TU WIEN Informatics

Efficient Cycle Detection on a Partially Reference Counted Heap

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Stefan Beyer, BSc
Matrikelnummer 01225423

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 26. Februar 2020

Stefan Beyer	Andreas Krall

TU WIEN Informatics

Efficient Cycle Detection on a Partially Reference Counted Heap

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Stefan Beyer, BSc
Registration Number 01225423

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, 26th February, 2020

Stefan Beyer	Andreas Krall

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Stefan Beyer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. Februar 2020

Stefan Beyer

Danksagung

Zu allererst will ich mich bei meinem Professor, Andreas Krall, für seine anhaltende Unterstützung und seine Bemühungen, mich fokussiert und motiviert zu halten, bedanken. Obwohl ich eine gefühlte Ewigkeit für diese Arbeit benötigt habe, gab er mir innerhalb weniger Stunden oder Tage hilfreiche Rückmeldungen, wann auch immer ich eine Frage oder etwas zu begutachten hatte. Ich habe den Eindruck, dass er sich wirklich um seine Studenten kümmert, und versteht wie sehr sie sich für ihre Diplomarbeit anstrengen, etwas das nicht selbstverständlich ist.

Außerdem will ich mich bei meiner Mutter für ihre ununterbrochene Bemühung bedanken, alles von mir fernzuhalten und mich stets zu unterstützen, sodass ich mich auf mein Studium konzentrieren konnte. Deine Ratschläge sind vielleicht nicht immer angekommen, aber du hast nie aufgehört an mich zu glauben. Ich hoffe du weißt, dass ich das ebenfalls nicht für selbstverständlich halte.

Ich will auch meiner Freundin Chiara danken, für die Motivation und die emotionale Unterstützung, die sie mir gegeben hat um diese Arbeit zu vollenden. Ohne dich, hätte ich wahrscheinlich noch viel mehr Zeit benötigt.

Zudem möchte ich mich noch beim Rest meiner Familie, meinen Freunden und Kollegen für ihre Gesellschaft und ihr offenes Ohr bedanken, das sie mir geliehen haben, wann auch immer ich jemanden zum Zuhören brauchte. Die gemeinsamen Ausfahrten mit meiner Mountainbike-Gruppe, die mir geholfen haben meinen Kopf freizubekommen wenn er voller Fragen war, habe ich sehr geschätzt. Sehr wertvoll waren auch die tollen Erfahrungen mit meiner Improtheatergruppe, die mir Selbstbewusstsein gegeben und mich zum Lachen gebracht haben, auch wenn mir manchmal nicht zum Lachen war. Ohne euch allen hätte ich diese Arbeit wohl ebenfalls nicht fertig bekommen.

Zu guter Letzt möchte ich mich auch beim PyPy-Team bedanken, unter anderem bei Armin Rigo und Richard Plangger. Ihr alle habt mich nicht nur herzlichst Willkommen geheißen und mir die ursprüngliche Idee für meine Arbeit geliefert, sondern mir auch mit eurer Erfahrung und eurem Wissen geholfen, am richtigen Weg zu bleiben. Auch wenn ich manchmal zu stolz war, um eure Hilfe in Anspruch zu nehmen, weiß ich die Hilfestellungen und die Unterstützung, die ihr mir zukommen habt lassen, sehr zu schätzen.

Acknowledgements

First of all, I would like to thank my professor, Andreas Krall, for his continuing support and his efforts at keeping me focused and motivated to work on my thesis. Even though it took me almost forever to finish this piece of paper, he gave me helpful advice within a matter of hours or days, whenever I had a question or something to review. To me it seems like he really cares for his students and the hard work they put into their thesis, something that should not be taken for granted.

I would also like to thank my mother for her ongoing efforts, at keeping everything she possibly could away from me and always supporting me, so I could concentrate on my studies. The advice you gave me might not have always been heard, but you never stopped believing in me. I hope you know, that I also do not take this for granted.

I would also like to thank my girlfriend, Chiara, for the motivation and emotional support she gave me, to bring this thesis to the finish line. It would have probably taken me even longer, if not for you.

Also, I would like to thank all of my family, friends and colleagues for their company and friendly ear, when I needed someone to talk to. I really enjoyed the tours with my mountain bike crew, who helped me to free my mind, when I was full of questions. I also value the great experiences I had with my improvisational theatre group, who helped me gaining my confidence and made me laugh, when I needed it. I would not have finished this thesis, without all of you.

Last but not least I would thank the PyPy team, including but not limited to Armin Rigo and Richard Plangger, who not only gave me a warm welcome and the initial idea for this thesis, but also kept me on track by sharing their experience and knowledge. Even though I was sometimes too proud to seek help from you, I really apprieciate the support and guidance you gave me.

Kurzfassung

Zur automatisierten Speicherbereinigung, kurz Garbage Collection, existieren zwei grundlegende Ansätze: Tracing und Reference Counting. Ziel dieser Arbeit ist es, einen Algorithmus zu finden, der beide Ansätze auf effiziente Weise vereint, und der es ermöglicht Zyklen auf teilweise referenzgezählten Heaps zu erkennen, ein Problem das typischerweise in Sprachintegrationen auf Compiler-Ebene gefunden werden kann, und gleichzeitig die Auswirkungen auf die zu integrierenden Technologien und deren Garbage Collector niedrig hält. Zwei Versionen dieses Algorithmus wurden im PyPy Just-in-time-Compiler implementiert, um die Integration von CPython-Erweiterungsmodulen zu unterstützen. Der finale Algorithmus ist an Jython's JyNI-Integration [Ric16] angelehnt, kann aber als eigenständig betrachtet werden. Eine teilweise und eine vollständig inkrementelle Version dieses Algorithmus wurden entworfen und deren Korrektheit semiformell bewiesen. Die Implementierungen wurden mittels aufwändiger Tests verifiziert und mithilfe eigener Benchmarks verglichen. Die Ergebnisse zeigen, dass das Verhalten der vollständig inkrementellen Version des Algorithmus relativ gut ist und ein großer Teil des Mehraufwands kompensiert werden kann, der durch die vollständige Integration beider Ansätze entsteht.

Abstract

Two essential garbage collection techniques exist: tracing and reference counting. The goal of this work is to find an algorithm, to efficiently combine both approaches and collect cycles in partially reference counted heaps, a problem which is typically found in compiler-level language integrations, while keeping the impact on both integrated technologies and their existing garbage collector low. Two versions of the algorithm are implemented in the PyPy just-in-time compiler to support the integration of CPython extension modules. The final algorithm has been influenced by Jython's GC integrations for JyNI [Ric16], but still stands on its own. Semi- and fully-incremental versions of this algorithm are designed and their correctness is established by a semi-formal proof. The implementations are verified using sophisticated tests and their efficiency is measured by running several benchmarks. The results reveal, that the fully-incremental version of the algorithm seems to behave quite well and is able to compensate a lot of the introduced overhead, of fully integrating both garbage collection mechanisms.

Contents

CHAPTER 1

Introduction

Garbage collection has been around for more than half a decade, with the first papers reaching back to the 1960ies [JHM11]. The wide adoption amongst popular programming languages like Java, Python, JavaScript, C# etc. makes garbage collection an integrated part of modern software development. Research in this field is still conducted, even though the fundamentals are already well established. Two essential garbage collection techniques exist: tracing and reference counting. This thesis is concerned with combining those two approaches into a single algorithm efficiently, to cope with situations where both are used alongside each other.

1.1 Tracing and Reference Counting

Despite the huge amount of different algorithms and approaches to detect and free unused memory, they all boil down to two very simple basic algorithms. Tracing algorithms might be divided into mark-sweep, mark-compact and copying collection, but all of those algorithms have in common, that they need to trace all live objects, starting from the mutator roots. Roots are memory regions, that can be directly accessed by the application, like stack roots or static roots. All live objects are reachable from those roots, so if we trace the whole object graph starting from this roots, whether we mark or copy visited objects, we can determine the set of (potentially) live objects. Reference counting applies a different view on the liveliness of objects. Indstead of tracing the whole graph, the count of all ingoing references of an object is stored locally and increased or decreased accordingly, when new references to the object are created. Once the count drops to zero, the object can be freed instantly and without any further checks. Even though both techniques seem so fundamentally different, they can be subsumed into a universal theory of garbage collection [BCR04].

However, there is one severe issue with reference counting. While it works for most graphs, it can not detect dead, cyclic structures, as their reference count always stays above zero.

To solve this issue, a common approach is to use a technique called backup tracing, where a tracing algorithm is executed from time to time, to clear those cycles. This is also the point, where both approaches overlap, and an ideal opportunity to integrate them into a single algorithm. Even though this seems like an obvious solution, few research has been made in this field on how to do so efficiently, as we will see in the upcoming sections.

1.2 Cyclic Reference Counting

Forcing the application developer not to create reference cycles, transfers a lot of responsibility to the developer, so most language developers tend to prefer a different approach. Detecting such cycles automatically seems like a consistent strategy, as garbage collection is primarily introduced to keep the responsibility of freeing unused memory away from application developers. Thus, since the problem of reference cycles was first discovered, a lot of research has been made on how to solve this problem efficiently [SBYM13], [CYTW10], [HLM09].

Historically, several approaches have been proposed, to identify the reference, which creates a reference cycle, in order to clear them once they become unreachable [FW79], [Bro85], [Sal87], [PvEP88], [Axf90]. However, none of them has yet proven to be safe and efficient [JL96], . Trial deletion [Chr84], on the other hand, is a now widely adopted technique for handling cyclic structures [JHM11]. It is based on two observations:

- In any isolated cycle, all reference counts result from internal pointers (between objects in the structure).
- Garbage cycles can arise only from a pointer deletion that leaves a reference count greater than zero.

Trial deletion, as a side effect, might be used to identify only implicitly known mutator roots, but it can also be employed to perform partial tracing [MWL90]. Partial tracing, in contrast to global tracing, only traces those parts of the object graph, where dead cycles are suspected. It can only be applied on reference counted graphs and works as follows:

1. Deduct all internal references from the set of identified objects (including all reachable objects).
2. Scan for objects with a remainig (= external) reference count greater than zero.
3. Trace the graph, starting from these objects and increase the reference count of all reachable objects respectively.
4. Clear all objects (from this set), whose reference count is zero.

If the garbage collection algorithm is not able to identify such a set of objects, for example because of the specifics of the language, trial deletion might also be used on the whole object graph, as described in the original paper.

Since the discovery of trial deletion, several advanced approaches have been proposed, like concurrent or asynchronous cycle detection. Concurrent cycle detection allows trial deletion to be executed concurrently with the application (see next section), but in this case synchronization between the application and the garbage collector is necessary, which proves to be an issue in some cases, as we will see in the upcoming chapter. Asynchronous cycle detection delays the execution of the trial deletion algorithm, by collecting the set of potential objects and performing the actual algorithm at a later point in time, instead of eagerly performing trial deletion, each time a single object has been identified [Lin92]. Performance can also be improved, by statically recognizing objects that might never be part of a cycle, and ignoring such objects for most of the algorithm [BR01]. As we will see in Chapter 3, we can apply this optimization, as well as a variant of the asynchronous cycle detection.

1.3 Incremental Garbage Collection

Another general issue with garbage collection, that has seen a lot of attention by researchers, is how to keep the interruptions caused by the introduced algorithms to a minimum, also for reference counting [BR01], [JHM11]. In practice, some language developers seem to prefer incremental garbage collection, in contrast to concurrent approaches, as it is a relatively simple but effective approach to cope with long pause times. Concurrent approaches always need to be synchronized with the application, as both might access the same memory regions at the same time. Incremental approaches do not need sophisticated synchronization, their correctness is easier to proof and their implementations are easier to verify. This is because with incremental collection, either the garbage collector or the application runs at one point in time. However, the collection might be paused at any time, to allow the application to make progress. This seems like a good compromise between performance, pause times and simplicity for a lot of applications.

1.4 Partitioned Heaps

Applications not always manage their memory on one single, continous heap. Generational garbage collection is just one, but a very common reason for partitioning the heap [LH83], [Ung84]. Another one of these situations might be interoperability between different technologies [Inc]. Many modern programming languages offer support for C level modules, so legacy code written in C or other programming languages offering similar support, might be integrated into a single application on compiler level [jni], [csh], [pycb]. As soon as two or more languages are integrated, their garbage collection approaches also need to be integrated. Most of the times, this boils down to a partitioned heap, where each heap is managed by the respective language. Some integrations then force the application developer to refrain from creating reference cycles. Other integrations, offer automatic solutions for clearing such cycles, as we will see in the upcoming sections.

1.5 Partially Reference Counted Heap

In this special case, one partition of the heap is reference counted, when the other is not. Combining reference counting with tracing has already been explored in generational garbage collectors, with multiple generations using either tracing or reference counting [AP03], but this topic has not seen much attention outside of this area. When working with reference counted heaps, the mutator roots do not need to be known explicitly. Backup tracing algorithms are typically designed to cope with this issue, but typical tracing algorithms need explicit knowledge of all roots. This presents the first problem of integrating both approaches: how to find all roots, if only some of them are known explicitly? The second problem is, how to combine an existing tracing algorithm with an arbitrary backup tracing algorithm, as we might not want to introduce new garbage collection algorithms and implementations, for each integration. Finally, the question is, how to solve these problems efficiently, with low pause times.

1.6 Goal of this Work

The goal of this work is to find an algorithm, to efficiently collect cycles in partially reference counted heaps, typically found in compiler-level language integrations, while keeping the implications on both integrated technologies and their existing garbage collector low. More specifically, we will try to find an algorithm to integrate a rather simple reference counting scheme with a generational mark-sweep garbage collector. We will design two versions of this algorithm and measure its efficiency by implementing both versions in the PyPy just-in-time compiler and running several benchmarks. As a side goal, PyPy will then be able to collect such cycles, which it was previously unable.

The following chapter gives an introduction into PyPy and it's default garbage collector, as well as a short introduction into Richter's approach with Jython and Microsoft's approach with COM and .NET to solve similar problems.

CHAPTER 2

State of the Art

Apart from generational garbage collection, partially reference counted heaps have neither been extensively studied in theory nor widely adapted in practice. However, some scientific papers on this topic exist and there are also some applications that solve similar issues, like the one we would like to solve in PyPy. Before we take a look at how two partitioned heaps, one reference counted, the other one not, can be fully integrated with regards to garbage collection, let us take a look at how PyPy solves this problem currently.

2.1 PyPy and cpyext

PyPy is a just-in-time compiler for Python programs [pypc]. It features many optimizations, aiming to execute Python programs faster than the standard implementation *CPython* (see upcoming section). In contrast to the CPython interpreter, PyPy uses tracing instead of reference counting for garbage collection. It also features a compatibility layer for CPython extension modules, called *cpyext*. CPython extension modules offer application developers the opportunity to extend their Python application with C or C++ code. Some of them, like NumPy [num], are very popular amongst developers, so their support seems to be crucial. Those extension modules integrate into CPython's reference counting implementation for memory management. For PyPy, this presents a problem, as the default mark-sweep garbage collector is not compatible with CPython's reference counting scheme.

2.1.1 Garbage Collection

The current version of PyPy uses an incremental, generational garbage collector called *IncMiniMark* [pypa]. IncMiniMark is PyPy's default and recommended garbage collector. Other garbage collectors are also available, but are not covered here. Before we take a closer look at the garbage collection algorithm, let us take a quick look at the memory layout of the virtual machine.

2.1.2 Memory Layout

IncMiniMark handles objects in two generations on a partitioned heap. Objects are stored in various memory structures, depending on their size. Once objects are promoted to the second generation, they are never moved again, so their address will always stay the same.

- Small objects in the first generation are stored in the nursery, a fixed-sized memory buffer. In the second generation, they are stored in a memory hierarchy, which consists of arenas, pages and buffers. Arenas consist of continuous memory regions, which are allocated by the operating system They are managed in linked lists. Arenas contain multiple pages, which are also managed in lists. Pages contain a predefined number of buffers of equal size and each buffer can fit exactly one object. The exact location of a promoted object will thus be determined by its size and the number of free blocks within a suitable page.
- Large objects are always stored in a single buffer, which is directly allocated by the operating system. These buffers are managed in two stacks, one stack for each of the two generations. Large arrays are managed in two-level card tables within those buffers, card marking is used to aid the garbage collector.
- Medium objects are objects, which are too big to fit in arenas, but still fit in the nursery. Those objects are stored in the nursery in the first generation, but are moved into buffers in the second generation, where they are treated like large objects.
- Specific objects, independent of their size, might be directly allocated as second generation objects. This is determined by the language implementation.
- Some objects might also get pinned by the language implementation, so their memory address must not be changed by the collector. In case they are still in the nursery when they are pinned, they are forced to stay there and are not moved outside. If they get unpinned at a later point in time, they will eventually be promoted to the second generation.

2.1.3 Default Behavior

Before we take a look at the altered behaviour, once CPython extension modules have been loaded and the heap becomes partitioned into a reference counted and a non-reference counted partition, we will take a look at how the IncMiniMark collector behaves by default.

Two types of collections are executed: minor and major collections. Following the weak generational hypothesis, only objects in the young generation are collected in a minor collection to optimize performance. A simple copy collection is executed, where surviving objects are copied to the second generation. Minor collections are not incremental, in

contrast to major collections. Technically, major collections only process objects in the old generation. However, at the beginning of a major collection, a minor collection is triggered, so all surviving objects are then part of the old generation. Then, an incremental mark-sweep algorithm is executed, which collects those objects.

Minor Collection

A minor collection is typically executed, once the nursery has been filled and needs to be cleared. During a minor collection, surviving objects are promoted to the old generation, except for pinned objects in the nursery. An object of the young generation survives, if it is reachable by a root, or by an object in the old generation through an inter-generational pointer.

Small objects are moved outside of the nursery into arenas, medium objects are moved into separately managed buffers and references to large objects are moved from the young generation stack to the old generation stack.

Inter-generational pointers from the old generation to the young generation are recorded in a remembered set. This set is updated by a write barrier and implemented as a list. Using this list, objects from the young generation, which are directly referenced by an inter-generational pointer are promoted to the old generation. For inter-generational pointers to pinned objects, a special list is used. If card marking for large arrays is enabled, an additional list is used for those objects too.

Major Collection

Major collections are executed when the used memory has grown by a configurable amount. They implement an incremental, uniprocessor mark-sweep algorithm. This means, that mutator and collector execution is interleaved and never happens concurrently. As already noted, they trigger a minor collection, before they are executed. Actually, a minor collection is executed before each increment. Because of the incremental nature of the collector, synchronization between the mutator and the collector is needed.

The correctness of a mark-sweep implementation can be formally proven by the tricolor abstraction [DLM+76], [DLM+78]. In this abstraction, every object has one of three colors: white, grey and black. White objects have not been marked (yet). Grey colors have been marked, but their children might have not been marked already. Black objects have been fully processed, as all their children have also been marked (so they are either grey or black). Once all objects are colored black, the marking phase might terminate, as all reachable objects have been marked. All unmarked (white) objects can be sweeped, as they are unreachable and thus dead.

For incremental marking, write barriers are typically used to ensure that the invariant is not violated. PyPy uses a grey Boehm write barrier [BDS91]. This barrier is relatively simple: Black objects which are modified between two increments are reverted to gray and added to the working set. Objects are allocated white, but once they are dragged out of

the nursery they become grey. The same is true for objects which were allocated outside of the nursery and survive a minor collection. This means that after these preprocessing steps, the weak tricolor invariant is restored at the beginning of each major marking step.

A working set is used to keep track of all grey objects. This set is initialized at the beginning of a major collection, by adding all roots to the set, and processed during each increment. For each object which is processed, all directly referenced white objects are marked grey and added to the working set. Afterwards the object is marked black and removed from the working set. Once the incremental limit has been reached, the increment is over and the collector is paused, to allow the mutator to continue its execution. In case the set becomes empty before the limit is reached, the marking phase of major collection is over and the heap can be sweeped.

Sweeping

At the end of a minor collection, the nursery does not need to be sweeped explicitly, as all live objects have already been copied out of the nursery, so the pointer to the next free memory area only needs to be reset to the beginning of the buffer. The sweeping phase after a major collection is also relatively simple. All unmarked (white) objects are sweeped from the heap. Depending on the memory structure, the memory area is freed by the operating system or it is simply marked as unused.

Support for Finalizers and Weak References

The garbage collector also supports finalizers and weak references, which we will not cover here in detail, as their behaviour does not need to be adapted to support cyclic reference counting. This includes some special scenarios, like object shadows, which were introduced to support stable IDs, which are needed for functions like Python's *id()*.

2.1.4 Support for cpyext Modules

When the first CPython extension module is loaded, cpyext is initialized and the behaviour of the garbage collector changes, to incorporate cpyext-managed objects. At the end of the marking phase of each minor and major collection, additional code is executed to ensure the safety of the algorithm.

In detail, the integration works as follows: A reference from a PyPy-managed (non-reference counted) to a C-managed (reference counted) object is implemented as a reference to a non-reference counted proxy which is linked to the C-managed object. Conversely, a reference from a C-managed object to a PyPy-managed object is implemented as a reference counted proxy which is linked to the PyPy-managed object. Links are saved in an additional field in the header of the reference counted object and in a total of four lists (two for each generation, depending on the direction of the reference).

Unlinked, reference counted objects are freed when their reference count drops to zero and unlinked, non-reference counted objects are still sweeped, when they are unmarked at the end of the marking phase.

Linked objects are always kept alive by the additional routines, as long as the reference counted part is alive. Non-reference counted proxies are kept alive, so that they do not have to be recreated too often. Reference counted proxies keep the non-reference counted object alive, to guarantee the safety of the algorithm. Any reachable objects on both parts of the heap are also kept alive in this case. Also, linked, reference counted objects have an additional artificial reference count, so that they are not freed once their actual reference count drops to zero, as some of them might only be kept alive by their linked, non-reference counted object. Linked object pairs are only freed, when the actual reference count of the reference counted object is zero and the non-reference counted object is unmarked at the end of the marking phase.

These adaptions guarantee the safety of the algorithm, which means that no live objects are freed. But they lead to floating garbage, as objects in cyclic structures containing at least one reference counted object are never freed, even if they become unreachable. This issue should be fixed by our adaptions to the algorithm.

2.2 CPython

CPython is the standard implementation for Python [pyt]. It manages a purely reference counted heap, so we do not need to worry about any special cases, caused by references to and from non-reference counted objects. However, we can use parts of CPython's cyclic garbage collector for the design of our algorithm, so we will take a look at its collection logic [cpyb].

CPython's garbage collector implements a generational, global trial deletion algorithm. Objects are managed in lists (one for each generation) and the trial deletion algorithm is executed on the generation about to be collected and all prior generations. References from later generations are simply treated as external references.

Weak references and finalizers need to be specially handled. Weak references in Python also supported callbacks, when the weakly reference object dies. They might not be called, if the weak reference is about to die, so those callbacks need to be cleared beforehand. Python's legacy finalizers might not be called on cyclic structures, so objects using them and all objects they keep alive, are moved to a list (*gc.garbage*), where they can be manually cleared by the application developer. Modern finalizers should be called, but they might resurrect some objects, so all the set of all potentially dead objects needs to be processed again, after all modern finalizers have been called.

The implementation is neither concurrent nor incremental, which results in potentially high pause times, but the application of generational garbage collection should help reducing pause times, according to the weak generational hypothesis (a lot of newly created objects die young) [FF81]. The usage of five generations on the other hand is

questionable, as the strong generational hypothesis (younger, not newly created objects die sooner than older objects) is generally not that evident [Hay91].

2.3 Jython's JyNI

Jython is an alternative Python implementation on top of the Java virtual machine (JVM) [jyt]. Just like PyPy, Jython has a compatibility layer for CPython extension modules, which is called JyNI [Ric14]. JyNI faces a very similar problem: the garbage collector of the JVM also uses a mark-sweep algorithm to clear dead objects.

