Breadth-first Cycle Collection Reference Counting: Theory and a Rust Smart Pointer Implementation

Saverio Giallorenzo, Francesco Goretti

To cite this version:

Saverio Giallorenzo, Francesco Goretti. Breadth-first Cycle Collection Reference Counting: Theory and a Rust Smart Pointer Implementation. 40th ACM/SIGAPP Symposium on Applied Computing, SAC 2025, Mar 2025, Catania (IT), Italy. ⟨10.1145/3672608.3707785⟩. ⟨hal-04887627⟩

HAL Id: hal-04887627
https://hal.science/hal-04887627v1

Submitted on 15 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Breadth-first Cycle Collection Reference Counting: Theory and a Rust Smart Pointer Implementation

Saverio Giallorenzo
Alma Mater Studiorum - Università di Bologna
Italy
INRIA
France
saverio.giallorenzo2@unibo.it

Francesco Goretti
Alma Mater Studiorum - Università di Bologna
Italy
francesco.goretti2@studio.unibo.it

Abstract

We present a new garbage collection reference counting algorithm capable of collecting reference cycles—overcoming a known limitation of traditional reference counting. The algorithm's key features include resilience to errors during tracing, support for object finalisation, no need for supplementary heap memory during collection, and a fast breadth-first tracing approach that avoids stack overflows. We implement the algorithm as a Rust library that is idiomatic and highly compatible with the Rust ecosystem and that leverages Rust's type system and borrow checker to minimise unsafe code and prevent undefined behaviour. We report benchmarks that show that our proposal performs comparably to popular Rust alternatives and outperforms them when dealing with garbage cycles.

CCS Concepts

• **Software and its engineering → Garbage collection**.

Keywords

Garbage Collection, Reference Counting, Cycle Collection, Rust

ACM Reference Format:
Saverio Giallorenzo and Francesco Goretti. 2025. Breadth-first Cycle Collection Reference Counting: Theory and a Rust Smart Pointer Implementation. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25), March 31-April 4, 2025, Catania, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3672608.3707785

1 Introduction

Reference counting is one of the first techniques for garbage collection [9]. While it provides good performance, an important limitation regards the impossibility to free cyclic structures. Later refinements introduced companion mark-and-sweep routines, with the related overhead, to deal with cycle collection [3, 6, 19, 22]. Alternative approaches minimise overheads using cycle collectors [7, 8].

Inspired by state-of-the-art cycle collectors such as by Lins [18] and Bacon and Rajan [4], we present the Single-threaded Intrusive Lists Breadth-first Recycler (SILB-Recycler), a garbage collection reference counting algorithm able to collect reference cycles. SILB-Recycler distinctive features include: a) resiliency to fatal errors during the tracing phase, b) support for object finalisation, c) no need for supplementary heap memory during collection, and d) fast breadth-first tracing approach (no stack overflows).

Sketching out SILB-Recycler' logic, detailed in Section 2, in the absence of cycles, the algorithm behaves as a standard reference counting system, finalising and deallocating objects when their reference count reaches zero. However, if an object belongs to a cycle, the collector finalises and deallocates it when the last strong pointer to the cycle is destroyed — when a strong pointer is destroyed, the collector adds its object to a list of potential cyclic garbage.

Cycle identification follows two phases. The first is the *trace counting* phase, which performs a breadth-first traversal starting from the list of potential cyclic garbage, marking the visited objects as traced. Each object has a separate tracing counter incremented for each followed pointer. After having finished the visit, the collector compares the tracing counter with the reference counter of the object to identify potential roots, separating the objects between two list of non-root and root objects. The second phase, *trace roots*, begins the visit from the root objects, traversing all the references of the encountered objects. When the collector visits an object, it removes it from its list and marks it as such. Upon completion, the root list is empty and the non-root list contains all and only the collectible objects in unreachable cycles.

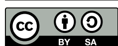
Besides introducing SILB-Recycler, in Section 3, we present an implementation of the proposed algorithm as a Rust library, called `rust-cc`. The library provides a SILB-Recycler-based smart pointer (that manages the memory it references) whose APIs follow those of Rust's standard library (e.g., `Rc`), achieving both familiarity for Rust developers and seamless integration within the Rust ecosystem. The library, highly integrated in Rust's package manager (Cargo), makes minimal use of `unsafe` code and leverages Rust's type system and borrow checker to prevent undefined behaviour.

In Section 4, we use `rust-cc` to benchmark SILB-Recycler against popular Rust alternatives, in particular, `gc` and `safe-gc`, resp. the de-facto standard garbage-collection library and an alternative safe implementation, and one implementing Bacon-Rajan [4]. The main takeaway is that `rust-cc` is generally on par with the fastest alternatives and faster than these when dealing with garbage cycles.

We conclude by positioning our contribution with related work, in Section 5, and drawing final remarks and future steps in Section 6.

2 Breadth-first Cycle Collection Reference Counting

We now present our novel cycle collection algorithm: the structure of the objects it manages, how finalisation and weak pointers work, and a discussion on the details of its logic.



This work is licensed under a Creative Commons 4.0 International License.
SAC '25, March 31-April 4, 2025, Catania, Italy
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0629-5/25/03
https://doi.org/10.1145/3672608.3707785

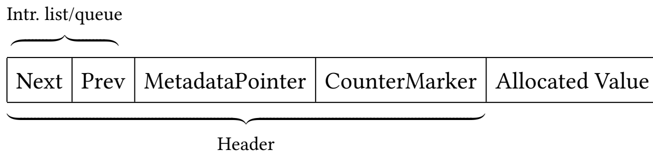


Figure 1: Layout of the traced objects.

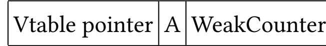


Figure 2: Metadata layout.

Figure 3: CounterMarker layout.

2.1 Structure of Traced Objects

The algorithm assumes that objects in memory have the header structure as depicted in Figure 1. The header contains the reference counter and the fields needed for cycle collection:

- Next and Prev implement lists/queue for cycle collection;
- MetadataPointer contains a pointer either directly to the structure collecting the operations working on the object, called "Vtable", or to a Metadata structure, depicted in Figure 2, used to manage weak pointers. The structure aggregates the Vtable pointer, the counter of weak references (WeakCounter), and the status bit A, which tracks whether it is possible to dereference the pointer to the object or not;
- CounterMarker, whose layout is depicted in Figure 3, holds the counters, marks, and status of the object.

