Retro-fitting Conservative Garbage Collection

Jacob Hughes

Thesis submitted in partial fulfilment for the degree of
Doctor of Philosophy

Faculty of Natural & Mathematical Sciences
Department of Informatics
King's College London

April 2023

Abstract

Garbage Collection (GC) is often considered to be an inherent property of a programming language, or a particular implementation of a language. While it is possible to retrofit a conservative GC to non-GC'd language implementations, the performance of the resulting implementation is considered to be slow, and the impact on the language itself non-idiomatic.

In this thesis I show that conservative GC has more applications than it is traditionally thought suitable for. First, I show that the implementation of V8, an industrial strength Virtual Machine (VM), can be simplified without worsening performance by retrofitting conservative GC in place of its existing precise GC which uses bookkeeping information available at runtime. Second, I show how conservative GC can be idiomatically added to the non-GCed language Rust. The latter raises several challenges around correctness and performance: I show each challenge has a good solution. My aim in this thesis is not to suggest that conservative GC is the right solution for all circumstances, but that it is, at least, often a better solution than has previously been thought.

Acknowledgements

First and foremost, I would like to thank my supervisor Laurence Tratt. His support, both academically and personally throughout my PhD has been invaluable and this thesis would not have been possible without him. I could not have asked for a better supervisor.

To my friends and colleagues at King's who have helped make the last few years enjoyable: Andrei, Iti, Lukas, and Pavel. A special thanks to Edd for always jumping on a call to help debug in times of need. I don't want to think about how many hours you have saved me!

I would like to thank Hannes, Michael, Anton, Nikos, and the rest of the V8 team at Google Munich. First, for offering me the incredible opportunity to intern with them twice, but also for putting up with my constant questions as I fought my way through many self-inflicted broken V8 builds.

Thanks to all my friends in Brighton and back home. There are too many of you to mention, but in particular: Amy, Sep, Tom, and Joe, who have all helped me when things got difficult, I am very grateful. A special thanks to Paul for providing me with a sofa to sleep on for a couple of months!

And finally, to my wonderful family: Mum, Dad, Debbie, Nan, Granddad, and of course my brother Jack. I couldn't have done it without your support. I love you all.

Contents

Chapter 1

Introduction

Garbage Collection (GC) is often considered to be an inherent property of a programming language implementation. It is possible to retrofit GC onto uncooperative language implementations using *conservative GC* [Boehm and Weiser, 1988], but performance is considered to be slow. In this thesis I show that conservative GC has wider uses than traditionally thought. My aim is not to suggest that conservative GC is the right solution for all circumstances, but that it is, at least, often a better solution than has previously been thought.

First, I modify a significant portion of V8 – a widely used, high performance, JavaScript VM – to show that semi-conservative GC can make Virtual Machines (VMs) less error-prone, with equivalent performance, to precise GC.

Second, I show that conservative GC can be idiomatically added to Rust, a non-GC'd language that uses the concept of *ownership* (based on affine types [Pierce, 2004]) to guarantee memory safety. This approach allows Rust programmers to write memory safe programs without runtime overhead. However, Rust's type system is too restrictive to express programs which directly operate on cyclic data structures. Instead, workarounds such as the use of weak reference counting or arenas [Goregaokar, 2021a] can protect the user from dangling pointers, but cannot prevent memory leaks, and are difficult to write. Because of this, there have been multiple attempts to add GC to Rust (see [Goregaokar, 2021b] for an overview of several of these), but the results either have significant ergonomic, performance, or soundness issues.

In this thesis I introduce a new approach to adding GC to Rust — ALLOY. At the language level, ALLOY provides a natural interface to programmers, making use of conservative GC at lower levels. However, ALLOY raises several challenges around correctness and performance, mostly related to finalisation, several of which are unique to, or exacerbated by, Rust. I show that these challenges have good solutions and that it is possible to extend Rust to statically rule out problematic finalisers, and that a series of optimisations can elide most finalisers.

1.1 Motivation

The majority of low-level software is written in systems programming languages such as C/C++ which use *manual memory management.* In such languages, the programmer is responsible for explicitly managing allocation and deallocation of objects in their program. This means that they must request memory for objects when they are created, and then release that memory when the object is no longer needed. This places a burden on the programmer to ensure that programs are *memory safe*: that is, they must be careful not to use memory they do not have access too, or free memory while it is still in use. Violating memory safety in this way has both correctness and security implications: both Microsoft and the Chromium project report that memory safety violations account for around 70% of their security vulnerabilities [Microsoft, 2019; The Chromium Developers, 2022].

An alternative is to use garbage collection: an automatic approach to managing memory. GCs typically come in two forms: *reference counting*, which counts the number of incoming references for each object, deallocating them when their counts reaches zero; and *tracing garbage collection*, where references in the program are periodically traced in order to find objects. Both kinds of GC are memory safe because they restrict the programmer's control over deallocation.

While manual memory management and garbage collection are fundamentally different, there are cases when they need to be used together. Systems programming languages such as C++ are often used to implement language virtual machines (VMs) for performance reasons. However, where the target language requires a tracing garbage collector, some form of GC is also needed to manage objects written in C++ code. For example, JavaScript VMs such as V8 [Google, 2008], SpiderMonkey [Mozilla, 1995], and JavaScriptCore [Apple, 2013] all use some form of GC internally to manage objects which are accessible from within their C++ code.

Garbage collection is also used where managing memory manually is too difficult. Rust – a systems programming language which uses the concept of *ownership* for memory safety – makes it very difficult to write data structures which contain cycles. This has motivated an active design space for some form of retro-fitted tracing collection [Goregaokar, 2021b]. This desire for non-GC'd languages to provide an opt-in form of GC has led to several retro-fitting attempts [Boehm and Weiser, 1988; Baker et al., 2007; Ager et al., 2013; Goregaokar, 2015]. However, for languages which were not designed with GC in mind during their inception, this is often fraught with both performance and correctness challenges.

1.1.1 The problem with retro-fitting garbage collection

Pointer identification

A major challenge when retro-fitting tracing garbage collection is finding the references to objects in the program. Systems languages and compilers make this difficult, or impossible, as pointers to objects can be 'lost' on the call stack or in the heap. The ideal, and universal, solution would be for such languages to allow *introspection* of a program's memory layout, telling it, for example, where each local variable lives within a frame, and which one is a reference to an object. Unfortunately, no mainstream systems language defines an introspection capability.

Instead, a retro-fitted GC can identify pointers *conservatively*: that is, the call stack and allocated blocks in the heap are exhaustively examined for possible pointers to objects. This requires the GC to know at what address a thread's stack began, and what address the stack is currently at. Each aligned word on the stack is checked to see whether it points to an instance of an object: if it does, that object is further traced conservatively to find other objects. Depending on the GC, checking whether an arbitrary word is a pointer to an object can be fairly expensive, particularly if the GC requires interior pointers to objects be translated to base pointers.

Other attempts to retro-fit tracing GC to languages rely on providing the collector with bookkeeping information, allowing it to identify pointers *precisely*. Precise pointer identification can take on two forms. First, the compiler can be modified to emit the location of pointers for the collector to use at runtime [Diwan et al., 1992]. Second, programmers interacting with GC'd objects can use *handles*, that is wrappers around object references, such that the handles inform the collector of the location of objects [Brooks, 1984]. Handles are less invasive, as they do not require modifications to the language, but they place the burden of correctness onto the programmer.

For performance reasons, some GCs [SUN Microsystems, 2006; Bartlett, 1988; Apple, 2013] use a hybrid of both conservative and precise pointer identification. This often involves precisely identifying pointers within objects on the heap, but using conservative identification for pointers on the stack or within registers where it is more difficult to provide bookkeeping information at runtime. Such GCs are often called *semi-conservative*.

Finalisation

When an object can be proven to no longer be used, the collector can run the object's *finaliser* (e.g. to close file descriptors or database connections). VM authors must implement finalisation carefully to prevent a number of known correctness problems [Boehm, 2003]. Unfortunately, such problems can be exacerbated when GC is retro-fitted to non-GC'd languages. In Rust, for example, finalisers present new problems, as they can violate the tight restrictions the language places on how references can be used [Matsakis, 2013]. For example, Rust references have a clearly defined lifetime guaranteed by the compiler. However, if used from within finalisers, which are called non-deterministically, the reference may be invalid.

1.1.2 A case for retro-fitted conservative GC

Conservative scanning is often eschewed for two reasons: first, it is considered slower, since it prohibits many GC optimisations which would move objects around in memory (though semi-conservative GCs may still move objects which are not pointed to conservatively [Bartlett, 1988]); and second, it has a reputation for overestimating which objects might be live, leading to longer than desired memory retention [Zorn, 1993]. Fortunately, there is now good evidence that conservative scanning tends to only slightly over-approximate the values which may be pointers [Shahriyar et al., 2014].

In this thesis I show that the performance of retro-fitted conservative collection is not as bad as previously thought. I do this by retrofitting semi-conservative GC into V8, an industrial strength JavaScript VM Machine (VM) which uses handles. I am able to show that, despite only migrating about half of V8's source code, conservative GC using direct references is no slower than the existing precise approach which uses handles. I also show that switching to conservative GC makes V8 more *ergonomic*. That is, it allows VM authors to write less code while using more idiomatic, and less error-prone C++ code.

I also show that retro-fitting conservative GC can be used to address two key ergonomic issues in Rust: writing cyclic data structures; and using Rust as a language for writing VMs. In this thesis, I introduce ALLOY, a new GC for Rust. ALLOY is sound from a Rust language perspective (though dependent on the compiler tool-chain to respect conservative GC's limitations), and has reasonable performance. It also allows Rust data structures with cycles to be expressed with ease, and can also be used in conjunction with existing memory management strategies in Rust. ALLOY uses tracing, conservative GC provided by the Boehm-Demers-Weiser (BDW) collector [Boehm and Weiser, 1988], though it introduces many Rust specific optimisations which are independent of the underlying collector. ALLOY also introduces a novel technique, known as *finaliser safety analysis*, which extends the Rust type system to reject programs whose finalisers are unsound.

ALLOY's performance is evaluated in several different ways: SOMRS and WLAMBDA, two externally written reference counted interpreters which I have retrofitted to use ALLOY; and an artificial heap allocating benchmark which we use to compare ALLOY against various existing GC implementations in Rust.

1.2 Contributions

The main contributions of this thesis are as follows:

1. A source transformation and performance comparison between two methods of representing pointers to garbage-collected objects in V8 – an industrial strength JavaScript VM.
2. The design of ALLOY– a tracing garbage collector for Rust.
3. Introducing the novel concept of finaliser safety analysis, showing that static soundness guarantees for finalisation in a GC for Rust can be achieved by extending the compiler and type system.
4. An optimisation in ALLOY to remove unnecessary finalisers in a GC for Rust where garbage collected and non-garbage collected objects are used together.

1.3 Publications

The majority of Chapter 3 forms a draft paper where I was the lead researcher:

- Jacob Hughes, Michael Lippautz, Hannes Payer, and Laurence Tratt. *Comparing the Performance of Handles and Direct References on VM Performance*

1.4 Synopsis

The structure of this thesis is as follows: Chapter 2 introduces the concepts in memory management needed to understand the rest of this thesis. It gives a brief overview of the two fundamental approaches to managing memory: manual memory management, and automatic memory management. It also provides background information about the different kinds of garbage collection algorithms available and their relative advantages and disadvantages. Chapter 3 explains the different ways a tracing garbage collector can identify pointers. The main focus of this chapter is a performance comparison between two ways a VM can refer to objects on the garbage collected heap: handles, which use a level of indirection to refer to objects, and directly referencing them while using conservative stack scanning during GC. Though both approaches are well established, the performance trade-offs are unclear. This chapter details how I migrated around half of V8 – an industrial strength JavaScript VM – to refer to objects using direct references instead of handles, with a performance evaluation which shows that even though the migration is partial, direct references are at least as fast as handles on the SPEEDOMETER2.1 benchmark suite. Chapter 4 gives an overview of the Rust programming language and how it uses ownership to manage memory without a garbage collector. It also shows how Rust programs can be written with reference counting or arenas where the ownership model does not fit. Chapter 5 gives an overview of where tracing garbage collection can be useful in Rust. The main focus of this chapter is the introduction of ALLOY– a conservative GC based on the Boehm-Demers-Weiser GC [Boehm and Weiser, 1988] I wrote for Rust so that writing cyclic data structures and implementing other GC'd languages in Rust is more ergonomic. Chapter 6 explains how finalisation works in ALLOY. It details the soundness and performance challenges that finaliser present, some of which are unique to a GC in Rust. The main focus of this chapter is on finaliser safety analysis (FSA), one of the main contributions of this thesis. It explains how FSA extends the Rust type system to ensure that finalisers in ALLOY are sound. This chapter also shows how finalisers in ALLOY can be optimised based knowledge from the Rust type system. Chapter 7 presents a performance evaluation of ALLOY.

Chapter 2

Background

Computer memory is finite, and for non-trivial programs it is not possible to know statically how much memory will be allocated on the heap. Over the course of a program's lifetime, its memory needs may go up and down many times. This means that for most programs some form of memory management is needed — that is, a way to allocate memory for the program when required, and to free it when no longer needed. There are two approaches to this: *manual memory management* and *automatic memory management*.

This chapter gives the necessary background for the reader to understand the thesis. Section 2.1 starts by explaining manual memory management. Section 2.2 introduces two forms of automatic memory management: reference counting and tracing garbage collection. Section 2.3 shows how the latter can be implemented in real systems. Finally, Section 2.4 shows that, in real systems, a combination of various different techniques is often used. Note that this chapter provides an overview of the memory management landscape, and that Chapter 3 and Chapter 4 contain in-depth background sections which are more specific to the contributions in this thesis.

2.1 Manual memory management

Languages such as C/C++ use manual memory management, where the programmer explicitly manages allocation and deallocation for objects in their program. This means that the programmer must request memory for objects when they are created, typically via explicit calls to the allocator (`malloc` / `new` in C/C++), and then release that memory when the object is no longer needed (`free` / `delete` in C/C++). In non-trivial programs, requiring the programmer to manually track and free memory can be complex and error-

```
 1 #include <stdlib.h>
 2
 3 int main() {
 4     int** arr = malloc(sizeof(int*) * 10);
 5     if (!arr) {
 6         return 1;
 7     }
 8
 9     // allocate memory for each row
10     for (int i = 0; i < 10; i++) {
11         arr[i] = malloc(sizeof(int) * 10);
12         if (!arr[i]) {
13             break;
14         }
15     }
16
17     free(arr);
18
19     // At this point the program has lost access to the memory located
20     // in the for loop, which constitutes a leak.
21
22     return 0;
23 }
```

Listing 2.1: An example of a memory leak in C. `main` creates a two-dimensional array by first allocating enough space for 10 inner rows (line 4), and then allocating additional memory for each row using a loop (line 11). At the end of main, the initial memory allocated for the array is freed (line 17) but not memory allocated for the inner rows, resulting in a memory leak.

prone. There are two primary considerations a programmer must think about when using a language with manual memory management: *memory safety* and *memory leaks*.

2.1.1 Memory safety

Memory safe programs only dereference valid pointers. This section explains the most relevant form of memory safety violations to this thesis: *use-after-free* violations.