Richter et al. proposed an algorithm to integrate the reference counting scheme used for those extension modules with Jython [Ric16]. Their approach is implemented on top of the JVM, where the unmanaged, reference counted graph is mirrored to the JVM-managed, non-reference counted graph. JVM's built-in mark-sweep implementation is then able to detect unreachable cyclic structures on the reference counted heap and across both heaps. This is necessary, as Jython aims to be compatible with any unmodified JVM implementation and they generally do not offer an interface to interact with the garbage collector on a low level. This is in stark contrast to PyPy, where the garbage collector can be modified to support any kind of low level integration. Anyway, we can use their approach and modify it, so we can efficiently clear refeference cycles in PyPy's garbage collector.

In JyNI, the reference counted graph is permanently mirrored to the non-reference counted graph in an asychronous manner. Built-in objects implement change notifications, so changes are directly applied on the mirrored graph. Other reference counted objects, which are implemented in the extension modules and cannot be adapted, need to be mirrored asynchronously, as reliable write barriers are not possible for raw C objects. This means that not only at the beginning of an increment, but also at the beginning of a major collection, the mirrored graph might not be synchronous with the real object graph. Thus, before objects are sweeped, dead structures need to be checked for consistency. Only if their internal references and their reference count matches, they can be sweeped. Weak references and finalizers are quite difficult to support in this case. We will not go into any details, but we will see that our modifications to the core algorithm simplify the handling of weak references and finalizers.

When mirroring the reference counted graph, the issue of only implicitly known external roots can be solved quite elegantly. They can easily be detected, as the reference count of externally referenced objects does not match the count of the ingoing references in the mirrored graph. Once we have identified these objects, we can treat them as roots of the whole object graph.

In PyPy and cpyext, we can omit the mirroring of reference counted objects and process them directly, which simplifies the algorithm dramatically, as we do not need any synchronization or special handling for weak references. The root detection can be implemented similar to CPython's root detection. However, this might probably result

in higher maximum pause times, as we will have to perform this in one single increment. But we can change the behaviour of our mark-sweep implementation, so that we are able to defer the marking phase of the reference counted heap to the last increment, which will only increase the pause time for this last increment and keep the pauses, caused by all other increments, on the same level.

On the other hand, we can modify Richter's approach, so it does need to mirror the reference counted graph permanently. Instead, we can create a low-level snapshot of the reference counted graph at the beginning of the reference counted marking phase, which we can throw away at the end of the marking phase. This should solve the problem of higher pause times, while still keeping the algorithm relatively simple. We can apply the same synchronization mechanism that JyNI does, to keep track of mutations to the graph between increments. Even in this case, we do not need any special handling for weak references. Finalizers need to be specially handled in both cases.

2.4 Microsoft .NET/COM integration

Microsoft's Component Object Model (COM) [coma] integration in .NET [dotb] also faces a similar problem. COM uses reference counting, while the Common Language Runtime (CLR), which executes .NET code, implements a mark-sweep algorithm. Microsoft even holds several patents for clearing cycles in this scenario [HK15], [HK17]. However, there is one crucial difference: application developers are not allowed to create strong cycles in COM, only cycles which are closed by a weak reference are allowed [comb]. This means that there can only exist (strong) cyclic structures on the non-reference counted (CLR-managed) heap and across both heaps. A specialized, local trial delection algorithm can thus be applied quite efficiently, as all cyclic structures that include reference counted objects have to include at least one cross-reference. The patented algorithm exploits this fact and additionally applies some optimizations, to detect such cycles more efficiently. As strong cycles in CPython extension modules are allowed, Microsoft's algorithm cannot be applied in our situation.

CHAPTER 3

Algorithms

As we have seen in the previous chapter, no specialized algorithm exists, to solve our problem in an efficient way. This means, that we have to find a new way of combining reference counting with tracing in our situation.

The trial deletion algorithm is the most popular choice for local tracing [JHM11]. It has the advantage, that it can operate without the exact knowledge of the roots of the heap. This is especially useful, as we are only aware of the subset of reference counted objects, which can potentially become part of a cycle. However, an ideally minimal set of candidate objects, which might currently be members of a garbage cycle, would be desirable. To acheive this, several approaches exists [MWL90], [HLM09], [CYTW10], [SBYM13], however due to the programming model, there is no exact way of telling which subset of objects is currently part of a cycle. Also, trial deletion does not seemlessly integrate with the mark-sweep algorithm used for the non-reference counted heap and by itself can only be used for purely reference counted systems.

But we can combine both approaches: First, we detect all roots of the non-reference counted heap using the standard mark-sweep techniques. Then, we execute the first phase of the trial deletion algorithm and simply treat all reference counted objects as potential members of a garbage cycle. This leaves us with a set of reference counted objects with a reference count greater than zero. Those objects have external references to them and can be treated like roots to our set of known reference counted objects. Now that we have collected all roots on both heaps, we can execute a global tracing algorithm. First, we execute the marking phase of a typical mark-sweep algorithm for the non-reference counted heap and the second phase of the trial deletion algorithm for the reference counted heap in conjunction (we keep on marking/incrementing the reference count of reachable objects if we cross heaps). Then, we clear the heap using the sweeping phase of the mark-sweep algorithm and the third phase of the trial deletion algorithm.

In reality, we additionally have to take care of weak references and finalizers, as well as of the specifics of the runtime and the programming model. To improve the properties of our algorithm, primarily pause times, we might also choose to execute the phases incrementally and in a slightly different order.

3.1 Semi-Incremental Algorithm

The proposed algorithm performs global tracing across the reference counted and non-reference counted parts of the heap.

The roots of the non-reference counted heap can be easily determined by scanning the stack and collecting all static roots, as all non-reference counted objects are managed by PyPy's memory system. The roots of the reference counted heap are not explicitly known, because of the nature of the programming model used for CPython extension modules. The algorithm calculates the roots of the reference counted heap by deducting all internal references from the objects reference counts and scanning the heap for objects with a reference count greater than zero, just like CPython's GC.

On a high level, after the set of root objects is known, all reachable objects are marked or, respectively, their reference count is newly calculated (we will also call this marking in the following descriptions, for the sake of simplicity). Afterwards, the remaining, unreachable objects are either unmarked or still have a reference count of zero, or both. Those objects are sweeped and the memory is freed.

On a more detailed level, the steps are executed in a slightly different order, to minimize pause times. This is because the marking phase of the PyPy-managed heap can be performed incrementally quite easily, as write barriers are reliable and fairly cheap to implement, while the marking (and root determination) phase of the reference counted heap has to be done at once, as reliable write barriers on C-managed objects are not feasible. In the following section, we will propose a more advanced approach for dealing with this issue, by taking a snapshot of the reference counted heap (see Section 3.2). In both cases, additional specifics of the C programming model and its implementation in PyPy have to be taken into account, making the final algorithm slightly more complex.

The resulting algorithm is a mixture of the algorithm PyPy's default, incremental mark-sweep garbage collector *IncMiniMark* currently implements [pypa], and an adaptation of the algorithm CPython's garbage collector currently implements [cpyb]. Before we have a look at the pseudocode and a proof of correctness, a description of the algorithm and its relation to the algorithms it is based on is given.

3.1.1 Relation to IncMiniMark and CPython's GC

By default, IncMiniMark is only able to detect dead objects managed by PyPy's memory system. This means that only objects created from Python code are considered during a collection. On the other side, objects managed by cpyext (PyPy's compatibility layer

for CPython extension modules) are only freed, when their reference count drops to zero. Even though there is rudimentary support for cpyext-managed objects in PyPy, encapsulated by the rawrefcount module and implemented mostly inside IncMiniMark, the algorithm is not able to detect all kinds of dead structures. The implemented algorithm only deals with objects at the border between PyPy-managed and cpyext-managed objects, and ensures that objects on both sides keep each other alive, in case there are references between them (we will call those references links further on, to simplify the description and also because the actual implementation in PyPy refers to them as links; when we speak of references, we only refer to intra-heap references; we use the term pointer, to refer to both references and links). The adapted version takes all cpyext-managed objects into account, by applying parts of CPython's garbage collection algorithm. CPython uses its garbage collector solely to detect dead reference cycles, all other objects are directly freed when their reference count drops to zero. In the adapted version, dead, cyclic structures within and across both heaps are detected, utilizing existing parts of PyPy's IncMiniMark.

3.1.2 Description

The algorithm described here is able to detect all kinds of dead objects and additionally supports weak references and finalizers. The Python language currently features two kinds of finalizers, referred to as modern and legacy finalizers, both of which are supported by the algorithm. For the sake of simplicity and because the existing algorithm in PyPy already handles weak references and finalizers correctly for PyPy-managed objects, only the handling of weak references and finalizers concerning cpyext-managed objects will be described here.

Terminology

Objects managed by PyPy's memory system are called *PyPy objects* and they reside on the *PyPy heap*. Objects managed by cpyext are called *C objects* and they reside on the *C heap*. C objects which can potentially be part of a cycle, and therefore implement the traverse and clear methods, are called *garbage collected C objects*. They are added to the *list of garbage collected objects* (a doubly linked list) before initialization and are removed from that list before they die. C objects which can normally not be part of a cycle are called *non-garbage collected C objects*. There is only one special case, where such objects can be part of a cycle, which is explained in the following paragraph.

Objects with cross-references are called *linked objects*. Conceptually, they are realized as two objects on both heaps, which are linked to each other. One object acts as a proxy to the real object on the other side and keeps it alive. There can only be at most one proxy for each real object, which means that linked objects can be viewed as pairs. As an optimization, to avoid creating and throwing away too many proxy objects, a linked C object that acts as a proxy to a PyPy object is cached and therefore kept alive (for later reuse), as long as the PyPy object is alive. As proxy objects do not contain any references to other objects, this does not impose a great memory overhead or keep any

other objects which would otherwise be dead alive. The links between those objects are managed by cpyext and stored in two lists, depending on which side (C object or PyPy object) acts as a proxy. Also non-garbage collected (reference counted) objects can be linked to PyPy objects. In this case, they might be part of a cycle, as PyPy objects can generally be part of cycles. This means, that in our algorithm, we also have to take this special case into consideration.

Tuples

Tuples are initially added to a separate list, because some tuples might only contain primitive types or non-garbage collected C objects. As tuples are immutable in Python, we do not have to take tuples containing only such objects into consideration during tracing, as they will never be part of a cycle. However, all other tuples will be added to the actual list of garbage collected objects, before the C heap is processed.

Support for Weak References

In CPython, callbacks of weak references within garbage cycles need to be cleared before the cycles are sweeped. Otherwise, it might happen that both, the weak reference and the weakly referenced object, are part of a cycle. If the weakly referenced object is sweeped before the weak reference, the callbacks of the weak reference are called. Those callbacks now might hold references to already dead objects, so they might access or resurrect them. In the worst case, this might lead to unexpected behaviour or crash the application. However, in our case weak references are implemented as PyPy objects, which are always sweeped before C objects during phase 5a. This means that their callbacks will never be called, so they don't need to be cleared beforehand.

Support for Legacy Finalizers

Legacy finalizers should never be called by the garbage collector, as they are not safe to be called on cyclic structures. They are only called, if the reference count of an object drops to zero outside of the collection cycle. Instead, objects with legacy finalizers and all objects they keep alive, and would be dead otherwise, are added to the global list of garbage (gc.garbage) during phase 6d. From this list, they can be accessed and cleared manually by the user.

Support for Modern Finalizers

A modern finalizer should be called by the garbage collector exactly once, in case the object is about to die. If it resurrects any object, this must be taken into account by the garbage collector. It might not be safe to call the finalizer during the execution of PyPy's garbage collector, so this must be done outside of the environment of the garbage collector during phase 6b. This means that if modern finalizers are used, at least two collection cycles are needed to free dead cyclic structures.

Outline of the Algorithm

Text colored in black is part of the basic algorithm, text colored in yellow is part of the weak reference support, text colored in green is part of the modern finalizer support, text colored in blue is part of the legacy finalizer support. If you are only interested in the basic algorithm, the colored text can be safely ignored. If, for example, you only want to understand the weak reference support, it should be sufficient to read the black and yellow text.

1. Collect the roots of the PyPy heap (static roots, stack roots, ...).
2. Mark all reachable objects on the PyPy heap (using only references on the PyPy heap) recursively, starting from the previously collected roots.
3. Mark all other objects, in case cpyext is enabled:
   a) Move tuples which might be part of a cycle to the list of garbage collected objects and remove all other tuples from the list of newly created tuples.
   b) Collect the roots of the C heap:
      i. Decrement the internal references from the reference count of all garbage collected C objects (standard case) and decrement the reference count of all non-garbage collected linked proxy objects which are directly referenced by a garbage collected C object (special case), by iterating over the list of garbage collected objects and utilizing their traverse-method.
      ii. Scan for garbage collected C objects with external references (reference count still greater than zero, or linked to a marked PyPy object). Treat those objects as roots of the C heap
   c) Mark the garbage collected C objects and all reachable PyPy objects, starting from the roots of the C heap, found in the previous step. Increment the reference count of all reachable garbage collected C objects and mark all reachable, unmarked PyPy objects recursively. Also, increment the reference count of directly referenced non-garbage collected linked objects (special case). When proccessing a garbage collected C object, move it into a new list, so unreachable objects remain in the old list. The old list is now called *list of dead objects*.
   d) Look if there are any objects with legacy finalizers in the list of dead objects. If so, move all *of those* objects and all objects reachable from those objects (including PyPy objects) into a separate list and mark them as live (increment their reference count or mark them).
   e) Look if there are any objects with modern finalizers in the list of dead objects. If so, move *all* objects from the list of dead objects into a seperate list. Also, mark all PyPy objects reachable from this set of objects recursively. This means that no C object and no PyPy object which is only kept alive by any C object dies this cycle.

f) Iterate over all linked object pairs with C proxies and look, if there are any non-garbage collected objects with a reference count greater than zero. Mark all reachable PyPy objects (important for linked non-garbage collected C objects which are not directly referenced by any garbage collected C object).

4. Prepare C objects for sweeping

   a) Iterate over all linked object pairs and look, if there are any non-garbage collected objects on the C heap which were only kept alive by their PyPy proxy. If the PyPy proxy is unmarked and about to die, add the corresponding C object to a separate list, which will be sweeped afterwards.

   b) During the previous step, clear the lists of linked objects by removing all pairs with unmarked PyPy objects.

5. Sweep the PyPy heap.

   a) Clear all weakrefs and call the remaining callbacks. Weakrefs in cycles die before the cycle, as they are implemented as PyPy objects, so we do not be careful not to resurrect any objects by calling the callbacks of the remaining weakrefs.

   b) Sweep all unmarked PyPy objects.

6. Sweep the C heap (outside of the context of the garbage collector)

   a) Sweep all objects in the list created in 4a, using their decref method.

   b) Call all modern finalizers available (iterate over the list created in 3e).

   c) Destroy all objects in the list of dead objects (created in 3c), using their clear and decref method. Those objects were most likely part of a cycle.

   d) Move all objects from the list created in 3d to the global list of garbage (gc.garbage) and eventually create PyPy wrappers for objects on the C heap. In case another collection cycle is triggered during this action, consider all objects on the C heap alive, by skipping steps 3b-3e and marking all linked PyPy objects in step 3f, so this list can be safely processed. After this step, these objects are kept alive by the gc.garbage list.

3.1.3 Proof

Outline

Mark-sweep is a formally proven algorithm for determining live (reachable) objects and removing dead (unreachable) objects from the heap. The basic algorithm is quite simple. First, the roots are determined and then all reachable objects are recursively marked. Finally, the heap is scanned for unmarked objects, which are sweeped from the heap. The algorithm can also be executed incrementally, by introducing write barriers to drag modified objects back into the working set and executing the root detection and marking

phase in increments, until the working set is empty. Then, the sweeping phase is executed in increments, without any write barriers, as dead objects cannot be modified.

In our case, the roots of the non-reference counted part of the heap are determined using well established methods, like scanning the stack and static parts of the heap. The roots of the reference counted part of the heap are determined using an algorithm called trial deletion, where a set of objects (in this case the whole heap) is iterated, and their internal reference count is deducted. The remaining objects with a reference count greater than zero are those with external references to them. This is the set of root objects.

What is different is the order of execution. Normally, all roots are collected and then the marking phase starts. However, in our case we collect the roots of the reference counted part of the heap after the whole non-reference counted part has been marked, using only the non-reference counted roots. This might seem like a substancial change, but it is very similar to what happens in an incremental mark-sweep algorithm. We already perform the marking phase of the non-reference counted part of the heap incrementally. If we think about the marking phase of the reference counted part of the heap as just another increment (where we treat all reference counted roots as newly detected roots), it is easy to see that the algorithm, at its core, has not been changed at all.

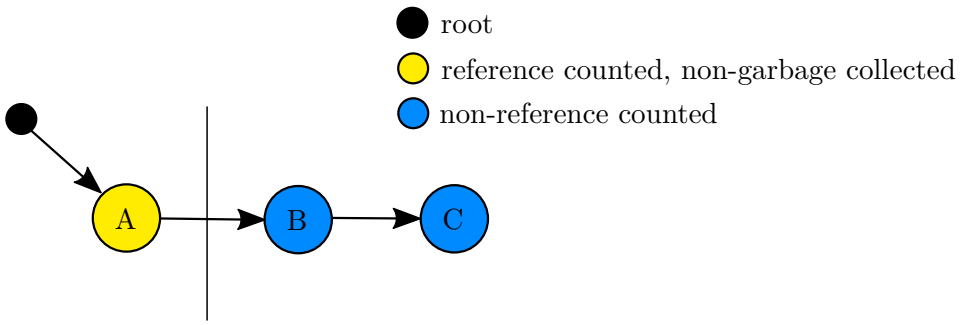

Figure 3.1: Non-garbage collected object keeping non-reference counted object alive

However, we have a special case, where some objects on the reference counted part of the heap are not garbage collected. Not garbage collected means, that they are not traced, because they cannot be part of a cycle. They are only freed when their reference count drops to zero (either during normal program execution or when clearing a cycle). When we think about the reasons why we need to perform mark-sweep on the reference counted part of the heap (which is to clear cycles), we quickly realize why such objects do not have to be traced. However, we still need to keep objects on the non-reference counted part of the heap alive, which are linked to such non-garbage collected objects (see Figure 3.1). There are also special cases like weak references or dictionaries, which are implemented on the non-reference counted part of the heap and can create cycles, but their reference counted wrapper is not garbage collected (which means they can in

fact be part o a cycle, see Figure 3.2).

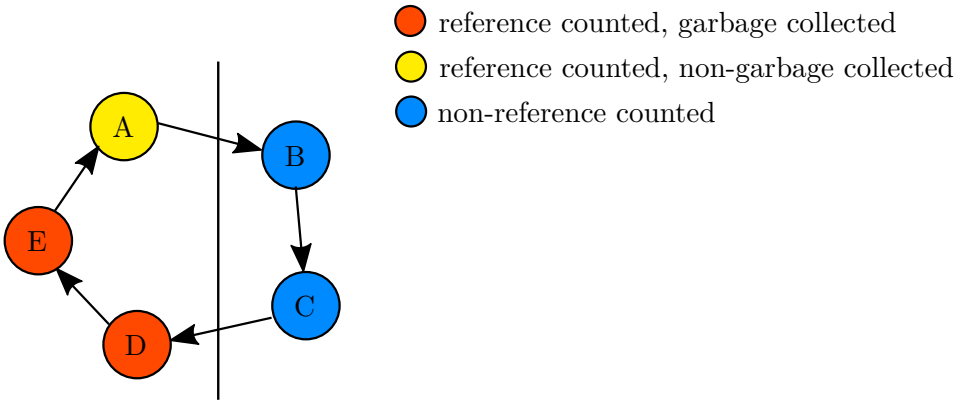

Figure 3.2: Cyclic structure with non-garbage collected object

We solve this issue, by taking these non-garbage collected, reference counted objects that are linked to a non-reference counted object into account, during the marking phase. They are included in the root detection phase of the reference counted part of the heap and will be marked just like garbage collected objects.

One thing left worth mentioning is, that if a non-garbage collected object happens to be indirectly referenced by an unreachable cycle, the non-garbage collected object itself will be sweeped during this collection, but any non-reference counted object that was kept alive by such a non-garbage collected object, will not be sweeped until the next full collection (see Figure 3.3). This is because the non-garbage collected object acts as a root for the non-reference counted object: The internal reference counts of objects A, B, C and E are deducted. D is not considered, because it is a non-garbage collected object which is not linked to a non-reference counted object. E is considered, because it is linked to a non-reference counted object. A, B and C now have a total reference count of zero, so they don't act as roots. E on the other hand still has a reference count of one, so it acts as a root. This means that during the marking phase, F is marked as a live object. During the sweeping phase, A-C are sweeped, D and E are removed alongside C, as their reference count drops to zero. During the next garbage collection, F is not marked again and sweeped.

Sweeping the heap can easily be done incrementally, as dead objects are unreachable and therefore have to keep their status. We only have to be careful about finalizers, which might resurrect objects. In case finalizers exist in the set of potentially dead objects, they must be executed before any object is freed. In our case, we deal with this situation by keeping all reference counted objects (and all reachable non-reference counted objects) alive, until the next collection cycle, if finalizers are detected in the set of dead reference counted objects. Between those two cycles we execute the finalizers and mark them as

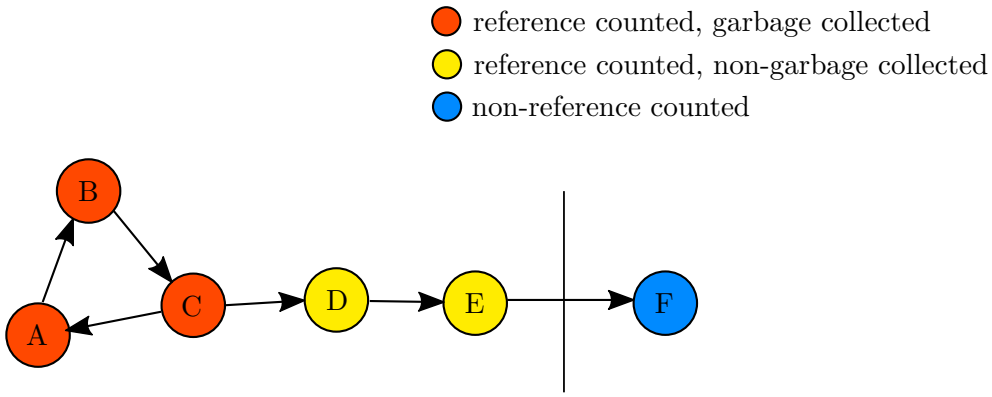

Figure 3.3: Dead, non-reference counted object kept alive by dying non-garbage collected object

executed (finalizers can only be executed once in Python). During the following cycle, we check the set of potentially dead objects from the last cycle again and look if any object has been resurrected, or if any finalizer still needs to be executed. If one of those two criterias is met, we have to check this set of objects again during this cycle. Otherwise, we can safely free the objects. As finalizers can only be executed once, at some point the objects will be freed, if they are not used any more.

Semi-Formal Proof of Correctness

For the semi-formal proof, we need to define some prepositions and assumptions first. Note, that we do not take weak references and finalizers into accout in the proof. However, we will show afterwards, that they fit nicely in our formalization. Also note that for simplicity, in the pseudocode, the non-reference counted marking phase (Phases 1 and 2) is not incremental. In the implementation, those two phases are executed incrementally. After Phase 2 has been completed, the rest of the algorithm is then executed in one single increment. We can conclude that the postconditions from the first two phases hold, no matter how those phases are executed, and that it should be trivial to replace both non-incremental phases with their incremental counterparts, in the context of our proof.

