Retrofitting a Concurrent GC onto OCaml

KC Sivaramakrishnan

Assistant Professor, CSE

@kc_srk

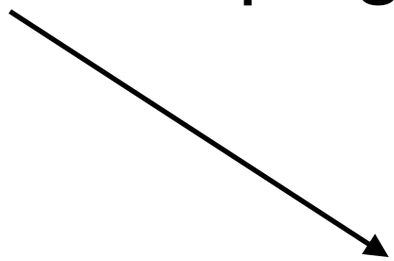
IIT MADRAS

OCaml

industrial-strength, pragmatic, functional programming language

OCaml

industrial-strength, pragmatic, functional programming language



Hindley-Milner Type Inference

Powerful module system

OCaml

industrial-strength, pragmatic, functional programming language

- Functional core with imperative and object-oriented features
- Native (x86, ARM, …), JavaScript, JVM

Hindley-Milner Type Inference

Powerful module system

OCaml

industrial-strength, pragmatic, functional programming language

- Functional core with imperative and object-oriented features
- Native (x86, ARM, …), JavaScript, JVM

Hindley-Milner Type Inference

Powerful module system

Facebook: REASON, Infer, flow, Hack, PXRE

Microsoft: Project Everest

Jane Street

The Coq Proof Assistant

MIRAGE OS

OCaml

industrial-strength, pragmatic, functional programming language

- Functional core with imperative and object-oriented features
- Native (x86, ARM, …), JavaScript, JVM

- Hindley-Milner Type Inference
- Powerful module system

No multicore support!

Facebook: REASON, Infer, flow, Hack

Microsoft: Project Everest

Jane Street

The Coq Proof Assistant

MIRAGE OS

Multicore OCaml

Multicore OCaml

- Native support for *concurrency* and *parallelism* in OCaml

Multicore OCaml

- Native support for *concurrency* and *parallelism* in OCaml
- I lead the project from IIT Madras (U Cambridge earlier) with collaborators from
  - **Academia:** University of Cambridge, UK & INRIA, Paris, France
  - **Companies:** Jane Street, Tarides, OCaml Labs, Segfault Systems

Multicore OCaml

- Native support for *concurrency* and *parallelism* in OCaml
- I lead the project from IIT Madras (U Cambridge earlier) with collaborators from
  - **Academia:** University of Cambridge, UK & INRIA, Paris, France
  - **Companies:** Jane Street, Tarides, OCaml Labs, Segfault Systems
- Expected to hit mainline in mid 2020

Multicore OCaml

- Native support for *concurrency* and *parallelism* in OCaml
- I lead the project from IIT Madras (U Cambridge earlier) with collaborators from
  - **Academia:** University of Cambridge, UK & INRIA, Paris, France
  - **Companies:** Jane Street, Tarides, OCaml Labs, Segfault Systems
- Expected to hit mainline in mid 2020

Stephe Dolan, Leo White, Anil Madhavapeddy, Daniel Hillerstrom, Tom Kelly, Sadiq Jaffer, Sunil Nimmagadda, Damien Doligez, Xavier Leroy, David Allsopp, Anmol Sahoo, Gemma Gordon and many other ***open source contributors***

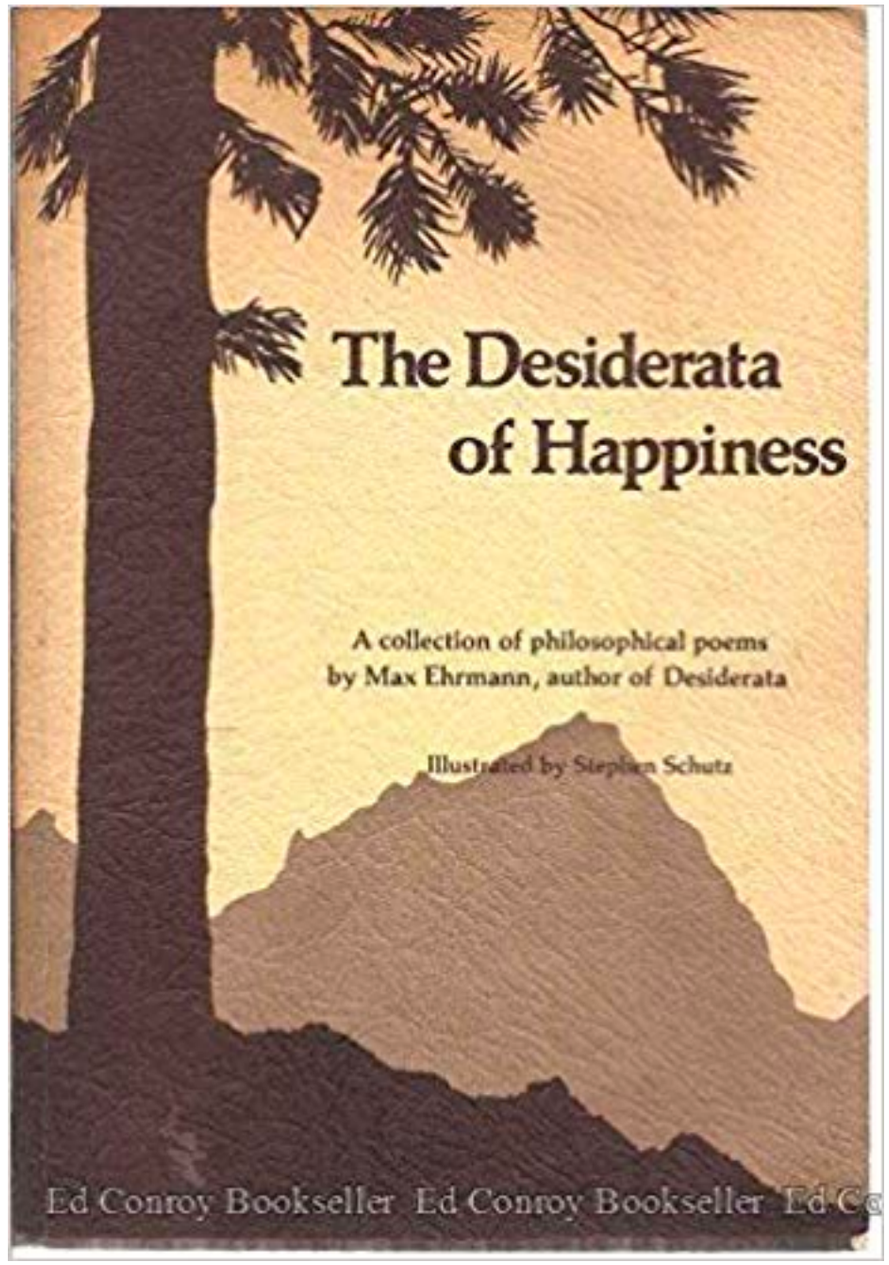
Multicore OCaml

- Native support for *concurrency* and *parallelism* in OCaml
- I lead the project from IIT Madras (U Cambridge earlier) with collaborators from
  - **Academia:** University of Cambridge, UK & INRIA, Paris, France
  - **Companies:** Jane Street, Tarides, OCaml Labs, Segfault Systems
- Expected to hit mainline in mid 2020

Stephe Dolan, Leo White, Anil Madhavapeddy, Daniel Hillerstrom, Tom Kelly, Sadiq Jaffer, Sunil Nimmagadda, Damien Doligez, Xavier Leroy, David Allsopp, Anmol Sahoo, Gemma Gordon and many other ***open source contributors***

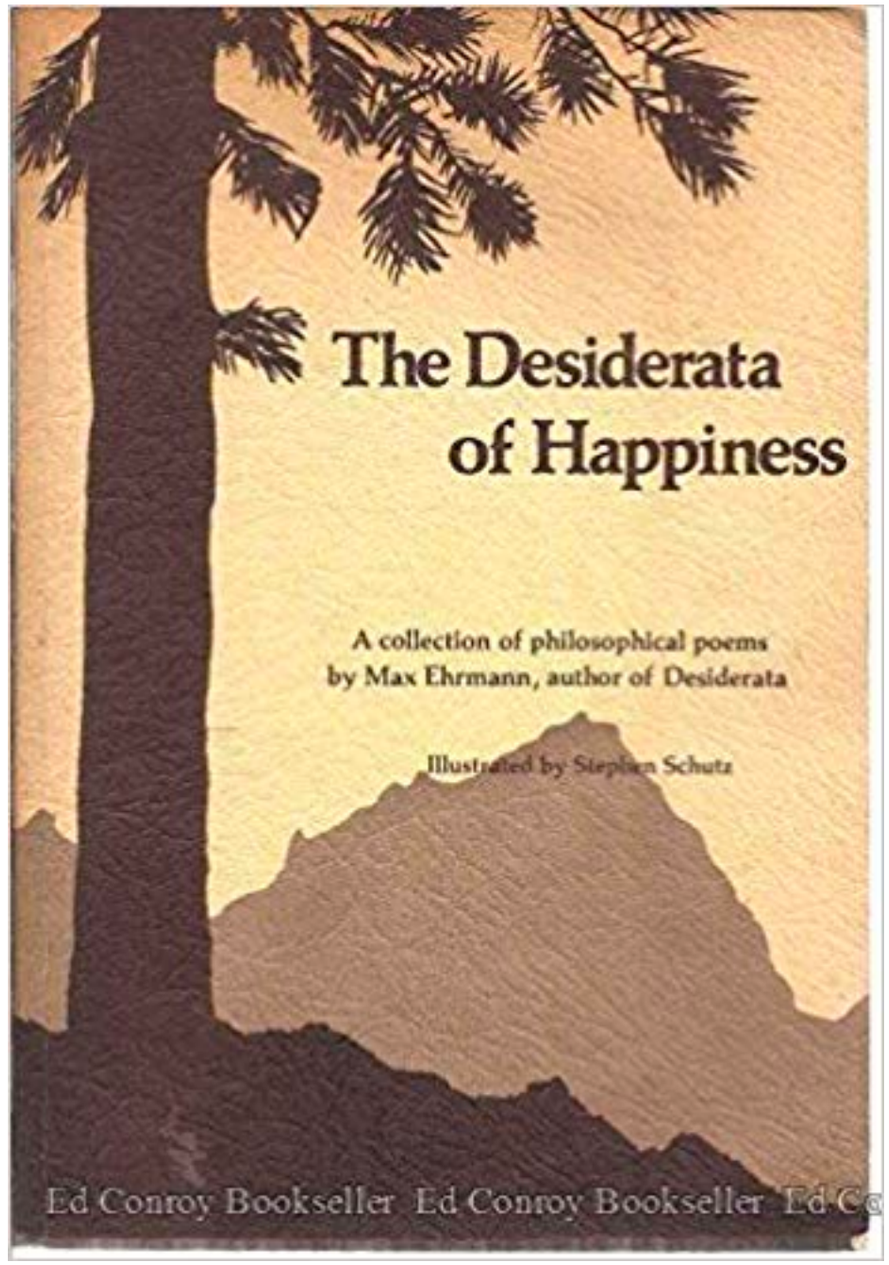
- In this talk,
  - Adding a concurrent GC to OCaml without destroying baseline performance
  - Opportunities and Impact of ***open*** (source) research

Multicore OCaml GC: Desiderata



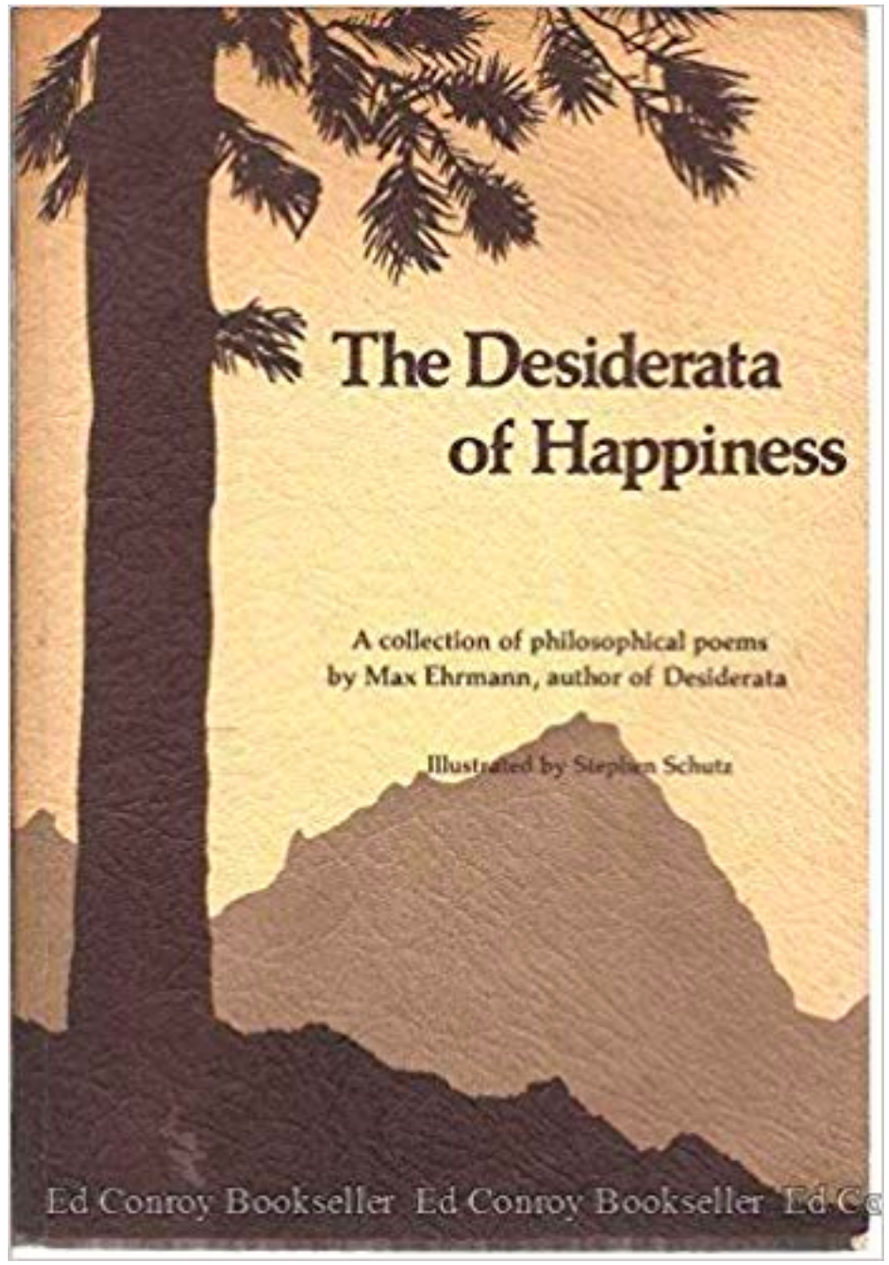
Multicore OCaml GC: Desiderata

- *Code backwards compatibility*
  - ✦ Do not break existing code



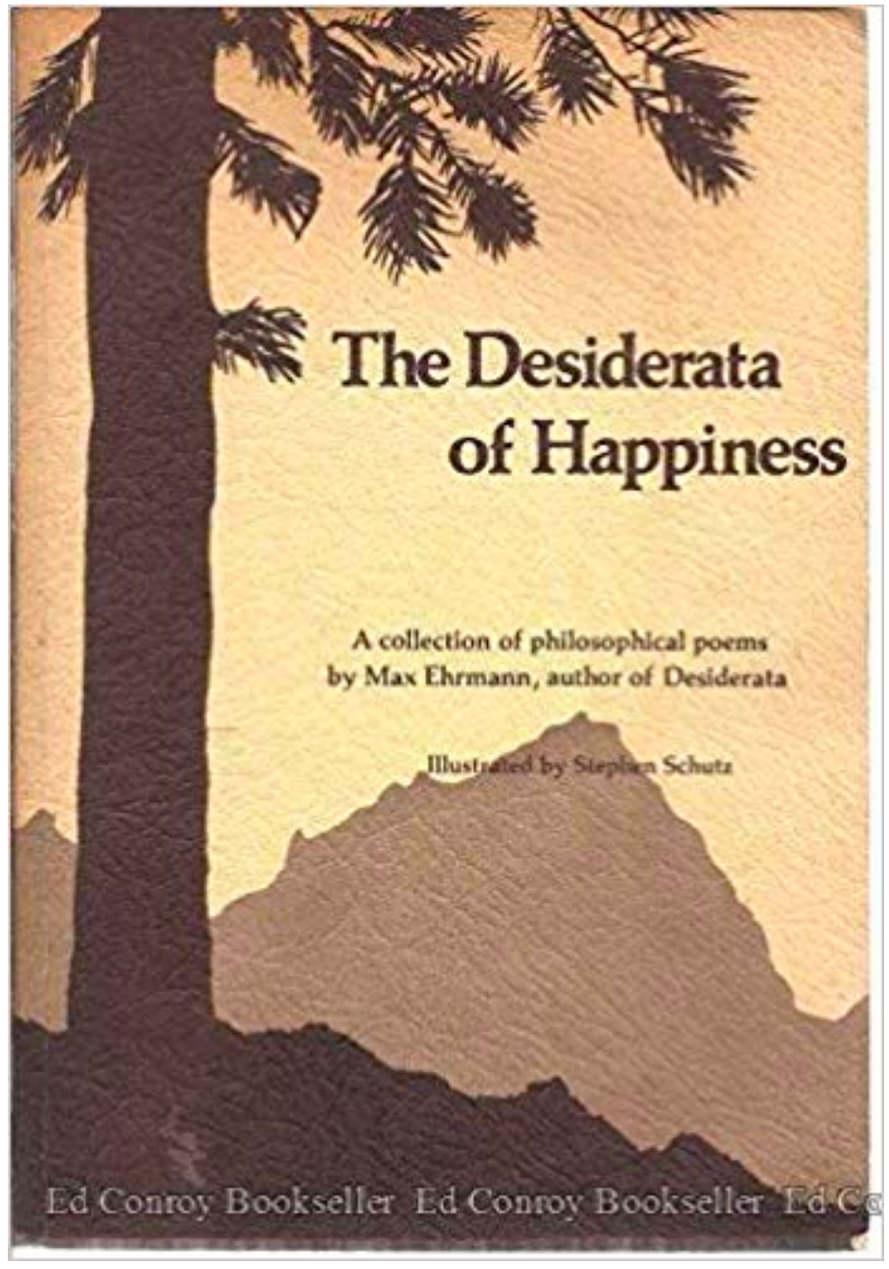
Multicore OCaml GC: Desiderata

- *Code backwards compatibility*
  - ✦ Do not break existing code
- *Performance backwards compatibility*
  - ✦ Do not slow down existing programs



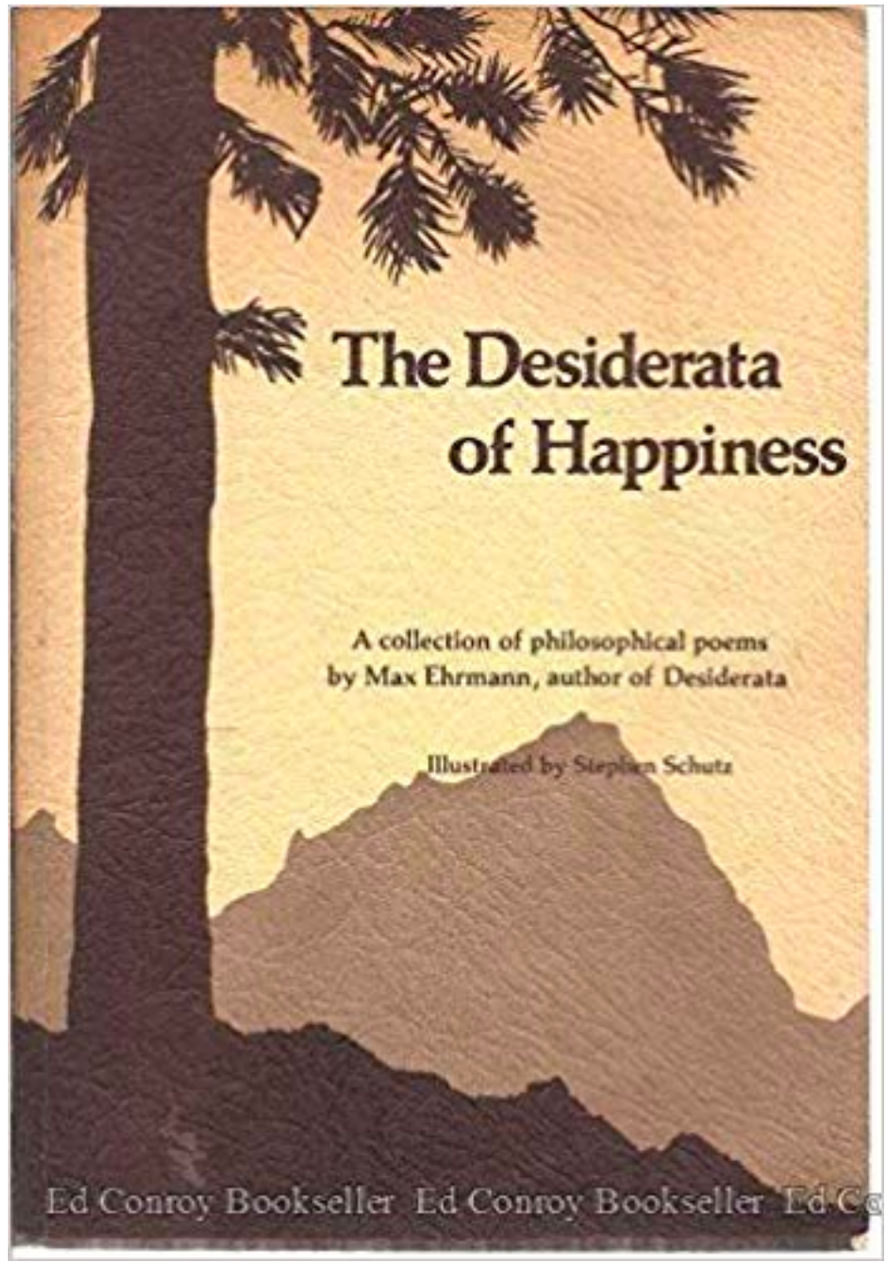
Multicore OCaml GC: Desiderata

- *Code backwards compatibility*
  - ✦ Do not break existing code
- *Performance backwards compatibility*
  - ✦ Do not slow down existing programs
- *Minimise pause times*
  - ✦ Latency is more important than throughput



