Multicore OCaml GC

KC Sivaramakrishnan, Stephen Dolan
Multicore OCaml
Multicore OCaml

- Adds native support for concurrency and parallelism in OCaml
Multicore OCaml

- Adds native support for concurrency and parallelism in OCaml
- **Fibers** for concurrency, **Domains** for parallelism
  - M fibers over N domains
  - M >>> N
Multicore OCaml

- Adds native support for concurrency and parallelism in OCaml
- Fibers for concurrency, Domains for parallelism
  - M fibers over N domains
  - M >>> N
- This talk
  - Overview of multicore GC with a few deep dives.
Multicore OCaml

- Adds native support for concurrency and parallelism in OCaml
- **Fibers** for concurrency, **Domains** for parallelism
  - M fibers over N domains
  - M >>> N
- This talk
  - Overview of multicore GC with a few deep dives.
Outline

• Difficult to appreciate GC choices in isolation

• Begin with a GC for a *sequential purely functional* language
  ✦ Gradually add mutations, parallelism and concurrency
Purely functional
Purely functional

- Stop-the-world mark and sweep
Purely functional

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

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

- Stop-the-world mark and sweep
- Tri-color marking
  - States: White (Unmarked), Grey (Marking), Black (Marked)
- White $\rightarrow$ Grey (mark stack) $\rightarrow$ Black
- Mark stack is empty $\Rightarrow$ done
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
- Tri-color invariant: No black object points to a white object
Purely functional

registers  stack  heap  mark stack
Purely functional

- Pros
  - Simple
  - Can perform the GC incrementally
    - ...\_mutator\_\_mark\_\_mutator\_\_mark\_\_mutator\_\_sweep\_...
Purely functional

• Pros
  ✦ Simple
  ✦ Can perform the GC incrementally
    ✤ ...|–mutator–|–mark–|–mutator–|–mark–|–mutator–|–sweep–|...

• 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
  - 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
  - Survivors promoted to major heap

- Roots are registers and stack
  - purely functional => no pointers from major to minor
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

\[
(* \text{ Before } r := x *)
\]
\[
\text{let write_barrier } (r, x) =
\text{if is_major } r \text{ && is_minor } x \text{ then}
\text{remembered_set.add } r
\]

- 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

A

B

C
Mutations — Major GC

A   C

B
Mutations — Major GC
Mutations — Major GC

• Mutations are problematic if both conditions hold
  1. Exists Black $\rightarrow$ White
  2. All Grey $\rightarrow$ White* $\rightarrow$ White paths are deleted
Mutations — Major GC

• Mutations are problematic if both conditions hold
  1. Exists Black $\rightarrow$ White
  2. All Grey $\rightarrow$ White* $\rightarrow$ White paths are deleted

• Insertion/Dijkstra/Incremental barrier prevents 1
Mutations — Major GC

• Mutations are problematic if both conditions hold
  1. Exists Black $\rightarrow$ White
  2. All Grey $\rightarrow$ White* $\rightarrow$ White paths are deleted

• Insertion/Dijkstra/Incremental barrier prevents 1
Mutations — Major GC

- Mutations are problematic if both conditions hold
  1. Exists Black $\rightarrow$ White
  2. All Grey $\rightarrow$ White* $\rightarrow$ White paths are deleted

- Insertion/Dijkstra/Incremental barrier prevents
Mutations — Major GC

• Mutations are problematic if both conditions hold
  1. Exists Black $\rightarrow$ White
  2. All Grey $\rightarrow$ White* $\rightarrow$ White paths are deleted

• Insertion/Dijkstra/Incremental barrier prevents 1
Mutations — Major GC

- Mutations are problematic if both conditions hold
  1. Exists Black $\rightarrow$ White
  2. All Grey $\rightarrow$ White* $\rightarrow$ 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 $\rightarrow$ White
  2. All Grey $\rightarrow$ White* $\rightarrow$ 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 $\rightarrow$ White
  2. All Grey $\rightarrow$ White* $\rightarrow$ 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 $\rightarrow$ White

2. All Grey $\rightarrow$ White* $\rightarrow$ 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 $\rightarrow$ White
  2. All Grey $\rightarrow$ White* $\rightarrow$ White paths are deleted