Definitions:

- $rc(x)$ ... the object is reference counted (a C object)
- $nonrc(x) := \neg rc(x)$ ... the object is not reference counted (a PyPy object)
- $gc(x)$ ... the object is garbage collected
- $root(x)$ ... root of the non-reference counted heap (PyPy heap)
- $ref(x, y)$ ... there exists a reference from object x to object y
- $link(x, y)$ ... objects x and y are linked to each other, x is a proxy
- $link\_rc(x, y) := link(x, y) \wedge rc(x)$

- $link\_nonrc(x, y) := link(x, y) \wedge nonrc(x)$
- $marked(x)$ ... the object x is marked or has a reference count greater than zero

Assumptions:

1. $\forall x.(nonrc(x) \supset gc(x))$ ... non-reference counted objects are garbage collected
2. $\forall x.(root(x) \supset nonrc(x))$ ... roots (of the non-rc heap) are non-reference counted
3. $\forall x, y.((rc(x) \leftrightarrow \neg rc(y)) \supset \neg ref(x, y))$ ... there are no references between the heaps
4. $\forall x, y.(link(x, y) \supset (rc(x) \leftrightarrow \neg rc(y)))$ ... links can only be cross-heap
5. $\forall x, y.\nexists z.(link(x, y) \supset ref(x, z))$ ... proxy objects can't have outgoing references
6. $\forall x, y, z.((link(x, y) \wedge link(x, z) \supset (y = z)) \wedge (link(y, x) \wedge link(z, x)) \supset (y = z))$ ... objects can only be linked with exactly one other object

Also, we have to define, what a live (garbage collected) object is. We use a recursive definition:

$live\_objs := live\_base \cup live\_pointers$
$live\_base := live\_roots \cup live\_trialdel\_roots \cup live\_linked\_roots$
$live\_pointers := live\_refs \cup live\_links \cup live\_wrappers$

Base cases:

- $live\_roots := \{x | root(x)\}$
- $live\_trialdel\_roots := \{x | gc(x) \wedge \exists y.(\neg gc(y) \wedge ref(x, y))\}$
- $live\_linked\_roots := \{x | \neg gc(x) \wedge \exists y.z(\neg gc(y) \wedge ref(y, x) \wedge link\_rc(x, z)))\}$

Recursion (via pointers):

- $live\_refs := \{x | gc(x) \wedge \exists y.(y \in live\_objs \wedge ref(y, x))\}$
- $live\_links := \{x | \exists y.(y \in live\_objs \wedge link(y, x))\}$
- $live\_proxies := \{x | \exists y.(y \in live\_objs \wedge link\_rc(x, y)\}$

Note, that the set of $live\_linked\_roots$ consists only of non-garbage collected objects. Also, in the sets of $live\_links$ and $live\_proxies$ some non-garbage collected objects might be contained. These are the special cases described in the previous chapters. Even though they are non-garbage collected objects, we can still mark them and treat them in the same way we treat garbage collected objects.

Theorem 1 *At the end of the marking phase, before we sweep unmarked garbage collected objects, all live garbage collected objects are marked, including some special non-garbage collected objects.*

Proof Our goal is to show that exactly those objects in the set of $live\_objs$ are marked before the heap is sweeped. For this, let us have a look at the pseudocode, annotated with postconditions (see Algorithm 1). All of those postconditions are accumulative, so after every method the postconditions from the previous methods should still hold. In the end we will show, that our set of accumulated postconditions correspond to our overall

Algorithm 1 Semi-Incremental Algorithm

Ensure:

1 $\forall x.(root(x) \supset marked(x))$
2 $\forall x, y.((nonrc(x) \wedge ref(x, y) \wedge marked(x)) \supset marked(y))$
3 $\forall x, y.((\neg gc(x) \wedge gc(y) \wedge ref(x, y)) \supset marked(y))$
4 $\forall x, y, z.((\neg gc(x) \wedge \neg gc(y) \wedge ref(x, y) \wedge link\_rc(y, z)) \supset marked(y))$
5 $\forall x, y.((link\_nonrc(x, y) \wedge marked(x)) \supset marked(y))$
6 $\forall x, y.((rc(x) \wedge ref(x, y) \wedge gc(y) \wedge marked(x)) \supset marked(y))$
7 $\forall x, y.((link\_rc(x, y) \wedge marked(x)) \supset marked(y))$
8 $\forall x, y.((link\_rc(x, y) \wedge marked(y)) \supset marked(x))$

9 **procedure** MARK
10 COLLECTNONRCROOTS
11 MARKNONRC
12 COLLECTRCTRIALDEL
13 COLLECTRCLINKED
14 MARKRC
15 MARKPROXIES

postcondition (that all exactly all live garbage collected objects are marked). First, let us show that all of the individual postconditions actually hold.

Algorithm 2 Semi-Incremental Algorithm - Phase 1

Ensure:

1 $\forall x.(root(x) \supset marked(x))$

2 **procedure** COLLECTNONRCROOTS
3 **for all** *root* in *NonrcRoots* **do**
4 Mark *root* and add it to *working_set*

Postconditions 1 and 2 are quite straightforward, as the two methods (Algorithm 2 and Algorithm 3) constitute the marking phase of a typical mark-sweep algorithm. We only have to be careful about not violating the postcondition of Algorithm 3 later on. Even though we execute these phases incremental, the non-incremental version is given here for simplicity.

Postconditions 3 and 4 are concerned with reference counted roots (Algorithm 4). Before Phase 3, all reference counted objects are marked, which means their reference count is greater than zero. Otherwise, they would have already been cleared by the standard

Algorithm 3 Semi-Incremental Algorithm - Phase 2

Ensure:

1 $\forall x.(root(x) \supset marked(x))$
2 $\forall x, y.((nonrc(x) \wedge ref(x, y) \wedge marked(x)) \supset marked(y))$

```
3  procedure MarkNonrc
4      while working_set ≠ ∅ do
5          Pop x from working_set
6          for all y in References(x) do
7              if not IsMarked(y) then
8                  Mark y and add it to working_set
```

Algorithm 4 Semi-Incremental Algorithm - Phase 3

Ensure:

1 $\forall x.(root(x) \supset marked(x))$
2 $\forall x, y.((nonrc(x) \wedge ref(x, y) \wedge marked(x)) \supset marked(y))$
3 $\forall x, y.((\neg gc(x) \wedge gc(y) \wedge ref(x, y)) \supset marked(y))$
4 $\forall x, y, z.((\neg gc(x) \wedge \neg gc(y) \wedge ref(x, y) \wedge link\_rc(y, z)) \supset marked(y))$

```
5  procedure CollectRcTrialdel
6      for all x in RcGcObjList do                    ▷ Subtract internal references
7          for all y in References(x) do
8              if IsGc(y) or IsLinkedProxy(y) then
9                  Subtract 1 from reference count of y
10     for all x in RcGcObjList do
11         if IsMarked(x) then
12             Add it to working_set
13     for all x in LinkedRcProxyList do
14         Decrement reference count of x             ▷ Remove artificial reference count
15         if IsMarked(x) then
16             Add it to working_set
```

reference counting procedure. The first loop now unmarks all garbage collected objects and linked proxies, except those with external references, by deducting all internal references from the reference count. After the first loop, only objects with external references remain marked, as their reference count is still greater than zero. In the second and third loop, the reference counted heap is scanned for marked objects, which are added to the working set. Also, the reference count of linked is deducted by the artifical reference

count added when linking to an non-reference counted object. We might add this reference count again, during Phase 6 (Algorithm 7). If we now analyze both postconditions, we see that only reference counted objects are concerned, because non-reference counted objects might not have ingoing references from non-garbage collected (and thus reference counted) objects. These ingoing references from non-garbage collected objects represent the external references and respectively their reference count, after the internal references have been deducted. It should be easy to see now, that only such garbage collected objects and such linked reference counted proxies remain marked, with one exception: linked reference counted objects. We will handle them in the next phase.

Algorithm 5 Semi-Incremental Algorithm - Phase 4

Ensure:

1 $\forall x.(root(x) \supset marked(x))$
2 $\forall x, y.((nonrc(x) \wedge ref(x, y) \wedge marked(x)) \supset marked(y))$
3 $\forall x, y.((\neg gc(x) \wedge gc(y) \wedge ref(x, y)) \supset marked(y))$
4 $\forall x, y, z.((\neg gc(x) \wedge \neg gc(y) \wedge ref(x, y) \wedge link\_rc(y, z)) \supset marked(y))$
5 $\forall x, y.((link\_nonrc(x, y) \wedge marked(x)) \supset marked(y))$

```
6  procedure CollectRcLinked
7      for all x in LinkedNonrcProxyList do
8          y ← PointsTo(x)
9          Decrement reference count of y            ▷ Remove artificial reference count
10         if IsMarked(x) then
11             Increment reference count of y and add it to working_set
```

Postcondition 5 is quite simple: we remove the artificial reference count from linked, reference counted objects in case their non-reference counted proxy is unmarked. The remaining linked, reference counted objects are therefore only marked, if their non-reference counted proxy is marked (or if they have other ingoing references, which were detected in the previous phase). Now that we have detected all roots of the reference counted heap, we are ready to mark all reachable objects.

Phase 5 is primarily concerned with fulfilling Postconditions 6 and 7, however, as we will be marking linked non-reference counted objects (Postcondition 7), we also have to be careful about not violating Postconditions 2 and 5. On a high-level, we perform the same mark-sweep algorithm, as we did during Phase 1. On a more detailed level, we have to distinguish between reference counted and non-reference counted objects and between references and links. However, according to our knowledge about mark-sweep algorithms, it should not be too difficult to see, that all of the concerned postconditions hold, once the working set is empty.

The last phase is quite straightforward. Phase 6 adds an artificial reference count to all

Algorithm 6 Semi-Incremental Algorithm - Phase 5

Ensure:

1 $\forall x.(root(x) \supset marked(x))$
2 $\forall x, y.((nonrc(x) \wedge ref(x, y) \wedge marked(x)) \supset marked(y))$
3 $\forall x, y.((\neg gc(x) \wedge gc(y) \wedge ref(x, y)) \supset marked(y))$
4 $\forall x, y, z.((\neg gc(x) \wedge \neg gc(y) \wedge ref(x, y) \wedge link\_rc(y, z)) \supset marked(y))$
5 $\forall x, y.((link\_nonrc(x, y) \wedge marked(x)) \supset marked(y))$
6 $\forall x, y.((rc(x) \wedge ref(x, y) \wedge gc(y) \wedge marked(x)) \supset marked(y))$
7 $\forall x, y.((link\_rc(x, y) \wedge marked(x)) \supset marked(y))$

```
8  procedure MarkRc
9      while working_set ≠ ∅ do
10         Pop x from working_set
11         if IsRc(x) then
12             for all y in References(x) do
13                 if IsGc(y) or IsLinkedProxy(y) then
14                     if not IsMarked(y) then
15                         Add y to working_set
16                     Increment the reference count of y
17             for all y in Links(x) do
18                 if not IsMarked(y) then
19                     Mark y and add it to working_set
20         else
21             for all y in References(x) do
22                 if not IsMarked(y) then
23                     Mark y and add it to working_set
24             for all y in Links(x) do
25                 if not IsMarked(y) then
26                     Add y to working_set
27                 Increment the reference count of y
```

reference counted proxies to keep them alive, if the non-reference counted object they point to stays alive. It should be easy to see, that after this phase, Postcondition 8 holds. As proxies might not have any outgoing references, we do not need to worry about violating any other postconditions.

Algorithm 7 Semi-Incremental Algorithm - Phase 6

Ensure:

1 $\forall x.(root(x) \supset marked(x))$
2 $\forall x, y.((nonrc(x) \wedge ref(x, y) \wedge marked(x)) \supset marked(y))$
3 $\forall x, y.((\neg gc(x) \wedge gc(y) \wedge ref(x, y)) \supset marked(y))$
4 $\forall x, y, z.((\neg gc(x) \wedge \neg gc(y) \wedge ref(x, y) \wedge link\_rc(y, z)) \supset marked(y))$
5 $\forall x, y.((link\_nonrc(x, y) \wedge marked(x)) \supset marked(y))$
6 $\forall x, y.((rc(x) \wedge ref(x, y) \wedge gc(y) \wedge marked(x)) \supset marked(y))$
7 $\forall x, y.((link\_rc(x, y) \wedge marked(x)) \supset marked(y))$
8 $\forall x, y.((link\_rc(x, y) \wedge marked(y)) \supset marked(x))$

9 **procedure** MarkProxies
10 **for all** x in *LinkedRcProxyList* **do**
11 $y \leftarrow$ PointsTo(x)
12 **if** IsMarked(y) **then**
13 Increment the reference count of x

Now that we have shown that all of those postconditions hold at the end of the marking phase, we can reorder them and combine them into more compact expressions.

Root marking:

- 1: $\forall x.(root(x) \supset marked(x))$
- 3: $\forall x, y.((\neg gc(x) \wedge gc(y) \wedge ref(x, y)) \supset marked(y))$
- 4: $\forall x, y, z.((\neg gc(x) \wedge \neg gc(y) \wedge ref(x, y) \wedge link\_rc(y, z)) \supset marked(y))$

Recursive marking (via pointers):

- 2,6: $\forall x, y.((ref(x, y) \wedge gc(y) \wedge marked(x)) \supset marked(y))$
- 5,7: $\forall x, y.((link(x, y) \wedge marked(x)) \supset marked(y))$
- 8: $\forall x, y.((link\_rc(x, y) \wedge marked(y)) \supset marked(x))$

By comparing these postconditions to the definition of all live objects, we can conclude that all of those objects have been marked. If we additionally take into consideration, that before the marking phase no non-reference counted object was marked and that after Phase 4 only reference counted objects in the *working_set* were marked, we can conclude that exactly those objects have been marked.

Weak References

Weak references do not influence the correctness of the marking phase. We only have to make sure that the weak references get notified about the death of the object they are referencing and their callbacks are called correctly (after the object has died). As

we have already mentioned earlier, problems with weak references in cyclic structures cannot arise, due to the order in which the heaps are sweeped.

Legacy Finalizers

Legacy finalizers are processed just after the marking phase. All objects about to be freed are scanned for legacy finalizers and in case some are found, all reachable objects are marked and moved to a separate list. Only the remaining objects might be sweeped.

Modern Finalizers

Modern finalizers are processed just before the sweeping phase. All reference counted objects about to be freed are scanned for modern finalizers and in case some are found, all reference counted objects about to be freed are marked and moved to a separate list, including all reachable non-reference counted objects. No object will be freed during this cycle, except non-reference counted objects which are not reachable by any reference counted object (and thus can never be resurrected by a reference counted object with a modern finalizer). Before the next collection cycle, ideally all modern finalizers should have been called. In some rare cases, some modern finalizers might not have been called before the next cycle, in which case we will have to wait until one of the following cycles for the separate list to be processed. Once all of the finalizers of the objects in this separate list have been called, during the next cycle, we perform root detection and marking on the reference counted heap on all reference counted objects, including those that were moved to the separate list in previous cycles. In case we find roots in the separate list or encounter them during tracing, we move them from the separate list back into the set of live objects. All remaining objects in the separate list are definitely dead by now and can be sweeped after the current marking phase, as those objects can not be connected to the set of potentially dead objects, which might contain finalizers, and can therefore also not be resurrected by any finalizer.

3.2 Fully-Incremental Algorithm

This fully-incremental algorithm is an adaption of the semi-incremental algorithm, described in the previous chapter. While the original algorithm is only incremental with respect to the non-reference counted heap, this algorithm is also incremental with respect to the reference counted heap. It is based on the implementation of the garbage collector for CPython extension modules in the JyNI, but differs in the order of execution and integration into the existing garbage collector [Ric16].

3.2.1 Description

The main difference of the fully-incremental with respect to the semi-incremental algorithm is, that before the roots of the reference counted heap are collected, a lightweight snapshot of the garbage collected part of the reference counted heap (including linked non-garbage

collected objects) is taken. The snapshot is lightweight, because only references and metadata are stored in the snapshot. The whole root collection and marking phase of the reference counted heap then operates on this snapshot. As the snapshot will not change, even if we pause the collector, we only have to halt the user program to take the snapshot and can execute the rest of the marking phase incrementally. However, at the end of the marking phase, we have to check if the snapshot has been changed, to prevent sweeping live objects. We have to do this, because in contrast to the snapshotted reference counted heap, the non-reference counted heap can change during marking (which we cope with write barriers), and we have to synchronize those changes. Otherwise, a situation like depicted in Figure 3.4 might arise.

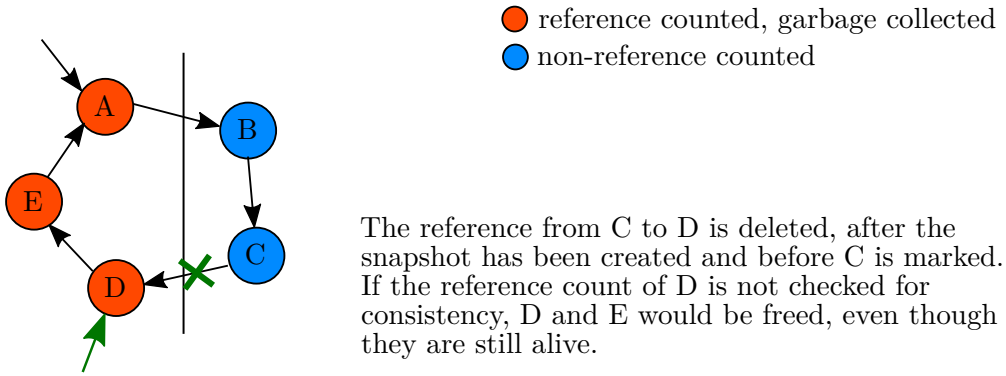

Figure 3.4: Example, why we need snapshot consistency checks

We cope with these kinds of situations by halting the user program and comparing the unmarked subgraph of the snapshot with the actual object graph, like in Jython and JyNI. If an object does not exist in the actual object graph any more, it has probably been deleted because its reference count dropped to zero. This is fine, because we would have sweeped it anyway. If the object still exists, its outgoing references and its reference count must still be the same. Otherwise we can conclude, that the subgraph has changed, which means some objects in the subgraph were still alive at some point during the collection and might still be alive now (dead objects can never change). If we detect such inconsistencies, we do not sweep any object in the subgraph (and any non-reference counted object reachable from the subgraph), otherwise, it is save to sweep the subgraph. In case there were actually dead objects in the subgraph and we had to keep the subgraph, we will process them again during the following cycle, where we hopefully won't detect any inconsistencies. If a lot of objects change during the increments, this might lead to a situation where we will never actually able to sweep any dead objects. We have to be careful about this situation, when deciding how long our increments are going to be.

However, we can conclude, that the safest way to prevent this situation is to take the snapshot as late as possible during the collection cycle and make our increments as long as possible (although not too long, otherwise we loose all of the benefits of incremental garbage collection).

3.2.2 Pseudocode

This time, the pseudocode is given as a fully-incremental algorithm. It consists of three phases, which will be executed in order. The individual methods are taken from the semi-incremental algorithm and perform the same tasks. However, the three methods *CollectRcTrialdel*, *CollectRcLinked* and *MarkRc* are executed on the snapshot, taken at the end of Phase 0. *MarkProxies* is executed after the snapshot has been synchronized with the real object graph at the end of Phase 2.

In contrast to the semi-incremental version, where we annotated the pseudocode with postconditions, we will only explain the two new methods concerning the creation and synchronization of the snapshot, and then proof afterwards, that nothing has changed with respect to the correctness of the algorithm.

Algorithm 8 Fully-Incremental Algorithm

```
1  procedure MarkIncrement
2      if phase = 0 then
3          CollectNonrcRoots
4          MarkNonrc
5          if finished_phase then
6              TakeSnapshot
7              phase ← 1
8      else if phase = 1 then
9          CollectRcTrialdel
10         CollectRcLinked
11         if finished_phase then
12             phase ← 2
13     else if phase = 2 then
14         MarkRc
15         if finished_phase then
16             SyncSnapshot
17             MarkProxies
18             phase ← 3
```

Snapshot Creation

During the creation of the snapshot, we make a lightweight copy of the whole garbage collected part of the reference counted heap. We include linked reference counted proxies, because they might also be part of a cycle and should be treated like garbage collected objects during the marking phase of the reference counted heap. As proxies might not have any outgoing references, we do not need to copy any references for those objects. Our lightweight copy primarily consists of the reference count and the outgoing references of each object. However we are only interested in internal references, because these are the only references we need for calculating the number of external references, during our root detection via trial deletion. For linked proxies and linked objects, we also save a pointer to the linked non-reference counted object/proxy.

This is the only information needed, for performing the following phases (Phases 1 and 2). The methods in this phases remain basically unchanged, except that they operate on the snapshot when it comes to reference counted objects. They still use the actual object graph, when it comes to non-reference counted objects. Also, they are now executed incrementally, as we will be able to detect changes of the reference counted graph during the synchronization of the snapshot later on. Changes on the non-reference counted heap will still be detected using write barriers, like in the semi-incremental version of the algorithm.

Algorithm 9 Fully-Incremental Algorithm - Snapshot Creation

```
1  procedure TakeSnapshot
2      snapshot ← ∅
3      for all x in RcGcObjList do
4          snapshot_refs ← ∅
5          for all y in References(x) do
6              if IsGc(y) or IsLinked(y) then
7                  Add y to snapshot_refs
8          x_snap ← LightweightCopy(x, snapshot_refs)
9          Add x_snap to snapshot
10     for all x in LinkedRcProxyList do
11         x_snap ← LightweightCopy(x, ∅)
12         Add x_snap to snapshot
```

Snapshot Synchronization

When synchronizing the snapshot, we have to check all unmarked (and potentially dead) objects and see if any reference counts or references have changed. If any reference or reference count has been changed, we have to keep the whole reference counted heap alive and also mark any reachable non-reference counted objects. Otherwise, we can unmark

all potentially dead objects in the object graph (all reference counted objects are marked before synchronization, otherwise their reference count would have been zero and they would have already been freed). However, in both cases, we have to be careful about newly created objects. We have to keep these objects and any object that is reachable from these new objects alive.

Algorithm 10 Fully-Incremental Algorithm - Snapshot Synchronization

```
1  procedure SyncSnapshot
2      consistent ← true
3      for all x in RcGcObjList ∪ LinkedRcProxyList do
4          if InSnapshot(x) then
5              x_snap ← ToSnapshot(x)
6              if not IsMarked(x_snap) then                ▷ Compare references
7                  if RefCount(x) <> OriginalRefCount(x_snap) then
8                      consistent ← false
9                  refs ← References(x)
10                 refs_snap ← References(x_snap)
11                 if Count(refs) <> Count(refs_snap) then
12                     consistent ← false
13                 for i = 0 ... Count(refs) do
14                     if refs[i] <> refs_snap[i] then
15                         consistent ← false
16                 Add x to dead_list
17         else
18             if IsLinked(x) then
19                 y ← LinkedObject(x)
20                 if not IsMarked(y) then
21                     consistent ← false
22     if consistent then                    ▷ Unmark all potentially dead rc objects
23         for all x in dead_list do
24             Set the reference count of x to 0
25     else
26         for all x in LinkedRcProxyList do          ▷ Mark all reachable non-rc objs
27             y ← LinkedObject(x)
28             MarkReachable(y)
```

3.2.3 Semi-Formal Proof of Correctness

Outline

The fully-incremental algorithm essentially performs the same steps as the semi-incremental version. What is left to proof is, that the consistency check of the snapshot integrates well with the write barrier of the non-incremental marking procedure, so that in the end the adapted black-white hypothesis of the tricolor abstraction (see Subsection 2.1.3) holds. Also, we have to show that new reference counted objects, which have been added since the snapshot was taken, are considered respectively. Weak references and finalizers also have to be taken into account, however, like in the proof of the semi-incremental version, we will show afterwards, how they integrate with the proof.