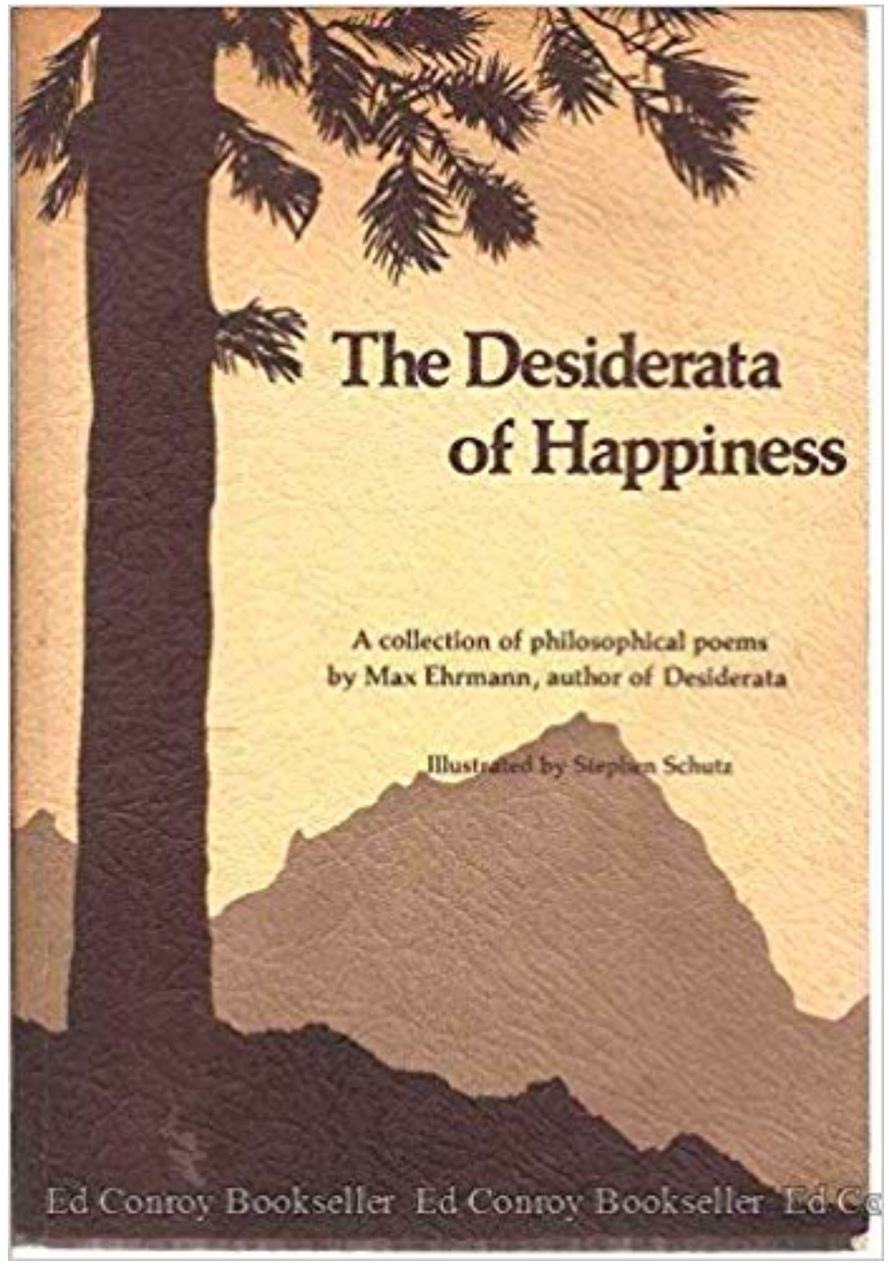
Multicore OCaml GC: Desiderata

- *Code backwards compatibility*
  - ✦ Do not break existing code
- *Performance backwards compatibility*
  - ✦ Do not slow down existing programs
- *Minimise pause times*
  - ✦ Latency is more important than throughput
- *Performance predictability and stability*
  - ✦ Slow and stable better than fast but unpredictable



Multicore OCaml GC: Desiderata

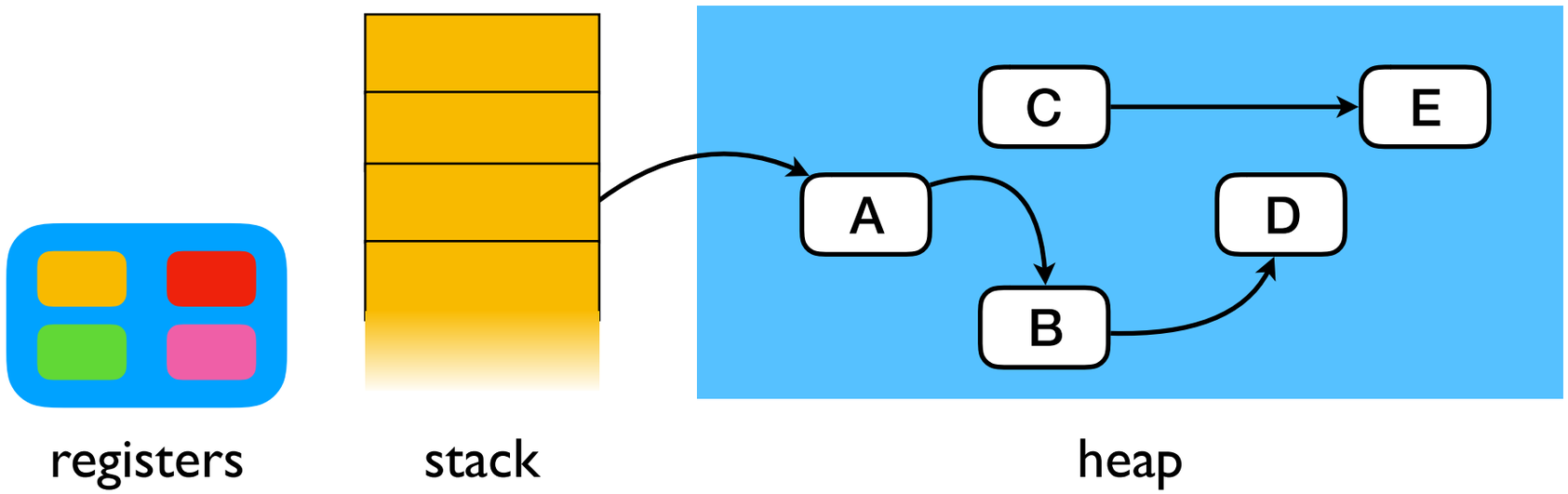
- *Code backwards compatibility*
  - ✦ Do not break existing code
- *Performance backwards compatibility*
  - ✦ Do not slow down existing programs
- *Minimise pause times*
  - ✦ Latency is more important than throughput
- *Performance predictability and stability*
  - ✦ Slow and stable better than fast but unpredictable
- *Minimize knobs*
  - ✦ 90% of programs should run at 90% peak performance by default



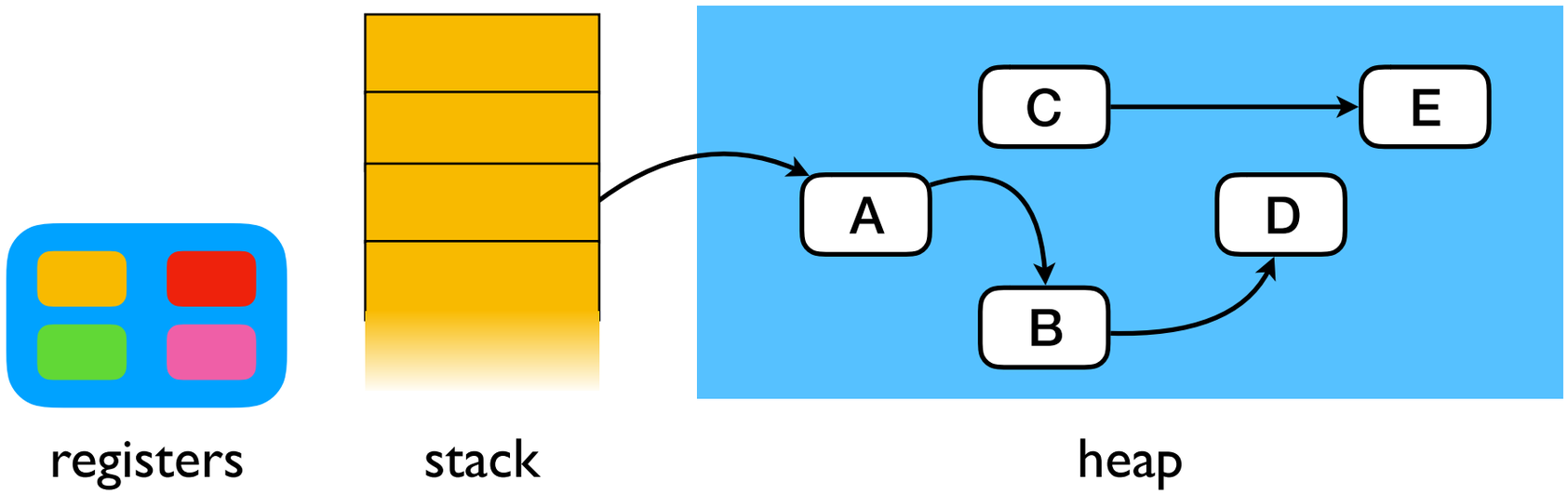
Outline

- Difficult to appreciate GC choices in isolation
- Begin with a GC for a *sequential purely functional* language
  - ✦ Gradually add mutations, parallelism and concurrency

Sequential purely functional

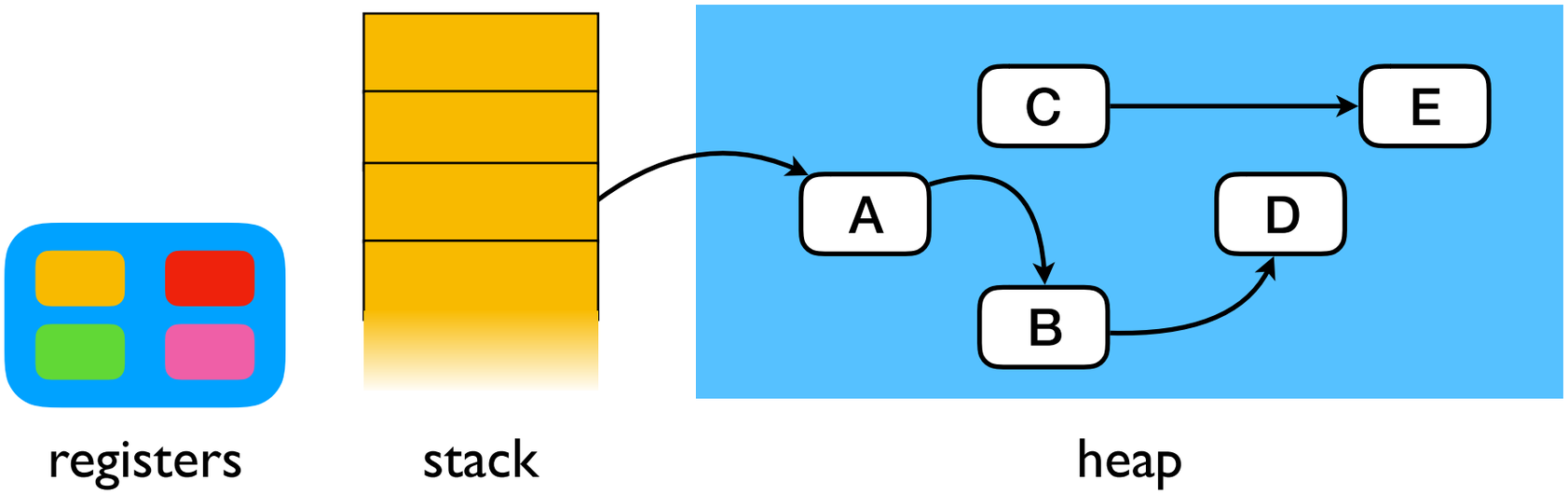


Sequential purely functional



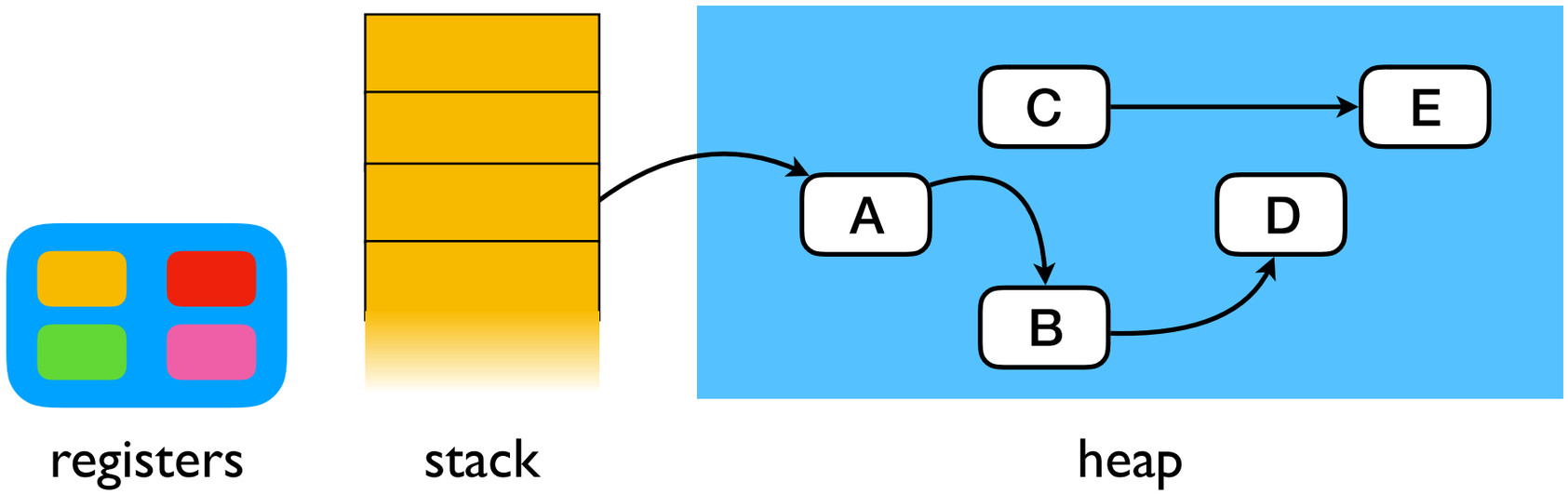
- Stop-the-world mark and sweep

Sequential purely functional



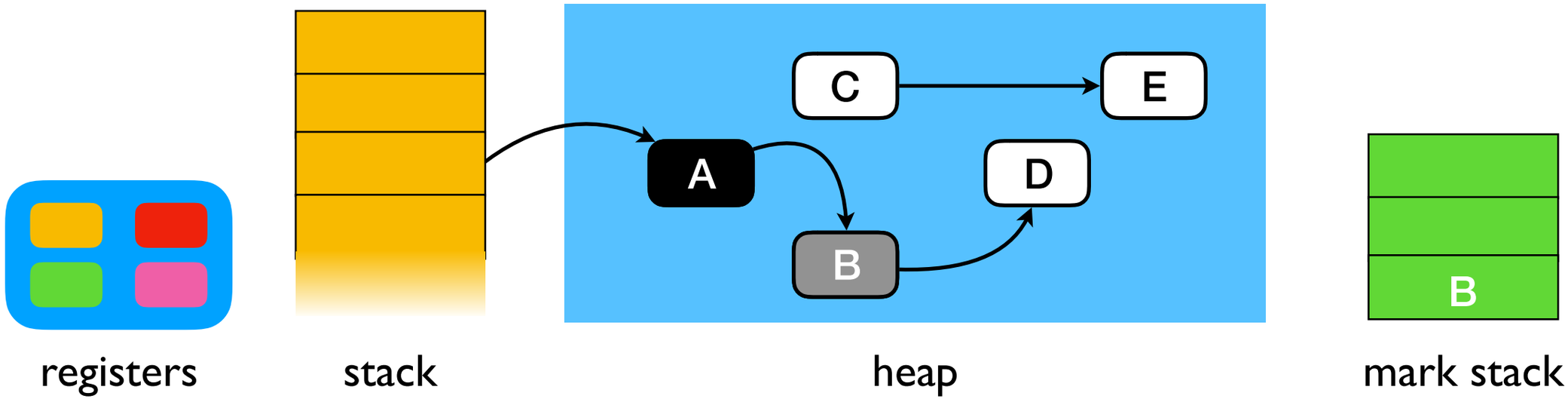
- Stop-the-world mark and sweep
- Tri-color marking
  - ✦ States: White (Unmarked), Grey (Marking), Black (Marked)

Sequential purely functional



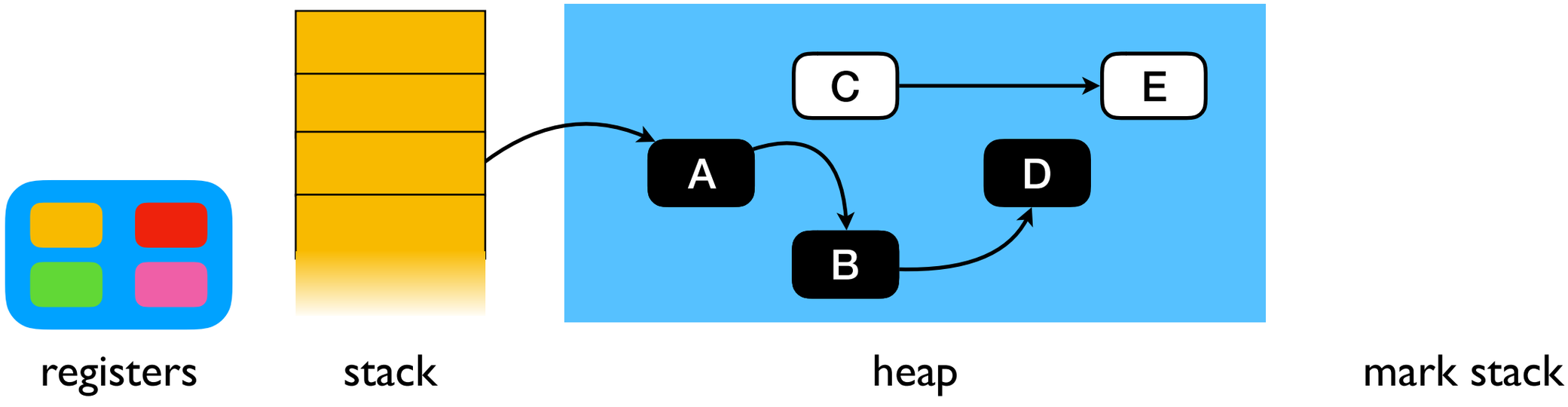
- Stop-the-world mark and sweep
- Tri-color marking
  - ✦ States: White (Unmarked), Grey (Marking), Black (Marked)
- White —> Grey (mark stack) —> Black

Sequential purely functional



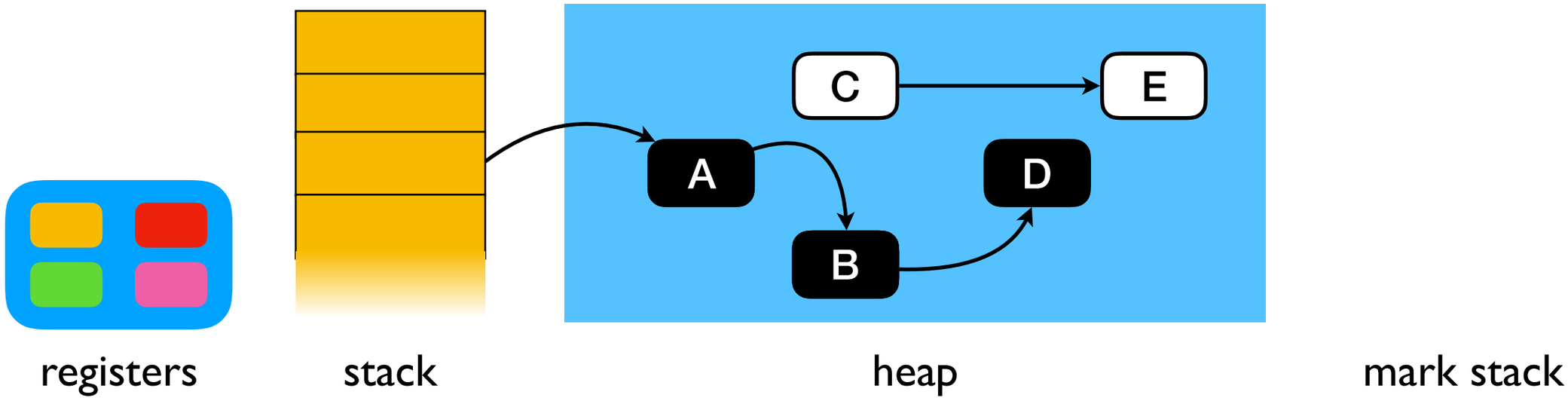
- Stop-the-world mark and sweep
- Tri-color marking
  - ✦ States: White (Unmarked), Grey (Marking), Black (Marked)
- White —> Grey (mark stack) —> Black

Sequential purely functional



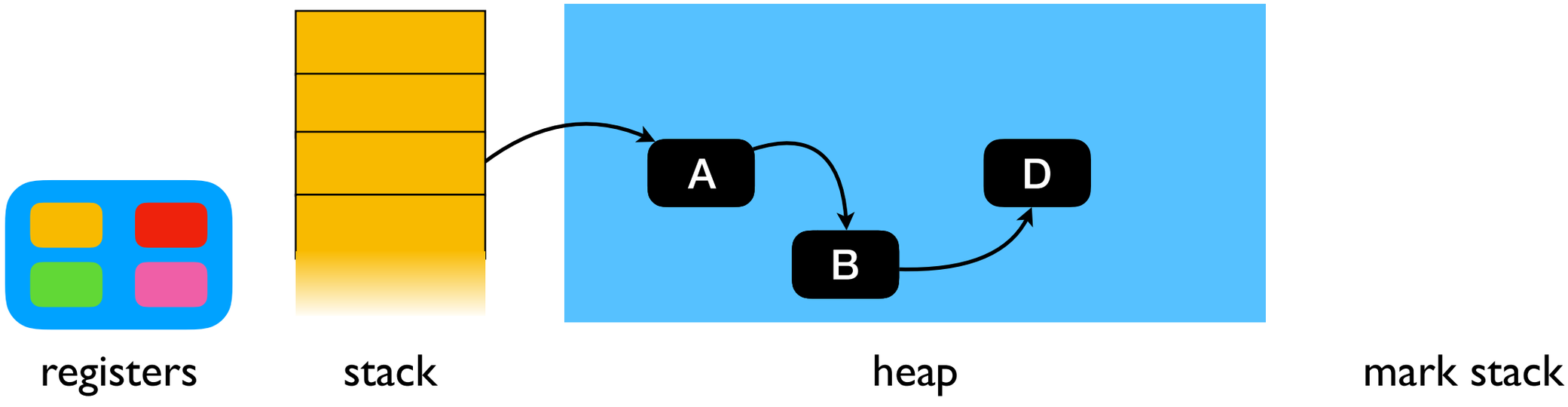
- Stop-the-world mark and sweep
- Tri-color marking
  - ✦ States: White (Unmarked), Grey (Marking), Black (Marked)
- White —> Grey (mark stack) —> Black
- Mark stack is empty => *done marking*
  - ✦ Tri-color invariant: *No black object points to a white object*

