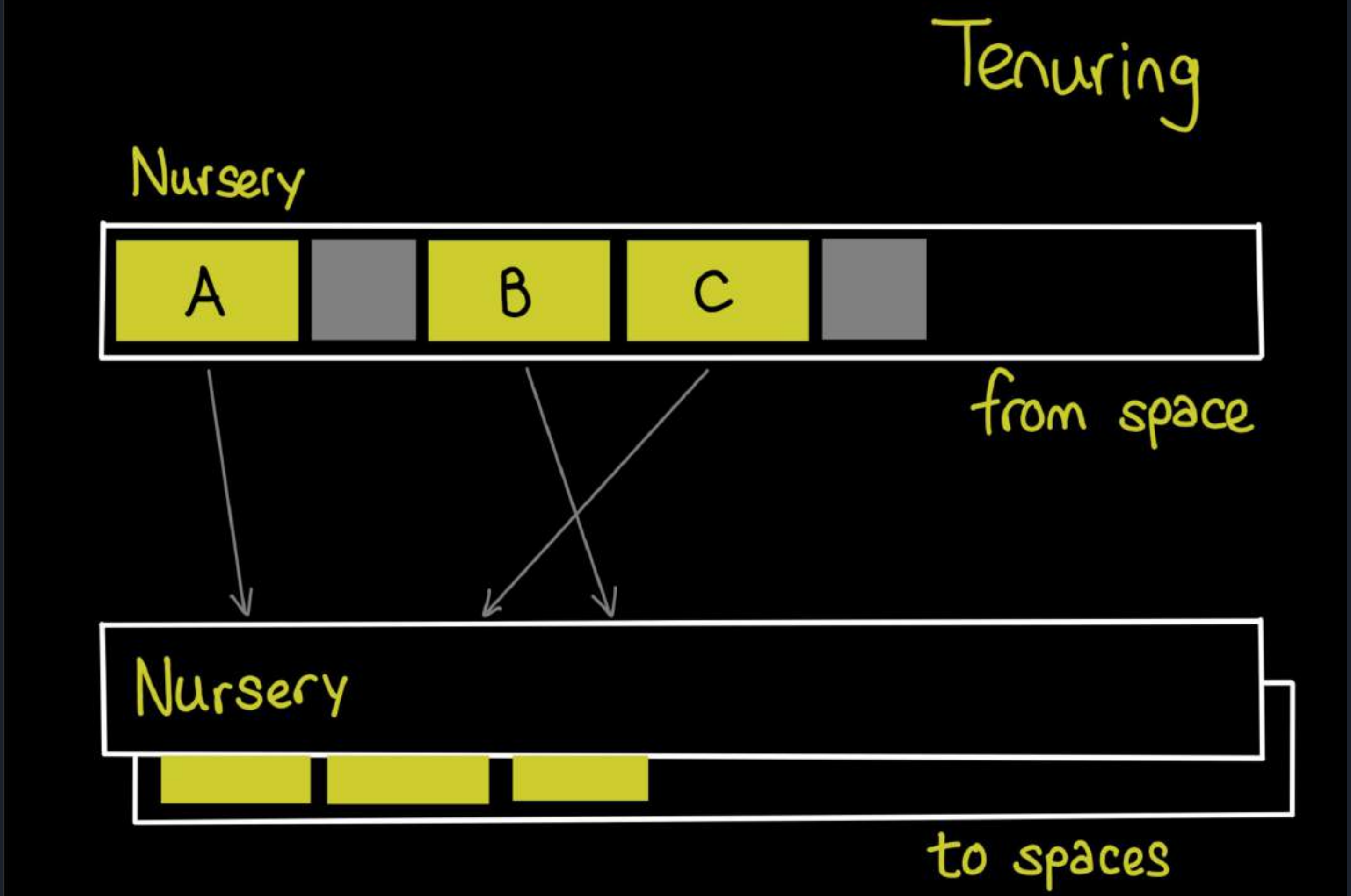




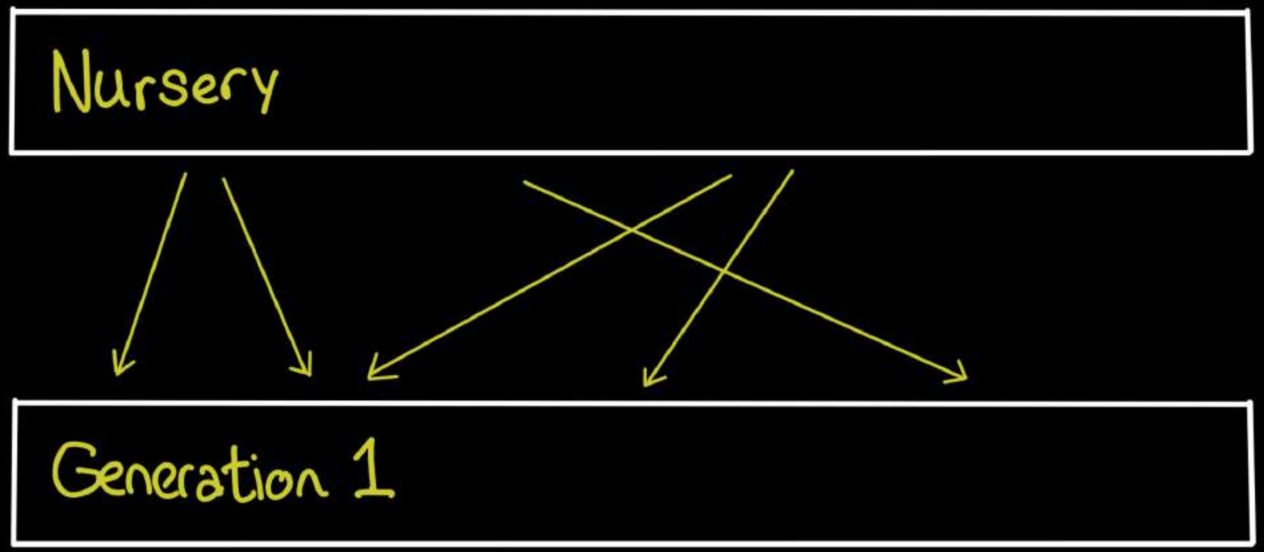

Slides from Richard Jones 2010



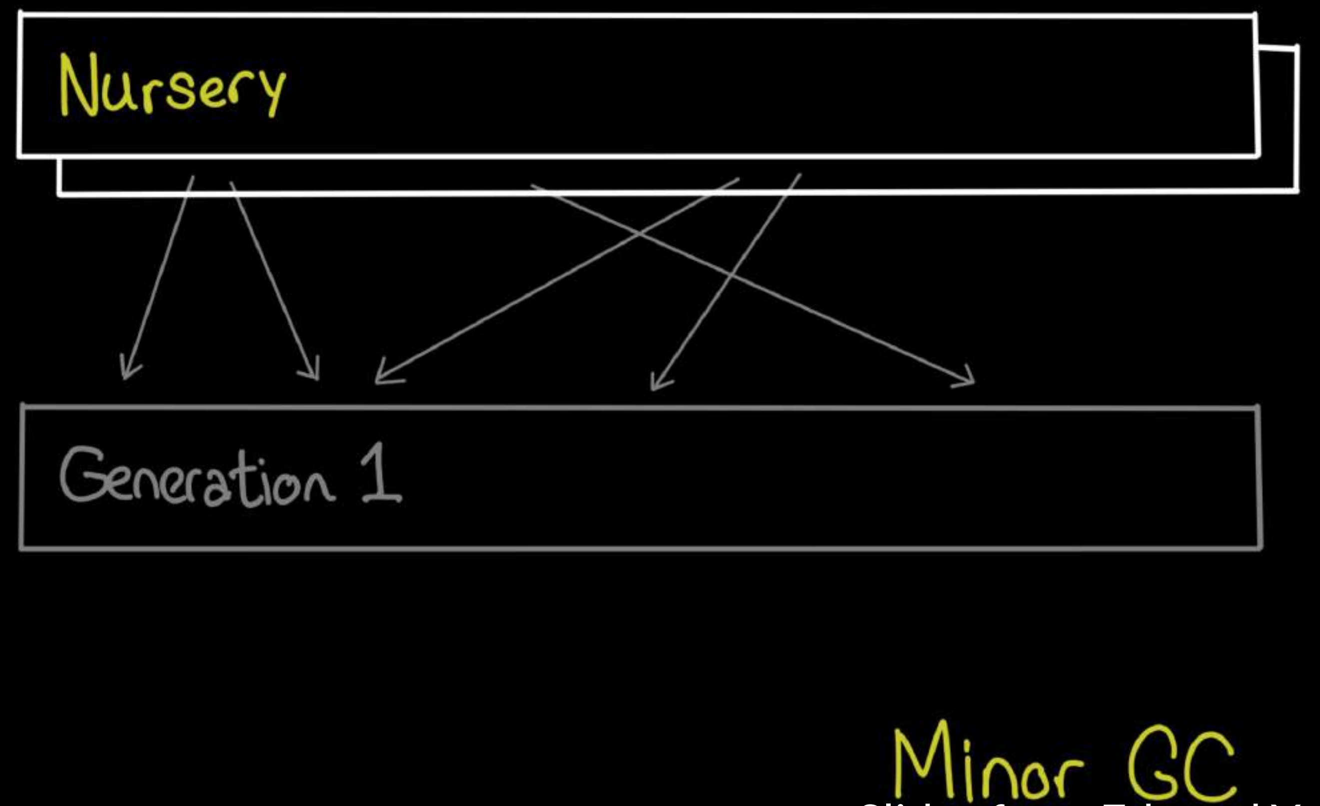

Slides from Edward Yang 2015




Slides from Edward Yang 2015




Slides from Edward Yang 2015




Slides from Edward Yang 2015

```
mk_exit()
    entry:
        Hp = Hp + 16;
        if (Hp > HpLim) goto gc;

        v::I64 = I64[R1] + 1;

        I64[Hp - 8] = GHC_Types_I_con_info;
        I64[Hp + 0] = v::I64;

        R1 = Hp;
        Sp = Sp + 8;
        jump (I64[Sp + 0]) ();

    gc: HpAlloc = 16;
        jump stg_gc_enter_1 ();
}
```