Semi-Formal Proof of Correctness

For the semi-formal proof, we will use the black-white hypothesis from the tricolour abstraction of typical mark-sweep algorithms. We will show, that the black-white hypothesis, no black (= marked) object exists, which points to a white (= unmarked) object, holds in our situation. Afterwards, we can reduce the proof of this hypothesis and two additional proofs to the proof, that no live object might be freed.

Theorem 2 *Black-white hypothesis: No marked object exists which points to an unmarked object.*

Proof (indirect): There exists a marked object which points to an unmarked object.

Case 1
There exists a marked non-reference counted object which points to an unmarked non-reference counted object:
$\exists x, y.(marked(x) \wedge non\_rc(x) \wedge ref(x, y) \wedge \neg marked(y))$

$\perp$ The write barrier prevents this situation.

Case 2
There exists a marked reference counted object which points to an unmarked reference counted object:
$\exists x, y.(marked(x) \wedge rc(x) \wedge ref(x, y) \wedge \neg marked(y))$

$\perp$ Inconsistent snapshot (reference count not matching).

Case 3
There exists a marked reference counted proxy which is linked to an unmarked non-reference counted object:
$\exists x, y.(marked(x) \wedge link\_rc(x, y) \wedge \neg marked(y))$

$\perp$ We add linked non-reference counted objects to the working set, during our marking phase. In case we encounter newly created/linked reference counted proxies when we synchronize the snapshot, we treat the snapshot inconsistent .

Case 4

There exists a marked non-reference counted proxy which is linked to an unmarked reference counted object:
$\exists x, y.(marked(x) \wedge link\_nonrc(x, y) \wedge \neg marked(y))$

$\perp$ We keep checking the snapshot for such links and as long as they exist, continue the marking phase. Directly afterwards, we synchronize the snapshot.

Case 5

There exists a marked non-reference counted object which is linked to an unmarked reference counted object:
$\exists x, y.(marked(x) \wedge link\_rc(y, x) \wedge \neg marked(y))$

$\perp$ At the end, after the snapshot has been synchronized during *MarkProxies*, all links are iterated and all unmarked reference counted proxies are marked, in case the non reference counted object is marked.

Theorem 3 *No unmarked non-reference counted root, or unmarked reference counted object which is referenced by a live non-garbage collected object exists.*

Proof (indirect): There exists an unmarked non-reference counted root, or an unmarked reference counted object which is referenced by a live non-garbage collected object.

Case 1

There exists an unmarked non-reference counted root: $\exists x.(root(x) \wedge \neg marked(x))$

$\perp$ Non-reference counted root detection is executed at the beginning of each increment.

Case 2

There exists an unmarked garbage collected reference counted object which is referenced by a live non-garbage collected object:
$\exists x, y.((\neg gc(x) \wedge gc(y) \wedge ref(x, y)) \supset \neg marked(y))$

$\perp$ Old objects: inconsistent snapshot (reference count not matching), newly added objects: see proof below.

Case 3

A linked, non-garbage collected reference counted object with an external reference from a non-garbage collected object is unmarked:
$\exists x, y, z.((\neg gc(x) \wedge \neg gc(y) \wedge ref(x, y) \wedge link\_rc(y, z)) \supset \neg marked(y))$

$\perp$ Old objects: inconsistent snapshot (reference count not matching), newly added objects: see proof below.

Theorem 4 *Objects, which have been added since the last snapshot, and all objects reachable from them, are kept alive.*

Proof We distinguish between four cases:

Case 1

Non-reference counted objects:

Newly added non-reference counted objects are detected by the write barrier, in case they were added to the object graph via an existing object, or otherwise detected during the root collection at the beginning of each increment. We continue the marking phase, as long as all new objects and objects reachable from those objects have been marked.

Case 2

Reference counted proxy:

We enumerate the list of all reference counted proxies when synchronizing the snapshot. If we encounter an object that is not in the snapshot, we keep it marked. In case the non-reference counted object is marked, all reachable objects are already marked. In case it is not marked, we treat the snapshot inconsistent. In this case we will keep all reference counted objects marked (new and old). Then, at the end of the synchronization, we would mark all reachable non-garbage collected objects.

Case 3

Reference counted garbage collected objects (except proxies):

We enumerate the list of all reference counted garbage collected objects when synchronizing the snapshot. If we encounter an object, that is not in the snapshot, we keep it marked. In case it references any unmarked garbage collected object, the reference count of this object must have been increased. This would result in an inconsistent snapshot, in which case we will keep all reference counted objects marked (new and old). Then, at the end of the synchronization, we would mark all reachable non-garbage collected objects.

Case 4

Reference counted non-garbage collected objects (except proxies):

We do not directly process such objects during our garbage collection procedure, so the objects themselves will always stay marked and alive, as long as they are connected to the object graph. In case they point to an unmarked reference counted object (garbage collected or linked), they would increase their reference count. This would result in an inconsistent snapshot, in which case we will keep all reference counted objects marked (new and old). Then, at the end of the synchronization, we would mark all reachable non-garbage collected objects.

If we now combine the three proofs presented above, we can conclude, that no live object will be freed: All newly added objects are marked, all roots are marked and no marked object points to an unmarked object, which means all live objects must be marked before our sweeping phase.

Weak References and Legacy Finalizers

Weak references and legacy finalizers are handled in exactly the same way as in the semi-incremental version. Legacy finalizers are processed non-incrementally to keep the rest of the algorithm as simple as possible, as their usage is probably declining and in most application, they should have been replaced by modern finalizers anyway. It should not be too hard to see, that this not change the correctness of the algorithm.

Modern Finalizers

In the non-incremental version of the algorithm we ended up with a separate list of objects, if some objects about to be sweeped had modern finalizers. We processed this list during the following collection cycle, after all modern finalizers had been called.

Now, instead of directly processing the objects in this list during root detection and marking, we simply include them in the snapshot. If we encounter a reference from the rest of the object graph to an object in the separate list, we join both sets (some objects have probably been resurrected), otherwise we keep the list separate. If we have managed to keep the objects separate, then, when we check the consistency of the snapshot, we can treat those objects like a separate subset.

If the separate list is still consistent with the snapshot, we can sweep these objects without further ado, even if the rest of the dead objects from this collection cycle is inconsistent with the real graph. We can safely do this, because we can be sure that the objects from the separate list are not reachable from the rest of the potentially dead objects, otherwise we would have detected such a reference, during the creation of the snapshot, or we would have marked linked objects after the synchronization of the snapshot, when we mark non-reference counted objects reachable from finalizers.

CHAPTER 4

Implementation

The algorithms presented in the last chapter, were implemented in PyPy, in order to compare their performance, as described in the next chapter. The implementation can be found in the official repository: `https://foss.heptapod.net/pypy/pypy/tree/branch/cpyext-gc-cycle`. In this chapter, we will describe the changes to PyPy in detail and also present, how the correctness of the implementation was verified.

4.1 PyPy Architecture

To understand how the algorithms are implemented in PyPy, we first have to explain the architecture of PyPy and how the existing cpyext code integrates with this system.

4.1.1 RPython

PyPy is written in RPython. RPython is a language and an infrastructure for writing virtual machines for dynamic languages. The name RPython derives from the fact, that the RPython language is a restricted subset of Python. The RPython compiler itself is written in Python. This is where the name PyPy comes from (Python written in RPython written in Python), but it does not change the fact, that RPython can be used to write virtual machines for any dynamic language.

From a standpoint of a language developer who uses RPython to implement a new virtual machine for a language, it is only neccessary to implement an interpreter in RPython. The RPython subsystem then translates this interpreter statically into C code and utilizes a C compiler to compile it into machine code. In fact Python is a meta-programming language for RPython, which means the RPython compiler does not need to interpret RPython programs, but instead uses a Python interpreter to include RPython programs and uses the internal interpretation of the loaded Python program to generate the C code.

What makes the RPython infrastructure special, compared to more traditional toolchains, is that it does not only generate an interpreter, it also integrates a tracing JIT compiler into the executable. This JIT compiler is triggered under certain circumstances during interpretation. It then traces the execution of the interpreter, during the interpretation of the program, by recording its execution, using an additional internal representation of the generated machine code. By doing this and by the nature of tracing JIT compilers (and specific design choices in combination with clever optimizations), the RPython subsystem is essentially compiling the trace of the interpreted program. This enables developers to rapidly develop JIT compilers for any language, with minimal effort compared to traditional approaches.

4.1.2 Garbage Collector

To keep things simple for the language developer, the garbage collector and the memory system are implemented in RPython and do not need to be implemented specifically for the target language. However, this makes it somewhat difficult to implement garbage collection for CPython extension modules. Objects in those modules are externally controlled by C code, which means the memory occupied by those objects cannot simply be managed by RPythons memory system.

4.1.3 Rawrefcount extension

Fortunately, we are able to modify the garbage collector and implement template methods, so we can access those externally controlled objects from RPython. Similar code already exists for the cpyext module in RPythons *rawrefcount* extension. This code needs to be extended, to be able to access all the neccessary information from reference counted heap. Unfortunately, this means that we have to introduce some language specific concerns in the otherwise language agnostic garbage collector. For the sake of this thesis, this is an acceptable compromise, however for future implementations, a more general approach would be desirable. Probably the biggest challenges here would be, to cope with different styles of memory management of the externally controlled objects (reference counting vs. tracing; explicit vs. implicit external references; etc.) and still maintain acceptable performance.

4.1.4 Cpyext module

Cpyext is implemented as a module for PyPy, written in Python, RPython and C. It bridges the gap between CPython extension modules written in C and PyPy's representation of a Python program. It also makes RPython's garbage collector aware of the objects managed by C code. However, the existing implementation of the object management is very minimal and only prevents premature collection of potentially live objects (on both heaps). Objects managed by C are only freed, if their reference count drops to zero, which means cycles consisting of at least one such object are never collected. PyPy objects referencing such objects add an additional artifical reference count, to prevent

them from being freed, as their actual reference count might otherwise be zero. As the cpyext module was originally created to add rudimentary support for those modules, the restrictions were reasonable in order to keep the implementation simple. However, for a more sophisticated support, this implementation needs to be extended.

4.2 Extending Rawrefcount

The implementations of our algorithms are primarily located alongside the code of RPython's garbage collector, encapsulated in three interchangable classes (two for each implementation and one for a dummy, that mimics the existing behaviour). However, as we need to interoperate with CPython's memory model, several other modules also needed to be adapted.

4.2.1 Extending the GC Interface

To control the extended behaviour of the garbage collector in certain situations (e.g. when the process is forked) and to be able to access reference counted objects that need to be processed in the implemented language (finalizers, etc.), the garbage collector now offers additional methods, that can be invoked by language developers.

To enable interoperation between RPython's garbage collector and CPython's garbage collection infrastructure, the garbage collector mainly needs access to two pieces of code: CPython's tp_traverse method and CPython's global list of garbage collected objects. Additional methods are required, to convert between the two object representations. For more advanced integrations, three interfaces have also been added: a method to query the finalization type of objects, a pointer to the list of newly allocated tuples and a method to determine, if a tuple might be or become part of a reference cycle.

All of the above methods and lists are injected into RPython's garbage collector when the cpyext module is loaded, alongside the already existing methods for the rudimentary GC support. Currently, the extended interface is only implemented in the default garbage collector incminimark. In the future, other garbage collectors might also be supported. To be able to pass the required methods to the garbage collector, several other parts of the RPython framework and PyPy needed to be extended.

4.2.2 Extending cpyext

With regards to garbage collection, cpyext is responsible for the initialization of the internal structures of CPython's garbage collection infrastructure. It also needs to provide an implementation for the GC template methods, either by directly implementing the medoths in RPython or by delegating the calls to pieces of C code.

4.2.3 Extending the Testing Framework

To be able to test all common scenarios and their corner cases conveniently, it was neccessary to extend the testing framework. The main issue with the way the existing tests for the garbage collector were written, was that they were relatively long for what they were trying to achieve and also quite hard to read and understand. As we need more complex test cases, to test our implementation and writing those tests in the existing way proved to be quite tedious and error prone, creating such tests had to be simplified. This lead to the idea of an abstraction, where creators of such test cases, do no longer need to worry about the internals of the testing environment. It seemed to make sense to integrate a language, to describe those test cases. After some research, the extensible DOT graph description language seemed to be a good fit to describe object graphs, spanning across both heaps [dota].

DOT graph description language

The DOT graph description language is a simple text-based language to describe directed and undirected graphs. Attributes might be added to nodes or edges to describe their graphical representation in those graphs. The language itself does not define a fixed set of attributes, which means additional attributes might be defined, like several popular tools for DOT graphs already do. Those new attributes do not neccessarily need to influence the graphical representation, but they might encode arbitrary metadata. We can make use of this, to encode additional information needed for our tests. Low-level parsers, which already exist for Python, will be able to handle those attributes, so we do not need to write our own parsers. As most tools will ignore unknown attributes, we can use them to generate, edit or visualize our custom DOT graphs, that we can annotate at any time with our custom attributes, either from Python code or in a text editor. This enables a lot of possibilities, like automated test-case generation or test case visualization, which we would not have, if we would write those tests manually.

Semantic Extensions of the DOT language

The following attributes were added to the nodes of the graph, to implement our test cases:

- *type* (mandatory)
  - P = PyPy object
  - B = linked object at the border
  - C = CPython object
- *alive* (mandatory)
  - y = object is alive and should not be collected
  - n = object is dead and should be collected
- *rooted* (optional; for PyPy or border objects)

  - y = object is in PyPy's root set
  - n = object is not in PyPy's root set (default)

- *ext_refcnt* (optional; for CPython or border objects):
  positive integer (default 0): additional reference count, resulting from references external to the managed set of objects

- *tuple* (optional; for CPython or border objects)
  - y = object is a tuple and should be added to the tuple_list
  - n = object is not a tuple and will be added to the global list of objects (default)

- *tuple_type* (mandatory for tuples)
  - 0 = tuple contains only primitive types and should be untracked
  - 1 = tuple contains gc types and should be promoted to the global GC list
  - 2 = tuple is not initialized yet and should be kept in tuple list

- *gc* (optional; for CPython or border objects)
  - y = object is garbage collected (default)
  - n = object is not garbage collected

- *tracked* (optional; for garbage collected objects)
  - y = object is tracked in the global list of objects or list of tuples (default)
  - n = object is not tracked in any list

- *finalizer* (optional; for CPython or border objects)
  - modern = object has a modern finalizer
  - legacy = object has a legacy finalizer
  - (empty) = object has no finalizer (default)

- *resurrect* (optional; for objects with a modern finalizer)
  name of the node (object) that should be resurrected, when the finalizer of this object is executed

- *delete* (optional; for objects with a modern finalizer)
  name of the node (object) to which the edge (reference) should be removed, when the finalizer of this object is executed

- *garbage* (optional)
  - y = the object should have been added to gc.garbage after the collection, because it is kept alive by a dead object with a legacy finalizer
  - n = the object should never be added to gc.garbage (default)

- *added* (optional; for fully-incremental algorithm)
  - after_snap = the object is created, after the snapshot has been created

  - linked_after_snap = a linked proxy for the object is created, after the snapshot has been created
  - (empty) = the object exists prior to the first collection (default)

- *removed* (optional; for CPython or border objects; for fully-incremental algorithm)
  - after_snap = the object will be removed from the graph, after the snapshot has been created, because its reference count drops to zero
  - (empty) = the object will not be removed (default)

The following attributes were added to the edges of the graph:

- *weakref* (optional; for references on the reference counted heap)
  - y = reference is a weak reference and does not increase the reference count
  - n = a regular reference which increases the reference count (default)

- *added* (optional; for fully-incremental algorithm)
  - after_snap = the reference was added to the graph, after the snapshot has been created
  - (empty) = the reference was created prior to the first collection (default)

- *removed* (optional; for fully-incremental algorithm)
  - after_snap = the reference will be removed from the graph, after the snapshot has been created
  - (empty) = the reference will not be removed (default)

Note, that the alive attribute could be inferred from the the topology of the graph and the other attributes. However, this is not done during the execution of the tests for several reasons, but rather lies in the responsibility of the creator of the test case.

Integration into the Testing Framework

The new DOT tests were integrated as data driven tests into the existing pytest-based testing framework. DOT files are discovered automatically from the respective folder and the results are displayed individually in the test results. Some sanity checks are automatically performed prior to the actual test, to ensure that the files are correctly parsed and the testing framework works as expected.

4.3 Semi-Incremental Implementation

This implementation is based on a combination of PyPy's generational garbage collector and CPython's generational backup tracing algorithm. As it is simpler than the fully-incremental implementation, common solutions, like the abstraction of reference counted objects, are described here, for better understanding.

4.3.1 Abstraction of reference counted objects

Architecturally, one of the goals was to implement the entire algorithm for reference counted objects in a more or less language agnostic way. This is because of the fact, that the garbage collector of PyPy is implemented in the RPython subsystem and the goal was to introduce as little PyPy specific concerns in the underlying system as possible. This would imply that the implementation, can also be used for other runtimes or programming languages that face similar issues. However, it is questionable, how many other languages will need this rather specific interface in practices. Nervertheless, it is not necessary to include any Python-specific code to create JIT compilers for other languages, especially if the interface is not even used.

On a detailed level, the following changes have been made to the existing RPython interface for reference counted object support:

Lists

Two lists are now passed to the garbage collector upon initialization of the reference counted subsystem:

- Global list of objects: This is a list where all reference counted objects are tracked, which might be part of a reference cycle. This list is similar to the lists used in CPython's garbage collector, except there is only one list and not several lists for each generation of objects, like in CPython.
- Tuple list: This list contains all newly allocated tuples. This is an optimization for immutable objects (see Subsection 4.3.2).

Methods

The following methods have been added to the internal interface of the garbage collector, which can be used by language developers:

- *next_cyclic_isolate*: Returns the next object of a dead cycle, that should be cleared. If the method returns null, all objects have been cleared and, as a result, should have been swept from the heap.
- *deactivate_rawrefcount_cycle*: Deactivate the cycle detection algorithm temporarily and treat all reference counted objects, as if they were alive. This is for example used in PyPy during the initialization of a forked process, before the cpyext subsystem is reinitialized, as this might otherwise lead to crashes or unexpected behaviour.
- *activate_rawrefcount_cycle*: Reactivates the cycle detection.
- Finalization support (see Subsection 4.3.4):
  - *cyclic_garbage_head*: Returns the first object of the list of cyclic isolates, that might contain finalizers.
  - *cyclic_garbage_remove*: Removes object returned by cyclic_garbage_head.

- *next_garbage_pypy*: Returns the next non-reference counted object, which should be added to the garbage list.
- *next_garbage_pyobj*: Returns the next reference counted object, which should be added to the garbage list.
- *begin_garbage*: Temporarily disables parts of the code in the cycle detection algorithm. This can be used during the construction of the garbage list, to prevent unexpected behaviour.
- *end_garbage*: Reenables the respective parts of the code.

Template Methods

To call language-specific code from the GC, the following template methods have been added:

- *pyobj_as_gc*: Return the GC header of the reference counted object.
- *gc_as_pyobj*: Return the reference counted object belonging to the GC header.
- *traverse*: Call the passed callback on every object, the reference counted object references.
- *finalizer_type*: Return the type of finalizer, the object implements (see Subsection 4.3.4).
- *clear_weakref_callback*: Clear the callback(s) when a weak reference to this object dies (see Subsection 4.3.5).
- *tuple_maybe_untrack*: Remove the tuple from the list of tuples, if possible (see Subsection 4.3.2).

4.3.2 Tuples

Tuples are immutable in Python, but they might still be part of a reference cycle. As we have already described in the previous subsection, there exists a list for newly allocated tuples, which is made available to the garbage collector, as well as a hook to remove tuples from this list. This is because some tuples might never become part of a reference cycle (e.g. tuples of integers) and can therefore be ignored in future collection cycles, which most likely improves the overall performance of our garbage collector.

Tuples are not directly added to the global list when they are allocated, but are instead added to the special list described above. During the next collection cycle, only tuples which might be part of a cycle (tuples that contain tuples, or other objects that might be part of a cycle) are moved to the global list. All other tuples are removed from this list, as they will never cause a reference cycle. The tuple_maybe_untrack hook is used to perform this task, which means the logic of deciding which tuples to untrack is left up to the language developer, in case this interface is reused for other languages.

In CPython's garbage collector this optimization is implemented, by promoting only tuples that might be part of a cycle into the second generation. As our garbage collector

does not use generations to organize reference counted objects, we adapted this approach as described above, to fit our needs.

4.3.3 Clearing Cycles

Dead cyclic structures are detected by our algorithm as described in Section 3.1. However, they are not directly removed from the heap during the sweeping phase of our mark-sweep collector, but instead moved to a set, which is processed by cpyext (using the next_cyclic_isolate method). This is done outside of the scope of the garbage collector, because it is not safe to call the deallocation routine from the context of the garbage collector, as it might try to execute interpreted (or JIT compiled) Python code, which is not supported. The respective cpyext routine is executed asynchronously after an increment of a garbage collection cycle and simply calls the next_cyclic_isolate method in a loop. As long as objects are returned, it increments their reference count for safety, then calls their *tp_clear* method and then decrements their reference count again. At the end of a collection cycle, once the last object has been processed in this way, this should have deallocated all objects in cyclic structures that were detected, because their reference count should have dropped to zero.

4.3.4 Finalization

There are two types of finalizers in Python, which are supported by the cycle detection implementation: modern and legacy finalizers. Modern finalizers (*tp_finalize*) have been introduced by PEP 442 [pep] and are the successor of legacy finalizers (*tp_del*), which have been deprecated with this proposal, but are still supported by current Python implementations. To provide backwards compatibility and to show that our algorithm is compatible with all types of finalizers, we decided to implement both of them.

Modern finalizers

Our implementation is able process modern finalizers, as described in Subsection 3.1.3. If the garbage collector encounters objects with modern finalizers (using the finalizer_type hook), all cyclic isolates are moved to a separate collection. In this case, instead of being cleared by cpyext, all finalizers are called in the same asynchronous routine, using the cyclic_garbage_head and cyclic_garbage_remove methods and calling the tp_finalize slot of the returned objects. Then, during the following collection cycle, the set is processed again by the garbage collector. If the set is still unreachable from outside, it is moved back to the set of dead objects and processed as described in Subsection 4.3.3. Otherwise, it is moved to the global list of objects and needs to be processed again.

Legacy finalizers

Legacy finalizers are not save to call on cyclic structures, so they should never be called by our garbage collector, not even in our asynchronous cpyext routine. Therefore, they are moved to a list of garbage objects, accessible to the application developer (gc.garbage),

including all of the objects they keep alive. From there, application developers are able to free these objects, by manually breaking their cycles.

During a collection cycle, the garbage collector determines the set of objects with legacy finalizers, using the finalizer_type hook. To determine the complete set of objects they keep alive, reference counted objects are traced using the traverse hook and moved to a separate list. Non-reference counted objects are also traced and all unmarked objects are marked with a new flag (GCFLAG_GARBAGE). This tracing process is continued across heap boundaries, in case linked objects are encountered, until all reachable objects have been processed.

After the collection cycle, instead of clearing those objects, the cpyext routine retreives those objects by using the next_garbage_pypy and next_garbage_pyobj methods. They are then appended to the gc.garbage list. This part of the cpyext routine is enclosed by begin_garbage and end_garbage calls, so the garbage collector does not interfere. This is necessary, as the garbage collector might be triggered during this routine, as we will be creating linked proxy objects for each reference counted object which does not already have a linked object.