Since it is particularly important for implementing the logic of SILB-Recycler, we detail the structure of the CounterMarker:

- MA is a two-bit field which can assume one of four statuses:
  - `NON_MARKED`: no list (or queue) contains the object;
  - `IN_POSSIBLE_CYCLES`: the object might be a cycle root;
  - `IN_LIST`: the object is in one of the lists;
  - `IN_QUEUE`: the object is in the breath-first traversal queue.
- Tracing counter is the counter of the strong pointers traced during collection. This counter stores the number of references to each object encountered during the traversal. Using a separate counter from the reference counter is necessary against faults, which could halt the collection at any time. As such, decrementing the reference counter is not a viable option since we could not restore it at faults. Using the tracing counter also benefits execution times, since we replace the expensive reference counter restoration by resetting the tracing counter once per managed object.
- ME is a one-bit field that holds the MetadataPointer status — when it points to the Vtable is 0 and 1 for the Metadata.
- FIN is a one-bit field that tracks the finalisation status — set to 1 after the execution of the finaliser;
- Reference counter is the counter of the total strong pointers.

Additionally, we use the CounterMarker to indicate whether the object has already been dropped, setting to the maximum value (all ones) the tracing counter — if the tracing counter is equal to its maximum value, then we already ran the destructor of the object.

2.2 Finalisation

Finalisers support the release (e.g., clean-up and state updates to enforce consistency) of resources that an object may hold and no longer needs, e.g., file handles, network/database connections. SILB-Recycler supports the execution of finalisers when an object's reference counter reaches 0 and before freeing identified garbage cycles.

Since finalisers can "resurrect" objects, before collection, SILB-Recycler checks that any collectible object did not return accessible.

When the reference counter has reached 0, SILB-Recycler verifies that the reference counter remained at 0 after finalisation, making it safe to remove the object. Otherwise, it places the object in the `POSSIBLE_CYCLES` list (the list of potentially cyclical garbage), since it can belong to a garbage loop. During cycle collection, SILB-Recycler needs to perform the identification of the cycles after a finalisation. In this case, the collector reinserts the traced objects within the `POSSIBLE_CYCLES` list and restarts the cycle identification procedure from the beginning. Since this mechanism can lead to non-termination, we set an upper bound on the number of executions of the cycle identification routine within the same collection session — the finaliser of each object can run at most once and the values allocated during the execution of a finaliser are not finalised.

2.3 Weak pointers

At an object's creation, MetadataPointer contains the direct pointer to the object's Vtable. When the runtime creates the first weak pointer to the object, SILB-Recycler creates the Metadata structure and sets the pointer to the object's Vtable therein, sets to 1 both the fields A and WeakCounter, sets the MetadataPointer to reference the new structure, and updates CounterMarker to reflect this change.

Besides pointing to the Metadata structure, the weak pointers also have a pointer to the object and accessing a weak pointer entails checking the accessibility status (cf. A in Figure 2) of the Metadata structure before dereferencing the pointer to the object. Since the metadata structure has no cycles, SILB-Recycler keeps it allocated until it freed the objects and the WeakCounter reached 0.

Separating an object from its Metadata allows SILB-Recycler to free the object's allocation irrespective of its weak pointers' status.

2.4 Pseudocode Procedures of SILB-Recycler

We detail (using pseudocode) the main procedures of SILB-Recycler.

For simplicity, the pseudcode presented in this section omits the parts of the algorithm related to managing faults, which make SILB-Recycler resilient to fatal errors. However, these are minimal and regard the calls to finalisers (finalize), tracing (trace), and destroyers. In all these cases, the algorithm empties all lists and queue (root, non-root, etc.) and unmarks the traced objects.

At an object's creation, SILB-Recycler sets it `NON_MARKED`.

When SILB-Recycler creates a strong pointer to an object, it removes the latter from `POSSIBLE_CYCLES` (if present), since it has a positive reference counter and cannot be in a garbage cycle.

When a strong pointer goes out of scope, SILB-Recycler follows Procedure 1, which decrements the reference counter and finalises and frees the object if the reference counter reached zero. More

Procedure 1 Strong pointer destructor.

```
1:  proc sp_destroy(sp: SP)
2:      if sp.mark = IN_LIST then
3:          sp.RC ← sp.RC − 1
4:      else if sp.RC > 1 then
5:          sp.RC ← sp.RC − 1
6:          add_to_possible_cycles(sp)
7:      else
8:          if finalisation_enabled() && is_to_finalise(sp) then
9:              set_finalised(sp)
10:             finalise(sp.value)
11:             if sp.RC > 1 then        ▷ The object was resuscitated
12:                 sp.RC ← sp.RC − 1
13:                 add_to_possible_cycles(sp)
14:                 return
15:         sp.RC ← sp.RC − 1
16:         ▷ Make sure sp is not in POSSIBLE_CYCLES ◁
17:         remove_from_possible_cycles(sp)
18:         set_dropped(sp)
19:         drop_object(sp.value)
20:         drop_metadata(sp)
21:         FREE(sp)
```