Sequential purely functional



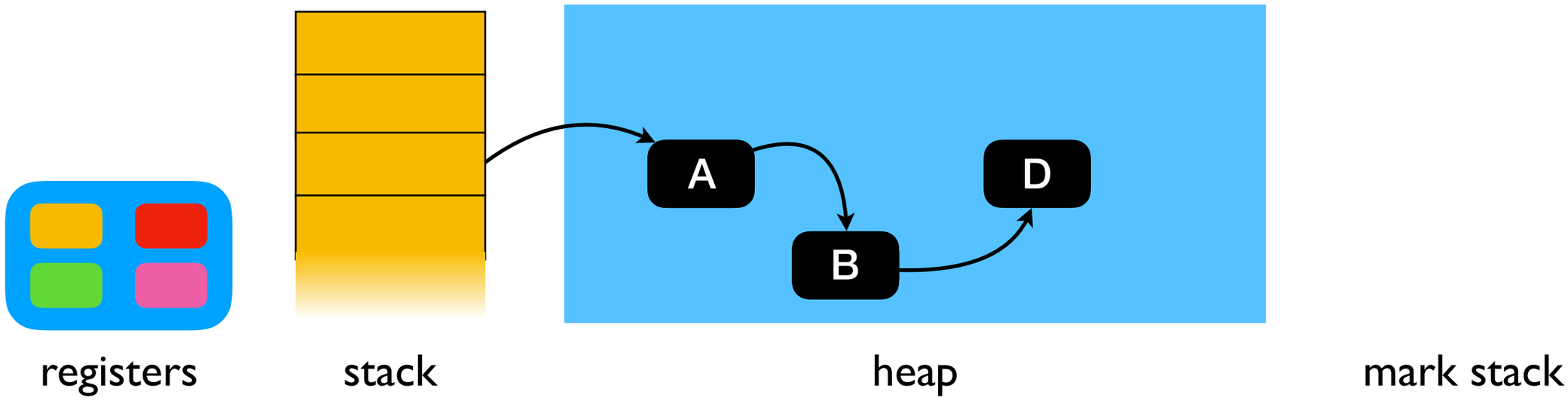
- Stop-the-world mark and sweep
- Tri-color marking
  - ✦ States: White (Unmarked), Grey (Marking), Black (Marked)
- White —> Grey (mark stack) —> Black
- Mark stack is empty => *done marking*
  - ✦ Tri-color invariant: *No black object points to a white object*
- Sweeping : walk the heap and free white objects

Sequential purely functional

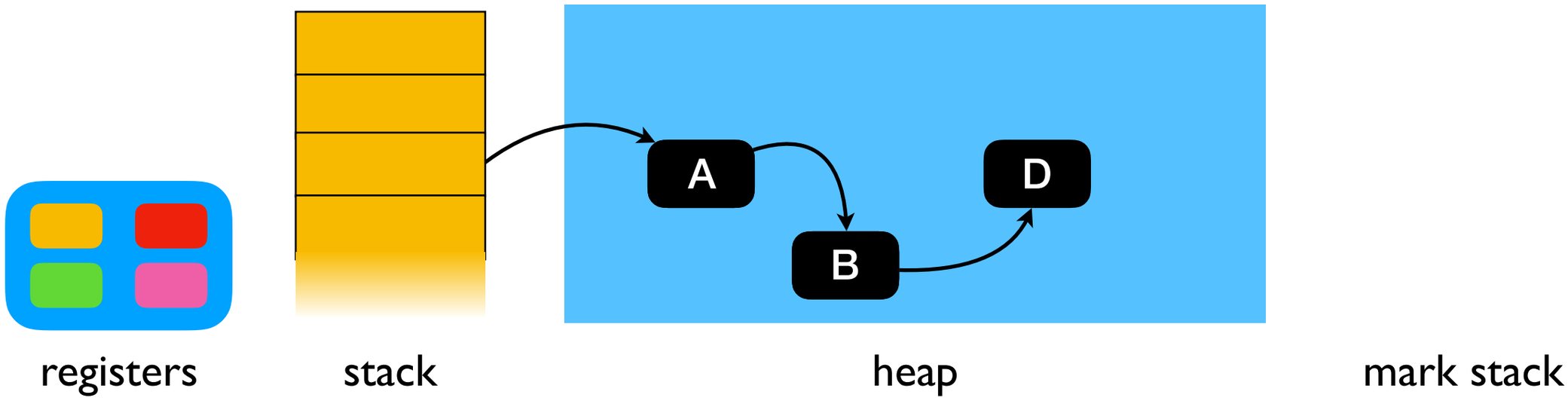


- Stop-the-world mark and sweep
- Tri-color marking
  - ✦ States: White (Unmarked), Grey (Marking), Black (Marked)
- White —> Grey (mark stack) —> Black
- Mark stack is empty => *done marking*
  - ✦ Tri-color invariant: *No black object points to a white object*
- Sweeping : walk the heap and free white objects

Sequential purely functional

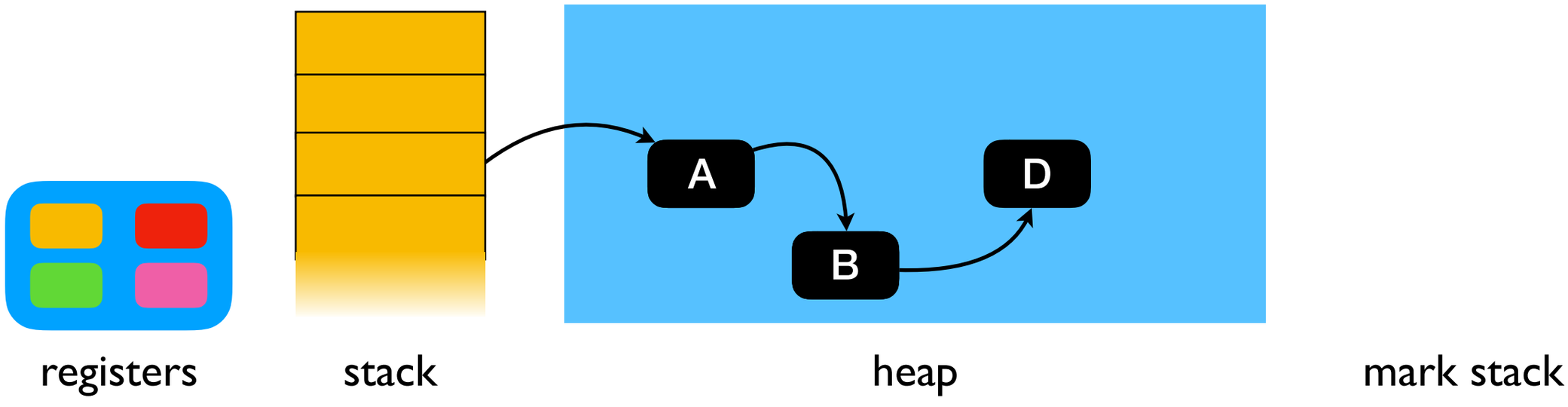


Sequential purely functional



- Pros
  - ✦ Simple
  - ✦ Can perform the GC incrementally
    - ✤ …|–mutator–|–mark–|–mutator–|–mark–|–mutator–|–sweep–|…
    - ✤ *Minimise GC pause times!*

Sequential purely functional



- Pros
  - ✦ Simple
  - ✦ Can perform the GC incrementally
    - ✤ …|–mutator–|–mark–|–mutator–|–mark–|–mutator–|–sweep–|…
    - ✤ *Minimise GC pause times!*
- Cons
  - ✦ Need to maintain free-list of objects => allocations overheads + fragmentation

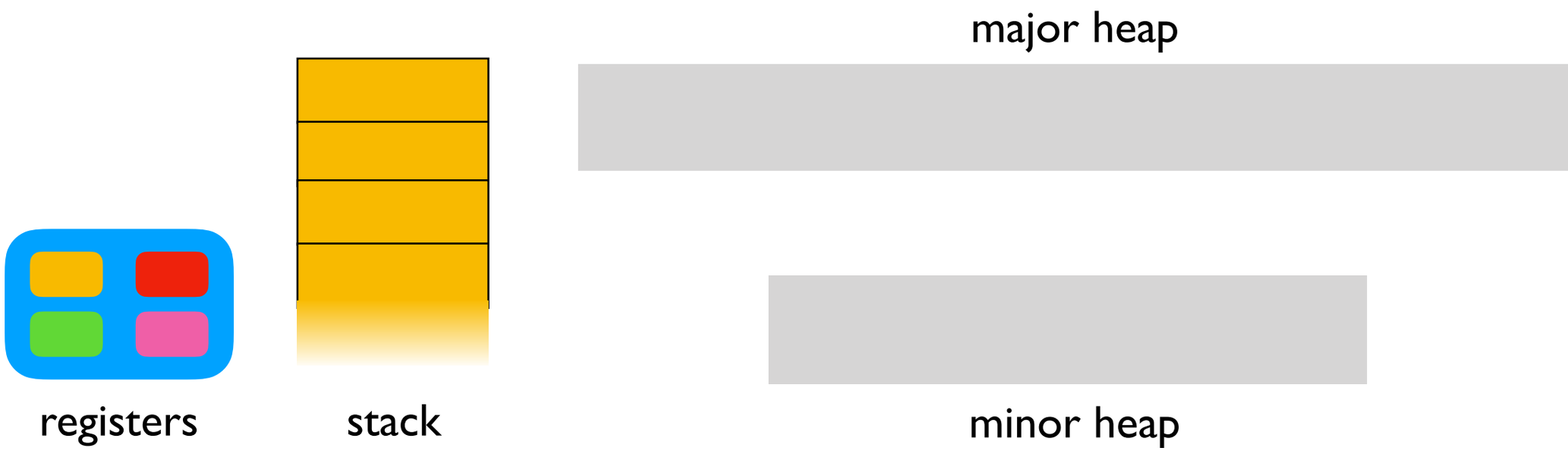
Generational GC

Generational GC

- Generational Hypothesis
  - ✦ Young objects are much more likely to die than old objects

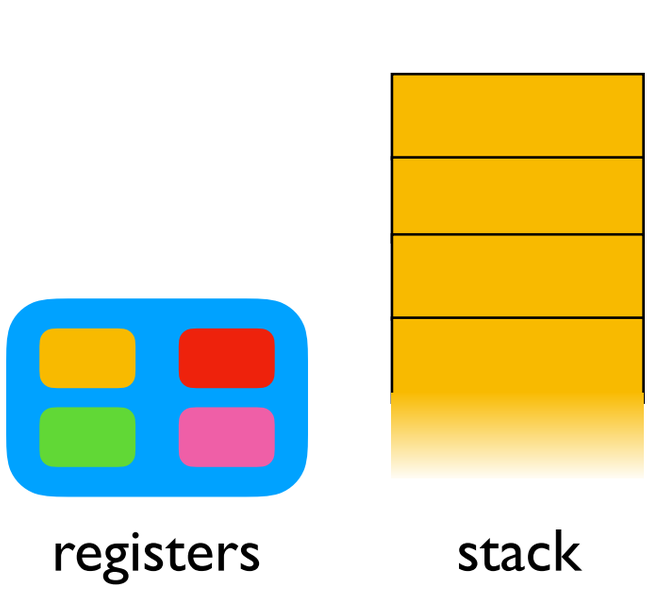
Generational GC

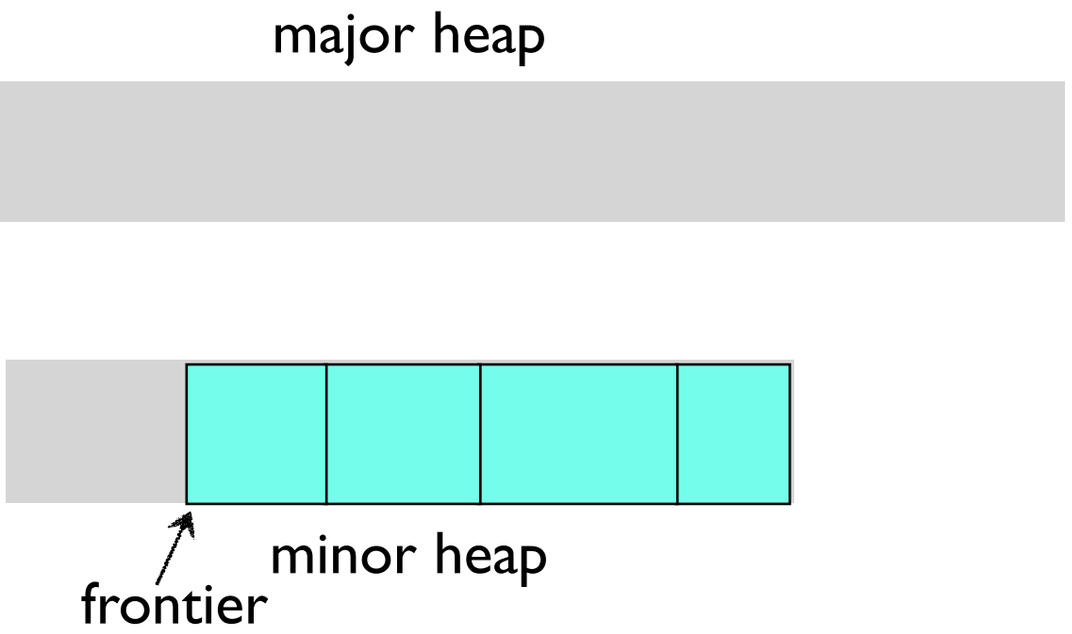
- Generational Hypothesis
  - ✦ Young objects are much more likely to die than old objects



Generational GC

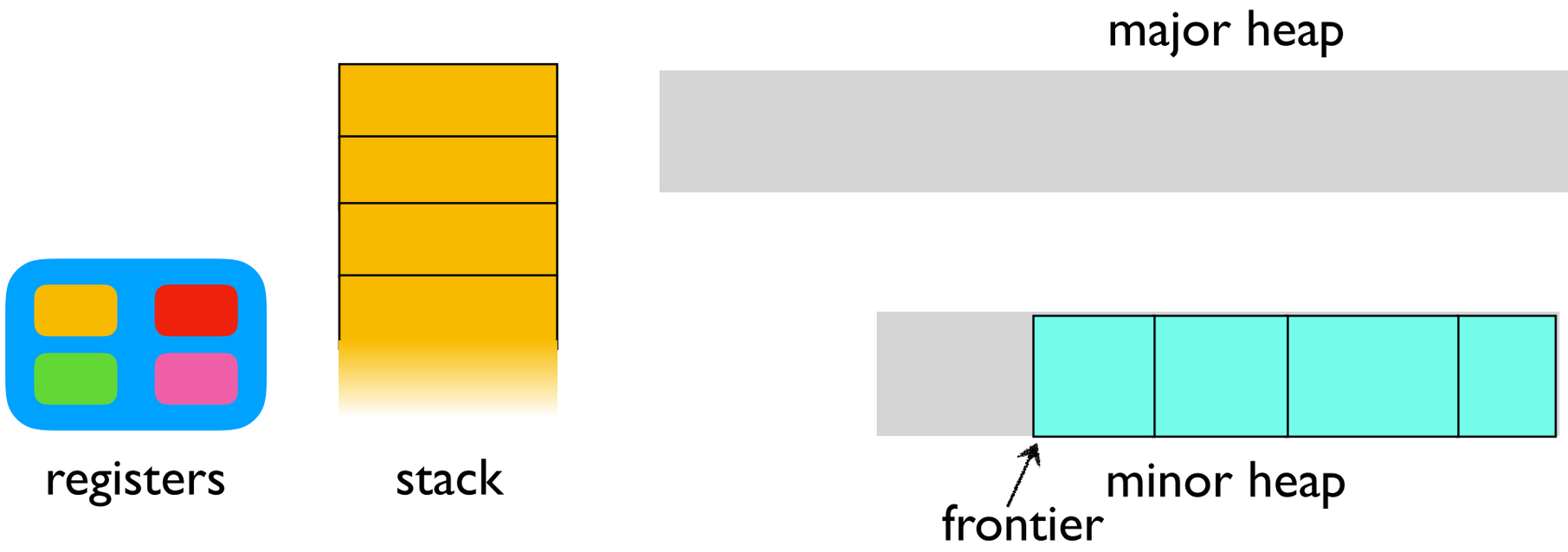
- Generational Hypothesis
  - ✦ Young objects are much more likely to die than old objects





Generational GC

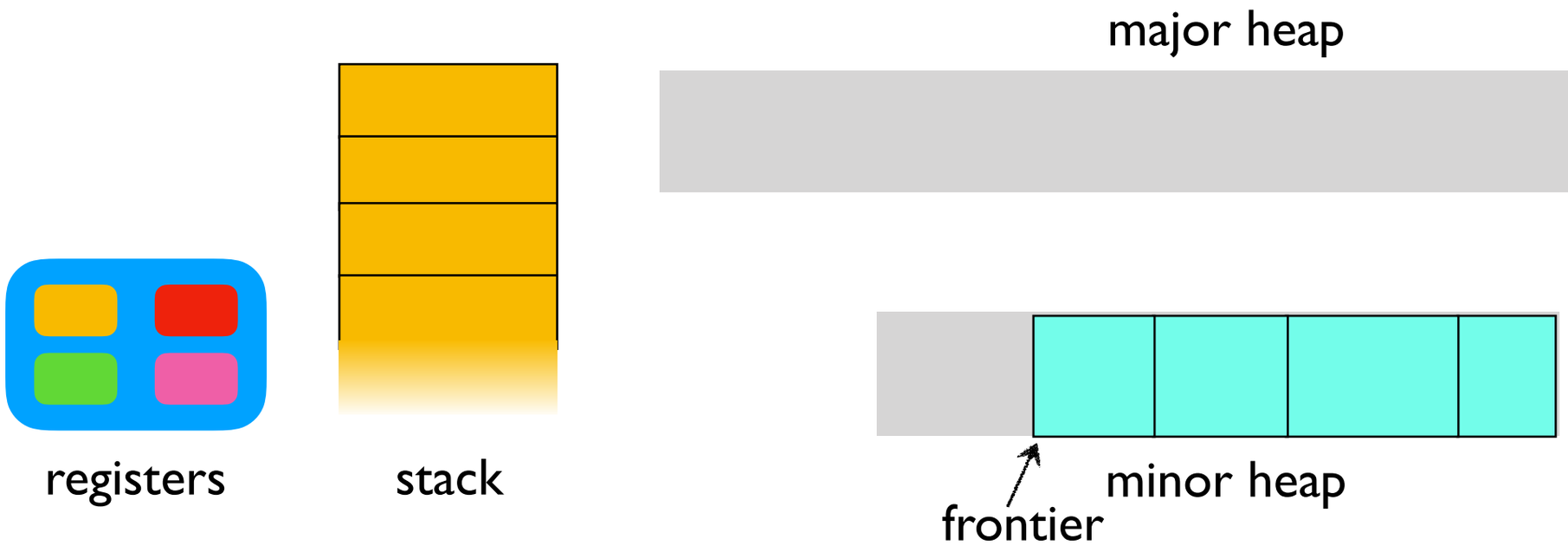
- Generational Hypothesis
  - ✦ Young objects are much more likely to die than old objects



- Minor heap collected by copying collection
  - ✦ Roots are registers and stack
  - ✦ Survivors promoted to major heap

Generational GC

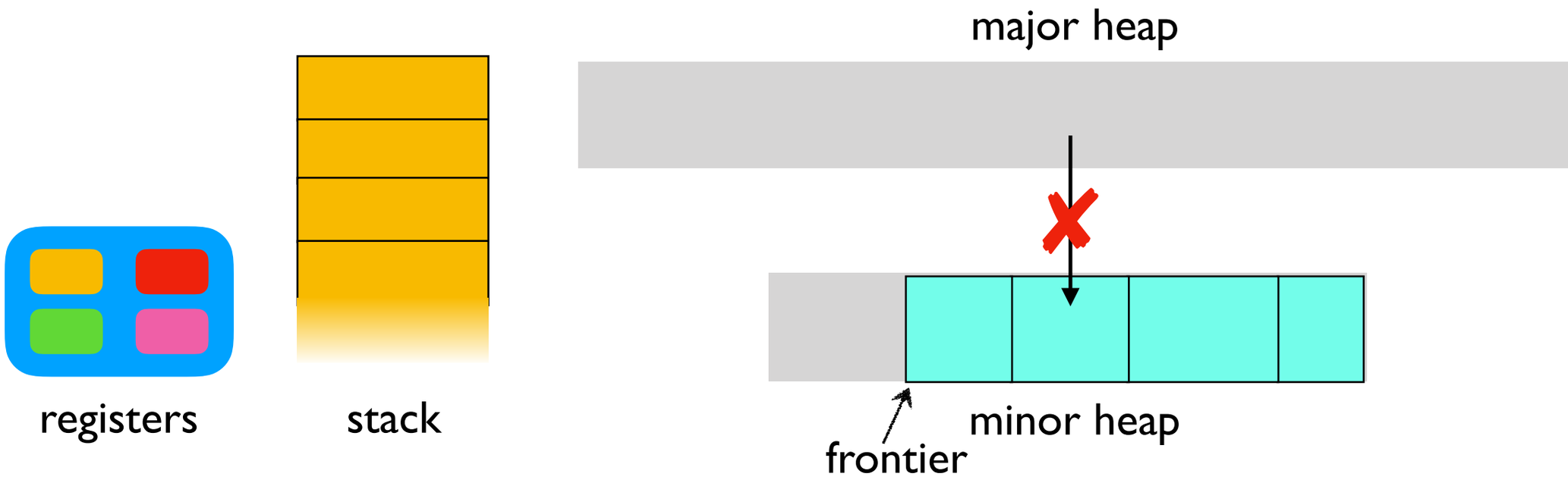
- Generational Hypothesis
  - ✦ Young objects are much more likely to die than old objects



- Minor heap collected by copying collection
  - ✦ Roots are registers and stack
  - ✦ Survivors promoted to major heap
- *Only touches live objects. Typically, < 10% of total. (c.f mark-and-sweep)*

Generational GC

- Generational Hypothesis
  - ✦ Young objects are much more likely to die than old objects



- Minor heap collected by copying collection
  - ✦ Roots are registers and stack
  - ✦ Survivors promoted to major heap
- *Only touches live objects. Typically, < 10% of total. (c.f mark-and-sweep)*
- Purely functional => no major to minor pointers

Mutations

Mutations

- OCaml does not prohibit mutations
  - ✦ Mutable references, Arrays…

Mutations

- OCaml does not prohibit mutations
  - ✦ Mutable references, Arrays…
- *Encourages it with syntactic support!*

Mutations

- OCaml does not prohibit mutations
  - ✦ Mutable references, Arrays…
- *Encourages it with syntactic support!*

```
type client_info =
  { addr: Unix.inet_addr;
    port: int;
    user: string;
    credentials: string;
    mutable last_heartbeat_time: Time.t;
    mutable last_heartbeat_status: string;
  }
```