- Insertion/Dijkstra/Incremental barrier prevents 1

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

(* Before $r := x$ *)

```plaintext
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

- Domain.spawn : (unit -> unit) -> unit
Parallelism — Minor GC

- Domain.spawn : (unit -> unit) -> unit
- Collect each domain’s young garbage independently?
Parallelism — Minor GC

- Domain.spawn : (unit -> unit) -> unit
- Collect each domain’s young garbage independently?

Diagram:

- Major heap
- Minor heap(s)
  - Domain 0
  - ... domain n
Parallelism — Minor GC

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

- Collect each domain’s young garbage independently?

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

- Domain.spawn : (unit -> unit) -> unit
- Collect each domain’s young garbage independently?

- 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
- Collect each domain’s young garbage independently?

- 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
- Collect each domain’s young garbage independently?

**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 \( \text{is-major}(r) \) \&\& \( \text{is-minor}(x) \), then promote\((x)\).

- *Too much promotion.* Ex: work-stealing queue
Parallelism — Minor GC

major heap

minor heap(s)

domain 0 \ldots \text{domain } n
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
Efficient read barrier check

- Given x, is x an integer\(^1\) or in shared heap\(^2\) or own minor heap\(^3\)
Efficient read barrier check

- Given x, is x an integer\(^1\) or in shared heap\(^2\) or own minor heap\(^3\)
- Careful VM mapping + bit-twiddling
Efficient read barrier check

- Given x, is x an integer\(^1\) or in shared heap\(^2\) or own minor heap\(^3\)

- Careful VM mapping + bit-twiddling

- Example: 16-bit address space, \(0xPQRS\)
  
  - Minor area \(0x4200 – 0x42ff\)
  - Domain 0: \(0x4220 – 0x422f\)
  - Domain 1: \(0x4250 – 0x425f\)
  - Domain 2: \(0x42a0 – 0x42af\)
Efficient read barrier check

- Given \( x \), is \( x \) an integer\(^1\) or in shared heap\(^2\) or own minor heap\(^3\)

- *Careful VM mapping + bit-twiddling*

- Example: 16-bit address space, \( 0xPQRS \)
  - Minor area \( 0x4200 \) – \( 0x42ff \)
  - Domain 0: \( 0x4220 \) – \( 0x422f \)
  - Domain 1: \( 0x4250 \) – \( 0x425f \)
  - Domain 2: \( 0x42a0 \) – \( 0x42af \)

- Integer \texttt{low\_bit}(S) = 0x1, Minor \( PQ = 0x42 \), \( R \) determines domain
Efficient read barrier check

- Given x, is x an integer\(^1\) or in shared heap\(^2\) or own minor heap\(^3\)
- *Careful VM mapping + bit-twiddling*
- Example: 16-bit address space, 0xPQRS
  - Minor area 0x4200 – 0x42ff
  - Domain 0: 0x4220 – 0x422f
  - Domain 1: 0x4250 – 0x425f
  - Domain 2: 0x42a0 – 0x42af

- Integer low\_bit(S) = 0x1, Minor PQ = 0x42, R determines domain
- Compare with y, where y lies within domain => *allocation pointer!*
  - On amd64, allocation pointer is in r15 register
Efficient read barrier check

# %rax holds x (value of interest)
xor %r15, %rax
sub 0x0010, %rax
test 0xff01, %rax

# Any bit set => ZF not set => not foreign minor
Efficient read barrier check

# %rax holds x (value of interest)
xor %r15, %rax
sub 0x0010, %rax
test 0xff01, %rax
# Any bit set => ZF not set => not foreign minor

Integer

# low_bit(%rax) = 1
xor %r15, %rax
# low_bit(%rax) = 1
sub 0x0010, %rax
# low_bit(%rax) = 1
test 0xff01, %rax
# ZF not set
Efficient read barrier check