Freeing memory too early means that it is deallocated while the program still holds a reference to it. This reference becomes a *dangling pointer*, because the location it points to is no longer valid memory. The correctness of the program is violated if the program then uses this reference later on. This could manifest as a crash because the hardware notified the OS of an access violation (typically a segmentation fault in the case of Unix-like OS's, or a general protection fault in Windows). In other cases, more subtle and hard to debug errors can arise, such as a memory corruption where the program appears to continue working normally, but reads from (or writes to) incorrect places in memory. This kind of error can be exploited in attacks for reasons such as privilege escalation [Xu et al., 2015].

A variant on this is a *double free*, where an allocator's `free` function is called more than once on the same reference. This may corrupt the state of the allocator, or free memory in use by another part of the program. As with use-after-free violations, this is due to a dangling pointer, and is a common source of security vulnerabilities. Fortunately, several dynamic tools have been developed to help to identify and prevent use-after-free vulnerabilities [Serebryany et al., 2012; Nethercote and Seward, 2007; Ainsworth and Jones, 2020].

2.1.2 Memory leaks

The second kind of concern in manually managed languages are *memory leaks*.[1] Memory leaks occur when memory which is no longer used is not released. The most common example of this can be seen in Listing 2.1, where the programmer forgets to call `free` (or equivalent) on memory that is no longer needed. In addition to causing a program to consume more memory than it needs to, memory leaks can cause performance problems because of poor spatial locality of objects. In the worst case scenario, memory leaks can cause out-of-memory (OOM) crashes, or slow programs to a crawl because the requested memory is too large to fit into RAM and is regularly swapped in from the disk (known as *page thrashing*). Memory leaks are difficult to detect as they are often subtle, and do not cause an immediate crash. As with use-after-free detection tools, there also exist dynamic tools help to identify and prevent memory leaks [Nethercote and Seward, 2007; Novark et al., 2009; KDE, 2013].

2.1.3 RAII

To help deal with some of the pitfalls of manual memory management, C++ developed a programming idiom known as *Resource Acquisition is Initialization* (RAII) [Stroustrup, 1997, Section 14.4]. RAII makes it harder to introduce memory bugs by tying object lifetimes to resource management. In other words, the *constructor* of an object is responsible for acquiring the resource (whether that is a heap allocation, file handle, database socket, etc.), whereas the *destructor* is responsible for releasing that resource. This means that as long as the object is alive, the resource is available.

C++ has specific language support for RAII: an object's destructor is called automatically when it goes out of lexical scope. When used to manage a heap allocation, `delete` (C++'s deallocation operator) can be called from inside the object's destructor. This allows the programmer to use *scope-based deallocation*, instead of relying on the more error-prone

[1]Though, as we will see, they are not unique to languages with explicit deallocation!

```
 1 #include <iostream>
 2
 3 class Employee {
 4 private:
 5     int* id; // a pointer to an int, which is allocated on the heap.
 6 public:
 7     // Constructor
 8     Employee(int id) {
 9         this->id = new int;
10         *this->id = id;
11     }
12
13     // Destructor
14     ~Employee() {
15         std::cout << "Employee␣" << *id << "␣destructor␣called" << std::endl;
16         delete id; // frees the heap allocated int.
17     }
18 };
19
20 int main() {
21     Employee alice(1);
22
23     {
24         Employee bob(2);
25     } // bob's destructor called.
26
27     return 0;
28 } // alice's destructor called.
```

Listing 2.2: An example of using RAII to manage a heap allocated integer. The `Employee` class has a single member, `id` which is a pointer to an `int`. The `Employee` constructor allocates this value on the heap (line 9), and its destructor deallocates it (line 16). When `main` creates the `Employee` instance `alice`, its destructor is run at the end of `main` automatically, freeing its heap allocation. The `bob` instance is created in an inner scope, so its destructor is run first.

method of explicitly calling `delete` when they think they are finished with it. Listing 2.2 shows an example of how RAII can be used to automatically deallocate an object's memory when it goes out of scope.

2.2 Automatic memory management

Automatic memory management is a programming language feature where unused memory is reclaimed automatically, relieving the programmer of the burden of having to do this themselves. This is also known as *garbage collection*.[2] There are two fundamental

[2]The term GC is somewhat overloaded, and can mean different things to different people. It is generally understood that garbage collection encompasses both forms of automatic memory management [Bacon et al., 2004], however, most uses of the term GC in literature refer specifically to tracing garbage collection. This thesis will adopt a similar convention henceforth: using GC to refer exclusively to tracing garbage collection, unless otherwise stated.

approaches: reference counting; and tracing garbage collection. This section will give a basic overview of both approaches, their relative advantages and disadvantages, and why one would choose one over the other.

2.2.1 Reference counting

Reference counting was first used in 1960 [Collins, 1960]. As the name suggests, it works by maintaining a "count" of the number of references that point to each object. When a new reference to an object is created, the reference count for that object is incremented. Conversely, when a reference to an object is destroyed, the count is decremented. When an object's reference count reaches zero, it is no longer reachable, and its memory is reclaimed.

Advantages and disadvantages

The main advantage of reference counting is that an object can be freed as soon as the last reference to it is destroyed. This makes it very appealing for real-time applications where low latency is important to maintain responsiveness (e.g. in GUI programming where responsiveness is important for operations such as closing windows or dialogue boxes). Reference counting can also be simpler to implement than tracing garbage collection, as the latter usually requires platform specific knowledge to identify pointers.[3]

Reference counting has several disadvantages. First, the overhead of incrementing and decrementing the count has a performance overhead on write operations. Not only does this require extra instructions for updating the count in potential hot paths, but it can also pollute the cache [Jones et al., 2016, p. 59]. In addition, if thread-safety is required, reference count operations have a higher overhead as they must be performed atomically.

Second, reference counting requires an extra machine word to store the count for each object. This is because, in the worst case, the number of references to an object could be equal to the number of objects in the heap, so the count must be word sized. In languages such as Java, where objects tend to be small but vast in quantity [Dieckmann and Hölzle, 1999, p. 16], the count overhead can quickly add up.

[3]Of course, high performance reference counted implementations are anything but simple: *deferred* reference counting implementations can remove much of the count overhead from temporaries on the stack [Blackburn and McKinley, 2003]; and *coalesced* reference counting can can often remove count operations that cancel each other out [Shahriyar et al., 2012]. Some systems are even able to replace atomic count operations with non-atomic ones where they can guarantee that this does not affect thread-safety [Ungar et al., 2017].

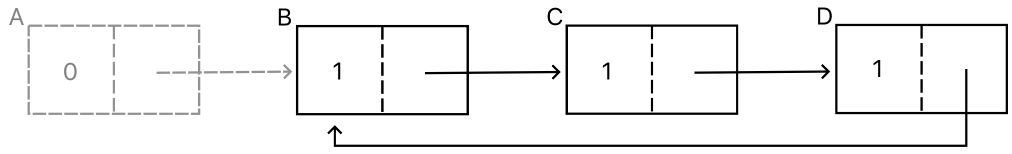


Figure 2.1: An example of a graph of objects which will never be reclaimed due to a cyclic reference. Even after **a** is deleted, objects **b** still maintains a count of one because of **d**'s reference to it.

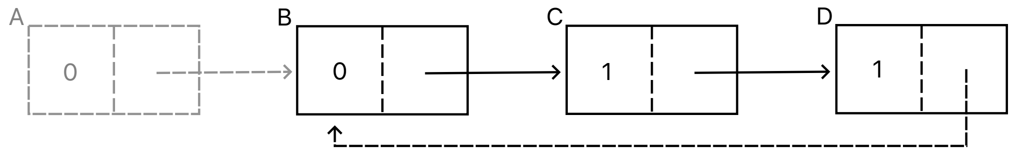


Figure 2.2: An example of using a weak reference to break a reference cycle. **b** is now pointed to by a weak reference from **d**, allowing all objects in the cycle to be freed.

Third, when a large chain of objects are recursively deallocated at once, a reference counted system can cause noticeable pauses to the application. If deleting a reference removes the root to a large graph, the application thread will be blocked from progressing while each node is freed one-by-one. Worse still, this can block all threads in multi-threaded systems if locks are involved during deallocation [Boehm, 2004].

Finally, a fundamental problem with naive reference counting is that it cannot free objects with cyclic references. In other words, an object which refers to itself – directly or indirectly – will never be reclaimed in a naive reference counting system (see Figure 2.1 for an example). This is because even if all other references are removed, the count will never reach zero. Without some kind of additional support, none of the objects in a cycle can be freed, causing a memory leak. For this reason, many reference counting systems use some other mechanism to handle cycles.

In the next subsections, two possible approaches to overcoming this problem are shown: using weak references, and an algorithm for cycle reclamation.

Weak references

Weak references are references which do not form part of the count, so do not protect an object from being deallocated. Unlike regular *strong references*, a weak reference can be used to break the link in cycles of strongly connected objects (shown in in Figure 2.2).

The problem with using weak references is that the burden is placed on the programmer to ensure they do not leak memory, requiring them to correctly break cycles. In large graph-like data structures this can be difficult to keep track of.

Cycle collection

There have been several proposed solutions to allowing a reference counting system to automatically reclaim data structures made up of objects which contain cyclic references without programmer intervention [Lins, 1992; Bacon and Rajan, 2001]. Each algorithm has different performance characteristics, but they are based on the same two fundamental observations. The first observation is that the most interesting operation to cycle detection is when a reference count is decremented to a non-zero value as this is the state where a cycle may exist. If a reference count is decremented to zero, the garbage has been found, and cycle detection is not necessary.

The second observation is that all reference counts in a garbage cycle are *internal*. In other words, if the counts between objects in a cycle are subtracted, and there aren't any externally incoming references, then they will cancel each other out, identifying it as a garbage cycle.

Based on these observations, what each cycle detection algorithm has in common is that if an object's reference count is decremented and does not reach zero, then it is a potential *root* of a garbage cycle. These roots can be added to a work-list, and at some later stage can be the starting points for a local cycle detection scan. This scan involves a depth-first-search, where every reachable object is decremented to see if their counts reach zero. If zero counts are discovered, then the second observation holds, and the cycle of objects can be considered a garbage cycle.

2.2.2 Tracing garbage collection

Tracing garbage collection was first developed in 1960 as a way of automatically managing memory in Lisp [McCarthy, 1960]. In order to know which objects to collect, a garbage collector must know which objects are going to be accessed at some point in the program. Such objects are referred to as *live*. True *liveness* is an undecidable problem, so tracing garbage collection uses pointer reachability as a conservative approximation. An object is reachable if it can be reached either directly, or indirectly (by following a chain of pointers from fields in other objects) from a set of known roots in the program. An object which is no longer reachable is considered dead and a garbage collector can reclaim its memory at some point in the future.

Tracing GCs reclaim garbage periodically, which means that there is a delay between when an object was last used, and when its memory becomes available for future reuse. This lag is referred to as *heap drag* [Röjemo and Runciman, 1996]. Reclaiming garbage is not cost-free, so regardless of how many objects exist there is a performance and space trade-off: more frequent garbage collections will reduce the amount of heap drag, but will end up having to do more work.

Finalisation

Finalisation is the term given in tracing garbage collection to the process in which objects run methods to execute some cleanup code before they are reclaimed. Such methods are referred to as *finalisers*.

Finalisers are similar to destructors in manually managed languages with RAII, or reference counted systems. However, GC literature often distinguishes them due to the key difference that destructors are called deterministically on object destruction, whereas finalisers are run non-deterministically. In other words, a tracing GC does not reclaim objects immediately after they are unreachable, but instead at some point later in time. This means that all we can know about a given object's finaliser is that it may run at some point in the future. Listing 2.3 shows a simple example of a using a finaliser to close a file handle in Java.

Advantages and disadvantages

As with reference counting, one of the main advantages of using tracing garbage collection instead of manual memory management is that it allows programmers to write programs which are memory safe,[4]. Use-after-free violations are not possible with tracing garbage collection because, by definition, dangling references cannot occur in a tracing collector: an object which has a reference to it is considered reachable, so it will not be reclaimed. In addition, because reclaiming memory is the responsibility of the collector, tracing GC is often easier to use.

Tracing garbage collection can deal with data structures which include cycles. In contrast to reference counting, from the programmers perspective, cyclic data structures "just work", with no extra effort needed to ensure they are collected. This is due to how a tracing GC determines how an object is reachable (explained in Section 2.3).

The main disadvantage of tracing garbage collection is that because of the collector runtime, it may be unsuitable for applications which have constrained memory or require real-time responsiveness.

[4]However, in Section 5.3.2 I show how incorrectly using a particular kind of tracing garbage collection – known as *conservative* collection – can actually violate memory safety.

```java
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class FileHandler {
    private File file;
    private FileWriter writer;

    public FileHandler(String filename) {
        file = new File(filename);
        try {
            writer = new FileWriter(file);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void write(String data) {
        try {
            writer.write(data);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void finalize() throws Throwable {
        try {
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            super.finalize();
        }
    }
}
```

Listing 2.3: Using a finaliser to close a file handle in Java.

Memory leaks are also possible in tracing garbage collection – though far less likely than its manually managed or reference counted cousins. It is possible for programmers to accidentally hold on to a reference to an object (e.g. via a global variable), preventing anything referenced by it from being collected.

2.3 Tracing garbage collection implementation

In this section, I briefly explain the three main algorithms for tracing GC: mark-sweep; mark-compact; and semi-space collection. Each of these algorithms have different performance and space trade-offs and, as will become apparent, there is overlap between the techniques which they use. One particular trade-off in contention between various implementations is *throughput* (how much work can be done in a given time) and *latency* (the responsiveness of the application). Unfortunately, as we will see, these factors are often in tension with each other. High performance industrial strength GCs such as those used in Java's HotSpot JVM or JavaScript's V8 usually use a combination of different implementation approaches.

When talking about GC, it is useful to separate the core of the GC from the user program. The term *collector* refers to the system component responsible for garbage collection, whereas the *mutator* is used to refer to the user program (so called because from the point of view of the collector it simply mutates the object graph [Dijkstra et al., 1978]).

2.3.1 Mark-sweep garbage collection

All tracing algorithms share one thing in common: in order to free garbage, they must be able to locate all the reachable (i.e. live) objects in a program. An object is considered reachable if it is referenced – directly or indirectly – from a set of known *roots*. To achieve this, the algorithm for a mark-sweep GC can be split into two distinct *mark* and *sweep* phases. In the mark phase, a transitive closure over the object graph is performed. That is, starting from a set of roots – references which form the entry points in to the object graph – the collector traces through each reference it finds, marking each object along the way. Once this is finished, all reachable objects will have been marked and are thus considered live.

The sweep phase then scans the heap linearly and frees all objects which were not marked. Marked objects are reset ready for the next collection. This can be done efficiently by flipping the meaning of the mark bit from one collection to the next.

In its most naive form, mark-sweep garbage collection would pause the mutator entirely in order to perform a collection, before handing control back to the mutator to resume the program. Such a process is referred to as *stopping the world*. This is fine when throughput is the only concern, but in more modern applications, lower latency is often desired, reducing the acceptable time for the collector to pause the mutator.

One technique to lower latency is to perform *incremental marking* [Baker Jr and Hewitt, 1977], where instead of one big stop the world pause, the program is interrupted more frequently to perform smaller chunks of marking which can interleave with the mutator. In the common case, this can improve the latency of a program, but long pauses can still happen under memory pressure when the collector tries to keep up with a large number of allocations.

Unfortunately, there is no free lunch, and incremental marking comes at the cost of throughput: the mutator can change the state of the object graph after each chunk of marking happens so it must be able to inform the collector of this. The mutator does this using a small fragments of code before write operations known as *barriers* [Pirinen, 1998]. These are used to give the collector a chance to do some work, and to ensure that certain invariants in the object graph are maintained. The exact barriers used depend on which invariants the collectors wish to maintain, a topic which is outside the scope of this thesis.