4.3.5 Weak References

Weak references in cpyext are implemented as linked proxies to weak referencex in PyPy. This might seem like an unnecessary overhead, but it implicitly solves a problem, that would otherwise need special attention. CPython's garbage collector needs to be especially aware of weak reference callbacks in cyclic structures: Python offers a way for application developers to be notified, when weakly referenced objects die. When sweeping cyclic structures, the respective clear methods are called and some weakly referenced objects might die during this call. This is an issue, because then these callbacks are also called and some of them might resurrect objects which are not yet cleared. In CPython this is solved, by detecting and clearing such callbacks, before clearing the cyclic structures. In PyPy, weak references (and their callback) always die before the cyclic structures are cleared, so problematic callbacks do not need to be cleared beforehand. This is because the actual weak references are, as mentioned in the beginning of this subsection, non-reference counted objects. Those non-reference counted objects will be sweeped after the marking phase of the garbage collector, because the proxy that kept them alive is about to be cleared by cpyext. As cpyext clears the respective cyclic structure after the corresponding sweeping phase, it does not need to worry about any problematic callbacks.

4.4 Fully-Incremental Implementation

The fully-incremental implementation shares a lot of the code with the semi-incremental implementation. The main difference, not only algorithmically but also implementation-wise, it that the marking phase is executed on a snapshot of the heap. This implies, that a snapshot has to be created and this snapshot has to be synchronized. We will explain

how those methods have been implemented and how this implementation relates to the semi-incremental implementation. The algorithmic differences were already described in Section 3.2.

4.4.1 Snapshot Creation

The snapshot is created in the last increment of the default marking phase of the non-reference counted heap, directly after all objects, that are reachable from all non-reference counted roots, have been marked. First, two large memory buffers to save the snapshot are allocated. One memory buffer is used to store a light copy of the objects themselves and the second memory buffer is used to store the references between those objects. References are stored in an array of raw adresses, to the location of the object in the buffer.

The fields of the buffer used for the light copy look as follows:

- *pyobj*: the address of the original object or proxy
- *pypy_link*: the address of the original, non-reference counted object or proxy
- *status*: the marking status of this object
- *refcnt_original*: the original reference count, which is used to check if the references to the objects have been changed, when the snapshot is synchronized at the end of the marking phase
- *refcnt*: a field for the refence count, that is used to calculate the roots of the graph, which is initially set to the original reference count
- *refs_index*: the first index of the outgoing references in the array of references
- *refs_len*: the number of outgoing references

The snapshot includes all objects in the global list of objects (tuples are eventually moved to the list beforehand), all non-garbage collected linked proxies (as they might also be part of a cycle) and all cyclic isolates from the last cycle (for details see Section 3.2.2).

All of these objects need to be scanned four times in total, in order to create the snapshot. This is facilitated in three steps:

1. All objects are iterated and their count as well as the sum of their reference count are calculated (first scan).

2. Light copies of all objects are created in the memory buffer, one after another (second scan). For each object, all outgoing references are copies to the second memory buffer (third scan). They initially point to the original object and will be remapped in step two, by (temporarily) storing the index of the object in the snapshot in a field of the original object. The content of this field is backed up in a dictionary, in case it needs to be restored afterwards (only needed for linked objects).

3. References are remapped, so they point to the copy of the object in the snapshot (fourth scan). Also, the original content of the fields that have been used for the object's index is restored.

Non-garbage collected proxies are handled like garbage collected objects, with regards to the creation of the snapshot, except that index of the object in the memory buffer of the snapshot is temporarily stored in the link field.

Other non-garbage collected objects should not be added to the snapshot, which means we have to ignore such objects, in case we encounter them during tracing. This is quite easy, because their link field is typically zero, except non-garbage collected objects that are linked to a non-reference counted proxy. To exclude these objects too, we set their link fields to zero and temporarily store it in a dictionary, before we start tracing the object graph. We restore the previous state, after the snapshot has been created.

4.4.2 Marking Phase

The marking phase is divided into several sub-phases, each of which will be executed in at least one separate increment:

1. Root Determination: all internal reference counts are subtracted from the refcnt field of the objects within the snapshot.

2. Root Collection: the snapshot is scanned for objects which have a refcnt greater than zero. All of those objects are added to a working set.

3. Incremental Marking: at the beginning of an increment, newly added reference counted proxies are added to a separate working set (one for non-reference counted objects). Then both working sets are processed. When an object is marked, outgoing references and linked objects are traced in both cases and unmarked objects are added to either one of the working sets. First, reference counted objects are marked, then non-reference counted objects are marked. If the working set for reference counted objects increased during the second step, the process is repeated. Once the incremental limit is reached, the increment is over and the working sets are kept for the next increment. If the working sets are empty before the incremental limit is reached, the marking phase is finished. However, the increment is not finished, as we have to synchronize the snapshot in the same increment.

4.4.3 Snapshot Synchronization

The snapshot is synchronized in the last increment of the marking phase, directly after all reachable objects have been marked. To ensure safety, we keep all objects, that have been added after the snapshot was created. We want to sweep all objects, which are not marked reachable, but only in case the snapshot is still consistent with the real graph.

To achieve this, we use the following strategy: we unmark all objects, that still exist in the original graph (some could have already died as a result of a reference count dropping to zero). Also, we check if the reference count and all outgoing references are still matching. If we encounter an inconsistency, we stop the synchronization and mark all objects again, that we have just unmarked. If the snapshot is consistent, we free all unmarked objects afterwards. In detail, this works as follows:

1. We sync all non-garbage collected proxies, by searching for such objects in the snapshot. By ensuring that these objects are always at the beginning of the snapshot when creating it, we can stop looking for them, after we have found an object that is not a non-garbage collected proxy. We can be sure, that those objects still exist on the real graph, as they are only freed by the garbage collector and never as a result of a reference count dropping to zero. This is, because they always have an artifical reference count as a result of their link to a non-garbage collected object. Afterwards, we scan the current list of non-garbage counted proxies and look for non-garbage collected proxies, where the non-reference counted object is unmarked. In this case, we have to abort the synchronization, as this is also an inconsistent state (it seems that new proxies have been added, which were not considered during the marking phase).

2. All garbage collected are synchronized, by iterating over the current list of garbage collected objects. We can distinguish between new and old objects, because only objects that existed when the snapshot was created have a valid reference to the snapshot in their cyclic reference count field. New objects are simply kept alive. In case they are linked to a non-reference counted object and this object is unmarked, we abort the synchronization, as we did for non-garbage collected object. Old objects are checked for consistency and eventually, potentially dead objects are moved to a separate list.

3. Isolates from the last collection are synchronized. We also perform this step, if the rest of the snapshot was inconsistent. In case this subset is consistent, we can still make some progress and free these objects (and vice-versa). As there can be no newly added objects in this case, this is rather simple and basically works like the synchronization for all other garbage collected objects. Potentially dead objects are moved to a separate list for dead isolates.

If the snapshot was consistent, we can simply proceed to the sweeping phase, as all dead objects should be unmarked by now. If it was inconsistent, we move all objects which have already been moved to separate lists back to the actual lists and mark all objects, which have not yet been processed. Then, we continue to the sweeping phase, which should not free any potentially live object. Objects in cyclic isolates from the last collection are handled in the same manner.

4.5 Verification

To verify, that the implementation is consistent with the algorithm and that all premises, which the proof of the algorithm assumes, are fulfilled, we systematically employ unit test. To ensure that all premises are fulfilled, we condense them into a list of testable units, which are verified using the existing testing environment. To ensure that the implementation is consistent with the described algorithm, we use equivalence partitioning. First, we describe those partitions and why the input data in the partitions behaves equally. Then, we present one exemplary test case for each of the partitions, which we have implemented and verified using our testing environment, described in Subsection 4.2.3.

4.5.1 Premises

The semi-formal proofs of correctness in Subsections 3.1.3 and 3.2.3 implicitly or explicitly assume several premises. To check if those premises are fulfilled in practice, we created a list of them and implemented a unit test for each of them. Note, that we have not included the already existing (and tested) premises of the implemented mark-sweep algorithm, as we assume that those premises are valid. Below you can find a list of these premises, with the name of the test(s) which can be found in the following files: rpython/rlib/test/test_rawrefcount.py, pypy/module/cpyext/text/test_cpyext.py Premises without automatic tests, were verified manually, using trace logs.

1. Garbage collected, reference counted objects are added to the global list of objects, when they are created: test_gc_track
2. Reference counted tuples added to the list of tuples, when they are created: test_gc_tuple_track
3. Tuples are correctly tracked or untracked, depending on their content in the tuple_maybe_untrack method: test_gc_tuple_track
4. An artificial reference count is added, when objects are linked: test_create_link_pyobj
5. Objects are unlinked, when their proxy dies: test_collect_p_dies, test_collect_o_dies
6. (Non-)reference counted proxies are added to the set of (non-)reference counted proxies, when they are linked: test_create_link_pyobj, test_create_link_pypy
7. Objects in the list of dead, cyclic objects are sweeped.
8. Objects in the list of dead, non-cyclic objects are sweeped.
9. All template methods described in Subsection 4.3.1 work as specified.
10. Modern finalizers are eventually called.
11. Objects, reachable from isolated objects with legacy finalizers, are added to the gc.garbage list.

4.5.2 Evquivalence Partitioning

To systematically verify that the implemented algorithm is consistent with the semi-formally proven description of the algorithm, we apply equivalence partitioning. Equivalence partitioning assumes, that out of the infinite set of possible inputs, all inputs in a partition behave in exactly the same way. According to this assumption, it is sufficient to provide a test for one single input data for each partition, in order to verify the correctness of the whole partition. If we work with multiple dimensions, we can combine those dimensions, to cover several of them into one single unit-test. Once all partitions in all dimensions are tested, we have verified the correctness of the whole implementation. The dimensions of our implementation are listed below:

1. Non-reference counted objects: y/n
2. Tracked, garbage collected, reference counted objects: y/n
3. Untracked, garbage collected, reference counted objects: y/n
4. Non-linked, non-garbage collected, reference counted objects: y/n
5. Cross references:
   a) No cross references
   b) At most one reference counted proxy per subgraph
   c) At most one non-reference counted proxy per subgraph
   d) More than one proxy in at least one subgraph
6. Unreachable cycles, including reference counted objects: y/n
7. Reachable cycles, including reference counted objects: y/n
8. Non-cyclic subgraph, including reference counted objects: y/n
9. Reference counted tuples, containing only primitive values: y/n
10. Reference counted tuples, containing garbage collected objects: y/n
11. Tracked, uninitialized reference counted tuples: y/n
12. Modern finalizers:
    a) No modern finalizers
    b) Only reachable rc objects
    c) Mixed reachable objects
13. Legacy finalizers:
    a) No legacy finalizers
    b) Only reachable rc objects
    c) Mixed reachable objects
14. Live weak references: y/n

15. Dead weak references within cycles: y/n
16. Dead weak references in non-cyclic subgraphs: y/n

Additional dimensions for fully-incremental implementation:

i1. New reference counted objects added beteween increments: y/n
i2. New non-reference counted objects added beteween increments: y/n
i3. New reference counted proxies created beteween increments: y/n
i4. New non-reference counted proxies created beteween increments: y/n
i5. Non-linked, reference counted objects removed beteween increments: y/n
i6. Reference between non-reference counted objects added between increments: y/n
i7. Reference between non-reference counted objects removed between increments: y/n

The test cases (dot tests), which altogether cover all partitions, are listed below. The respective paritions each test case covers are also included (duplicates are omitted):

A) *free_cross_multi_2c*: 1y, 2y, 3n, 4n, 5d, 6y, 7n, 8n, 9n, 10n, 11n, 12a, 13a, 14n, 15n, 16n, i1n, i2n, i3n, i4n, i5n, i6n, i7n
B) *keep_cpython_self*: 1n, 5a, 6n, 7y
C) *keep_pypy_self*: 2n
D) *keep_cpython_untracked*: 3y
E) *keep_nocycle_nogc*: 4y, 5b, 8y
F) *keep_cross_simple_1a*: 5c
G) *free_cpython_tuple_2*: 9y
H) *free_cpython_tuple_1*: 10y
I) *free_cpython_tuple_3*: 11y
J) *free_finalizer_simple_2*: 12b
K) *free_finalizer_simple_1a*: 12c
L) *garbage_cpython_simple_1*: 13b
M) *garbage_cross_simple_1*: 13c
N) *free_cpython_weakref_simple_4*: 14y
O) *free_cpython_weakref_simple_1*: 15y
P) *keep_cpython_weakref_simple_1*: 16y
Q) *keep_cpython_inc_2*: i1y, i2y
R) *keep_cpython_inc_3*: i3y
S) *keep_cpython_inc_5*: i4y

T) *keep_cpython_inc_6*: i5y, i7y

U) *keep_cpython_inc_1*: i6y

CHAPTER 5

Results

In this chapter, we will take a look at the results of several benchmarks, to compare the implementations of the algorithms, that we described in the previous chapters. Most of the benchmarks were specifically created to test the characteristics of our algorithms and their implementations, so a description of the benchmarks will be given first. Then, we will look at the results of the benchmarking process and interpret our findings.

5.1 Benchmarks

Our benchmarks are going show the differences in performance of our implemented algorithms with regards to memory usage, execution time and pause times. We execute artificial microbenchmarks and real-world application benchmarks, which make heavy usage of garbage collected, reference counted objects. The benchmarks are executed on a machine with an Intel(R) Core(TM) i7-8550U CPU processor with four 1.80GHz cores and enabled hyper threading, 16GB DDR3 Dual-Channel-RAM on an Ubuntu 18.04.3 LTS operating system. All benchmarks are executed repeatedly, to ensure the validity of our results. For every execution a separate process is started, to avoid variation due to the warmup behaviour of the JIT.

We execute each benchmark in four environments on this machine:

- *Base*: the existing implementation (no cycle detection of reference counted objects)
- *Mark*: the implementation of the semi-incremental algorithm
- *Inc5k*: the implementation of the fully-incremental algorithm, with an incremental limit of an equivalent of 5,000 reference counted objects for cross-heap marking
- *Inc50k*: same as Inc5k, with an incremental limit of 50,000

5.1.1 Microbenchmarks

In our microbenchmarks we measure the peformance of our implementations in several, specially created scenarios. Those scenarios include creating (many, very large, etc.) reference counted cycles, cross-heap cycles, and cycles with non-cyclic cross references. By comparing the results to the current implementation, which cannot detect most of the constructed cycles, we will be able to see the impact of cycle detection. We will also take a brief look at some practical issues with cpyext, which might influence our results.

Object layout

The objects on both heaps consist of five fields to store up to five references and of one field to store some values, which will be initialized to a random string of uniform size (1KB). This payload should help us, to make the differences in memory consumption more clear and should reflect the fact, that in reality most memory structures also store valuable data, apart from references to other objects.

Liveliness

We will execute each benchmark in several variants and with different percentages of live objects (0%, 50% and 100%). Depending on the benchmark, we will keep the requested ratio of objects alive, by adding the requested percentage of constructed object graphs to a list or by artificially incrementing the reference count of one of the root objects. To see how big the benefits with regards to memory consumption can possibly be, we keep 0% of the objects alive. To make the overhead of cycle detection more visible, we keep 100% of the objects alive. For a more balanced view, we additionally execute the benchmarks with an equal number of live and dead objects, by keeping 50% of the objects alive.

Simple Cycle

In this benchmark we create X number of cycles within the reference counted heap. The cycles consist of linked lists of length Y and they keep Z non-garbage collected objects alive. We execute this benchmark in three variants:

1 *count*: a lot of cycles (X = 2^{14}, Y = 10, Z = 10)

2 *size*: very long lists (X = 10, Y = 2^{14}, Z = 10)

3 *nongc*: a lot of non-garbage collected objects (X = 10, Y = 10, Z = 2^{14})

Cross-Heap Cycle

Here we also create cycles using linked lists of length X, but with Y number of cross-references. We create Z number of cycles.

1 *size*: very long lists (X = 2^{14}, Y = 10, Z = 10)

2 *cross*: a lot of cross-references (X = 10, Y = 2^{14}, Z = 10)

3 *count*: a lot of cycles (X = 10, Y = 10, Z = 2^{14})

Cross-Heap References

In this microbenchmark, the cycles reside purely on one side of the heap, but they keep objects on the other side of the heap alive. Depending on which heap the cycle resides on, we distinguish between two directions: non-rc to rc (non-reference counted cycle that keeps reference counted objects alive) and rc to non-rc (reference counted cycle that keeps non-reference counted objects alive). For both directions, we create X number of object graphs with Y non-reference counted objects an Z reference counted objects.

1 *count*: a lot of graphs (X = 2^{14}, Y = 10, Z = 10)

2 *nonrc*: a lot of non-rc objects (X = 10, Y = 2^{14}, Z = 10)

3 *rc*: a lot of rc objects (X = 10, Y = 10, Z = 2^{14})

5.1.2 Application Benchmarks

In order to perform meaningful application benchmarks, we need suitable applications, which interact with garbage collected, reference counted objects via the cpyext module. We decided on using UI applications, created on top of GTK for our benchmarks.

GTK [gtk] is a multi-platform toolkit for creating graphical user interfaces. It is written in C and uses GObject as an object system to implement its UI components. It can also be used for Python applications, via the PyGObject [pygb] language binding, which is implemented as a CPython extension module. As there are a lot of open source Python applications, which implement their user interface using PyGObject, like Gnome Music, Gnome Tweaks or PyChess, we can use these applications in order to compare both algorithms and to measure the overhead of cycle detection. To create these benchmarks, the Linux Desktop Testing Project [ldt] is used to automate user interaction.

While there are several other open source applications and libraries implemented as CPython extension modules, like the popular NumPy library for scientific data processing, only very few actually make use of garbage collected reference counted objects. In most cases, the only objects of this kind they use are internal classes like tuples and collections.

Even though we are able to measure the overhead of cycle detection even in those scenarios, this is not the focus of this thesis, as the overhead heavily depends on the implemenation of those internal classes. For example, reference counted lists might be implemented as reference counted objects containing reference counted items, or as proxies to a non reference counted list containing proxies to reference counted items (in cpyext they are currently implemented as proxies to non reference counted lists containing reference counted items, implementing a special traverse method). We face similar issues with dictionaries and tuples. If they would be implemented differently, we would most likely see a different overhead, because they would or would not have already been processed by the existing mark-sweep implementation. We will witness this issue with respect to tuples in Section 5.5.

For an overall performance analysis, this means that we would also have to compare different implementations of those internal classes, but this is outside the scope of this thesis. From a practical standpoint, probably few applications that only use such predefined classes, when it comes to garbage collected reference counted objects, are going to need cycle detection, as the chances are relatively low that cycles are even created (e.g. NumPy).

Gnome Music

Gnome Music [gnob] is a music player, developed for the Gnome desktop environment [gnoa]. It can be used to listen to locally stored music, but it can also be used to stream music over the internet. In our benchmark, we only play locally stored music, as the tested version of PyPy crashes when trying to stream music. The benchmark launches the application with one album present in the library and switches between the different main views (Albums, Artist, Songs, Playlists). It then starts playing some songs from this album, pauses them and switches between songs. The hash of the git commit used for this benchmark is 45612fa.

Gnome Tweaks

Gnome Tweaks [gnoc] is a graphical interface to modify advanced settings of Gnome 3 desktop environments. For example, it can be used to switch between visual themes for applications or icons. In this benchmark, we browse through and change several of these settings, like the scaling factor for the system fonts. The hash of the git commit used for this benchmark is 11d9d1db.

PyChess

PyChess [pyca] is a chess application, where users can play against the computer. It offers several additional features, like an interactive tutorial to teach the user about chess moves. We use a fraction of this tutorial for our benchmark, which will be launched several times during one execution, to simulate several games of chess.

PyGame

PyGame [pyga] is an SDK to develop multimedia applications, like games. It offers several built-in demo scenarios, to showcase its features. We use one rather simple scenarios (testsprite) for our benchmarks, which we will run for about 10 seconds. It animates a fixed number of sprites, which move diagonally across the screen and bounce off the borders of the animated area.

Quod Libet

Quod Libet [quo] is a music playing application, similar to Gnome Music. In contrast to Gnome Music, it offers more advanced configuration options, which can be configured by

the user in a separate dialog. In our benchmark, we perform similar steps as in Gnome Music, but also browse through some of these settings.

5.2 Expected Results

In case all objects stay alive, we would generally expect the Base environment to offer the fastest execution times, with the least memory consumption. In this case all other environments would only introduce additional overhead in terms of processing time and memory, because they would never do anything productive (free memory). Depending on the memory overhead the other implementations introduce, we would at some point expect the Base environment to have the highest memory consuption, if an increasing number of objects in cyclic structures die. But even in this case, we would still expect the Base environment to be the fastest, as we would not have to detect and free cycles. In terms of pause times, we cannot make any precise predictions, except that the pause times of the Mark environment will probably be the highest, in case a lot of objects stay alive, as we would have to process them all in one single increment.

If we compare the Mark (non-incremental) and the Inc5k/Inc50k (incremental) environments, we would obviously expect lower pause times for the incremental environments. However, due to the fact that we have to take a snapshot at the beginning of our incremental collection, which we have to synchronize with our actual object graph at the end of our collection, this might not be the case for all scenarios. Also, if the incremental limit is too high and the whole graph is processed in one single increment, it might result in exactly the opposite effect, considering the additional synchronization overhead. Anyway, we would expect the Mark environment to be faster and to have a lower memory consumption than the incremental environments, as we save space and time because of the lack of a snapshot. Also, in our incremental environments, the snapshot and the working set need to stay in memory between increments, which would also increase the average memory consumption. However, due to the different memory layout of the snapshot, which might be more efficient to process, we could actually gain performance benefits, that might even outweight the costs of taking and synchronizing the snapshot.

5.3 Microbenchmarks

Now we will take a look at the measurements taken during the executions of our benchmarks (Tables 5.1 - 5.4) and compare them to our expected results. We will see, that in some cases, the actual results differ from the expected results, for various reasons. Although there is no environment, that outperforms all other environments, we will see clear differences and tendencies on which implementation might be the most useful in practice. Also, the results point out several weaknesses of our algorithms and their implementations, which we can use as a reference point for future improvements. Still, we need to keep in mind that these are artifical benchmarks, and it is highly unlikely to encounter some of these situations in practice. One benchmark is in some way special, as

there are actually no significant differences for most measurements, due to the way this benchmark is constructed (Cross Reference: nonrc, non-rc to rc). We will exclude this benchmark for most of our considerations, unless otherwise noted.

5.3.1 Memory

Average and maximum memory consumption are the most important values we measure during the execution of our benchmarks, as our primary goal is to save memory, by freeing cyclic objects. As we need to allocate some memory during our collections (especially by creating a snapshot in the incremental implementations), it is also important to capture this overhead.

Effectiveness

The effectiveness of our algorithms can be measured, by creating a lot of objects, which die after a relatively short time. In the *0% alive* columns, we see the measurements in case all of the created cyclic structures die, directly after they have been fully constructed. Both, mean and maximum memory consumption, are equally important in this consideration.

Base vs. Rest As expected, we are able to free large amounts of memory in our Mark, Inc5k and Inc50k environments, whereas in Base the memory consumed is about the same as if 100% of our objects would be alive (as we are unable to free any cyclic structures). The only notable exceptions to this rule are found in the Cross Reference non-rc to rc benchmarks, as the non-reference counted objects can also be freed in our Base environment. Still, in this benchmark the Base environment uses more memory on average, as the algorithm is unable to free any reference counted objects. Depending on the benchmark, we measure up to 95% lower average memory usage, compared to Base.

Mark vs. Inc5k vs. Inc50k With only few exceptions, the average and maximum memory consumption are slightly higher for Inc5k and Inc50k compared to Mark. The average memory consumption is slightly lower for Inc5k and Inc50k compared to Mark in the Cross Reference nonrc benchmarks, but this is not a significant difference.