```assembly
# %rax holds x (value of interest)
exr %r15, %rax
sub 0x0010, %rax
test 0xff01, %rax
# Any bit set => ZF not set => not foreign minor
```

<table>
<thead>
<tr>
<th>Integer</th>
<th>Shared heap</th>
</tr>
</thead>
<tbody>
<tr>
<td># low_bit(%rax) = 1</td>
<td># PQ(%r15) != PQ(%rax)</td>
</tr>
<tr>
<td>xor %r15, %rax</td>
<td>xor %r15, %rax</td>
</tr>
<tr>
<td># low_bit(%rax) = 1</td>
<td># PQ(%rax) is non-zero</td>
</tr>
<tr>
<td>sub 0x0010, %rax</td>
<td>sub 0x0010, %rax</td>
</tr>
<tr>
<td># low_bit(%rax) = 1</td>
<td># PQ(%rax) is non-zero</td>
</tr>
<tr>
<td>test 0xff01, %rax</td>
<td>test 0xff01, %rax</td>
</tr>
<tr>
<td># ZF not set</td>
<td># ZF not set</td>
</tr>
</tbody>
</table>
Efficient read barrier check

# %rax holds x (value of interest)
xor %r15, %rax
sub 0x0010, %rax
test 0xff01, %rax
# Any bit set => ZF not set => not foreign minor
Efficient read barrier check

# %rax holds x (value of interest)
xor %r15, %rax
sub 0x0010, %rax
test 0xff01, %rax
# Any bit set => ZF not set => not foreign minor

Own minor heap

# PQR(%r15) = PQR(%rax)
xor %r15, %rax
# PQR(%rax) is zero
sub 0x0010, %rax
# PQ(%rax) is non-zero
test 0xff01, %rax
# ZF not set
Efficient read barrier check

# %rax holds x (value of interest)
xor %r15, %rax
sub 0x0010, %rax
test 0xff01, %rax
# Any bit set => ZF not set => not foreign minor

Own minor heap

# PQR(%r15) = PQR(%rax)
xor %r15, %rax
# PQR(%rax) is zero
sub 0x0010, %rax
# PQ(%rax) is non-zero
test 0xff01, %rax
# ZF not set

Foreign minor heap

# PQ(%r15) = PQ(%rax)
# S(%r15) = S(%rax) = 0
# R(%r15) != R(%rax)
xor %r15, %rax
# R(%rax) is non-zero, rest 0
sub 0x0010, %rax
# rest 0
test 0xff01, %rax
# ZF set
Promotion
How do you promote objects to the major heap on read fault?
How do you promote objects to the major heap on read fault?

Several alternatives

1. Copy the object to major heap.
   ✤ Mutable objects, Abstract_tag, …

2. Move the object closure + minor GC.
   ✤ False promotions, latency, …

3. Move the object closure + scan the minor GC
   ✤ Need to examine all objects on minor GC
Promotion

- **How do you promote objects to the major heap on read fault?**

- Several alternatives

  1. Copy the object to major heap.
     - Mutable objects, Abstract_tag, …
  2. Move the object closure + minor GC.
     - False promotions, latency, …
  3. Move the object closure + scan the minor GC
     - Need to examine all objects on minor GC

- **Hypothesis:** most objects promoted on read faults are young.

  - 95% promoted objects among the youngest 5%
How do you promote objects to the major heap on read fault?

Several alternatives

1. Copy the object to major heap.
   ✤ Mutable objects, Abstract_tag, …

2. Move the object closure + minor GC.
   ✤ False promotions, latency, …

3. Move the object closure + scan the minor GC
   ✤ Need to examine all objects on minor GC

Hypothesis: most objects promoted on read faults are young.
   ✤ 95% promoted objects among the youngest 5%

Combine 2 & 3
Promotion
Promotion

- If promoted object among youngest x%,
  - move + fix pointers to promoted object
    - Scan roots = registers + current stack + remembered set
    - Younger minor objects
    - Older minor objects referring to younger objects (mutations!)
Promotion

- If promoted object among youngest x%,
  - move + fix pointers to promoted object
    - Scan roots = registers + current stack + remembered set
    - Younger minor objects
    - Older minor objects referring to younger objects (mutations!)

\[
\begin{align*}
(* \ r := x *) \\
let \ write\_barrier \ (r, x) = \\
  & \text{if is_major } r \ \&\& \text{ is_minor } x \ \text{then} \ \\
  & \ \ \ \ \text{remembered_set.add } r \\
  & \text{else if is_major } r \ \&\& \text{ is_major } x \ \text{then} \\
  & \ \ \ \ \text{mark(!}r\text{)} \\
  & \text{else if is_minor } r \ \&\& \text{ is_minor } x \ \&\& \text{addr } r > \text{addr } x \ \text{then} \\