A mark-sweep collector's latency can also be optimised by making it *concurrent*. A concurrent collector is one which can perform marking or sweeping (or both) at the same time as the mutator. In modern GC implementations, most of the marking can be done concurrently to the mutator, and very short pauses are needed only to scan a thread's call stack [Flood et al., 2016]. Concurrent garbage collection is notoriously difficult to implement: synchronisation primitives are needed when accessing the object graph and similar barriers to incremental marking are often used to preserve invariants.

A high performance collector will often use parallel collector threads to perform both mark and sweep phases. This should not be confused with concurrent collection, where the mutator and collector can work on the object graph simultaneously.

2.3.2 Mark-compact garbage collection

A garbage collector based on the mark-sweep algorithm described above can suffer from *memory fragmentation*. Over time, contiguous blocks of memory become harder to allocate as frequent allocation and deallocation results in memory being sparsely filled with smaller contiguous blocks. This can make it harder (and sometimes impossible) to allocate a contiguous block of the requested size. In addition to reducing the effective available memory, this can cause two kinds of performance problems: first, allocating requires more work as the parts of the allocator responsible for mapping memory, known as *freelists*, must be scanned looking for a contiguous chunk of memory that fits the requested size; second, a program containing objects with poor locality in memory can be slower because of the negative effects this has on the CPU cache.[5]

[5]Fragmentation is not a unique to garbage collection. Languages which use manual memory management also suffer from fragmentation because, apart from through limited interfaces such as `realloc`, they cannot move allocated blocks in memory. This is because in general there is no mechanism for identifying and then updating references in the program. More recently, the *Mesh* allocator provides a solution to this by using the indirection of virtual and physical page addresses to compress physical pages in memory behind the user program's back [Powers et al., 2019].

As the name suggests, a mark-compact (or compacting) collector addresses fragmentation by *moving* live objects closer together after collection to make the available memory more contiguous. There are three types of compaction algorithms: arbitrary compaction, where object order is not preserved after compaction [Saunders, 1974]; linearising compaction, where objects are relocated next to objects they refer to [Jones et al., 2016, p.31]; and sliding compaction, where gaps between objects are compressed, but the order of the objects in memory is preserved [Suzuki and Terashima, 1995; Flood et al., 2001]. Each technique has different performance and space trade-offs, however, most modern collectors preserve object locality in some way, as empirical observations have found that this has an important impact on the cache friendliness of the program [Huang et al., 2004].

A crucial component of a mark-compact collector is that after moving objects, the references to those objects in the mutator must be updated to point to the new object location. In GCs which perform concurrent compaction, that is compaction which happens concurrently with the mutator, a *forwarding pointer* is usually placed in the header of the old object, so that the mutator can locate the new address for the moved object. A read barrier may also be needed to ensure the correct version of an object is being read [Tene et al., 2011].

A mark-compact collector can increase the effective memory available to a program. It also has performance benefits of improved object locality, and faster allocation because allocating in a contiguous heap can be done efficiently with a single bounds check (known as *contiguous* or *bump* allocation) [Blackburn et al., 2004]. Of course, these benefits are not free. As well as the increased complexity that compaction adds, the additional passes over the heap required by compaction tends to lead to worse throughput than mark-sweep collection. As a result, high performance GCs such as V8 tend to avoid compaction unless their heuristics have determined that fragmentation needs addressing [Jones et al., 2016, p.41].

2.3.3 Semi-space garbage collection

The final approach to tracing GC is *semi-space collection* (also referred to as stop-and-copy) [Cheney, 1970]. A semi-space collector works by dividing the heap into two equally sized partitions known as *tospace* and *fromspace*. New objects are bump allocated in fromspace until it becomes full. At this point, a collection takes place and live objects in fromspace are moved to the beginning of tospace (a process commonly referred to as *scavenging*). The remaining objects in fromspace are considered garbage and can be ignored as new allocations now begin at the next available location in tospace. The roles of tospace and fromspace have now been effectively flipped.

The semi-space algorithm has the advantage that allocation is very fast as it is always a simple bump allocation. However, this comes at a cost that the available heap size is effectively halved. The consequence of this is that a copying collector will perform more collection cycles than their other tracing counterparts.

2.4 Combining memory management strategies

Neither mark-sweep, mark-compact, or copying collection can be considered the best choice for all use-cases. Rather, they each have different trade-offs that must be carefully considered by the implementation. As a result, it is common for high performance tracing GC implementations to combine elements of each based on the performance characteristics they wish to tune for [Bacon et al., 2004].

A common example is that high performance GCs use a form of *generational* collection to take advantage of the *weak-generational hypothesis*: the empirical observation that most objects die young [Ungar, 1984]. The basic idea is that much recovered spaced can be achieved with minimal collector effort by focusing only on collecting younger objects. Generational collectors divide the heap into logically separate old and young generations. Typically, these generations are phsyically separated too, so that short-lived objects can be allocated separately from long-lived ones. A generational GC typically allocates objects in the *young generation* first, before moving them to the old generation after they survive a certain number of collection cycles.[6]

High performance collectors such as the Dart VM [Google, 2019a], Java's HotSpot JVM [SUN Microsystems, 2006], and the V8 JavaScript VM [Google, 2008] use a form of copying collection for their young generation: taking advantage of the faster allocation it provides, and the knowledge that though collection cycles will be more frequent, garbage will be regularly collected since their objects will be shorter lived. It is common for such collectors to then use a form of mark-sweep or mark-compact for the old generation.

Hybrid reference counting and tracing garbage collection language implementations are also common. Python's CPython implementation uses reference counting as its primary GC mechanism with a secondary tracing mark-sweep collector to free cycles [Galindo Salgado, 2022].

[6]Objects which are known to live longer can be allocated directly in the old generation, in a process known as *pretenuring*.

Chapter 3

Retro-fitting conservative garbage collection in V8

Some programming language Virtual Machines (VMs) use *handles*, that is wrappers around pointers to heap objects, to allow the root set of garbage collected objects to be precisely enumerated. Other VMs use *direct references* to heap objects, requiring the C/C++ call stack to be conservatively scanned for values that appear to be pointers to heap objects. The performance trade-offs of these two approaches are unclear: in particular, do direct references gain more performance by removing a level of indirection than they pay through conservative stack scanning (including the costs of checking whether arbitrary words point to heap objects)?

This chapter studies the performance difference between these two approaches on V8 – a large, industrial strength VM with a codebase containing roughly 1.75MLoc. Because of this complexity, and the considerable engineering effort involved, I migrate around half of V8's handles to use direct references. However, I show that, even though a partial migration imposes additional runtime costs and the conservative stack scanner is naive, direct references are at least as fast as handles on the Speedometer2.1 benchmark suite.

This chapter is structured as follows. Section 3.1 begins by explaining the difference between writing VMs using handles or references and the implications this has on the GC for locating references. Section 3.2 offers some background on V8's approach using handles before detailing the migration strategy in Section 3.4 for retro-fitting conservative collection. Section 3.5 shows a performance evaluation of the migration strategy. Finally, an overview of the related work is provided in Section 3.7.

3.1 Handles or direct references?

This section provides a brief overview of direct references and handles in the context of VMs. To help make this concrete, I do so for a hypothetical VM, written in C, that implements an object orientated language. In this VM, every language-level object is an instance of a `struct` called `Obj`, which, for the sake of simplicity, we will assume has a single field `data` of type `int`.[1] We assume that the VM contains a GC that runs in a thread concurrently with other VM mutator threads, and thus the GC can free the memory of any unreachable `Obj` at any point.

3.1.1 Direct references

The most 'natural' way to program this hypothetical VM is to use direct references. We assume that the GC exposes a single function `Obj *gc_new_obj()`, which returns a pointer to a block of memory with a freshly initialised `Obj`. When we want to create a new object, we call `gc_new_obj`, assign the result to a local variable, and then operate upon its `data` field:

```
1 Obj *n = gc_new_obj();
2 n.data = 1;
```

As this example suggests, 'direct references' is the term for what many would see as the 'normal' way of programming in C. Unlike most normal C programming, however, this VM has a concurrent GC: it can run immediately after line 1 has executed, but before line 2 has executed. If the GC were not to notice that `n` is pointing to a live `Obj`, it would free that `Obj`'s memory, causing the program to misbehave (e.g. the program might segfault or overwrite a portion of memory now used in another thread). In other words, at that point in the program, `n` is a *root*, that is an entry point into the graph of objects created by the VM. For our purposes, we can consider the root set to be all references to objects stored in C-level variables, bearing in mind that at run-time such variables may be stored as part of a function frame on the C call stack.[2]

The ideal, and universal, solution would be for C's language specification to allow *introspection* of a function's frame layout, telling it, for example, where each local variable lives within a frame. For each frame on the stack, the GC could then read the value of only those variables whose compile-time type is `Obj*`, considering the objects they reference as roots. Local variables of other types, which are not of interest to the GC, would be

[1]This precludes our hypothetical VM from tracing through objects, which is out of scope for this chapter.

[2]There are other hiding places for such references, ranging from thread-locals to registers to intermediate variables: for introductory purposes, these can be considered minor variations on the C call stack problem.

ignored. This would allow the GC to *precisely* enumerate the GC roots (i.e. to include all live objects as root while not including any non-live objects as roots). Unfortunately, due to the performance costs of persisting type information at runtime, no mainstream systems language defines an introspection capability. To address this problem, a garbage collector can be split into two categories depending on how it identifies references: *precise*; and *conservative* collection.

3.1.2 Conservatively identifying references on the stack

Many VMs using direct references turn to *conservative stack scanning*, where the C call stack is exhaustively examined for possible pointers to instances of `Obj`. This requires the GC to know at what address a thread's stack began, and what address the stack is currently at. Each aligned word on the stack is then checked to see whether it potentially points to an instance of `Obj`: if it does, that `Obj` is considered a root. Depending on the GC, checking whether an arbitrary word is a pointer to an `Obj` can be fairly expensive, particularly if the GC requires interior pointers to objects be translated to base pointers — in this thesis, I consider these costs inherent to conservative stack scanning.

Conservative stack scanning inherently over-approximates the root set, because random words on the stack may accidentally point to an `Obj`, keeping it alive longer than necessary. Fortunately, in practice relatively few objects tend to be kept alive unnecessarily long: the most extensive study we know of suggests the false detection rate in Java programs is under 0.01% of live objects [Shahriyar et al., 2014].

Objects identified as live via conservative scanning must also be pinned in memory, as the GC cannot know whether pointers to those objects on the stack are 'genuine' or 'accidental'. This prevents those objects from being moved by the collector, as a stack value which was misclassified as a pointer would have its value erroneously updated. Fully conservative GCs, that is GCs which conservatively scan all parts of memory, from the call stack to the heap (e.g. the Boehm-Demers-Weiser GC [Boehm and Weiser, 1988]) are thus unable to move any objects in memory. However, many GCs are semi-conservative, that is most references to objects are determined precisely, with only roots on the stack discovered conservatively.

Conservative scanning occupies an odd position in software. Technically speaking, the way it works violates the rules of most compilers which, unaware of the existence of GC, manipulate stack layouts for optimisation purposes. In rare cases, compiler optimisations have been known to hide pointers from a conservative GC [Google, 2020]. Furthermore, some programming techniques (e.g. XOR lists) thoroughly defeat it. However, because conservative scanning is widely used (e.g. the well-known Boehm-Demers-Weiser GC and WebKit [Pizlo, 2017]), it is well supported in practise.

3.1.3 Precisely identifying references on the stack

One method of precisely identifying references on the call stack is by dynamically maintaining a *shadow stacks* [Henderson, 2002]. A shadow stack is a global data structure which precisely identifies the location of references on each stack frame. The main disadvantage of using a shadow stack is the performance cost it has at runtime: any stack operation which involves references must update the shadow stack. Since such operations are often frequent, this cost adds up.

Another method is to use an approach known as *dynamic pointer provenance tracking* [Banerjee et al., 2020]. This is a combination of static analysis and runtime taint propagation to locate the allocation from which a pointer in the program came from. Currently, this approach has only been realised on single-threaded applications and support for multi-threading is an open research area.

The most efficient way to precisely identify stack roots is for the compiler to emit their location in a way that can be accessed as a constant at runtime. Some compilers offer this as a concept known as *stack maps*, where the compiler defines a way for a program to introspect the call stack at specific locations. Stack maps offer the same functionality as language-defined introspection, but do so in a way that is not portable between compilers. In LLVM for example, a stack map records the locations of pointers in a stack frame at a particular instruction address.

Stack maps are stored in the binary of an application to be used at runtime. This has a memory overhead which means that it is often impractical to include them at every possible instruction address in a program.[3] Instead, stack maps are generated for specific *safe points* in a program. That is, a point in the program where it is safe for the collector to pause the mutator in order to schedule a GC cycle.[4]

At a minimum, safe points are needed at any call to an allocator so that a GC cycle can be scheduled if an allocation failed. In addition, safe points are also needed at loop back edges if there are parallel mutator threads. This is because if one thread fails to allocate due to memory pressure, it may not be possible to schedule a GC cycle if another thread is spinning in a non-allocating tight loop. This could cause the program to crash because

[3]Though Stichnoth et al. [1999] show a compression technique for how this can be done in the JVM, with the stack map entries about 20% of the generated code size

[4]The term safe point is somewhat overloaded as it is also used in GC terminology to refer to specific points in the program where parallel mutator threads can block when a collection is requested.

it is unable to halt each mutator thread safely to perform GC. In order to GC more frequently, safe points (and thus stack maps) are needed at more points in the program. This represents a trade-off between GC frequency and the programs code size due to increase number of stack maps. In the case of the Modula-3 language, compressed stack maps increased the code size by around 13% [Diwan et al., 1992].

Unfortunately, stack maps are highly implementation specific and poorly supported by C/C++ compilers. GCC does not support stack maps, and while LLVM does have some stack map support – introduced to support Azul's Falcon JVM [Reames, 2017] – it is experimental, incomplete, and unsupported, with particular problems at higher optimisation levels [Project, 2014].

Stack maps are also notoriously difficult when used in languages with optimising compilers. Diwan et al. [1992] show that certain compiler optimisations, such as those which obtain an inner pointer and throw away the base pointer, must be disabled.

3.1.4 Handles

An alternative approach to direct references and conservative scanning has become known as handles. Modern VMs use handles in two distinct ways: in this section we will consider *comprehensive handles*, as found in e.g. [Kalibera and Jones, 2011] (in Section 3.2 we will consider the other kind of handles).

Comprehensive handles add a level of indirection to all object references, with the indirection being the 'handle'. Handles are stored at a fixed point in memory, with one handle per object. In the context of our hypothetical VM, this means that we never store references to `Obj` directly, instead storing references to a `Handle` struct. When the VM wants to access an `Obj` it must *tag* the corresponding `Handle`; this informs the GC that the `Handle` is a root. When the VM is finished with the `Obj` it must untag the corresponding `Handle`. We maintain a tag count for each `Handle`, as it may be tagged multiple times before being untagged: when the tag count goes to zero, the `Handle` is no longer a root.

Our hypothetical GC needs three altered/new functions: `Handle *gc_new_obj()` returns a pointer to a handle, where the handle points to a freshly initialised `Obj`; `Obj *tag(Handle *)` which increments a handle's tag count by one and returns a pointer to the underlying `Obj`; and `void untag(Handle *)` which decrements the handle's tag count by one. Because such a function updates important state used by the GC, the collector must not be invoked while it is executed.

