Modular Garbage Collectors in Ruby

Peter Zhu

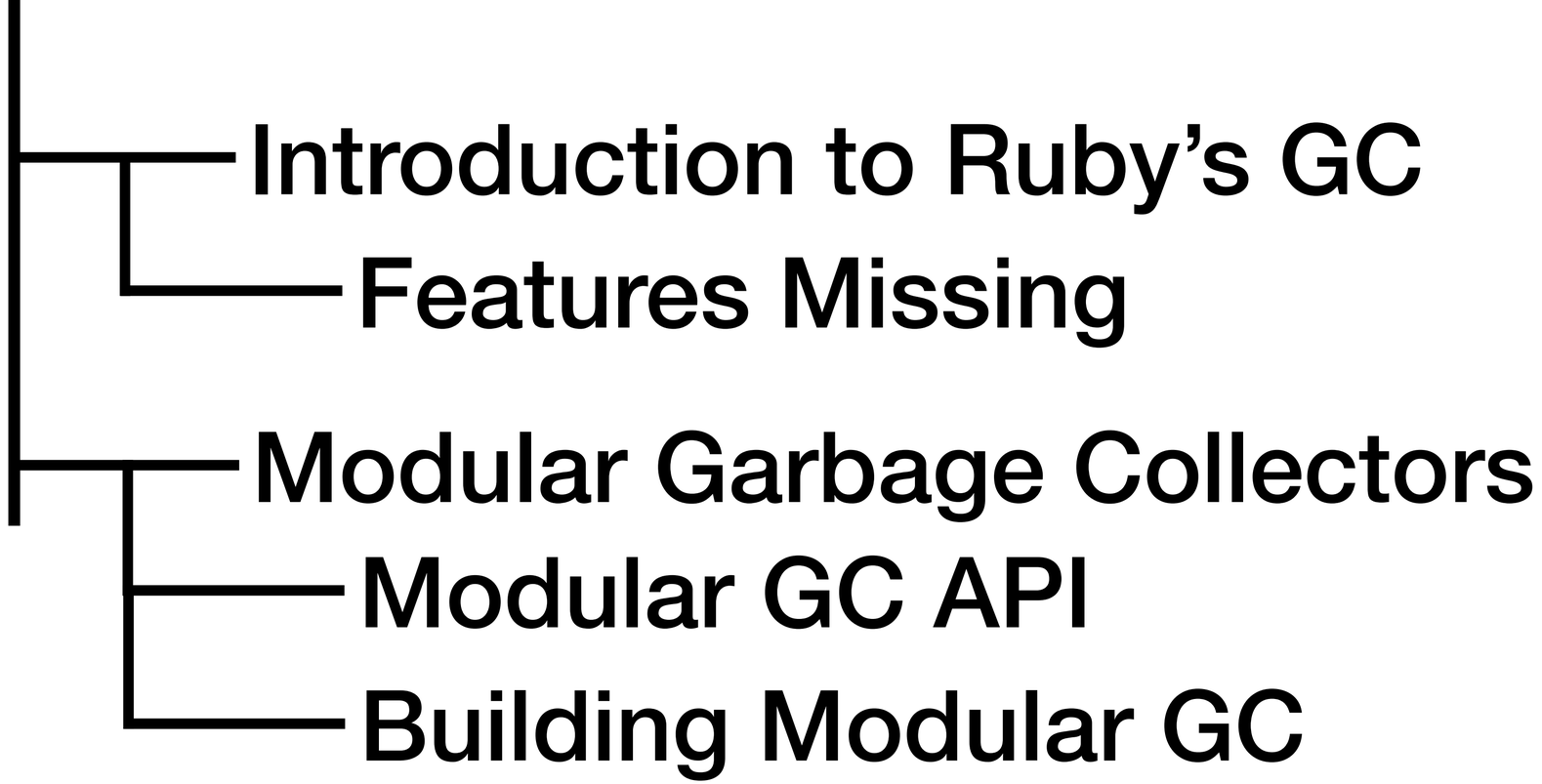
Ruby Core Committer
Senior Developer, Shopify

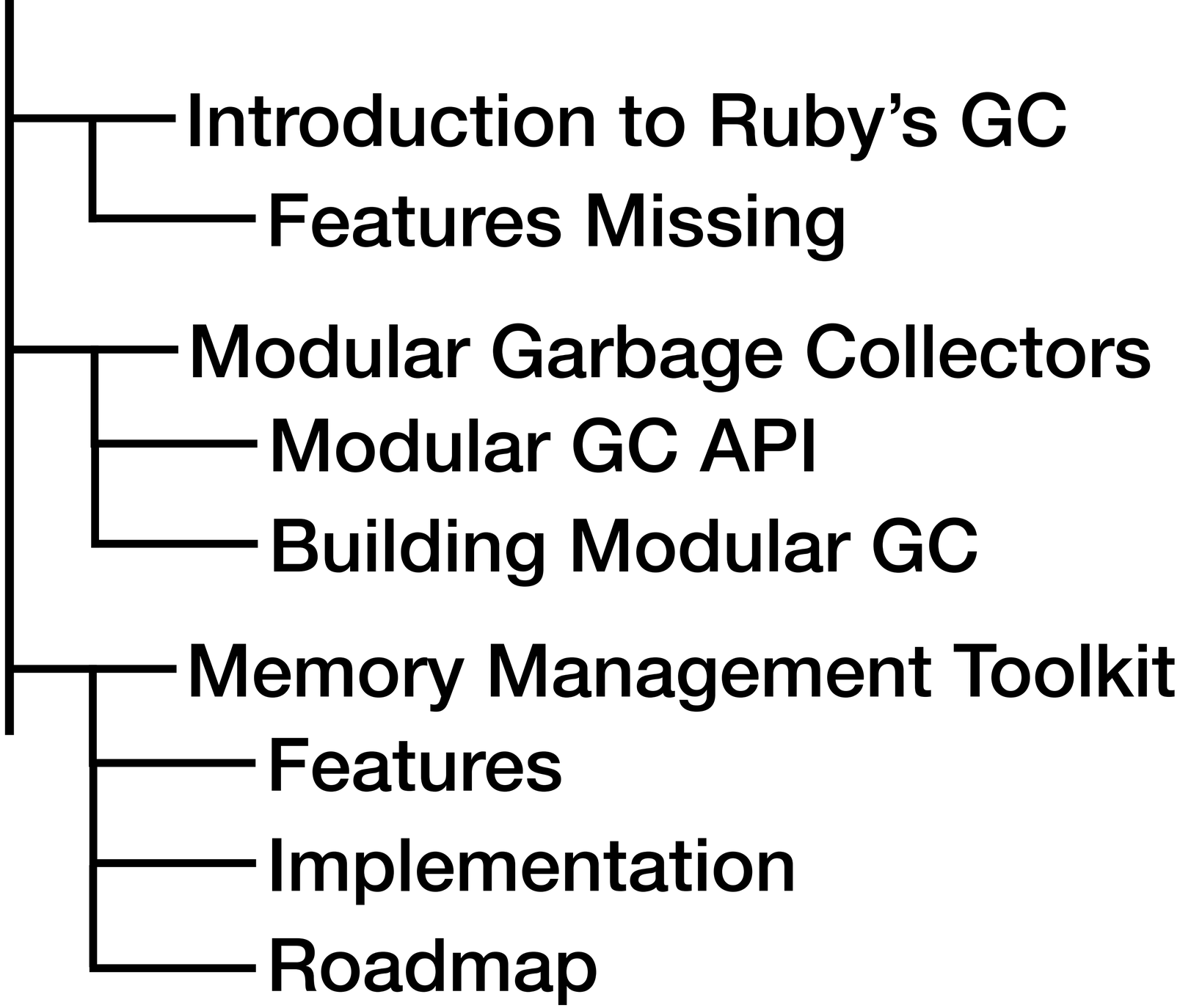
blog.peterzhu.ca/assets/rubykaigi_2025_slides.pdf

Peter Zhu

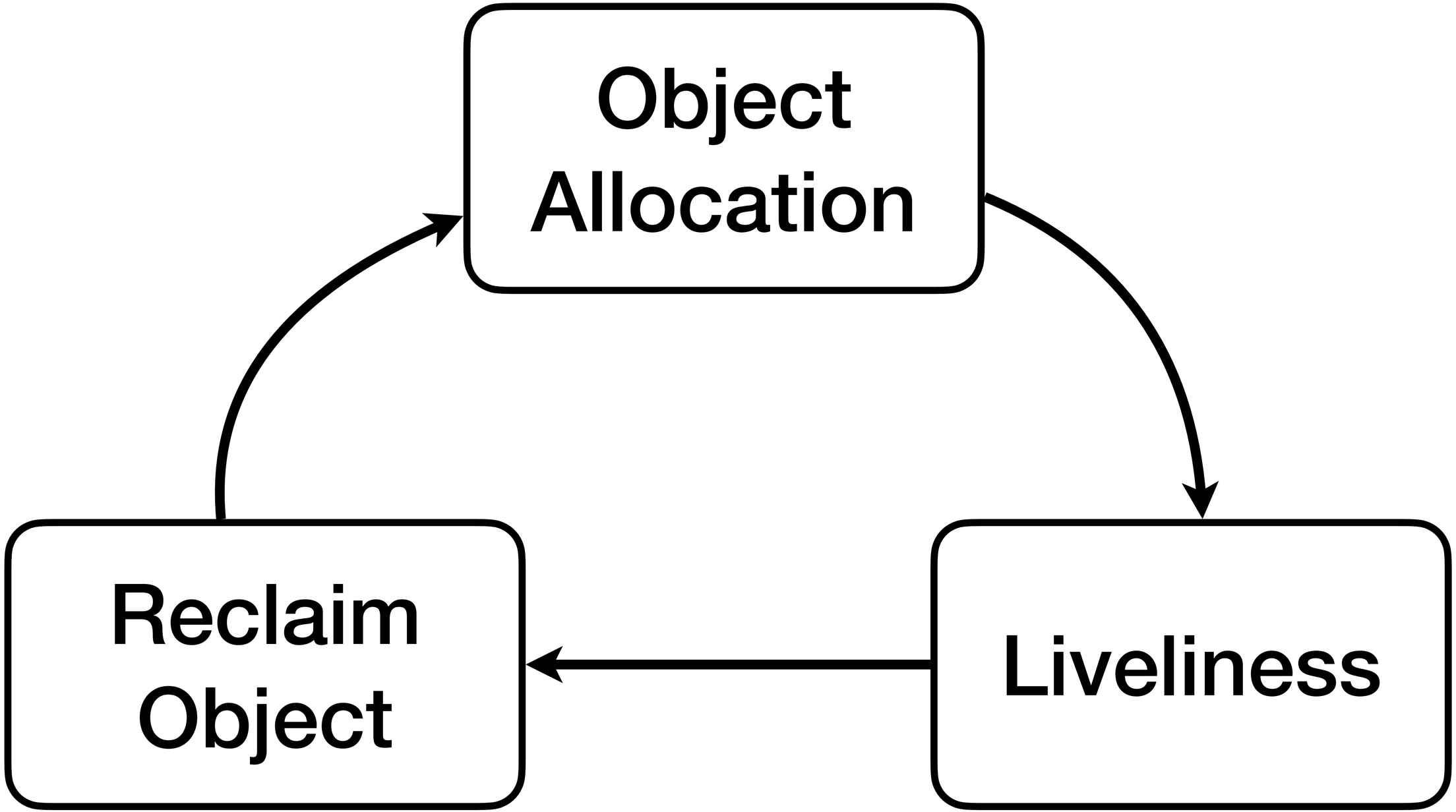
- Based in Toronto, Canada
- Ruby Core Committer
- Senior Developer on the Ruby Infrastructure team at Shopify
- Co-author of Variable Width Allocation and `RUBY_FREE_AT_EXIT` in Ruby
- Author of `ruby_memcheck` and `autotuner`
- Photography geek, follow me @peterzhu.photos on Instagram!


Introduction to Ruby’s GC
Features Missing

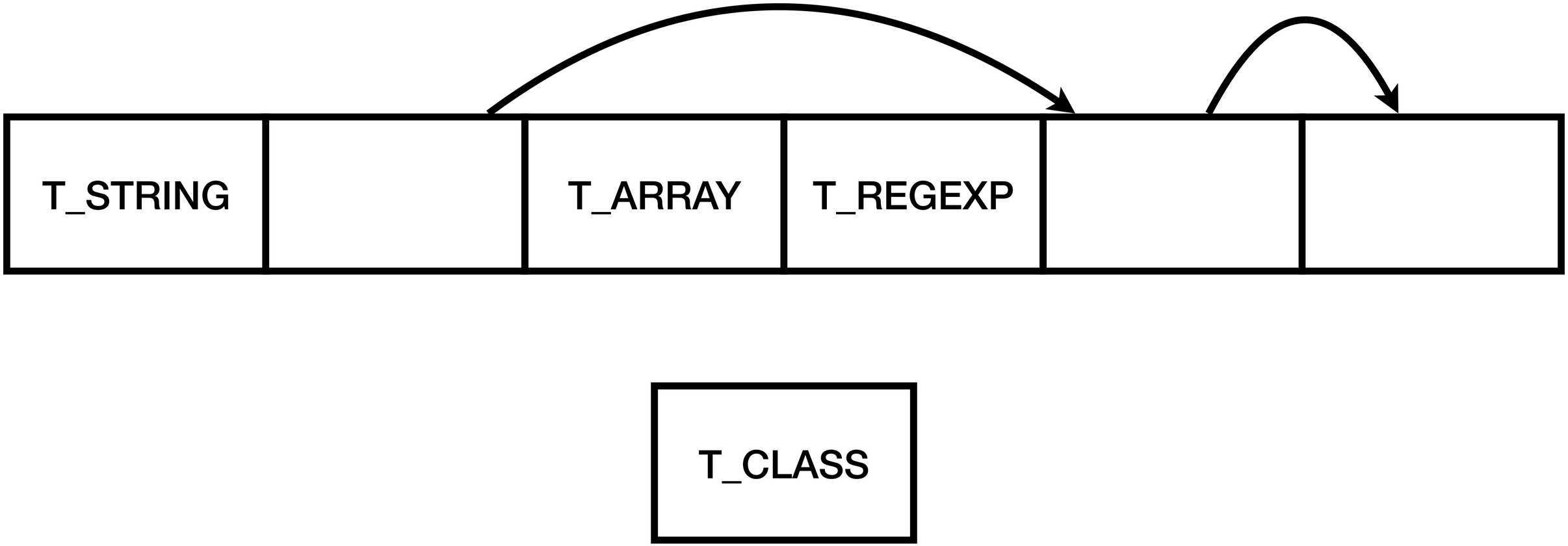

Introduction to Ruby's GC
Features Missing
Modular Garbage Collectors
Modular GC API
Building Modular GC


Introduction to Ruby's GC
Features Missing
Modular Garbage Collectors
Modular GC API
Building Modular GC
Memory Management Toolkit
Features
Implementation
Roadmap

What is a Garbage Collector?