Procedure 2 Garbage cycles identification.

```
1:  proc identify_garbage_cycles()
2:      root_list ← new List
3:      non_root_list ← new List
4:      queue ← new Queue
5:      ▷ Trace counting ◁
6:      for all q ∈ {POSSIBLE_CYCLES, queue} do
7:          while q.is_not_empty() do
8:              sp ← q.remove_first()
9:              trace(sp.value)
10:             if sp.RC = sp.TC then
11:                 non_root_list.add(sp)
12:             else
13:                 root_list.add(sp)
14:     ▷ Trace roots ◁
15:     for all q ∈ {root_list, queue} do
16:         while q.is_not_empty() do
17:             sp ← q.remove_first()
18:             sp.mark ← NOT_MARKED
19:             trace(sp.value)
```

precisely, if the reference counter would reach 0, SILB-Recycler executes the object's finaliser before decrementing the counter, to soundly allow the creation of new possible references to the object during the finaliser's execution — SILB-Recycler performs the finalisation only if enabled and not already executed on the object (FIN), marked before its execution (line 9). After having run the finaliser, the collector frees the object if the reference counter is still 1, i.e., the finaliser did not "resurrect" the object. Otherwise, it follows the procedure for objects with a reference counter greater than one (described below). Before these steps, the procedure checks whether the cycle collector marked the object as a member of a garbage cycle (line 2). The check is necessary since the procedure drop_object within Procedure 5 destroys the strong pointers contained by members of garbage cycles, which would incorrectly free objects that the cycle collector is already handling. If the reference counter is greater than one, SILB-Recycler inserts the object in the `POSSIBLE_CYCLES` list, since it may be a garbage cycle root. When inserting objects in `POSSIBLE_CYCLES` (using add_to_possible_cycles), we set the object's marking to `IN_POSSIBLE_CYCLES` and reset its tracing counter — we avoid adding objects twice if already present.

When SILB-Recycler drops and frees the object, the object can be in the `POSSIBLE_CYCLES` list. In this case, the collector removes the object (using remove_from_possible_cycles) before dropping it.

Procedure 2 identifies the garbage cycles to collect. After having reset the necessary lists and queue, it performs the two tracing phases, *trace counting* and *trace roots*, which process each element at the head of the lists/queue until their exhaustion.

Trace counting starts the tracing from the `POSSIBLE_CYCLES` list, since it contains the candidate roots of garbage cycles. After the tracing of the references contained inside an object (line 9), depending on the values of the object's reference and tracing counters, the procedure inserts it within the `non_root_list` or `root_list`.

Slightly simpler, *trace roots* unmarks the allocation before tracing the object. The traversal starts from the elements of `root_list`, since it contains the externally reachable objects. This traversal leaves in `non_root_list` only the members of garbage cycles.

Although we omit error-handling code for brevity, we outline its rationale for Procedure 2, which concerns the main fault management logic of the algorithm. Briefly, if a fatal error occurs during either of the calls to `trace` (lines 9 and 19), we handle the error by emptying all the lists and the queue — `root_list`, `non_root_list` and `queue` — and unmarking every contained object before halting the collection. Moreover, if a fault occurs during *trace counting* (line 9), we also unmark the object being traced (referenced by `sp`).

The trace operation (cf. Procedure 3) in Procedure 2 encodes the tracing logic, which, for strong pointers, avoids tracing an object's fields and co-operates with Procedure 2 to implement the two breath-first heap traversals. The procedure differs depending on the tracing phase. During *trace counting*, if the object is marked

- `IN_POSSIBLE_CYCLES` or `IN_QUEUE`, Procedure 2 handles the tracing and insertion of the object in a list. As such, we need to only increment the tracing counter.
- `IN_LIST`, the object is either in `non_root_list` or `root_list`. We increment the tracing counter and, if the reference counter is equal to the latter, we move the object into `non_root_list` (Procedure 2 already handled it).
- `NON_MARKED`, the object has never been traced. Thus, the tracing counter must be reset and set to one (single assignment), followed by the queuing of the object for traversal.

During *trace roots*, we only move the object inside the queue if found in `non_root_list`, since `root_list`'s elements are already traced by Procedure 2. We know the traversal cannot visit untraced objects, since every traced object is in a list at the start of this phase. An object is inside `non_root_list` if its mark is `IN_LIST` and its reference and tracing counters are equal.

Procedure 3 Tracing procedure for strong pointers.

```
1: proc sp_trace(sp: SP)
2:   if is_trace_counting() then                          ▷ Trace counting
3:     if sp.mark ∈ {IN_POSSIBLE_CYCLES, IN_QUEUE} then
4:       sp.TC ← sp.TC + 1
5:     else if sp.mark = IN_LIST then
6:       sp.TC ← sp.TC + 1
7:       if sp.RC = sp.TC then
8:         root_list.remove(sp)
9:         non_root_list.add(sp)
10:    else                                              ▷ sp.mark = NON_MARKED
11:      sp.TC ← 1
12:      sp.mark ← IN_QUEUE
13:      queue.add(sp)
14:  else                                                ▷ Trace roots
15:    if sp.mark = IN_LIST && sp.RC = sp.TC then
16:      non_root_list.remove(sp)
17:      sp.mark ← IN_QUEUE
18:      queue.add(sp)
```

Procedure 4 performs cycle collection. Without finalisation, it only identifies the cycles and frees them, done by Procedure 2 and Procedure 5. With finalisation, it re-runs the cycle identification after having executed the finalisers. To avoid divergence, it executes the cycle collection up to a maximum of 10 times — avoiding pathological non-termination due to cycles of "resurrection".

After having identified the objects in garbage cycles, we run their finalisers (those yet to run, lines 5–10 of Procedure 4). If we execute at least one finaliser, then we re-insert the objects in `POSSIBLE_CYCLES` and the loop continues (lines 13–14). Otherwise, we free the identified garbage cycles using Procedure 5. Note the insertion of the finalised object in `POSSIBLE_CYCLES` at line 14; necessary for resuming at the next cycle collection if the loop reached its maximum executions.

Procedure 5 frees the identified garbage cycles. Since objects can access each other at destruction, we call all the destructors before freeing memory. Hence, this procedure performs two iterations over `non_root_list`. In the first, it sets to "dropped" the status of the objects and calls drop_object to execute their destructor — followed by recursively calling the procedure on each field of the destructed object. In the second iteration, we free the metadata structure or mark it inaccessible, freeing the objects — if an object has the metadata structure, we can free it if there are no weak pointers referencing it; otherwise, we set the accessible flag to false, preventing access via those weak pointers.

SILB-Recycler upgrades weak to strong pointers, returning `None` otherwise, depending on four conjunctive conditions:

- the metadata structure is accessible (the object is allocated);
- the reference counter is greater than 0;
- the object has not been dropped — it would be unsafe to provide a pointer to a destroyed object;
- the object is not inside `non_root_list` when the collector is executing Procedure 5 (otherwise the object is part of a garbage cycle and the collector is going to destroy it).