An example of this API in use looks like:

```
Handle *h = gc_new_obj();
Obj *o = tag(h);
o.data = 1;
untag(h);
```

Handles have important advantages. First, tagging and untagging allows the GC to precisely determine the root set by iterating over all handles and considering as a root any handle with a tag count greater than 0. Second, handles are fully portable and require no explicit language or compiler support. Third, any `Handle` with a zero tag count can have its underlying `Obj` safely moved. In other words, handles make it trivial to write precise, moving GCs. Comprehensive handles also have the virtue that moving an `Obj` requires only updating its corresponding `Handle`.

There are however disadvantages. First, a handle API without compiler support is easy to misuse: forgetting to `tag` a handle or `untag`ing a handle twice leads to undefined behaviour. Finding such API misuse is notoriously hard, and VMs such as V8 have developed home-grown linters that scan for obvious API misuse, though none that we know of can catch all possible kinds of API misuse. Second, handles' double level of indirection implies at least some performance loss relative to the single indirection of direct references. Third, handles have additional run-time costs beyond those of double indirection: each handle consumes memory; since handles cannot move, they can cause memory fragmentation; and tagging / untagging a handle requires memory reads and writes.

3.2 V8 background

V8 is a widely used JavaScript and WebAssembly VM, embedded in Chromium-based browsers (e.g. Chrome) and other well known systems (e.g. NodeJS). In this section we introduce some general V8 background, before describing V8's use of handles.

V8 is largely written in C++. A single V8 process can run many unrelated programs by encapsulating them in *isolates*, each of which has a private heap. Each heap is separated into young and old generations. Young generation-only collection uses a semi-space strategy that is a variant of Halstaed's approach [Halstead, 1984] (though V8 uses dynamically-sized lists of segments to allow for work stealing). A young and old generation collection uses a mark-sweep-compact strategy where marking and sweeping are performed concurrently [Degenbaev et al., 2018], and compaction occurs during idle time [Degenbaev et al., 2016]. Collection can also be performed during idle time to reduce latency from the user's perspective.

3.2.1 Handles in V8

V8 uses handles to determine the root set of objects. Unlike the simple VM described in Section 3.1.4, V8 uses *partial handles*: that is, it uses direct references in those places where it can precisely track them (e.g. object slots), and handles in those places where it would otherwise lose track of roots (e.g. stack references). Although V8 has various kinds of handles (e.g. root handles, persistent handles, eternals, and traced reference handles [Google, 2019b]), most of these have very narrow, specific uses:[5] we focus in this chapter exclusively on 'normal' handles

Comprehensive handles as described in Section 3.1.4 persist throughout a VM's lifetime, with each heap object having a single handle pointing to it. In contrast, V8's partial handles are temporary, being regularly created and destroyed: any object may, both at any given point and over its lifetime, have multiple handles pointing to it. This has a significant impact on the way handles are used within V8.

In essence, when V8's C++ code wishes to operate on a JavaScript object, it must create a handle to it, destroying that handle when it is finished. This can be thought of as a kind of 'lock': the handle guarantees that the object is kept alive while C++ code works upon object, without having to worry that pointers to the object will be 'lost' on the C++ call stack.

As with other VMs using this style of partial handles, V8 faces two challenges: how to make partial handles easy to use for the programmer; and how to make their regular creation and destruction fast.

The C++ `Handle` class is used pervasively throughout V8: most functions that operate on heap objects either take or return `Handle` instances. The `Handle` class exposes a pointer-like API (e.g. the dereferencing '*' operator on a handle returns an `Object`) that makes it relatively transparent in use. A `Handle` instance contains a pointer to the underlying handle but no additional storage (i.e. `size_of(Handle<T>) == size_of(T*)`, giving programmers confidence about handle storage and moving costs.

A challenge with partial handles is knowing when they are no longer needed. To help with this, V8 uses *handle scopes*, which can be thought of as pools of handles: newly created handles are automatically placed in the 'current' handle scope; and when a handle scope

[5]There are also a few parts of V8 where handles are forbidden and GC is prohibited.

```
1  void main() {
2    HandleScope scope(GetIsolate());
3    Handle<String> str = foo();
4  }
5
6  Handle<String> foo() {
7    HandleScope scope(GetIsolate());
8    Handle<String> a = NewString("a");
9    Handle<String> b = NewString("b");
10   return scope.CloseAndEscape(a);
11 }
```

Listing 3.1: A simplified example of V8's C++ code, showing the use of handles. `main` creates a handle scope (line 2), and then calls `foo` which creates a further handle scope (line 7). Two new heap objects are created, each of which produces a handle (lines 8 and 9), ensuring that both objects are considered as roots. `foo` cannot directly return a handle to its caller, as that handle will be automatically destroyed when the handle scope is destroyed by RAII at the end of the function call. The `CloseAndEscape` method adds a new handle to `main`'s handle scope pointing to the `"a"` string, and passes a reference to that handle to the caller, allowing the underlying object to safely 'escape' the handle scope it was created in.

is no longer needed all of the handles in it are destroyed. When V8 code creates a new `HandleScope` instance, that handle scope is pushed onto a stack, whose topmost element is the current handle scope. C++'s RAII mechanism is used to automatically destroy `HandleScope` instances, which also destroys all the handles it contains.

Handle scopes ensure that C++ code dealing with objects remains relatively terse, and reduces several opportunities for programmer mistakes. However, C++ code must carefully ensure that it does not leak a pointer to an object beyond the lifetime of the handle scope that references that object — doing so causes undefined behaviour. V8 has various lints (for example *gcmole*) which search the codebase for possible API misuse and warn when such instances are found. However, lints cannot identify all possible misuses: some misuses are later detected in debug builds, but some misuses end up in release code, where they can become a significant security concern.

Listing 3.1 shows a simplified snippet of V8's C++ code, demonstrating how `Handle`s and `HandleScope`s interact. In this example, two handle scopes are created, with the second handle scope wanting to return a handle that outlives its handle scope. Handle scopes expose a `CloseAndEscape` method which allow a handle to be safely moved to the parent handle scope.

Handle scopes are an important performance optimisation, since destroying a handle scope causes its backing storage containing multiple handles to be destroyed in one go: neither individual deallocation or any form of handle compaction is necessary.

A further optimisation is based on the observation that within a given handle scope it is common for the same object to be referenced multiple times. Although it is always correct to create multiple handles to the same object, it is inefficient, since each handle requires memory, requiring the storage area pointed to by a handle scope to be enlarged. When a handle for an object is requested, V8 thus performs a linear scan through the current handle scope to see if an existing handle to that object is present, only creating a new handle if none exists.

Though handles and handle scopes are needed in V8 to precisely enumerate the roots, JavaScript provides introspection, so object slots in the V8 heap can be traced precisely because their layout information is available to the collector.

Although in most of this chapter we consider V8 in isolation, we are particularly interested in how our changes affect V8 when running browser code. That means that we also need to consider the interactions between V8 and *Blink*, Chrome's rendering engine. V8 exposes a simplified `Handle`-like class called `Local`, which also uses two levels of indirection, to 'external' applications such as Blink. Moving objects between Blink and V8 requires converting to/from `Handles`/`Locals`, though the default configuration optimises these conversions to a compile-time cast (i.e. there is no run-time copying of data).

3.3 Why switch to direct references in V8?

3.3.1 Ergonomics

A full migration from V8's handle based architecture to direct references would make the V8 codebase more ergonomic. That is, it would be easier for VM developers to read and write code and reduce the surface of possible programming errors which can be introduced. This is for two main reasons. First, the creation of `HandleScope` objects take place in over 17,000 locations inside the V8 codebase. A full migration to direct references would allow these to be removed. This would reduce the overall size of the codebase and simplify the parts which deal with on-stack references to heap objects.

Second, direct references allow for more idiomatic usage of pointers in C++. V8 developers would no longer need to adhere to the handle scoping rules (Section 3.2), and could refer to garbage-collected heap objects using ordinary C++ references. The lifetime of object references would be tied to lexical scope, something which C++ programmers will already be familiar with.

3.3.2 Performance

The major performance problem with handles is that each heap object dereference requires an additional level of pointer indirection. This constitutes at least an extra load instruction; has negative cache behaviour; and can prevent certain compiler optimisations. In addition, the memory cost for storing handles at runtime, alongside the frequent creation and destructions of handle scopes can be removed.

Conservative stack scanning is also a promising approach for V8 since over 80% of V8's GC cycles occur during the JavaScript event loop,[6] when there can be, by design, no pointers on the call stack. This means that it would only need to take place on allocation failure, and the likelihood of needing to pin objects in memory would be greatly reduced. However, the main performance trade-offs when using this conservative approach is that objects marked in the GC via a conservatively identified pointer cannot be moved, which could negatively impact the effectiveness V8's compaction.

3.4 Gradually migrating from handles to direct references

Unfortunately, while migration from handles to direct references sounds conceptually easy, on a codebase of V8's size and complexity it is practically laborious. In total, I migrated around 743 files, with 73,544 lines changed equating to roughly half of the handle usage in V8. The migration evaluated in this chapter is the result of my second migration attempt. In this section I explain my general migration strategy, the challenges I faced, and what parts of V8 have migrated.

3.4.1 Migration strategy

Simplifying slightly, my first migration attempt tried to move the codebase wholesale from handles to direct references — after considerable effort this failed. This was due to a combination of factors but two are of particular note. First, many parts of V8 quite reasonably make use of implicit properties of handles that are not captured by the type system or API. This primarily includes type casts to other types which assume a level of indirection. These must be manually inspected and altered to work correctly with direct references. Second, when I inevitably hit bugs due to my changes, I found it difficult to debug and work out which of my many changes caused a particular problem.

[6]Based on data from Chrome's *User Metric Analysis* for Chrome's stable release branch (M110) on Windows

This chapter is the result of my second migration attempt. I used the experience from the first failed attempt to devise a scheme that allows gradual migration of small portions of V8 from handles to direct references. The migration is 'hidden' behind a compile-time flag that is false by default, allowing changes to be incrementally upstreamed without affecting mainstream users.

The most obvious part of the gradual migration approach is to realise that handles and direct references must be able to co-exist for as long as the migration takes. Thus, this lead to the introduction of a new C++ class which I will refer to in this chapter as `DirectRef`.[7] which stores a direct reference to objects on the heap while exposing the same API as `Handle`. Once a full migration is complete, the `DirectRef` wrapper around the pointer has no impact and can be removed entirely.

In many cases, migrating from handles to direct references simply requires replacing `Handle` with `DirectRef`. However, many parts of V8 make assumptions about handles, normally for performance reasons, that are not encoded in the `Handle` API.

For example, string builders and array backing stores are used extensively by JavaScript code. When constructing and reallocating their value in memory, a handle's intermediate pointer is modified in-place so that this propagates to other handles using this value. This is done because it makes initialising such objects easier for the V8 developer, as they rely on the implicit knowledge that all heap references use indirection. In contrast, migrating such places to use direct references means ensuring they are used exclusively so as not to invalidate other references.[8]

Similarly, the parts of V8 that enable a transition from C++ code to JIT-compiled machine code and back are written in a macro-assembler which is translated into each platform's assembly format. The macro-assembler does not refer to the `Handle` class directly but generates code which implicitly follows handle conventions. This had to be carefully adjusted to follow the state of migration in C++ code.

Another difficulty was ensuring that direct references can interact correctly handles during the migration. The chief problem is that the `Handle` class constructors need a reference to an isolate because they need to know which isolate the handle scopes belong to, but the `DirectRef` class constructors do not. When moving from a migrated portion of code using `DirectRef` to unmigrated code using `Handle`, we thus do not know which isolate to create the new handle in. I could have required `DirectRef` to store a reference to an isolate, but this would be awkward and muddy performance comparisons. Instead, I assume that a

[7]The actual name of this class is the confusing, and contradictory, `DirectHandle`.

[8]When I started my first migration attempt, I had to look for challenging parts of the code like this manually. Motivated by this work, V8 has since developed a `Handle<T>::PatchValue` API which captures all the challenging places in a way that makes finding them fairly easy.

`Handle` created from a `DirectRef` lives in the current isolate. When this assumption is incorrect (for example, when the JIT compiler hands compilation off to another thread), this almost always results in an immediate crash. Fixing such a crash can be tedious, as the necessary isolate needs to be threaded through one or more layers.

3.4.2 Performance challenges during migration

Since migrated and unmigrated code frequently interact, moving from code which uses direct references to code using handles or back again requires additional computation.

Migrated code that interfaces with unmigrated code must create handles. This sometimes leads to the surprising outcome that migrating a small portion of code created a significant slowdown in runtime performance; in general, migrating a small additional portion of code restored performance. In the early stages of migration this problem was frequently encountered. However, since migration sometimes imposes handle creation costs on the callers of a function, it was often unclear which callers of a migrated function were causing performance problems. I developed a simple 'profiler' which pinpoints how often a line of source code creates handles, allowing us to quickly identify such locations.

When moving from unmigrated code to migrated code we have to take account of V8's uses of *tagged pointers*, which avoids storing small integers as full objects on the heap. The tagging is done on the 'pointer' from a handle to a heap object: if the top bit is set to 1, this is really a pointer; if the top bit is set to 0, it is a small integer. I use the same pointer tagging scheme in `DirectRef`. However, when converting from handles to direct references, *null handles* (which are often used as place holders) must be treated with care. To avoid allocating backing memory, the 'nullness' is represented by a null pointer to the handle itself. In other words, a null `Handle` instance itself contains a null pointer, but if a small integer is stored, the `Handle` points to a handle whose tagged pointer has a top bit set to 0. We have to flatten this representation for `DirectRef`. Thus, converting from `Handle`s to `DirectRef`s requires an extra compare-and-branch to deal with null handles correctly: though the cost of this check is small, it imposes a performance cost in the general case.

Since the migration of V8 is partial, these performance costs show up in the later experiments. A fully migrated V8, however, would show no such costs.

3.4.3 What was migrated

When deciding what parts of V8 to migrate, I wanted to choose those parts whose migration would give us greatest overall confidence in terms of correctness and performance. That meant that we chose some of the hardest parts to migrate, and those that are considered as the most likely performance bottlenecks. I thus migrated most of the following parts of V8:

1. The V8 embedder API layer (where V8 interacts with other applications, such as the Blink rendering engine in Chrome).
2. The interface between C++ code and V8's JIT compiled JavaScript runtime.
3. JavaScript object definitions and implementations.
4. Intrinsics, builtins, and runtime functions.

3.4.4 Conservative stack scanning

Direct references require conservative stack scanning, so I needed to add a conservative stack scanner to V8. Rather than writing one fully from scratch, I was able to borrow some parts of the conservative stack scanner which were already engineered in Chrome using Blink's GC *Oilpan*, specifically the platform specific assembly stubs for spilling registers during a stop-the-world pause.

The conservative stack scanner tests each word on the call stack to see whether values constructed from such words look like pointers to heap object. For each value, we first check if the value points into a page in the heap; if it does not, we quickly move on. If the value does point into a page in the heap, we then check if it points to an object or empty space. This is done by scanning the freelist in the allocator. If the value does point to an object, we then query the isolate's heap layout to find out the object's base pointer. Our scanner is not yet production ready: compared to Oilpan's scanner, it is relatively unoptimised.

3.5 Evaluation

In order to understand the performance effects of migrating from handles to direct references, I conducted an experiment on two variants of V8 running the Speedometer2.1 benchmark suite. This section starts by outlining the methodology (Section 3.5.1) before examining the results (Section 3.5.2).