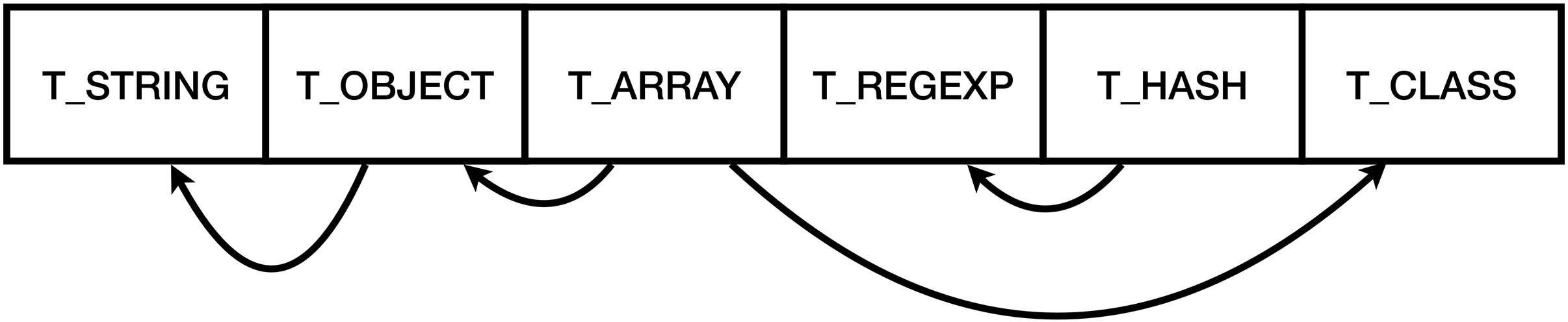

Object
Allocation
Liveliness
Reclaim
Object

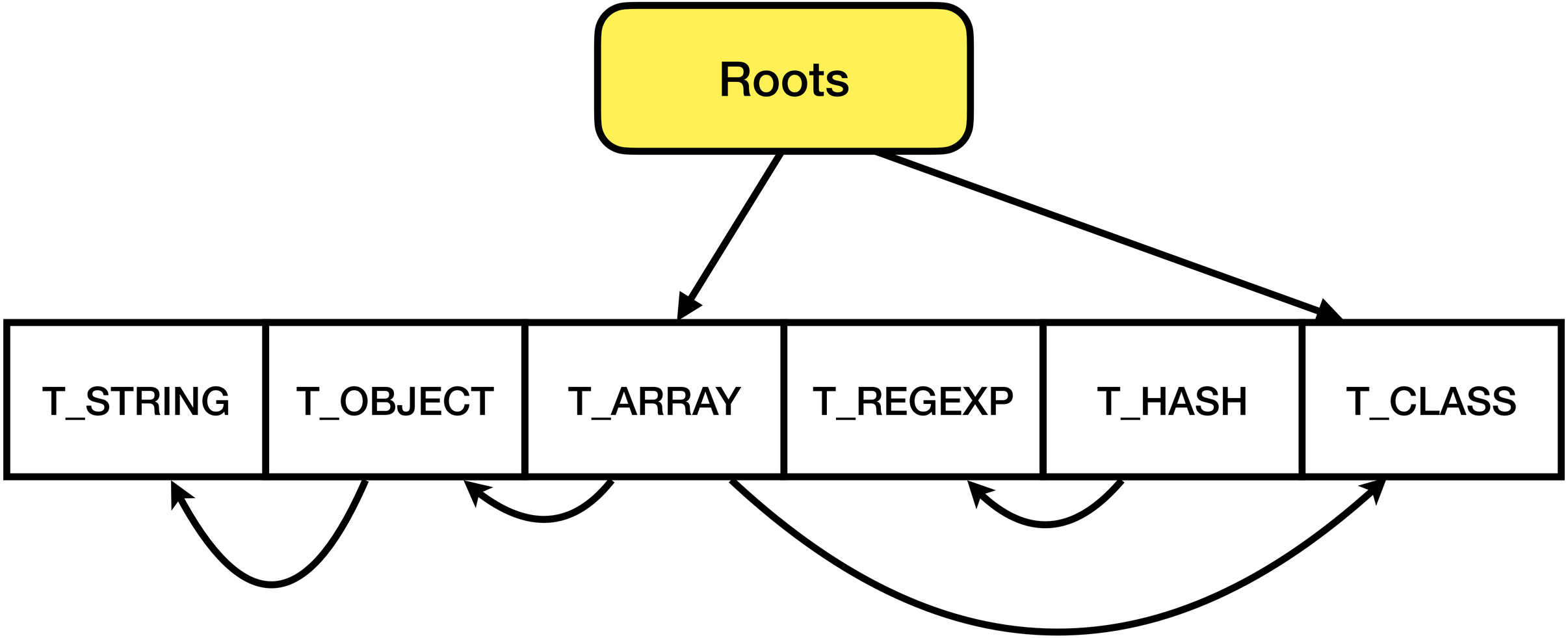
Free List Allocator

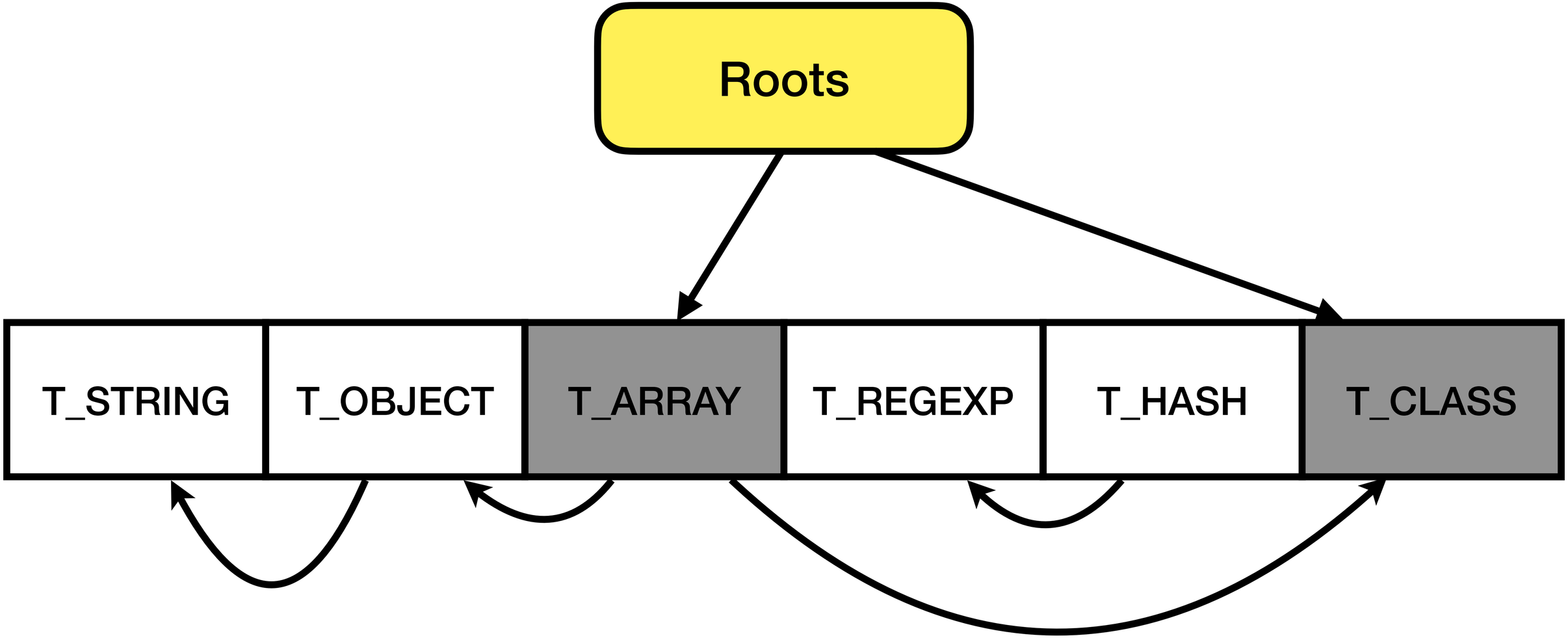

T_STRING
T_ARRAY
T_REGEXP
T_CLASS

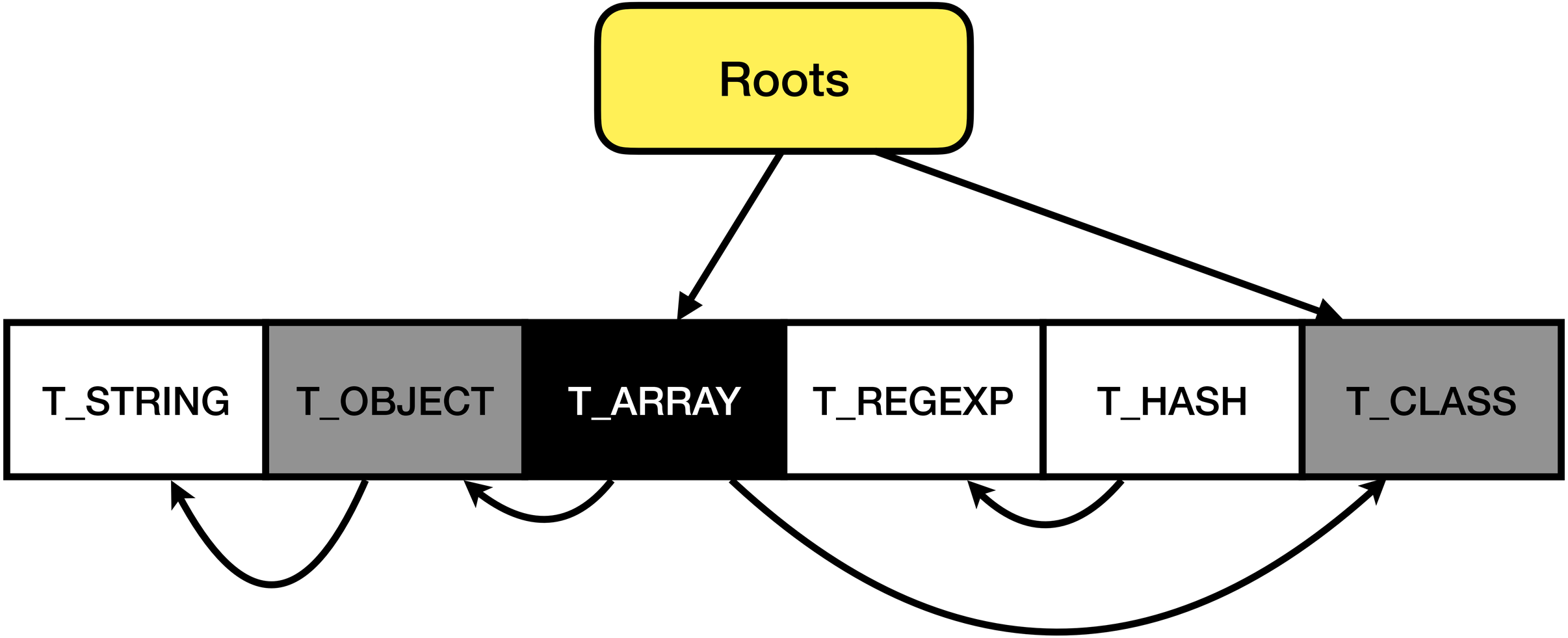
Ruby’s Mark-Sweep-Compact Garbage Collector

