On synchronising Garbage Collection across language boundary

Florian Freitag

Abstract

Garbage collection is broadly used by programming languages to find unused memory and deallocate it. This automatic memory management is well researched but becomes more complex if multiple languages are involved. This paper compares two works of research in which synchronization is used to fix edge cases in garbage collection across language boundary.

Introduction

Since the invention of automatic memory management (also known as garbage collection) in the 1960s by McCarthy[1] a lot of research went into improving and advancing this technology. For example, the infamous *Garbage Collection Handbook* builds on a corpus of over 2500 publications [2].

Today, most programming language runtimes use some kind of automatic memory management [2]. While this improves developer comfort, it increases the complexity of integrating languages with each other. Moreover, cyclic references across the runtime border can cause leaks, if the garbage collectors (*GCs*) are not properly synchronized [3].

Modern garbage collectors either use tracing, reference counting or a hybrid solution of both [2]. Synchronizing both approaches into a single collector might be necessary to when integrating with foreign APIs [4].

This paper compares two approaches of synchronization of garbage collection. In *Collecting Cyclic Garbage across Foreign Function Interfaces* the authors synchronize two garbage collectors to avoid leaks and improve performance [3]. While in *Efficient Cycle Detection on a Partially Reference Counted Heap* the author introduces a novel algorithm to extend a tracing GC to interact with reference counted objects [4]. Finally, the paper discusses the possibility of combining both approaches in future papers.

Background

Garbage collection

All data that no longer can be accessed is considered garbage. In most languages this means there is no variable pointing to that object and no other accessible object has a reference to it.

```
1 def getPaul():
2   food = Food("schnitzel")
3   city = City("VIE")
4   return Person("Paul", city)
5
6 def main():
7   paul = getPaul()
8   paulClone = paul
```

Listing 1: Creation of unreachable data.

In Listing 1 the function `getPaul` creates a `Person` object to which two variables point. Even though no variable points to the `City` object created in line 3 it is reachable by either dereferencing `paul.city` or `paulClone.city`. However, the `Food` object created in line 2 can no longer be reached and is therefore considered garbage.

It is the task of the garbage collector to eventually detect this and deallocate the object to avoid that the program uses unnecessary resources.

When and how the collector does this is a trade-off between throughput, pause time and space and different strategies might be advantageous for different use cases [2].

Tracing collection

The first garbage collection algorithm developed was the tracing garbage collector [1]. This one is also often called a *Mark and Sweep* collector, for the two stages that make them up.

In the marking phase the collector starts at the *roots* (which commonly are all variables that are currently in scope) and marks the objects which they reference. Next it recursively marks all object which can be reached from already marked objects.

In the sweeping phase the collector traverses all objects that were allocated and frees all that aren't marked, as they are no longer reachable.

This is the simplest form of a tracing garbage collector and many more advanced variants exists which might use concurrency, generations, partitioning, compacting and other techniques to improve performance [2]. However, they all have in common that they (at least partially) traverse (trace) the *reference graph*.

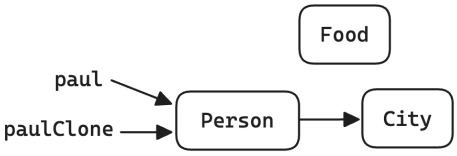

Figure 1: Reference graph for Listing 1.

After line 8 in Listing 1 the reference graph looks like in Figure 1. If a collection would be triggered at that point the tracing would start roots which in this case are just the two variables in scope. From there it would mark first the `Person` and then the `City` object and therefore deallocating the unmarked `Food` object in the sweeping phase.

The advantage of correctly implemented tracing collectors is their soundness (they will never free used objects) and completeness (they will find all garbage). However, they introduce some pause time on the program, therefore reducing throughput and require more space since it might be a while till the next collection is triggered.

Reference counting

Reference counting addresses some of the shortcomings of tracing garbage collectors [5]. Here, each object holds their own reference count which stores how many other objects or variables point to it. Once the reference count reaches zero it can no longer be reached and can therefore be safely deallocated.