3.5.1 Methodology

The overall question we would like the study to answer is: are handles or direct references faster? While this thesis can provide an accurate assessment of handle's performance, it can only provide a lower bound of direct references' performance: around half of V8's (large) codebase remains unmigrated, and the conservative stack scanner I have implemented is naive, and thus slower than a production version. I am also forced to non-generational GC because the young generation scavenger is a moving semi-space collector, and work to migrate this to a non-moving generational GC has not yet been carried out. Fortunately, and despite these restrictions, this study is able to provide valuable insights. To try and answer this question, this study is divided into two sub-experiments.

The experiment performed is a straightforward comparison between handles and direct references, with the GC running as normal (subject to the restrictions noted earlier). This implicitly includes the full costs of both handles (creating handles; destroying handles; an extra level of indirection) and direct references (conservative stack scanning). A limitation of this experiment is that I must disable heap compaction when a GC is scheduled and there are frames on the C++ stack, as the compacter is not yet aware of conservative stack scanning.

I evaluate each browser variant on the well known SPEEDOMETER2.1 browser benchmark suite [WebKit, 2022] using Chrome's Crossbench benchmark runner. SPEEDOMETER2.1 consists of 16 widely used JavaScript web frameworks all modeling the same application, TodoMVC: a todo program using the model-view-controller pattern. This benchmark suite executes various performance critical pieces of a JavaScript VM and the web rendering engine overall, and is therefore considered one of the most relevant workloads for web browser engine optimizations. Crossbench was used to run SPEEDOMETER2.1 for 30 *process executions*, where each browser instance is closed down and a fresh one is started. SPEEDOMETER2.1 records wall-clock times using JavaScript's `performance.now()`, reporting the results back to Crossbench. I report 99% confidence intervals for each benchmark.

I use Chrome version 112.0.55572 and V8 version 11.2.28. The benchmarks were run on a Xeon E3-1240v6 3.7GHz CPU cores, with 32GB of RAM, and running Debian 8. I disabled turbo boost and hyper-threading in the BIOS and set Linux's CPU governor to *performance* mode. I observed the `dmesg` after benchmarking and did not observe any oddities such as the machine overheating.

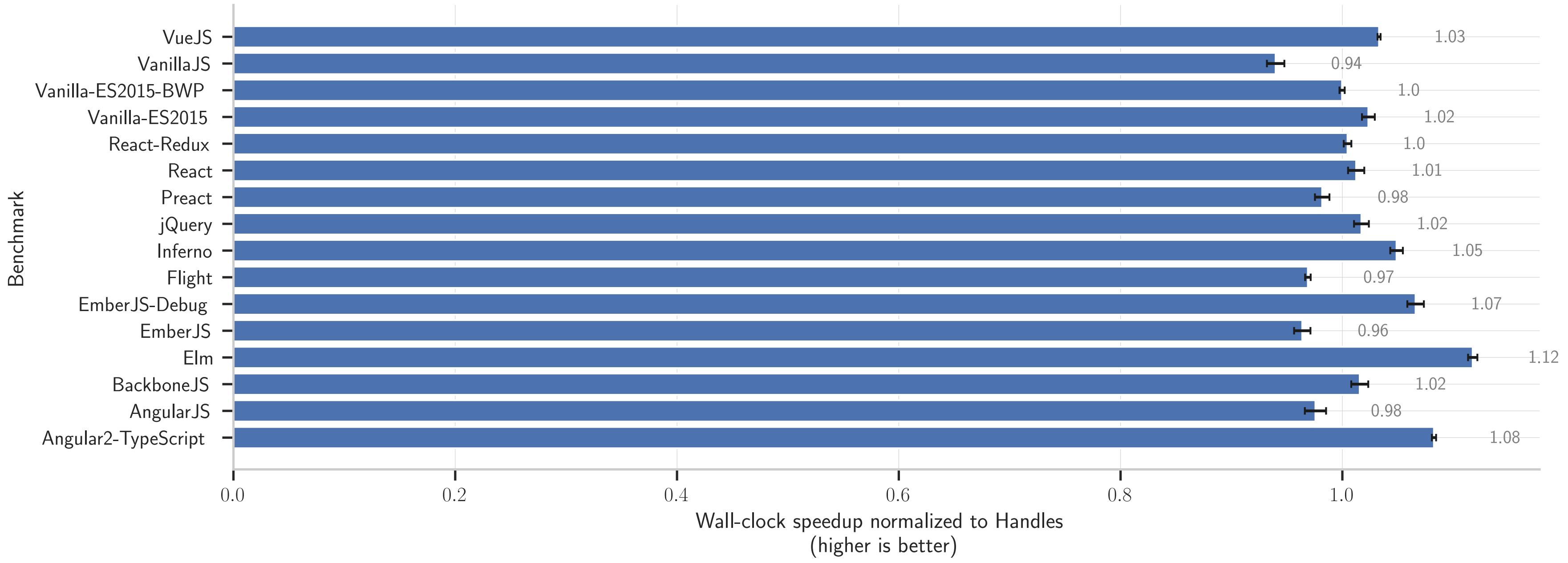


Figure 3.1: Results from both experiments on SPEEDOMETER2.1 for 30 process executions, which shows the speedup of the configuration using direct references with conservative stack scanning normalised to using handles. The error bars show 99% confidence intervals.

3.5.2 Results

Fig. 3.1 shows the results for this study. For both configurations of V8, the garbage collector scheduled a full GC cycle 24 times for each iteration of the benchmark suite. The results show that – even though V8 has only been partly migrated, that there remain costs for moving between migrated and unmigrated code, and the conservative stack scanner is naive – direct references with conservative stack scanning performs between 1-12% faster on 9 benchmarks. However, in 5 benchmarks this configuration regressed by 2-6%, while the *Vanilla-ES2015-BWP* and *React-Redux* benchmarks observed no statistically significant difference.

It is possible (indeed, likely) that completing the migration from handles to direct references, and improving the performance of the conservative stack scanner, will meaningfully improve upon this lower bound.

In order to understand better which parts of the code are responsible for the variations in performance for each benchmark, I used Linux's `perf` tool to profile the execution of them as single runs. Two benchmarks serve as exemplars of the effects of the migration.

The jQuery benchmark (where direct references are 2% faster than handles in Figure 3.1), puts particular stress on interactions between the Blink's DOM heap and V8's JavaScript heap — a part of V8 which was almost entirely migrated (see Section 3.4.3). The perf profile showed that direct references were approximately 15% faster in these parts of the embedder API.

The EmberJS benchmark (where direct references are 4% slower than handles in Figure 3.1) puts stress on: entry to parts of the JIT compiler; a section of the inline cache; and JavaScript debug objects. Other than small parts of the inline cache, none of these sections of V8 have been converted to use direct references. However, each of them interacts frequently with parts of V8 which have been migrated: in other words, frequent conversion costs to/from handles due to the partial migration exist which I expect to improve once more of V8 is migrated. Our profiling data shows a 30% slow down in parts of the execution engine (responsible for invoking JavaScript functions) which interact with non-migrated code.

3.6 Threats to validity

We have only migrated about half of V8 to direct references. It is possible that there are challenges in the remainder of the code base that would change our view of the merits of direct references. We have tried to mitigate this by migrating what we consider to be the most challenging parts of V8 first. While V8 is only partly migrated, there are also performance overheads involved in moving between migrated and unmigrated code (e.g. the null checks explained in Section 3.4.2): our results can only establish a lower bound for direct references performance.

JavaScript VMs such as V8 are somewhat unusual amongst modern VMs in two respects: JavaScript is single-threaded; and it is based around an event loop which naturally gives frequent opportunities to perform garbage collection without pointers existing on the C stack. This eases the cost of conservative stack scanning, as over 80% of V8's GC cycles occur during the JavaScript event loop, when there can be, by design, no pointers on the call stack.

These two factors may have a notable impact on our understanding of the performance merits of direct references and handles. We would have to migrate at least one non-JavaScript VM (e.g. HotSpot) to understand whether these are significant factors or not.

There are two differences between the configurations of Chrome/V8 run in the experiments and those run by normal users. First, I have to disable V8's young generation, because it does not support pinning immovable objects which were located by conservative stack scanning. Second, I do not have the ability to generate or use the profile-guided-optimisation data that production builds have access to. At best, these two factors mean that our Chrome/V8 builds will be slower than their production cousins; at worst, it could be that this changes the relative performance of handles and direct references.

Objects identified via conservative stack scanning cannot be moved. However, since per-object pinning has not yet been implemented, I am forced to prevent heap compaction whenever GC occurs and there might be pointers on the stack.

SPEEDOMETER2.1 is a small, synthetic benchmark suite: it can not tell us about the performance of all possible programs. However, its ubiquity does mean that its strengths and weaknesses as a benchmark suite are widely acknowledged, and it allows a good degree of comparison across JavaScript VMs and browsers.

3.7 Related work

The relative merits of direct pointers and handles have a long history in GC and VMs, though quantitative comparisons are few in number. Generalising somewhat, one technique or the other has tended to hold sway either at different points in time, or within different communities.

Early Smalltalk VMs used handles, generally calling them ‘object tables’. Though the reasons for this are not made it explicit, the VMs that did so used a compacting GC, so it is reasonable to assume this was the motivation [Krasner, 1983, p. 18]. Handles are often used in VMs that descend from the Smalltalk tradition (e.g. HotSpot [Oracle, 2007], Dart [Egorov, year]).

To the best of my knowledge, the most comprehensive study of handles is [Kalibera and Jones, 2011] which evaluates different styles of handles in the context of the Ovm’s real-time GC, which uses a concurrent compacting GC. Although Kalibera and Jones [2011] use the term ‘direct references’ to refer to Ovm’s initial state, and uses conservative stack scanning, these are not the same as ‘direct references’ in this chapter. Due to Ovm’s concurrent compacting GC, mutator threads may still reference an object’s ‘previous’ location, necessitating a level of indirection to ‘direct references’ to get the current location. They show that, in their context, ‘thin’ or ‘fat’ handles were more effective than (their use of the term) ‘direct references’ because it allowed objects to be moved without requiring forwarding pointers. This is a notably different context to V8, hence my rather different conclusions.

Firefox’s JavaScript engine SpiderMonkey uses handles in a similar fashion to V8 [Mozilla, 1995], including similar lints to catch handle misuse.

The use of direct pointers implicitly requires conservative scanning of at least the stack. However, most (perhaps all) programming languages and most (perhaps all) compilers have rules which suggest that conservative stack scanning is undefined behaviour. Most obviously, there is no way of writing a conservative stack scanner in C/C++ which is not undefined behaviour. In practise, fortunately, conservative stack scanning ‘works’, as perhaps best evidenced by the long history of the Boehm-Demers-Weiser (BDW) GC [Boehm and Weiser, 1988] which uses conservative scanning for the stack and other possible location of GC roots (though note that the first conservative scanning GC appears to have been [Caplinger, 1998]). BDWGC has been ported to most platforms in its long history, and used by a wide variety of software (including relatively modern software such as Inkscape): it is plausible that its very existence has been the reason why conservative stack scanning continues to be supported in practise across languages and compilers.

Oilpan [Ager et al., 2013], the GC for the Blink rendering engine uses conservative stack scanning to find roots on the C stack but is able to precisely trace object slots. This is a similar design to that I am proposing for V8 in this Chapter, though Oilpan cannot move objects in the heap.

Safari’s underlying engine WebKit (of which the JavaScript VM, JavaScriptCore, is a part) use direct pointers and conservative stack scanning [Pizlo, 2017], which is very similar to what I propose for V8 in this chapter. WebKit has had at least two different GCs over time (the current GC, [Pizlo, 2017], mostly runs concurrently to JavaScript execution, similarly to V8), but this has not changed the details relevant to this chapter. Unlike V8, WebKit's GC never moves objects, even across generations (instead using 'sticky mark bits' that implicitly mark a given object as belonging to a certain generation). Where V8 only uses precise stack scanning for C/C++ code, using stack maps for JIT compiled code, WebKit uses conservative stack scanning for both C/C++ code and JIT compiled code, slightly increasing the chances of objects being incorrectly identified as live.

3.8 Future work

Of course, in this work I have only converted around half of V8's 1.75MLoc to use direct references. A complete migration is necessary to understand the full picture. In addition, future work outside the scope of this thesis could include: the implementation of non-moving generational GC, and a more production ready stack scanner. This would allow a closer comparison of release versions of Chrome today.

Chapter 4

Existing memory management in Rust

This chapter gives a brief overview of the Rust programming language [Mozilla, 2010], with a particular focus on Rust's approach to memory management, as this is necessary to understand ALLOY. It is not assumed that the reader is familiar with Rust.

4.1 Rust memory management basics

Rust is a statically-typed, general-purpose programming language [Klabnik and Nichols, 2018]. Rust has high-level functional programming constructs as well as low-level features such as support for inline assembly and a foreign function interface (FFI) allowing it to call (and be called from) C. This makes Rust popular for both low-level systems programming and complex programs where performance is important, such as parts of Mozilla's Firefox web browser [Mozilla, 2012].

Arguably Rust's most interesting feature is that it can guarantee memory safety without using tracing garbage collection or reference counting.[1] Instead, Rust's type system ensures that *references* (as distinct from pointers) are guaranteed to point to valid memory, which can be checked at compile-time. These rules are the focus of this section, as they are key to understanding the memory management story in Rust, and why one may wish to extend it with a garbage collector such as ALLOY.

[1]Though, as we will see in Section 4.2, Rust does provide optional reference counting

4.1.1 Ownership

The foundation for memory safety in Rust is its concept of ownership based on affine types [Pierce, 2004]. Each value in Rust has a single owner, and rebinding a value to a new variable will use *move semantics*, where the ownership of the value is transferred, and the old variable is invalidated. Consider the following example:

```
1  fn main() {
2      let mut v = Vec::new(); // v: Vec<u64>
3      v.push(1);
4      foo(v); // ownership of v is transferred to foo
5      // v.push(3); // compile-error: v has been moved into foo
6  }
7
8  fn foo(mut v: Vec<u64>) {
9      v.push(2);
10 }
```

A new empty vector `v` is created inside Rust's `main` entry-point function (line 2). A vector is a resizable heap allocated array which contains elements of a single type (`u64`, unsigned 64-bit integers in this case). Variables are *immutable* by default, so the `mut` syntax indicates that `v` should be *mutable*. `v` is then passed as a parameter to `foo`, the ownership of `v` is transferred, which means its contents can no longer be accessed from inside `main`. If we were to uncomment line 5, we would be attempting to use a value after a move (line 5) which would cause a compiler error.

```
error[E0382]: borrow of moved value: ‘v‘
 --> src/main.rs:5:5
  |
2 |     let mut v = Vec::new(); // v: Vec<u64>
  |         ----- move occurs because ‘v‘ has type ‘Vec<u64>‘,
  |               which does not implement the ‘Copy‘ trait
3 |     v.push(1);
4 |     foo(v);
  |         - value moved here
5 |     v.push(3);
  |     ^^^^^^^^^ value borrowed here after move
```

The final part of ownership is that at the end of `foo`, when the vector goes out of scope, Rust will automatically call its *destructor*. There are clear parallels here with RAII in C++. In fact, Rust's ownership rules can be thought of as a more strict version of this: the Rust compiler ensures that values are not used after they are moved, and are automatically destructed when they go out of scope.

Destructors

Astute readers might wonder how Rust knew to deallocate the vector. In short, the `Vec` type has a destructor method associated with it which Rust calls when it goes out of scope.