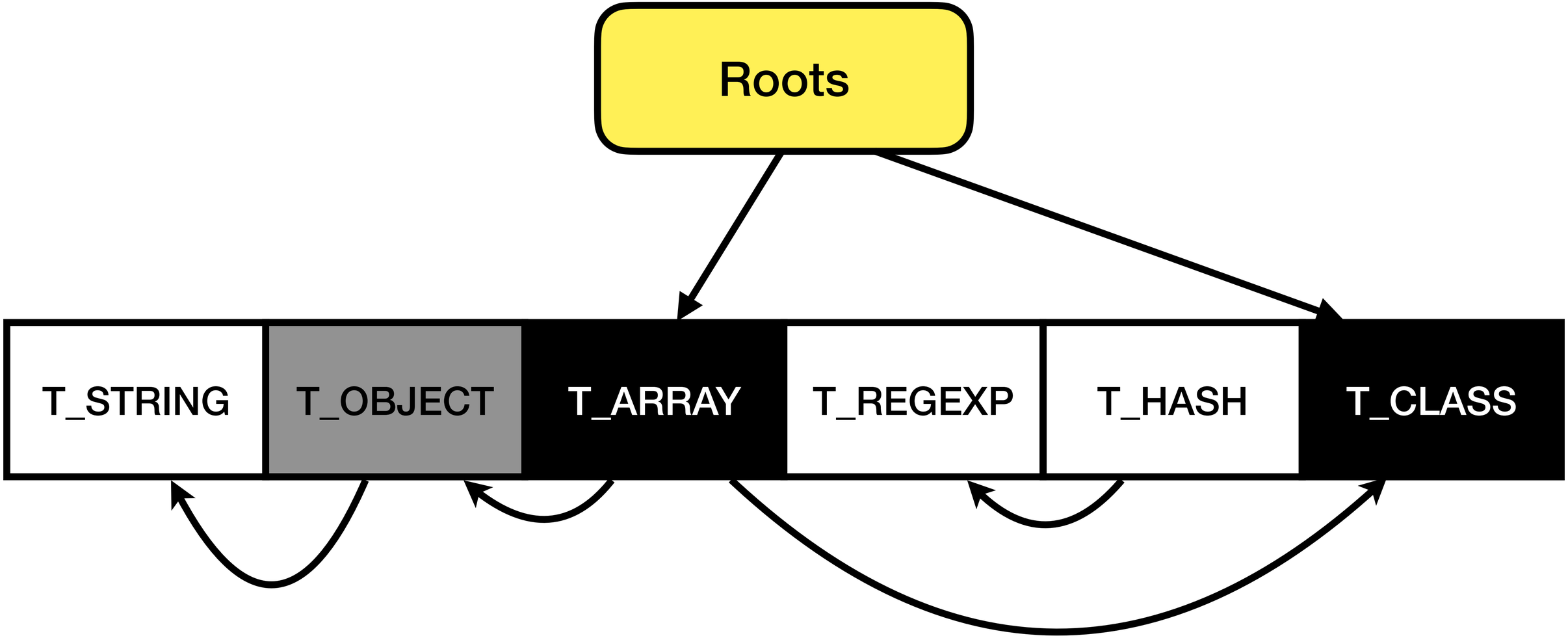
Marking

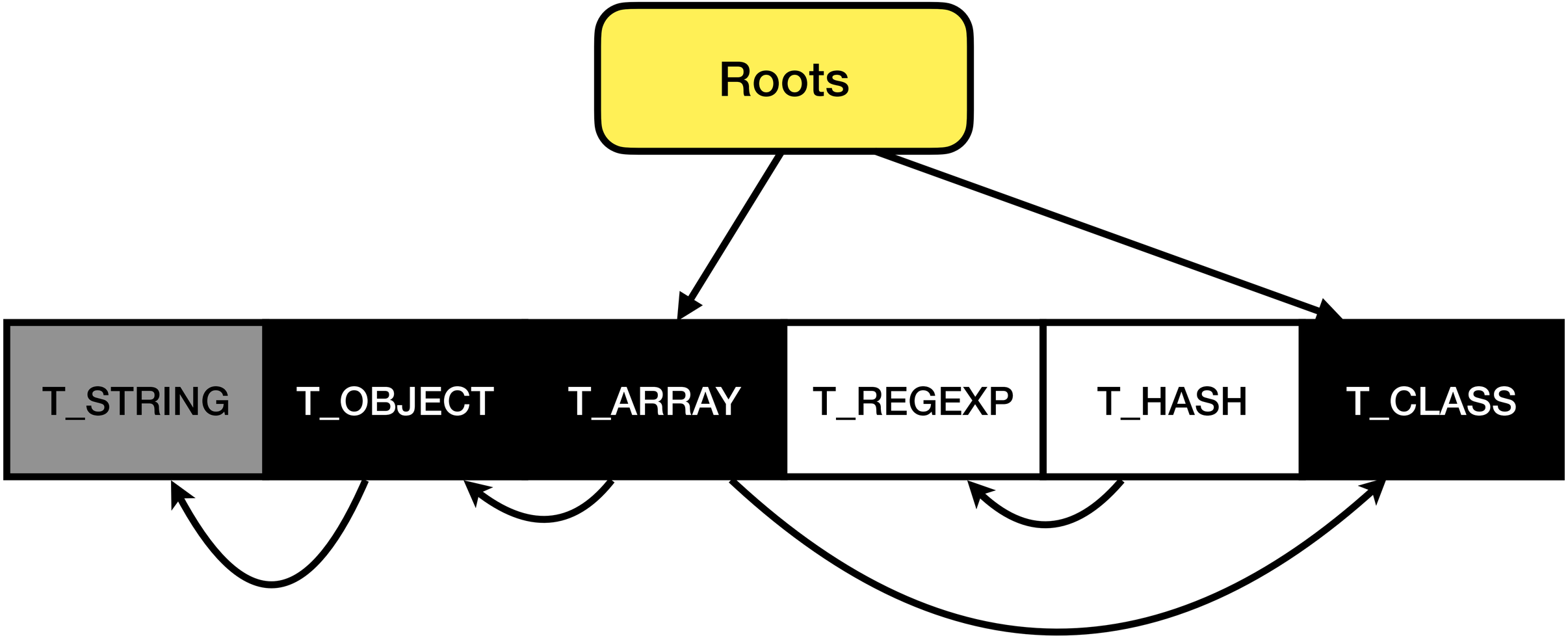

T_STRING
T_OBJECT
T_ARRAY
T_REGEXP
T_HASH
T_CLASS

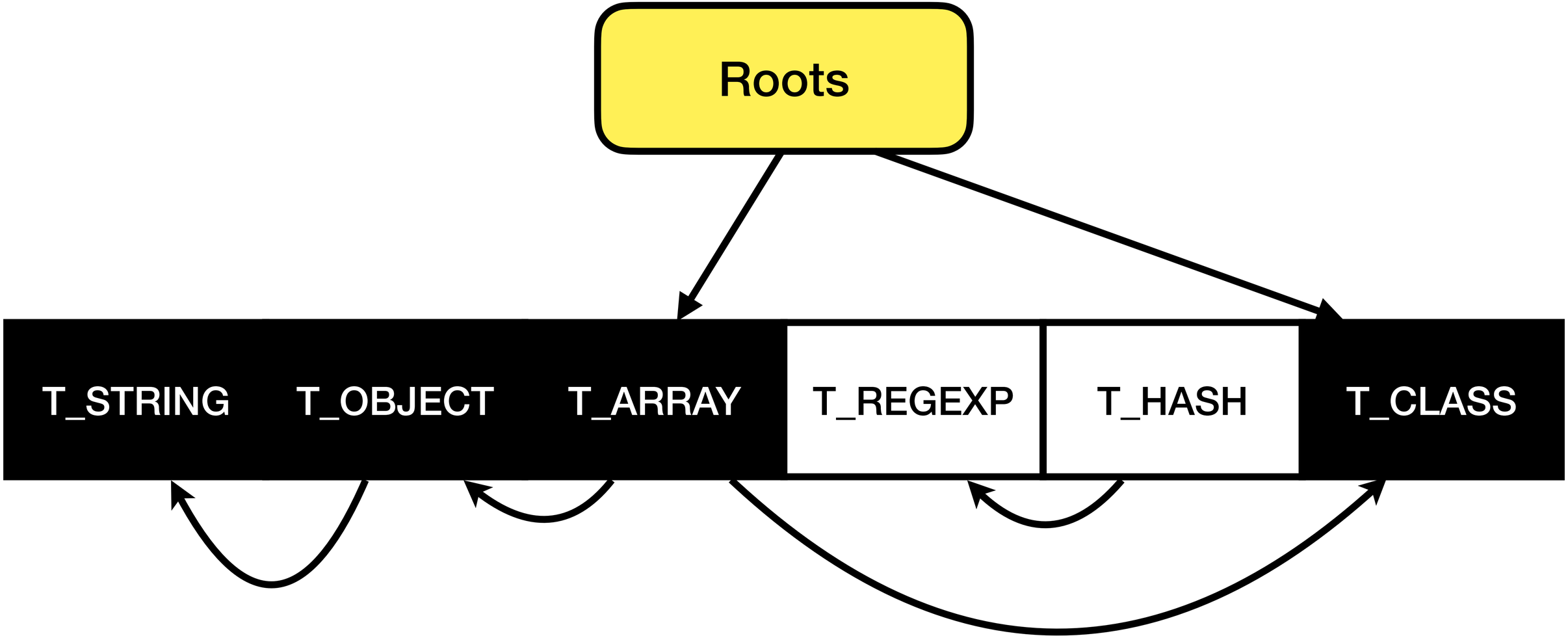

Roots
T_STRING
T_OBJECT
T_ARRAY
T_REGEXP
T_HASH
T_CLASS


Roots
T_STRING
T_OBJECT
T_ARRAY
T_REGEXP
T_HASH
T_CLASS


Roots
T_STRING
T_OBJECT
T_ARRAY
T_REGEXP
T_HASH
T_CLASS


Roots
T_STRING
T_OBJECT
T_ARRAY
T_REGEXP
T_HASH
T_CLASS

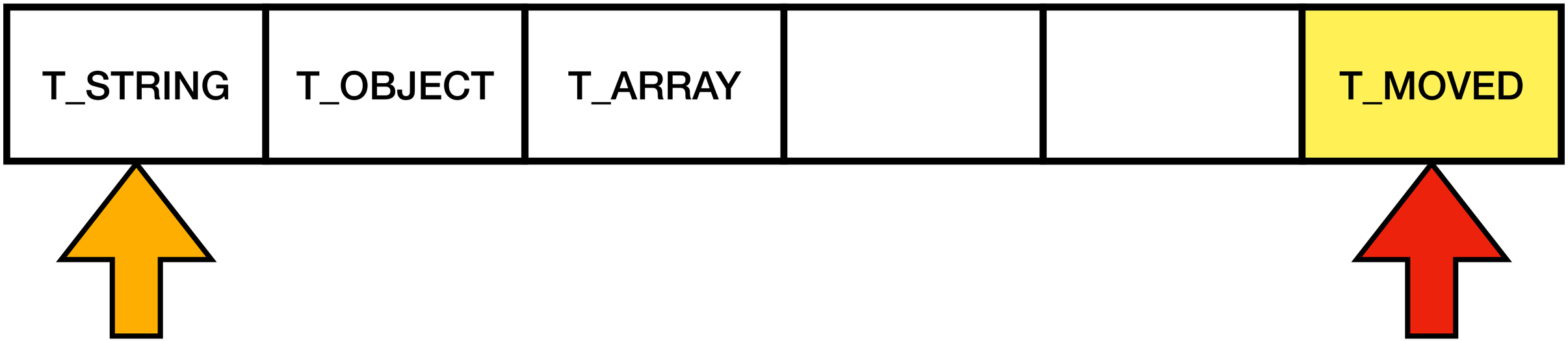

Roots
T_STRING
T_OBJECT
T_ARRAY
T_REGEXP
T_HASH
T_CLASS

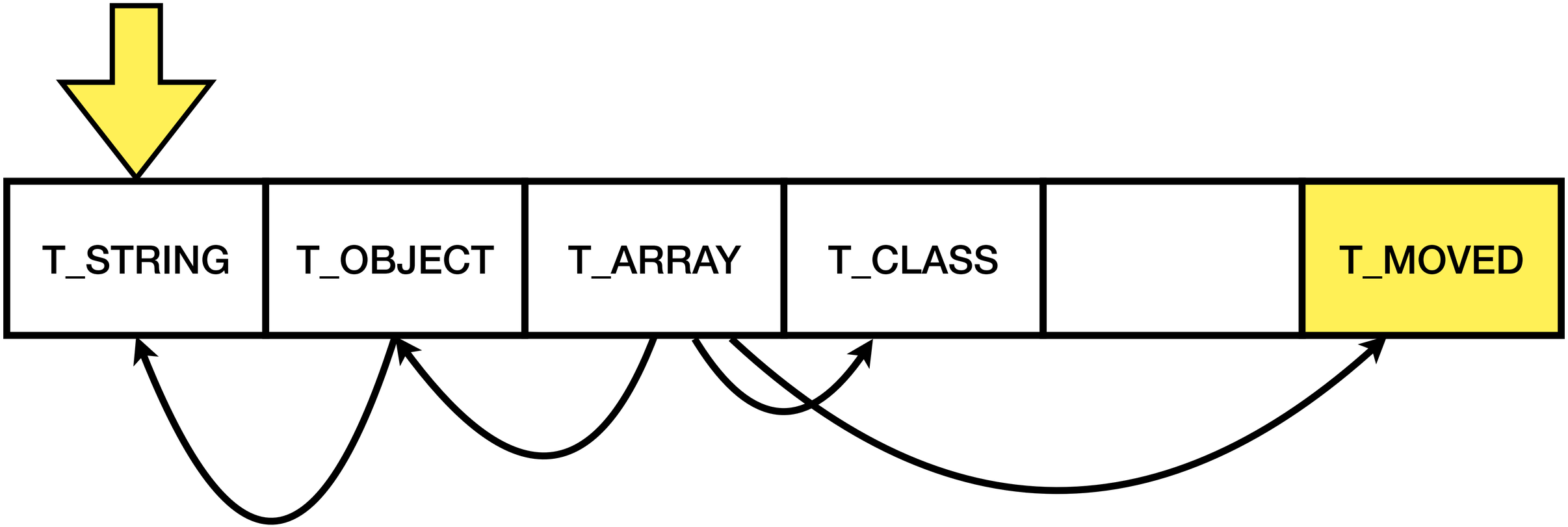

Roots
T_STRING
T_OBJECT
T_ARRAY
T_REGEXP
T_HASH
T_CLASS

Sweeping


T_STRING
T_OBJECT
T_ARRAY
T_REGEXP
T_HASH
T_CLASS

Compaction


T_STRING
T_OBJECT
T_ARRAY
T_MOVED


T_STRING
T_OBJECT
T_ARRAY
T_CLASS
T_MOVED

Garbage Collection in Ruby

Jul 03, 2023 (updated at Oct 02, 2023)

1. Overview

Ruby's garbage collector code lives in a single file called `gc.c`. "Garbage collector" is probably not the best term to call it because in addition to garbage collection, the code inside `gc.c` is responsible for memory allocation and management. In other words, the whole lifecycle of an object is managed by the garbage collector.

2. Primitive object types

What's Missing in Ruby's Garbage Collector?