Procedure 4 Start cycle collection.

```
1: proc collect_cycles()
2:   if finalisation_enabled() then
3:     loop 10 times                                     ▷ Avoid non-termination
4:       identify_garbage_cycles()
5:       has_finalised ← false
6:       for all sp ∈ non_root_list do
7:         if is_to_finalise(sp) then
8:           has_finalised ← true
9:           set_finalised(sp)
10:          finalise(sp.value)
11:      if has_finalised then
12:        ▷ We finalised some values, we recheck the cycles ◁
13:        for all sp ∈ non_root_list do
14:          add_to_possible_cycles(sp)
15:      else
16:        drop_non_root_list()
17:        return                  ▷ Garbage cycles have been collected
18:  else
19:    identify_garbage_cycles()
20:    drop_non_root_list()
```

Procedure 5 Free garbage cycles.

```
1: proc drop_non_root_list()
2:   for all sp ∈ non_root_list do
3:     set_dropped(sp)
4:     drop_object(sp.value)
5:   for all sp ∈ non_root_list do
6:     drop_metadata(sp)
7:     FREE(sp)
```

A weak pointer that goes out of scope decrements the weak counter and frees the metadata if the counter reached 0 and the structure is not accessible (i.e., we already freed the object).

3 A Rust Safe Interface SILB-Recycler Library

We implement SILB-Recycler as a Rust [26] library, called `rust-cc`, released on `crates.io` at https://crates.io/crates/rust-cc under either the MIT or Apache-2.0 licences, source code available at https://github.com/frengor/rust-cc.

We design `rust-cc` with flexibility and safety in mind and take inspiration for its public API from Rust standard library's reference-counted pointers to provide a familiar and idiomatic experience for Rust developers and to help them integrate it within Rust's ecosystem. Notably, `rust-cc` makes minimal usage of `unsafe`[1] in its API and safe code can use all its features. Thanks to Rust's type system and borrow checker, the compiler can check and prevent any undefined behaviour encountered when using `rust-cc`. The integration within Cargo [25] — Rust's package manager — allows the library to parametrise all its features, which developers can enable when needed, possibly reducing compile times and improving performance (cf. benchmarks with(out) finalisers in Section 4.2).

[1] A scoping mechanism that disables the compilers' static checks, e.g., useful when dealing with low-level memory-management logic that would be too complex/inefficient to implement so that the compiler can successfully perform its static checks.

```
unsafe trait Trace: Finalize {
    fn trace( &self, ctx: &mut Context<'_> );
}
trait Finalize {
    fn finalize( &self ) {}
}
```

Listing 1: Mandatory traits for cycle-collectable types.

rust-cc exposes the Cc<T> smart pointer (i.e., a pointer which manages the memory it references), which is a single-threaded reference-counted strong pointer collected using SILB-Recycler. Specifically, Cc<T> has a similar API to the reference-counted pointer Rc from Rust's standard library. The associated function new allocates and returns the first strong pointer to a new cycle-collected value, while the clone method increments the reference counter and returns a new strong pointer to an already-existing value. Cc also supports the dereferencing operator (*), which returns a compile-time checked reference to the pointed value.

Users can obtain weak pointers using the downgrade method, which they can convert back to Ccs using the upgrade method.

rust-cc also supports cleaners, i.e., closures that execute clean-up code when the runtime drops their enclosing structure. Specifically, the user can define a Cleaner field of a cycle-collected type and register clean-up closures executed at the field's destruction. Users can directly execute registered clean-up actions, but these invocations prevent the closures from running at their Cleaner's drop — rust-cc allows only one execution of any clean-up action.

Thanks to the design of SILB-Recycler, it is always safe for the methods related to collection (trace, finalize, drop) of cycle-collectable values to panic, i.e., to raise errors.

3.1 Mandatory Traits of Cycle-collectable Types

Rust traits are interfaces one can implement in a generic way, allowing the definition of shared functionalities between types.

To use a type with rust-cc, i.e., to make it cycle-collectable, it must implement the Trace and Finalize traits, shown in Listing 1.

Finalize allows the finalisation of cycle-collected values. Since it is an option, even though the developer can disable the usage of finalisers, they have to implement the Finalize trait to allow compatibility with libraries that have it enabled.

Trace enables tracing the type's fields, performed by calling the trace method on the fields of the type. Since it is not possible to express every requirement of the Trace trait in the Rust type system — we would need to specify *behaviour* rather than structure, e.g., tracing Ccs not owned by self — we mark it unsafe. Thus, implementations must adhere to the following safety requirements to avoid undefined behaviour:

(1) trace every Cc instance *exclusively* owned by self. No other Cc instance can be traced;[2]
(2) during the same tracing phase, two trace calls on the same value must *behave the same*, i.e., they must trace the same Cc instances.[3] If a panic happens during the second trace call, then the Cc instances traced during the second call must be a subset of the Cc instances traced in the first one.
(3) trace must not create, clone, move, dereference or drop Ccs.
(4) destructors in cycle-collectable types cannot create, clone, move, dereference, drop or call methods on any Cc instance[4].

To allow safe code to implement the Trace trait, rust-cc provides a homonymous derive macro. The macro emits an implementation which traces every type's field not marked with the attribute #[rust_cc(ignore)]. Using the attribute is safe and useful when some fields are not traceable, e.g., when deriving Trace for a type which contains fields like Cell — from Rust's standard library — that does not implement Trace. Ignoring a field may leak memory if the field contains a Cc, but it never leads to undefined behaviour.

To respect the destructors' safety requirement (4), the derive macro emits an empty destructor implementation, which always satisfies the safety requirements and prevents the user from implementing a potentially wrong custom destructor. Experienced users can unsafely implement a custom destructor by applying the attribute #[rust_cc(unsafe_no_drop)] to the type definition, which suppresses the emission of the destructor implementation.

rust-cc also provides a derive macro for Finalize, which emits an empty implementation, useful when finalisers are unnecessary.

Implementation Details. rust-cc triggers collections when allocating new values and determined by a *threshold* (over the allocated bytes) and an *adjustment percentage*.