Mutations

- OCaml does not prohibit mutations
  - ✦ Mutable references, Arrays…
- *Encourages it with syntactic support!*

```
type client_info =
  { addr: Unix.inet_addr;
    port: int;
    user: string;
    credentials: string;
    mutable last_heartbeat_time: Time.t;
    mutable last_heartbeat_status: string;
  }


let handle_heartbeat cinfo time status =
  cinfo.last_heartbeat_time <- time;
  cinfo.last_heartbeat_status <- status
```

Mutations

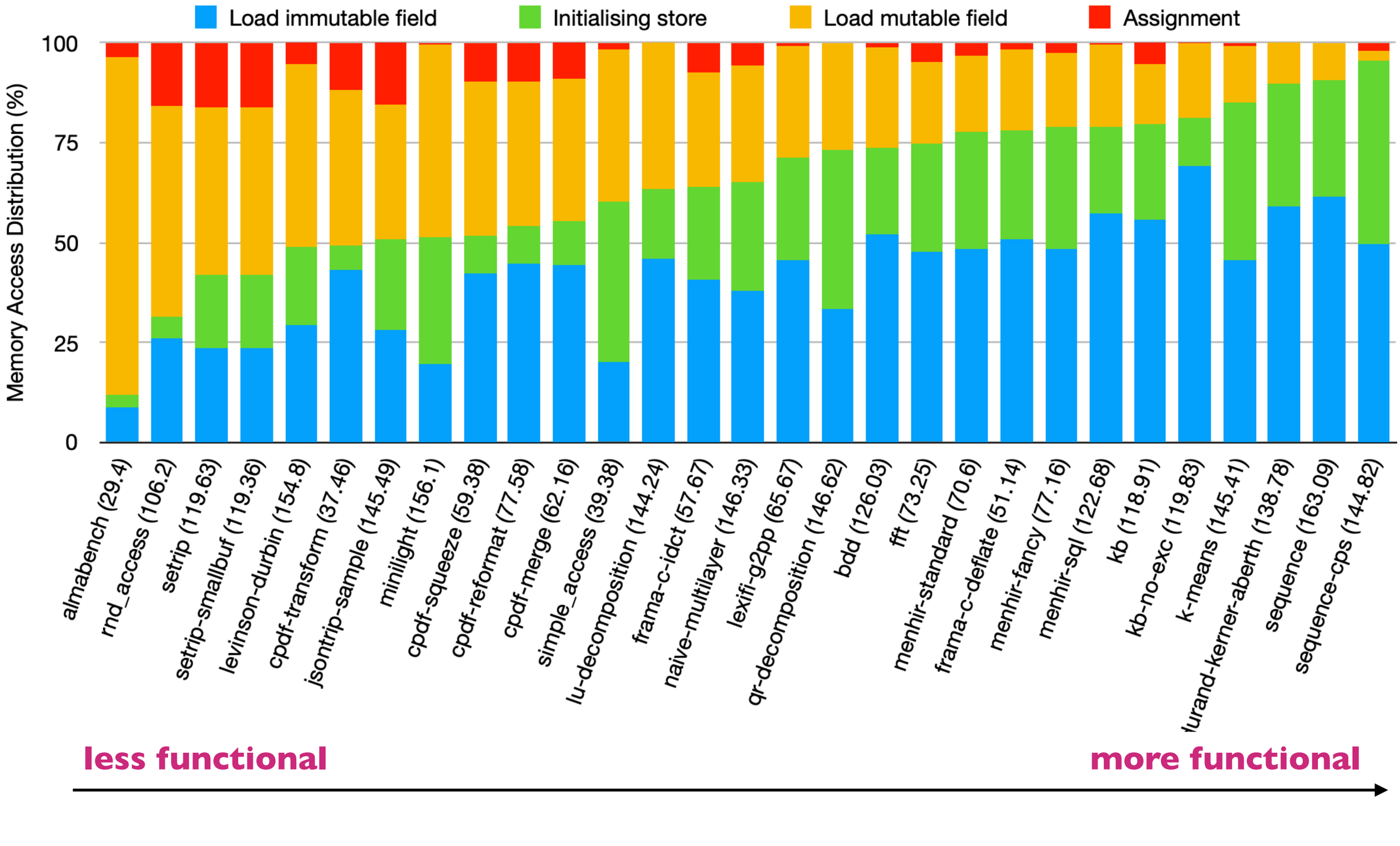
- OCaml does not prohibit mutations
  - ✦ Mutable references, Arrays…
- *Encourages it with syntactic support!*

```
type client_info =
  { addr: Unix.inet_addr;
    port: int;
    user: string;
    credentials: string;
    mutable last_heartbeat_time: Time.t;
    mutable last_heartbeat_status: string;
  }


let handle_heartbeat cinfo time status =
  cinfo.last_heartbeat_time <- time;
  cinfo.last_heartbeat_status <- status
```

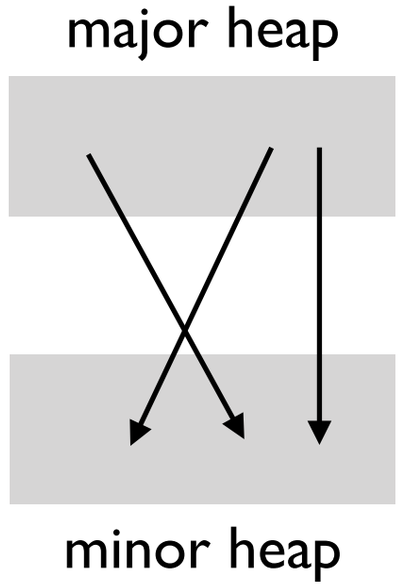
  - ✦ Mutations are pervasive in real-world code

Mutations


Load immutable field
Initialising store
Load mutable field
Assignment
Memory Access Distribution (%)
100
75
50
25
0
almabench (29.4)
rnd_access (106.2)
setrip (119.63)
setrip-smallbuf (119.36)
levinson-durbin (154.8)
cpdf-transform (37.46)
jsontrip-sample (145.49)
minilight (156.1)
cpdf-squeeze (59.38)
cpdf-reformat (77.58)
cpdf-merge (62.16)
simple_access (39.38)
lu-decomposition (144.24)
frama-c-idct (57.67)
naive-multilayer (146.33)
lexifi-g2pp (65.67)
qr-decomposition (146.62)
bdd (126.03)
fft (73.25)
menhir-standard (70.6)
frama-c-deflate (51.14)
menhir-fancy (77.16)
menhir-sql (122.68)
kb (118.91)
kb-no-exc (119.83)
k-means (145.41)
lurand-kerner-aberth (138.78)
sequence (163.09)
sequence-cps (144.82)
less functional
more functional

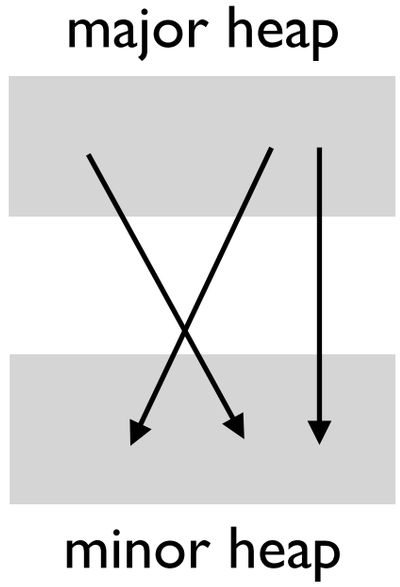
Mutations — Minor GC

- Old objects might point to young objects



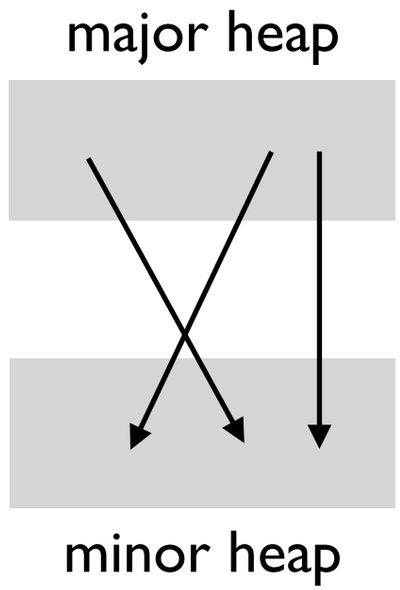
Mutations — Minor GC

- Old objects might point to young objects
- Must know those pointers for minor GC
  - ✦ (Naively) scan the major GC for such pointers



Mutations — Minor GC

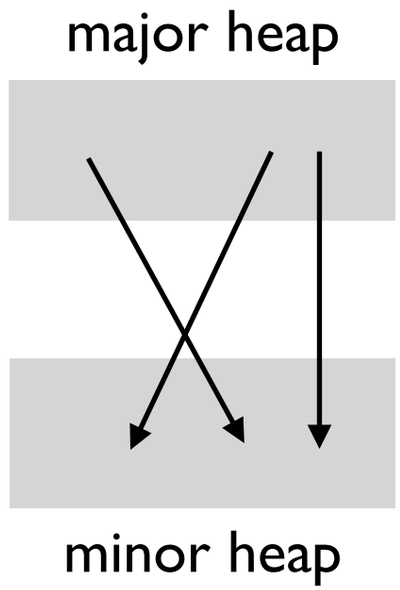
- Old objects might point to young objects
- Must know those pointers for minor GC
  - ✦ (Naively) scan the major GC for such pointers
- Intercept mutations with write barrier




```
(* Before r := x *)
let write_barrier (r, x) =
  if is_major r && is_minor x then
    remembered_set.add r
```

Mutations — Minor GC

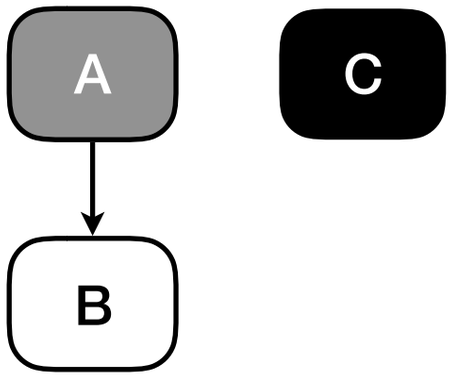
- Old objects might point to young objects
- Must know those pointers for minor GC
  - ✦ (Naively) scan the major GC for such pointers
- Intercept mutations with write barrier



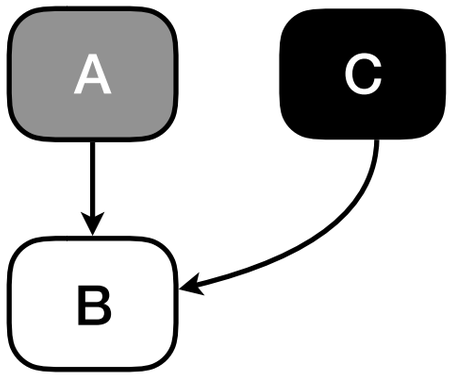

```
(* Before r := x *)
let write_barrier (r, x) =
  if is_major r && is_minor x then
    remembered_set.add r
```

- *Remembered set*
  - ✦ Set of major heap addresses that point to minor heap
  - ✦ Used as root for minor collection
  - ✦ Cleared after minor collection.

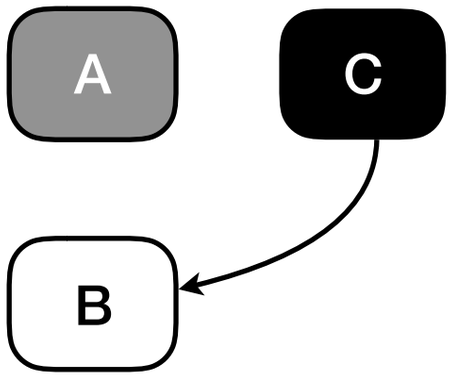
Mutations — Major GC



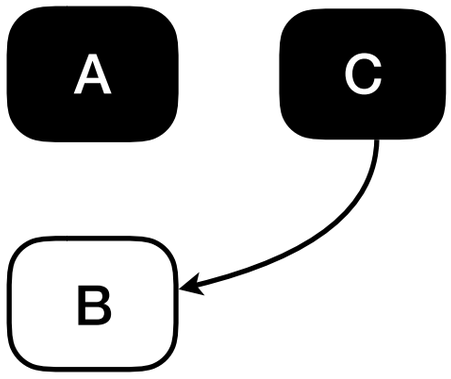
Mutations — Major GC



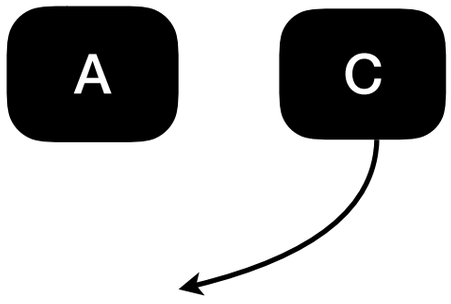
Mutations — Major GC



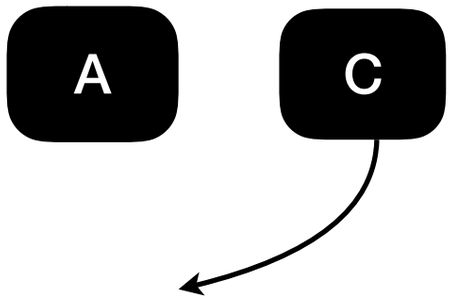
Mutations — Major GC



Mutations — Major GC

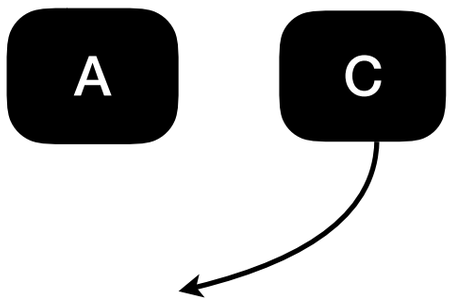


Mutations — Major GC



- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted

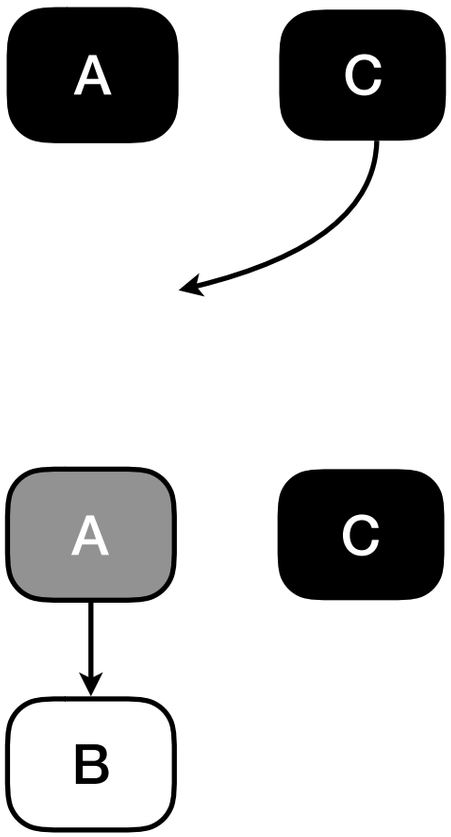
Mutations — Major GC



- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted

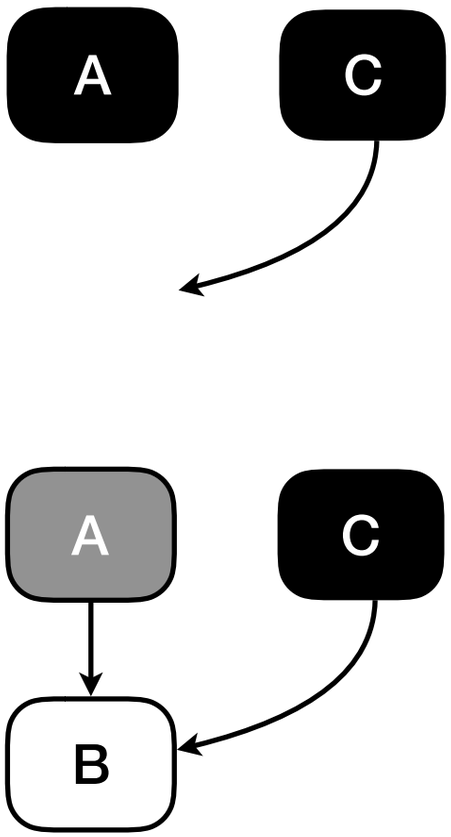
- Insertion/Dijkstra/Incremental barrier prevents 1

Mutations — Major GC



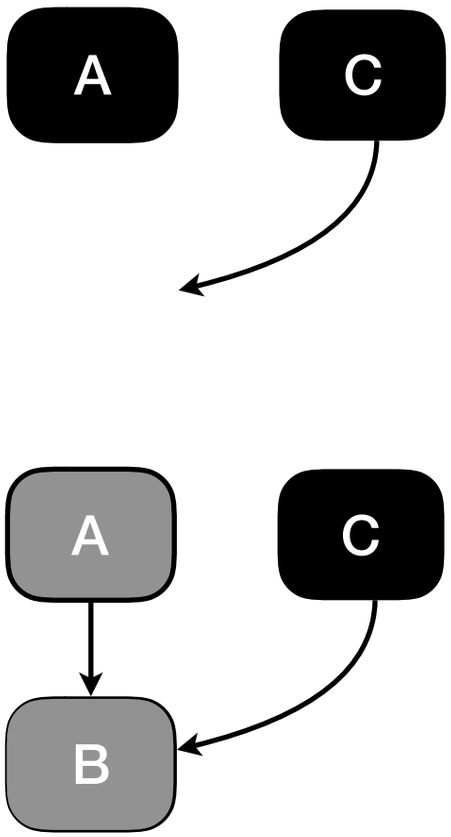
- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted
- Insertion/Dijkstra/Incremental barrier prevents 1

Mutations — Major GC



- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted
- Insertion/Dijkstra/Incremental barrier prevents 1

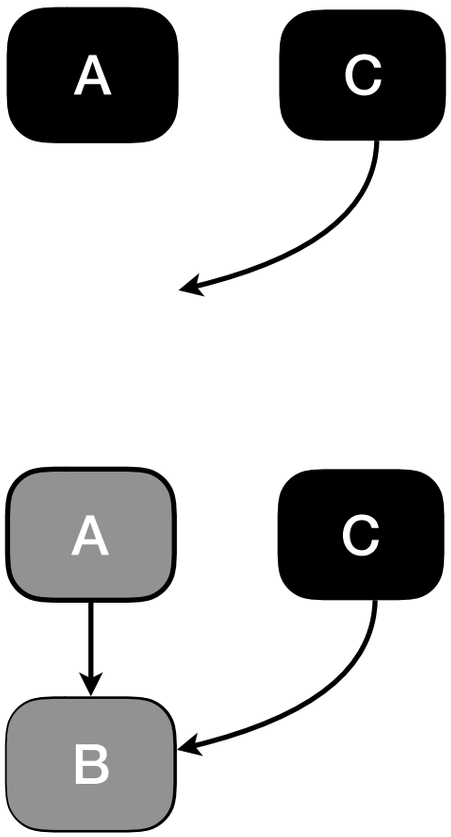
Mutations — Major GC



- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted

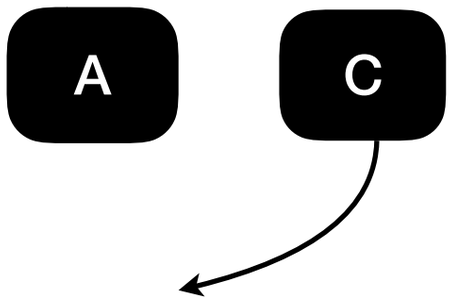
- Insertion/Dijkstra/Incremental barrier prevents 1

Mutations — Major GC

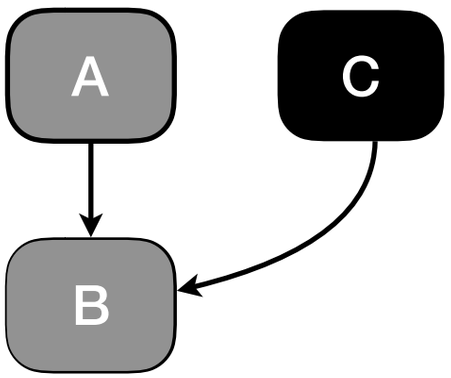


- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted
- Insertion/Dijkstra/Incremental barrier prevents 1
- Deletion/Yuasa/snapshot-at-beginning prevents 2

Mutations — Major GC

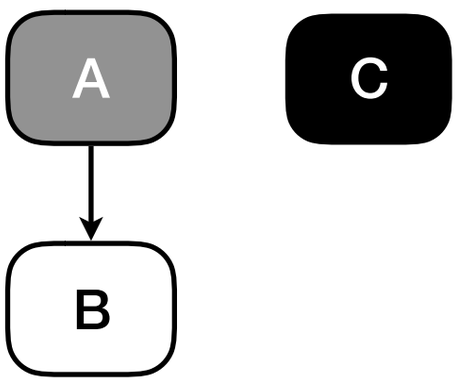


- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted

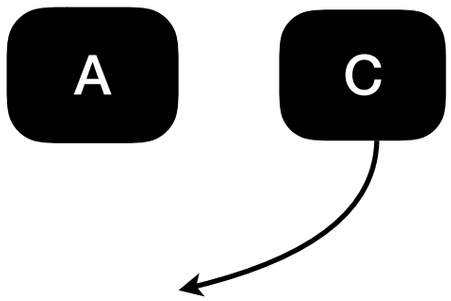


- Insertion/Dijkstra/Incremental barrier prevents 1

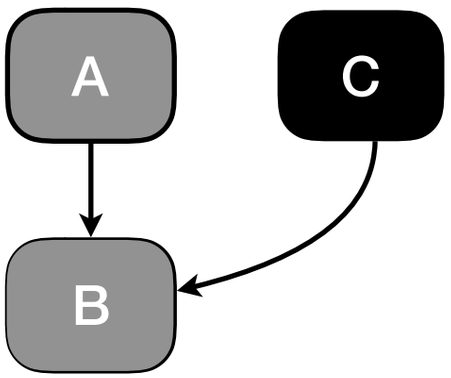
- Deletion/Yuasa/snapshot-at-beginning prevents 2



Mutations — Major GC

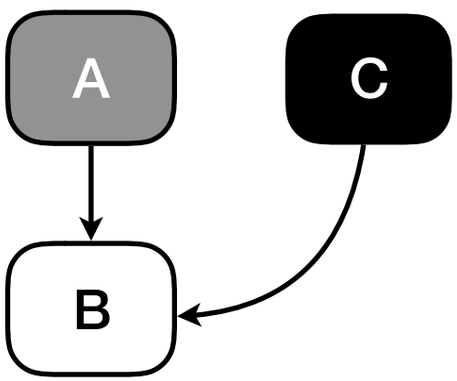


- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted

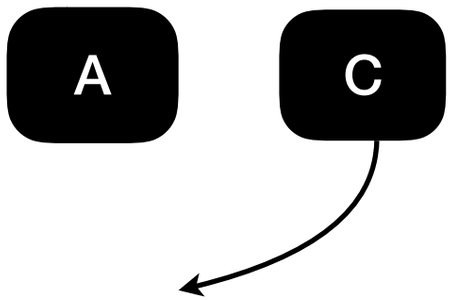


- Insertion/Dijkstra/Incremental barrier prevents 1

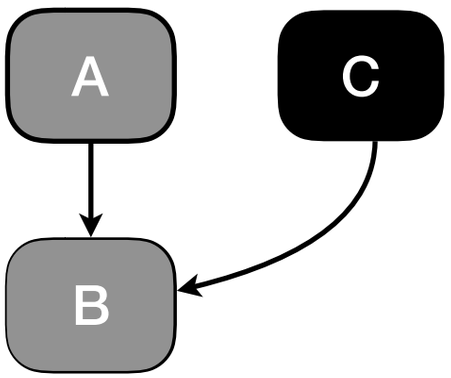
- Deletion/Yuasa/snapshot-at-beginning prevents 2