Data in those systems is therefore collected as soon as it becomes garbage, reducing the memory load. In general, this also removes the need for pauses on the program (often called the *mutator*) as there is no need to trace the reference graph.

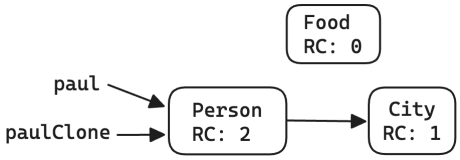

Figure 2: Reference counters for Listing 1.

Figure 2 depicts the state off all reference counters after line 8 has run. Of course, at that point the `Food` object would no longer exist as it has already been freed.

While reference counters solve some problems from tracing collectors, they come with their own set of limitations. The most significant one is that they are not complete and leak cyclic garbage [2].

```
1 def friends():
2   a = Person()
3   b = Dog()
4   a.friend = b
5   b.friend = a
```

Listing 2: Creation of unreachable object

For example, consider the function in Listing 2. In line 2 the `Person` object has a reference count of 1 as the variable `a` points to it. Similarly line 3 initializes `Dog` with one reference. After the lines 4 and 5 both objects have a reference count of 2 as they point to each other and each have a variable. Now, once the function returns, both variables are no longer in scope and therefore the reference count of both objects are set to 1.

However, both objects should be considered garbage since they are no longer reachable, but because of the cyclic references they hold each other alive.

This is a known problem and there are a couple of solutions [2]. For example, some languages push the problem to the programmer which is instructed to use *weak references* in cyclic data structures, which don't increase the reference counter of the object they are pointing to. Some other implementations combine reference counting with occasional tracing collection. However, the most common solution is *trial deletion* [2].

Trial deletion

Trial deletion is a partial tracing algorithm which does not need to traverse the complete reference graph but only a sub graph where garbage is suspected.

There are two laws which trial deletion exploits [2]. First, in any cyclic garbage structure all references to any object must be from other objects inside the structure. Second, garbage cycles can only be created from a pointer deletion that leaves a reference count greater than zero (because otherwise the object would already be deallocated).

Trial deletion starts by temporary decrementing the reference counter of one object in the suspected garbage structure. Next, it recursively traverses all reachable objects and decrements their reference count accordingly. If at the end all objects in the structure have a reference count of zero, the whole structure can be deallocated. Otherwise, there must be an external reference to the structure and all temporary reference count changes will be reversed.

Foreign function interface

It is quite common that a language needs to interact with another. For example, many languages have a C based foreign function interface (FFI) since some operating systems only expose system APIs as C code (Windows, macOS).

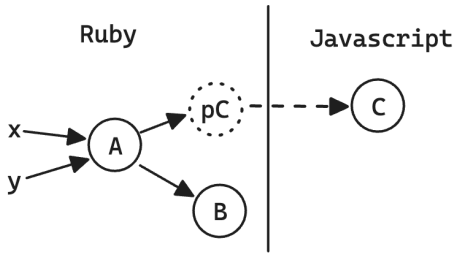

Figure 3: Proxy object for FFI.

If both languages share a heap, they can directly access all fields on the objects and call methods on *remote objects*. If they don't share a heap it might make sense to introduce *proxy objects*. A proxy object hides the communication with the runtime in which the remote object lives. Therefore, accessing a field on the proxy object will send a message to the foreign runtime, access the field on the remote object and return its value [3]. Figure 3 shows such a proxy object `pC` for `C`.

In runtimes with tracing collectors, foreign function interfaces become even more complex because neither collector has access to the complete reference graph. For example, Figure 3 wouldn't work as intended. While the Ruby runtime knows that `C` is still reachable via `x` and `y`, the Javascript runtime doesn't know that and would wrongly deallocate the object making the garbage collection unsound.

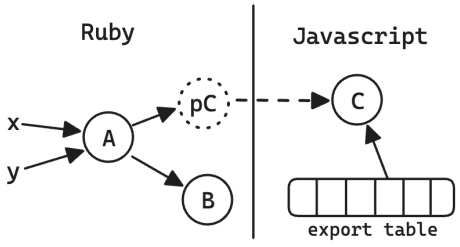

Figure 4: Export table for GCs with FFI.