When the allocated bytes exceed the threshold, rust-cc starts a new collection. If the number of allocated bytes still exceeds the threshold after the collection, we double the threshold to adjust the triggering to the size of the allocated data. Otherwise, rust-cc regulates the threshold by checking whether its value multiplied by the adjustment percentage is greater than the allocated bytes. If it is the case, then rust-cc halves the value of the threshold until the multiplied value is lower than that of the allocated data.

The cycle collector in rust-cc seamlessly supports cleaners (no special cases) thanks to ignored fields. Indeed, the Trace implementation of Cleaner is empty, avoiding tracing the Ccs captured by the registered clean-up closures. Hence, the pointed objects are alive and not collected until after the execution of all the cleaners.

The rust-cc test suite includes 80+ thorough tests. Its continuous integration infrastructure checks every commit using GitHub Actions, which run the test suite using the tool cargo-hack [11]. This tool executes all tests for each combination of rust-cc features, enabling comprehensive checks — compared to the complexity of devising their combination by hand. Furthermore, since rust-cc uses unsafe code, we run the test suite with a detector of undefined behaviour for Rust programs (Miri [10]).

[2]Tracing can happen at most once to support ignoring some fields. In Rust, every value has a unique owner — a variable or another value — which binds the former's lifetime to the latter's. We impose exclusive ownership because some types provide "shared ownership", binding the value's lifetime to that of the last "standing" owner. By requiring exclusive ownership over the traced Ccs, we disallow tracing Cc instances behind references (&, &mut) or with shared ownership (e.g., Rc<Cc<...>>).

[3]Calls to the trace method happen during *tracing phases*. To detect if successive trace calls happen within the same phase, we use a function which returns false when traces belong to different phases (spec. to calls on different values). Referring to the pseudocode shown in Section 2.4, two trace calls can happen during the same *tracing phase* if called during the same execution of Procedure 2. This requirement ensures that the traced objects do not change between the *trace counting* and *trace roots* phases.

[4]Since the members of garbage cycles can access each other during the execution of their destructors, it is not safe to access Ccs, as they could point to dropped objects.

4 Evaluation

We evaluate our SILB-Recycler implementation both qualitatively and quantitatively; respectively, we compare `rust-cc`'s design/features against its main competitors in the Rust ecosystem, and we report the latter's and `rust-cc`'s performance under 4 benchmarks.

4.1 Qualitative Evaluation

We gather the main single-threaded garbage collectors in the Rust ecosystem, considering the most all-time downloaded ones at the time of writing (reported in brackets) from crates.io: `gc` [14] (219k), `bacon-rajan-cc` [12] (10k), `broom` [5] (7K), and `safe-gc` [13] (1k). Notably, none of the alternatives supports cleaners.

`gc` is one of the most used garbage collectors in the Rust ecosystem. It is a mark-and-sweep garbage collector, and it presents an API similar to that of `rust-cc`, but with some important limitations:

(1) it is not possible to use the `RefCell` type — the main one for interior mutability — within collectable types, but it is necessary to use the `GcCell` type instead, limiting developer's options in designing their data structures and reducing compatibility with the rest of the Rust ecosystem;
(2) excluding a field's tracking is unsafe;
(3) no support for weak pointers;
(4) while the use of the derive macro is close to `rust-cc`, the trait `Trace` is more complex to implement manually.

`bacon-rajan-cc` is a cycle collector that closely implements the algorithm presented by Bacon and Rajan [4], hence the name. It presents an API similar to that of `rust-cc`, but it does not support finalisation and automatic collection. Moreover, the library is unsound, as the trait `Trace` is not marked as unsafe to implement, leading to possible undefined behaviour caused by safe code.

`safe-gc` is a library that implements a garbage collected arena using a mark-and-sweep algorithm, it includes no `unsafe` code, and uses data structures and smart pointers from the standard library. The API/feature-set of `safe-gc` is quite different w.r.t. `rust-cc`:

(1) allocations happen via the `alloc` method of a `Heap` instance, which implements an arena of garbage collected objects of possibly heterogeneous types;
(2) there are two types of pointers, a root and a non-root garbage collected one, and their correct usage is the users's responsibility (incorrect use never leads to undefined behaviour);
(3) users cannot directly dereference pointers, but they need to pass the pointer to `Heap`'s `get` and `get_mut` methods;
(4) no support for finalisation and weak pointers.

Despite its strong safety guarantees, `safe-gc` requires more explicit management, adding complexity/cognitive overhead. The inability to directly dereference pointers (accessed using special methods) makes the API less ergonomic and potentially slower.

`broom` is a library that implements a garbage collected arena using a mark-and-sweep algorithm similar to `safe-gc`, but with the fundamental difference that objects allocated within the arena can be of only one type, which greatly reduces its flexibility.

4.2 Quantitative Evaluation

To compare the performance of `rust-cc` and the considered alternatives, we devise 4 benchmarks: *stress test*, *binary trees*, *binary trees with parent pointers* and *large linked list*. We derive and adapt the first three benchmarks from the `shredder` [23] library and provide the source code of the tests at https://github.com/frengor/rust-cc-benchmarks.

We run all benchmarks under Linux Mint (kernel version 5.4.0) on a machine equipped with a Ryzen 9 5900X 12 core 24 threads CPU and 32 GB of RAM, using the `cset-shield` command to exclusively allocate a core for each benchmark. We implement the tests using resp. `rust-cc` 0.6.1, `gc` 0.5.0, `bacon-rajan-cc` 0.4.0, `safe-gc` 1.1.1, and `broom` 0.3.2 and compile them under Rust 1.81.0. To provide context and further insights on the results, we also implement and report on the *binary trees*, *binary trees with parent pointers*, and *large linked list* benchmarks using the `Rc` and `Arc` pointers from Rust's standard library (which do not support automatic cycle collection).

We collect the measurements using the `criterion.rs` library [1] with the parameters code `-C opt-level=3 codegen-units=1 lto=thin`, i.e., enabling all compilation optimisations, reducing parallel compilation (high parallel compilation may produce slower code), and enabling link-time optimisations.

We measure 100 samples for each pair benchmark-alternative where each (`Criterion.rs`) sample consists of many iterations of the benchmark. The number of iterations derives from a set recording timeout and the time taken by each benchmark instance (executed after a warm-up period), e.g., `rust-cc` totals 800 iterations under *stress test*. We present the results of each benchmark both in tabular form and as violin plots, considering `rust-cc` with and without finalisation to fairly compare it with libraries that do not support the option.