Mutations — Major GC

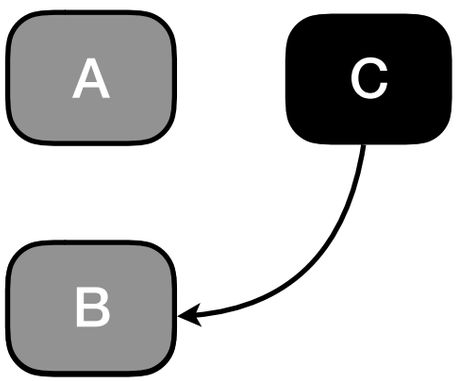


- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted

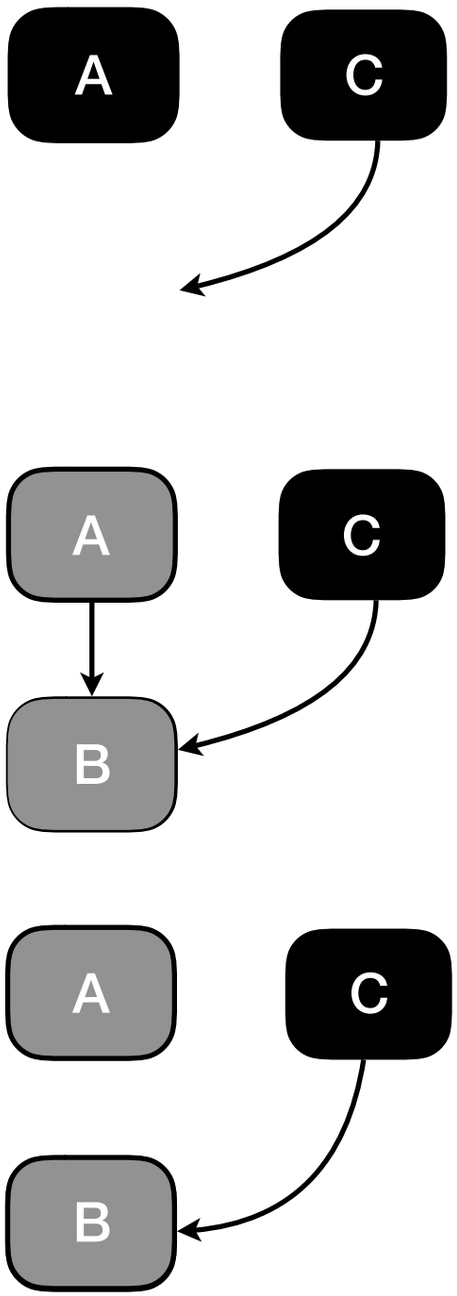


- Insertion/Dijkstra/Incremental barrier prevents 1

- Deletion/Yuasa/snapshot-at-beginning prevents 2

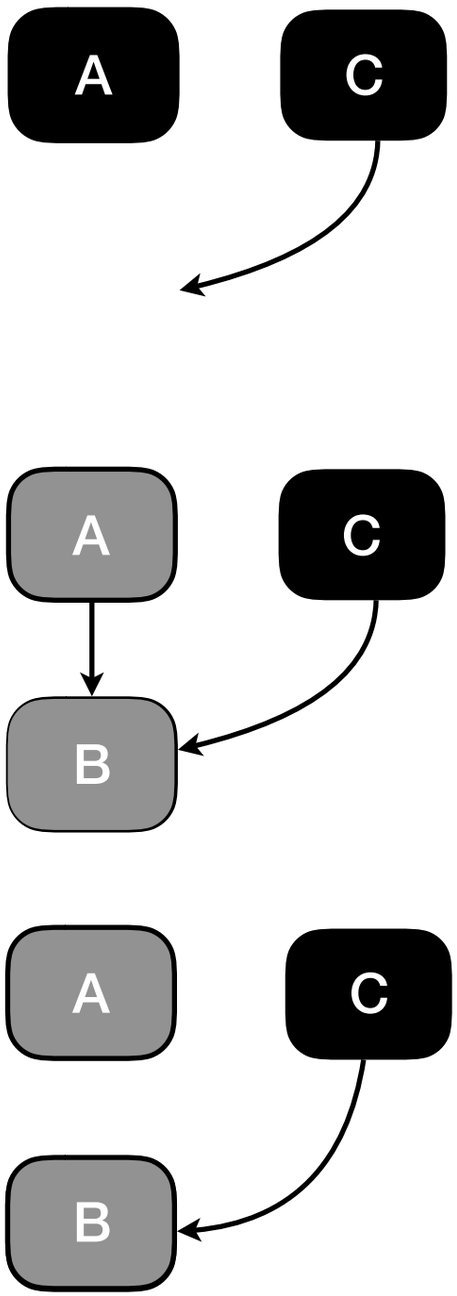


Mutations — Major GC



- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted
- Insertion/Dijkstra/Incremental barrier prevents 1
- Deletion/Yuasa/snapshot-at-beginning prevents 2
  - ✦ Amount of marking work in a cycle is fixed (snapshot-at-the-beginning barrier.

Mutations — Major GC



- Mutations are problematic if both conditions hold
  1. Exists Black —> White
  2. All Grey —> White* —> White paths are deleted
- Insertion/Dijkstra/Incremental barrier prevents 1
- Deletion/Yuasa/snapshot-at-beginning prevents 2
  - ✦ Amount of marking work in a cycle is fixed (snapshot-at-the-beginning barrier.

```
(* Before r := x *)
let write_barrier (r, x) =
  if is_major r && is_minor x then
    remembered_set.add r
  else if is_major r && is_major x then
    mark(!r)
```

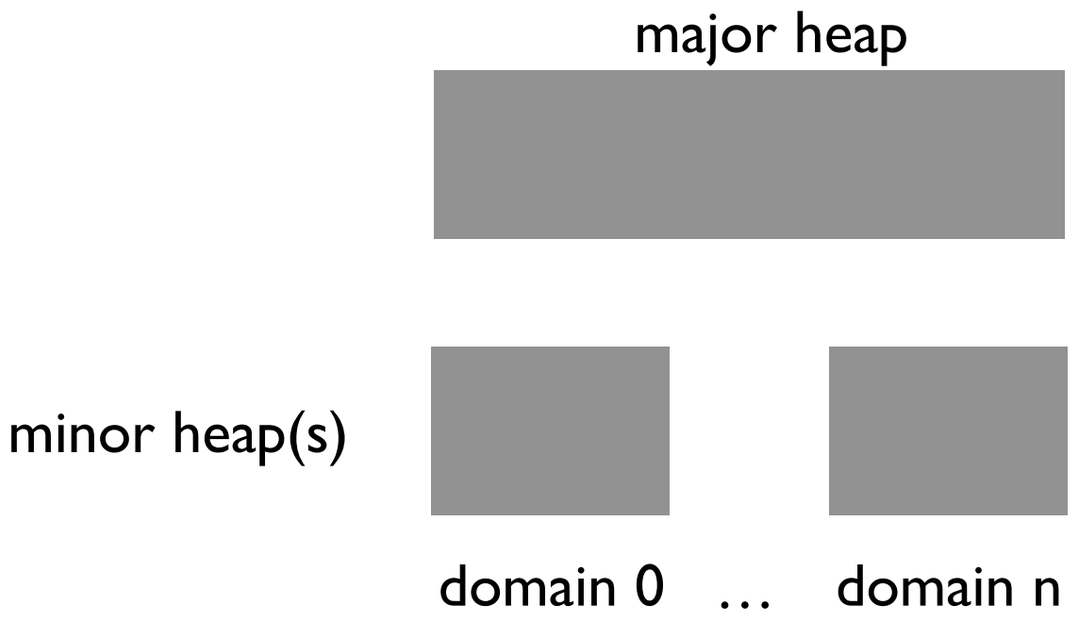
Parallelism — Minor GC

Parallelism — Minor GC

- `Domain.spawn : (unit -> unit) -> unit`

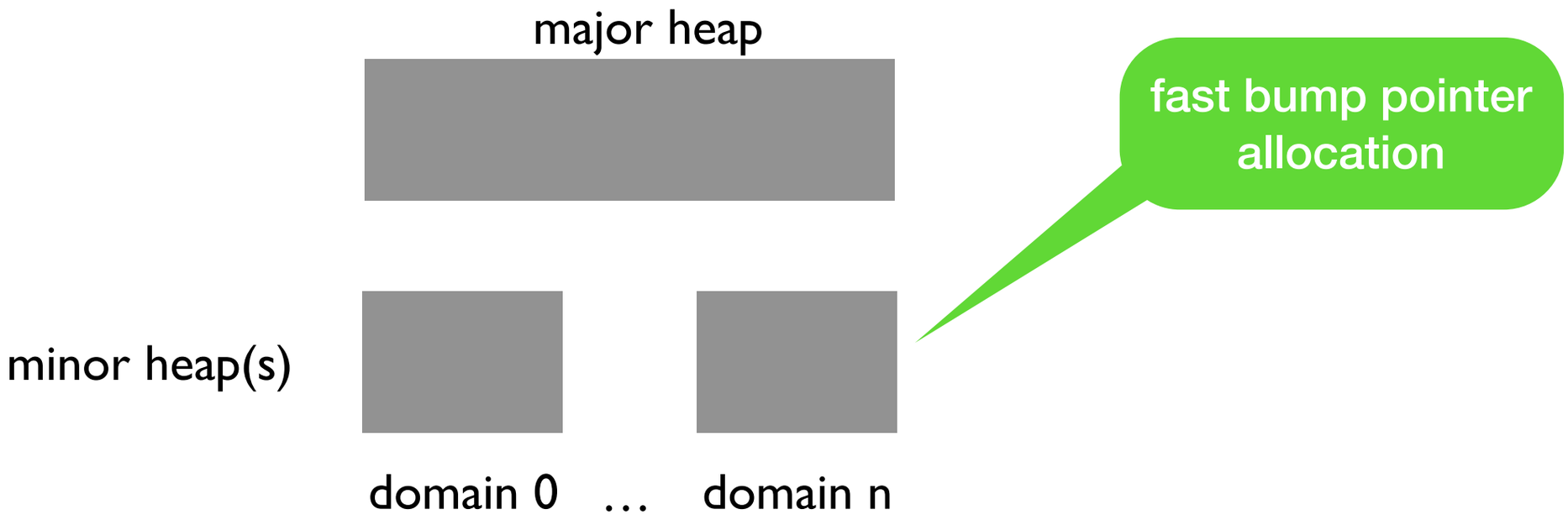
Parallelism — Minor GC

- `Domain.spawn : (unit -> unit) -> unit`



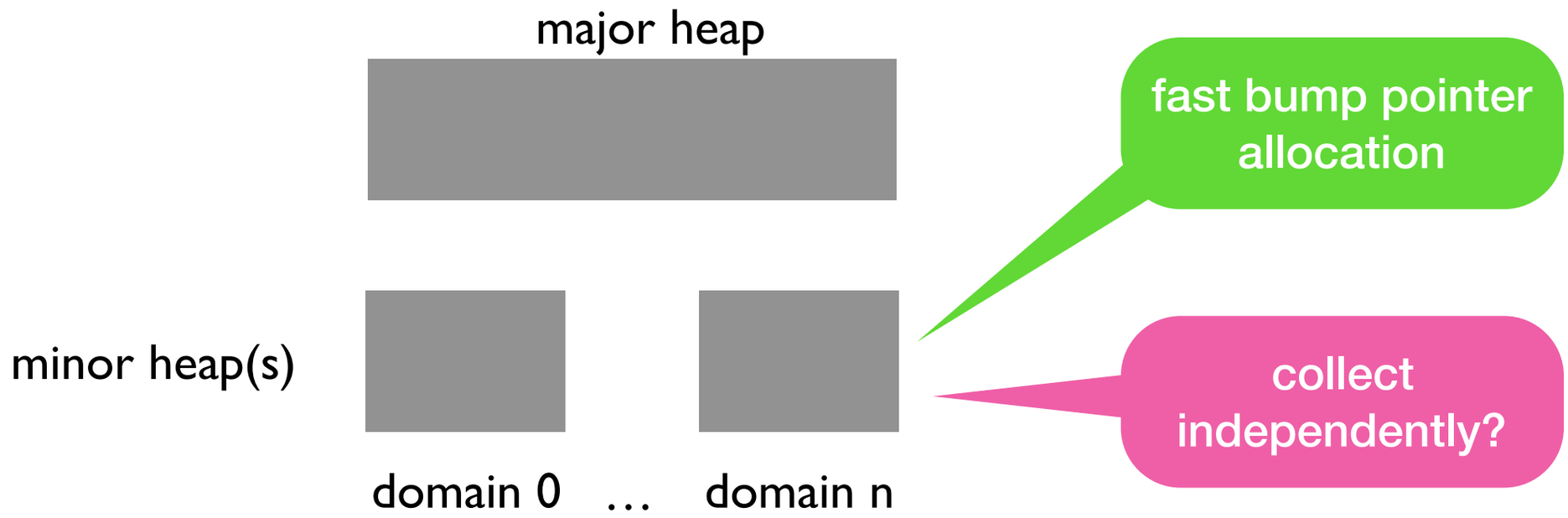
Parallelism — Minor GC

- `Domain.spawn : (unit -> unit) -> unit`



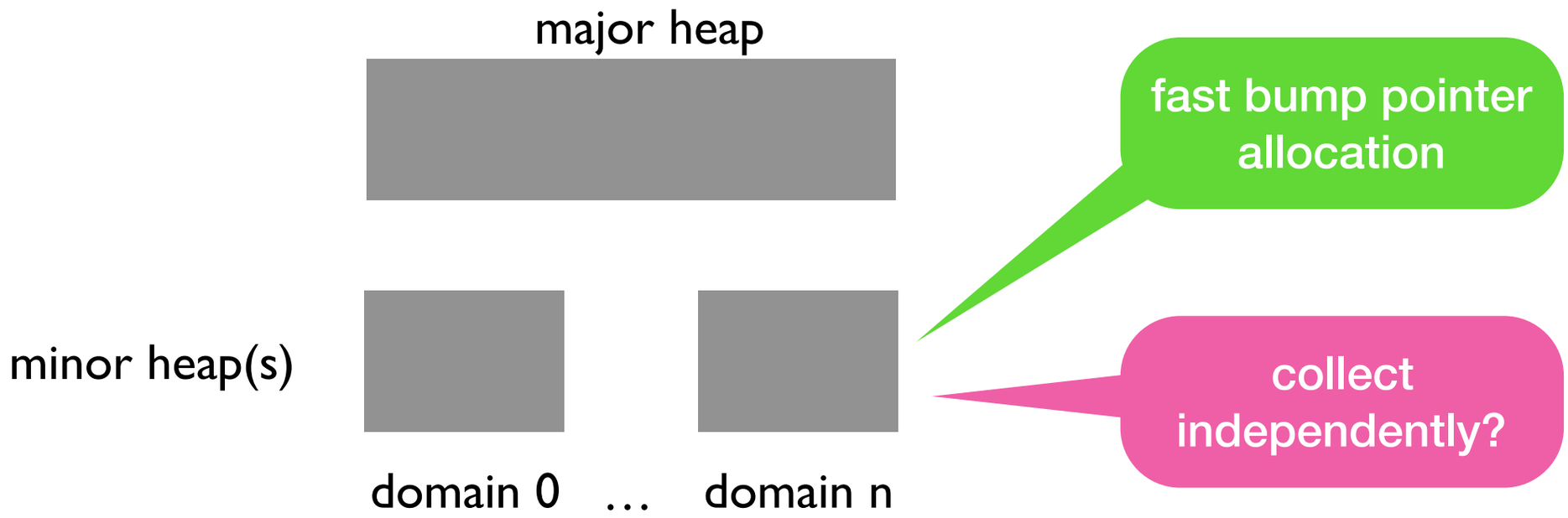
Parallelism — Minor GC

- `Domain.spawn : (unit -> unit) -> unit`



Parallelism — Minor GC

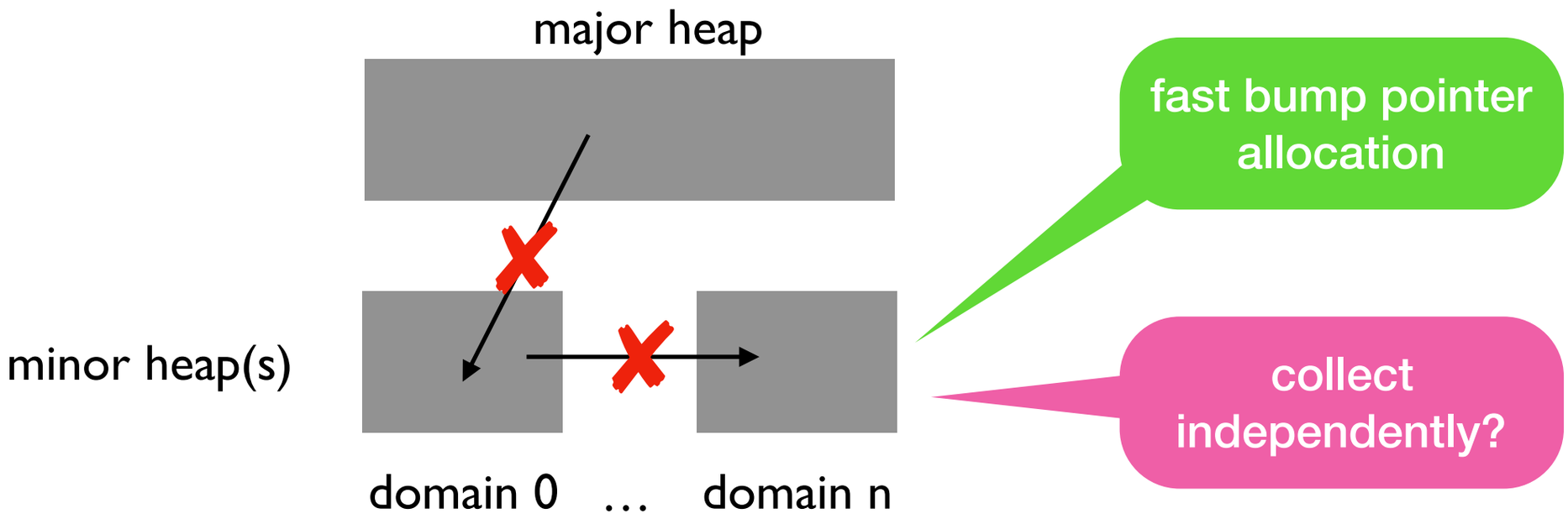
- `Domain.spawn : (unit -> unit) -> unit`



- Invariant: Minor heap objects are only accessed by owning domain

Parallelism — Minor GC

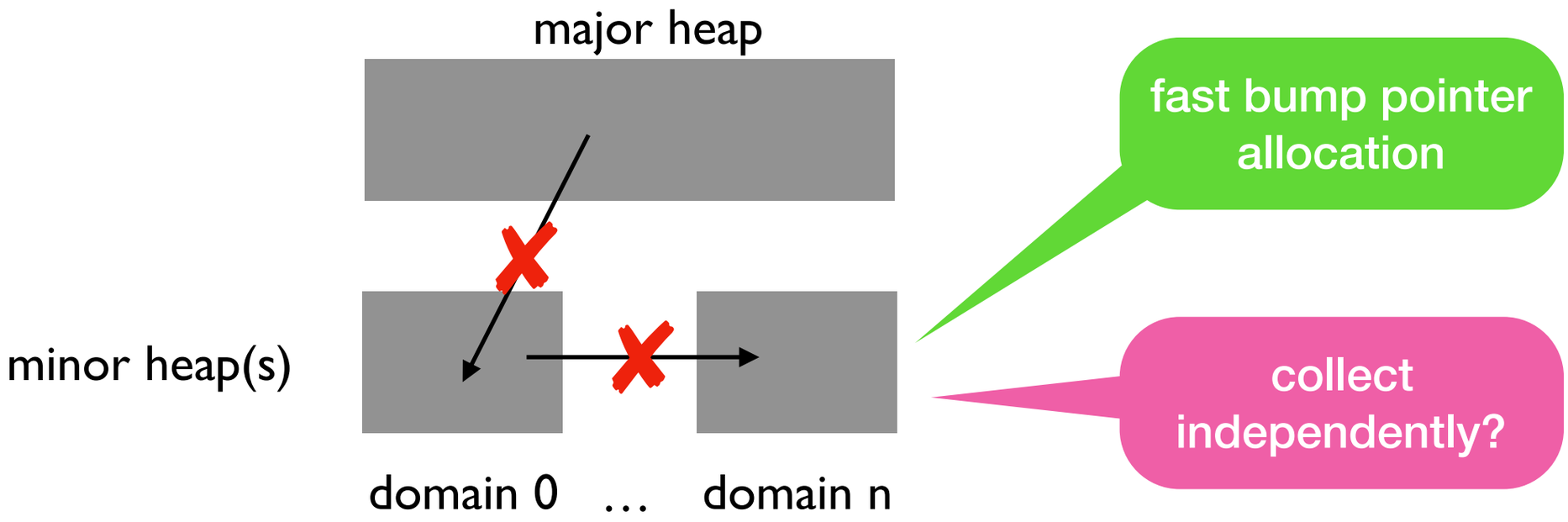
- `Domain.spawn : (unit -> unit) -> unit`



- Invariant: Minor heap objects are only accessed by owning domain
- Doligez-Leroy POPL'93
  - ✦ No pointers between minor heaps
  - ✦ No pointers from major to minor heaps

Parallelism — Minor GC

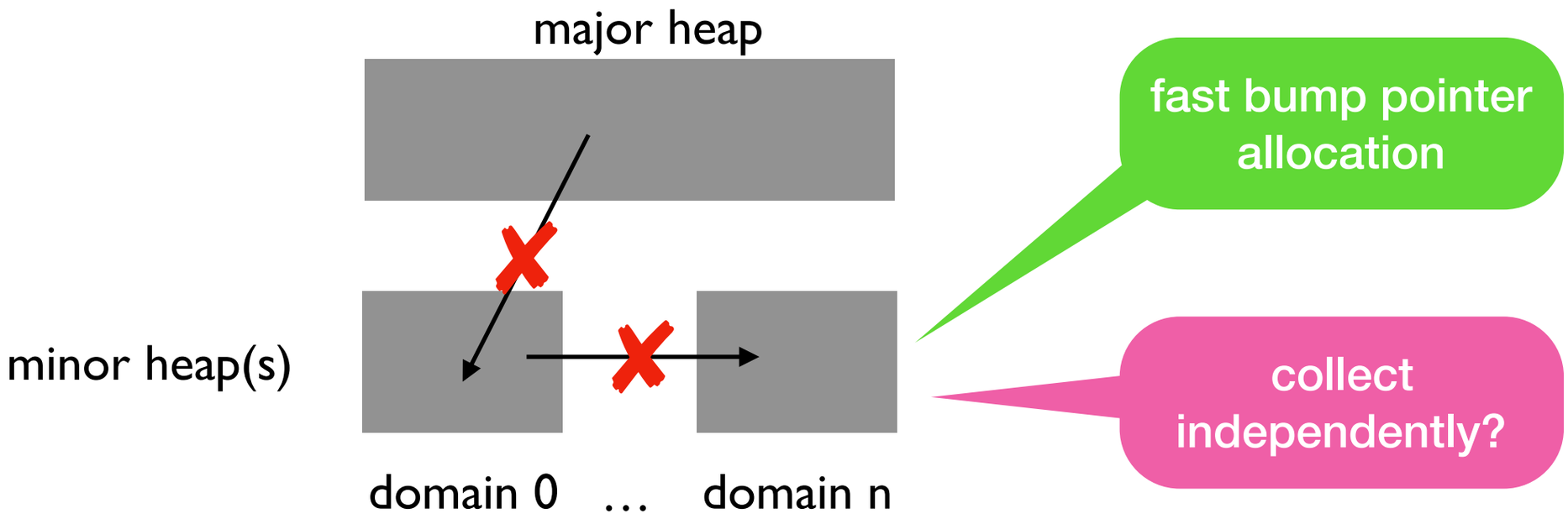
- `Domain.spawn : (unit -> unit) -> unit`



- Invariant: Minor heap objects are only accessed by owning domain
- Doligez-Leroy POPL'93
  - ✦ No pointers between minor heaps
  - ✦ No pointers from major to minor heaps
- Before r := x, if is_major(r) && is_minor(x), then promote(x).