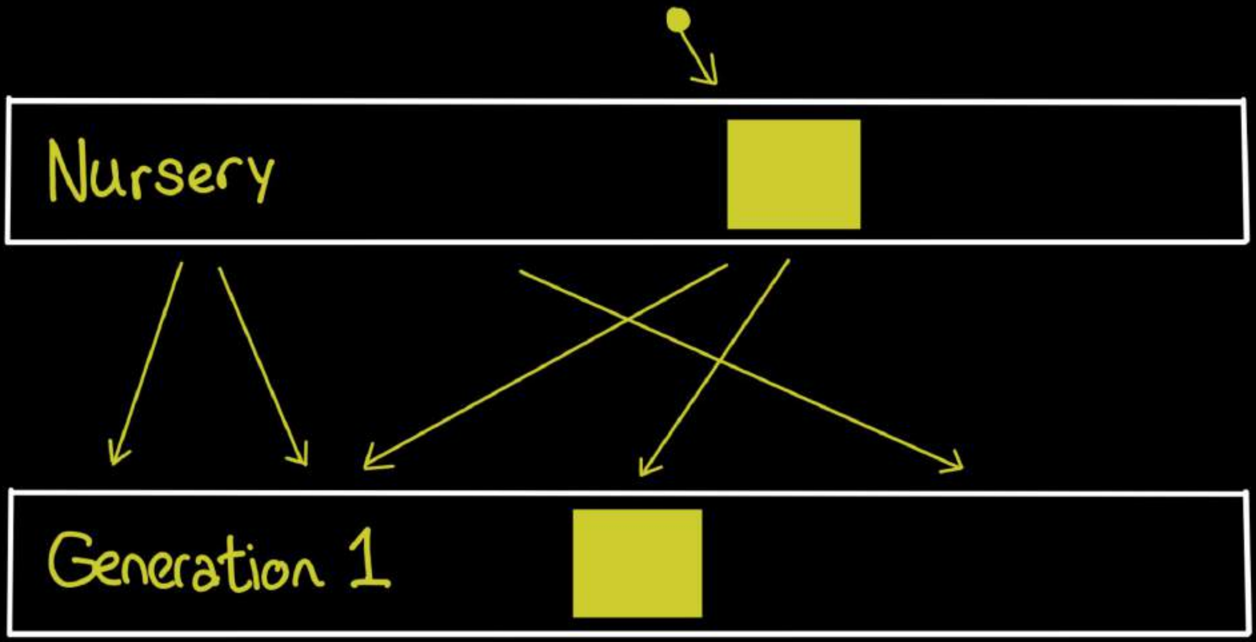
Slides from Edward Yang 2015

# What about Purity?

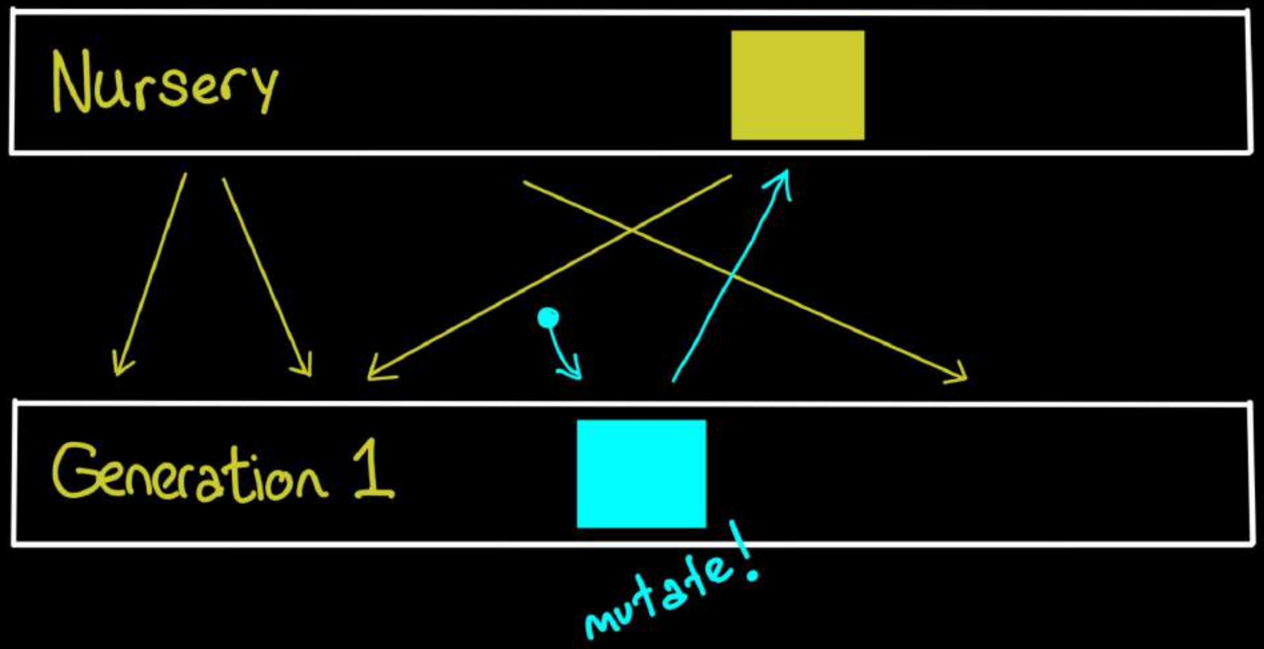
→ Write Barriers

→ Parallel Garbage Collection

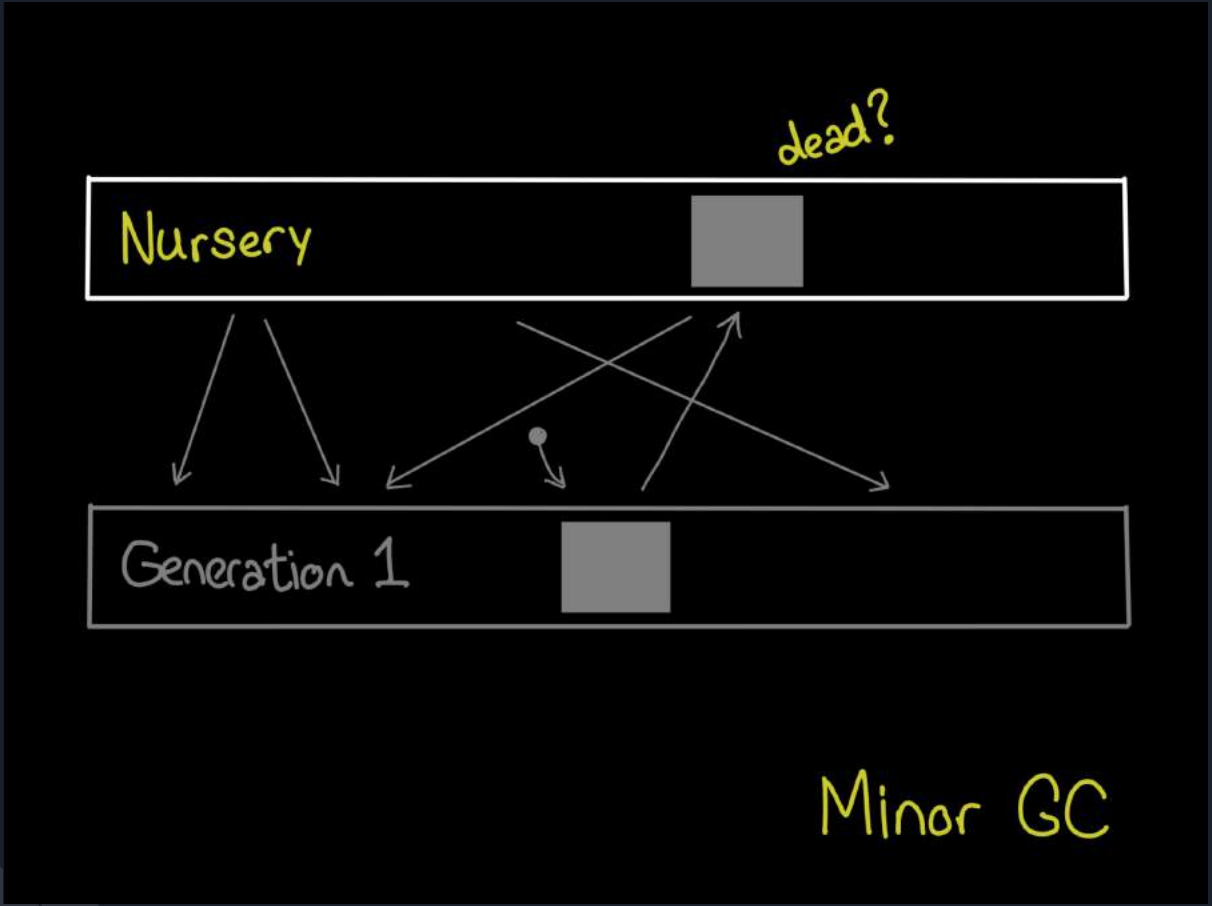
Slides from Edward Yang 2015



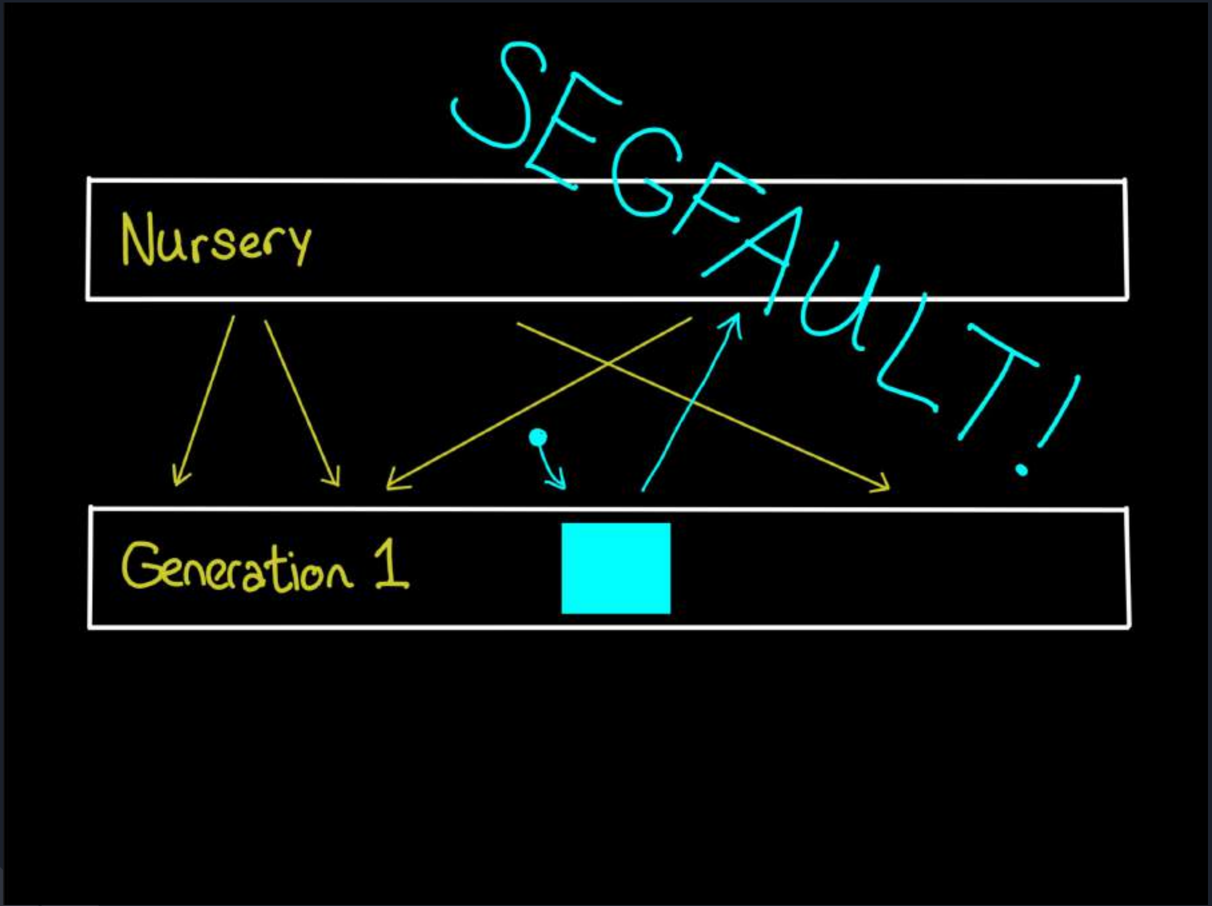

Slides from Edward Yang 2015



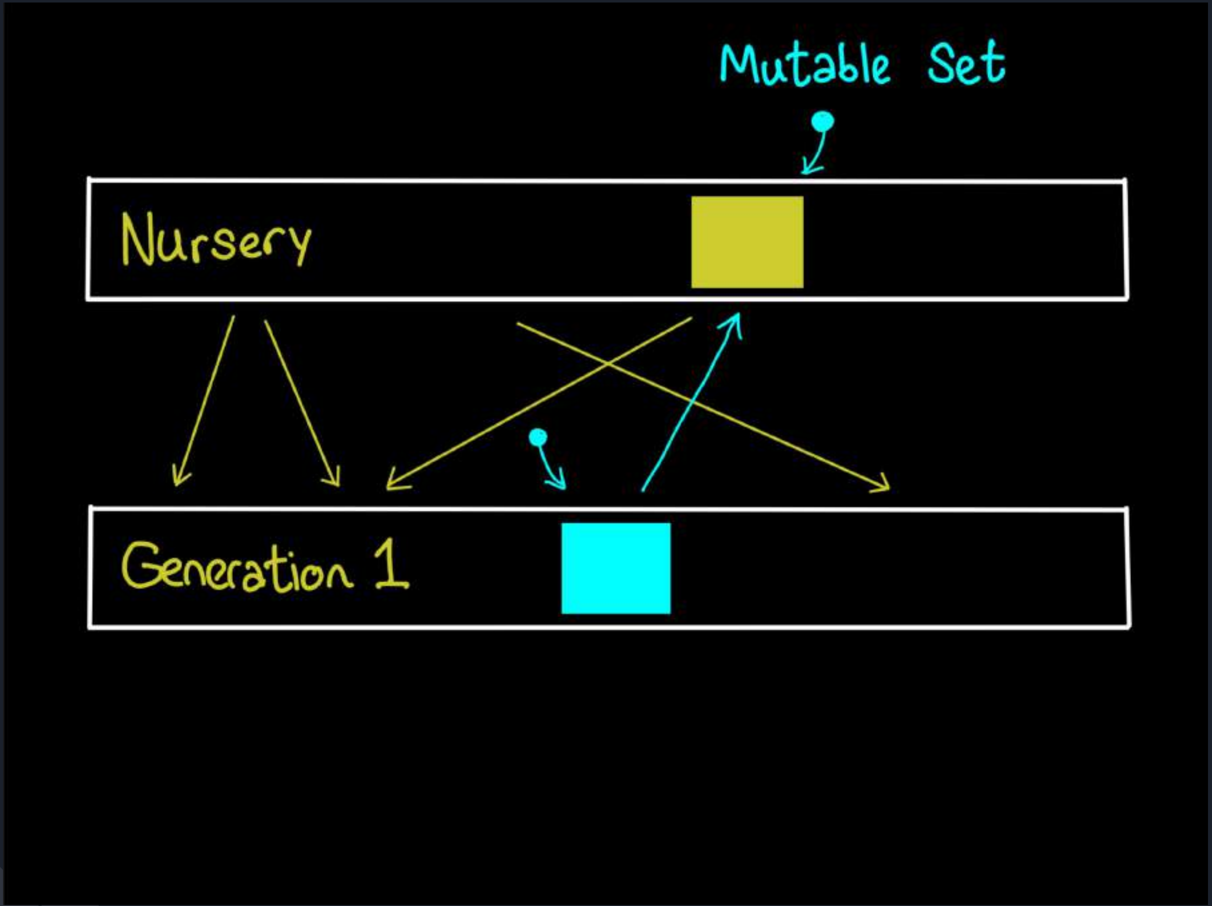

Slides from Edward Yang 2015




Slides from Edward Yang 2015




Slides from Edward Yang 2015

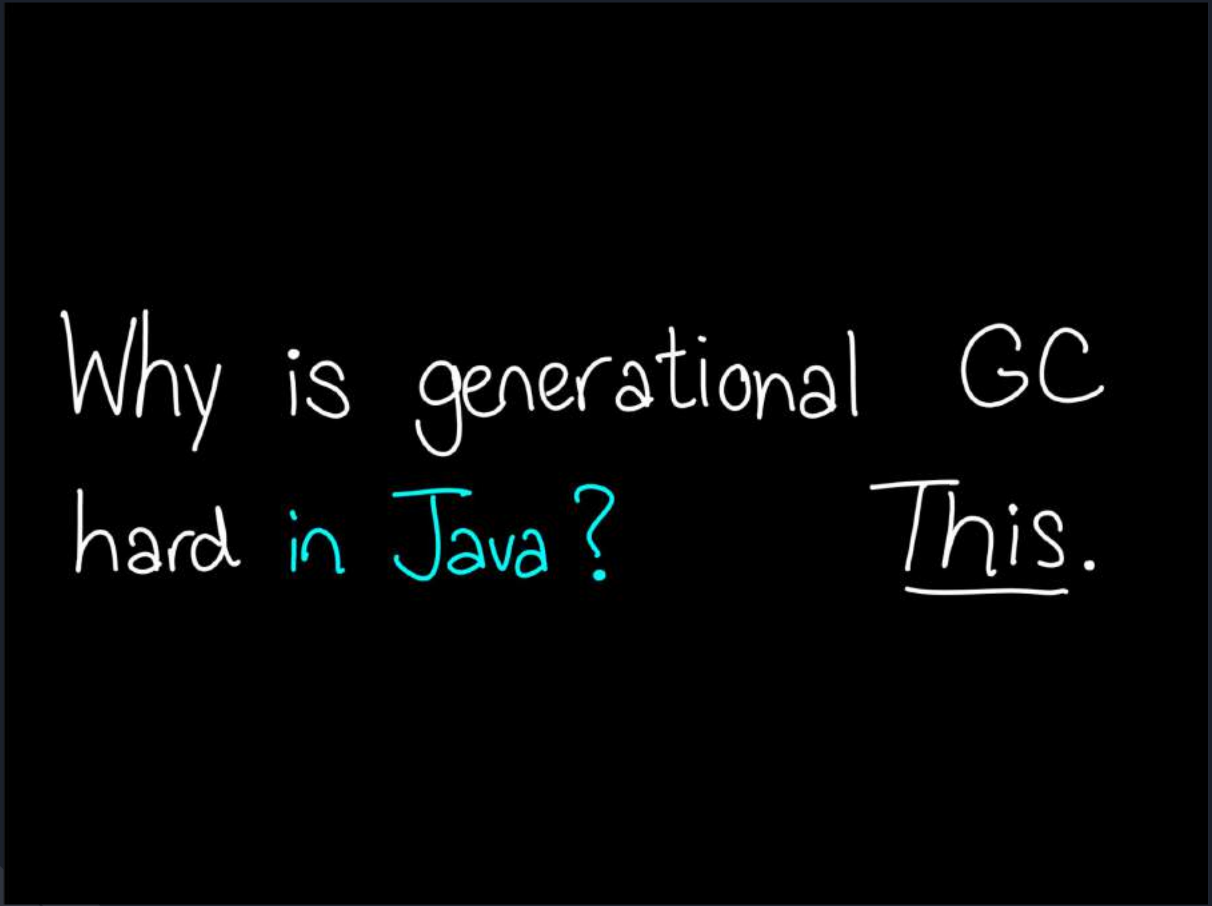



Slides from Edward Yang 2015

Why is generational GC hard?