Library	Avg Time	St. Dev.	St. Dev. %	Δ_t%
bacon-rajan-cc	20.45 ms	±0.24	1.15%	+29.73%
broom	64.65 ms	±0.25	0.39%	+310.14%
gc	53.93 ms	±0.29	0.54%	+242.13%
rust-cc	15.76 ms	±0.15	0.96%	-
rust-cc, no fin.	15.68 ms	±0.15	0.93%	−0.51%
safe-gc	70.97 ms	±0.29	0.41%	+350.26%

Table 1: *Stress test* benchmark.

4.3 Stress test benchmark

The *stress test* benchmark creates a directed graph with $2^{15} + 1$ vertices and $2^{15} + 1$ random edges, maintaining a strong pointer to each vertex, which are then gradually destroyed by performing collections at regular intervals.

We report in Table 1 the average execution time, the standard deviation (absolute and in percentage) and the percentage time delta between `rust-cc` and the alternatives. We visualise the data in Figure 4, where the x-axis tracks the time in milliseconds, and we distribute on different lines of the y-axis the candidates for clarity.

From Table 1 and Figure 4, both `rust-cc` and `bacon-rajan-cc` are approximately 3 times faster than the alternatives, with `rust-cc` being a few milliseconds quicker. The performance difference with other competitors is likely due to the ability of both cycle collectors to avoid tracing the entire heap during each collection. As expected, disabling finalisation makes `rust-cc` (slightly) faster.

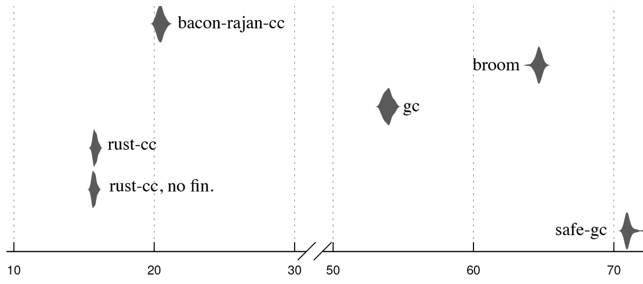


Figure 4: Violin plot of *stress test* (x-axis in ms).

Library	Avg Time	St. Dev.	St. Dev. %	Δ_t %
arc	4.12 ms	±0.01	0.18%	−18.53%
bacon-rajan-cc	4.47 ms	±0.00	0.03%	−11.69%
broom	25.10 ms	±0.51	2.03%	+396.02%
gc	12.52 ms	±0.02	0.13%	+147.37%
rc	4.08 ms	±0.00	0.04%	−19.46%
rust-cc	5.06 ms	±0.00	0.01%	-
rust-cc, no fin.	4.85 ms	±0.00	0.02%	−4.11%
safe-gc	13.56 ms	±0.02	0.15%	+167.95%

Table 2: *Binary trees* benchmark.

4.4 Binary trees

The *binary trees* benchmark measures the efficiency and impact of reference counting in the absence of cycles. In the benchmark, we create and destroy complete binary trees, with no parent pointers and a maximum height of 10. We use the same representation for *stress test* for both the tabular data in Table 2 and the visualisation in Figure 5. The main observation we report is that algorithms using reference counting (note the presence of `rc` and `arc`) are faster (and perform similarly) than the alternatives. Notably, `bacon-rajan-cc` is slightly faster than `rust-cc`. We expected this behaviour since `bacon-rajan-cc` does not perform automatic collections, making only use of reference counting during the benchmark execution and saving time compared to `rust-cc`. Indeed, disabling automatic collection, `rust-cc` becomes slightly faster than `bacon-rajan-cc`.

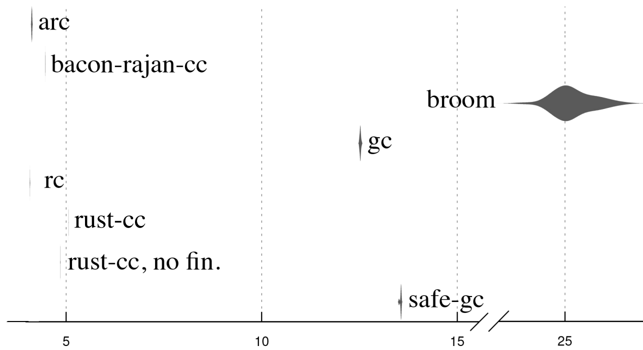


Figure 5: Violin plot of *binary trees.* (x-axis in ms).

Library	Avg. Time	St. Dev.	St. Dev. %	Δ_t%
arc	4.82 ms	±0.01	0.11%	−67.36%
bacon-rajan-cc	22.01 ms	±0.07	0.34%	+49.10%
broom	28.69 ms	±0.74	2.57%	+94.36%
gc	14.88 ms	±0.02	0.16%	+0.80%
rc	4.56 ms	±0.02	0.41%	−69.11%
rust-cc	14.76 ms	±0.01	0.08%	
rust-cc, no fin.	10.70 ms	±0.01	0.07%	−27.54%
safe-gc	18.96 ms	±0.03	0.15%	+28.42%

Table 3: *Binary trees with parent pointers* benchmark.

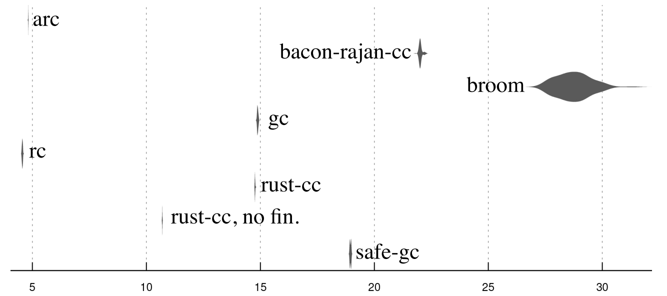


Figure 6: Violin plot of *binary trees with parent pointers* (x-axis in ms).