To solve this problem each runtime keeps an *export table* reachable from it's root set. It mirrors all remote references to a local object and therefore keeps the objects alive even if they are just reachable from a remote runtime.

Synchronizing two collectors

In their paper *Collecting Cyclic Garbage across Foreign Function Interfaces* the authors synchronies two tracing garbage collectors [3].

The authors identified two problems with the status qou. First, cyclic garbage across language boundary is never freed and reference structures with many boundary crossings (which is referred to as *zigzag garbage* in the paper) needs many collection cycles to be freed.

For cyclic garbage the problem arises from the export table which behaves like a simplified reference counter and therefore introduces the same problem reference counters have.

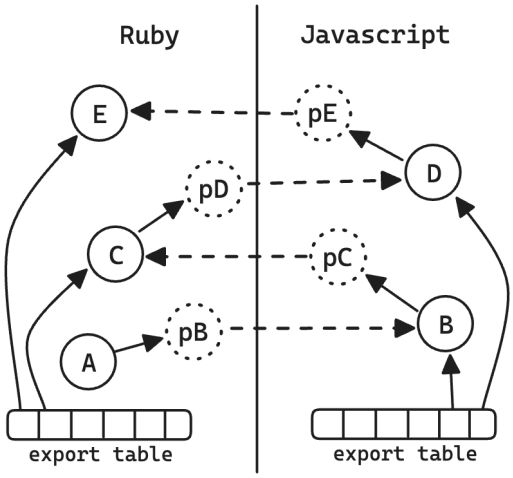

Figure 5: Zigzag garbage.

Figure 5 shows a chain of five objects with four boundary crossings, since none of the objects are reachable from the root set, they are all considered garbage. Now when the garbage collector in the ruby runtime runs it will only deallocate the `A` object and the proxy object `pB`. Deallocating `pB` will also remove the reference from the export table to `B` inside the Javascript runtime. However, since the ruby collector doesn't have any information about the reference graph on the Javascript side it cannot know `C` and `E` are also no longer needed and leaves them alive. This means before `C` can be freed, Javascript must run a collection and free `B` first. Therefore, to deallocate the complete structure five garbage collection cycles in the desired order have to run.

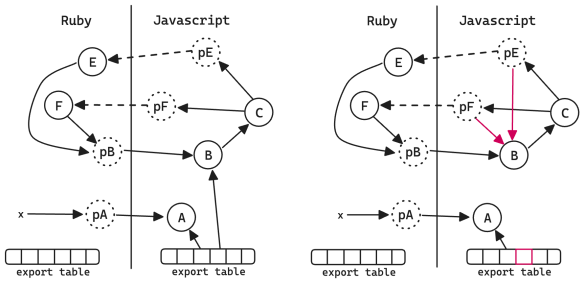

Figure 6: Mirrored reference graph (left before, right after).

The novel solution *ReGraph* described in the paper solves all those issues by giving one runtime access to the complete reference graph.

Right before Ruby is ready to trigger a collection it builds its current reference graph, compresses it to just include necessary information and sends it over to Javascript. There the graph is rebuild and the references from the export table are removed. Now, a collection is forced which has all the information to free all its objects of cyclic garbage or zigzag garbage at once. Once the Javascript finished its collection Ruby finally collects it's garbage.

In their paper the authors also prove that their algorithm is sound and complete.

To implement ReGraph they only needed to modify 63 lines of code to CRuby, exposing some internal APIs and the rest is implemented in Ruby and Javascript without modifying the Javascript runtime.

In benchmarks ReGraph almost always reduces the heap size in programs with remote references, sometimes quite drastically. In programs without remote references, it only marginally increases the heapsize.

However, the improvement on runtime is much less admirable. Often runtime is increased and in one especially complex benchmark by over 300%.

Synchronising two approaches

In *Efficient Cycle Detection on a Partially Reference Counted Heap* the author extends PyPy's default mark and sweep garbage collection algorithm to better integrate reference counted objects from CPython extensions [4].