**This.**

Slides from Edward Yang 2015

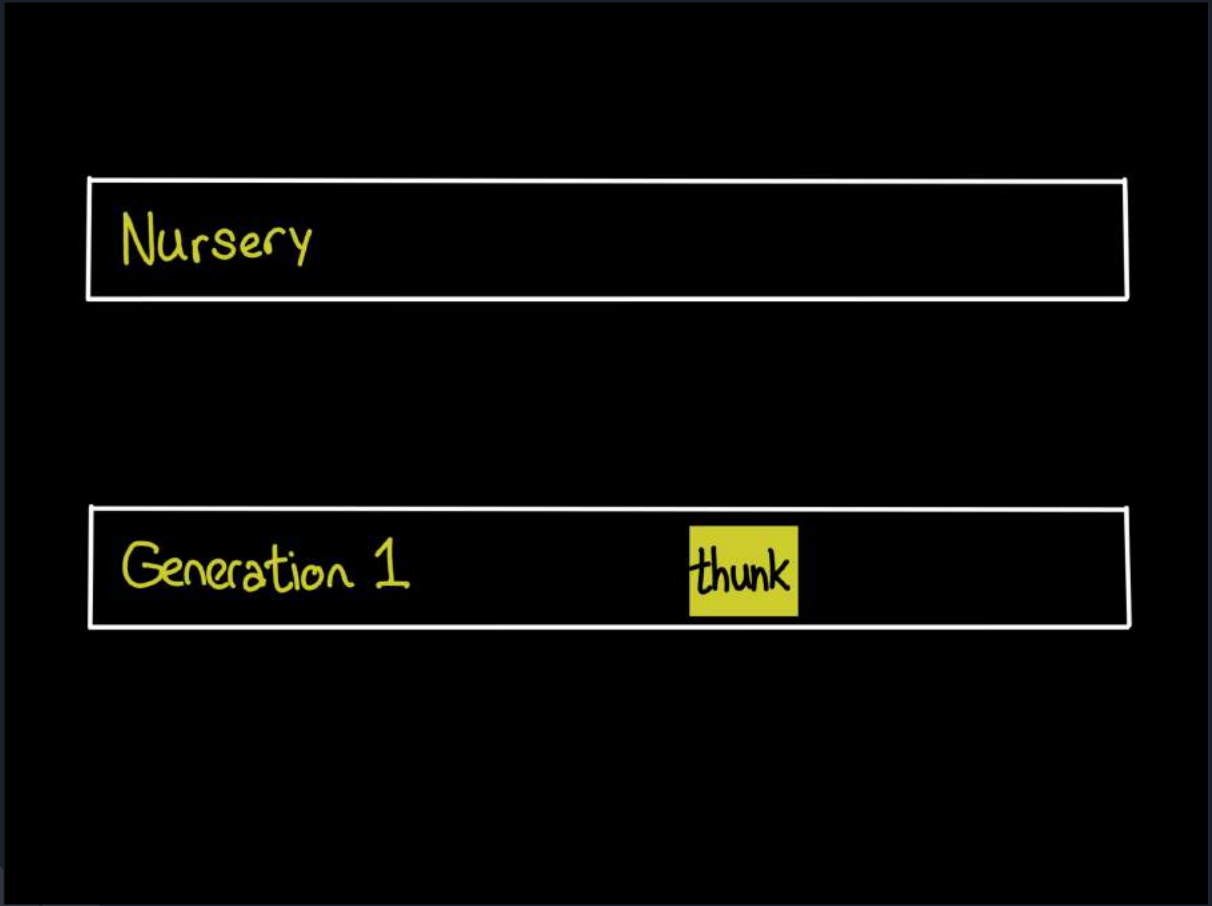



Slides from Edward Yang 2015

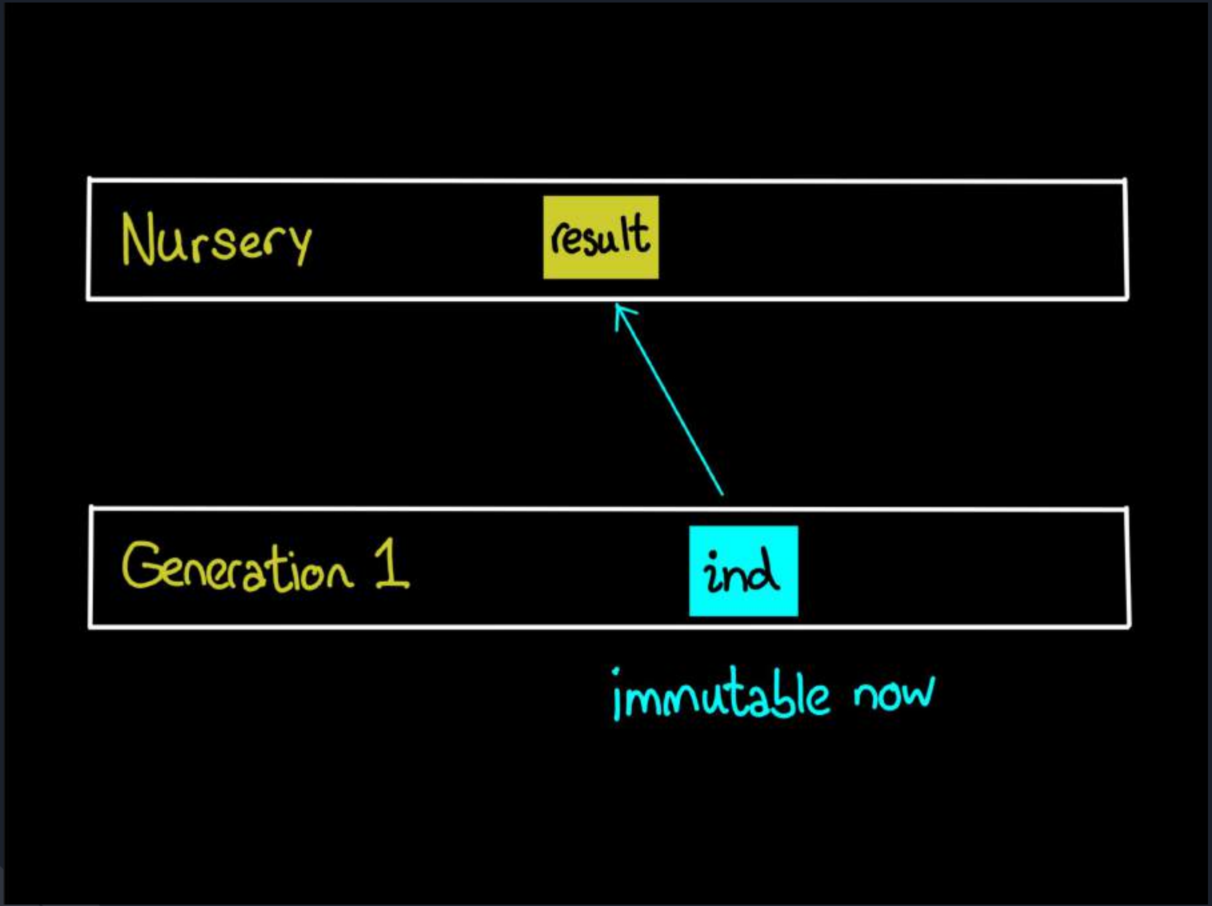
# Purity to the rescue

- Mutation is rare
- IORefs are slow anyway
- Laziness is a special kind of mutation