4.5 Binary trees with parent pointers

This benchmark builds on *binary trees* by adding parent pointers to create reference cycles. Specifically, each internal node of the generated trees participates in three cycles, two with the children and one with the parent node. All internal pointers in the trees are strong pointers, except in the benchmarks for `Rc` and `Arc`, which use weak pointers to the parent nodes because, otherwise, they would not collect the reference cycles. Following the layout of previous benchmark, we report the data in Table 3 and visualise it in Figure 6.

From the results, the alternatives using reference counting are the fastest, with `rc` and `arc` on top. Next, we find `rust-cc` and `gc`, both taking ∼15 ms. Contrarily to the *binary trees* benchmark, `bacon-rajan-cc` is substantially slower than `rust-cc` (49%), empirically demonstrating the performance advantage provided by our proposal. Notably, disabling finalisation further improves the performance of `rust-cc` by 28%, making it the closest to `Rc`/`Arc`.

4.6 Large linked list

This benchmark creates several doubly linked lists, each containing 4096 elements, and collects them. As with the *binary trees with parent pointers* benchmark, we only use strong pointers, except for `Rc` and `Arc`, which use weak pointers for the pointers to the previous nodes. Following the previous benchmarks' layouts, we respectively report in Table 4 and Figure 7 the data and its visualisation.

Also in this case, `Rc` and `Arc` are the fastest, followed by `gc`, which is 12% faster than `rust-cc`. We attribute the good performance of `gc` for this case to the distinctive feature of the benchmark, i.e., that all the linked lists have the same dimension, while the structures of the tree-based benchmarks have different sizes. Moreover, also

Library	Avg. Time	St. Dev.	St. Dev. %	Δ_t%
arc	3.33 ms	±0.00	0.06%	−55.76%
bacon-rajan-cc	14.42 ms	±0.06	0.45%	+91.31%
broom	24.10 ms	±0.70	2.91%	+219.78%
gc	6.65 ms	±0.01	0.12%	−11.82%
rc	2.48 ms	±0.01	0.38%	−67.05%
rust-cc	7.54 ms	±0.00	0.06%	
rust-cc, no fin.	5.31 ms	±0.01	0.20%	−29.54%
safe-gc	13.40 ms	±0.00	0.02%	+77.76%

Table 4: Results of `large linked list`.

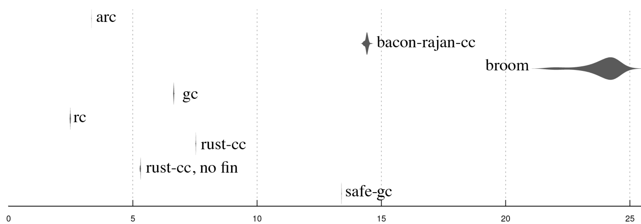


Figure 7: Violin plot of *large linked list* (x-axis in ms).

the number of lists created and destroyed in the case is the same, while in the tree-based ones the number changes depending on the structures' size. Considering that gc performs recursive rather than queue-based tracing, we conjecture that `rust-cc` pays a cost in moving allocations in the queue before tracing. We deem this point interesting to explore in future refinements of our approach and implementation. We confirm the observations made about `rust-cc` and `bacon-rajan-cc` with *binary trees with parent pointers*, where the latter is substantially slower than `rust-cc` (here, 91%) — disabling finalisation improves `rust-cc`'s performance by ca. 30%.

5 Related Work

Reference counting is an efficient garbage collection method, but it struggles with cyclic data structures. In the last 3 decades, several proposals refined the basic technique to overcome the issue.

Bobrow and Christopher [7, 8] presented two of the first proposals of a garbage collection algorithm able to collect circularly linked inaccessible structures. In particular, Christopher's algorithm requires no additional information beyond that required by a reference count scheme and the garbage collector does not have to find pointers outside the heap. Roy et al. [24] applied cycle-collection reference counting to the problem of garbage collection in object-oriented databases. Their algorithm keeps track of auxiliary reference count information to detect and collect cyclic garbage, working correctly also in the presence of concurrently running transactions and system failures. Lins [18] proposed an algorithm for cyclic reference counting in garbage collection that addresses inefficiencies of previous methods in dealing with cyclic data structures, particularly in object-oriented languages where sharing and cyclic structures are common. The main innovation is the introduction of a queue system that delays the mark-scan phase, allowing for more efficient memory management. Bacon and Rajan [4] introduced the algorithm that inspired SILB-Recycler, which can perform concurrent cycle collection operating without global searches. Lin and Hou [17] presented a "lightweight" cyclic reference counting algorithm based on partial tracing and considering a single sub-graph, instead of individual cycles, as the basic unit of cycle collection. Widemann [28] proposed a reference-counting garbage collector designed for functional programming languages that balances and mitigates the complexity of maintaining a subset of marked edges (ensuring that every cycle contains at least one marked edge) and the inefficiencies of local mark-and-scan procedures for cycle detection.

SILB-Recycler distinguishes itself from these proposals by being resilient to fatal errors during the tracing phase, supporting object finalisation, not needing supplementary heap memory during collection, and applying a fast breadth-first approach for tracing.

Looking at the `rust-cc` implementation, the technique of emitting an empty destructor implementation from the `Trace` derive macro was first used by the gc library. The library also inspired the `Finalize` trait signature (and derive macro) and the threshold used to trigger automatic collections, even though it lacks the adjustment percentage. The `Trace` signature is similar to the one of `bacon-rajan-cc`, however the library's equivalent of the `Context` type (cf. Listing 1) is different from the one `rust-cc` uses. Cleaners have been originally introduced in Java [20] but are a common and pattern-specific concept in Rust programming.

6 Conclusion

We present SILB-Recycler, a novel cycle collection reference-counting algorithm resilient to fatal errors during the tracing phase, supporting object finalisation and weak pointers, fast breadth-first tracing approach that avoids stack overflows and the need for supplementary heap memory during collection.

We implement SILB-Recycler as a Rust [26] library, `rust-cc`, presenting its API and performing both a qualitative and a quantitative comparison with its main alternatives in the Rust ecosystem. Qualitatively, the library APIs follow those of Rust's standard library, achieving both familiarity for Rust developers and seamless integration within the Rust ecosystem — additionally, the library makes minimal use of `unsafe` code and leverages Rust's type system and borrow checker to prevent undefined behaviour. Quantitatively, `rust-cc` is one of the most efficient alternatives, in particular it is the fastest option when dealing with garbage cycles.