Different GC Algorithms

Different GC Algorithms

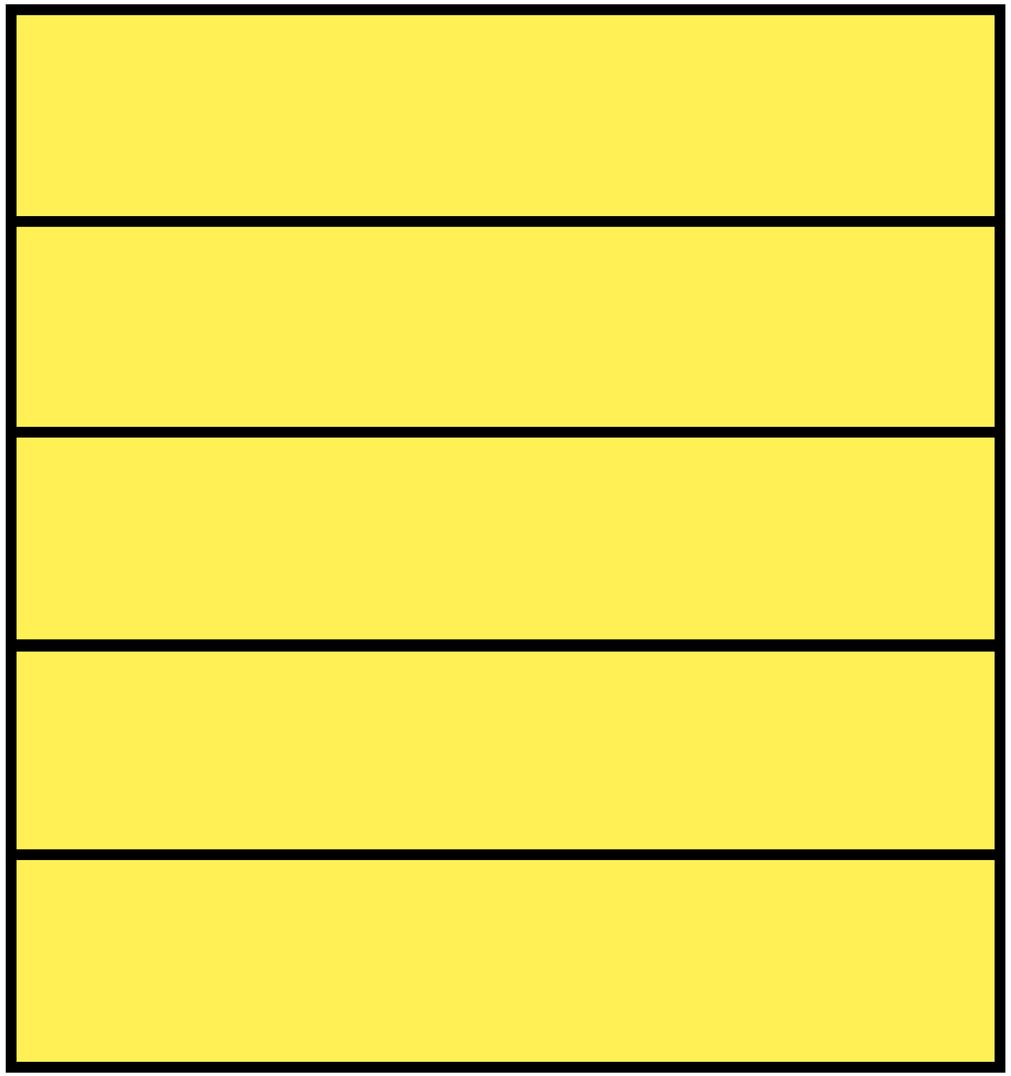
~~Reference Counting~~

Different GC Algorithms

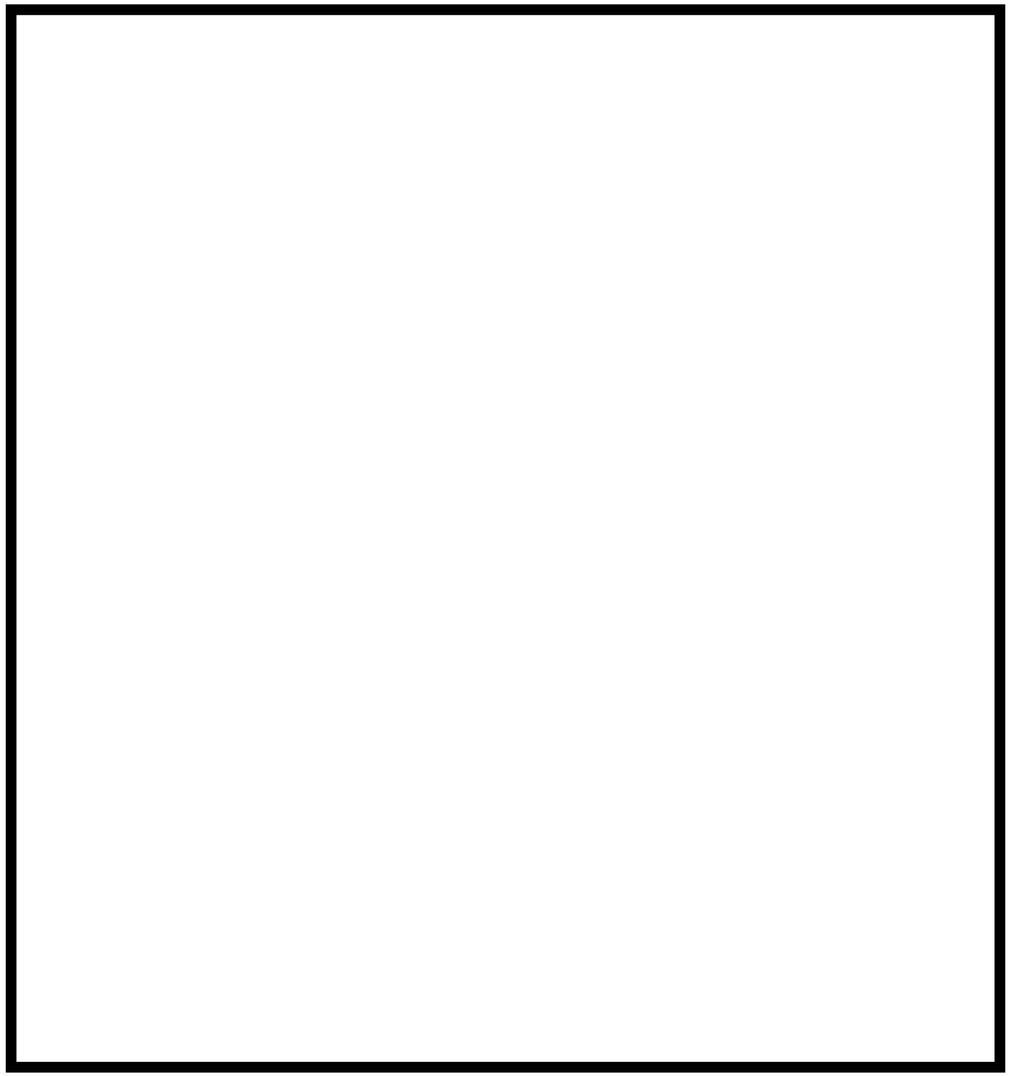
~~Reference Counting~~

Semispace Copying

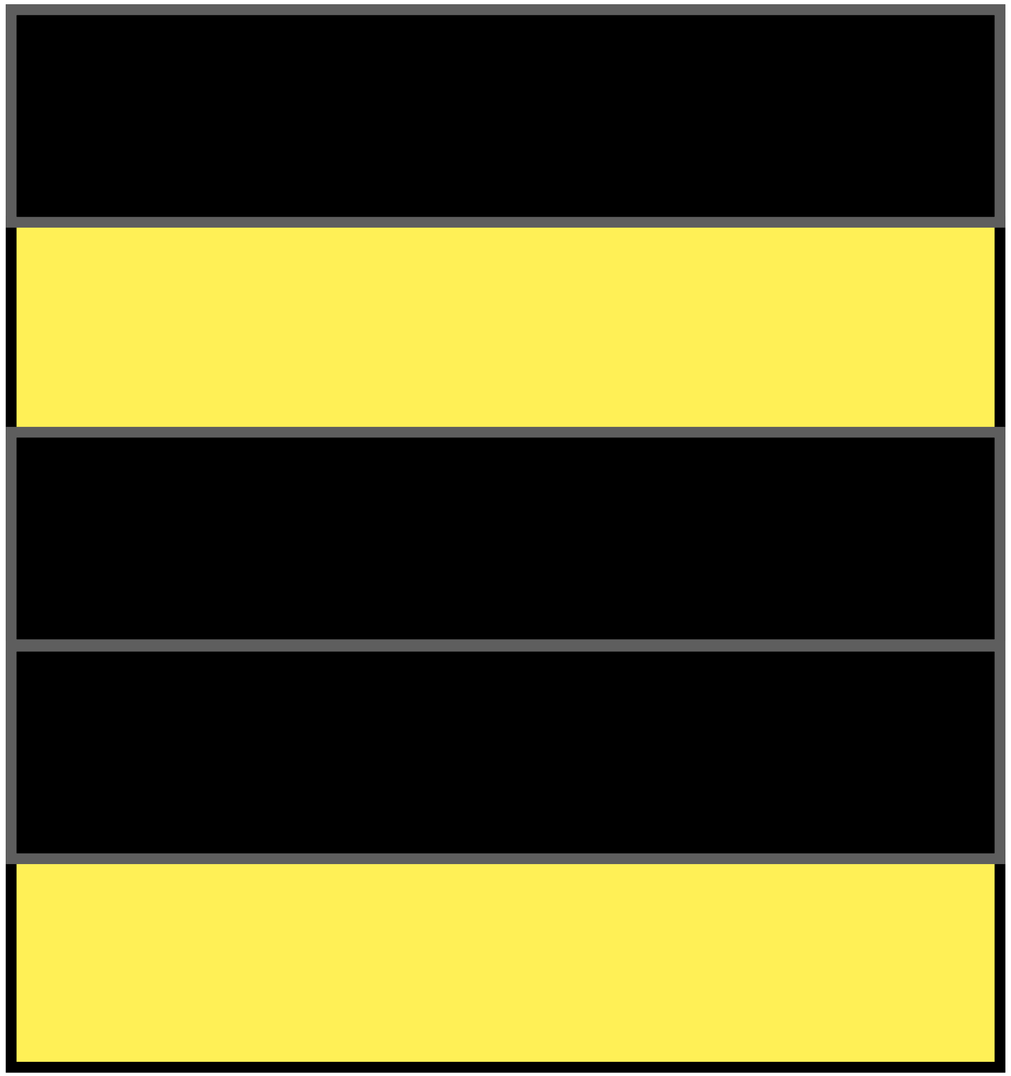
From-space



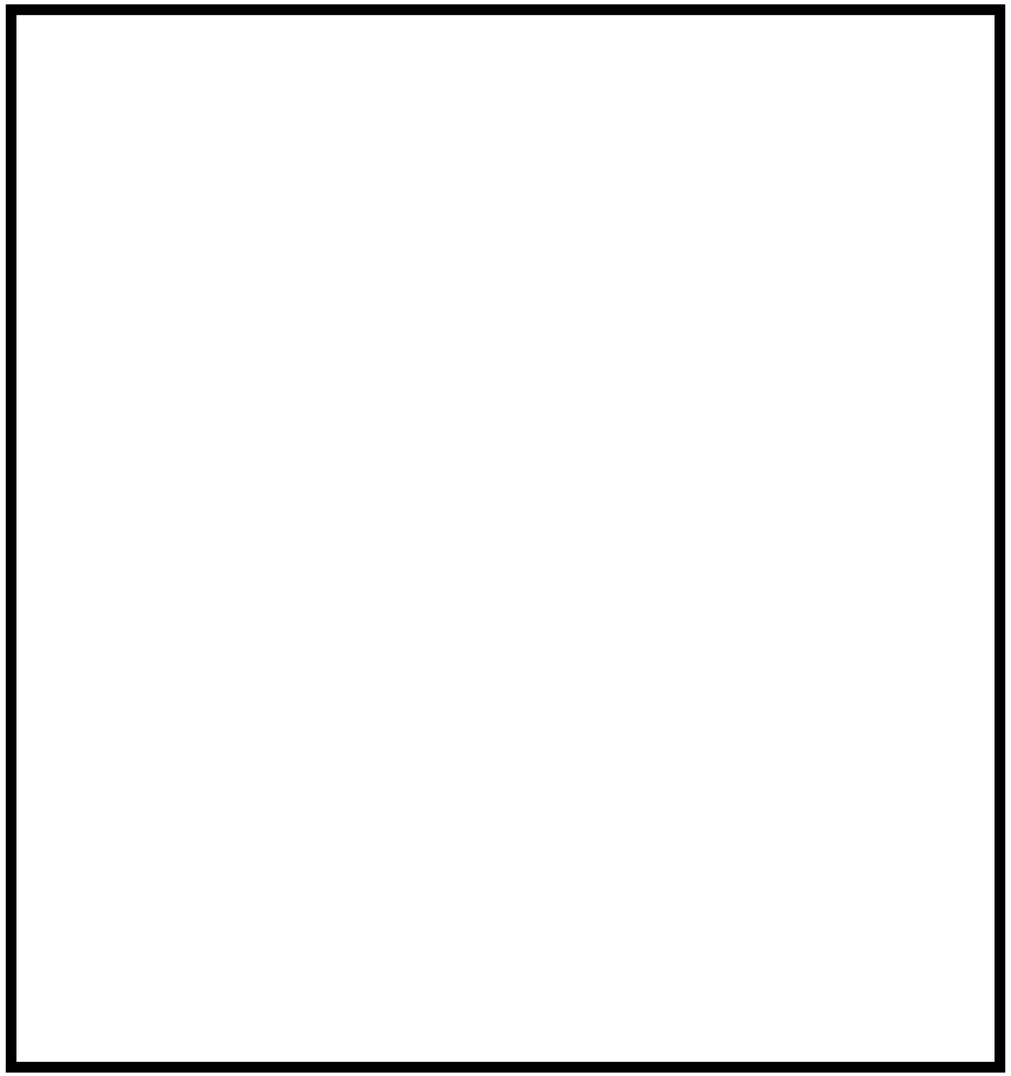
To-space



From-space



To-space



Different GC Algorithms

~~Reference Counting~~

~~Semispace Copying~~

Different GC Algorithms

~~Reference Counting~~

~~Semispace Copying~~

Immix

Parallelism

Dynamic Object Sizes

shopify.engineering/ruby-variable-width-allocation

BLOG | DEVELOPMENT

Optimizing Ruby's Memory Layout: Variable Width Allocation

Shopify is improving CRuby's performance in Ruby 3.2 by optimizing the memory layout in the garbage collector through the Variable Width Allocation project.

Engineering at

In this blog post, I'll be introducing how Shopify is improving CRuby's performance in Ruby 3.2 by optimizing the memory layout in the garbage

by Peter Zhu

Published on Dec 25, 2022

Dynamic Object Sizes

Rewrite Ruby GC

~~Rewrite Ruby GC~~

Modular Garbage Collectors

bugs.ruby-lang.org/issues/20470

Feature #20470 CLOSED

Edit Watch Like 1 Copy ···

Extract Ruby's Garbage Collector

« Previous | 3 of 71 | Next »

Added by peterzhu2118 (Peter Zhu) 11 months ago. Updated about 2 months ago.