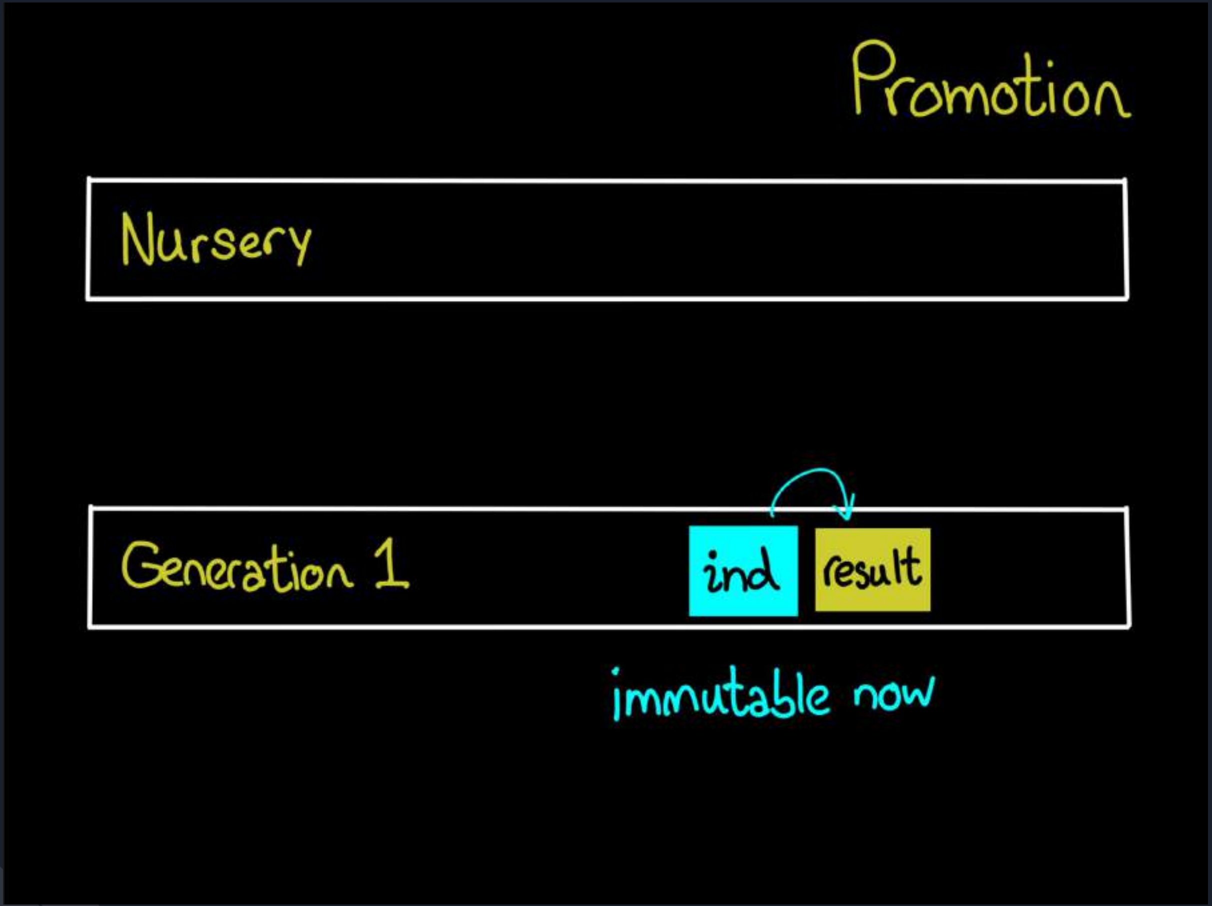
Slides from Edward Yang 2015




Slides from Edward Yang 2015



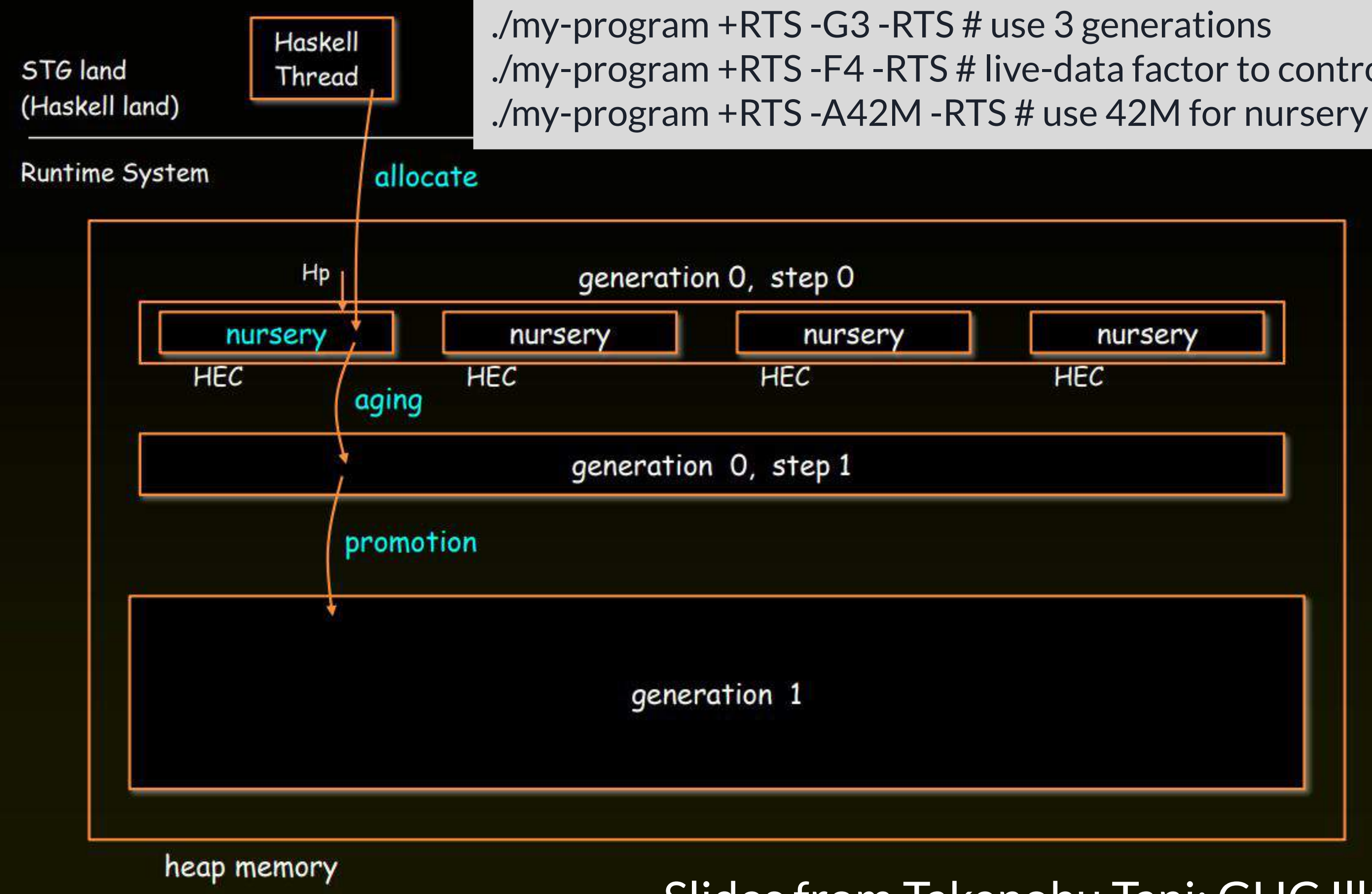

Slides from Edward Yang 2015




Slides from Edward Yang 2015

# GC, nursery, generation, aging, promotion

```
./my-program +RTS -G3 -RTS # use 3 generations
./my-program +RTS -F4 -RTS # live-data factor to control old gen. GC
./my-program +RTS -A42M -RTS # use 42M for nursery (gen0 step0)
```




Slides from Takenobu Tani: GHC Illustrated

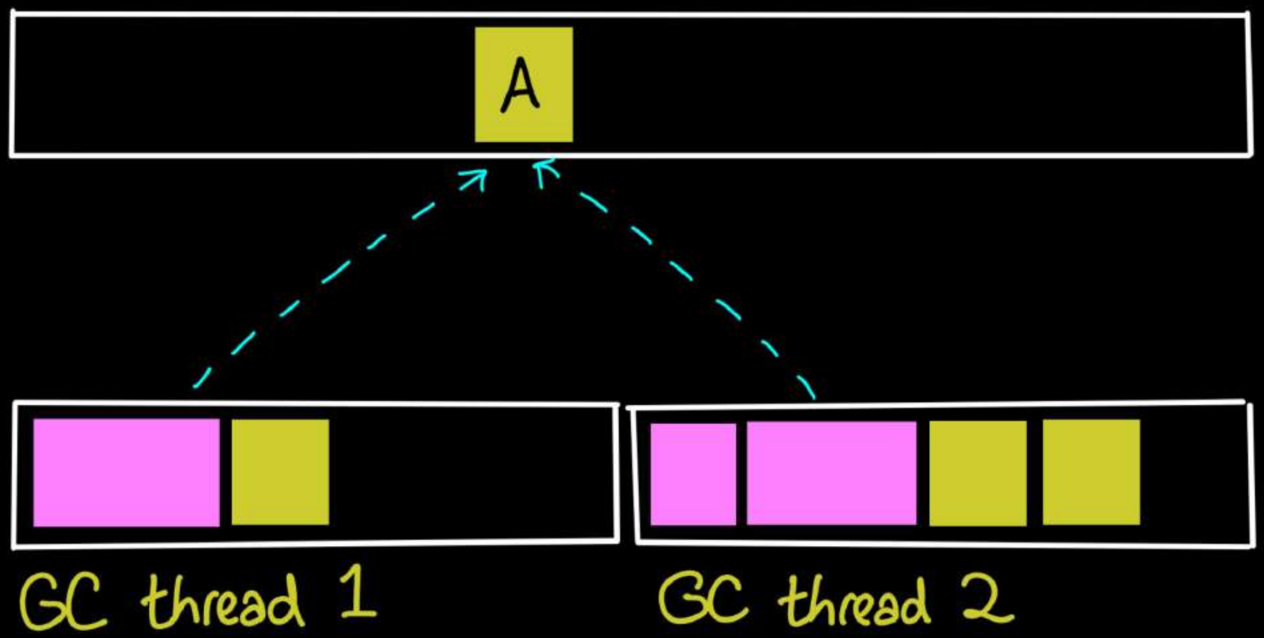
# Parallel GC and Compacting GC

- Parallel GC
  - Copying GC in distinct *blocks* of the heap in parallel
  - Can use load-balancing
  - Still stop-the-world (*not concurrent)*
- Compacting GC:
  - Collect the oldest gen. using a mark-sweep-compact collector
  - May reduce memory pressure for programs with large heaps
  - Still stop-the-world (*not concurrent)*