  & \ \ \ \ \text{promotion_set.add } r
\end{align*}
\]
Promotion

• If promoted object among youngest x%,
  ✦ move + fix pointers to promoted object
    ❖ Scan roots = registers + current stack + remembered set
    ❖ Younger minor objects
    ❖ Older minor objects referring to younger objects (mutations!)

(* 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)
  else if is_minor r && is_minor x && addr r > addr x then
    promotion_set.add r

• Otherwise, move + minor GC
Parallelism — Major GC
Parallelism — Major GC

- OCaml’s GC is *incremental*, needs to be *concurrent* w/ parallelism
Parallelism — Major GC

- OCaml’s GC is *incremental*, needs to be *concurrent* w/ parallelism
- Design based on VCGC from Inferno project (ISMM’98)
Parallelism — Major GC

- OCaml’s GC is *incremental*, needs to be *concurrent* w/ parallelism
- Design based on VCGC from Inferno project (ISMM’98)
  - Allows mutator, marker, sweeper threads to concurrently
Parallelism — Major GC

- OCaml’s GC is *incremental*, needs to be *concurrent* w/ parallelism
- Design based on VCGC from Inferno project (ISMM’98)
  - Allows mutator, marker, sweeper threads to concurrently
- Multicore OCaml is MCGC
Parallelism — Major GC

- OCaml’s GC is *incremental*, needs to be *concurrent* w/ parallelism
- Design based on VCGC from Inferno project (ISMM’98)
  - Allows mutator, marker, sweeper threads to concurrently
- Multicore OCaml is MCGC
  - States: **Unmarked** | **Marked** | **Garbage** | **Free**
Parallelism — Major GC

- OCaml’s GC is *incremental*, needs to be *concurrent* w/ parallelism

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

- Multicore OCaml is MCGC
  - States: **Unmarked**, **Marked**, **Garbage**, **Free**
  - Domains alternate between mutator and gc thread
Parallelism — Major GC

- OCaml’s GC is *incremental*, needs to be *concurrent* w/ parallelism

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

- Multicore OCaml is MCGC
  ✦ States
    - Unmarked
    - Marked
    - Garbage
    - Free
  ✦ Domains alternate between mutator and gc thread
  ✦ GC thread
    - Unmarked ➔ Marked ➔ Garbage ➔ Free
Parallelism — Major GC

- OCaml’s GC is *incremental*, needs to be *concurrent* w/ parallelism

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

- Multicore OCaml is MCGC
  - States: Unmarked, Marked, Garbage, Free
  - Domains alternate between mutator and gc thread
  - GC thread: Unmarked → Marked → Garbage → Free
  - Marking is *racy* but *idempotent*
Parallelism — Major GC

- OCaml's GC is *incremental*, needs to be *concurrent* w/ parallelism
- Design based on VCGC from Inferno project (ISMM’98)
  - Allows mutator, marker, sweeper threads to concurrently
- Multicore OCaml is MCGC
  - States: Unmarked, Marked, Garbage, Free
  - Domains alternate between mutator and gc thread
  - GC thread: Unmarked → Marked, Garbage → Free
  - Marking is *racy* but *idempotent*
- Stop-the-world
Parallelism — Major GC

- OCaml’s GC is *incremental*, needs to be *concurrent* w/ parallelism
- Design based on VCGC from Inferno project (ISMM’98)
  - Allows mutator, marker, sweeper threads to concurrently
- Multicore OCaml is MCGC
  - States: Unmarked, Marked, Garbage, Free
  - Domains alternate between mutator and gc thread
  - GC thread: Unmarked $\rightarrow$ Marked $\rightarrow$ Garbage $\rightarrow$ Free
  - Marking is *racy* but *idempotent*

- Stop-the-world
Concurrency — Minor GC

- **Fibers**: vm-threads, 1-shot delimited continuations
  - stack segments on heap
Concurrency — Minor GC

- **Fibers**: vm-threads, 1-shot delimited continuations
  - stack segments on heap

- stack operations are not protected by write barrier!
Concurrency — Minor GC

- **Fibers**: vm-threads, 1-shot delimited continuations
  - stack segments on heap

- **Stack operations are not protected by write barrier!**
Concurreny — Minor GC

- **Fibers**: vm-threads, 1-shot delimited continuations
  - stack segments on heap