Status: Closed
Assignee: -
Target version: -
[ruby-core:117765]

Description Quote

Extract Ruby's Garbage Collector

Background

As described in [Feature #20351], we are working on the ability to plug alternative garbage collector implementations into Ruby. Our goal is to allow developers and researchers to create and experiment with new implementations of garbage collectors in Ruby in a simplified way. This will also allow experimentation with different GC implementations in production systems so users can choose the best GC implementation for their workloads.

Implementation

GitHub PR: 10721

In this patch, we have split the current `gc.c` file into two files: `gc.c` and `gc_impl.c`.

`gc.c` now only contains code not specific to Ruby GC. This includes code to mark objects (which the GC implementation may choose not to use) and wrappers for internal APIs that the implementation may need to use (e.g. locking the VM).

`gc_impl.c` now contains the implementation of Ruby's GC. This includes marking, sweeping, compaction, and statistics. Most importantly, `gc_impl.c` only uses public APIs in Ruby and a limited set of functions exposed in `gc.c`. This allows us to build `gc_impl.c` independently of Ruby and plug Ruby's GC into itself.

Demonstration

After checking out the branch, we can first configure with `--with-shared-gc`:

```
$ ./configure --with-shared-gc
...
$ make -j
...
```

Let's now change the slot size of the GC to 64 bytes:

```
$ sed -i 's/\(#define BASE_SLOT_SIZE\).*/\1 64/' gc_impl.c
```

We can compile `gc_impl.c` independently using the following commands for clang or gcc (you may have to change the last `-I` to match your architecture and platform):

```
$ clang -Iinclude -I. -I.ext/include/arm64-darwin23 -undefined dynamic_lookup -g -O3 -dynamiclib -o libgc.dylib gc_impl.c
$ gcc -Iinclude -I. -I.ext/include/x86_64-linux -Wl,-undefined,dynamic_lookup -fPIC -g -O3 -shared -o libgc.so gc_impl.c
```

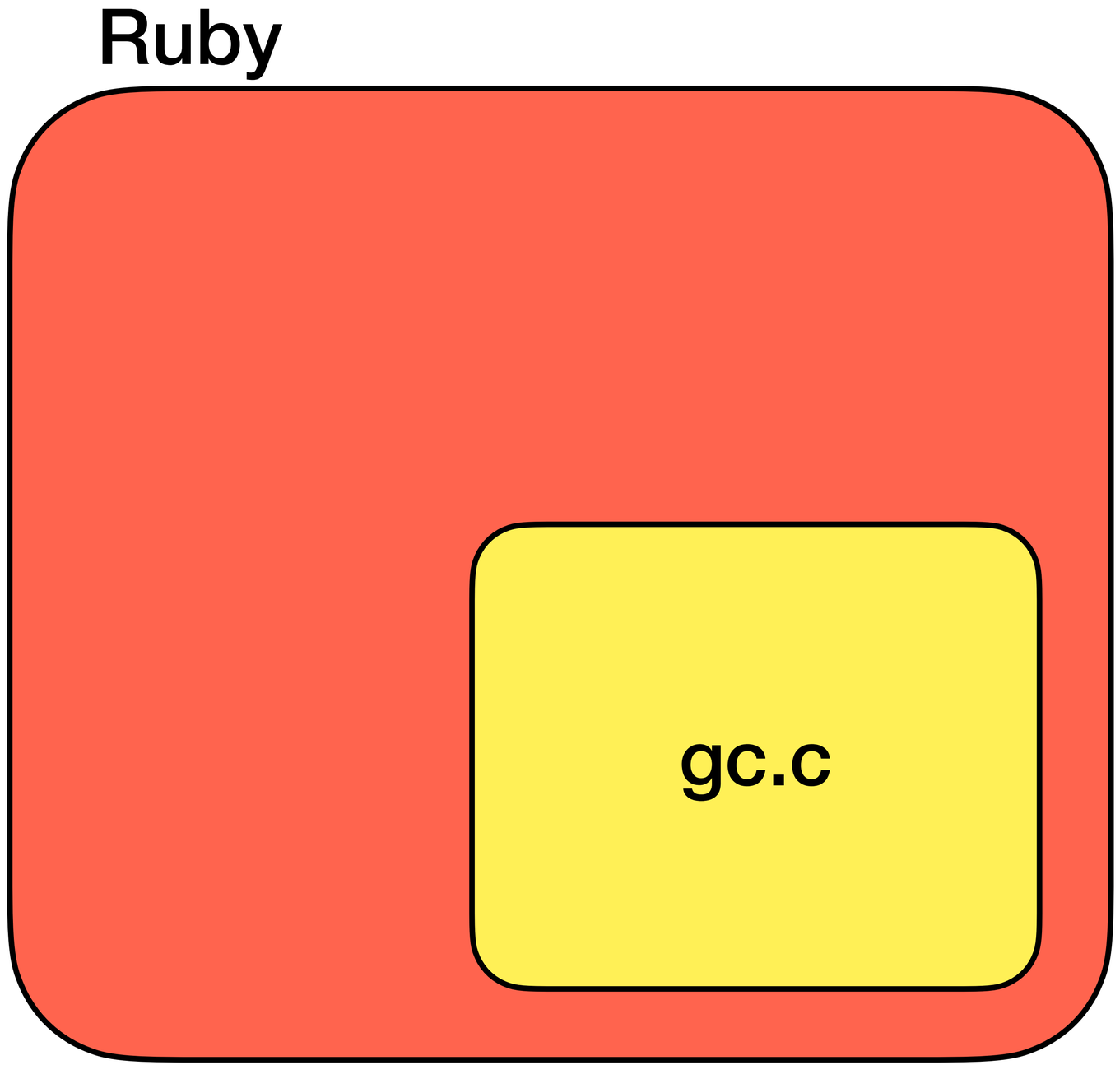
We can see that by default, the slot size is 40 bytes and objects are 40 bytes in size:

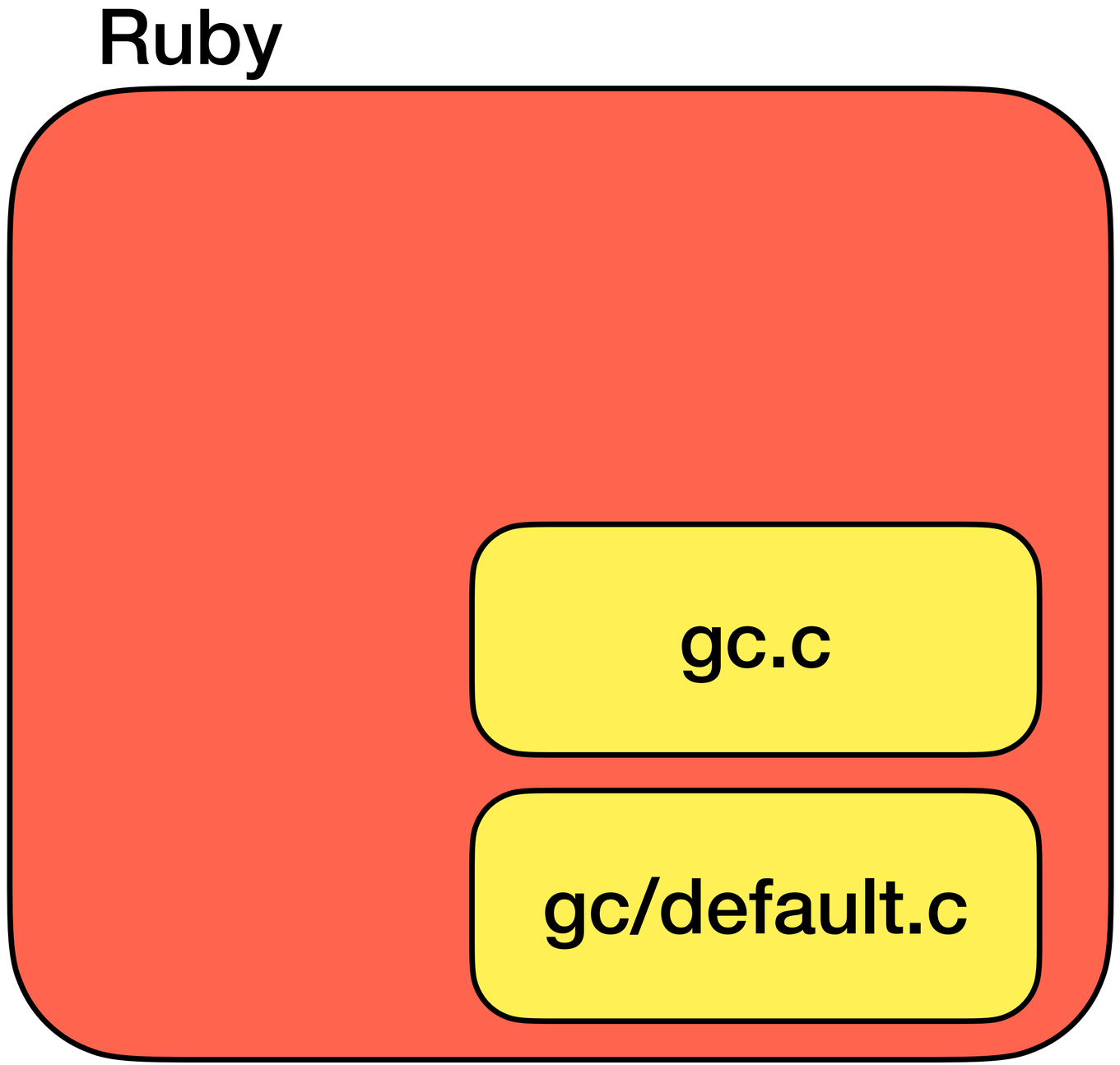
```
$ ./ruby -e "puts GC.stat_heap(0, :slot_size)"
40
$ ./ruby -robjspace -e "puts ObjectSpace.dump(Object.new)"
{"address":"0x1054a23f0", "type":"OBJECT", "shape_id":3, "slot_size":40, "class":"0x10528fd38", "embedded":true, "ivars":0, "memsize":40, "flags":{"wb_protected":true}}
```

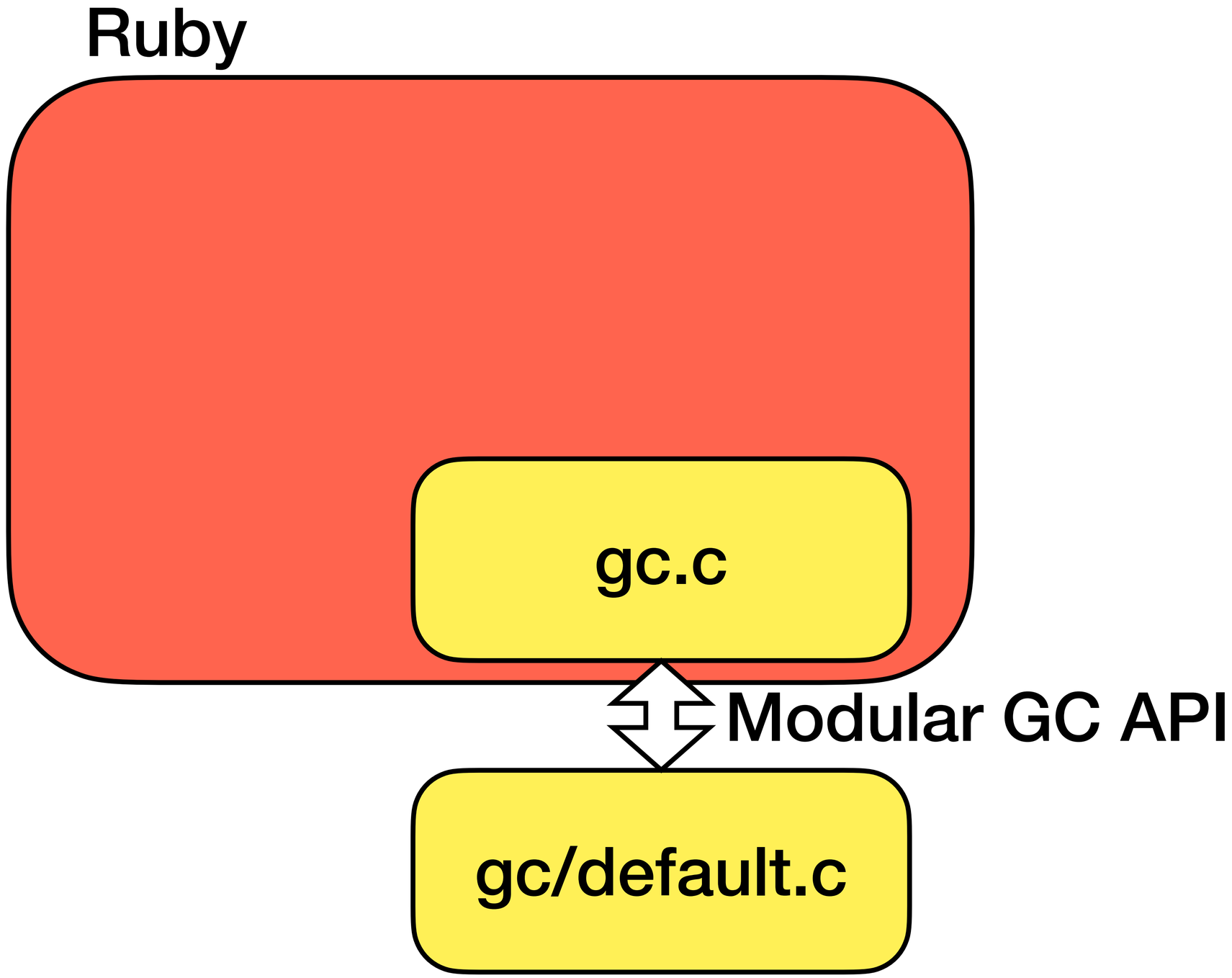
We can now load our new GC using the `RUBY_GC_LIBRARY_PATH` environment variable (note that you may have to change the path to the DSO):

```
$ RUBY_GC_LIBRARY_PATH=./libgc.dylib ./ruby -e "puts GC.stat_heap(0, :slot_size)"
64
```