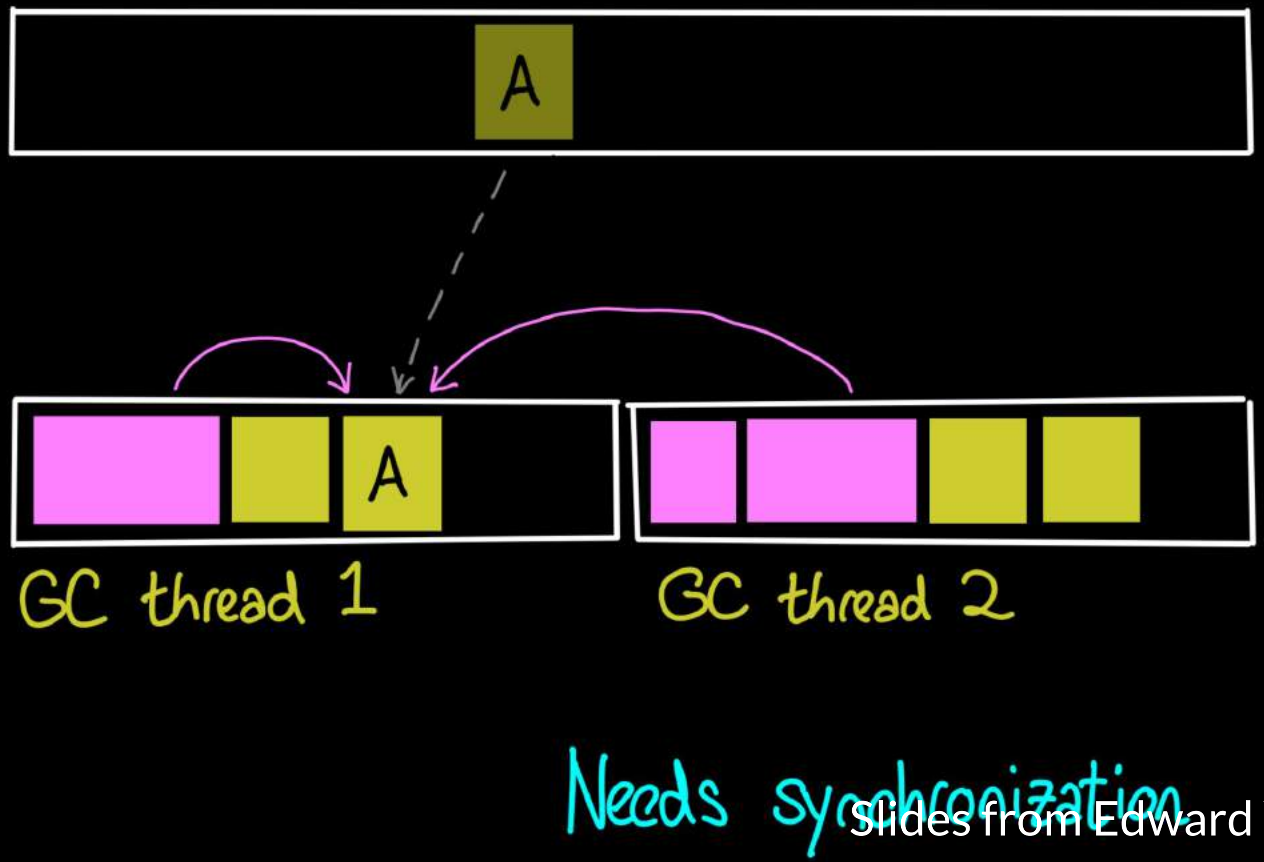
# Parallel GC

Idea: Split heap into blocks, and parallelize the scavenging process

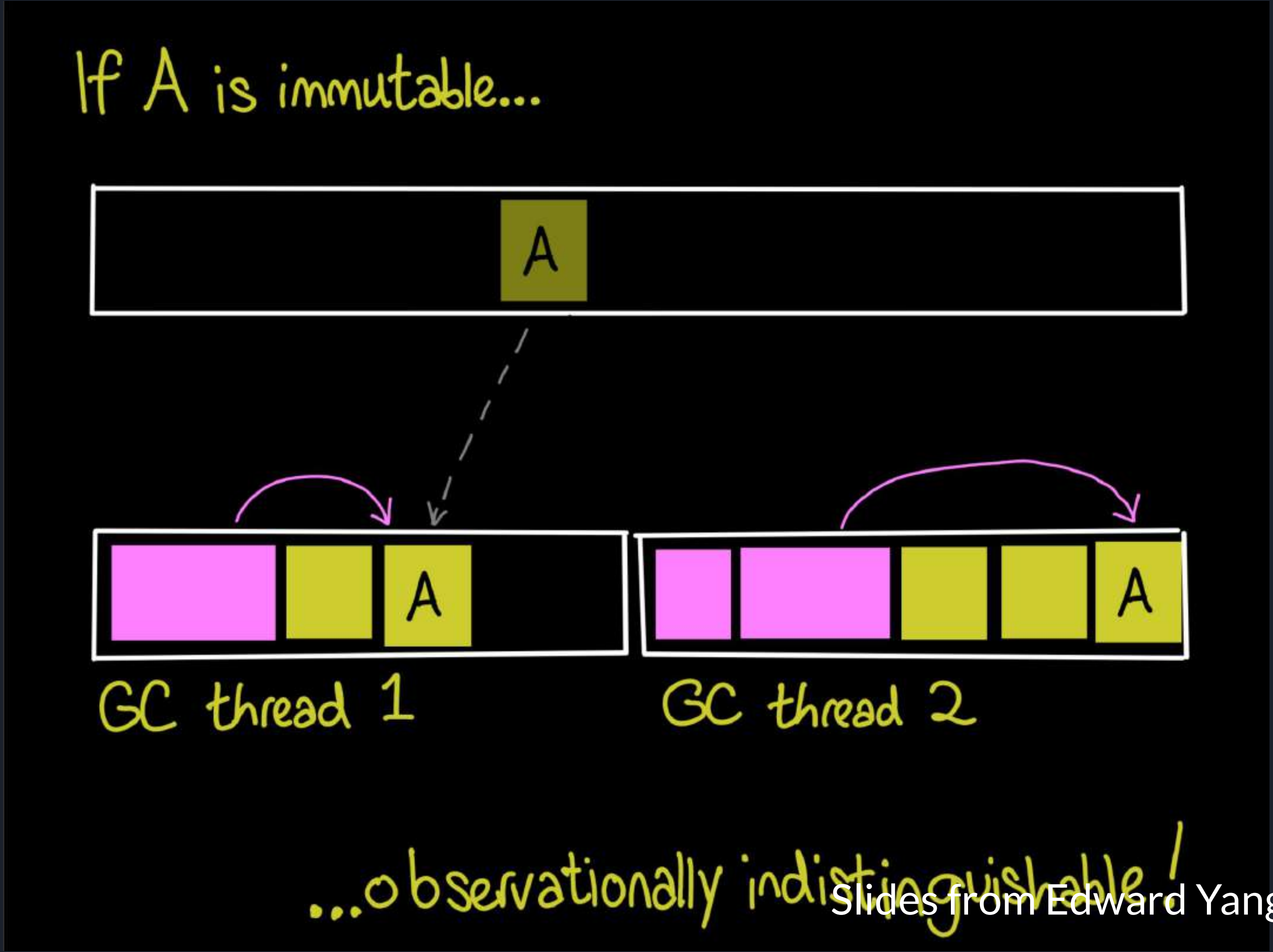
Slides from Edward Yang 2015



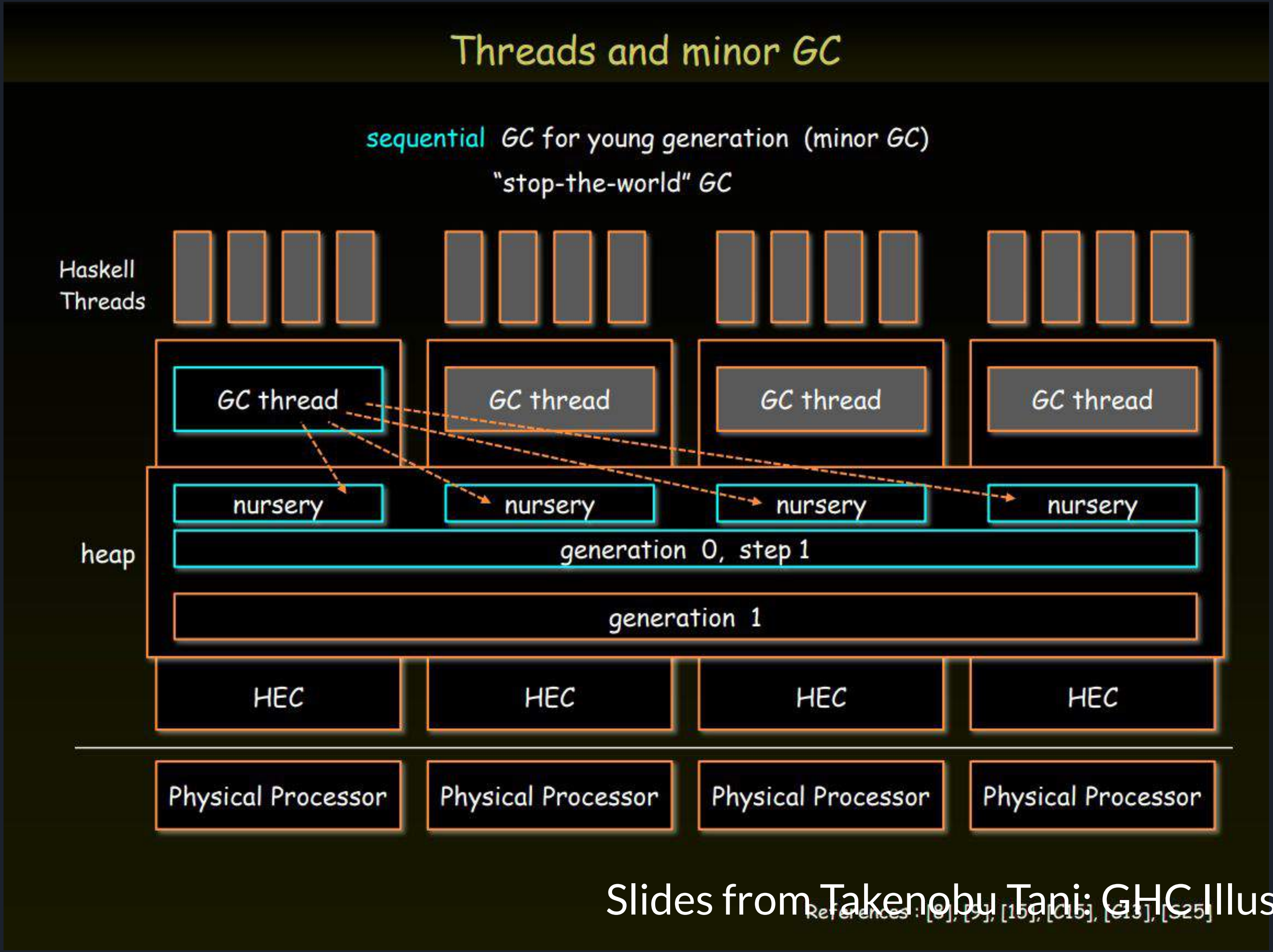

Slides from Edward Yang 2015



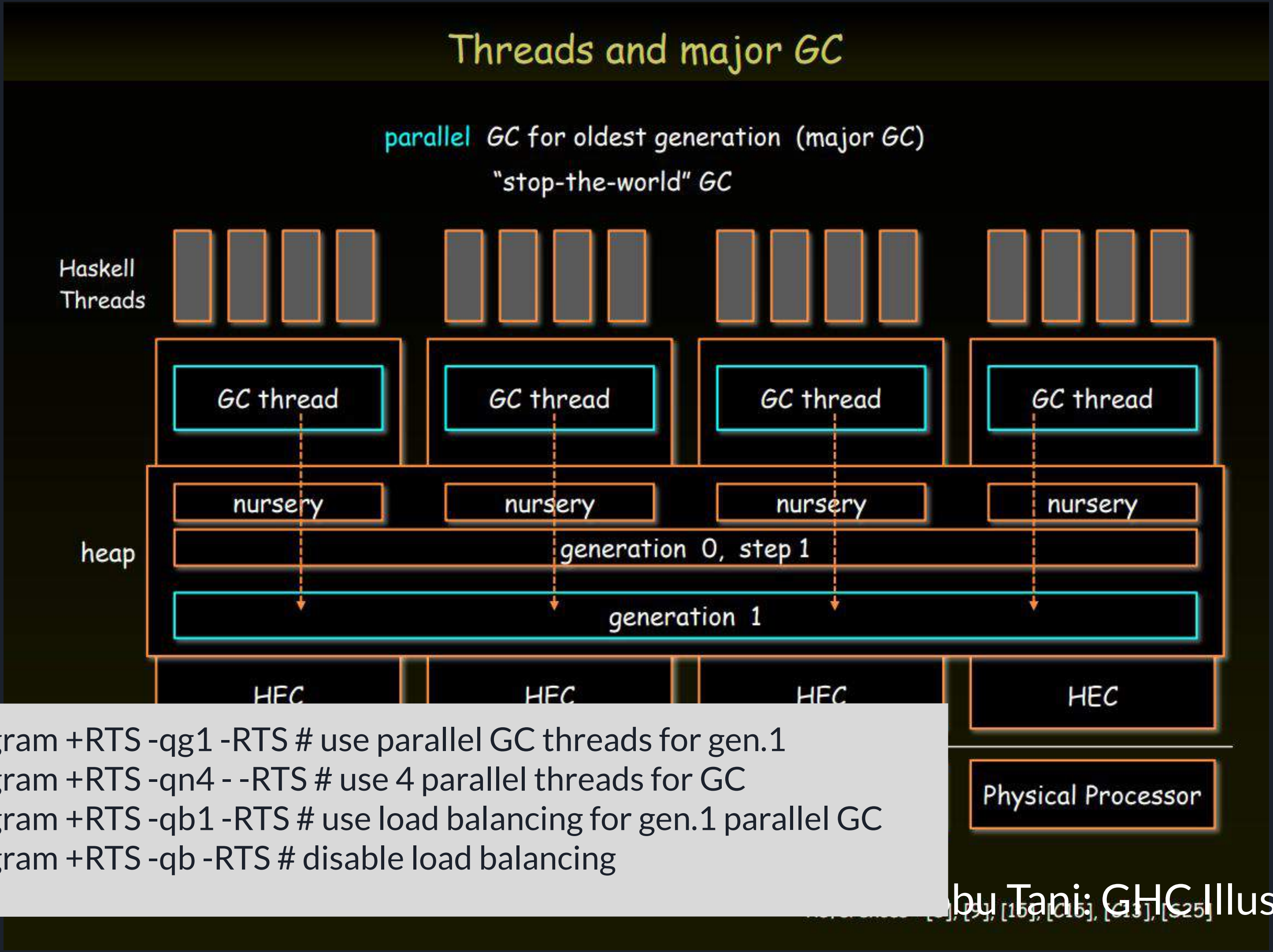

Slides from Edward Yang 2015




Slides from Edward Yang 2015




Slides from Takenobu Tani: GHC Illustrated



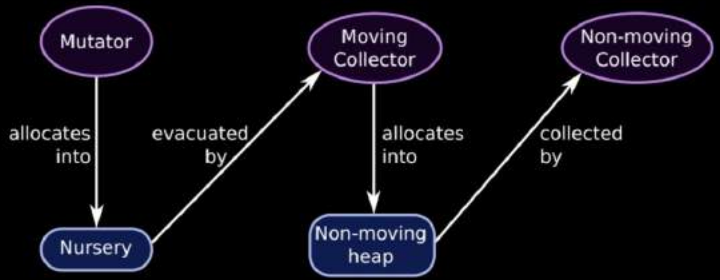

./my-program +RTS -qg1 -RTS # use parallel GC threads for gen.1
./my-program +RTS -qn4 - -RTS # use 4 parallel threads for GC
./my-program +RTS -qb1 -RTS # use load balancing for gen.1 parallel GC
./my-program +RTS -qb -RTS # disable load balancing

bu Tani: GHC Illustrated

# New: Non-Moving GC

- Advantages of Copying GC:
  - Efficient allocation, data locality, no fragmentation
- Disadvantages
  - Cannot run incrementally nor concurrently
  - Cannot collect a fraction of the heap without scanning all of it
  - Global stop of unknown length (latency)
- Controlling the latency:
  - Incremental collection (bounded amount of work)
  - Concurrent collection (while mutators are running)
- *Here*: only for the old(-est) generation