- **stack operations are not protected by write barrier!**

- **Remembered fiber set**
  - Set of fibers in major heap that were ran in the current cycle of domain x
  - Cleared after minor GC
Concurrency — Promotions

- Fibers transitively reachable are not promoted automatically
  - Avoids false promotions
Concurrency — Promotions

- **major heap**
  - Node r
  - Node x
- **minor heap (domain 0)**
  - Node f
  - Node z
  - Remembered set

説明が足りない場合は、より詳しい説明を求めてください。
Concurrency — Promotions

- Fibers transitively reachable are not promoted automatically
  - Avoids false promotions
Concurrency — Promotions

- Fibers transitively reachable are not promoted automatically
  - Avoids false promotions
  - Promote on continuing foreign fiber
Concurrence — Promotions

- Fibers transitively reachable are not promoted automatically
  - Avoids false promotions
  - Promote on continuing foreign fiber
Concurrency — Promotions
Concurrency — Promotions

• Recall, promotion fast path = move + scan and forward
  ✦ Do not scan remembered fiber set
    ✷ Context switches $\ll$ promotions
Concurrency — Promotions

- Recall, promotion fast path = move + scan and forward
  - Do not scan remembered fiber set
    - Context switches << promotions
- Scan lazily before context switch
  - Only once per fiber per promotion
  - In practice, scans a fiber per a batch of promotions
Concurrency — Major GC
Concurrency — Major GC

- (Multicore) OCaml uses deletion barrier
Concurrency — Major GC

- (Multicore) OCaml uses deletion barrier
- Fiber stack pop is a deletion
  - Before switching to unmarked fiber, complete marking fiber
Concurrency — Major GC

- (Multicore) OCaml uses deletion barrier
- Fiber stack pop is a deletion
  - Before switching to unmarked fiber, *complete* marking fiber
- Marking is racy but idempotent
  - *Race between mutator (context switch) and gc (marking) unsafe*
Concurrency — Major GC

- (Multicore) OCaml uses deletion barrier
- Fiber stack pop is a deletion
  - Before switching to unmarked fiber, complete marking fiber
- Marking is racy but idempotent
  - Race between mutator (context switch) and gc (marking) unsafe
Summary

- Multicore OCaml GC
  - Optimize for latency
  - Independent minor GCs + mostly-concurrent mark-and-sweep

<table>
<thead>
<tr>
<th></th>
<th>Mutations</th>
<th>Concurrency</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Minor GC</strong></td>
<td>rem set</td>
<td>rem fiber set</td>
<td>local heaps</td>
</tr>
<tr>
<td><strong>Promotions</strong></td>
<td>o2y rem set</td>
<td>lazy scanning</td>
<td>read faults</td>
</tr>
<tr>
<td><strong>Major GC</strong></td>
<td>deletion barrier</td>
<td>mark &amp; switch</td>
<td>MCGC</td>
</tr>
</tbody>
</table>
Questions?
Backup Slides
Purely functional GC
Purely functional GC

- Stop-the-world mark and sweep
Purely functional GC

- Stop-the-world mark and sweep
Purely functional GC

- Stop-the-world mark and sweep
Purely functional GC

- Stop-the-world mark and sweep
- 2-pass mark compact
  - Fast allocations by bumping the frontier
Purely functional GC

- Stop-the-world mark and sweep
- 2-pass mark compact
  - Fast allocations by bumping the frontier
- All heap pointers go right
Purely functional GC

- Mark roots
Purely functional GC

- Mark roots
- Scan from *frontier* to *start*. For each marked object,
  - Mark reachable object & *reverse pointers*
Purely functional GC

- Mark roots
- Scan from frontier to start. For each marked object,
  - Mark reachable object & reverse pointers
- Scan from start to frontier. For each marked object,
  - Copy to next available free space & reverse pointers pointing left
Purely functional GC
Purely functional GC

- **Pros**
  - Simple & fast allocation
  - Efficient use of space
Purely functional GC

Pros

✦ Simple & fast allocation
✦ Efficient use of space

Cons

✦ Need to touch all the objects on the heap
✦ Compaction as default is leads to long pause times