To understand how this works, we first need to introduce a new concept in Rust known as *traits*. A trait is a way of specifying behaviour in an abstract way which can be defined for a type. Traits are very similar to interfaces in other languages, where a single trait can define behaviour which can be shared between many types.[2]

In order to have destructor behaviour in Rust, a type must implement the `Drop` trait. Consider a struct with a single field which implements the `Drop` trait:

```rust
// Defined in the Rust core library
trait Drop {
    fn drop(&mut self);
}

struct S {
    a: u64,
}

impl Drop for S {
    fn drop(&mut self) {
        println!("Dropping␣S");
    }
}
```

The `Drop` trait has a single method `drop`, which gets called automatically by Rust when a value goes out of scope. Here, `S` implements the `Drop` trait, so when `S` goes out of scope its drop method is called, causing “Dropping S” to be printed to stdout. In Rust parlance, when a value is destructed, it is referred to as having been *dropped*.

[2]There are some differences between traits and interfaces, but those details are not particularly relevant for this thesis.

Types such as `Vec` which allocate memory on the heap (for the backing store) implement `Drop` to deallocate that memory.[3] Destructors play a big part in the rest of this thesis, so they are explained in more detail in Section 6.2.

4.1.2 Borrowing

Rust's exclusive ownership rules allow it to avoid garbage collection.[4] However, they are fairly restrictive. It would be cumbersome and inefficient to write programs which can only use values by transferring ownership back and forth each time. Rust addresses this with the concept of *borrowing*, where values can be accessed using borrowed references instead of transferring ownership.

There are two kinds of references in Rust: '`&`' immutable (or shared) references; and '`&mut`' mutable (or unique) references. Rust programs which use borrowing must adhere to two rules.

1. A value may have an unlimited number of simultaneous immutable references or a single mutable reference.
2. A reference must never outlive its *referent* (the value it points to).

These rules are checked at compile-time by Rust's *borrow checker*, meaning that a Rust program which successfully compiles is guaranteed to be memory safe.

Listing 4.1 shows how the previous example would look if `foo` borrowed a reference to the vector instead.

It is easy to see how these borrow rules guarantee data-race freedom: a mutating thread can never access aliased data. What is less obvious is that these rules also prevent a class of memory safety bugs which occur when single-threaded shared mutability is permitted. One such example is that Rust prevents *iterator invalidation*, where an iterator becomes invalid because a data structure is mutated while it is being iterated over, as shown in Figure 4.2.

[3]This is a simplification. In reality, the backing store (`RawVec`) has its own `Drop` implementation which does the deallocation.

[4]However, as we will see, they make it impossible to write certain data structures in Rust (such as doubly-linked-lists) without using some form of garbage collection.

```
 1 fn main() {
 2     let mut v = Vec::new(); // v: Vec<u64>
 3     v.push(1);
 4     let vref = &mut v;
 5
 6     // This is not allowed, because it's not possible to use v
 7     // while a reference to it exists:
 8     // v.push(2);
 9
10     foo(vref);
11
12     // This is allowed, because v was not moved and vref expired
13     // at the end of foo:
14     v.push(3);
15 }
16
17 fn foo(v: &mut Vec<u64>) {
18     v.push(2);
19 }
```

Listing 4.1: An example of borrowing in Rust. A reference to the vector, `vref`, is created and passed to `foo`. To ensure that nothing aliases a mutable reference, it is illegal to use the vector via the owned variable `v` while `vref` is still alive (line 8). When `vref` expires at the end of `foo`, it is safe to mutate the vector through `v` again (line 14).

```
fn main() {
    let mut v = vec![1,2,3,4];

    for i in v.iter_mut() {
        v.push(*i);
    }
}
```

```
error[E0499]: cannot borrow ‘v‘ as mutable
more than once at a time
 --> src/main.rs:5:5
  |
4 | for i in v.iter_mut() {
  |          ------------
  |          |
  |          first mutable borrow occurs
  |          here
  |          first borrow later used here
5 |     v.push(*i);
  |     ^^^^^^^^^^ second mutable borrow
  |                occurs here
```

Listing 4.2: An example invalid Rust program and its compilation error. This fails to compile because two mutable borrows overlap: the variable `v` which owns the vector; and the borrow from the iterator to the vector element `i`. If this program were to compile, it would be potentially memory unsafe. The call to `push` resizes the vector, which could cause it to be reallocated in the heap. The iterator reference `i` would then be invalid as it points to the old location in memory.

4.1.3 Lifetimes

In Section 4.1.2, we see that Rust references are only valid provided they do not outlive their referent. The borrow checker ensures that this is adhered to using a construct known as *lifetimes*. Each reference in Rust has a lifetime, which begins when it is created and

```
 1 let mut v = vec![1, 2];
 2
 3 // borrow 'a starts
 4 let vref = &'a mut v
 5
 6 // borrow 'b starts
 7 println!("v:{}", &'b v);
 8 // borrow 'b ends
 9
10 vref.push(3);
11 // borrow 'a ends
```

(a) Overlapping mutable references

```
 1 let mut v = vec![1, 2];
 2
 3 // borrow 'a starts
 4 let vref = &'a mut v
 5
 6 vref.push(3);
 7 // borrow 'a ends
 8
 9 // borrow 'b starts
10 println!("v:{}", &'b v);
11 // borrow 'b ends
```

(b) Refactored to adhere to borrow rules

Listing 4.3: An example of how Rust uses lifetimes to ensure programs adhere to the borrow rules. The lifetime annotations `'a` and `'b` have been deliberately added. Program **a** does not compile because a reference to `v` is created (line 7) while a mutable reference `vref` already exists. `vref` has a lifetime `'a`, which expires on line 11. While this reference is alive, no other reference may exist. Program **b** fixes this by rearranging when the `println!` happens so that the two references do not have overlapping lifetimes.

ends when it is destroyed. In the examples shown up until now, the borrow checker has been able to infer the lifetimes of references.

These borrow rules are a key component of Rust's memory safety because they prevent common bugs based on aliasing and invalid references. Listing 4.3 shows how lifetimes are used to prevent overlapping borrows.

It is not always possible for the borrow checker to infer the lifetimes of references. In such cases, the programmer must provide annotations where lifetimes could be ambiguous.

4.1.4 Cloning and copying data

Rust provides a way to explicitly duplicate an object via the `Clone` trait. The `Clone` trait has a single `clone` method which returns a duplicated version of the object.[5] Listing 4.4 shows how the `Clone` trait can be used to provide clone semantics for an object in Rust.

In addition to cloning, types which implement the `Copy` trait can be duplicated simply by copying their bits. The `Copy` trait has no methods: instead it is used as a special kind of marker trait which gives a type *copy semantics* instead of *move semantics*:

```
let mut x = 1; // ‘u64‘ implements the ‘Copy‘ trait.
foo(x);
let y = x + 1; // This is valid because a copy of ‘x‘ was passed to foo.
```

[5]It is up to individual types to define the exact semantics of the clone in their implementation of the `clone` method.

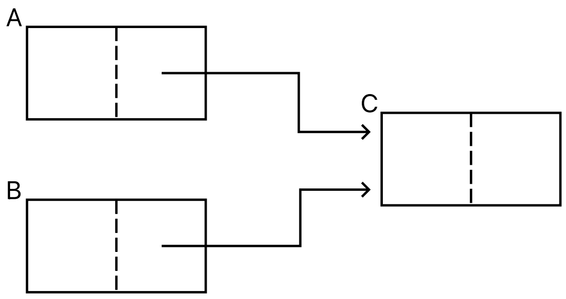
```
1 let mut v1 = Vec::new();
2 v1.push(1);
3 v1.push(2);
4 v1.push(3);
5
6 let v2 = v1.clone(); // clone returns a duplicated vector
7 foo(v2); // Move v2 into foo
8
9 v1.push(4); // Not a use-after-move, because foo took a different object.
```

Listing 4.4: An example of cloning objects in Rust. Here, a deep copy of `v1` is returned by calling `clone` on it (line 6). This duplicated vector is then moved into `foo`, which means we can still use the original vector on line 9.

Here it is fine to use `x` after the call to `foo` because a copy was passed to foo. Copiable types help to make writing Rust more ergonomic, as they do not have to adhere to the ownership rules.

4.2 Shared ownership using reference counting

Rust's ownership semantics mean that every object has a single owner: without some kind of additional mechanism, it is very difficult to represent data structures such as graphs where each node conceptually has multiple owners. For example, consider an object `c` which should be jointly owned by `a` and `b`:



It is not possible to represent this using Rust's ownership semantics,[6] so to address this, Rust allows certain objects to be reference counted. Listing 4.5 shows how such an example can be written in Rust using reference counting and Figure 4.1 shows how this program would be represented in memory.

[6]This is not strictly true: there are workarounds, such as using a vector's indices to represent edges in a graph, though this is fraught with challenges, and can be seen as a form of "punning" around Rust's ownership system. Such techniques do have their uses, however, and are explained in more detail in Section 4.3.2

```
1  use std::rc::Rc;
2
3  struct Node {
4      name: &'static str,
5      child: Option<Rc<Node>>,
6  }
7
8  fn main() {
9      let c = Rc::new(Node { name: "c", child: None });
10     let a = Rc::new(Node { name: "a", child: Some(Rc::clone(&c)) });
11     let b = Rc::new(Node { name: "b", child: Some(Rc::clone(&c)) });
12
13     std::mem::drop(c); // decref "c" count to 2
14
15 } // b dropped: decref'd to 0, dropping its underlying ‘Node‘
16   //            recursively decrefs 'c' to 1
17
18   // a dropped: decref'd to 0, dropping its underlying 'Node'
19   //            recursively decrefs 'c' to 0
```

Listing 4.5: This example uses Rust's `Rc` *smart pointer*: a reference to an object which has additional reference counted semantics. The calls to `Rc::new` (lines 9-11) tell Rust to allocate `Node` objects on the heap and use reference counting to manage their memory. Additional reference counted references to "c" are then obtained by cloning the original `Rc` on lines 10-11 (incrementing the underlying object's count). When an `Rc` goes out of scope its drop method is called which decrements the `Node`'s count.

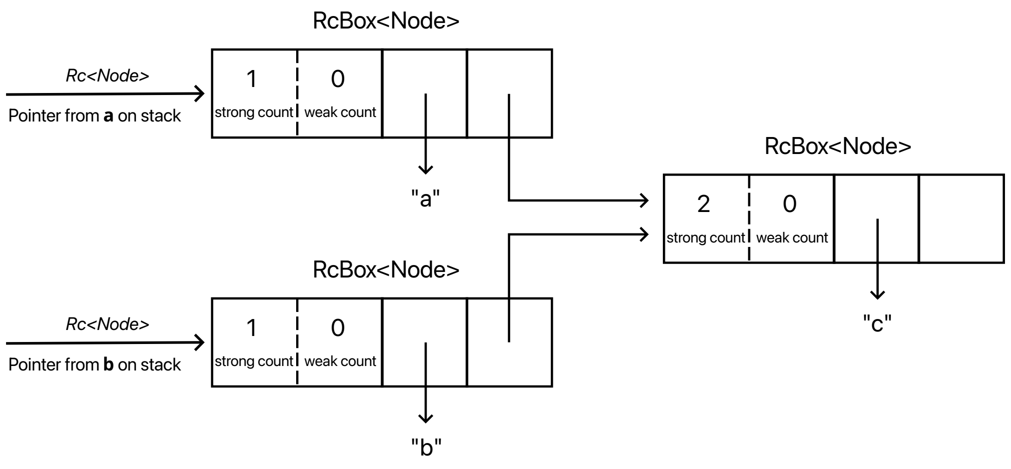


Figure 4.1: Memory representation of Listing 4.5, a shared data structure which uses `Rc`.

The `Rc` smart pointer in this example is single-threaded: that is, its count operations are not synchronised. Where reference counting is needed between threads, an `Arc` (atomic reference count) can be used instead.[7]

As enumerated in Section 2.2.1, reference counting comes with several disadvantages. Unfortunately, in Rust there are two more. First, obtaining a new reference must be done with an explicit call to `clone()`, which can make it cumbersome to use.

[7] C++ programmers will notice that this is similar to `std::shared_pointer`

```
let a = RefCell::new(1);
let my_borrow = a.borrow_mut();
*a.borrow_mut() = 2; // Runtime panic. Two mutable borrows!
*my_borrow = 3;
```

Listing 4.6: An example of a program which crashes at runtime due to multiple mutable borrows

Second, and more importantly, if more than one reference to an object exists, the compiler can no longer reason about the borrow rules. If we recall from Section 4.1.2: at any given time, an object can have either one mutable reference or any number of immutable references to it. The implication of this is that a reference counted object with more than one shared reference could never be mutated! Fortunately, Rust solves this with a design pattern known as *interior mutability*, which I explain in the next subsection.

4.2.1 Interior mutability

In Section 4.1.2 we see that Rust's borrow checker ensures that programs adhere to the borrow rules at compile-time. However, occasionally this can be too restrictive, preventing some otherwise memory-safe programs from compiling.

To address this, Rust uses a design pattern known as *interior mutability*, which allows values to be mutable when their ownership is shared, such as when using reference counting. Types which use interior mutability bypass the compile-time checking of the borrow rules, deferring the checks to runtime instead. This allows certain idioms that could not otherwise be encoded to be used, albeit with some runtime cost.

Interior mutability is achieved using a *mutable container*: a type which provides its inner type with mutation semantics. There are several mutable containers in Rust, but in this section we will look at the most common one, `RefCell`. Consider the following immutable `u64`, which is placed inside a `RefCell`:

```
let a = RefCell::new(1);
*a.borrow_mut() = 2
```

`a` is an immutable variable, so the `RefCell` allows it to be mutated through its `borrow_mut` method. Unlike the compile-time borrow rules surrounding ordinary Rust references, `borrow_mut` performs a runtime check to ensure that it is never called while an existing borrow to the `RefCell`'s content exists. Listing 4.6 shows how a program will panic at runtime when this is the case.

```
 1 struct Node {
 2     name: String,
 3     next: Option<Rc<RefCell<Node>>>,
 4 }
 5
 6 impl Drop for Node {
 7     fn drop(&mut self) {
 8         println!("The␣next␣node␣is␣{}", self.next.unwrap().borrow());
 9     }
10 }
11
12 fn main() {
13     let a = Rc::new(RefCell::new(Node {
14         name: String::from("a"),
15         next: None,
16     }));
17
18     let b = Rc::new(RefCell::new(Node {
19         name: String::from("b"),
20         next: None,
21     }));
22
23     a.borrow_mut().next.insert(Rc::clone(&b));
24     b.borrow_mut().next.insert(Rc::clone(&a));
25 }
```

Listing 4.7: An example of a reference counted cycle in Rust. This example creates two nodes, `a` and `b`, before creating cyclic references between them (lines 23-24). This causes a memory leak, resulting in the drop methods for `Node`, `Rc`, and `String` to never be called.

A `RefCell` is implemented by pairing its contents with a word-sized "borrow count" which is incremented and decremented by borrow operations. It therefore has both a performance and space overhead.

Interior mutability is made possible because of a concept known as *unsafe Rust*, where compile-time borrow checking can be disabled for code surrounded by `unsafe {}` blocks. As its name suggests, unsafe Rust is a dangerous tool because the programmer is responsible for ensuring that unsafe code does not violate memory safety. It is used to provide safe abstractions, such as `RefCell`, where compile-time borrow checking of its contents can be disabled in favour of its runtime interface.