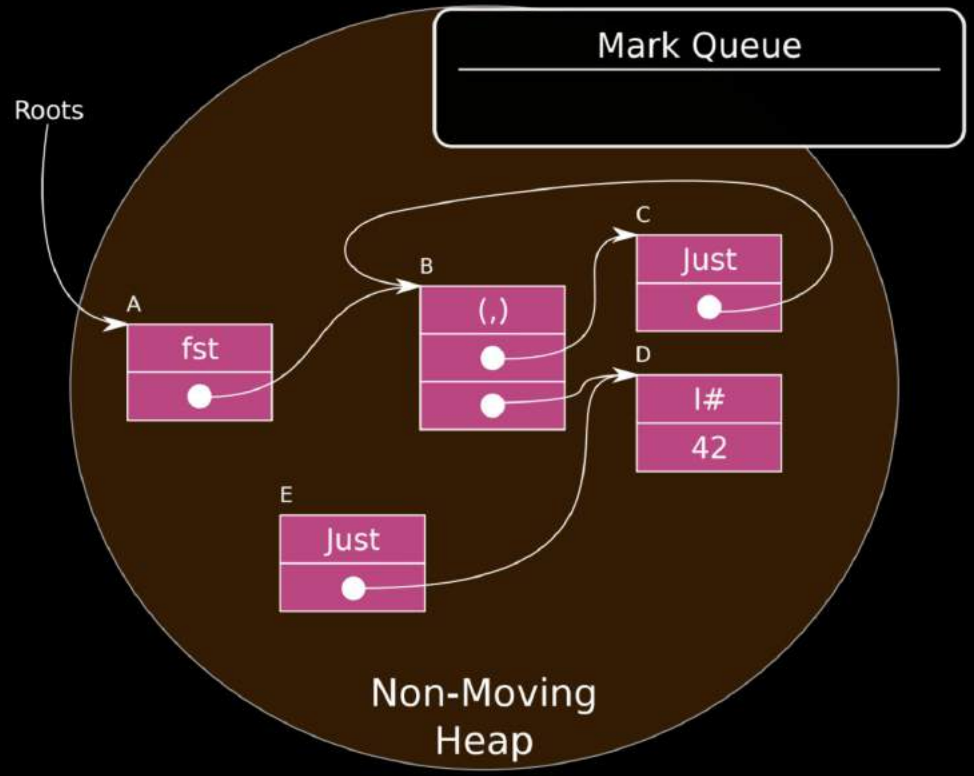
# Hybrid moving/non-moving collector




- 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

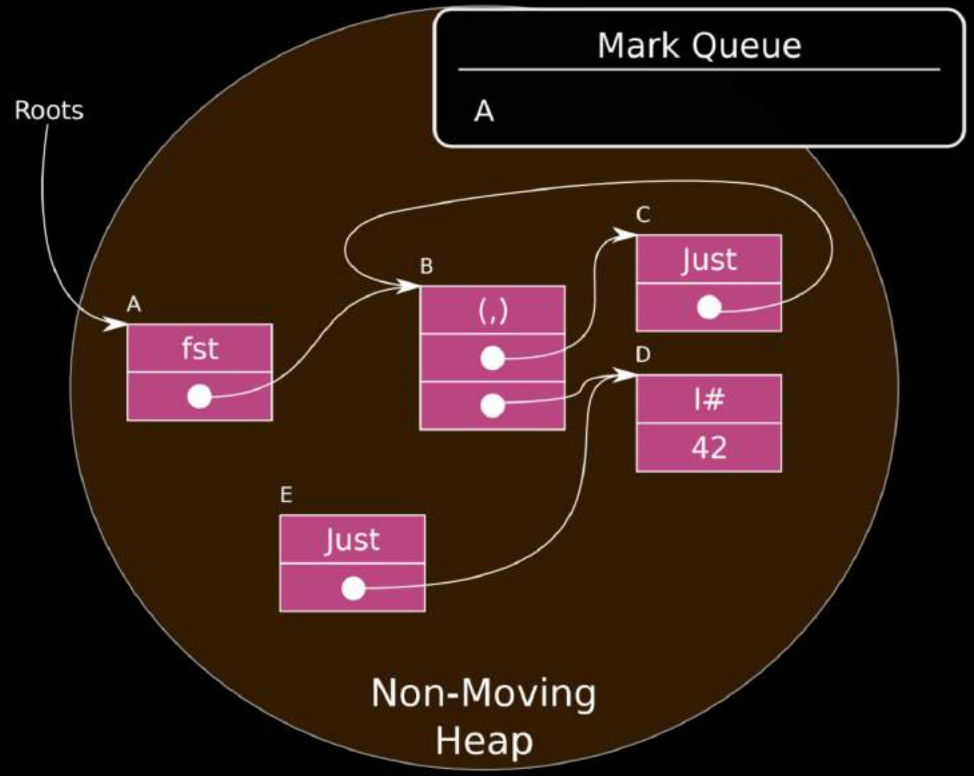
Slides from Ben Gamari 2018

# Mark & Sweep Garbage Collection



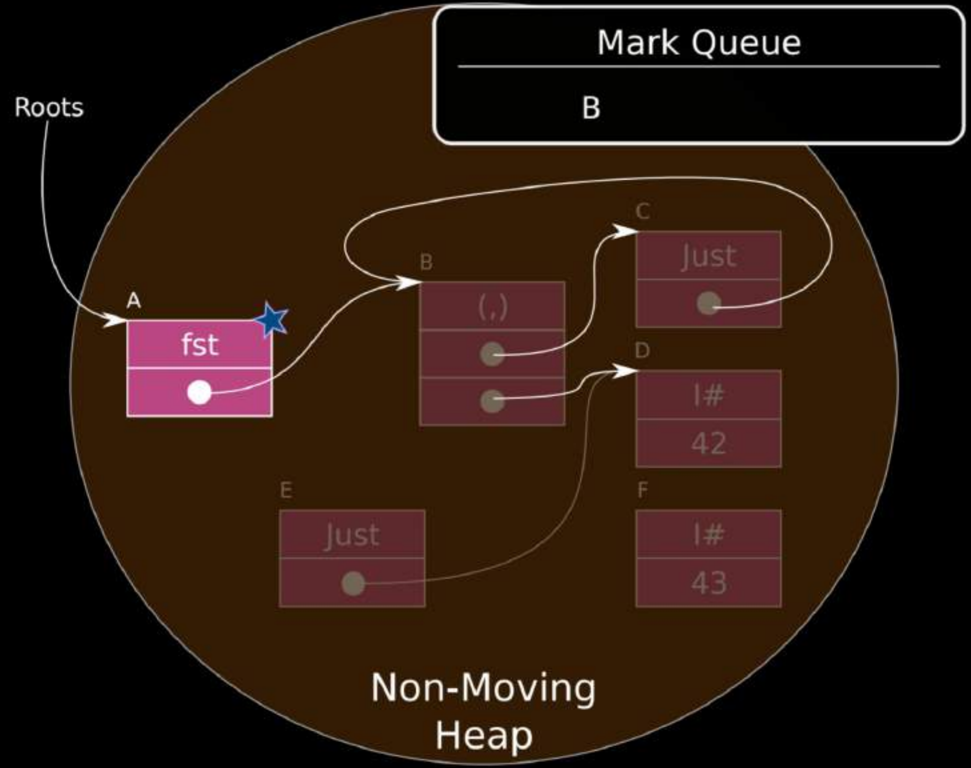

Slides from Ben Gamari 2018

# Mark & Sweep Garbage Collection: Collect roots



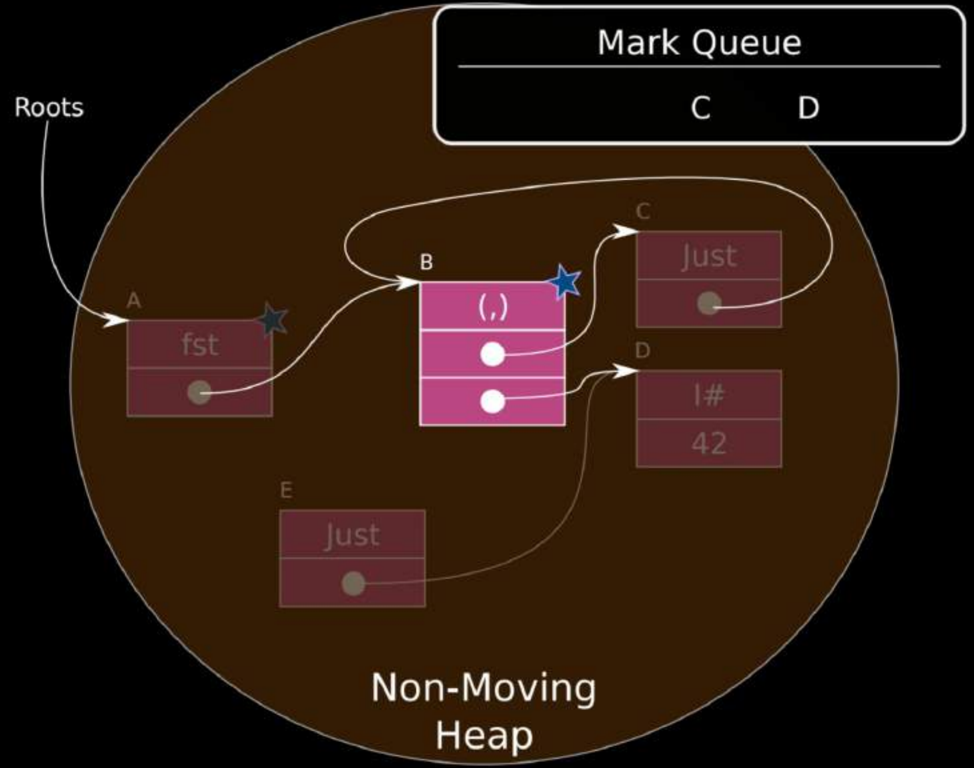

Slides from Ben Gamari 2018

# Mark & Sweep Garbage Collection: Marking (A)



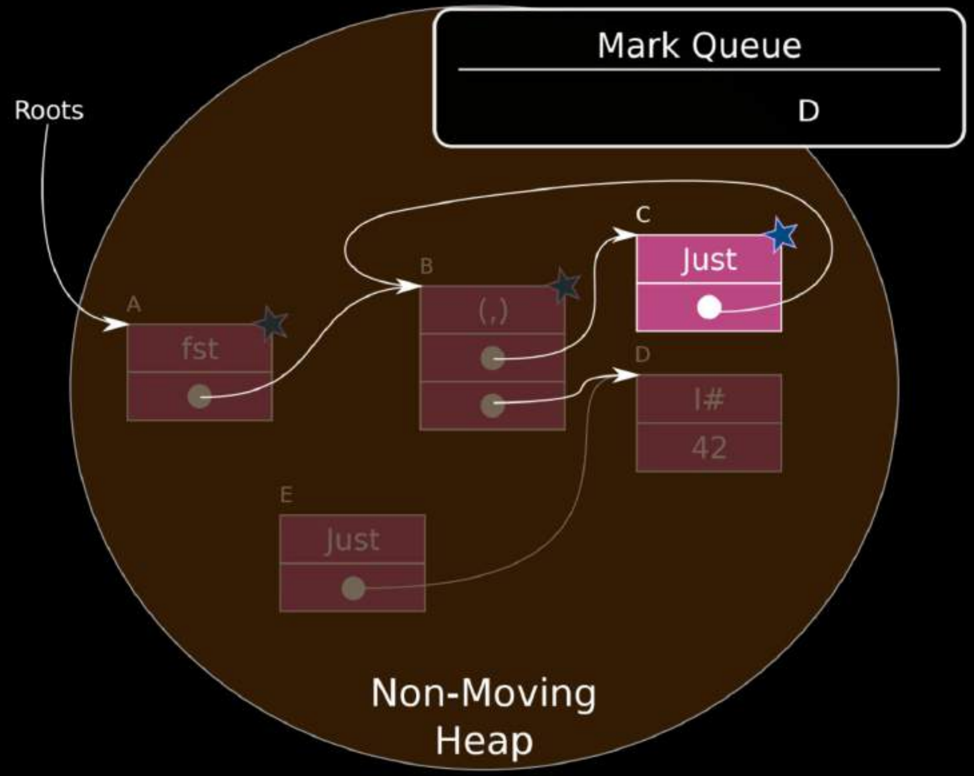

Slides from Ben Gamari 2018

# Mark & Sweep Garbage Collection: Marking (B)



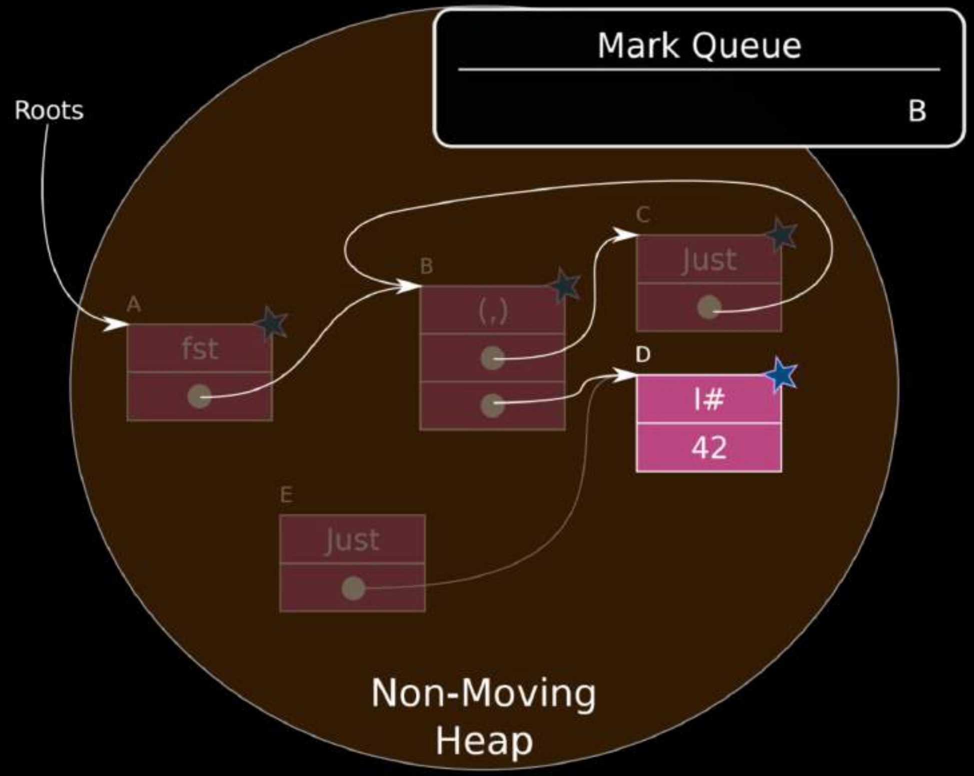

Slides from Ben Gamari 2018