CPython is the most common Python implementation and uses reference counting to manage it's heap. To detect and free cyclic garbage CPython has a special tracing garbage collector [6]. CPython, allows developers to easily integrate with Python code with C code of which many libraries take advantage.

Therefore, to allow such libraries to continue working in the PyPy runtime, it is forced to support reference counted objects.

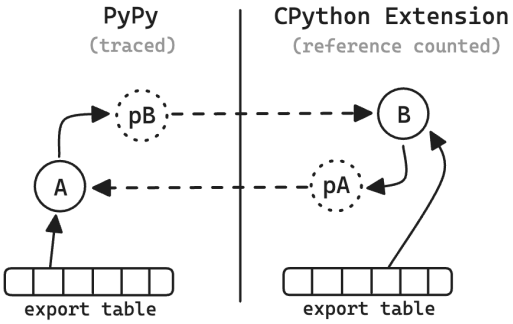

Figure 7: Cyclic garbage in PyPy.

By default the PyPy doesn't trace reference counted objects at all and therefore leaks all cyclic garbage structures that contain at least on reference counted object [4]. Figure 7 shows such a structure with one traced object `A` and one reference counted object `B`. Here `A` cannot be freed as it is kept alive by the export table (the author calls this *Links* in his paper, but for consistency and clarity I will stick to export table).

Trial deletion can resolve cycles entirely made up of reference counted objects but would fail at structured of mixed object as seen in Figure 7. Tracing garbage collection already can take care of cyclic garbage entirely made up of traced objects.

The author now combines both algorithms by, interleaving steps from both, therefore extending the default garbage collection to also trace reference counted objects.

In the paper the author provides a semi-formal proof for completeness and many benchmarks.

In applications with many cyclic structures the algorithm can decrease memory footprint and computation time. However, the additional complexity increases pause times as the marking phase has more work to do and can even increase memory footprint [4].

Conclusion and Future Work

While both papers solve problems of leaking cyclic structures around language borders, the context is so different that the solutions have remarkably little overlap.

Even though the paper *Collecting Cyclic Garbage across Foreign Function Interfaces* solves its problem across two different languages, that detail is only important to the point that there need to be two different garbage collectors that cannot share a heap space. However, the research could also be applied to distributed computing with two instances of the same runtime.

The authors, acknowledge the similarity of their problem space to distributed garbage collection and compare their implementation to some approaches in that field [3]. However, the authors don't discuss if ReGraph is applicable as a distributed garbage collector, which it should be.

While the implementation of *Efficient Cycle Detection on a Partially Reference Counted Heap* is impressive, algorithmically it doesn't provide much value as the paper is often focused on implementation details and the knowledge that tracing can find cycles in reference counted objects is known [2].

Related Work

In *Collecting Cyclic Garbage across Foreign Function Interfaces* the authors synchronize two garbage collectors by sending the reference graph of one runtime to the other. Therefore, the receiving runtime has a complete graph an can detect cycles [3].

Efficient Cycle Detection on a Partially Reference Counted Heap the author extends a tracing garbage collector to incorporate and trace foreign reference counted objects [4].

McCarthy introduced the concept of automatic memory management and the first tracing garbage collector in *Recursive functions of symbolic expressions and their computation by machine* [1].

To tackle some problems of tracing collectors and their overhead Collins invents reference counting in *A method for overlapping and erasure of lists* [5].

The garbage collection handbook is one of the most comprehensive summaries in the field of automatic memory management and not only explains many algorithms but also discusses their applications and shortcomings [2].

References

[1] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part I", *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[2] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2023.

[3] T. Yamazaki, T. Nakamaru, R. Shioya, T. Ugawa, and S. Chiba, "Collecting Cyclic Garbage across Foreign Function Interfaces: Who Takes the Last Piece of Cake?", *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 591–614, 2023.

[4] S. Beyer, "Efficient cycle detection on a partially reference counted heap", 2020.

[5] G. E. Collins, "A method for overlapping and erasure of lists", *Communications of the ACM*, vol. 3, no. 12, pp. 655–657, 1960.

[6] P. S. Foundation, "Garbage Collector Design". [Online]. Available: https://devguide.python.org/internals/garbage-collector/index.html