Extracting Ruby's Default GC


Ruby
gc.c


Ruby
gc.c
gc/default.c


Ruby
gc.c
Modular GC API
gc/default.c

Modular GC API

gc/gc_impl.h

```
// Bootup
void *rb_gc_impl_objspace_alloc(void);
void rb_gc_impl_objspace_init(void *objspace_ptr);
void *rb_gc_impl_ractor_cache_alloc(void *objspace_ptr, void *ractor);
void rb_gc_impl_set_params(void *objspace_ptr);
void rb_gc_impl_init(void);
size_t *rb_gc_impl_heap_sizes(void *objspace_ptr);
// Shutdown
void rb_gc_impl_shutdown_free_objects(void *objspace_ptr);
void rb_gc_impl_objspace_free(void *objspace_ptr);
void rb_gc_impl_ractor_cache_free(void *objspace_ptr, void *cache);
// GC
void rb_gc_impl_start(void *objspace_ptr, bool full_mark, bool immediate_mark,
bool immediate_sweep, bool compact);
bool rb_gc_impl_during_gc_p(void *objspace_ptr);
void rb_gc_impl_prepare_heap(void *objspace_ptr);
void rb_gc_impl_gc_enable(void *objspace_ptr);
void rb_gc_impl_gc_disable(void *objspace_ptr, bool finish_current_gc);
bool rb_gc_impl_gc_enabled_p(void *objspace_ptr);
void rb_gc_impl_stress_set(void *objspace_ptr, VALUE flag);
VALUE rb_gc_impl_stress_get(void *objspace_ptr);
VALUE rb_gc_impl_config_get(void *objspace_ptr);
void rb_gc_impl_config_set(void *objspace_ptr, VALUE hash);
// Object allocation
VALUE rb_gc_impl_new_obj(void *objspace_ptr, void *cache_ptr, VALUE klass, VALUE
flags, VALUE v1, VALUE v2, VALUE v3, bool wb_protected, size_t alloc_size);
size_t rb_gc_impl_obj_slot_size(VALUE obj);
size_t rb_gc_impl_heap_id_for_size(void *objspace_ptr, size_t size);
bool rb_gc_impl_size_allocatable_p(size_t size);
// Malloc
void *rb_gc_impl_malloc(void *objspace_ptr, size_t size);
void *rb_gc_impl_calloc(void *objspace_ptr, size_t size);
void *rb_gc_impl_realloc(void *objspace_ptr, void *ptr, size_t new_size, size_t
old_size);
void rb_gc_impl_free(void *objspace_ptr, void *ptr, size_t old_size);
void rb_gc_impl_adjust_memory_usage(void *objspace_ptr, ssize_t diff);
// Marking
void rb_gc_impl_mark(void *objspace_ptr, VALUE obj);
void rb_gc_impl_mark_and_move(void *objspace_ptr, VALUE *ptr);
void rb_gc_impl_mark_and_pin(void *objspace_ptr, VALUE obj);
void rb_gc_impl_mark_maybe(void *objspace_ptr, VALUE obj);
void rb_gc_impl_mark_weak(void *objspace_ptr, VALUE *ptr);
void rb_gc_impl_remove_weak(void *objspace_ptr, VALUE parent_obj, VALUE *ptr);
// Compaction
bool rb_gc_impl_object_moved_p(void *objspace_ptr, VALUE obj);
VALUE rb_gc_impl_location(void *objspace_ptr, VALUE value);
// Write barriers
void rb_gc_impl_writebarrier(void *objspace_ptr, VALUE a, VALUE b);
void rb_gc_impl_writebarrier_unprotect(void *objspace_ptr, VALUE obj);
void rb_gc_impl_writebarrier_remember(void *objspace_ptr, VALUE obj);
// Heap walking
void rb_gc_impl_each_objects(void *objspace_ptr, int (*callback)(void *, void *,
size_t, void *), void *data);
void rb_gc_impl_each_object(void *objspace_ptr, void (*func)(VALUE obj, void
*data), void *data);
// Finalizers
void rb_gc_impl_make_zombie(void *objspace_ptr, VALUE obj, void (*dfree)(void *),
void *data);
VALUE rb_gc_impl_define_finalizer(void *objspace_ptr, VALUE obj, VALUE block);
void rb_gc_impl_undefine_finalizer(void *objspace_ptr, VALUE obj);
void rb_gc_impl_copy_finalizer(void *objspace_ptr, VALUE dest, VALUE obj);
void rb_gc_impl_shutdown_call_finalizer(void *objspace_ptr);
// Object ID
VALUE rb_gc_impl_object_id(void *objspace_ptr, VALUE obj);
VALUE rb_gc_impl_object_id_to_ref(void *objspace_ptr, VALUE object_id);
// Forking
void rb_gc_impl_before_fork(void *objspace_ptr);
void rb_gc_impl_after_fork(void *objspace_ptr, rb_pid_t pid);
// Statistics
void rb_gc_impl_set_measure_total_time(void *objspace_ptr, VALUE flag);
bool rb_gc_impl_get_measure_total_time(void *objspace_ptr);
unsigned long long rb_gc_impl_get_total_time(void *objspace_ptr);
size_t rb_gc_impl_gc_count(void *objspace_ptr);
VALUE rb_gc_impl_latest_gc_info(void *objspace_ptr, VALUE key);
VALUE rb_gc_impl_stat(void *objspace_ptr, VALUE hash_or_sym);
VALUE rb_gc_impl_stat_heap(void *objspace_ptr, VALUE heap_name, VALUE
hash_or_sym);
const char *rb_gc_impl_active_gc_name(void);
// Miscellaneous
struct rb_gc_object_metadata_entry *rb_gc_impl_object_metadata(void *objspace_ptr,
VALUE obj);
bool rb_gc_impl_pointer_to_heap_p(void *objspace_ptr, const void *ptr);
bool rb_gc_impl_garbage_object_p(void *objspace_ptr, VALUE obj);
void rb_gc_impl_set_event_hook(void *objspace_ptr, const rb_event_flag_t event);
void rb_gc_impl_copy_attributes(void *objspace_ptr, VALUE dest, VALUE obj);
```

```
// Bootup
void *rb_gc_impl_objspace_alloc(void);
void rb_gc_impl_objspace_init(void *objspace_ptr);
void *rb_gc_impl_ractor_cache_alloc(void *objspace_ptr, void *ractor);
void rb_gc_impl_set_params(void *objspace_ptr);
void rb_gc_impl_init(void);
size_t *rb_gc_impl_heap_sizes(void *objspace_ptr);
// Shutdown
void rb_gc_impl_shutdown_free_objects(void *objspace_ptr);
void rb_gc_impl_objspace_free(void *objspace_ptr);
void rb_gc_impl_ractor_cache_free(void *objspace_ptr, void *cache);
// GC
void rb_gc_impl_start(void *objspace_ptr, bool full_mark, bool immediate_mark,
bool immediate_sweep, bool compact);
bool rb_gc_impl_during_gc_p(void *objspace_ptr);
```

```
// Bootup
void *rb_gc_impl_objspace_alloc(void);
void rb_gc_impl_objspace_init(void *objspace_ptr);
void *rb_gc_impl_ractor_cache_alloc(void *objspace_ptr, void *ractor);
void rb_gc_impl_set_params(void *objspace_ptr);
void rb_gc_impl_init(void);
size_t *rb_gc_impl_heap_sizes(void *objspace_ptr);
// Shutdown
void rb_gc_impl_shutdown_free_objects(void *objspace_ptr);
void rb_gc_impl_objspace_free(void *objspace_ptr);
void rb_gc_impl_ractor_cache_free(void *objspace_ptr, void *cache);
// GC
void rb_gc_impl_start(void *objspace_ptr, bool full_mark, bool immediate_mark,
bool immediate_sweep, bool compact);
bool rb_gc_impl_during_gc_p(void *objspace_ptr);
void rb_gc_impl_prepare_heap(void *objspace_ptr);
void rb_gc_impl_gc_enable(void *objspace_ptr);
void rb_gc_impl_gc_disable(void *objspace_ptr, bool finish_current_gc);
bool rb_gc_impl_gc_enabled_p(void *objspace_ptr);
void rb_gc_impl_stress_set(void *objspace_ptr, VALUE flag);
VALUE rb_gc_impl_stress_get(void *objspace_ptr);
VALUE rb_gc_impl_config_get(void *objspace_ptr);
```

```
void rb_gc_impl_init(void);
size_t *rb_gc_impl_heap_sizes(void *objspace_ptr);
// Shutdown
void rb_gc_impl_shutdown_free_objects(void *objspace_ptr);
void rb_gc_impl_objspace_free(void *objspace_ptr);
void rb_gc_impl_ractor_cache_free(void *objspace_ptr, void *cache);
// GC
void rb_gc_impl_start(void *objspace_ptr, bool full_mark, bool immediate_mark,
bool immediate_sweep, bool compact);
bool rb_gc_impl_during_gc_p(void *objspace_ptr);
void rb_gc_impl_prepare_heap(void *objspace_ptr);
void rb_gc_impl_gc_enable(void *objspace_ptr);
void rb_gc_impl_gc_disable(void *objspace_ptr, bool finish_current_gc);
bool rb_gc_impl_gc_enabled_p(void *objspace_ptr);
void rb_gc_impl_stress_set(void *objspace_ptr, VALUE flag);
VALUE rb_gc_impl_stress_get(void *objspace_ptr);
VALUE rb_gc_impl_config_get(void *objspace_ptr);
void rb_gc_impl_config_set(void *objspace_ptr, VALUE hash);
// Object allocation
VALUE rb_gc_impl_new_obj(void *objspace_ptr, void *cache_ptr, VALUE klass, VALUE
flags, VALUE v1, VALUE v2, VALUE v3, bool wb_protected, size_t alloc_size);
size_t rb_gc_impl_obj_slot_size(VALUE obj);
size_t rb_gc_impl_heap_id_for_size(void *objspace_ptr, size_t size);
bool rb_gc_impl_size_allocatable_p(size_t size);
```

```
bool rb_gc_impl_during_gc_p(void *objspace_ptr);
void rb_gc_impl_prepare_heap(void *objspace_ptr);
void rb_gc_impl_gc_enable(void *objspace_ptr);
void rb_gc_impl_gc_disable(void *objspace_ptr, bool finish_current_gc);
bool rb_gc_impl_gc_enabled_p(void *objspace_ptr);
void rb_gc_impl_stress_set(void *objspace_ptr, VALUE flag);
VALUE rb_gc_impl_stress_get(void *objspace_ptr);
VALUE rb_gc_impl_config_get(void *objspace_ptr);
void rb_gc_impl_config_set(void *objspace_ptr, VALUE hash);
// Object allocation
VALUE rb_gc_impl_new_obj(void *objspace_ptr, void *cache_ptr, VALUE klass, VALUE
flags, VALUE v1, VALUE v2, VALUE v3, bool wb_protected, size_t alloc_size);
size_t rb_gc_impl_obj_slot_size(VALUE obj);
size_t rb_gc_impl_heap_id_for_size(void *objspace_ptr, size_t size);
bool rb_gc_impl_size_allocatable_p(size_t size);
// Malloc
void *rb_gc_impl_malloc(void *objspace_ptr, size_t size);
void *rb_gc_impl_calloc(void *objspace_ptr, size_t size);
void *rb_gc_impl_realloc(void *objspace_ptr, void *ptr, size_t new_size, size_t
old_size);
void rb_gc_impl_free(void *objspace_ptr, void *ptr, size_t old_size);
void rb_gc_impl_adjust_memory_usage(void *objspace_ptr, ssize_t diff);
// Marking
void rb_gc_impl_mark(void *objspace_ptr, VALUE obj);
```

```
size_t rb_gc_impl_heap_id_for_size(void *objspace_ptr, size_t size);
bool rb_gc_impl_size_allocatable_p(size_t size);
// Malloc
void *rb_gc_impl_malloc(void *objspace_ptr, size_t size);
void *rb_gc_impl_calloc(void *objspace_ptr, size_t size);
void *rb_gc_impl_realloc(void *objspace_ptr, void *ptr, size_t new_size, size_t
old_size);
void rb_gc_impl_free(void *objspace_ptr, void *ptr, size_t old_size);
void rb_gc_impl_adjust_memory_usage(void *objspace_ptr, ssize_t diff);
// Marking
void rb_gc_impl_mark(void *objspace_ptr, VALUE obj);
void rb_gc_impl_mark_and_move(void *objspace_ptr, VALUE *ptr);
void rb_gc_impl_mark_and_pin(void *objspace_ptr, VALUE obj);
void rb_gc_impl_mark_maybe(void *objspace_ptr, VALUE obj);
void rb_gc_impl_mark_weak(void *objspace_ptr, VALUE *ptr);
void rb_gc_impl_remove_weak(void *objspace_ptr, VALUE parent_obj, VALUE *ptr);
// Compaction
bool rb_gc_impl_object_moved_p(void *objspace_ptr, VALUE obj);
VALUE rb_gc_impl_location(void *objspace_ptr, VALUE value);
// Write barriers
void rb_gc_impl_writebarrier(void *objspace_ptr, VALUE a, VALUE b);
void rb_gc_impl_writebarrier_unprotect(void *objspace_ptr, VALUE obj);
void rb_gc_impl_writebarrier_remember(void *objspace_ptr, VALUE obj);
// Heap walking
```

```
void rb_gc_impl_mark(void *objspace_ptr, VALUE obj);
void rb_gc_impl_mark_and_move(void *objspace_ptr, VALUE *ptr);
void rb_gc_impl_mark_and_pin(void *objspace_ptr, VALUE obj);
void rb_gc_impl_mark_maybe(void *objspace_ptr, VALUE obj);
void rb_gc_impl_mark_weak(void *objspace_ptr, VALUE *ptr);
void rb_gc_impl_remove_weak(void *objspace_ptr, VALUE parent_obj, VALUE *ptr);
// Compaction
bool rb_gc_impl_object_moved_p(void *objspace_ptr, VALUE obj);
VALUE rb_gc_impl_location(void *objspace_ptr, VALUE value);
// Write barriers
void rb_gc_impl_writebarrier(void *objspace_ptr, VALUE a, VALUE b);
void rb_gc_impl_writebarrier_unprotect(void *objspace_ptr, VALUE obj);
void rb_gc_impl_writebarrier_remember(void *objspace_ptr, VALUE obj);
// Heap walking
void rb_gc_impl_each_objects(void *objspace_ptr, int (*callback)(void *, void *,
size_t, void *), void *data);
void rb_gc_impl_each_object(void *objspace_ptr, void (*func)(VALUE obj, void
*data), void *data);
// Finalizers
void rb_gc_impl_make_zombie(void *objspace_ptr, VALUE obj, void (*dfree)(void *),
void *data);
VALUE rb_gc_impl_define_finalizer(void *objspace_ptr, VALUE obj, VALUE block);
void rb_gc_impl_undefine_finalizer(void *objspace_ptr, VALUE obj);
void rb_gc_impl_copy_finalizer(void *objspace_ptr, VALUE dest, VALUE obj);
```