# Mark & Sweep Garbage Collection: Marking (C)



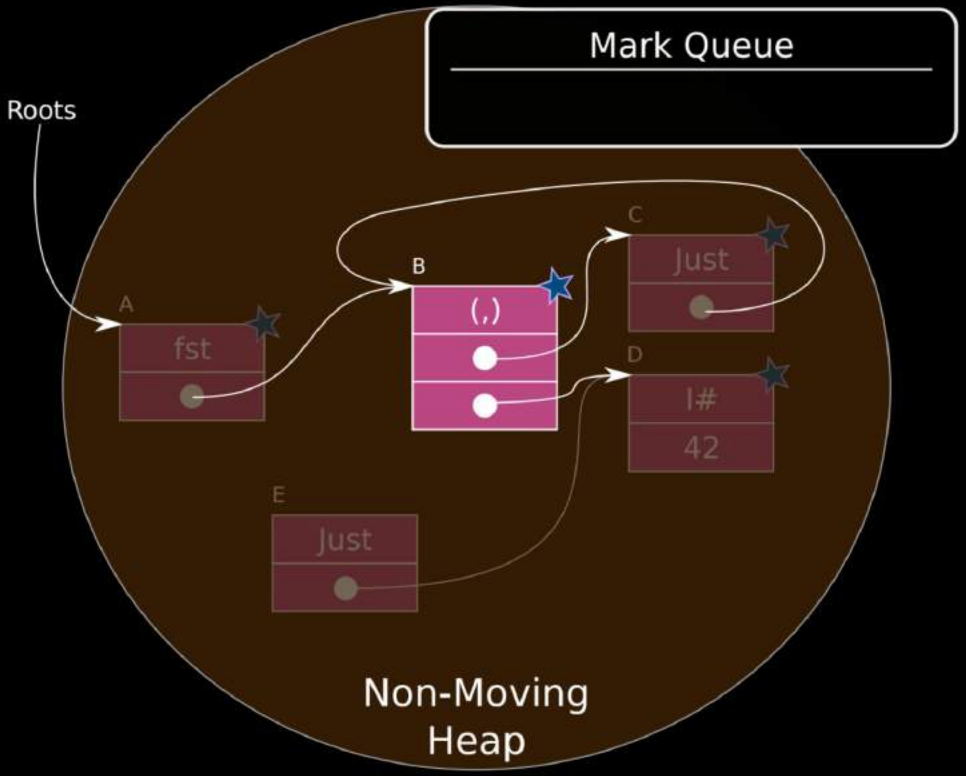

Slides from Ben Gamari 2018

# Mark & Sweep Garbage Collection: Marking (D)



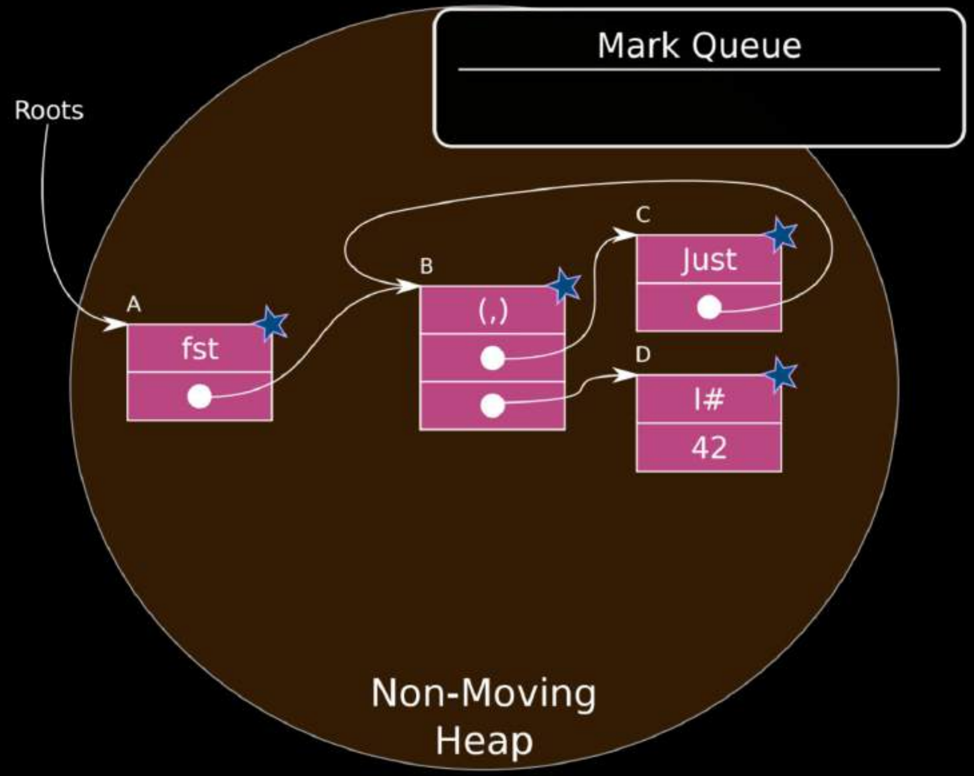

Slides from Ben Gamari 2018

# Mark & Sweep Garbage Collection: Marking (B)




Slides from Ben Gamari 2018

# Mark & Sweep Garbage Collection: Sweep

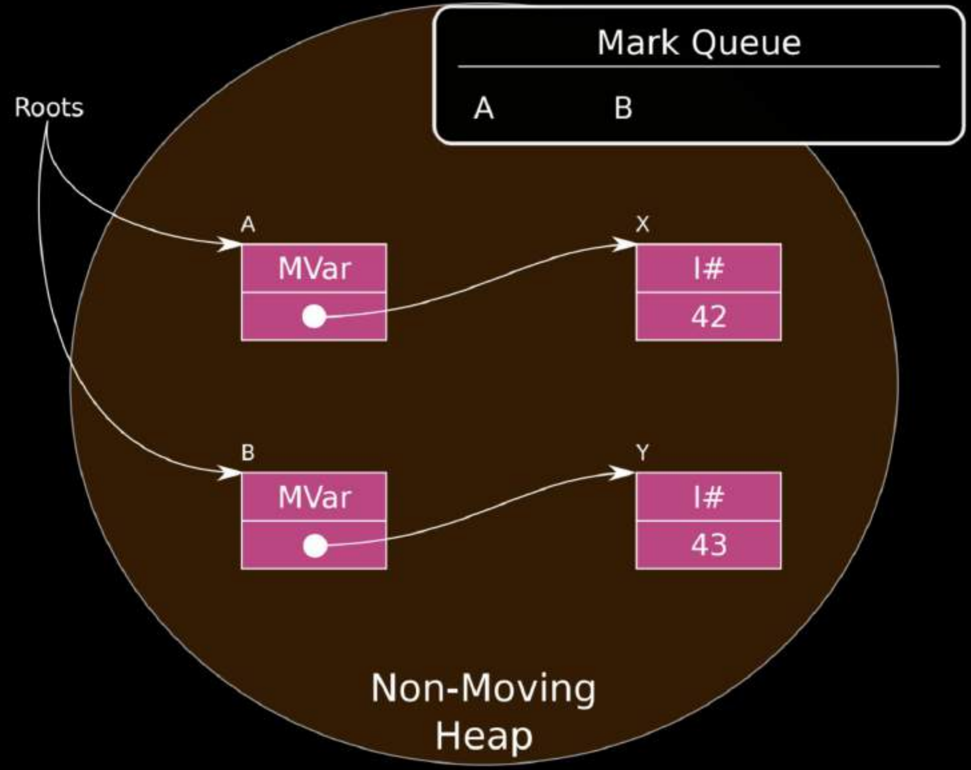



Slides from Ben Gamari 2018

# Nonmoving GC for the oldest generation

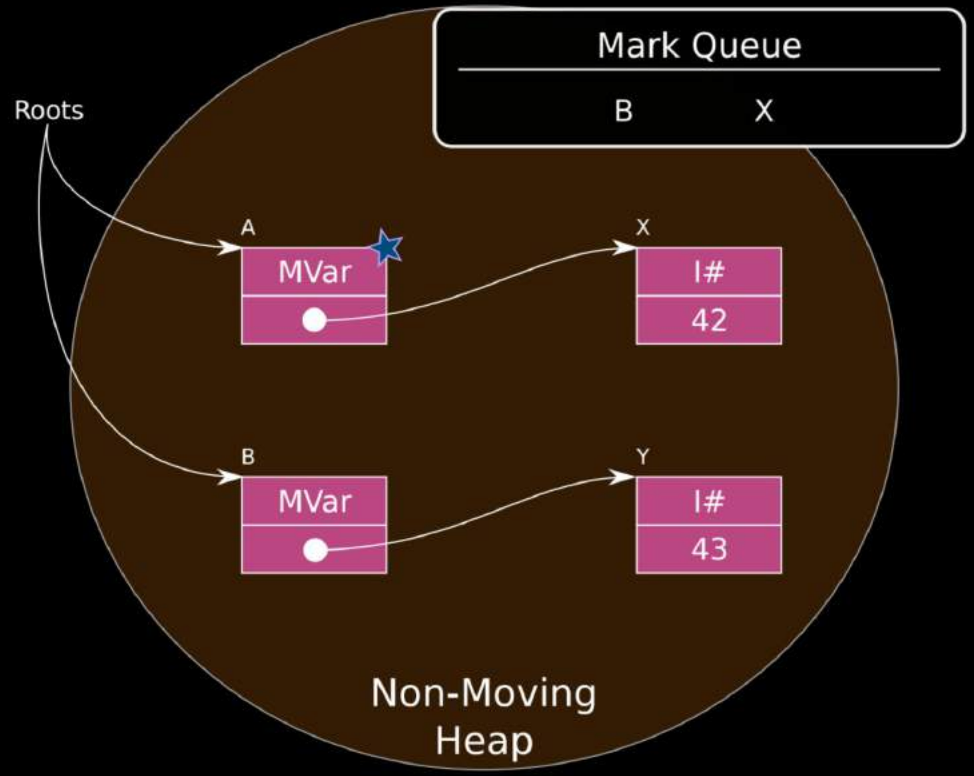
- References remain valid (nothing is moved/copied)
- Sweep phase leaves a fragmented heap behind
  - 
  - Needs more management for the allocator
- Mark/Sweep runs (mostly) concurrently with mutator(s)