Significant differences can be found in the Cross Cycles benchmarks. If we look at the raw data, we can see that, especially for the cross variant, but also for the other two variants, the number of increments per major collection seem to escalate for the incremental implementations. In all variants, the working set is increasing during a collection cycle, because quite a lot of new border objects are added between two increments (which are added to the working set at the beginning of each increment). As the incremental limit seems to be too low for this case, the working set is only slowly getting smaller and a high number of increments is needed to complete a collection cycle. This increases the timespan between the creation and the deletion of the cyclic structures and thus greatly increases the average and maximum memory consumption. It also delays the next collection, which leads to a significant decrease in the number of major collections,

Table 5.1: Benchmark Results - Simple Cycle 2^{14}

			0% alive				50% alive				100% alive			
			Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	count	263	97	99	98	263	208	208	204	262	312	309	305
		size	132	97	101	100	135	122	123	125	132	168	167	166
		nongc	134	101	101	101	139	119	115	112	145	156	151	161
		∅	**177**	**98**	**100**	**100**	**179**	**150**	**148**	**147**	**180**	**212**	**209**	**211**
	Max	count	461	109	112	112	462	297	300	300	457	466	465	464
		size	244	132	136	135	246	191	197	197	243	253	262	260
		nongc	244	132	136	136	246	178	176	178	253	248	251	256
		∅	**316**	**124**	**128**	**128**	**318**	**222**	**224**	**225**	**317**	**322**	**326**	**327**
Time (s)	GC	count	0.48	0.34	0.35	0.35	0.48	1.47	1.27	1.24	0.47	2.05	1.73	1.64
		size	0.14	0.17	0.19	0.19	0.14	0.28	0.28	0.28	0.14	0.44	0.42	0.42
		nongc	0.14	0.16	0.16	0.16	0.14	0.22	0.20	0.20	0.14	0.34	0.28	0.29
		∅	**0.25**	**0.22**	**0.23**	**0.23**	**0.25**	**0.66**	**0.58**	**0.57**	**0.25**	**0.94**	**0.81**	**0.78**
	Total	count	0.74	0.52	0.52	0.53	0.75	1.72	1.53	1.51	0.74	2.33	2.03	1.95
		size	0.26	0.26	0.27	0.27	0.25	0.38	0.38	0.39	0.26	0.56	0.54	0.55
		nongc	0.25	0.24	0.24	0.24	0.25	0.32	0.30	0.30	0.25	0.46	0.41	0.42
		∅	**0.42**	**0.34**	**0.35**	**0.35**	**0.42**	**0.81**	**0.74**	**0.73**	**0.42**	**1.12**	**0.99**	**0.97**
Pauses (ms)	Mean	count	8.37	5.50	3.49	3.53	8.39	24.25	4.46	11.33	8.32	35.92	4.25	14.01
		size	5.47	6.31	2.93	4.27	5.45	11.27	2.23	6.09	5.51	17.48	2.00	7.70
		nongc	5.46	5.78	3.52	3.52	5.46	8.86	4.92	5.00	5.37	12.94	6.61	6.72
		∅	**6.43**	**5.86**	**3.31**	**3.77**	**6.43**	**14.79**	**3.87**	**7.47**	**6.40**	**22.11**	**4.29**	**9.48**
	Max	count	69.0	22.1	11.9	12.4	69.3	234.0	71.7	73.6	68.1	397.4	112.4	106.0
		size	31.8	28.1	16.6	16.3	32.1	87.8	34.0	34.7	32.1	140.2	51.5	52.0
		nongc	32.1	24.9	15.5	15.3	32.1	58.9	31.9	31.6	31.5	102.2	33.2	34.2
		∅	**44.3**	**25.0**	**14.7**	**14.6**	**44.5**	**126.9**	**45.8**	**46.6**	**43.9**	**213.3**	**65.7**	**64.1**

Table 5.2: Benchmark Results - Cross Cycles 2^{14}

		0% alive			50% alive			100% alive		
		Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean												
	size	1800	542	798	603	1798	1217	1536	1320	1816	1978	2169	2164
	cross	2204	604	1278	1284	2215	1430	1743	1752	2210	2537	2317	2310
	count	2213	112	219	219	2226	1527	1691	1464	2229	2561	2354	2367
	∅	**2073**	**419**	**765**	**702**	**2080**	**1391**	**1657**	**1512**	**2085**	**2359**	**2280**	**2280**
	Max												
	size	3876	637	1143	690	3870	2368	2879	2465	3878	3924	4113	4056
	cross	4282	674	2285	2282	4290	2619	3210	3209	4282	4346	4545	4545
	count	4303	115	237	235	4313	2674	3332	2752	4305	4364	4554	4450
	∅	**4154**	**475**	**1222**	**1069**	**4158**	**2554**	**3140**	**2809**	**4155**	**4211**	**4404**	**4350**
Time (s)	GC												
	size	3.73	6.17	5.00	5.39	3.75	7.36	6.24	6.27	3.78	10.12	8.69	9.19
	cross	5.07	9.50	7.37	6.88	5.14	11.61	9.54	8.95	5.14	15.61	12.29	12.04
	count	5.19	2.81	3.74	3.47	5.23	10.82	8.22	7.35	5.28	16.17	11.89	13.24
	∅	**4.66**	**6.16**	**5.37**	**5.25**	**4.70**	**9.93**	**8.00**	**7.52**	**4.73**	**13.97**	**10.96**	**11.49**
	Total												
	size	5.13	7.26	6.18	6.41	5.16	8.62	7.71	7.49	5.20	11.57	10.33	10.68
	cross	6.80	10.96	9.19	8.59	6.86	13.21	11.58	10.86	6.87	17.38	14.54	14.16
	count	6.92	4.07	5.07	4.70	6.97	12.44	10.12	8.91	7.02	17.98	14.04	15.17
	∅	**6.28**	**7.43**	**6.81**	**6.57**	**6.33**	**11.42**	**9.80**	**9.09**	**6.36**	**15.65**	**12.97**	**13.34**
Pauses (ms)	Mean												
	size	14.24	13.49	6.67	9.50	13.67	24.68	6.06	11.97	13.00	34.91	5.04	12.80
	cross	20.05	22.82	0.10	0.09	20.31	45.27	0.10	0.10	20.30	60.72	0.12	0.11
	count	19.93	4.90	4.51	4.20	18.54	41.11	7.48	13.54	17.91	51.61	6.74	13.38
	∅	**18.08**	**13.73**	**3.76**	**4.60**	**17.51**	**37.02**	**4.55**	**8.54**	**17.07**	**49.08**	**3.96**	**8.77**
	Max												
	size	514.9	195.3	170.6	85.5	506.0	1235.5	336.7	317.6	508.2	2678.5	487.9	518.7
	cross	841.0	334.7	287.8	277.3	855.1	2287.9	445.7	412.5	848.8	4155.0	564.5	537.7
	count	853.6	13.5	23.9	22.9	830.7	2273.4	374.9	394.6	832.3	4316.2	566.9	618.3
	∅	**736.5**	**181.2**	**160.8**	**128.5**	**730.6**	**1932.3**	**385.8**	**374.9**	**729.7**	**3716.5**	**539.8**	**558.2**

Table 5.3: Benchmark Results - Cross Reference 2^{14} (rc to non-rc)

			0% alive				50% alive				100% alive			
			Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	count	256	93	108	108	250	189	216	192	255	303	263	282
		nonrc	133	92	89	90	132	111	102	96	133	135	138	140
		rc	134	95	100	98	139	123	121	127	137	165	171	168
		∅	**174**	**93**	**99**	**99**	**174**	**141**	**146**	**138**	**175**	**201**	**191**	**197**
	Max	count	487	110	132	133	483	310	382	327	485	495	501	510
		nonrc	248	151	167	166	247	197	186	183	247	247	253	253
		rc	244	129	135	136	247	191	197	197	247	251	263	261
		∅	**326**	**130**	**145**	**145**	**325**	**232**	**255**	**236**	**326**	**331**	**339**	**341**
Time (s)	GC	count	0.46	0.24	0.32	0.30	0.46	0.58	0.49	0.47	0.46	1.02	0.57	0.73
		nonrc	0.23	0.13	0.14	0.13	0.23	0.18	0.17	0.15	0.24	0.26	0.28	0.26
		rc	0.15	0.17	0.20	0.17	0.15	0.28	0.28	0.26	0.15	0.44	0.41	0.39
		∅	**0.28**	**0.18**	**0.22**	**0.20**	**0.28**	**0.35**	**0.31**	**0.29**	**0.28**	**0.57**	**0.42**	**0.46**
	Total	count	0.65	0.38	0.46	0.43	0.65	0.75	0.67	0.64	0.65	1.22	0.77	0.92
		nonrc	0.27	0.17	0.18	0.16	0.27	0.22	0.21	0.19	0.27	0.30	0.32	0.29
		rc	0.27	0.26	0.29	0.25	0.27	0.38	0.39	0.35	0.27	0.57	0.52	0.51
		∅	**0.40**	**0.27**	**0.31**	**0.28**	**0.40**	**0.45**	**0.42**	**0.39**	**0.40**	**0.69**	**0.54**	**0.57**
Pauses (ms)	Mean	count	9.74	4.54	4.31	4.02	9.74	13.49	5.20	7.19	9.74	21.69	5.12	7.80
		nonrc	4.79	4.14	3.15	2.87	4.79	4.97	3.48	3.13	4.81	5.32	4.12	3.75
		rc	5.74	6.27	3.10	3.88	5.81	11.18	2.24	5.55	5.81	17.73	1.95	7.13
		∅	**6.76**	**4.98**	**3.52**	**3.59**	**6.78**	**9.88**	**3.64**	**5.29**	**6.79**	**14.91**	**3.73**	**6.23**
	Max	count	74.8	10.6	9.7	9.2	74.1	124.0	38.0	32.9	75.6	296.1	51.7	58.0
		nonrc	20.1	3.8	3.6	3.5	20.4	17.5	8.2	7.3	20.2	31.8	31.2	29.4
		rc	33.4	28.0	17.7	16.0	33.6	85.6	33.7	32.5	33.4	140.8	49.6	50.8
		∅	**42.7**	**14.2**	**10.3**	**9.6**	**42.7**	**75.7**	**26.6**	**24.3**	**43.1**	**156.2**	**44.2**	**46.1**

Table 5.4: Benchmark Results - Cross Reference 2^{14} (non-rc to rc)

		0% alive			50% alive			100% alive		
		Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	count	151	87	91	88	189	170	183	181	241	272	265	270
		nonrc	94	94	91	88	102	100	97	93	134	131	127	133
		rc	140	115	120	119	137	135	133	132	137	168	169	166
		∅	**128**	**99**	**101**	**98**	**143**	**135**	**138**	**136**	**171**	**190**	**187**	**190**
	Max	count	272	109	114	114	381	294	347	325	484	482	508	491
		nonrc	153	151	169	165	192	188	184	178	250	248	247	251
		rc	249	148	154	155	247	191	197	197	246	256	261	260
		∅	**225**	**136**	**146**	**145**	**273**	**224**	**243**	**234**	**327**	**329**	**339**	**334**
Time (s)	GC	count	0.21	0.16	0.19	0.17	0.27	0.41	0.37	0.39	0.42	0.78	0.57	0.64
		nonrc	0.13	0.13	0.14	0.13	0.16	0.16	0.17	0.15	0.23	0.23	0.22	0.21
		rc	0.15	0.23	0.25	0.23	0.15	0.32	0.33	0.31	0.15	0.43	0.42	0.39
		∅	**0.17**	**0.17**	**0.19**	**0.18**	**0.20**	**0.30**	**0.29**	**0.28**	**0.27**	**0.48**	**0.40**	**0.41**
	Total	count	0.41	0.29	0.34	0.29	0.47	0.60	0.58	0.57	0.63	0.99	0.78	0.84
		nonrc	0.17	0.17	0.18	0.16	0.20	0.20	0.20	0.19	0.27	0.27	0.26	0.25
		rc	0.27	0.32	0.34	0.32	0.27	0.43	0.44	0.41	0.27	0.56	0.55	0.51
		∅	**0.28**	**0.26**	**0.28**	**0.26**	**0.31**	**0.41**	**0.41**	**0.39**	**0.39**	**0.60**	**0.53**	**0.53**
Pauses (ms)	Mean	count	6.50	4.54	3.57	3.08	6.21	9.52	4.47	6.54	5.95	10.91	4.58	6.78
		nonrc	4.10	4.12	3.12	2.87	4.06	4.11	3.10	2.85	3.47	3.47	3.11	2.99
		rc	5.80	8.79	2.55	5.15	5.76	13.05	2.28	6.27	5.85	17.34	2.02	7.13
		∅	**5.47**	**5.81**	**3.08**	**3.70**	**5.34**	**8.89**	**3.28**	**5.22**	**5.09**	**10.57**	**3.24**	**5.63**
	Max	count	46.8	17.5	12.7	11.5	34.9	97.2	34.4	40.7	36.0	177.4	55.0	49.7
		nonrc	4.2	3.8	3.6	3.4	3.8	4.0	3.8	3.6	3.8	4.0	3.7	3.5
		rc	32.9	42.3	22.0	22.4	33.4	88.2	34.8	32.9	32.8	136.2	52.0	50.4
		∅	**27.9**	**21.2**	**12.8**	**12.4**	**24.0**	**63.1**	**24.3**	**25.7**	**24.2**	**105.9**	**36.9**	**34.5**

which we don't see for any of the other benchmarks. This also happens if all (or some) objects stay alive, but as the objects surviving anyway in these cases, we don't see such a large increase in maximum memory usage and even a decrease in average memory usage.

A solution to this problem would be higher, better or even adaptive incremental limits with respect to cross references. As we can see in the size variant, the higher incremental limit seems to prevent such an escalation.

Efficiency

To get a sense of the efficiency of our algorithms with regards to memory usage, we can measure the additional overhead of our implementations. This can be done by creating a lot of live objects, which will never be freed, but need to be processed in every collection. In the *100% alive* columns, we can see the results, in case all of the created cyclic structures stay alive until the end of our benchmarks. The maximum amout of memory, while in practice useful and important, is not a good indicator of the overhead of our implementaions in our benchmarks, as it heavily depends on the exact time the last collection is triggered. As the time of our last collection depends on a lot of factors, the average memory seems to be a better indicator of the actual overhead introduced by our algorithms.

Base vs. Rest As expected, we see that the average memory overhead is higher for Mark, Inc5k and Inc50k, compared to Base. Depending on the benchmark, the overhead ranges from about 4 to 19 percent.

Mark vs. Inc5k vs. Inc50k Interestingly, in some cases the average memory overhead of the Inc5k environment is significantly lower, than the memory overhead of the Mark enviroment (Cross Cycles: cross and count; Cross Reference: count). In these cases, the reason is probably a more efficient working set for cross references (as we deal with an extremely high number of them in exactly these benchmarks).

For our Simple Cycle benchmarks (and some other), the average memory overhead is about the same in Mark and Inc5k, whereas in other benchmarks (e.g. Cross Cycles: size) the average memory overhead is higher for Inc5k compared to Mark. The lower overhead might result from the fact that Inc5k is (surprisingly) faster than Mark in terms of pure GC time (for details, see Subsection 5.3.2), which, especially in late collections which take more time, as the number of objects steadily increases, results in less samples taken when memory consumption is high. In some benchmarks, this seems to outweight the effects of the snapshot and the working set. Depending on the actual benchmark, this results in a higher or lower average memory consumption.

For most microbenchmarks, we see that the maximum memory consumption with 100% of live objects is about the same for Base and Mark. This is most likely due to the fact, that at the end of our benchmark, the consumed memory is the highest, and that the probability of a collection right at the end of our benchmark is relatively low. The

differences we see are most likely just due to different timing of our memory samples. The two incremental environments actually have a higher maximum memory consumption in this case. This might be because of the increased chances, that a collection is still taking place at the end of our benchmarks, as we might still be in between two increments. In this case the snapshot is currently held in memory, which would result in an overall higher maximum memory consuption.

5.3.2 Time

Generally, cycle detection and collection take up additional time, but the modified heap might lead to changes in performance of the executed program [BCM04]. Positive effects on application performance could help reducing the performance penalty or even lead to faster execution times. In any way, we consider garbage collection time and total execution time as important factors in our benchmarks.

Impact

To measure the impact of the additional cycle detection on the reference counted heap, we take a look at the *0% alive* columns first. Afterwards, we will take a separate look at the overhead, by inspecting the *100% alive* columns.

Base vs. Rest In most benchmarks, the total execution time for the Mark, Inc5k and Inc50k is quite a lot lower than for the Base environment. In most of these cases, the GC time is also significantly lower. This is surprising, as we would expect higher GC times, if we need to process the reference counted heap. But it seems that in case a lot of cycles are created, the overhead of keeping them alive is higher than the overhead of marking and sweeping them. What is even more surprising is the fact, that even though the GC times are higher in the Simple Cycle benchmark (count and nongc variant), the total time is lower. If we look at the other benchmarks with lower total time more closely, we also see that the benefit in total runtime is greater than the benefit in GC runtime. This is probably because of a more efficient memory layout, if we regularily free unused objects.

In some benchmarks, the total time of Mark/Inc5k/Inc50k is higher or equal compared to the Base environment. This seems to be the case for all benchmarks with large graphs on the reference counted heap (Simple Cycle: size; Cross Cycles: size, cross; Cross Reference: rc). One possible explanation is, that the graphs are so large, that the overhead of keeping the graph, that is currently created when the collection is triggered, alive, is higher than the performance benefits. In all other benchmarks, even though a lot of them are created, the graphs are relatively small, so the overhead of keeping one of them alive might not be notable.

In any case, if we take a short look at the 50% alive columns, we quickly realize that we can only save on total execution time, if a lot of objects die quickly. This is true for all of our bechmarks.

Mark vs. Inc5k vs. Inc50k As expected, due to the synchronization overhead, the fully-incremental implementation is generally slightly slower than the semi-incremental one. Two notable exceptions can be found: the size and cross variants of the Cross Cycles benchmark. In these variants, the number of major collections is lower for the incremental implementation (as already noted in Subsection 5.3.1), thus the GC time and the total time are lower compared to Mark. In the Cross Reference benchmarks, for the nonrc and the rc variants the Inc50k environment is the fastest, but not significantly. The performance of the incremental environments is generally quite similar, depending on the benchmark one is sometimes slightly faster than the other.

Overhead

To evaluate the temporal overhead of our cyclic garbage collection, we take look at the *100% alive* columns. We would expect an overhead in all variants of our benchmarks, compared to the Base environment.

Base vs. Rest As expected, all other environments are slower than Base (by up to 215%), as Base does not perform tracing on the reference counted heap. Processing the reference counted heap in the other implementations is unable to improve the memory layout, as all objects survive and no objects are moved on the heap (unlike e.g. in mark-compact collectors [BCM04]).

Mark vs. Inc5k vs. Inc50k The incremental implementations are generally slightly faster, even though they introduce additional overhead by creating a snapshot and especially by synchronizing the snapshot at the end of the marking phase. But the snapshot seems to be faster to process during the marking phase, which results in an overall faster collection process in all benchmarks. Both incremental environments behave very similar, depending on the benchmark one is sometimes faster than the other.

5.3.3 Pauses

Keeping pause times low, is the main reason why incremental garbage collection is implemented. A good indicator are maximum pause times, as average pause times might only be a result of having a lot of extremely short pauses beside less frequent long pauses. Ideally, the average pause time is close to the maximum pause time, as this is an indicator that the pause times are quite evenly distributed.

100% alive

Our goal for the incremental environments is, to keep the pause times at a similar level as in the Base environment (which already performs most of its work in an incremental manner). The 100% alive columns are somewhat more suitable than the other columns for this comparison, as the workload stays the same (no objects are freed in any of the environments). As the semi-incremental Mark environment will only cause increased

maximum and average pause times (as all of the additional work is performed in the last increment of the otherwise unchanged Base implementation), we can use it as a reference to show the potential of our incremental algorithm.

Maximum Pause Times Generally, the results concerning maximum pause times are quite what we would expect. The maximum pause times for Mark compared to Simple are a lot higher for all benchmarks. The incremental collection helps, but still increases maximum pause times in most benchmarks. In benchmarks with a lot of cross references (all variants of the Cross Reference benchmarks), maximum pause times are lower for the fully-incremental environments, compared to the Base environment, as objects at the border have previously been all marked in the last increment, which seems to be less efficient than creating a snapshot in the last increment of the non-reference counted marking phase. There is no big difference between Inc5k and Inc50k in most benchmarks, because the snapshot creation and synchronization, which has to be done in one single increment, seems to take more time than all other increments. Most differences seem to be caused by the increasing working set between increments (new and changed objects are added to the working set) and in most benchmarks there are more increments in the Inc5k environment than in the Inc50k environment, because increments are generally shorter due to the lower incremental limit.

Average Pause Times If we take a look at the average pause times, we see that they are relatively low for most benchmarks in the incremental environments (compared to Base) and extremely low for one benchmark specifically. In the cross variant of the Cross Cycles benchmark, due to the escalation of increments per major collection, we have already witnessed in the previous subsections, the average pause time is extremely low (about 0.11 ms). We also see that the average pause time of Base is a little bit higher or in between the average pause times of Inc5k and Inc50k for most benchmarks. This indicates that the incremental limits seem appropriate and that the 'ideal' limit can probably be found somewhere in between 5,000 and 50,000 objects.

0% alive

In this comparison, the workload is not the same for all algorithms, as our Base environment will not be able to free most dead objects and objects that are freed are never processed again, so we will obviously see big differences of all other environments compared to the Base environment. The most interesting part will therefore be, how the incremental environments perform compared to the Mark environment.

Base vs. Rest Maximum pause times are significantly lower for all algorithms compared to Base (between 10 and 98 percent), because Base always has to keep the objects it is unable to free alive in the last increment. We see the same effect for average pause times in most benchmarks (especially for the incremental environments), but as already stated, average pause time are influenced by other factors as well (like incremental limits).

The only exception to this rule is the rc variant of the Cross Reference benchmark (direction non-rc to rc). This is because in this variant of the benchmark, Base does not need to keep any linked objects alive (non-rc objects die anyway, rc objects stay alive anyway). Still, the incremental environments are able to outperform the Base environment, because they seem to be able to handle cross references more efficiently than the Mark environment.

Mark vs. Inc5k vs. Inc50k If we compare the fully-incremental environments to the Mark environment, we generally witness that the average and maximum pause times are lower for Inc5k and Inc50k. The only notable exception is the count variant of the Cross Cycles benchmark, where Mark outperforms Inc5k and Inc50k in terms of maximum pause times. This is probably another effect of the escalating number of increments per major collection, as the increasing working set leads to overall longer collection cycles, which increases the time needed for synchronizing the snapshot at the end of the marking phase (which needs to happen in one single increment).

We can see the impact of this effect more accurately, if we compare the two incremental algorithms in the size variant of the Cross Cycles benchmark. Without this effect, both should have comparable maximum pause times, but due to the high number of increments, which seems to be escalating only in the Inc5k environment, the maximum pause times of Inc5k are significantly higher than Inc50k. In all other bechmarks, the maximum pause times are about the same for both incremental algorithms and the average pause times for the Inc5k are, as expected, slightly lower than in the Inc50k benchmark.

5.3.4 Scaling

To check, if the metrics of our implementations also scale well with increased heap sizes, we ran all microbenchmarks with different numbers of objects. To achieve this, we always scale the biggest variable according to the definition of the benchmark variants. We scale this variable exponentially, by keeping the base of two and incrementing the exponent by one each step, starting from 2^n, where the initial value of n varies, depending on the benchmark. The results are included in the appendix.

Fortunately our implementations all seem to scale reasonably well. The mean and maximum memory consumptions scale quite linearily for the 100% alive columns (very similar to the Base environment) and at most linearily for the 0% colums, where the scaling is in some benchmarks close to constant, but definitely quite a lot lower than in the Base environment. With regards to GC time and total runtime, all benchmarks scale somewhere between linearily and quadratically (the majority almost linearily) in the 0% columns, in all implementations. The Mark, Inc5k and Inc50k environments seem to scale worse than the Base environment in a lot of benchmarks, but still reasonably well. In the 100% column, we can see that all implementations scale about equally well for all benchmarks, most of them scale almost linearily. Compared to the Base environment, average and maximum pause times scale almost always better in the Mark, Inc5k and Inc50k environments, only for some variants the Base environment scales better than the

Mark environment. In nearly all cases, maximum pause times scale linearily or slightly better and almost all average pause times stay constant. In some variants, average pause times scale a little bit worse, but always better than linear.