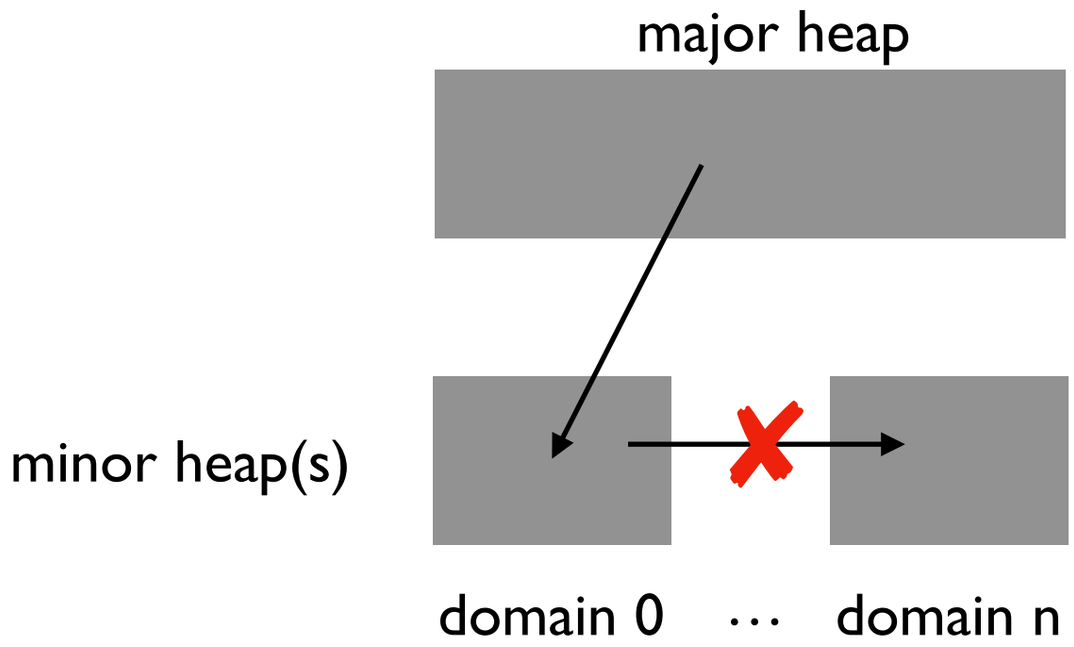
Parallelism — Minor GC

- `Domain.spawn : (unit -> unit) -> unit`

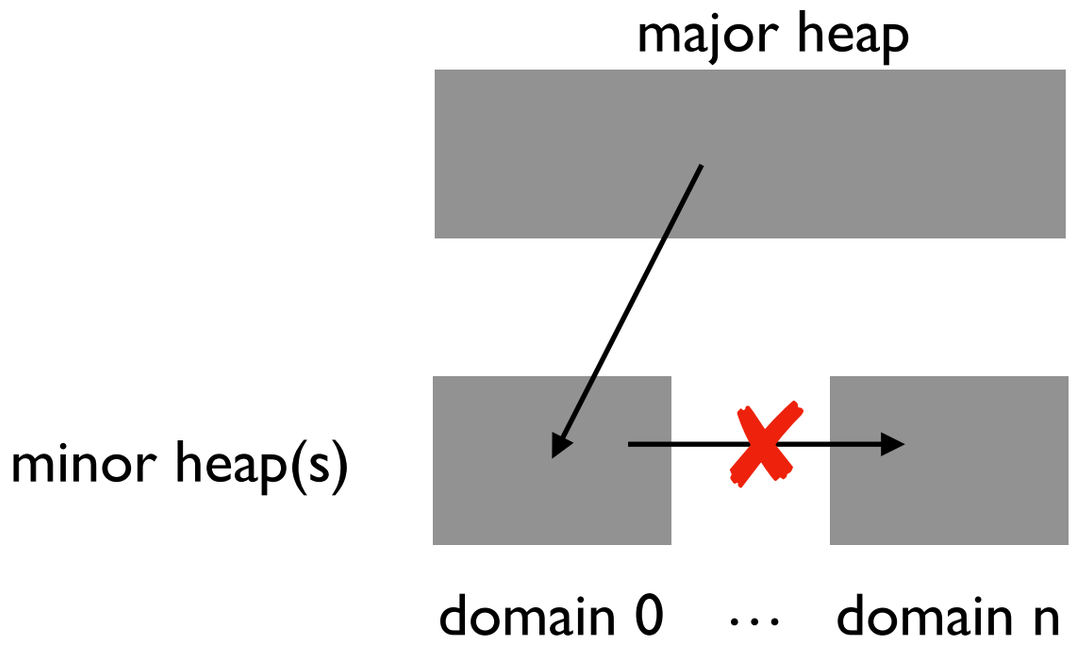


- Invariant: Minor heap objects are only accessed by owning domain
- Doligez-Leroy POPL'93
  - ✦ No pointers between minor heaps
  - ✦ No pointers from major to minor heaps
- Before r := x, if is_major(r) && is_minor(x), then promote(x).
- *Too much promotion.* Ex: work-stealing queue

Parallelism — Minor GC

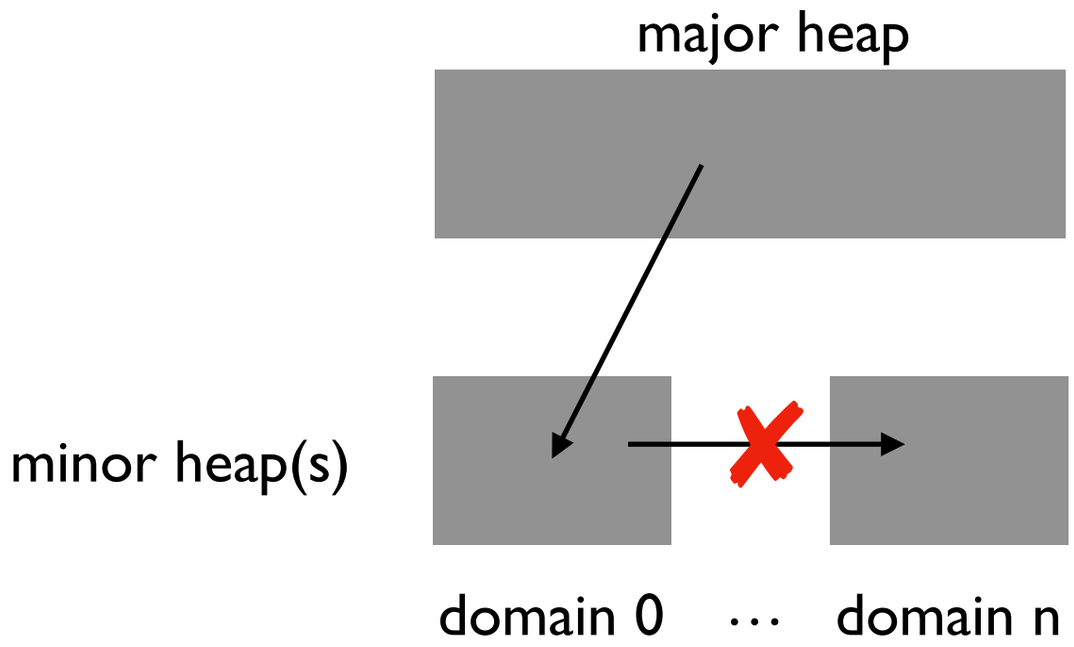


Parallelism — Minor GC



- Weaker invariant
  - ✦ No pointers between minor heaps
  - ✦ *Objects in foreign minor heap are not accessed directly*

Parallelism — Minor GC



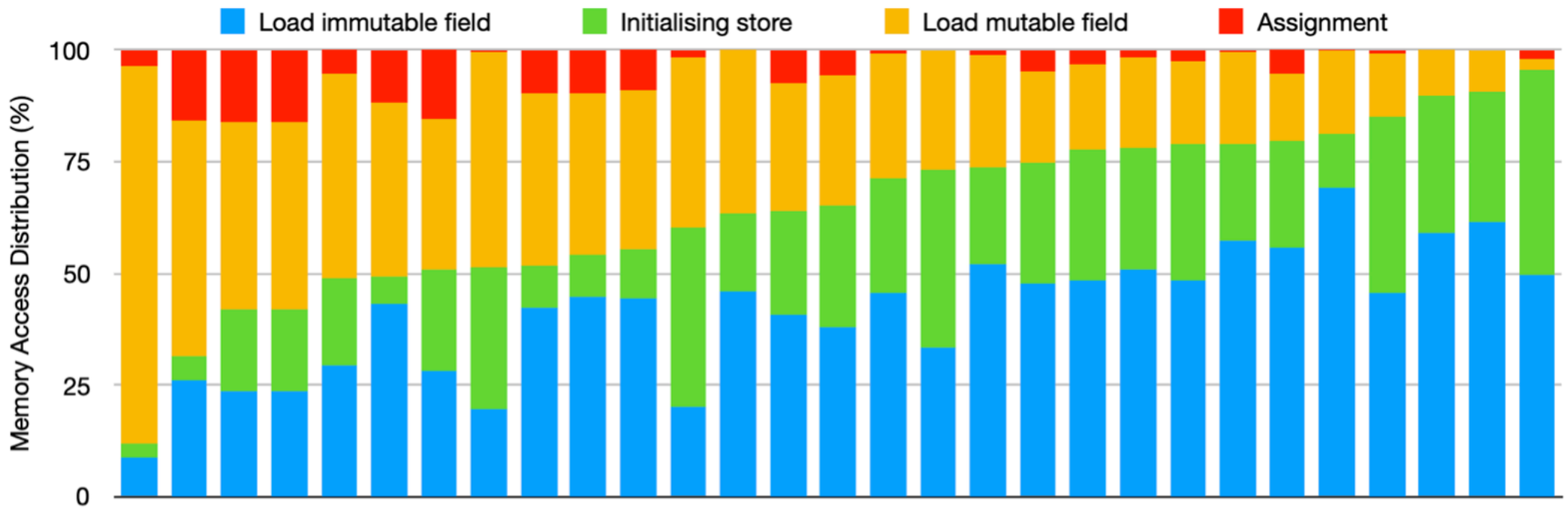
- Weaker invariant
  - ✦ No pointers between minor heaps
  - ✦ *Objects in foreign minor heap are not accessed directly*
- Read barrier. If the value loaded is
  - ✦ integers, object in shared heap or own minor heap => continue
  - ✦ object in foreign minor heap => *Read fault (Interrupt + promote)*

Efficient Read Barrier Check

- Given an address `x`, quickly compute `is_remote_minor(x)`

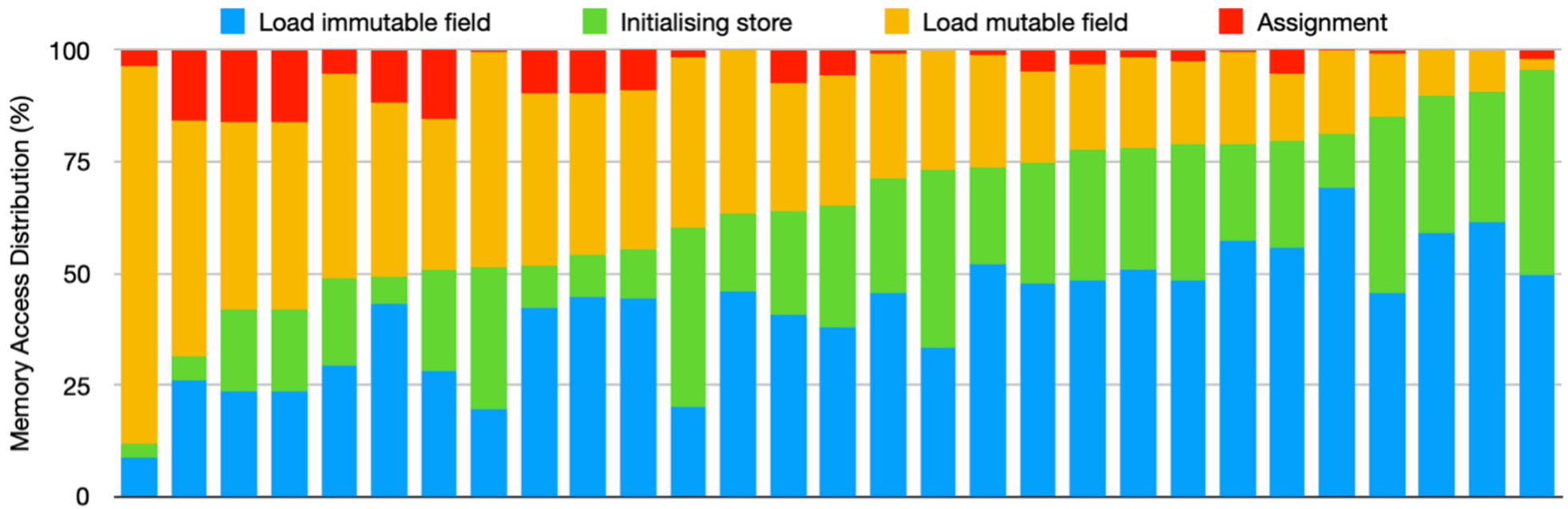
Efficient Read Barrier Check

- Given an address `x`, quickly compute `is_remote_minor(x)`



Efficient Read Barrier Check

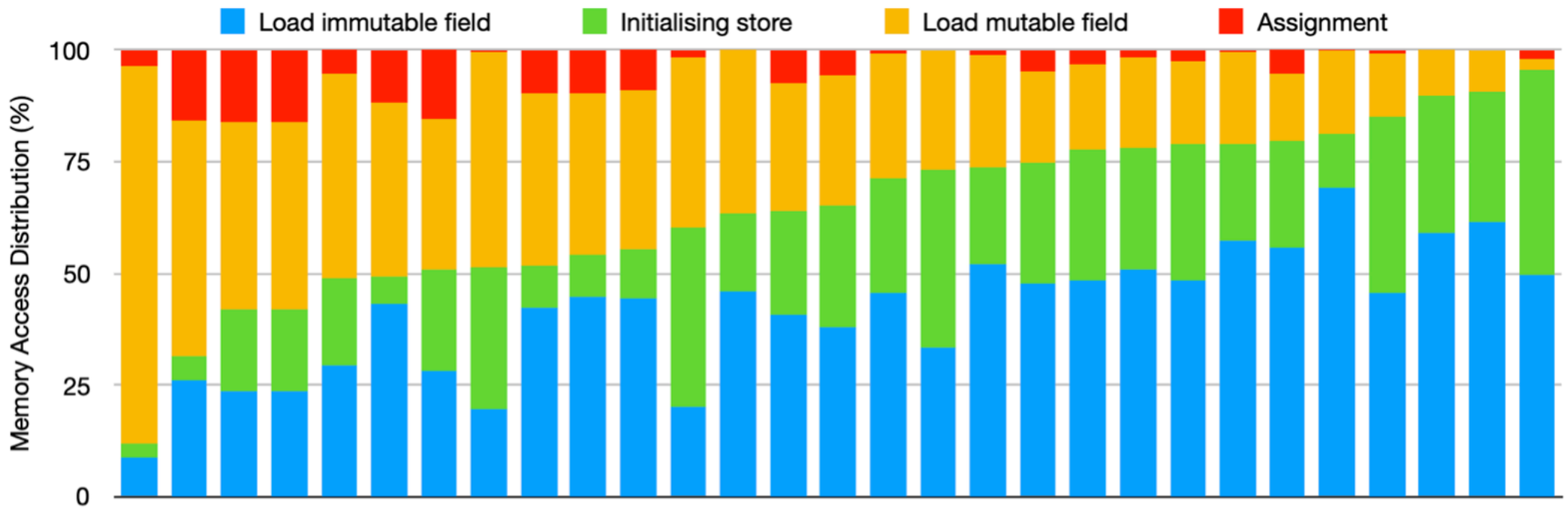
- Given an address `x`, quickly compute `is_remote_minor(x)`




```
# %rax holds x (value of interest)
xor %r15, %rax
sub 0x0010, %rax
test 0xff01, %rax
# ZF set => foreign minor
```

Efficient Read Barrier Check

- Given an address `x`, quickly compute `is_remote_minor(x)`




```
# %rax holds x (value of interest)
xor %r15, %rax
sub 0x0010, %rax
test 0xff01, %rax
# ZF set => foreign minor
```

Deep dive into Multicore OCaml Garbage Collector
http://kcsrk.info/multicore/gc/2017/07/06/multicore-ocaml-gc/

Parallelism — Major GC

Parallelism — Major GC

- OCaml's GC is *incremental*

Parallelism — Major GC

- OCaml's GC is *incremental*

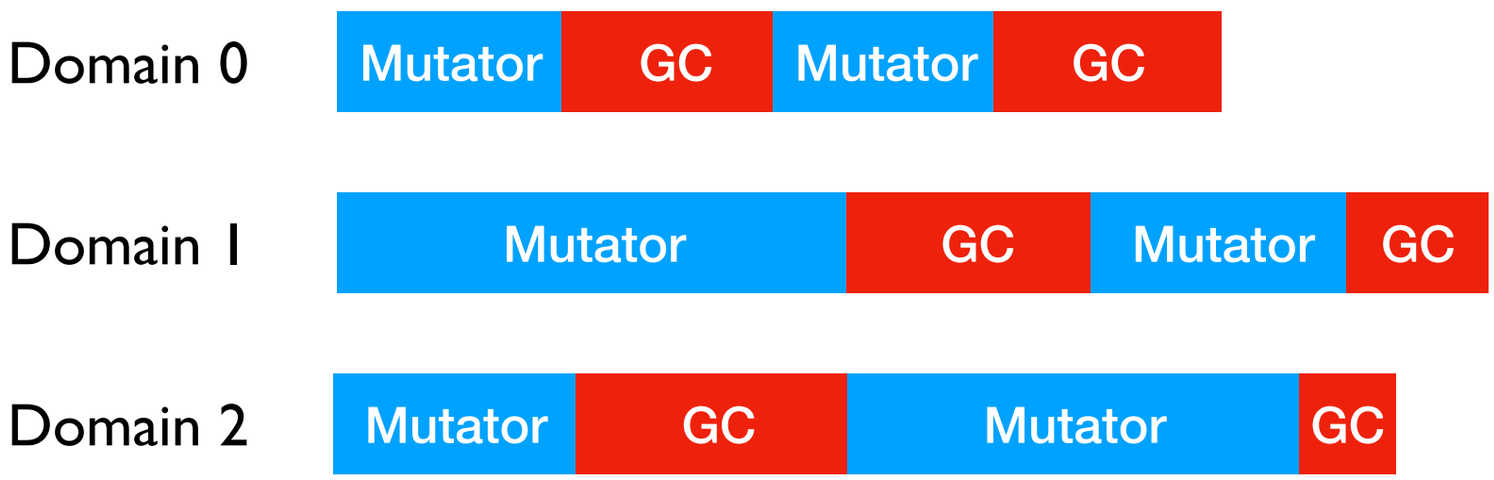
  - ✦ Incrementality minimises GC pauses
  - ✦ *Parallel (stop-the-world) collectors is not an option due to latency concerns*

Parallelism — Major GC

- OCaml's GC is *incremental*

  - ✦ Incrementality minimises GC pauses
  - ✦ *Parallel (stop-the-world) collectors is not an option due to latency concerns*

- Multicore OCaml's GC should be *concurrent (and incremental)*



Parallelism — Major GC

Parallelism — Major GC

- Design based on VCGC from Inferno project (ISMM'98)
  - ✦ Allows mutator, marker, sweeper threads to concurrently

Parallelism — Major GC

- Design based on VCGC from Inferno project (ISMM'98)
  - ✦ Allows mutator, marker, sweeper threads to concurrently
- In Multicore OCaml,
  - ✦ States Unmarked Marked Garbage Free

Parallelism — Major GC

- Design based on VCGC from Inferno project (ISMM'98)
  - ✦ Allows mutator, marker, sweeper threads to concurrently
- In Multicore OCaml,
  - ✦ States Unmarked Marked Garbage Free
  - ✦ Domains alternate between mutator and gc thread

Parallelism — Major GC

- Design based on VCGC from Inferno project (ISMM'98)
  - ✦ Allows mutator, marker, sweeper threads to concurrently
- In Multicore OCaml,
  - ✦ States Unmarked Marked Garbage Free
  - ✦ Domains alternate between mutator and gc thread
  - ✦ Marking: Unmarked → Marked Sweeping: Garbage → Free

Parallelism — Major GC

- Design based on VCGC from Inferno project (ISMM'98)
  - ✦ Allows mutator, marker, sweeper threads to concurrently
- In Multicore OCaml,
  - ✦ States Unmarked Marked Garbage Free
  - ✦ Domains alternate between mutator and gc thread
  - ✦ Marking: Unmarked → Marked Sweeping: Garbage → Free
  - ✦ Marking is *racy* but *idempotent*

Parallelism — Major GC

- Design based on VCGC from Inferno project (ISMM'98)
  - ✦ Allows mutator, marker, sweeper threads to concurrently
- In Multicore OCaml,
  - ✦ States Unmarked Marked Garbage Free
  - ✦ Domains alternate between mutator and gc thread
  - ✦ Marking: Unmarked → Marked Sweeping: Garbage → Free
  - ✦ Marking is *racy* but *idempotent*
- Marking & Sweeping done ⇒ stop-the-world

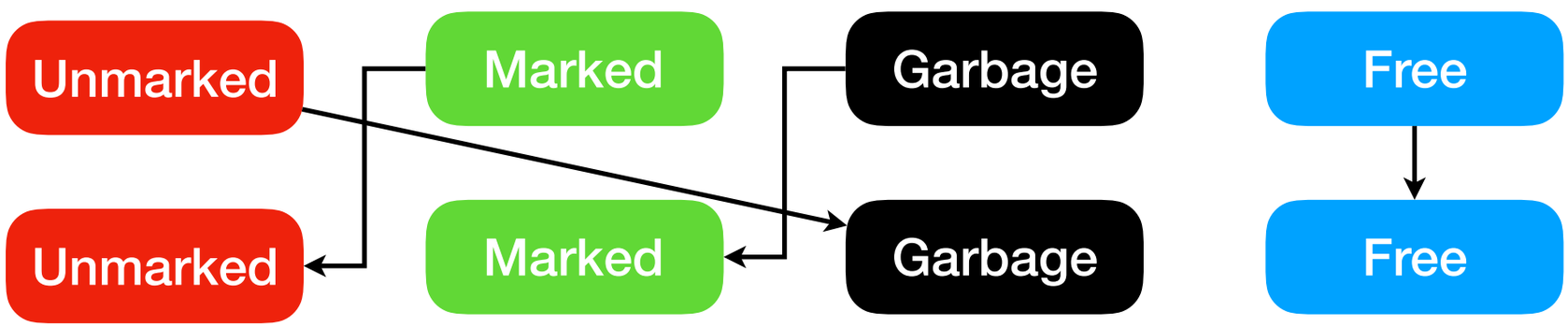
Parallelism — Major GC

- Design based on VCGC from Inferno project (ISMM'98)
  - ✦ Allows mutator, marker, sweeper threads to concurrently
- In Multicore OCaml,
  - ✦ States Unmarked Marked Garbage Free
  - ✦ Domains alternate between mutator and gc thread
  - ✦ Marking: Unmarked → Marked Sweeping: Garbage → Free
  - ✦ Marking is *racy* but *idempotent*
- Marking & Sweeping done ⇒ stop-the-world

Parallelism — Major GC

- Design based on VCGC from Inferno project (ISMM'98)
  - ✦ Allows mutator, marker, sweeper threads to concurrently
- In Multicore OCaml,
  - ✦ States Unmarked Marked Garbage Free
  - ✦ Domains alternate between mutator and gc thread
  - ✦ Marking: Unmarked → Marked Sweeping: Garbage → Free
  - ✦ Marking is *racy* but *idempotent*
- Marking & Sweeping done ⇒ stop-the-world

Yuasa!