gc/gc.h

```
unsigned int rb_gc_vm_lock(void);
void rb_gc_vm_unlock(unsigned int lev);
unsigned int rb_gc_cr_lock(void);
void rb_gc_cr_unlock(unsigned int lev);
unsigned int rb_gc_vm_lock_no_barrier(void);
void rb_gc_vm_unlock_no_barrier(unsigned int lev);
void rb_gc_vm_barrier(void);
size_t rb_gc_obj_optimal_size(VALUE obj);
void rb_gc_mark_children(void *objspace, VALUE obj);
void rb_gc_vm_weak_table_foreach(vm_table_foreach_callback_func callback,
vm_table_update_callback_func update_callback, void *data, bool weak_only, enum
rb_gc_vm_weak_tables table);
void rb_gc_update_object_references(void *objspace, VALUE obj);
void rb_gc_update_vm_references(void *objspace);
void rb_gc_event_hook(VALUE obj, rb_event_flag_t event);
void *rb_gc_get_objspace(void);
size_t rb_size_mul_or_raise(size_t x, size_t y, VALUE exc);
void rb_gc_run_obj_finalizer(VALUE objid, long count, VALUE (*callback)(long i,
void *data), void *data);
void rb_gc_set_pending_interrupt(void);
void rb_gc_unset_pending_interrupt(void);
void rb_gc_obj_free_vm_weak_references(VALUE obj);
bool rb_gc_obj_free(void *objspace, VALUE obj);
void rb_gc_save_machine_context(void);
void rb_gc_mark_roots(void *objspace, const char **categoryp);
void rb_gc_ractor_newobj_cache_foreach(void (*func)(void *cache, void *data), void
*data);
bool rb_gc_multi_ractor_p(void);
void rb_objspace_reachable_objects_from_root(void (func)(const char *category,
VALUE, void *), void *passing_data);
void rb_objspace_reachable_objects_from(VALUE obj, void (func)(VALUE, void *),
void *data);
void rb_obj_info_dump(VALUE obj);
const char *rb_obj_info(VALUE obj);
bool rb_gc_shutdown_call_finalizer_p(VALUE obj);
uint32_t rb_gc_get_shape(VALUE obj);
void rb_gc_set_shape(VALUE obj, uint32_t shape_id);
uint32_t rb_gc_rebuild_shape(VALUE obj, size_t heap_id);
size_t rb_obj_memsize_of(VALUE obj);
void rb_gc_prepare_heap_process_object(VALUE obj);
bool ruby_free_at_exit_p(void);
bool rb_memerror_reentered(void);
bool rb_gc_event_hook_required_p(rb_event_flag_t event);
void *rb_gc_get_ractor_newobj_cache(void);
void rb_gc_initialize_vm_context(struct rb_gc_vm_context *context);
void rb_gc_worker_thread_set_vm_context(struct rb_gc_vm_context *context);
void rb_gc_worker_thread_unset_vm_context(struct rb_gc_vm_context *context);
```

```
unsigned int rb_gc_vm_lock(void);
void rb_gc_vm_unlock(unsigned int lev);
unsigned int rb_gc_cr_lock(void);
void rb_gc_cr_unlock(unsigned int lev);
unsigned int rb_gc_vm_lock_no_barrier(void);
void rb_gc_vm_unlock_no_barrier(unsigned int lev);
void rb_gc_vm_barrier(void);
size_t rb_gc_obj_optimal_size(VALUE obj);
void rb_gc_mark_children(void *objspace, VALUE obj);
void rb_gc_vm_weak_table_foreach(vm_table_foreach_callback_func callback,
vm_table_update_callback_func update_callback, void *data, bool weak_only, enum
rb_gc_vm_weak_tables table);
void rb_gc_update_object_references(void *objspace, VALUE obj);
void rb_gc_update_vm_references(void *objspace);
void rb_gc_event_hook(VALUE obj, rb_event_flag_t event);
void *rb_gc_get_objspace(void);
size_t rb_size_mul_or_raise(size_t x, size_t y, VALUE exc);
void rb_gc_run_obj_finalizer(VALUE objid, long count, VALUE (*callback)(long i,
void *data), void *data);
```

```
unsigned int rb_gc_vm_lock(void);
void rb_gc_vm_unlock(unsigned int lev);
unsigned int rb_gc_cr_lock(void);
void rb_gc_cr_unlock(unsigned int lev);
unsigned int rb_gc_vm_lock_no_barrier(void);
void rb_gc_vm_unlock_no_barrier(unsigned int lev);
void rb_gc_vm_barrier(void);
size_t rb_gc_obj_optimal_size(VALUE obj);
void rb_gc_mark_children(void *objspace, VALUE obj);
void rb_gc_vm_weak_table_foreach(vm_table_foreach_callback_func callback,
vm_table_update_callback_func update_callback, void *data, bool weak_only, enum
rb_gc_vm_weak_tables table);
void rb_gc_update_object_references(void *objspace, VALUE obj);
void rb_gc_update_vm_references(void *objspace);
void rb_gc_event_hook(VALUE obj, rb_event_flag_t event);
void *rb_gc_get_objspace(void);
size_t rb_size_mul_or_raise(size_t x, size_t y, VALUE exc);
void rb_gc_run_obj_finalizer(VALUE objid, long count, VALUE (*callback)(long i,
void *data), void *data);
void rb_gc_set_pending_interrupt(void);
void rb_gc_unset_pending_interrupt(void);
void rb_gc_obj_free_vm_weak_references(VALUE obj);
```

```
void rb_gc_update_object_references(void *objspace, VALUE obj);
void rb_gc_update_vm_references(void *objspace);
void rb_gc_event_hook(VALUE obj, rb_event_flag_t event);
void *rb_gc_get_objspace(void);
size_t rb_size_mul_or_raise(size_t x, size_t y, VALUE exc);
void rb_gc_run_obj_finalizer(VALUE objid, long count, VALUE (*callback)(long i,
void *data), void *data);
void rb_gc_set_pending_interrupt(void);
void rb_gc_unset_pending_interrupt(void);
void rb_gc_obj_free_vm_weak_references(VALUE obj);
bool rb_gc_obj_free(void *objspace, VALUE obj);
void rb_gc_save_machine_context(void);
void rb_gc_mark_roots(void *objspace, const char **categoryp);
void rb_gc_ractor_newobj_cache_foreach(void (*func)(void *cache, void *data), void
*data);
bool rb_gc_multi_ractor_p(void);
void rb_objspace_reachable_objects_from_root(void (func)(const char *category,
VALUE, void *), void *passing_data);
void rb_objspace_reachable_objects_from(VALUE obj, void (func)(VALUE, void *),
void *data);
void rb_obj_info_dump(VALUE obj);
const char *rb_obj_info(VALUE obj);
bool rb_gc_shutdown_call_finalizer_p(VALUE obj);
```

Building Modular GC

github.com/ruby/ruby/blob/master/gc/README.md

Ruby's Garbage Collectors

This directory contains implementations for Ruby's garbage collector (GC). The GC implementations use the Modular GC API to interact with Ruby. For more details about this API, see the Modular GC API section.

Two GC implementations are included in Ruby:

- Default: The GC implementation that is used by default in Ruby. This GC is stable and production ready. The implementation uses a mark-sweep-compact algorithm.
- MMTk: An experimental implementation using the MMTk framework. The code lives in the ruby/mmtk repository and is synchronized here. MMTk provides a wide variety of GC algorithms to choose from. For usage instructions and current progress, refer to the ruby/mmtk repository.

Building guide

> **Tip**
>
> If you are not sure how to build Ruby, follow the Building Ruby guide.

> **Important**
>
> Ruby's modular GC feature is experimental and subject to change. There may be bugs or performance impacts. Use at your own risk.

1. Configure Ruby with the `--with-modular-gc=<dir>` option, where `dir` is the directory you want to place the built GC libraries into.
2. Build Ruby as usual.
3. Build your desired GC implementation with `make install-modular-gc MODULAR_GC=<impl>`. This will build the GC implementation and place the built library into the `dir` specified in step 1. `impl` can be one of:
   - `default`: The default GC that Ruby ships with.
   - `mmtk`: The GC that uses MMTk as the back-end. See Ruby-specific details in the ruby/mmtk repository.
4. Run your desired GC implementation by setting the `RUBY_GC_LIBRARY=<lib>` environment variable, where `lib` could be `default`,