5.3.5 Summary

During the analysis ouf our microbenchmarks, we found out, that if a sufficiently large number of objects are freed by detecting reference cycles, average and maximum memory consumption decrease significantly. If a lot of objects die, cycle detection can also offer performance gains in terms of total processing time. Average and maximum pause times are significantly lower for the fully-incremental implementations, compared to the semi-incremental implementation, which is not surprising. However, we also found out, that the runtime overhead of creating and synchronizing the snapshot, which is needed for the fully-incremental collection, is sometimes completely compensated (or even overcompensated) by the seemingly better memory layout during tracing. The memory overhead of the snapshot is notable, but seems to be quite reasonable. Finally, we were able to identify some weak spots in the fully-incremental implementations, especially in the Cross Cycles benchmark.

5.4 Application Benchmarks

We have already seen the differences of our implementations in special scenarios in the microbenchmarks. Now we will take a look at the differences in more realistic scenarios. Table 5.5 summarizes the results of our application benchmarks. As we will see in our analysis, every implementation/environment has its benefits and its drawbacks, depending on the application under test.

5.4.1 Memory

In the Gnome Tweaks benchmark, average and the maximum memory usage in all environments with cycle detection, is lower than in the Base environment (by about 2 to 3 percent). This means that quite a lot of objects could be freed and that the overhead of the collections and/or snapshots is worth the effort. In the Quod Libet benchmark, the average memory usage of the fully-incremental implementations is slightly lower than Base (by about 2%), even though the maximum memory usage is higher.

In the other three benchmarks, the average memory consumption of Mark, Inc5k and Inc50k is higher compared to Base (by about 2 to 4 percent). In almost all benchmarks, the maximum memory consumption is also higher (by up to 9%), with Gnome Music as an exception and PyChess as a special case, where the maximum memory consumption is slightly lower for Inc50k compared to Base. In any case, we have to be careful, because the standard deviation can be quite high, depending on the application and the environment, meaning that the maximum memory is not equally easy to predict/measure for each execution.

Table 5.5: Benchmark Results - Application Benchmarks

			Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	Gnome Music	207.02	210.17	210.43	210.69
		Gnome Tweaks	174.34	170.01	170.77	170.26
		PyChess	205.28	210.58	210.39	209.03
		Pygame	127.52	130.61	133.17	132.43
		Quod Libet	179.68	179.09	175.70	176.46
		∅	**178.77**	**180.09**	**180.09**	**179.77**
	Max	Gnome Music	224.08	223.12	225.75	225.80
		Gnome Tweaks	189.64	183.97	184.70	183.57
		PyChess	237.69	240.88	238.51	236.25
		Pygame	134.73	138.88	147.39	147.22
		Quod Libet	190.15	189.72	195.89	193.48
		∅	**195.26**	**195.32**	**198.45**	**197.27**
Time (s)	GC	Gnome Music	0.0909	0.0922	0.1030	0.1016
		Gnome Tweaks	0.1039	0.1200	0.1184	0.1140
		PyChess	0.6316	0.7002	0.6911	0.6977
		Pygame	1.0543	1.2734	1.2384	1.2421
		Quod Libet	0.2701	0.2949	0.3352	0.3084
		∅	**0.4302**	**0.4961**	**0.4972**	**0.4927**
Pauses (ms)	Mean	Gnome Music	10.68	11.99	9.45	9.27
		Gnome Tweaks	12.49	14.59	10.41	10.08
		PyChess	9.13	10.84	7.64	7.63
		Pygame	10.19	13.70	9.75	9.78
		Quod Libet	8.79	9.62	9.54	8.59
		∅	**10.26**	**12.15**	**9.36**	**9.07**
	Max	Gnome Music	11.4	15.2	17.0	17.1
		Gnome Tweaks	13.0	19.3	13.4	13.2
		PyChess	19.9	31.9	17.8	17.6
		Pygame	24.4	64.9	41.7	41.7
		Quod Libet	15.2	27.1	21.9	18.0
		∅	**16.8**	**31.7**	**22.4**	**21.5**

5.4.2 Time

In our application benchmarks, the total execution time is practically irrelevant, as we are measuring interactive applications and the execution time mostly depends on the speed of the UI automation framework. What is more relevant, is the time the garbage collection takes. Even though a better memory layout can also have an impact on the rest of the application, as we have seen in the microbenchmarks, it is hard to measure this impact for interactive applications and the effect is probably negligible for the tested applications.

In all bechmarks, Mark, Inc5k and Inc50k have increased GC times, compared to Base (between 9 and 24 percent). If we take the results from our microbenchmarks into account, this is most likely due to the fact that most objects stay alive and only a small number of objects in cyclic structures can be freed. Interestingly, the GC time is lower for the fully-incremental implementations in three of the five benchmarks, compared to the semi-incremental implementation. As the number of collections is the same (according to the raw data), this means that the collections themselves are faster. This is consistent to what we saw in our microbenchmarks.

5.4.3 Pauses

Maximum pause times are of peculiar interest, when it comes to interactive applications. Extremely long pauses lead to high response times of the user interface, which have a negative effect on usability, once a certain threshold is exceeded. According to Dabrowski and Munson, most users won't notice a delay below 150 milliseconds for keyboard interactions and 195 milliseconds for mouse interactions [DM01]. PyGame is somewhat special, as it is a framework for creating interactive games, most of which require fluent animations and thus much smaller delays. According to Valente et al. [VCF, p. 1] a commonly accepted lower bound for the number of frames per second (FPS) is 16, which means one frame should take at most 62.5 milliseconds to render. Ideally the frame rate should be between 50 and 60 FPS, so a single frame should be rendered within 16 to 20 milliseconds.

If we look at our results, even though the GC pauses might add up to already present delays, the pauses themselves still seem to be small enough for the tested UI applications in all environments, with the exception of Pygame. Here, the measured maximum pause times will definitely be noticable by users, especially as they will add up to the existing time spent for rendering a single frame. However, due to the recent improvements in the GC, our changes to the garbage collector will not influence the FPS, as they allow developers to disable automatic major collections during animations. As the additional code for detecting cycles is only executed during major collections, ideally users will not notice any difference.

Still, there are significant differences between the implementations and environments. In two of five benchmarks (Pygame and Quod Libet), the Mark environment has drastically higher pause times, compared to the Base environment (about twice as high). In the

other three benchmarks, the maximum pause times are also higher, but not as extreme. In the Pygame and the Quod Libet benchmarks, the fully-incremental implementation improves the maximum pause times compared to the Mark environment. However, the Base environment still offers significantly lower maximum pause times. In the case of Pygame, this is especially troublesome, as most games and multimedia applications are extremely sensitive, even to short hangups. However, recent improvements in the garbage collector, which allow the application to pause major collections and provide interfaces to execute them on demand, have the potential to mitigate this problem almost entirely [pypb]. In the Gnome Music benchmark, we also experience higher maximum pause times for the Inc5k and Inc50k environments, compared to the Base environment, probably because the snapshot creation and synchronization add quite some overhead. In the Gnome Tweak benchmark, the maximum pause times are only slightly higher for Inc5k and Inc50k.

In the Pychess benchmark on the other hand, the maximum pause times are slightly lower for Inc5k and Inc50k compared to Base. If we look at the standard deviation of the average pause times in the appendix, we can also see that it is significantly lower (89% and 81% lower) compared to the Base environment, which means that pause times are more uniform and predictable and thus less likely to cause interruptions. However, we can see the opposite effect in some other benchmarks.

In terms of average pause times, almost all benchmarks show higher values for Mark and lower values for Inc5k and Inc50k, compared to Base. This is most likely due to the fact that Mark performs additional computations in the last increment of Base and Inc5k/Inc50k introduce additional increments, which seem to be shorter than the average increment of Base.

5.4.4 Summary

In total, we could identify positive and negative effects of employing cyclic garbage collection on the applications under test. We saw that the average memory consumption decreased in Quod Libet, when employing incremental cycle detection. In Gnome Tweaks, the average and maximum memory consumption decreased, when using any kind of cycle detection. Even though the GC time increased, the maximum pause times did not significantly increase in the incremental environments, making the incremental implementation an ideal choice for Gnome Tweaks. In PyChess, maximum pause times decreased, when employing incremental cycle detection. Generally, average and maximum memory consumption, GC time and average and maximum pause times increased, but apart from very few exceptions, the changes were quite moderate.

5.5 Issues

Apart from rather theoretical issues, we identified using our microbenchmarks, we also recognized some issues, that might be more relevant in practice. However, these issues are

Table 5.6: Benchmark Results - cpyext

			Base	Mark	Inc5k	Inc50k
Time (s)	Total	simple.onearg(None)	0.25	0.27	0.25	0.25
		simple.onearg(i)	1.44	1.79	1.65	1.63
		simple.varargs	0.49	0.60	0.52	0.54
		simple.allocate_int	1.18	1.28	1.27	1.26
		simple.allocate_tuple	7.49	14.94	14.81	14.80
		Foo().noargs	0.22	0.23	0.22	0.22
		Foo().onearg(None)	0.23	0.24	0.23	0.24
		Foo().onearg(i)	1.39	1.86	1.69	1.69
		Foo().varargs	0.47	0.56	0.49	0.50
		len(Foo())	0.16	0.17	0.16	0.16
		Foo()[0]	0.30	0.32	0.30	0.31
		onearg(None)	0.25	0.26	0.25	0.25
		onearg(1)	0.31	0.29	0.28	0.29
		onearg(i)	1.41	1.85	1.67	1.64
		onearg(i%2)	1.43	1.92	1.69	1.65
		onearg(X)	0.31	0.29	0.28	0.30
		onearg((1,))	0.28	0.27	0.27	0.28
		onearg((X,))	1.68	1.91	1.95	1.92
		onearg((i,))	4.36	6.88	6.49	6.47
		getitem	0.76	0.72	0.73	0.73
		np.mean	1.49	1.54	1.58	1.56
		∅	**1.23**	**1.82**	**1.75**	**1.75**

outside of the scope of this thesis. Antonio Cuni created a small set of microbenchmarks for cpyext [cpya]. We ran those benchmarks several times and included the results in Table 5.6.

While the performance of all other benchmarks seems to stay on a comparable level, one specific benchmark highlights a relatively practical issue: Tuples are already quite slow when it comes to cpyext, but the situation seems to get even worse, if we also take cycle detection into account. As noted in the Chapter 4, the existing tuple implementation had to be extended, to support cyclic reference counting. This clearly seems to have a negative impact on the overall performance, almost doubling the total execution time of the *simple.allocate_tuple* benchmark. As this is a rather specific problem and also seems to be an implementation specific issue, we did not consider this problem as part of our thesis. Still, it should be considered, when applying the implementation in practice.

5.6 Summary

The benchmarking process revealed, that no implementation is clearly better than all other implementations, across the tested scenarios. Even though we were not able to identify a clear winner, we were able to gain some insights about the specifics of our algorithms and implementations in certain situations in terms of performance. Also, we observed that all implementations seem to scale relatively well.

Our microbenchmark showed, that if enough objects are freed by detecting reference cycles, not only average and maximum memory consumption decreased, but also the total runtime did, even though the time needed for garbage collection increased. They also reveiled an issue with the incremental limit of the fully-incremental implementation, where the working set steadily increases during a collection cycle. In case too many border objects are added between two increments, too few objects are processed during the increments. This leads to ever increasing working sets and long collection cycles with an unusually high number of increments.

In our application benchmarks, we saw that memory consumption increased for most applications. For only two applications enough cyclic memory could be freed on average, to compensate the additional memory needed for detecting cycles. Other benefits, like lower maximum memory consumption or lower maximum pause times, could also be identified, but they are limited to specific applications.

By comparing the results of the semi-incremental to the fully-incremental implementation, we could see that maximum pause times improved across virtually all benchmarks. We also witnessed, that even though the memory consumption generally increased due to the additional space needed for the snapshot in the fully-incremental implementation, the time needed for creating and synchronizing the snapshot seems to be compensated in almost all benchmarks by the improved memory layout during the marking phase. As the spatial overhead is quite reasonable in most cases, the use of a snapshot seems to be quite beneficial in terms of overall performance.

CHAPTER 6

Conclusion

In the previous chapter, we have seen how the implemented algorithms that we described in Chapter 3 perform. We will now summarize those findings and discuss, how the results of the benchmarking process should be interpreted with regards to the practical application of the implementations and future work in this area.

6.1 Application

We derived some general guidelines, on when which implementation should be used in practice. Depending on the actual application, the outcomes might differ and for optimal results, the individual characteristics of the application need to be taken into account.

For short running applications, the benefits of cycle detection seem to be small, especially as it often comes at the cost of increased pause times. However, for long running applications, freeing cyclic memory could make a crucial difference. As the impact on pause times is relatively low for the fully-incremental implementation, the Inc5k and Inc50k environments seem to be the best choice for interactive applications. Incorporating the results of our microbenchmarks, the choice for long running server applications depends on the memory layout, but in most cases the Inc5k or Inc50k environments still seem to be the best choice. For most short running batch applications, either the Base environment or the Inc5k/Inc50k environments offer the best performance. The choice heavily depends on the memory structure and on the desired performance characteristics of the application.

In the future, the choice of algorithm might also depend on further improvements and additional findings. This brings us to the following section.

6.2 Future Work

During our benchmarks, we have witnessed, that there is a very concrete issue with the working set of our fully-incremental implementation. This is definitely an area, where the existing implementation can be improved, as some applications might suffer from the detrimental behaviour in this scenario. Also, we have seen that the performance of the internal classes seems to deteriorate as an effect of cyclic reference counting. This is obviously an issue and should be investigated separately, as a result of this thesis.

Our presented approach should also be tested in similar situations, with different technologies and implementations, to back up our findings. As the core algorithm is relatively generic, the approach could also be applied to different tracing algorithms, such as mark-compact or copying garbage collectors. It should also be tested with different reference counting schemes. It would be interesting to see how the behaviour changes with different garbage collectors on both sides. The outcome is probably difficult to predict, as the level of potential optimizations might differ.

It should also be tested, how the approach performs, compared to more radical changes of the overall garbage collection process. For example, it could be compared to a fully reference counted garbage collector or a fully non-reference counted garbage collector. While it might not be desirable in practice, and while it was also not the goal of this thesis, to redevelop the entire collector for these kinds of integrations, it would be interesting to know from a scientific standpoint, which benefits hybrid approaches are able to offer and which downsides they have. In this case, especially the implications on program behaviour, e.g. the effects of immediate reclamation of non-cyclic object structures in reference counted systems, need to be taken into account. Then, language developers would have a deeper understanding and could make more profound decisions, on when to apply such hybrid approaches.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AP03] Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *International Conference on Compiler Construction*, pages 185–199. Springer, 2003.

[Axf90] Thomas H Axford. Reference counting of cyclic graphs for functional programs. *The Computer Journal*, 33(5):466–470, 1990.

[BCM04] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. Myths and realities: The performance impact of garbage collection. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):25–36, 2004.

[BCR04] David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. *SIGPLAN Not.*, 39(10):50–68, October 2004.

[BDS91] Hans-J Boehm, Alan J Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[BR01] David F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming*, pages 207–235, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[Bro85] David R Brownbridge. Cyclic reference counting for combinator machines. In *Conference on Functional programming languages and computer architecture*, pages 273–288. Springer, 1985.

[Chr84] Thomas W Christopher. Reference count garbage collection. *Software: Practice and Experience*, 14(6):503–507, 1984.

[coma] Com homepage. `https://docs.microsoft.com/en-us/windows/win32/com/the-component-object-model`. Accessed: 2020-02-08.

[comb] Rules for managing reference counts. `https://docs.microsoft.com/en-us/windows/win32/com/rules-for-managing-reference-counts`. Accessed: 2020-02-08.

[cpya] Cpyext benchmarks. https://github.com/antocuni/cpyext-benchmarks. Accessed: 2020-01-15.

[cpyb] Design of cpython's garbage collector. https://devguide.python.org/garbage_collector/. Accessed: 2020-02-06.

[csh] Interoperability (c sharp programming guide). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interop/. Accessed: 2020-02-02.

[CYTW10] Lin Chin-Yang and Hou Ting-Wei. An efficient approach to cyclic reference counting based on a coarse-grained search. *Information Processing Letters*, 111(1):1 – 10, 2010.

[DLM+76] Edsger W Dijkstra, Leslie Lamport, Alain J Martin, Carel S Scholten, and Elisabeth FM Steffens. On-the-fly darbage collection: an exercise in cooeration. In *Language hierarchies and interfaces*, pages 43–56. Springer, 1976.

[DLM+78] Edsger W Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and Elisabeth FM Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[DM01] James R Dabrowski and Ethan V Munson. Is 100 milliseconds too fast? In *CHI'01 Extended Abstracts on Human Factors in Computing Systems*, pages 317–318. ACM, 2001.

[dota] Dot (graph description language). https://en.wikipedia.org/wiki/DOT_(graph_description_language). Accessed: 2020-01-30.

[dotb] .net homepage. https://dotnet.microsoft.com/. Accessed: 2020-02-08.

[FF81] John K Foderaro and Richard J Fateman. Characterization of vax macsyma. In *Proceedings of the fourth ACM symposium on Symbolic and algebraic computation*, pages 14–19, 1981.

[FW79] Daniel P Friedman and David S Wise. Reference counting can manage the circular invironments of mutual recursion. *Information Processing Letters*, 8(1):41–45, 1979.

[gnoa] Gnome. https://www.gnome.org/. Accessed: 2020-01-15.

[gnob] Gnome music. https://wiki.gnome.org/Apps/Music. Accessed: 2020-01-14.

[gnoc] Gnome tweaks. https://wiki.gnome.org/Apps/Tweaks. Accessed: 2020-01-14.

[gtk] The gtk project. https://www.gtk.org/. Accessed: 2020-01-14.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 33–46, 1991.

[HK15] Michael John Hillberg and Raja Krishnaswamy. Managing object lifetime in a cyclic graph, June 9 2015. US Patent 9,053,017.

[HK17] Michael John Hillberg and Raja Krishnaswamy. Managing object lifetime in a cyclic graph, April 4 2017. US Patent 9,613,073.

[HLM09] Ting-Wei Hou, Chin-Yang Lin, and Tien-Yan Ma. A single-trace cycle collection for reference counting systems. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, pages 40–45. IEEE, 2009.

[Inc] Xamarin Inc. Xamarin.android - garbage collection. https://developer.xamarin.com/guides/android/advanced_topics/garbage_collection/. Accessed: 2017-03-31.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[JL96] Richard Jones and Rafael D Lins. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, 1996.

[jni] Java native interface. https://en.wikipedia.org/wiki/Java_Native_Interface. Accessed: 2020-02-02.

[jyt] Jython homepage. https://www.jython.org/. Accessed: 2020-02-08.

[ldt] Gnu ldtp. https://ldtp.freedesktop.org/wiki/. Accessed: 2020-01-14.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[Lin92] Rafael D Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992.

[MWL90] Alejandro D Martínez, Rosita Wachenchauzer, and Rafael D Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34(1):31–35, 1990.

[num] Rules for managing reference counts. https://numpy.org/. Accessed: 2020-02-11.

[pep] Pep 442 – safe object finalization. https://www.python.org/dev/peps/pep-0442/. Accessed: 2020-01-22.

[PvEP88] EJH Pepels, MCJD van Eekelen, and Marinus Jacobus Plasmeijer. *A cyclic reference counting algorithm and its proof*. Department of Theoretical Computer Science and Computational Models, Faculty ..., 1988.

[pyca] Pychess. http://pychess.org/about/. Accessed: 2020-01-14.

[pycb] Python/c api reference manual. https://docs.python.org/3/c-api/index.html. Accessed: 2020-02-02.

[pyga] Pygame. https://www.pygame.org/wiki/about. Accessed: 2020-01-15.

[pygb] Pygobject. https://pygobject.readthedocs.io/en/latest/. Accessed: 2020-01-14.

[pypa] Incremental garbage collector in pypy. https://morepypy.blogspot.com/2013/10/incremental-garbage-collector-in-pypy.html. Accessed: 2020-02-08.

[pypb] Pypy for low-latency systems. https://morepypy.blogspot.com/2019/01/pypy-for-low-latency-systems.html. Accessed: 2020-01-15.

[pypc] Pypy homepage. http://pypy.org/. Accessed: 2020-02-08.

[pyt] Python homepage. https://www.python.org/. Accessed: 2020-02-08.

[quo] Quod libet. https://quodlibet.readthedocs.io/en/latest/. Accessed: 2020-01-15.

[Ric14] Stefan Richthofer. Jyni-using native cpython-extensions in jython. *arXiv preprint arXiv:1404.6390*, 2014.

[Ric16] Stefan Richthofer. Garbage collection in jyni-how to bridge mark/sweep and reference counting gc. *arXiv preprint arXiv:1607.00825*, 2016.

[Sal87] Jon D Salkild. Implementation and analysis of two reference counting algorithms. *Master's thesis, University College, London*, 1987.

[SBYM13] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S McKinley. Taking off the gloves with reference counting immix. *ACM SIGPLAN Notices*, 48(10):93–110, 2013.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM Sigplan notices*, 19(5):157–167, 1984.

[VCF] LUIS VALENTE, AURA CONCI, and BRUNO FEIJÓ. Real time game loop models for single-player computer games.

Appendix

Additional Benchmark Results

These are the additional, aggregated results of the benchmarking process, as described in Chapter 5. The raw results are not included, as this would have been too much data.

The standard deviation was calculated over all runs for each parameter. It is typically rather low, except for some benchmarks. The deviation for application benchmarks is generally higher.

The microbenchmarks have also been tested on their scaling behaviour, by increasing the power of the respective variable, as described in the benchmarking chapter. The interval for each benchmark and variant differs and is given below:

- Simple Cycle
  - count: 2^7 - 2^{16}
  - size: 2^8 - 2^{17}
  - nongc: 2^8 - 2^{17}
- Cross Cycles
  - size: 2^5 - 2^{14}
  - cross: 2^5 - 2^{14}
  - count: 2^5 - 2^{14}
- Cross Reference (both directions)
  - count: 2^8 - 2^{17}
  - nonrc: 2^{10} - 2^{19}
  - rc: 2^8 - 2^{17}

The denoted values are the fitted exponent of an exponential growth function, which was calculated using scipy's curve_fit method. A value of one means, that the algorithm probably scales linearily, a value of two means, that the algorithm probably scales quadratically and a value of zero means, that the algorithm probably has a constant runtime.