# The problem of mutation



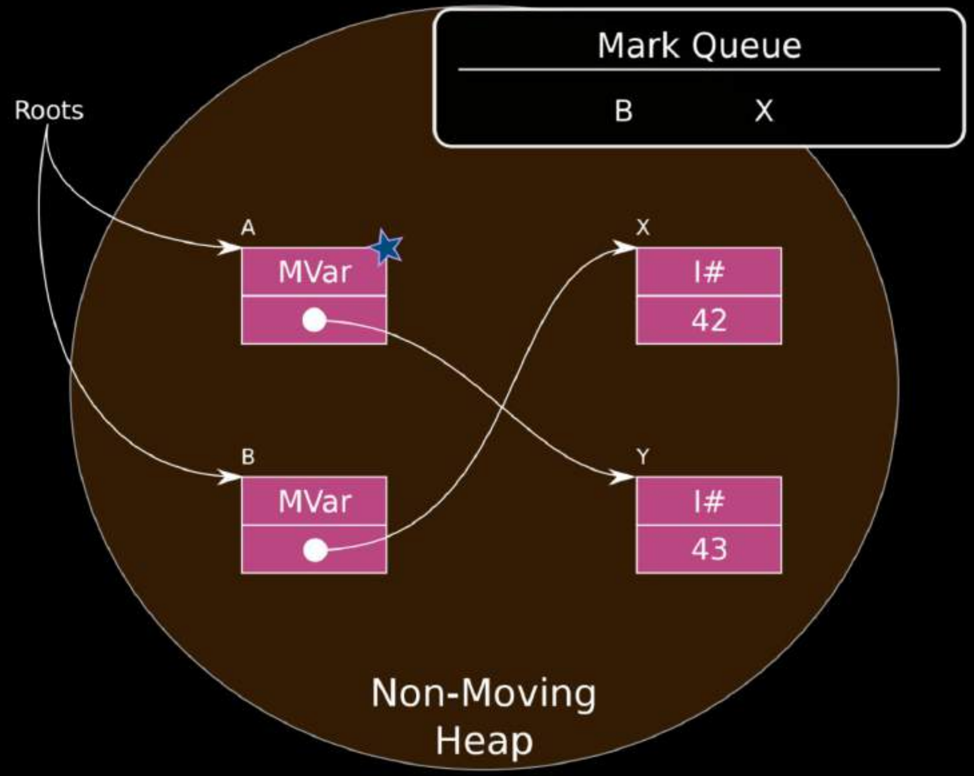

Slides from Ben Gamari 2018

# The problem of mutation



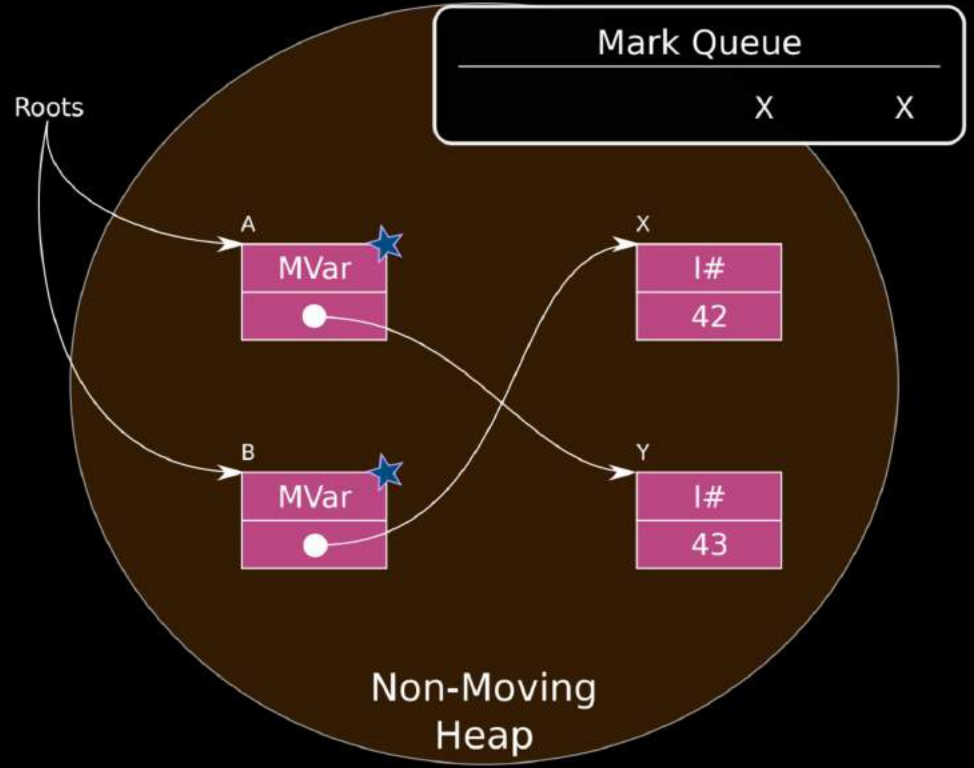

Slides from Ben Gamari 2018

# The problem of mutation



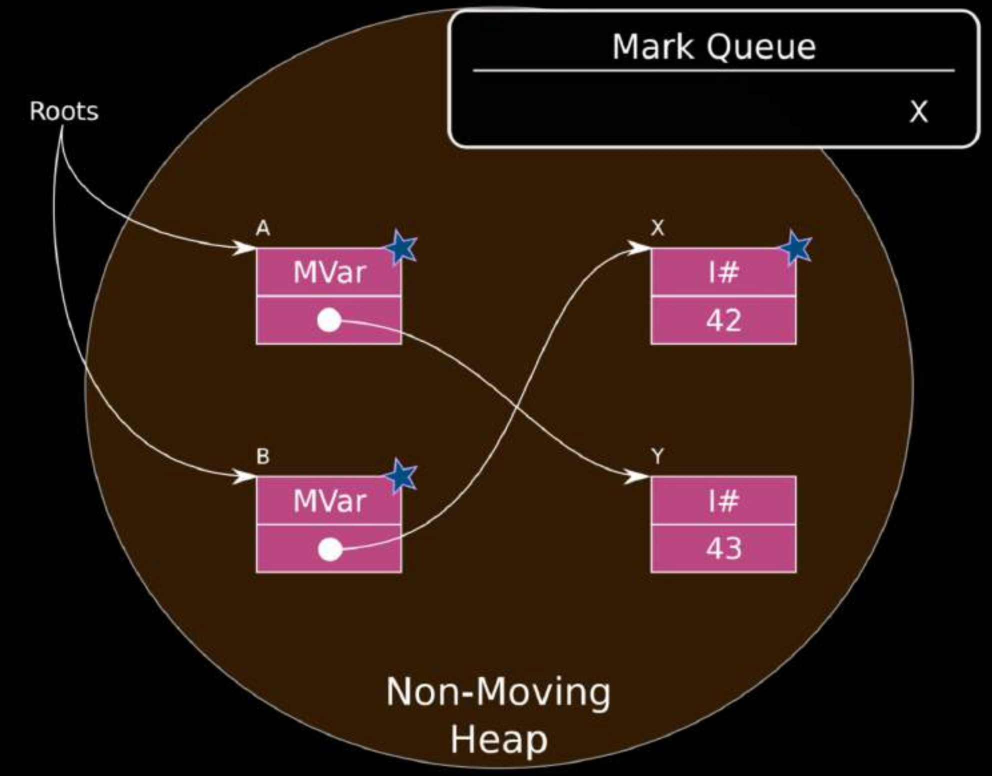

Slides from Ben Gamari 2018

# The problem of mutation



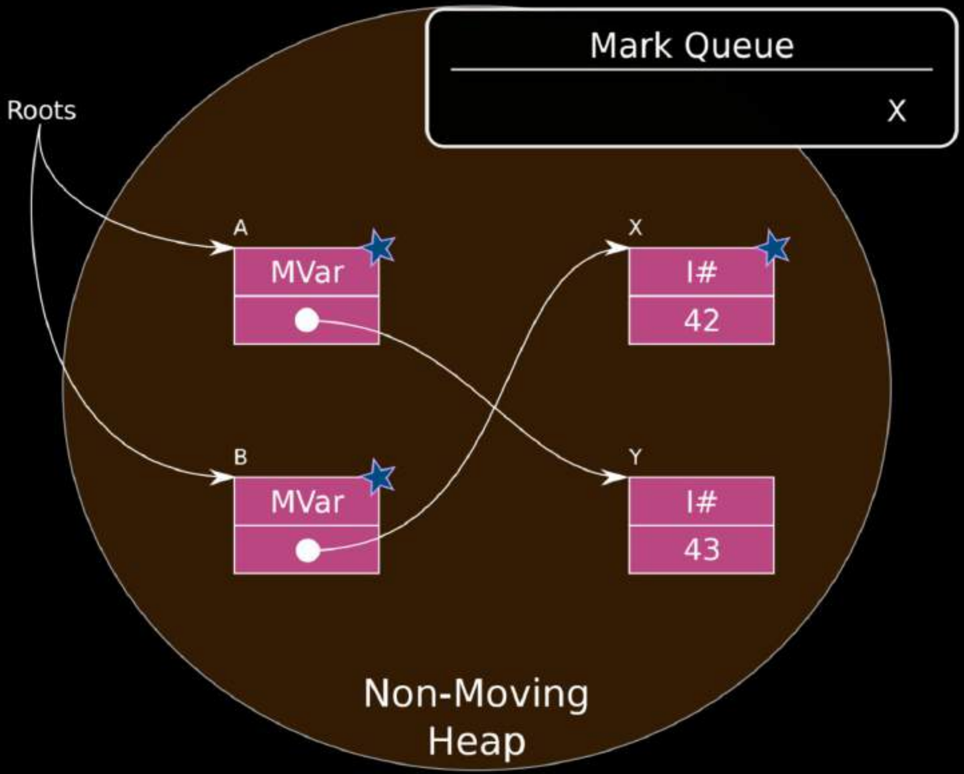

Slides from Ben Gamari 2018

# The problem of mutation



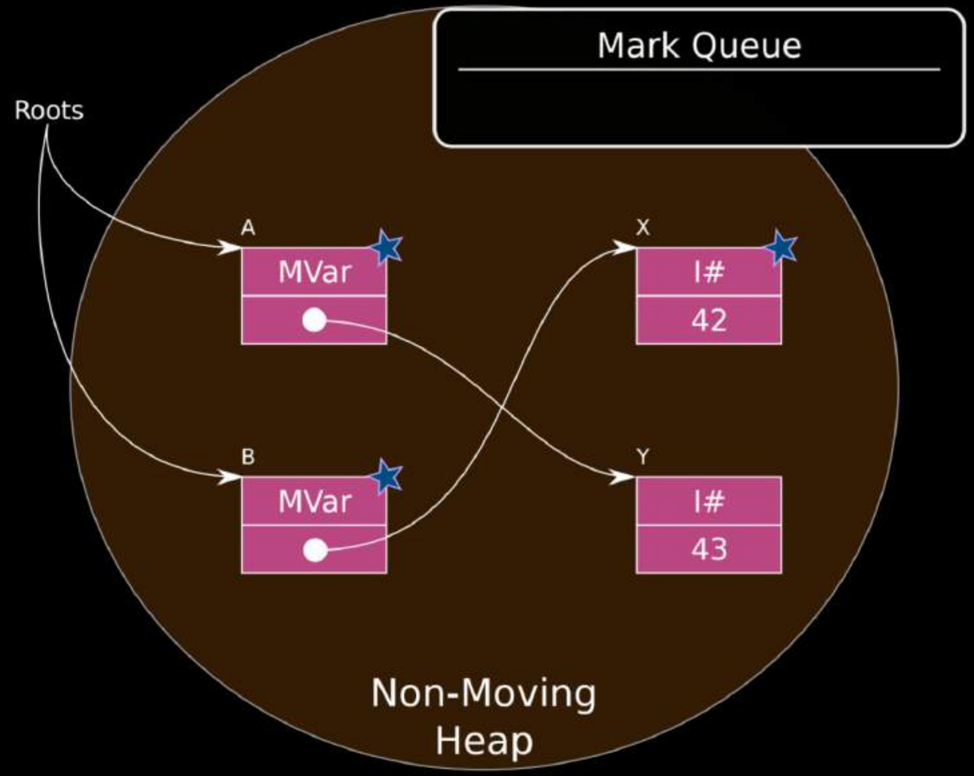

Slides from Ben Gamari 2018

# The problem of mutation



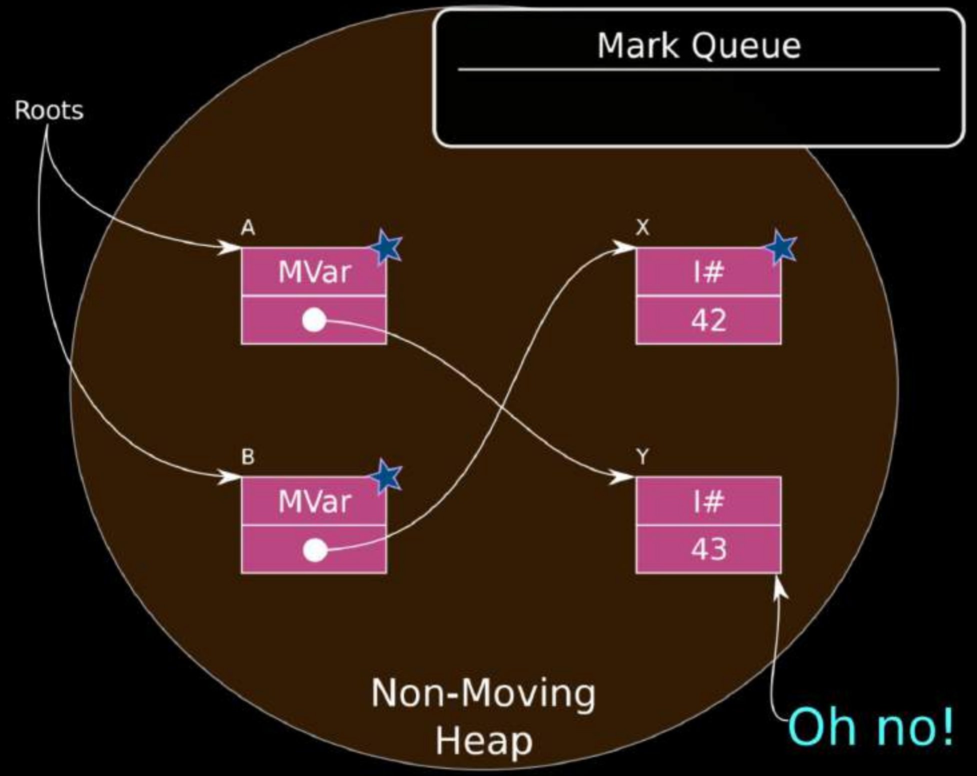

Slides from Ben Gamari 2018

# The problem of mutation




Slides from Ben Gamari 2018

# The problem of mutation




Slides from Ben Gamari 2018

# 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 $t_0$.

Slides from Ben Gamari 2018

# Snapshot-at-the-beginning

The snapshot invariant:

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

Consequently,

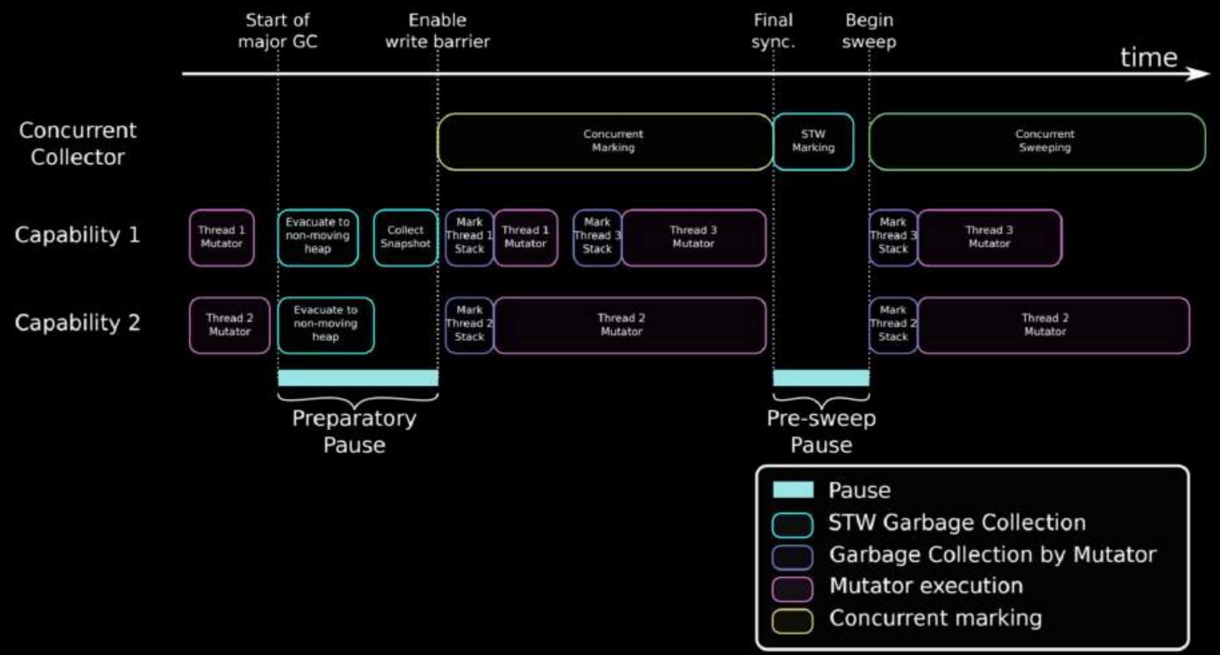
1. All objects live at $t_0$ will be retained.
2. Many objects dead at $t_0$ will be freed.
3. All objects allocated after $t_0$ will be retained.

How to accomplish this?

A write barrier.

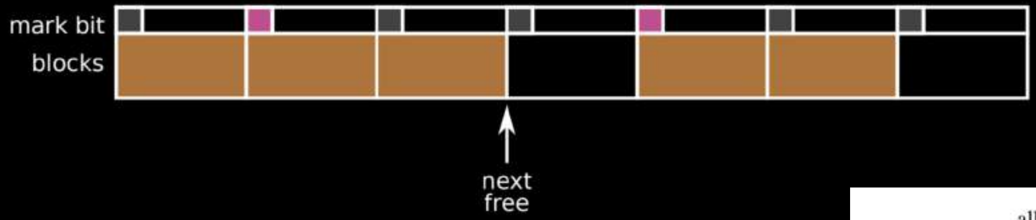
Slides from Ben Gamari 2018

# The lifecycle of a GC

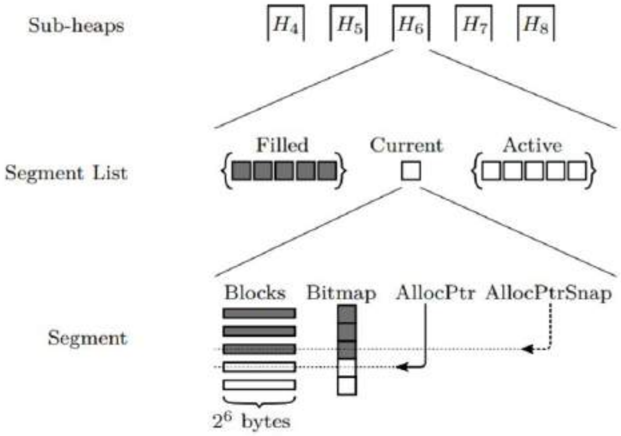



Slides from Ben Gamari 2018

# Allocator structure: The segment




ISMM '20, June 16, 2020, London, UK




e Ueno et
ısists of a
out shows
three sets
) heap ob-

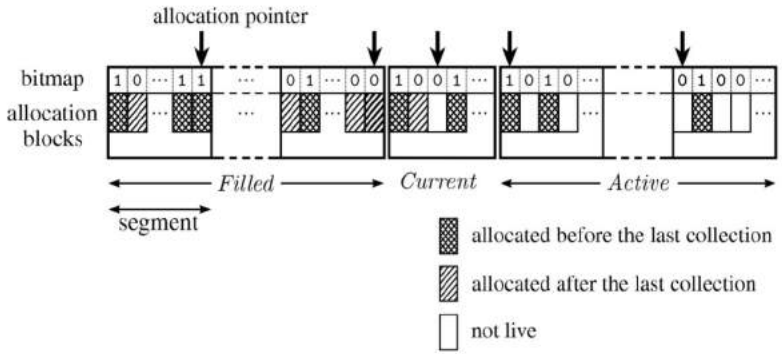
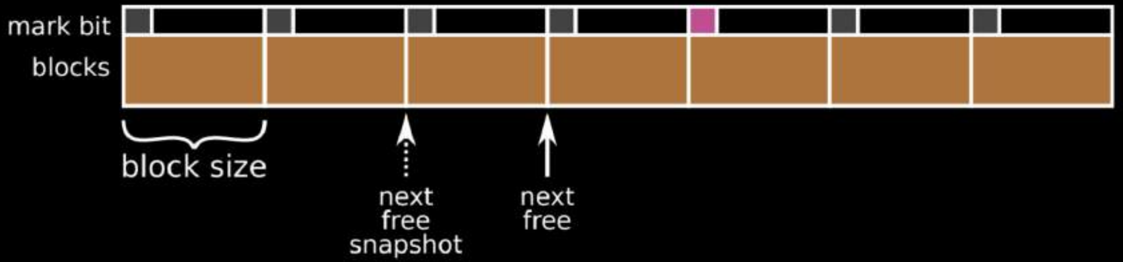



**Figure 1.** Structure of an allocation heap

From Ben Gamari 2018, Gamari/Dietz2020, UenoOhori2015

# 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.

Slides from Ben Gamari 2018

# What the new collector won't do...

Concurrent collection isn't a silver bullet:

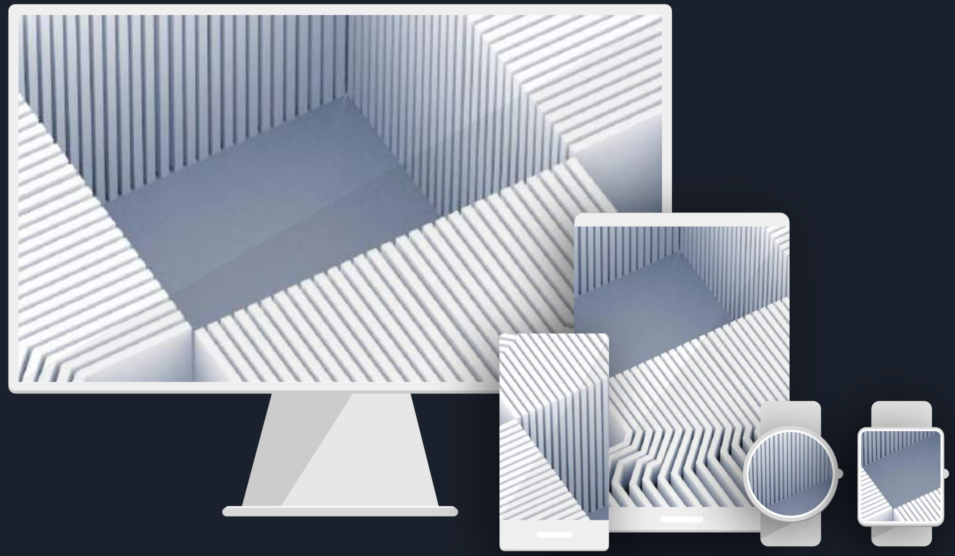
- It probably won't make your program run faster
- It won't make your program scale more effectively across cores (but this may be future work)
- It may reduce your program's memory footprint, but not by as much as you might expect
- It is not provide hard realtime latency guarantees
- It does not mow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

```
./my-program +RTS -xn -RTS
./my-program +RTS --nonmoving-gc -RTS
```

Slides from Ben Gamari 2018

# Thank you!

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur eleifend a diam quis suscipit.



# References

**Slides stolen from:**

- Edward Z Yang (Lecture about GHC RTS)
- Richard Jones (talk about GC and multicore)
- Ben Gamari (-early- design of the nonmoving GC)
- Takenobu Tani (GHC RTS illustrated)

**Good Reads:**

- **Short paper** / Paper nonmoving GC
- Prior Art (SML#, not Haskell)
- A summary of modern GC research
- (Original) Generational GC for GHC
- Incremental GC Experiment in GHC and successive further work
- Parallel GC
- Options in GHC manual