4.3 Cyclic data structures

So far in this chapter I've shown two kinds of memory management techniques on offer in Rust: first, statically enforced ownership is used to manage memory without a garbage collector; and second, where this is too restrictive, opt-in reference counting can be

```
 1 struct Node {
 2     name: String,
 3     strong_next: Option<Rc<RefCell<Node>>>,
 4     weak_next: Option<Weak<RefCell<Node>>>,
 5 }
 6
 7 fn main() {
 8     let a = Rc::new(RefCell::new(Node {
 9         name: String::from("a"),
10         strong_next: None,
11         weak_next: None,
12     }));
13
14     let b = Rc::new(RefCell::new(Node {
15         name: String::from("b"),
16         strong_next: None,
17         weak_next: None,
18     }));
19
20     a.borrow_mut().strong_next.insert(Rc::clone(&b));
21     b.borrow_mut()
22         .weak_next
23         .insert(Rc::downgrade(&Rc::clone(&a)));
24 }
```

Listing 4.8: An example of breaking the cycle from Listing 4.7. Here an `Rc` is "downgraded" to a weak ref (line 23).

used for shared ownership. Unfortunately, neither of these options alone can address a particular programming style which looms large — writing cyclic data structures.

This section outlines two common ways to address this in Rust: breaking reference counted cycles with weak pointers; and using an arena which groups allocations with the same lifetimes together.

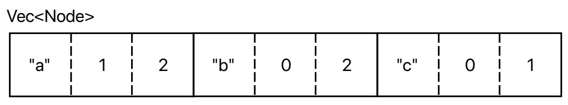
4.3.1 Reference counting with weak references

To understand how weak references in Rust can be used to break cycles, lets first create a strongly connected cyclic graph using reference counting. Listing 4.7 shows an example of a graph which will leak memory as its memory will never be collected.

Section 2.2.1 explains how weak references can be used to break such cycles due to them not forming part of the object's reference count. In Rust, this is achieved by *downgrading* `Rc` references to `Weak` references. In Listing 4.8 the cyclic data structure from the previous example in Listing 4.7 is modified so that the cyclic reference from `b` to `a` is downgraded to use a `Weak` reference instead.

```rust
struct Node {
    name: &'static str,
    edges: Vec<usize>,
}

impl Node {
    fn print(&self, arena: &Vec<Node>) {
        println!("Node␣{}␣points␣to:", self.name);
        for e in self.edges.iter() {
            println!("-->␣{}", arena[*e].name)
        }
    }
}

fn main() {
    let mut arena = Vec::new();

    let a = Node { name: "a", edges: vec![1, 2]};
    arena.push(a);

    let b = Node { name: "b", edges: vec![0, 2]};
    arena.push(b);

    let c = Node { name: "c", edges: vec![0, 1]};
    arena.push(c);

    for node in arena.iter() {
        node.print(&arena)
    }
}
```

(a) Implementation of an arena



(b) `Vec<Node>` in memory

Figure 4.2: An example of using an arena to represent a cyclic graph. The edges of each node are represented as indices into the arena's vector. This creates a form of "pseudo-reference", where shared ownership is modelled outside of Rust's type system.

4.3.2 Arenas

An alternative to using weak reference counting to implement cyclic data structures is to use *arenas*. An arena is a memory allocation strategy where a single chunk of memory can be pre-allocated and then used to store multiple objects with the same lifetime together [Hanson, 1990]. Arenas are commonly used as a performance optimisation when a user knows in advance that many objects of the same lifetime are going to be needed: instead of allocating each object individually, a chunk of memory can be pre-allocated for them, allowing them to all be freed at once.

In Rust, arenas are helpful for writing cyclic data structures [Goregaokar, 2021a]: objects in an arena can contain cyclic references between each other without fear of causing memory leaks, because they will all be deallocated once we are finished with them anyway.

Rust's ownership and borrow rules have no concept of arenas, so regular Rust references cannot be used cyclically between objects in a pre-allocated arena. Instead, this pattern is typically achieved through externally written libraries which use an existing, contiguous, heap allocating data structures. Figure 4.2 shows an example of how a vector can be used as an arena where each object refers to another object using its index in the vector.

Of course, using indices in this way is inflexible and causes us to lose many of the ergonomic benefits of dealing with references (such as automatic dereferencing). Libraries such as *elsa* [Goregaokar, 2018] and *typed-arena* [Chiovoloni, 2015] extend this fundamental idea with interior mutability to provide a cleaner API for arena based data structures.

Chapter 5

Alloy: a conservative garbage collector for Rust

This chapter introduces Alloy[1] – an implementation of Rust which supports opt-in garbage collection. Alloy uses the Boehm-Demers-Weiser garbage collector (BDWGC) 'under the hood' though Alloy tackles many Rust-specific challenges, and makes Rust-specific optimisations, that are independent of the collector implementation.

Alloy attempts to address the two main limitations with Rust's existing memory management approaches: writing cyclic data structures; and implementing or interfacing with other garbage collected languages. Alloy is not intended to replace these memory management approaches. Instead it gives Rust programmers the choice to make certain objects garbage collected in a way similar to the reference counting (`Rc`) library.

The chapter is structured as follows. Section 5.1 first explains why tracing GC is desired in Rust. In Section 5.2 I introduce Alloy, one of the main contributions of this thesis, outlining its goals and the design, before explaining how Alloy uses the BDWGC in Section 5.4. Finally, in Section 5.5 I show related work in the area of retro-fitting GC to Rust.

[1]In keeping with the metallic-themed naming tradition for Rust libraries, an alloy can contain iron so *technically* it can still rust!

5.1 Why provide a tracing GC for Rust?

5.1.1 Cyclic data structures

In Listing 4.3 I showed two techniques available in Rust for implementing cyclic data structures: using reference counting with weak pointers; and using arenas. Unfortunately, both approaches have clear ergonomic disadvantages.

Unfortunately, Section 4.3.1 shows that using reference counting with weak references to break cycles requires an awkward combination of: creating and manually cloning `Rc` references; interior mutability; and correctly downgrading strong references to weak references. In larger, more complex graphs this can become difficult to work with and error-prone: special care is needed to ensure that weak references are used correctly in order to avoid leaking data by creating inadvertent cycles.

The main disadvantage of using arenas for cyclic data structures is that individual objects cannot be deleted from the graph as all objects in the arena must have the same lifetime. For large graphs, such as representing DOM trees in a browser, the space overhead becomes too impractical for this to be used.

5.1.2 Implementing other languages

As a modern, memory-safe systems programming language, Rust has inevitably sparked interest as a language for implementing other languages [Project, 2018; West, 2018; Williams, 2018; WeirdConstructor, 2019]. In order to support tracing garbage collection in the target language, some form of GC support is also necessary for Rust. This is because a VM would both need to access GC'd objects from within Rust; and possibly even put objects created in Rust on the GC heap. Of course, this problem is not unique to Rust: V8 [Google, 2008] and WebKit [Apple, 1998] both implement tracing garbage collection to manage objects in their C++ runtime (see Section 3.7).

Tracing garbage collection for this purpose has been explored in various kinds: *luster* [West, 2018] (a Lua VM written in Rust), and *boa* [Williams, 2018] (a JavaScript VM written in Rust) both use various tracing GC implementations. In Section 5.5 I explain in detail how these tracing GCs work.

5.2 An introduction to Alloy

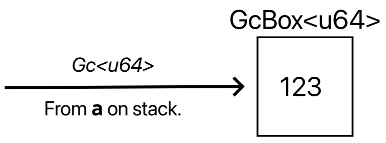
Alloy is an implementation of Rust which supports opt-in garbage collection. It has four main goals:

1. To provide opt-in garbage collection which can be used in cooperation with Rust's existing approaches to memory management (Chapter 4).
2. To make it easier and less error-prone to implement cyclic data structures in Rust than the existing options available in the language (Listing 4.3).
3. To be sound when used with safe Rust code: that is, using Alloy in safe Rust should not undermine Rust's normal safety guarantees.
4. To perform well enough to be used as a viable alternative when compared with other GC designs in Rust which are available.

5.2.1 Garbage collection with the Gc smart pointer

Alloy introduces a new smart pointer type, `Gc<T>`, which provides shared ownership of a value of type `T` allocated in the heap and managed by a garbage collector. Consider a simple example and its corresponding representation in memory:

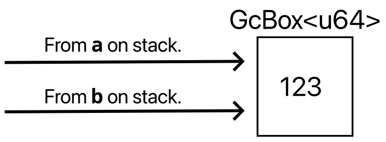
```
use std::gc::Gc;

fn main() {
    let a = Gc::new(123);
}
```



This creates a garbage collected object which contains the `u64` value `123`. A `Gc`'s data is stored in a `GcBox` internally. `GcBox`es are managed by the collector, though this is not visible to the user.

`Gc` references are copiable (i.e. they implement the `Copy` trait), with copied references pointing to the same object in the heap:

```
fn main() {
    let a = Gc::new(123);
    let b = a;
}
```



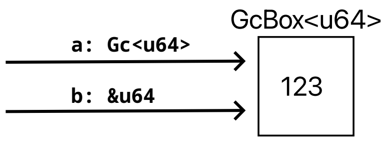
This makes `Gc` more ergonomic to use than `Rc`, as there is no need to call `clone` on a `Gc` to obtain another reference to its data.

The `GcBox` referenced by a `Gc` is guaranteed not to be freed while there are still references to it. When there are no longer any references, the collector will reclaim it at some point in the future. The garbage collector runs intermittently in the background, so `Gc` objects may live longer than they need to.

5.2.2 Dereferencing

A `Gc<T>` dereferences to `T` with the dereference (∗) operator:

```rust
fn main() {
    let a = Gc::new(123);
    let b = *a;
    foo(b);
}

fn print(int: u64) {
    println!("{}", int);
}
```



Here, the value can be copied out of the `Gc` into `b` because `u64`s are copiable. The `Gc` type also allows the dot operator to be used for calling methods of type `T` on a `Gc<T>`:

```rust
struct Wrapper(u64);

impl Wrapper {
    fn foo(&self) {
        ...
    }
}

fn main() {
    let a = Gc::new(Wrapper(123));
    a.foo();
}
```

5.2.3 Mutation

There is no way to mutate, or obtain a mutable reference (`&mut T`) to a `Gc<T>` once it has been allocated. This is because mutable references must not alias with any other references, and there is no way to know at compile-time whether there is only one `Gc` reference to the data.

As with other shared ownership types in Rust, interior mutability (Section 4.2.1) must be used when mutating the contents inside a `Gc`:

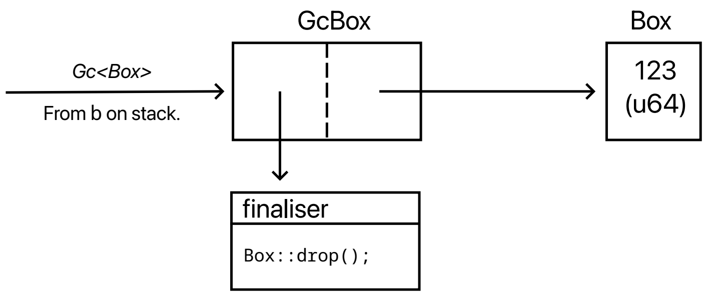
```rust
fn main() {
    let a = Gc::new(RefCell::new(123));
    *a.borrow_mut() = 456; // Mutate the value inside the GC
}
```

5.2.4 Finalisation

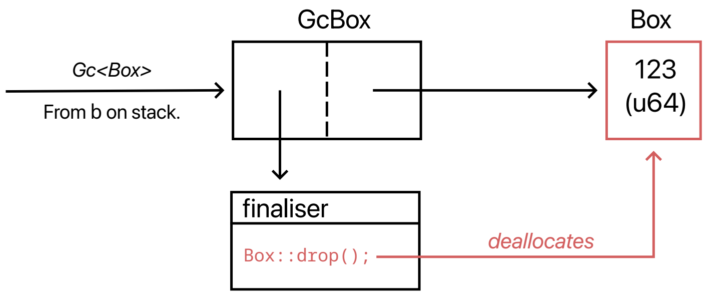
In Section 2.2.2, I give a gentle introduction to finalisation in garbage collection. In short, an object's finaliser in a GC runs some clean up code when that object becomes unreachable. ALLOY implements finalisers for a `Gc<T>` by calling `T`'s destructor when the collector determines it is no longer reachable. Consider the following example:

```rust
let a = Box::new(123);
let b = Gc::new(a);
```

Here `a` refers to a boxed integer on the heap which is not managed by the collector. It is placed inside a `Gc`, which has the following representation in memory:



Before the `Gc` referenced by `b` is collected, its finaliser is called, which calls drop on the non-GC'd box:



This allows a `Gc` object to "own" another, non-garbage-collected heap object, and deallocate its memory when it is no longer needed.

However, finalisation is a major source of challenges for a garbage collector in Rust: naively mapping them to Rust drop methods introduces soundness and performance problems. This is a large topic, so Chapter 6 explores the problems and Alloy's solutions in depth.

5.2.5 Concurrency

The Rust type system is able to guarantee statically that values are being used in a thread-safe way. It does this with the use of two special "marker" traits: `Send` and `Sync`.

A value whose type implements the `Send` trait can be transferred to other threads. Almost all types in Rust implement the `Send` trait save for a few exceptions. One such exception is the `Rc<T>` (reference counting) type. This does *not* implement `Send` because it does not perform count operations on its underlying contents atomically. If it were sent to another thread, it could race if both threads tried to update the count simultaneously. In contrast, the `Arc<T>` type (an atomic implementation of `Rc<T>`) does implement `Send` provided its inner type `T` is also `Send`.

The corollary to `Send` is the `Sync` trait, which can be implemented on types whose values perform mutation in a thread-safe manner. For example, a `Mutex<T>` implements `Sync` because it provides exclusive access to its underlying data atomically. On the other hand, a `RefCell` (discussed in Section 4.2.1) does not implement `Sync` because it only provides single-threaded interior mutability. This is because its underlying mechanism to increment and decrement borrow counts are not performed atomically.

The `Send` and `Sync` marker traits are a special kind of trait known as *auto traits*. This means that they are automatically implemented for every type, unless the type, or a type it contains, has explicitly opted out. Types can be opted out via a *negative impl*:

```
    impl !Send for T {}
```

If, for example, a struct `S` contains a field of type `T` from the example above, then the entire struct `S` will also not implement `Send`.

`Send` and `Sync` can be manually implemented on types, but doing so requires `unsafe` Rust code since the user must guarantee it is safe to use in a multi-threaded context.

Alloy's `Gc` reference is fundamentally thread-safe: if `T` implements `Send` + `Sync`, then `Gc<T>` will too. The `Gc<T>` type therefore conditionally implements `Send` + `Sync` depending on `T`:

```
unsafe impl<T: Send> Send for Gc<T> {}
unsafe impl<T: Sync> Sync for Gc<T> {}
```

5.3 An overview of soundness

Alloy is a conservative GC (Section 3.1.2), which means that by nature, it is unsound. This is because, technically speaking, the way conservative GC works violates the rules of most languages, most compilers, and most operating systems. In very rare cases, compilers have been known to perform optimisations which can obfuscate pointers from the collector [Google, 2020]. Fortunately, the ubiquity of conservative GCs in industrial strength VMs means that in practise it is well supported. If one can accept this caveat, Alloy is otherwise correct-by-design provided that users do not hide, accidentally or otherwise, pointers from the GC. Programming techniques which rely on *pointer obfuscation* [Boehm, 1996] are therefore not allowed in Alloy. This rules out the use of certain data structures such as XOR lists.