Table 1: Benchmark Results - Standard Deviation - Simple Cycle 2^{14}

			0% alive				50% alive				100% alive			
			Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	count	2	0	0	2	2	2	2	2	2	1	1	1
		size	1	5	1	4	5	4	5	5	2	5	5	3
		nongc	5	0	0	0	7	1	3	4	1	0	0	1
		∅	**3**	**2**	**0**	**2**	**5**	**2**	**3**	**4**	**2**	**2**	**2**	**2**
	Max	count	10	0	0	0	11	0	0	0	9	13	3	0
		size	2	2	2	1	6	0	0	0	1	4	3	0
		nongc	2	0	0	0	4	5	0	5	4	0	0	4
		∅	**5**	**1**	**1**	**0**	**7**	**2**	**0**	**2**	**4**	**6**	**2**	**1**
Time (s)	GC	count	0.00	0.00	0.00	0.00	0.01	0.02	0.02	0.01	0.00	0.03	0.04	0.01
		size	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.01	0.00	0.01
		nongc	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		∅	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.01**	**0.01**	**0.00**	**0.00**	**0.01**	**0.02**	**0.00**
	Total	count	0.01	0.00	0.01	0.00	0.01	0.02	0.02	0.01	0.01	0.03	0.05	0.00
		size	0.00	0.00	0.01	0.00	0.01	0.01	0.00	0.00	0.00	0.01	0.01	0.01
		nongc	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.01	0.00	0.00
		∅	**0.00**	**0.00**	**0.01**	**0.00**	**0.01**	**0.01**	**0.01**	**0.00**	**0.00**	**0.02**	**0.02**	**0.01**
Pauses (ms)	Mean	count	0.08	0.03	0.04	0.03	0.13	0.28	0.06	0.07	0.08	0.48	0.11	0.05
		size	0.12	0.05	0.06	0.04	0.18	0.20	0.03	0.06	0.06	0.47	0.02	0.11
		nongc	0.06	0.06	0.07	0.02	0.10	0.18	0.05	0.09	0.06	0.17	0.09	0.04
		∅	**0.08**	**0.05**	**0.05**	**0.03**	**0.14**	**0.22**	**0.05**	**0.07**	**0.07**	**0.37**	**0.07**	**0.07**
	Max	count	1.1	0.2	0.3	0.5	1.5	6.3	0.8	4.3	0.8	9.3	2.2	2.1
		size	0.8	0.5	0.7	0.7	1.1	1.4	0.6	0.4	0.9	9.3	0.9	0.8
		nongc	1.0	0.6	0.3	0.4	1.7	0.7	1.3	0.7	0.7	1.9	0.8	0.4
		∅	**1.0**	**0.4**	**0.5**	**0.5**	**1.4**	**2.8**	**0.9**	**1.8**	**0.8**	**6.8**	**1.3**	**1.1**

Table 2: Benchmark Results - Standard Deviation - Cross Cycles 2^{14}

			0% alive				50% alive				100% alive			
			Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	size	14	1	1	1	8	5	4	6	10	3	10	5
		cross	6	2	6	7	8	4	10	3	10	12	7	9
		count	6	0	1	1	9	102	5	4	6	5	5	3
		∅	**9**	**1**	**3**	**3**	**9**	**37**	**6**	**4**	**9**	**7**	**7**	**6**
	Max	size	14	0	0	0	16	12	3	0	16	12	7	0
		cross	3	10	3	6	7	7	3	4	5	2	6	5
		count	5	0	2	3	7	14	1	0	5	1	2	0
		∅	**7**	**4**	**2**	**3**	**10**	**11**	**3**	**1**	**9**	**5**	**5**	**2**
Time (s)	GC	size	0.06	0.09	0.03	0.03	0.03	0.12	0.04	0.05	0.03	0.09	0.18	0.12
		cross	0.02	0.08	0.33	0.19	0.02	0.05	0.39	0.18	0.03	0.37	0.08	0.41
		count	0.03	0.03	0.03	0.03	0.02	0.93	0.08	0.06	0.02	0.11	0.04	0.07
		∅	**0.04**	**0.07**	**0.13**	**0.08**	**0.02**	**0.37**	**0.17**	**0.10**	**0.03**	**0.19**	**0.10**	**0.20**
	Total	size	0.11	0.09	0.03	0.04	0.03	0.14	0.04	0.06	0.02	0.09	0.19	0.13
		cross	0.03	0.08	0.34	0.18	0.02	0.05	0.39	0.16	0.03	0.40	0.08	0.43
		count	0.03	0.04	0.03	0.04	0.03	0.95	0.10	0.07	0.03	0.13	0.05	0.08
		∅	**0.06**	**0.07**	**0.14**	**0.09**	**0.03**	**0.38**	**0.18**	**0.10**	**0.03**	**0.21**	**0.11**	**0.21**
Pauses (ms)	Mean	size	0.24	0.17	0.04	0.05	0.10	0.39	0.04	0.10	0.09	0.31	0.10	0.16
		cross	0.09	0.22	0.00	0.00	0.07	0.17	0.00	0.00	0.13	1.44	0.00	0.00
		count	0.12	0.05	0.03	0.05	0.08	1.67	0.07	0.11	0.08	0.23	0.02	0.07
		∅	**0.15**	**0.15**	**0.03**	**0.04**	**0.08**	**0.74**	**0.04**	**0.07**	**0.10**	**0.66**	**0.04**	**0.08**
	Max	size	5.0	5.3	2.4	1.5	6.7	17.1	5.2	4.9	6.4	26.5	15.7	12.4
		cross	5.6	7.2	5.5	4.0	5.0	10.3	30.6	4.7	7.4	88.9	12.6	3.3
		count	7.7	0.5	0.4	1.5	4.6	270.4	14.2	1.6	6.0	20.2	2.9	5.4
		∅	**6.1**	**4.3**	**2.8**	**2.3**	**5.5**	**99.3**	**16.7**	**3.7**	**6.6**	**45.2**	**10.4**	**7.0**

Table 3: Benchmark Results - Standard Deviation - Cross Reference 2^{14} (rc to non-rc)

			0% alive				50% alive				100% alive			
			Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	count	2	0	3	3	7	4	4	1	7	2	2	2
		nonrc	6	5	1	8	6	1	8	6	2	5	7	8
		rc	1	5	1	4	6	4	3	1	6	3	5	5
		∅	**3**	**3**	**2**	**5**	**6**	**3**	**5**	**2**	**5**	**3**	**5**	**5**
	Max	count	5	0	1	0	5	1	2	0	6	0	4	3
		nonrc	1	3	2	4	2	0	10	8	3	1	0	0
		rc	0	0	1	1	3	0	0	0	3	4	2	1
		∅	**2**	**1**	**1**	**2**	**3**	**0**	**4**	**3**	**4**	**2**	**2**	**1**
Time (s)	GC	count	0.01	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.01	0.01	0.01
		nonrc	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01
		rc	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.00
		∅	**0.00**	**0.00**	**0.01**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.01**	**0.01**	**0.00**
	Total	count	0.01	0.00	0.00	0.01	0.01	0.01	0.00	0.01	0.01	0.01	0.01	0.01
		nonrc	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01
		rc	0.00	0.00	0.01	0.01	0.00	0.01	0.00	0.00	0.00	0.01	0.02	0.01
		∅	**0.00**	**0.00**	**0.01**	**0.00**	**0.00**	**0.01**	**0.00**	**0.00**	**0.00**	**0.01**	**0.01**	**0.01**
Pauses (ms)	Mean	count	0.13	0.06	0.04	0.04	0.10	0.18	0.03	0.09	0.09	0.25	0.05	0.06
		nonrc	0.05	0.06	0.05	0.05	0.05	0.04	0.07	0.06	0.04	0.09	0.05	0.08
		rc	0.07	0.05	0.16	0.10	0.06	0.19	0.02	0.02	0.13	0.27	0.07	0.07
		∅	**0.08**	**0.06**	**0.08**	**0.06**	**0.07**	**0.14**	**0.04**	**0.06**	**0.09**	**0.20**	**0.06**	**0.07**
	Max	count	0.9	0.9	0.2	1.2	0.6	4.6	0.9	0.2	1.1	9.2	1.4	0.6
		nonrc	0.3	0.1	0.0	0.2	0.3	0.2	0.2	0.1	0.2	1.8	0.9	1.7
		rc	1.3	0.5	0.5	0.3	0.6	2.0	0.7	0.2	0.8	4.4	1.9	1.6
		∅	**0.8**	**0.5**	**0.3**	**0.6**	**0.5**	**2.3**	**0.6**	**0.2**	**0.7**	**5.2**	**1.4**	**1.3**

Table 4: Benchmark Results - Standard Deviation - Cross Reference 2^{14} (non-rc to rc)

			0% alive				50% alive				100% alive			
			Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	count	6	3	4	4	8	5	5	5	3	1	3	2
		nonrc	8	1	4	6	5	5	1	1	1	6	8	5
		rc	5	0	0	0	6	1	4	4	7	5	4	3
		∅	**6**	**1**	**3**	**4**	**7**	**4**	**3**	**3**	**4**	**4**	**5**	**3**
	Max	count	5	0	0	0	11	8	5	4	7	9	4	4
		nonrc	4	2	0	3	5	4	3	2	2	4	5	2
		rc	3	0	0	3	4	0	0	0	6	7	0	0
		∅	**4**	**1**	**0**	**2**	**7**	**4**	**3**	**2**	**5**	**7**	**3**	**2**
Time (s)	GC	count	0.00	0.00	0.00	0.00	0.01	0.01	0.00	0.00	0.00	0.01	0.01	0.00
		nonrc	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		rc	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.00
		∅	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.01**	**0.01**	**0.00**
	Total	count	0.00	0.01	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.00
		nonrc	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		rc	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.01	0.01	0.00
		∅	**0.00**	**0.00**	**0.00**	**0.00**	**0.01**	**0.01**	**0.00**	**0.00**	**0.00**	**0.01**	**0.01**	**0.00**
Pauses (ms)	Mean	count	0.06	0.11	0.03	0.06	0.14	0.21	0.04	0.07	0.04	0.08	0.07	0.04
		nonrc	0.04	0.05	0.02	0.02	0.10	0.06	0.04	0.04	0.05	0.05	0.02	0.04
		rc	0.04	0.07	0.04	0.05	0.03	0.11	0.03	0.04	0.06	0.29	0.02	0.06
		∅	**0.04**	**0.08**	**0.03**	**0.05**	**0.09**	**0.13**	**0.04**	**0.05**	**0.05**	**0.14**	**0.04**	**0.05**
	Max	count	0.8	0.4	0.7	1.0	0.7	4.0	1.3	0.3	0.4	2.2	0.9	1.2
		nonrc	0.0	0.2	0.0	0.0	0.2	0.2	0.2	0.4	0.3	0.5	0.1	0.2
		rc	0.7	1.4	0.9	0.8	0.4	1.6	1.0	0.2	0.7	2.3	0.2	1.3
		∅	**0.5**	**0.7**	**0.6**	**0.6**	**0.4**	**1.9**	**0.8**	**0.3**	**0.5**	**1.7**	**0.4**	**0.9**

Table 5: Benchmark Results - Standard Deviation - Application Benchmarks

			Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	Gnome Music	2.80	1.95	0.61	0.54
		Gnome Tweaks	0.65	0.67	0.63	0.78
		PyChess	2.82	1.58	3.45	3.72
		Pygame	0.62	0.46	1.62	1.83
		Quod Libet	0.83	0.68	1.36	1.34
		∅	**1.54**	**1.07**	**1.53**	**1.64**
	Max	Gnome Music	3.41	2.25	0.61	0.84
		Gnome Tweaks	0.83	1.15	0.93	1.26
		PyChess	1.62	1.88	6.77	6.68
		Pygame	0.46	0.42	0.26	0.49
		Quod Libet	0.81	0.86	2.67	2.12
		∅	**1.43**	**1.31**	**2.25**	**2.28**
Time (s)	GC	Gnome Music	0.0030	0.0092	0.0066	0.0074
		Gnome Tweaks	0.0040	0.0062	0.0086	0.0032
		PyChess	0.1222	0.0268	0.0136	0.0141
		Pygame	0.0210	0.0466	0.0346	0.0504
		Quod Libet	0.0088	0.0110	0.0302	0.0268
		∅	**0.0318**	**0.0199**	**0.0187**	**0.0204**
Pauses (ms)	Mean	Gnome Music	0.42	1.18	0.67	0.78
		Gnome Tweaks	0.57	0.64	0.57	0.24
		PyChess	1.31	0.47	0.14	0.25
		Pygame	0.35	0.34	0.22	0.26
		Quod Libet	0.35	0.25	0.99	0.72
		∅	**0.60**	**0.58**	**0.52**	**0.45**
	Max	Gnome Music	2.2	6.3	3.3	2.7
		Gnome Tweaks	0.7	2.2	0.6	0.4
		PyChess	6.2	7.0	3.5	2.6
		Pygame	1.8	1.6	1.7	1.4
		Quod Libet	4.5	3.0	5.8	6.1
		∅	**3.1**	**4.0**	**3.0**	**2.6**

Table 6: Benchmark Results - Scaling - Simple Cycle

			0% alive				50% alive				100% alive			
			Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	count	0.8653	0.1693	0.1744	0.1706	0.8629	0.7111	0.7221	0.7128	0.8742	0.8298	0.8373	0.8367
		size	0.8246	0.5440	0.5315	0.5373	0.8250	0.7658	0.7755	0.7691	0.8229	0.8403	0.8404	0.8363
		nongc	0.8319	0.5785	0.5782	0.5811	0.8155	0.7734	0.7654	0.7666	0.8120	0.7963	0.7962	0.7979
	Max	count	0.8957	0.0826	0.0888	0.0891	0.8971	0.7871	0.7848	0.7843	0.9038	0.8933	0.9012	0.8973
		size	0.8875	0.3856	0.4091	0.4103	0.8864	0.7765	0.8099	0.7921	0.8861	0.8942	0.9033	0.8992
		nongc	0.8912	0.4065	0.4321	0.4320	0.8851	0.8117	0.7956	0.7934	0.8799	0.8697	0.8647	0.8682
Time (s)	GC	count	1.3271	1.0416	1.0344	1.0369	1.3211	1.5306	1.5445	1.5320	1.3331	1.3326	1.3046	1.3446
		size	1.2140	1.7069	1.6176	1.5899	1.2079	1.6024	1.4821	1.4634	1.1979	1.4199	1.3372	1.3317
		nongc	1.2061	1.9926	1.8079	1.8001	1.2062	1.5549	1.5284	1.5145	1.2068	1.2488	1.2466	1.2759
	Total	count	1.2350	1.0152	1.0137	1.0144	1.2335	1.4809	1.4875	1.4735	1.2425	1.3034	1.2712	1.3046
		size	1.1610	1.5284	1.4651	1.4438	1.1551	1.5427	1.4346	1.4176	1.1469	1.3860	1.3033	1.2975
		nongc	1.1537	1.8052	1.6203	1.6163	1.1557	1.4810	1.4445	1.4311	1.1565	1.2228	1.2146	1.2393
Pauses (ms)	Mean	count	0.5458	-0.0851	-0.0438	-0.0596	0.5466	0.7241	0.2034	0.5450	0.5526	0.6793	0.2128	0.5008
		size	0.5143	0.3516	-0.1265	0.2172	0.5114	0.7473	0.0524	0.4283	0.5088	0.7274	0.1410	0.4214
		nongc	0.5149	0.6646	0.4883	0.4853	0.5155	0.7400	0.6458	0.6348	0.5163	0.6618	0.6298	0.6461
	Max	count	1.1249	0.0359	0.0649	0.0399	1.1063	1.1060	1.1378	1.0302	1.1200	1.0905	0.9977	1.0461
		size	1.0307	0.7617	0.5973	0.6300	1.0229	1.1660	1.0060	0.9158	1.0071	1.1775	0.9979	0.9970
		nongc	1.0256	1.0791	0.6484	0.6458	1.0148	1.1385	1.1246	1.1245	1.0206	1.0437	0.9349	0.9513

Table 7: Benchmark Results - Scaling - Cross Cycles

		0% alive			50% alive			100% alive		
		Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	size	0.9161	0.6002	0.6566	0.5781	0.9172	0.8035	0.8634	0.8068	0.9223	0.8907	0.9446	0.9422
		cross	0.9880	0.6629	0.7950	0.8018	0.9846	0.9264	0.9044	0.9078	0.9922	0.9795	0.9333	0.9328
		count	0.9929	0.1387	0.2543	0.2516	0.9886	0.9014	0.8831	0.8550	1.0491	0.9802	0.9422	0.9104
	Max	size	0.9568	0.5117	0.6393	0.4374	0.9577	0.9022	0.8953	0.8417	0.9595	0.9539	0.9557	0.9550
		cross	0.9605	0.5762	0.8204	0.8218	0.9584	0.9297	0.8941	0.8953	0.9622	0.9512	0.9574	0.9598
		count	0.9608	0.0653	0.1747	0.1692	0.9610	0.9474	0.9218	0.9156	0.9624	0.9493	0.9598	0.9367
Time (s)	GC	size	1.0109	1.2483	1.1200	1.2254	1.0051	1.1245	1.1224	1.0755	1.0047	1.1403	1.1968	1.1205
		cross	1.0881	1.3012	1.2018	1.2120	1.0785	1.2950	1.2279	1.2523	1.0837	1.1642	1.2316	1.2949
		count	1.0816	1.0055	0.9948	0.9907	1.0773	1.1997	1.1622	1.0662	1.1704	1.1771	1.2517	1.1190
	Total	size	0.9933	1.1942	1.0893	1.1646	0.9850	1.0946	1.0935	1.0477	0.9852	1.1137	1.1588	1.0957
		cross	1.0506	1.2464	1.1509	1.1617	1.0425	1.2407	1.1842	1.2060	1.0475	1.1406	1.1940	1.2477
		count	1.0420	0.9854	0.9768	0.9709	1.0360	1.1588	1.1237	1.0406	1.1001	1.1517	1.2121	1.0990
Pauses (ms)	Mean	size	0.2355	0.1389	-0.0405	0.0389	0.2208	0.3080	-0.0679	0.1784	0.2017	0.3420	-0.1249	0.1749
		cross	0.2322	0.2978	-1.6326	-1.7668	0.2346	0.4623	-1.6365	-1.7382	0.2311	0.4786	-1.6299	-1.7564
		count	0.2050	-0.1661	-0.1565	-0.1729	0.1816	0.4078	0.0019	0.2168	0.1483	0.4389	-0.0478	0.1863
	Max	size	1.0180	0.9123	0.8648	0.6450	1.0078	0.9903	1.0061	0.9830	1.0183	1.1463	1.0234	1.0892
		cross	1.1117	1.2291	0.9199	0.9554	1.0995	1.3330	1.0872	1.0992	1.1110	1.1257	0.9814	1.0104
		count	1.1054	0.1628	0.2572	0.2637	1.1047	1.1611	0.9399	1.0271	1.2952	1.1464	1.0772	1.0476

Table 8: Benchmark Results - Scaling - Cross Reference (rc to non-rc)

			0% alive				50% alive				100% alive			
			Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	count	0.9144	0.1437	0.1747	0.1658	0.9105	0.8183	0.8701	0.8977	0.9150	0.9027	0.9553	0.9092
		nonrc	0.9534	0.9588	0.9711	0.9746	0.9537	0.9294	0.9268	0.9293	0.9516	0.9536	0.9759	0.9564
		rc	0.8309	0.5512	0.5371	0.5307	0.8201	0.7637	0.7985	0.7677	0.8237	0.8385	0.8356	0.8388
	Max	count	0.9510	0.0701	0.0938	0.0918	0.9433	0.8719	0.9057	0.9524	0.9482	0.9474	0.9523	0.9695
		nonrc	0.9696	1.0127	1.0383	1.0376	0.9714	0.9774	0.9654	0.9659	0.9703	0.9710	0.9702	0.9701
		rc	0.8931	0.3825	0.4101	0.4053	0.8869	0.7767	0.8109	0.7912	0.8840	0.8916	0.9026	0.9018
Time (s)	GC	count	1.0140	1.0053	1.0084	0.9728	0.9945	1.1563	1.1602	1.1605	1.0097	1.1294	1.2357	1.1499
		nonrc	0.9600	0.9814	0.9891	0.9712	0.9589	0.9568	0.9450	0.9299	0.9462	0.9554	0.9659	0.9506
		rc	1.2097	1.6740	1.5678	1.5924	1.2213	1.5610	1.4388	1.4865	1.1996	1.4315	1.3308	1.3501
	Total	count	0.9937	0.9885	0.9930	0.9568	0.9732	1.1158	1.1217	1.1153	0.9889	1.1059	1.1957	1.1201
		nonrc	0.9572	0.9713	0.9787	0.9591	0.9552	0.9531	0.9435	0.9257	0.9425	0.9523	0.9631	0.9468
		rc	1.1587	1.4974	1.4211	1.4434	1.1684	1.5028	1.3943	1.4385	1.1479	1.3950	1.2983	1.3160
Pauses (ms)	Mean	count	0.2791	-0.0666	-0.0848	-0.1024	0.2315	0.4268	0.0754	0.2008	0.2755	0.4446	0.0657	0.1969
		nonrc	0.0713	-0.0053	0.0614	0.0513	0.0719	0.0335	0.0920	0.0949	0.0743	0.0771	0.0566	0.1207
		rc	0.5120	0.3345	-0.1190	0.2330	0.5214	0.7182	0.0474	0.4354	0.5117	0.7224	0.1347	0.4428
	Max	count	0.9492	0.1924	0.1956	0.1924	0.9244	1.0708	0.9613	1.0675	0.9449	1.1688	1.0368	1.0415
		nonrc	0.9890	0.0492	0.0819	0.0674	0.9851	0.8589	0.6878	0.9025	0.9717	0.9667	0.5315	0.9977
		rc	1.0114	0.7268	0.5673	0.6271	1.0250	1.1199	0.9915	0.9446	1.0157	1.2109	1.0125	1.0060

Table 9: Benchmark Results - Scaling - Cross Reference (non-rc to rc)

			0% alive				50% alive				100% alive			
			Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k	Base	Mark	Inc5k	Inc50k
Memory (MB)	Mean	count	0.8523	0.1486	0.1529	0.1515	0.9002	0.8228	0.8841	0.8305	0.9247	0.8865	0.9425	0.8728
		nonrc	0.9641	0.9634	0.9737	0.9742	0.9294	0.9396	0.9401	0.9369	0.9739	0.9743	0.9732	0.9758
		rc	0.8113	0.6811	0.6838	0.6828	0.8229	0.7709	0.7758	0.7690	0.8143	0.8323	0.8367	0.8401
	Max	count	0.8912	0.0755	0.0815	0.0786	0.9334	0.9151	0.8878	0.8433	0.9471	0.9489	0.9345	0.9335
		nonrc	1.0125	1.0127	1.0383	1.0363	0.9529	0.9572	0.9544	0.9513	0.9688	0.9684	0.9696	0.9685
		rc	0.8798	0.5381	0.5595	0.5655	0.8880	0.7848	0.7934	0.7871	0.8801	0.8877	0.9035	0.8973
Time (s)	GC	count	1.3971	1.0269	1.0012	1.0033	1.0681	1.1147	1.1905	1.0733	1.0355	1.0975	1.2160	1.0603
		nonrc	0.9931	0.9855	0.9915	0.9936	0.9693	0.9624	0.9557	0.9666	0.9835	0.9819	0.9758	0.9648
		rc	1.2191	1.9626	1.7750	1.8012	1.2007	1.4949	1.4184	1.4498	1.2177	1.3916	1.2943	1.3313
	Total	count	1.2797	0.9913	0.9774	0.9740	1.0183	1.0683	1.1294	1.0401	1.0069	1.0730	1.1697	1.0406
		nonrc	0.9826	0.9767	0.9822	0.9834	0.9630	0.9567	0.9514	0.9622	0.9785	0.9765	0.9711	0.9613
		rc	1.1641	1.8310	1.6659	1.6805	1.1461	1.4477	1.3779	1.4079	1.1655	1.3581	1.2639	1.2975
Pauses (ms)	Mean	count	0.5022	-0.0577	-0.1415	-0.1669	0.0997	0.1742	-0.0235	0.1900	-0.0691	0.1662	-0.1930	0.0651
		nonrc	-0.0115	-0.0067	0.0620	0.0508	-0.0015	0.0089	0.0540	0.0472	0.0002	-0.0046	0.0178	0.0087
		rc	0.5218	0.7110	-0.0277	0.3723	0.5107	0.7207	0.0627	0.4315	0.5219	0.7040	0.1403	0.4324
	Max	count	1.0603	0.2787	0.2619	0.2655	1.0791	1.0540	1.0677	1.0005	1.0740	1.0724	1.0680	0.9339
		nonrc	0.0746	0.0735	0.1003	0.0968	0.0936	0.0875	0.0968	0.0911	0.0727	0.0813	0.0848	0.0783
		rc	1.0380	1.0101	0.7476	0.7641	1.0218	1.0889	0.9686	0.9803	1.0229	1.1484	0.9723	1.0012