Concurrency

Concurrency

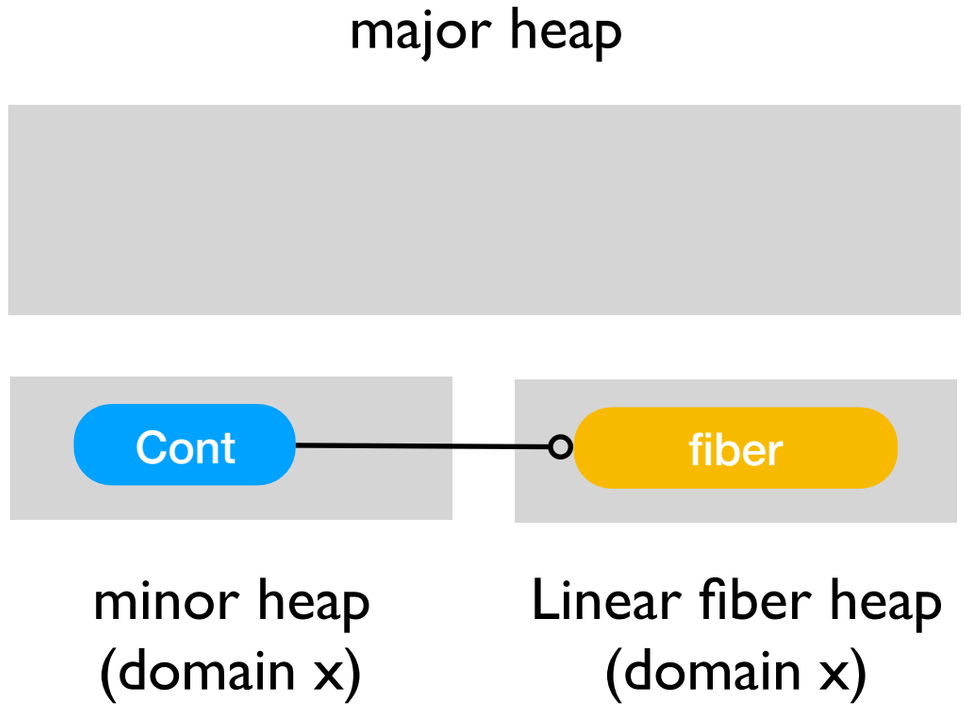
- Fibers: vm-threads, *linear* delimited continuations

Concurrency

- Fibers: vm-threads, *linear* delimited continuations
- Stack segments managed on the heap

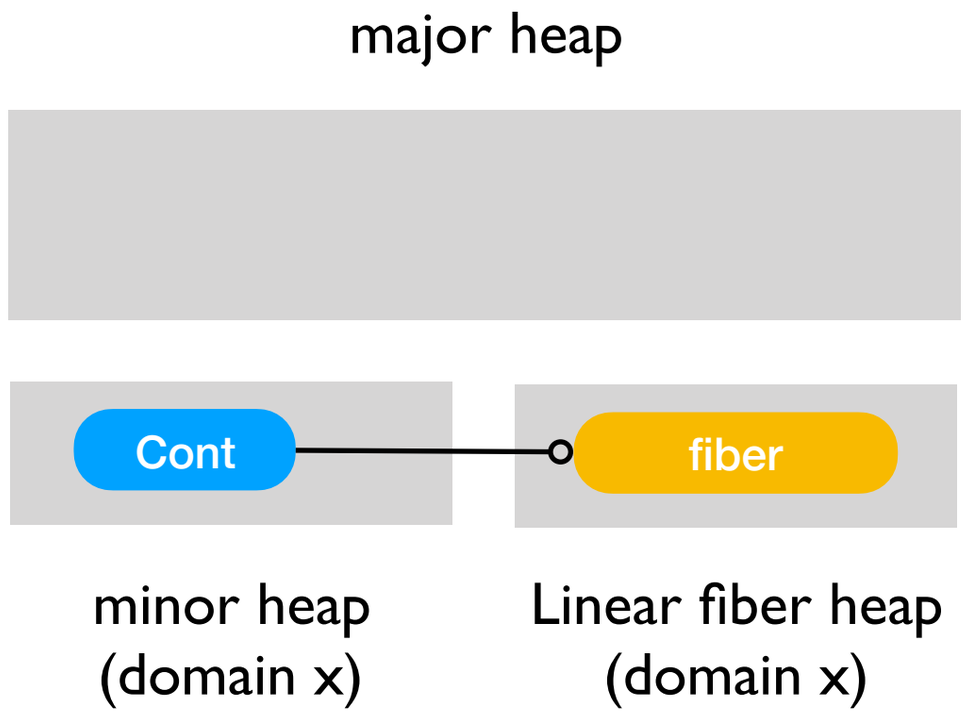
Concurrency

- Fibers: vm-threads, *linear* delimited continuations
- Stack segments managed on the heap



Concurrency

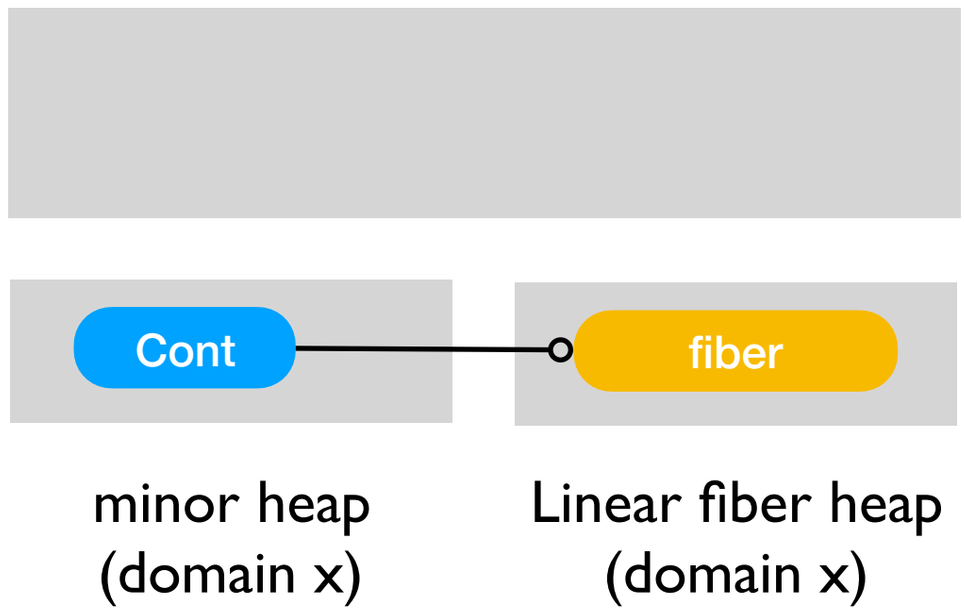
- Fibers: vm-threads, *linear* delimited continuations
- Stack segments managed on the heap



- Every fiber has a *unique* reference from a continuation object
  - ✦ *Fibers freed when continuations are swept*

Concurrency

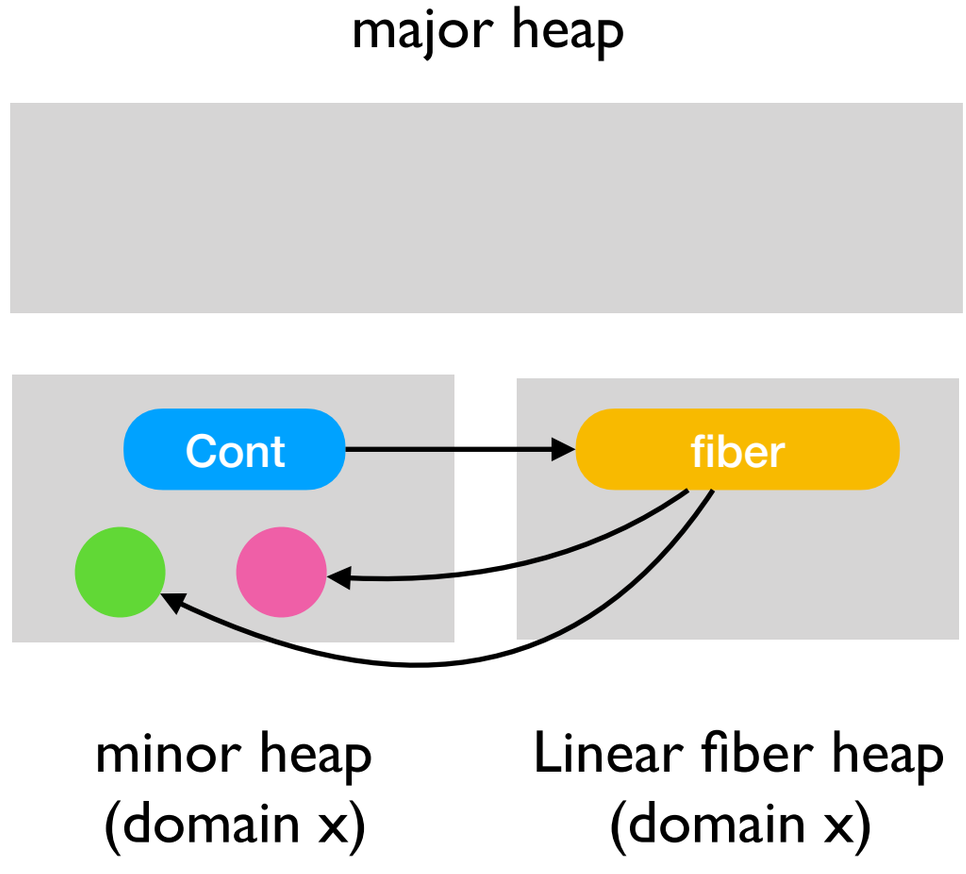
- Fibers: vm-threads, *linear* delimited continuations
- Stack segments managed on the heap



- Every fiber has a *unique* reference from a continuation object
  - ✦ *Fibers freed when continuations are swept*
- *No write barriers on fiber stack operations (push & pop)*
  - ✦ *Handle major and minor GC interactions specially*

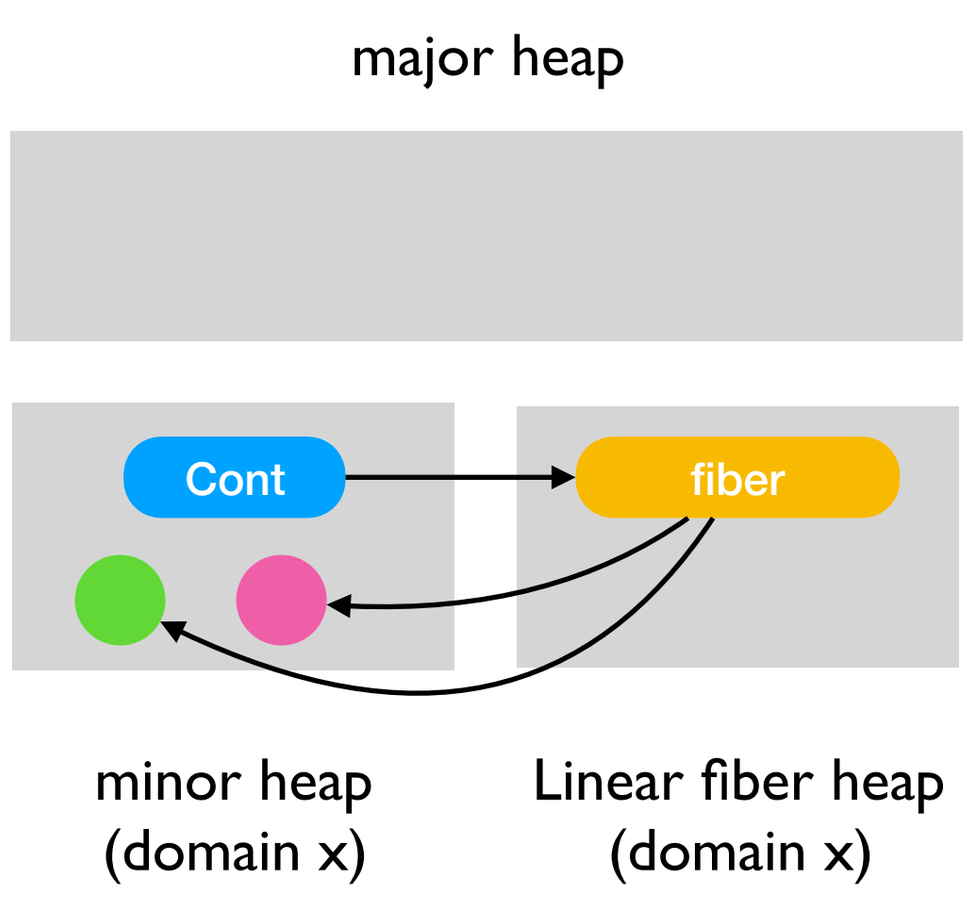
Concurrency — Minor GC

- Fibers may point to minor heap objects
  - ✦ *which fibers to scan among 1000s? (no write barriers on fiber stacks)*



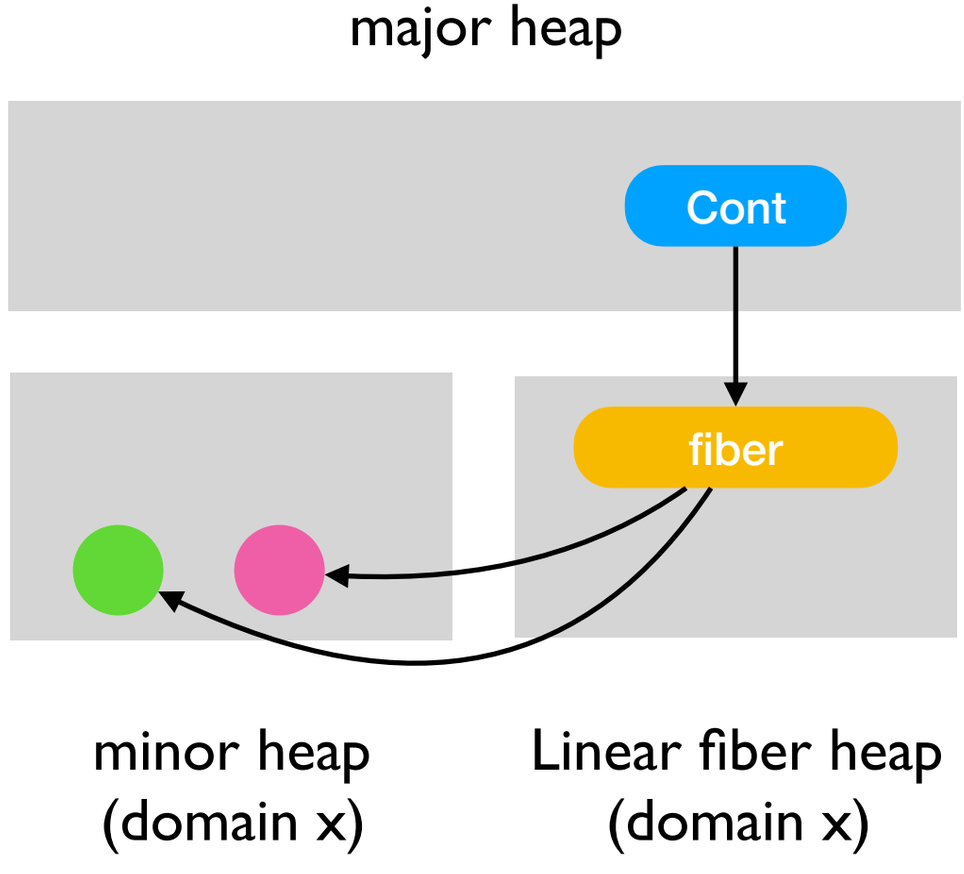
Concurrency — Minor GC

- Fibers may point to minor heap objects
  - ✦ *which fibers to scan among 1000s? (no write barriers on fiber stacks)*
- Fresh continuation object for every fiber suspension
  - ✦ *Continuation in minor heap => fiber suspended in current minor cycle*



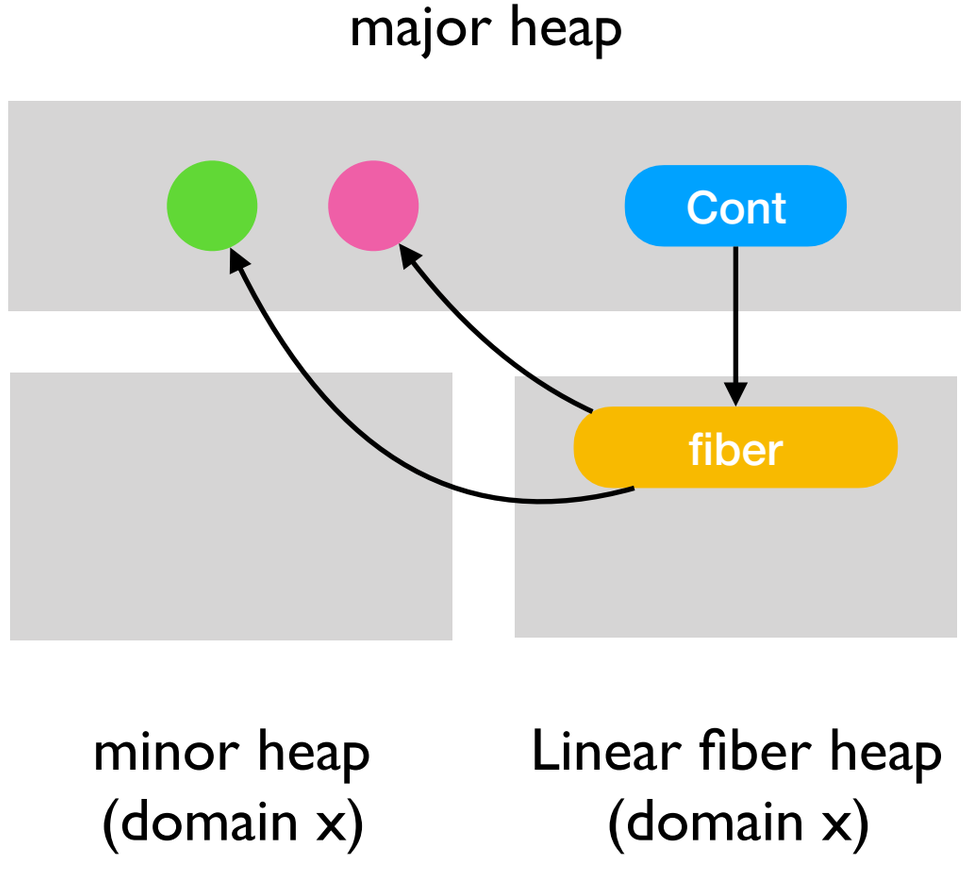
Concurrency — Minor GC

- Fibers may point to minor heap objects
  - ✦ *which fibers to scan among 1000s? (no write barriers on fiber stacks)*
- Fresh continuation object for every fiber suspension
  - ✦ *Continuation in minor heap => fiber suspended in current minor cycle*