Looking at future steps, we see potential to improve our proposal by integrating generational [16, 27] or age oriented [22] techniques to further reduce overhead and obtain better throughput. Another interesting direction is the integration of sliding-views [2, 15, 21] to obtain on-the-fly, concurrent cycle collection.

Acknowledgments

Research partly supported by project PNRR CN HPC - SPOKE 9 - Innovation Grant LEONARDO - TASI - RTMER funded by the NextGenerationEU European initiative through the MUR, Italy (CUP: J33C22001170001)

References

[1] Aparicio, Jorge. 2017. Criterion.rs. https://crates.io/crates/criterion. Accessed: 19-06-2024.

[2] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. 2003. An on-the-fly mark and sweep garbage collector based on sliding views. *SIGPLAN Not.* 38, 11 (oct 2003), 269–281. https://doi.org/10.1145/949343.949329

[3] Hezi Azatchi and Erez Petrank. 2003. Integrating Generations with Advanced Reference Counting Garbage Collectors. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2622)*, Görel Hedin (Ed.). Springer, 185–199. https://doi.org/10.1007/3-540-36579-6_14

[4] David F. Bacon and V. T. Rajan. 2001. Concurrent Cycle Collection in Reference Counted Systems. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2072)*, Jørgen Lindskov Knudsen (Ed.). Springer, 207–235. https://doi.org/10.1007/3-540-45337-7_12

[5] Barretto, Joshua. 2020. Broom. https://crates.io/crates/broom. Accessed: 20-06-2024.

[6] Stephen M. Blackburn and Kathryn S. McKinley. 2003. Ulterior reference counting: fast garbage collection without a long wait. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, Ron Crocker and Guy L. Steele Jr. (Eds.). ACM, 344–358. https://doi.org/10.1145/949305.949336

[7] Daniel G. Bobrow. 1980. Managing Reentrant Structures Using Reference Counts. *ACM Trans. Program. Lang. Syst.* 2, 3 (July 1980), 269–273. https://doi.org/10.1145/357103.357104

[8] Thomas W. Christopher. 1984. Reference Count Garbage Collection. *Softw. Pract. Exp.* 14, 6 (1984), 503–507. https://doi.org/10.1002/SPE.4380140602

[9] George E. Collins. 1960. A method for overlapping and erasure of lists. *Commun. ACM* 3, 12 (1960), 655–657. https://doi.org/10.1145/367487.367501

[10] The Rust Project Developers. 2016. Miri. https://github.com/rust-lang/miri. Accessed: 29-05-2024.

[11] Taiki Endo. 2019. cargo-hack. https://crates.io/crates/cargo-hack. Accessed: 29-05-2024.

[12] Fitzgerald, Nick. 2015. bacon-rajan-cc. https://crates.io/crates/bacon-rajan-cc. Accessed: 20-06-2024.

[13] Fitzgerald, Nick. 2024. safe-gc. https://crates.io/crates/safe-gc. Accessed: 20-06-2024.

[14] Goregaokar, Manish. 2015. gc. https://crates.io/crates/gc. Accessed: 20-06-2024.

[15] Yossi Levanoni and Erez Petrank. 2006. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.* 28, 1 (jan 2006), 1–69. https://doi.org/10.1145/1111596.1111597

[16] Henry Lieberman and Carl Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM* 26, 6 (jun 1983), 419–429. https://doi.org/10.1145/358141.358147

[17] Chin-Yang Lin and Ting-Wei Hou. 2007. A simple and efficient algorithm for cycle collection. *ACM SIGPLAN Notices* 42, 3 (2007), 7–13. https://doi.org/10.1145/1273039.1273041

[18] Rafael Dueire Lins. 1992. Cyclic Reference Counting with Lazy Mark-Scan. *Inf. Process. Lett.* 44, 4 (1992), 215–220. https://doi.org/10.1016/0020-0190(92)90088-D

[19] R. M. Muthukumar and D. Janakiram. 2006. Yama: A Scalable Generational Garbage Collector for Java in Multiprocessor Systems. *IEEE Trans. Parallel Distributed Syst.* 17, 2 (2006), 148–159. https://doi.org/10.1109/TPDS.2006.28

[20] Oracle and/or its affiliates. 2017. Java Platform, Standard Edition, Version 9 API Specification - Cleaner. https://docs.oracle.com/javase%2F9%2Fdocs%2Fapi%2F%2F/java/lang/ref/Cleaner.html. Accessed: 20-06-2024.

[21] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. 2007. An efficient on-the-fly cycle collection. *ACM Trans. Program. Lang. Syst.* 29, 4 (aug 2007), 20–es. https://doi.org/10.1145/1255450.1255453

[22] Harel Paz, Erez Petrank, and Stephen M. Blackburn. 2005. Age-Oriented Concurrent Garbage Collection. In *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3443)*, Rastislav Bodík (Ed.). Springer, 121–136. https://doi.org/10.1007/978-3-540-31985-6_9

[23] Peach, Gregor. 2020. shredder. https://crates.io/crates/shredder. Accessed: 19-06-2024.

[24] Prasan Roy, S. Seshadri, Abraham Silberschatz, S. Sudarshan, and Srinivas Ashwin. 1998. Garbage Collection in Object-Oriented Databases Using Transactional Cyclic Reference Counting. *VLDB J.* 7, 3 (1998), 179–193. https://doi.org/10.1007/S007780050062

[25] The Rust Foundation. 2015. Cargo. https://doc.rust-lang.org/cargo/index.html. Accessed: 18-09-2024.

[26] The Rust Foundation. 2015. Rust. https://www.rust-lang.org/. Accessed: 21-06-2024.

[27] David Ungar. 1984. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. Association for Computing Machinery, New York, NY, USA, 157–167. https://doi.org/10.1145/800020.808261

[28] Baltasar Trancón y Widemann. 2008. A reference-counting garbage collection algorithmfor cyclical functional programming. In *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*, Richard E. Jones and Stephen M. Blackburn (Eds.). ACM, 71–80. https://doi.org/10.1145/1375634.1375645