mmtk
MEMORY MANAGEMENT TOOLKIT

MMTk Features

GC Plans

GC Plans

NoGC

GC Plans

NoGC

Mark-Sweep

GC Plans

NoGC
Mark-Sweep
Mark-Compact

GC Plans

NoGC
Mark-Sweep
Mark-Compact
Semispace Copying

GC Plans

NoGC
Mark-Sweep
Mark-Compact
Semispace Copying
Immix

GC Plans

NoGC
Mark-Sweep
Mark-Compact
Semispace Copying
Immix
LXR

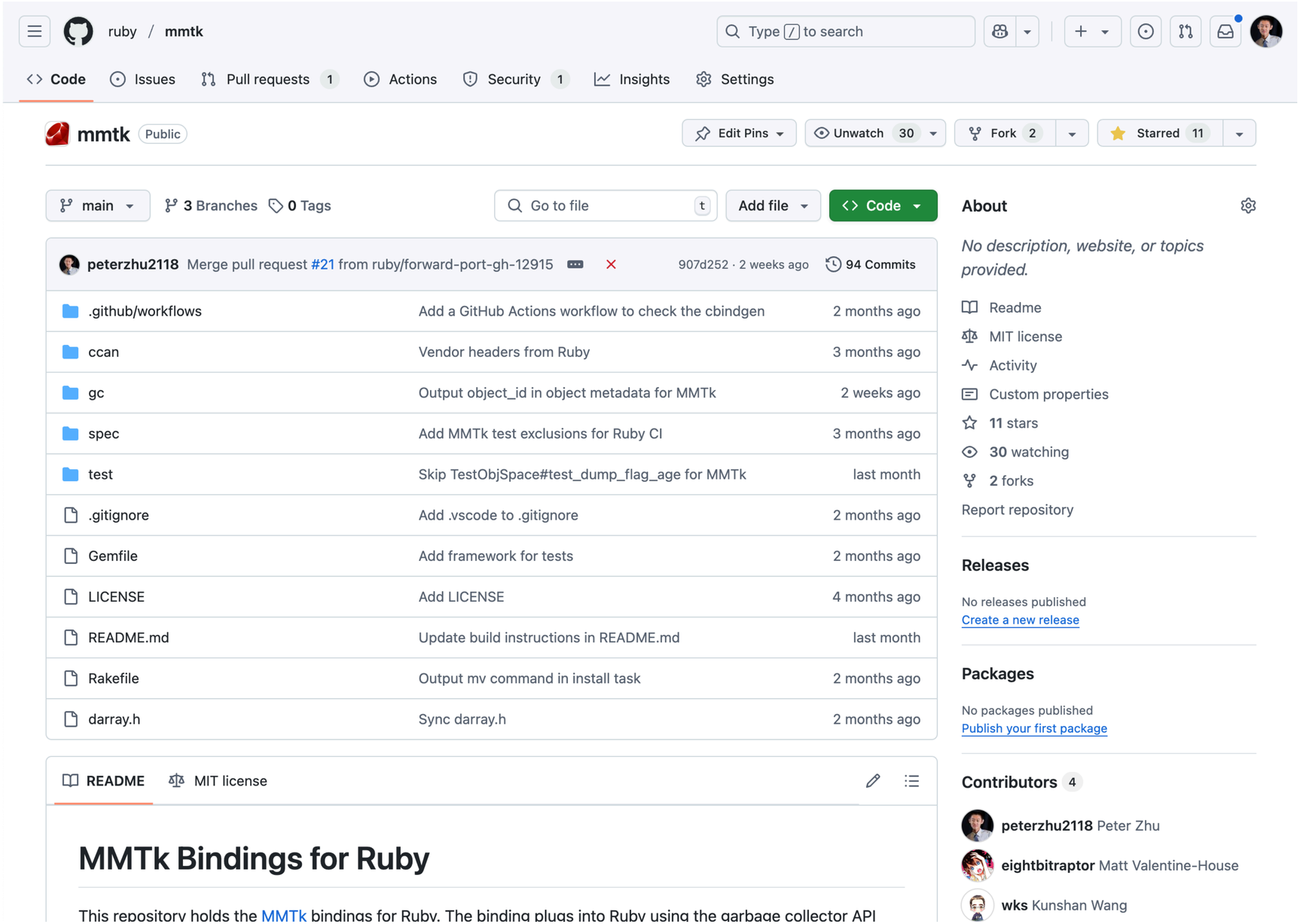
Parallelism

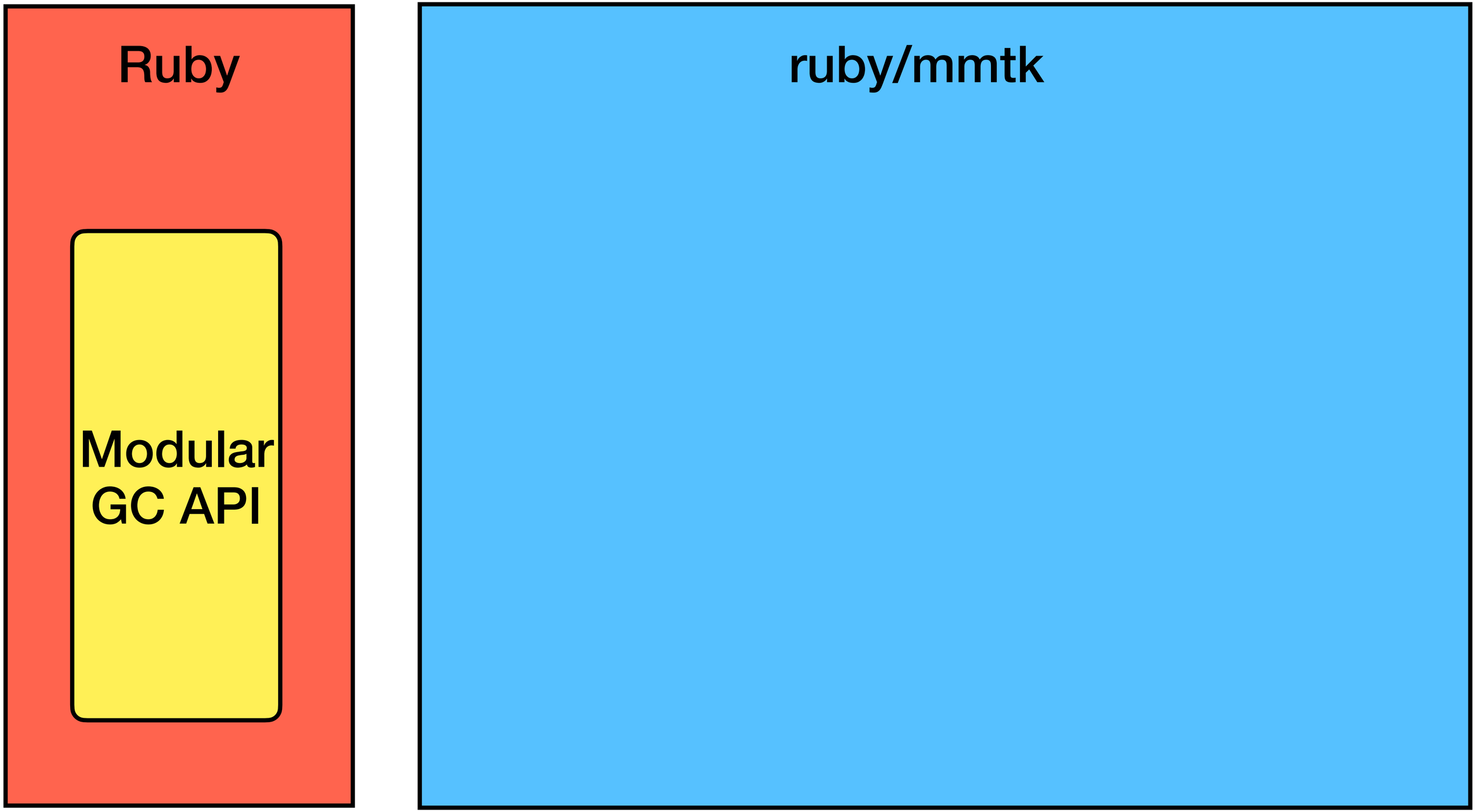
Two Implementations

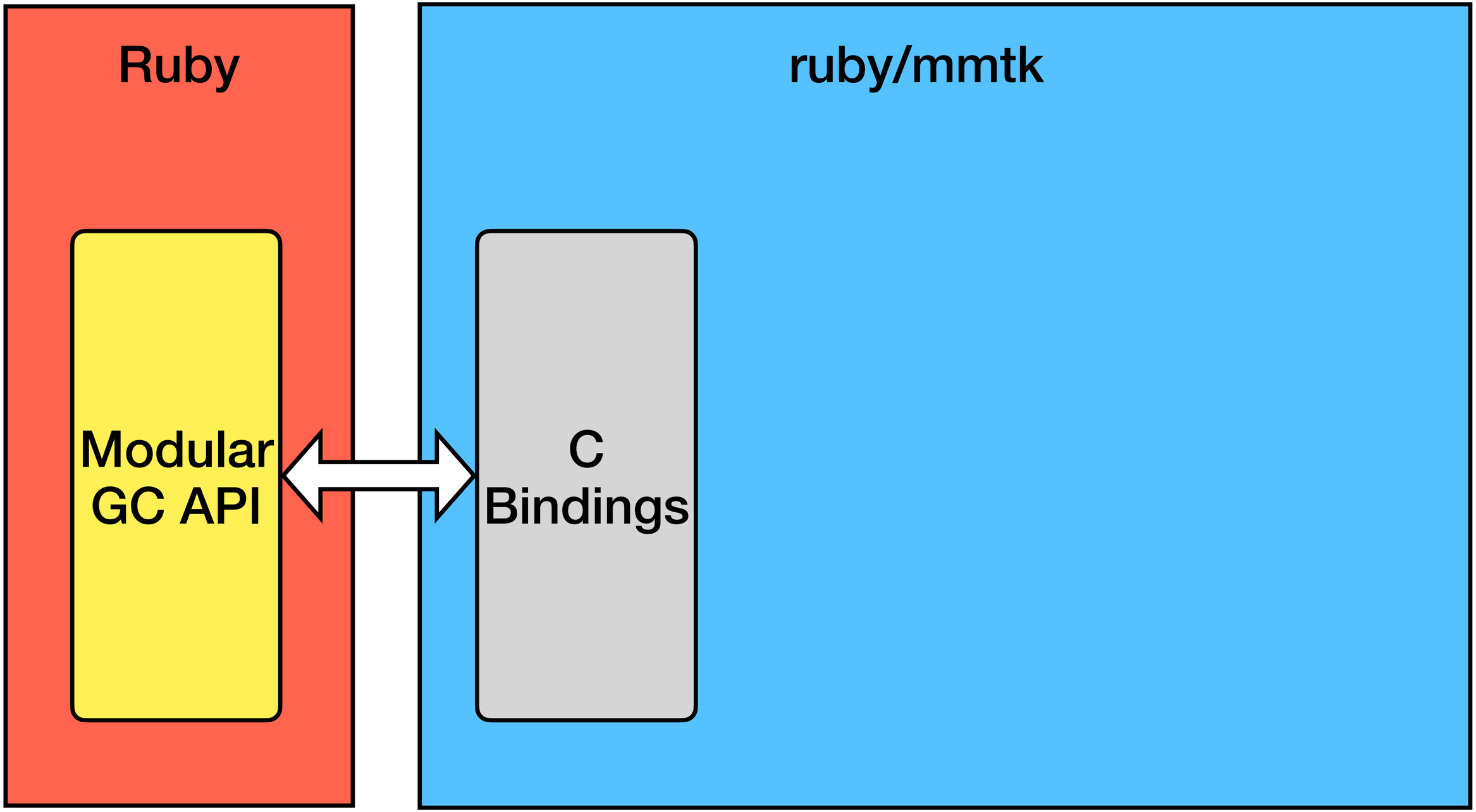
MMTk Team’s Implementation

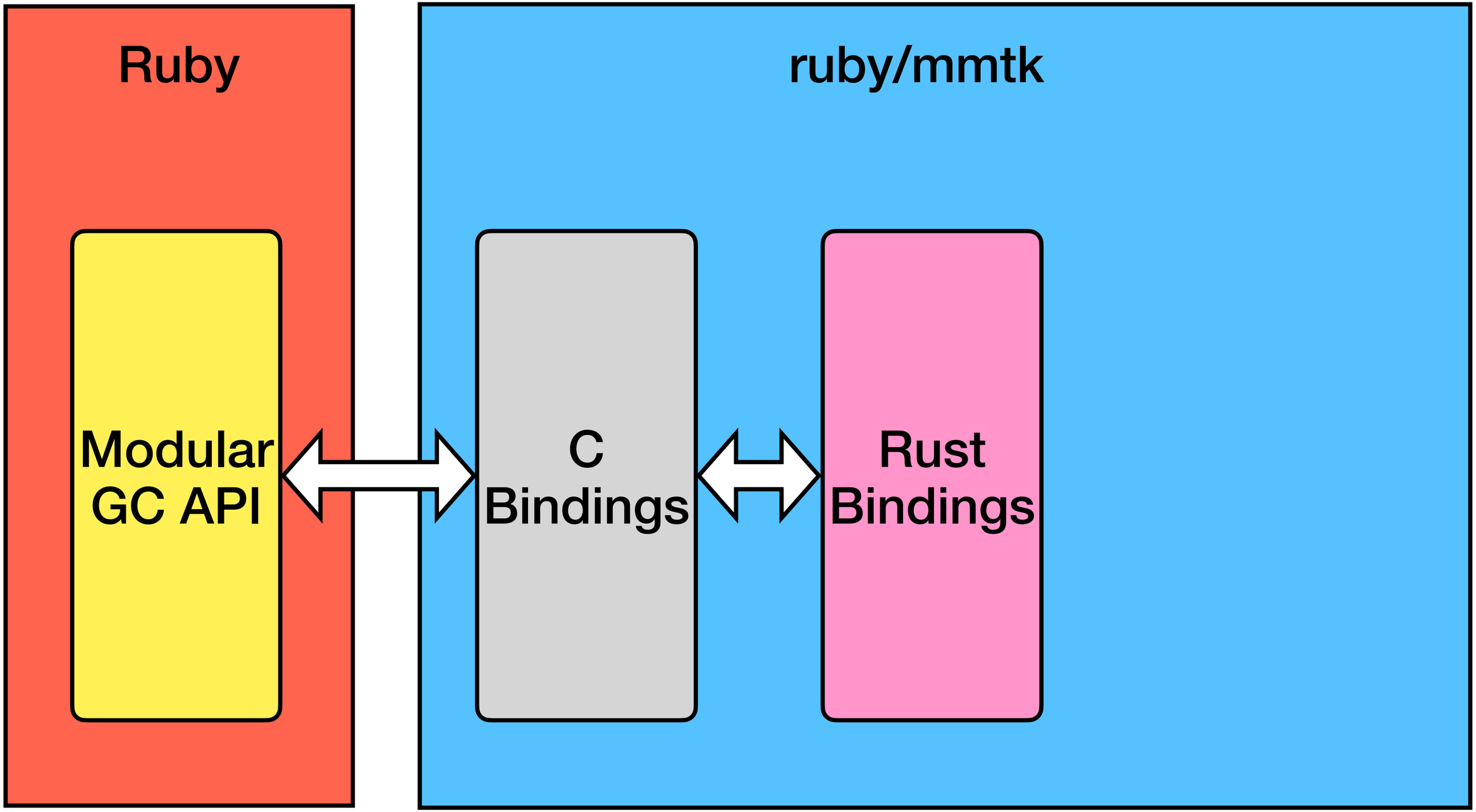
Reimplementing on Modular GC

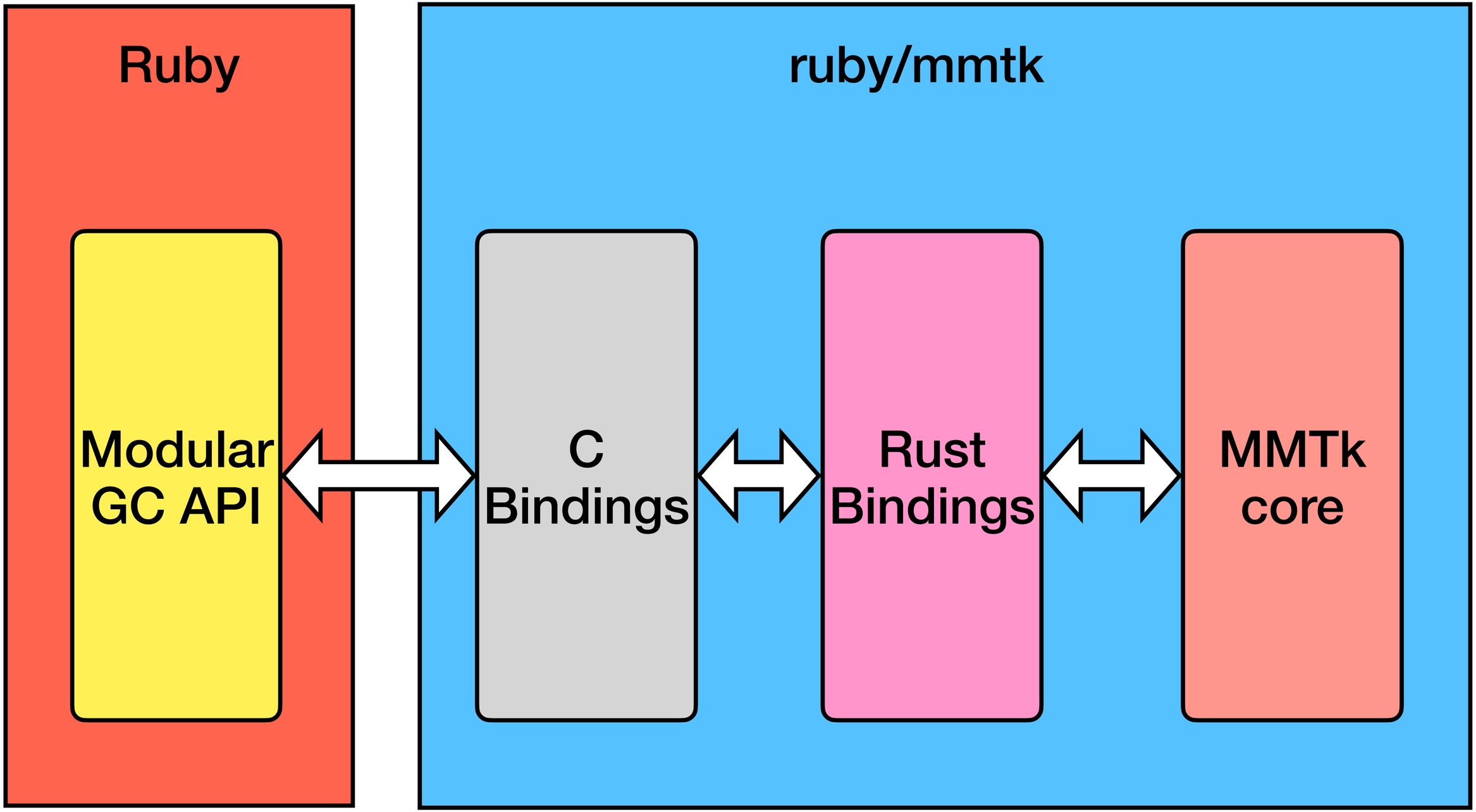
github.com/ruby/mmtk




Ruby
Modular
GC API
ruby/mmtk


Ruby
ruby/mmtk
Modular
GC API
C
Bindings


Ruby
ruby/mmtk
Modular
GC API
C
Bindings
Rust
Bindings


Ruby
ruby/mmtk
Modular GC API
C Bindings
Rust Bindings
MMTk core

Future Roadmap

Support for Moving Plans

Support for Generational GC

Improve Parallelism

Conclusion

Build Your Own GC!

Thank You!

blog.peterzhu.ca

peter@peterzhu.ca

@peterzhu2118

@peterzhu2118@ruby.social

@peterzhu.ca

@peterzhu.photos