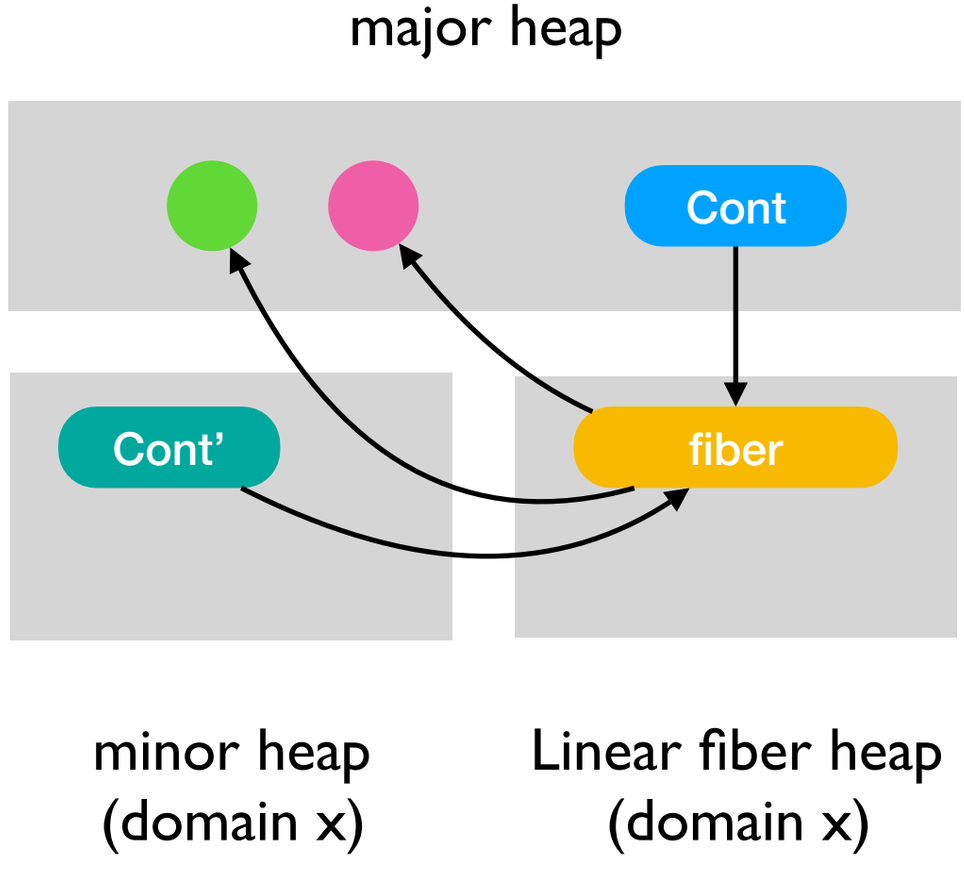
Concurrency — Minor GC

- Fibers may point to minor heap objects
  - ✦ *which fibers to scan among 1000s? (no write barriers on fiber stacks)*
- Fresh continuation object for every fiber suspension
  - ✦ *Continuation in minor heap => fiber suspended in current minor cycle*



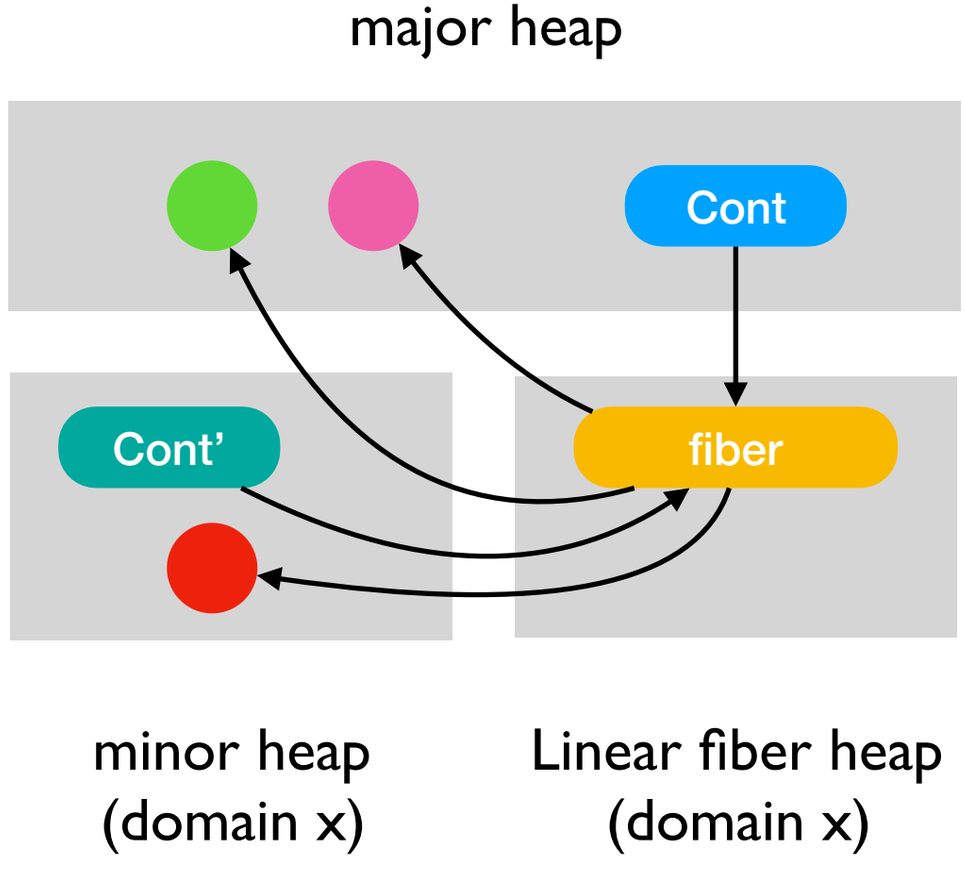
Concurrency — Minor GC

- Fibers may point to minor heap objects
  - ✦ *which fibers to scan among 1000s? (no write barriers on fiber stacks)*
- Fresh continuation object for every fiber suspension
  - ✦ *Continuation in minor heap => fiber suspended in current minor cycle*



Concurrency — Minor GC

- Fibers may point to minor heap objects
  - ✦ *which fibers to scan among 1000s? (no write barriers on fiber stacks)*
- Fresh continuation object for every fiber suspension
  - ✦ *Continuation in minor heap => fiber suspended in current minor cycle*



Concurrency — Major GC

Concurrency — Major GC

- (Multicore) OCaml uses yuasa/deletion barrier
  - ✦ Fiber stack pop is a deletion *(but no write barrier)*

Concurrency — Major GC

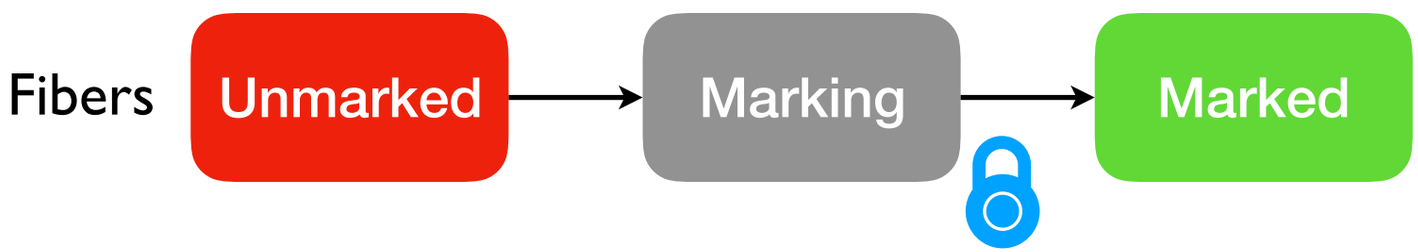
- (Multicore) OCaml uses yuasa/deletion barrier
  - ✦ Fiber stack pop is a deletion *(but no write barrier)*
- Mutator before switching to unmarked fiber, *completes* marking the fiber

Concurrency — Major GC

- (Multicore) OCaml uses yuasa/deletion barrier
  - ✦ Fiber stack pop is a deletion *(but no write barrier)*
- Mutator before switching to unmarked fiber, *completes* marking the fiber
- Marking is racy
  - ✦ *For fibers, race between mutator (context switch) and gc (marking) unsafe*

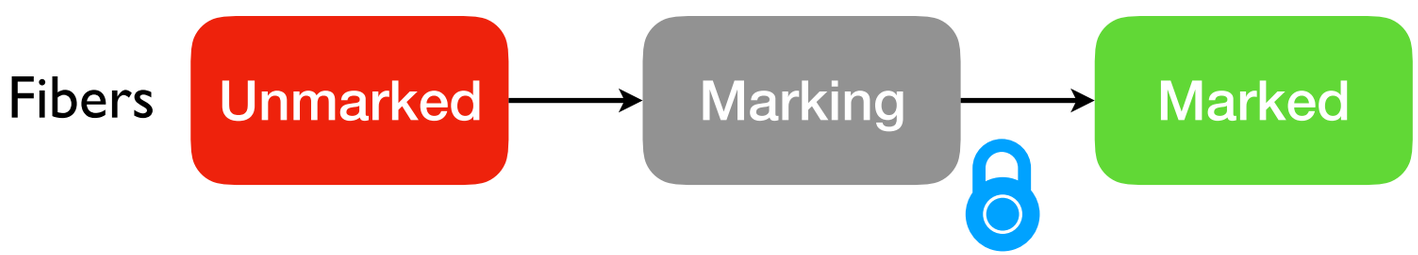
Concurrency — Major GC

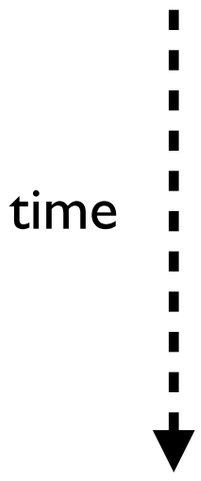
- (Multicore) OCaml uses yuasa/deletion barrier
  - ✦ Fiber stack pop is a deletion *(but no write barrier)*
- Mutator before switching to unmarked fiber, *completes* marking the fiber
- Marking is racy
  - ✦ *For fibers, race between mutator (context switch) and gc (marking) unsafe*



Concurrency — Major GC

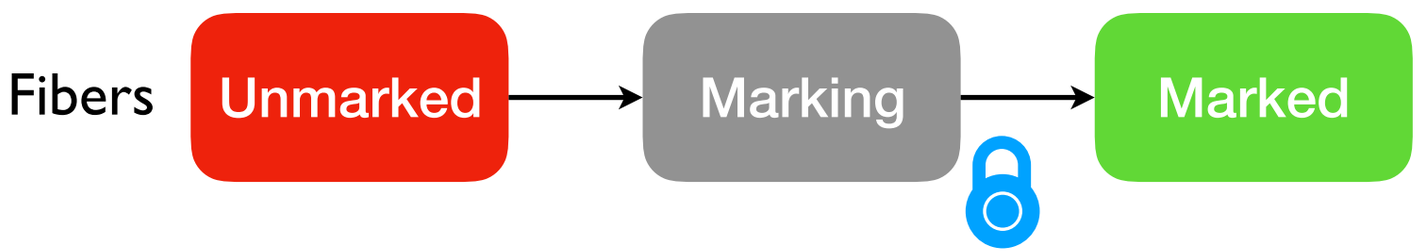
- (Multicore) OCaml uses yuasa/deletion barrier
  - ✦ Fiber stack pop is a deletion *(but no write barrier)*
- Mutator before switching to unmarked fiber, *completes* marking the fiber
- Marking is racy
  - ✦ *For fibers, race between mutator (context switch) and gc (marking) unsafe*

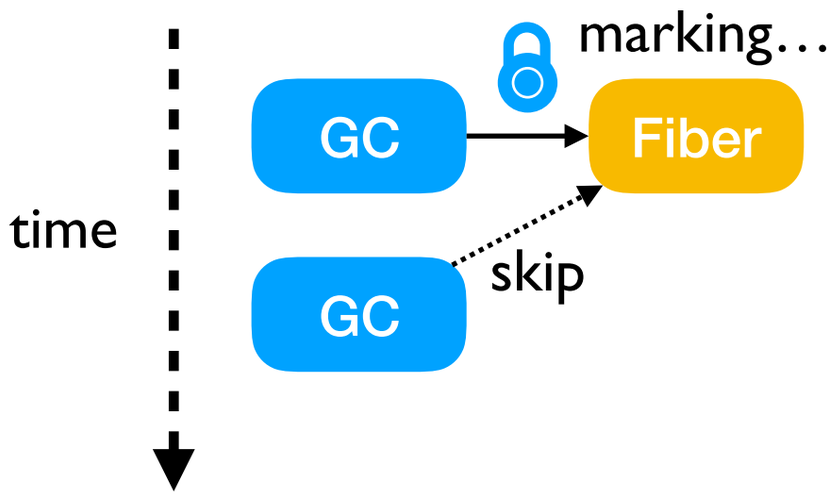




Concurrency — Major GC

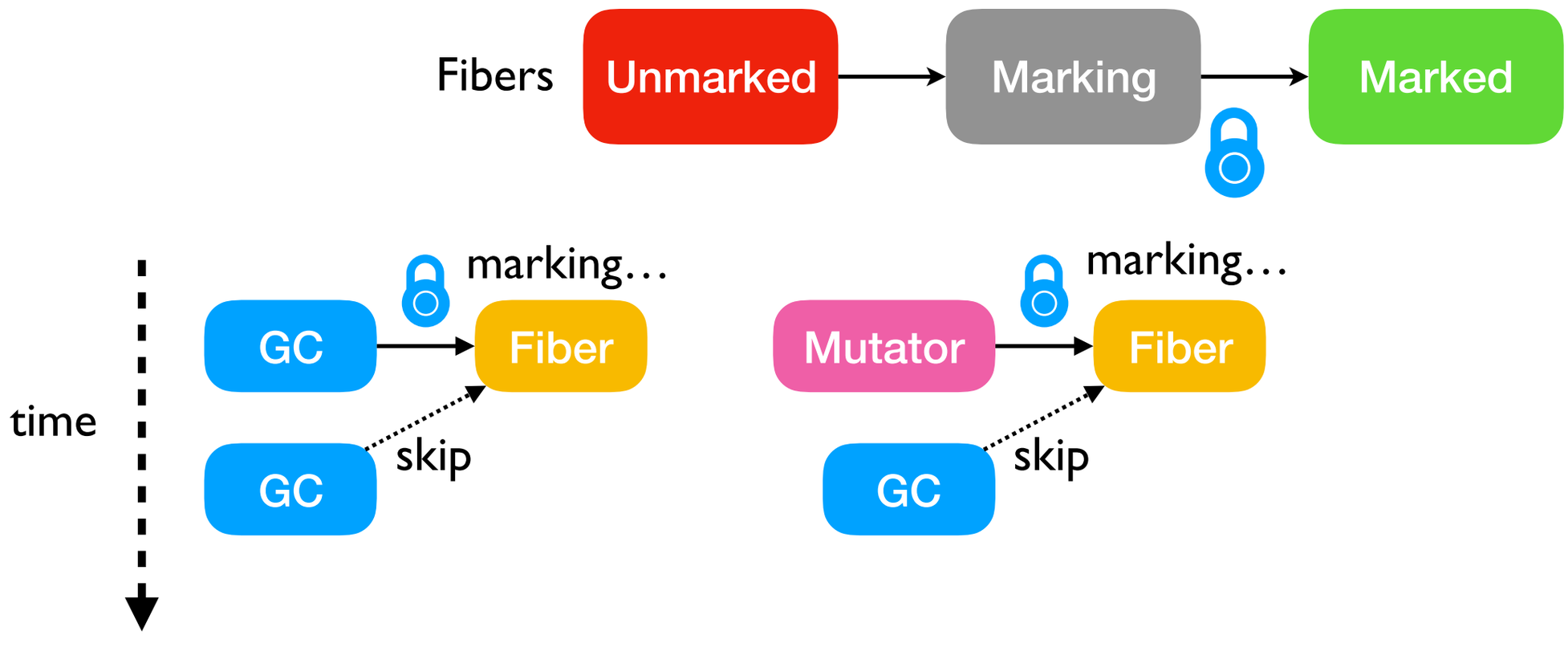
- (Multicore) OCaml uses yuasa/deletion barrier
  - ✦ Fiber stack pop is a deletion *(but no write barrier)*
- Mutator before switching to unmarked fiber, *completes* marking the fiber
- Marking is racy
  - ✦ *For fibers, race between mutator (context switch) and gc (marking) unsafe*





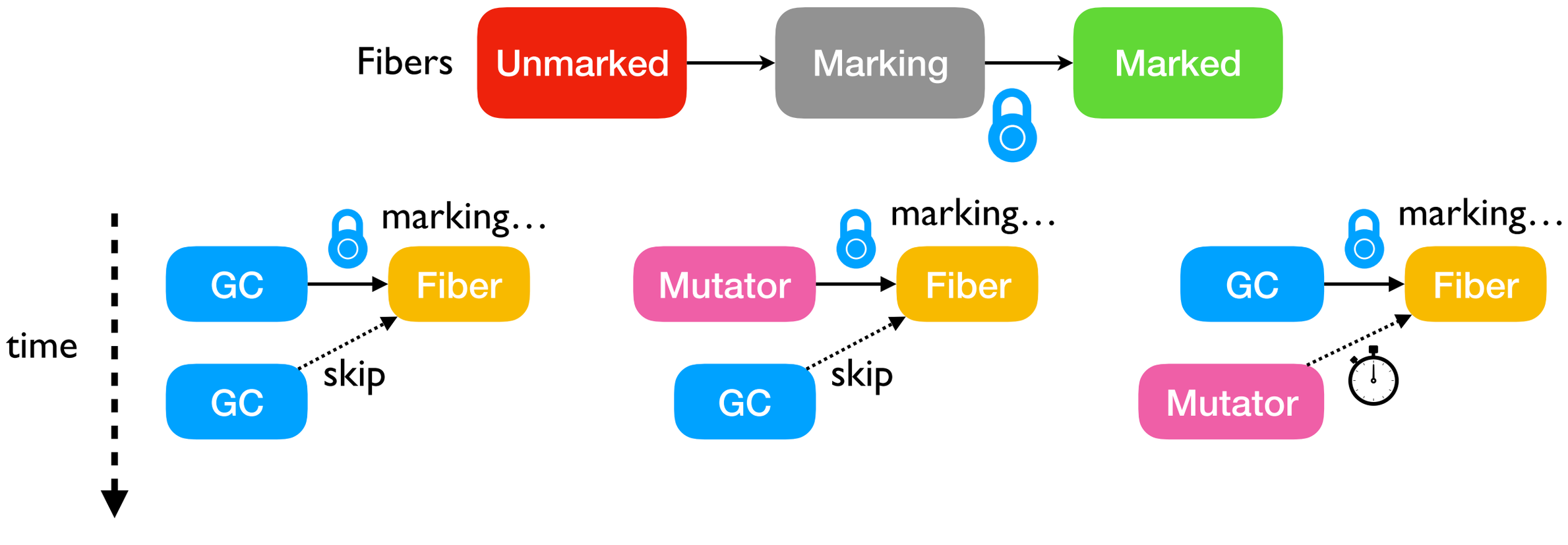
Concurrency — Major GC

- (Multicore) OCaml uses yuasa/deletion barrier
  - ✦ Fiber stack pop is a deletion *(but no write barrier)*
- Mutator before switching to unmarked fiber, *completes* marking the fiber
- Marking is racy
  - ✦ *For fibers, race between mutator (context switch) and gc (marking) unsafe*



Concurrency — Major GC

- (Multicore) OCaml uses yuasa/deletion barrier
  - ✦ Fiber stack pop is a deletion *(but no write barrier)*
- Mutator before switching to unmarked fiber, *completes* marking the fiber
- Marking is racy
  - ✦ *For fibers, race between mutator (context switch) and gc (marking) unsafe*



Performance

Performance

- Serial performance
  - ✦ Multicore benchmarking CI: micro and macro

Performance

- Serial performance
  - ✦ Multicore benchmarking CI: micro and macro
- Parallel Benchmarks
  - ✦ Multicore http server, model-checker, mathematical kernels…
  - ✦ Intel Core i9 (x86_64), 8 domains (parallel threads)

Performance

- Serial performance
  - ✦ Multicore benchmarking CI: micro and macro
- Parallel Benchmarks
  - ✦ Multicore http server, model-checker, mathematical kernels…
  - ✦ Intel Core i9 (x86_64), 8 domains (parallel threads)
- Latency is our primary concern
  - ✦ Minor GC pause times (trunk & multicore) = ~1-2 ms
  - ✦ Avg. 50th percentile pause times = ~4 ms (1-2 ms on trunk)
  - ✦ Avg. 95th percentile pause times = ~7 ms (3-4 ms on trunk)

Performance

- Serial performance
  - ✦ Multicore benchmarking CI: micro and macro
- Parallel Benchmarks
  - ✦ Multicore http server, model-checker, mathematical kernels…
  - ✦ Intel Core i9 (x86_64), 8 domains (parallel threads)
- Latency is our primary concern
  - ✦ Minor GC pause times (trunk & multicore) = ~1-2 ms
  - ✦ Avg. 50th percentile pause times = ~4 ms (1-2 ms on trunk)
  - ✦ Avg. 95th percentile pause times = ~7 ms (3-4 ms on trunk)
- Throughput is easier => add more domains

Open (source) research

Open (source) research

- ***Open-source r&d multiples research impact***

Open (source) research

- ***Open-source r&d multiples research impact***
  - ✦ Not just throwing code over the wall

Open (source) research

- ***Open-source r&d multiples research impact***
  - ✦ Not just throwing code over the wall

Open (source) research

- ***Open-source r&d multiples research impact***
  - ✦ Not just throwing code over the wall

MultiMLton

Open (source) research

- ***Open-source r&d multiples research impact***
  - ✦ Not just throwing code over the wall

MultiMLton

THE UNIVERSITY OF EDINBURGH

Inria
INVENTEURS DU MONDE NUMÉRIQUE

Jane Street

Multicore OCaml

Microsoft Research

Tezos

UNIVERSITY OF CAMBRIDGE

Open (source) research

- ***Open-source r&d multiples research impact***
  - ✦ Not just throwing code over the wall

MultiMLton

- ***Open-source r&d is reproducible => furthers science***

Open (source) research

- ***Open-source r&d multiples research impact***
  - ✦ Not just throwing code over the wall

MultiMLton

- ***Open-source r&d is reproducible => furthers science***
  - ✦ More users => more citations ~=> greater impact.

Research Opportunities in (Multicore) OCaml

Research Opportunities in (Multicore) OCaml

- All of the software and tools are *freely available and actively maintained*

Research Opportunities in (Multicore) OCaml

- All of the software and tools are *freely available and actively maintained*
- Research Opportunities

Research Opportunities in (Multicore) OCaml

- All of the software and tools are *freely available and actively maintained*
- Research Opportunities

1. Analysis of performance regressions in OCaml

   - How have new features have impacted overall performance?
   - Root cause analysis based on commit history & performance.

Research Opportunities in (Multicore) OCaml

- All of the software and tools are *freely available and actively maintained*
- Research Opportunities

1. Analysis of performance regressions in OCaml
   - How have new features have impacted overall performance?
   - Root cause analysis based on commit history & performance.
2. Machine learning to tune GC knobs
   - Conjecture: GC knobs are optimised for average case, where as real deployments have only a few important performance sensitive programs
   - Dimensionality reduction

Research Opportunities in (Multicore) OCaml

- All of the software and tools are *freely available and actively maintained*
- Research Opportunities

1. Analysis of performance regressions in OCaml
   - How have new features have impacted overall performance?
   - Root cause analysis based on commit history & performance.
2. Machine learning to tune GC knobs
   - Conjecture: GC knobs are optimised for average case, where as real deployments have only a few important performance sensitive programs
   - Dimensionality reduction
3. Concurrency testing and verification for Multicore OCaml

Questions?

https://github.com/ocamllabs/ocaml-multicore

http://kcsrk.info