In a GC for Rust, soundness violations can easily occur in object finalisation. Alloy prevents this with modifications to the Rust compiler to ensure that all finalisers are sound at compile-time. I explain these issues in depth, in addition to how Alloy solves them, in Chapter 6.

The rest of this section begins by explaining how Alloy uses the BDWGC to identify which objects are reachable when performing a collection. It then describes how some Rust programs can inadvertently hide such references from the collector, and the restrictions this places on the user in order to ensure that programs can be correctly written with Alloy.

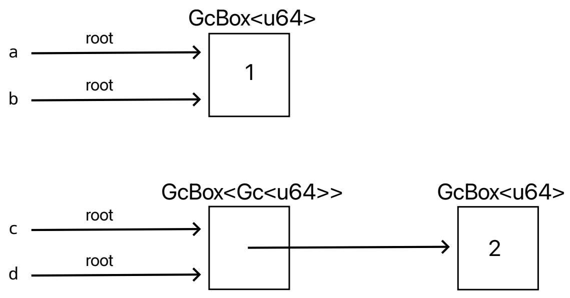
5.3.1 Determining the reachability of garbage collected objects

Finding the roots

When Alloy performs a collection, it must first identify the roots from which the rest of the object graph is traced. Such roots exist on the call stack, in registers, and in segments of the program which store global values. When a collection is scheduled, the BDWGC spills register values to the stack so that their contents can be scanned for pointers along with the rest of the stack [Boehm and Weiser, 1988]. The call stack is exhaustively examined for possible pointers to instances of objects, with each aligned word on the stack is checked to see whether it points to an instance of an object: if it does, that object is considered a root. Figure 5.1 shows an example of what Alloy considers roots to garbage collected objects.

```
// ‘a‘ exists on the stack.
let a = Gc::new(1);
let b = a; // obtain copy


let c = Gc::new(Gc::new(2));
// obtain a rust (&) ref
let d = c.as_ref();
```

(a) Roots on the stack



(b) Representation in memory

Figure 5.1: An example showing values on the stack which are considered roots to `GcBox`es.

Garbage collected objects in other heap objects

In Alloy, references to garbage collected objects can be stored in traditional, non-garbage-collected Rust objects:

```
fn main() {
    let v = Vec::new();
    v.push(Gc::new(1));
    v.push(Gc::new(2));
}
```

Here, the vector contains two references to garbage collected objects which are managed by Alloy. Even though the vector itself is not garbage collected, its backing store must still be traced during a collection in order to locate GC objects. To support this, every allocation in a Alloy must use the BDW allocator – even those which are not garbage collectable. This is because the BDW allocator stores bookkeeping information such as mark bits and the memory block size which are needed during a collection.

Rust provides a convenient way to set the global allocator for a program, and because Alloy extends the standard library to include the BDW allocator, programs can easily be made Alloy-compliant. This is shown in Listing 5.1.

This ensures that every heap allocation (except those created using a `Gc::new()`) is made using the BDW allocator's `GC_malloc_uncollectable` function. This allocates a non-garbage-collected block which the collector is aware of and can scan for pointers to other garbage collected objects. As with the call stack, the BDWGC scans all allocated blocks in memory that are reachable from the root-set conservatively word-by-word.

```rust
use std::gc::GcAllocator;

#[global_allocator]
static ALLOCATOR: GcAllocator = GcAllocator;

fn main() {
    ...
}
```

Listing 5.1: Setting the global allocator to use the BDWGC in Alloy using Rust's `global_allocator` attribute.

5.3.2 Pointer obfuscation

As a systems programming language, Rust permits operations directly on pointers. This includes: casting pointers to and from integer types; pointer arithmetic; and bitwise operations on pointers. All three of these operations can be used to obfuscate a pointer, hiding it from the collector and causing it to erroneously determine that an object is unreachable. In order for Alloy to be sound, the user must not obfuscate any pointers in this way.

Pointer casting and word alignment

For Alloy to be able to locate pointers during a marking, all references, raw pointers, and machine-word sized integers (`usize`) must be word-aligned. This is because Boehm scans the stack and heap blocks for pointers word-by-word, so non-word-aligned values may be missed.

It is easy to see that Alloy needs to be able to identify objects via references or raw pointers, and thus requires them to be word-aligned. The reason this is also true for `usize`s is more subtle, and is necessary because it is possible in Rust to cast between raw pointers and word-sized integers using the `as` keyword. Consider the following:

```rust
let gc = Gc::new(Value);
let gc_ref = gc.as_ref(); // Get a &Value reference.
let ptr_to_int = (gc_ref as *const Value) as usize;
```

We first obtain a reference to the `Value` stored inside the `Gc` before casting it to a raw pointer, and then a `usize` (a word-sized unsigned integer). If `gc` and `gc_ref` were to go out of scope, the `ptr_to_int` is enough to keep the GC'd object alive, because when Alloy scans the stack, it would correctly identify that `ptr_to_int` looks like a pointer to a valid GC object.

```
 1 fn make_obfuscated() -> usize {
 2     let a = Gc::new(123_u64);
 3 
 4     // Get a raw pointer to the underlying allocation
 5     let aptr = a.as_ref() as *const u64;
 6 
 7     // Use the bitwise NOT operator to obfuscate the pointer
 8     return !(aptr as usize)
 9 }
10 
11 fn main() {
12     let obf = make_obfuscated();
13 
14     ...
15 
16     // GC cycle here. The ‘Gc‘ is potentially unreachable!
17 
18     let reify = (!obf) as *const u64;
19 
20     // ‘unsafe‘ is needed to dereference a raw pointer
21     let value = unsafe { *reify };
22 }
```

Listing 5.2: An example of how pointer obfuscation in Rust can hide a pointer from the collector. ALLOY uses the BDWGC to conservatively scan the stack, so this allocation could be missed if its only remaining reference is the one obfuscated on line 8. Fortunately, however, it requires an unsafe block to dereference (line 21).

Fortunately, Rust references must always be word-aligned anyway: constructing a non-aligned Rust reference is already undefined behaviour. Howver, there is no such existing requirement for Raw pointers or `usize`s. When initialized on the stack, or boxed on the heap, `usize` values are already word aligned. The problem comes as they can be stored as non-aligned fields in a *packed struct*.

ALLOY includes a compiler lint, enabled by default, which prevents users from writing packed structs which misalign pointer fields, shown in Listing 5.3. Fortunately, packed structs are strongly discouraged in Rust anyway because of undefined behaviour due to misalignment that can occur when using them with references.

5.4 The collector

ALLOY uses the Boehm-Demers-Weiser GC (BDWGC) as the collector implementation [Boehm and Weiser, 1988]. The BDWGC is a conservative mark-sweep collector. Similar to V8's GC in Chapter 3, the BDWGC scans the Rust program's call stack conservatively to look for pointers to garbage-collected objects when marking. The rest of this section describes the features available in BDWGC which are used by ALLOY.

```
#[repr(packed)]
struct Packed {
    x: u16,
    y: *mut usize,
}
```

(a) Misaligned struct

```
error: packed structs cannot contain
    *mut usize.
   |
LL |   #[repr(packed)]
   |   --------------- help: remove this
   |   attribute
LL | / struct Packed {
LL | |     x: u16,
LL | |     y: *mut usize,
LL | | }
   | |_^
```

(b) Lint error in Alloy

Listing 5.3: A lint preventing a packed struct with a misaligned pointer from compiling in Alloy. The `repr(packed)` attribute forces Rust to strip any padding, and only align the struct `Packed` to a byte. This would cause the field `y` to be missed by Alloy during its conservative marking as it is no longer word-aligned.

5.4.1 Blacklisting

Alloy uses the BDWGC's blacklisting mechanism. This means that if, during marking, the collector identifies a bit-pattern which resembles a pointer to a block which is not yet allocated, it will prevent objects from being allocated in that block in the future (i.e. blacklist them). This is designed to reduce false-positives in pointer identification which can leak memory.

5.4.2 Incremental marking and parallel collection

Alloy uses the BDWGC's incremental marking to reduce the latency. This means that instead of using single stop-the-world (STW) pause, bits of marking are performed in smaller chunks that interleave with the mutator to reduce the latency of the program. This does not require any modifications to Alloy as it identifies modified objects by relying on the OS to determine which pages have been dirtied.

The BDWGC uses parallel collector threads for both mark and sweep phases.

5.4.3 Non-moving generational collection

The BDWGC is a non-moving collector. However, it provides a form of non-moving generational GC using sticky mark-bits which Alloy enables by default. This is implemented by not clearing the mark bit between minor collections. This way, objects that were live at the previous collection are considered the old generation [Demers et al., 1989].

5.4.4 Parallel mutator threads

In order to support thread-safe `Gc`s in Alloy, the BDWGC must be able to scan each thread's call stack for roots. I extend the Rust compiler to register newly spawned threads with BDWGC's collector, and to unregister them when they are destroyed.

Alloy relies on the BDWGC's signal spin implementation to come to a GC safepoint. That is, when a mutator thread comes under allocation pressure and needs to schedule a GC, the BDWGC will send a SIGPWR signal to each registered thread and has them spin in a signal handler while the collection cycle takes place.

The main disadvantage of this approach is that it makes use of implementation-defined behaviour because it relies on the target OS's mechanism for pausing threads. The BDWGC provides implementations for most platforms, but it is not portable. An implementation where Rust inserts thread pause safepoints at appropriate locations would largely solve these issues, though at the expense of considerable implementation effort.

5.5 Related work

This section gives an overview of existing approaches to including garbage collection in Rust in particular, though it is worth noting that there has been interest in using Rust as a library for building GCs for other languages [Lin et al., 2016, 2017].

Throughout Rust's history, there have been several attempts to introduce some form of tracing garbage collection [Klock, 2015, 2016; Goregaokar, 2016]. In fact, early versions of Rust explored using a form of this as a first class feature of the language through the use of *managed pointers* (with the syntax `@T`). This was removed fairly early in Rust's development before the first stable release, and was only implemented as reference counting. Since then, there have been several attempts at a more advanced form of GC than the reference counting library `Rc` to the language. This section will explain those approaches, and how they differ from Alloy.

The Bacon-Rajan cycle collector

bacon-rajan-cc [Fitzgerald, 2015] is a Rust library implementation of the Bacon-Rajan cycle collecting reference counting implementation [Bacon and Rajan, 2001]. bacon-rajan-cc is single-threaded, and provides thread-local reference counted boxes with the `Cc<T>` type. Though not intended as a general purpose GC, it is designed to make it easier to manage cyclic data structures in Rust. Listing 5.4 shows a small example of how

```rust
struct Node {
    name: String,
    next: Option<Cc<Node>>
}

fn main(){
    let a = Cc::new(Node { name: String::from("a"), next: None });
    {
        let b = Cc::clone(&a); // create new ref ‘a‘, incrementing the ref count.
    } // ‘b‘ goes out of scope, decrementing the ref count.
}
```

Listing 5.4: An example of using BACON-RAJAN-CC. This looks very similar to reference counting in Rust with `Rc`. However, because the reference count decrement for `b` does not cause the count to reach zero, a pointer to the shared string value is added to an internal thread-local `ROOTS` vector in order to be checked for potential cycles later.

it can be used as a drop-in replacement for `Rc<T>`. The key point to note is that once a reference count is decremented via `Cc`'s drop method, if it does not cause a count to reach zero, a pointer to the value is added to a `ROOTS` vector for later processing.

One can then manually trigger cycle reclamation by calling the `collect_cycles` function, which performs a local trace over the values in the `ROOTS` worklist looking for `Cc` cycles.

In order to know how to trace an object's fields looking for other `Cc`s, BACON-RAJAN-CC provides a `Trace` trait, which must be implemented for `T` in order to be used in a `Cc<T>`. For `Cc<Node>`, the `Trace` implemention would look like this:

```rust
impl Trace for Node {
    fn trace(&self, tracer: &mut Tracer) {
        // Tell the trace method it must traverse ‘next‘
        // during cycle detection.
        self.next.trace(tracer)
    }
}
```

BACON-RAJAN-CC provides implementions of `Trace` for most standard library types, including `Option<T>`, which makes `Cc<Node>` possible. This can then be used instead of `Rc` to build cyclic data structures which do not leak. Listing 5.5 shows an example of a cyclic graph using BACON-RAJAN-CC.

Unlike ALLOY, BACON-RAJAN-CC is purely library based: it does not require any modifications to the Rust compiler to work. In addition, as a reference counted approach, it has the advantages and disadvantages of reference counting over ALLOY's tracing GC – except, of course, that it can collect cycles!

```
 1 use bacon_rajan_cc::{collect_cycles, Cc, Trace, Tracer};
 2 use std::cell::RefCell;
 3
 4 impl Trace for Node {
 5     fn trace(&self, tracer: &mut Tracer) {
 6         // Tell the trace method it must traverse ‘next‘
 7         // during cycle detection.
 8         self.next.trace(tracer)
 9     }
10 }
11
12 struct Node {
13     name: &’static str,
14     next: RefCell<Option<Cc<Node>>>,
15 }
16
17 impl Drop for Node {
18     fn drop(&mut self) {
19         println!("Dropping␣{}", self.name)
20     }
21 }
22
23 fn main() {
24     let a = Cc::new(Node {
25         name: "a",
26         next: RefCell::new(None),
27     });
28
29     let b = Cc::new(Node {
30         name: "b",
31         next: RefCell::new(None),
32     });
33
34     // Create cyclic references between nodes ‘a‘ and ‘b‘.
35     *a.next.borrow_mut() = Some(b);
36     *a.next.borrow_mut().as_ref().unwrap().next.borrow_mut() = Some(Cc::clone(&a));
37
38     drop(a);
39     collect_cycles(); // Prints:
40                       // "Dropping b"
41                       // "Dropping a"
42 }
```

Listing 5.5: An example of creating and then reclaiming a small cyclic graph without leaks using BACON-RAJAN-CC. This example creates two nodes, `a` and `b`, before creating cyclic references between them using interior mutability (lines 35-36). The trait `Trace` is implemented for `Node`, informing BACON-RAJAN-CC that it must traverse the `next` field during cycle collection (lines 4-10). Later, the explicit call to `drop` (line 38) decrements `a`’s reference count, but it is not deallocated because the cycle prevents the count reaching zero. It is only after the `collect_cycles` call (line 39) that both `a` and `b` are dropped and then deallocated.

```rust
fn main() {
    // The underlying ‘GcBox‘ containing the string has a root count of 1.
    let a = Gc::new(String::from("Hello"));

    // Create a new gc pointer, the ‘GcBox‘ root count is 2.
    let b = a.clone();

    // Put the cloned gc pointer into a heap vector, the root count is
    // unchanged.
    let mut c = Vec::new();
    c.push(b);

    // Create a new GC object, ‘a‘ is unrooted, and ‘d‘ becomes the root
    // instead.
    let d = Gc::new(a);
}
```

Listing 5.6: An example showing what values are considered roots for garbage collection in Rust-GC.

However, there are two limitations unique to bacon-rajan-cc. First, it does not support multi-threading (though a concurrent cycle detection algorithm is possible [Bacon and Rajan, 2001]). Second, cycle reclamation must be manually triggered by the mutator, which could lead out-of-memory problems if cyclic garbage reaches a certain size and is not collected. However, one can imagine a threshold-based approach to cycle collection where cycle detection happens once `ROOTS` reaches a certain size. For example, the size of `ROOTS` could be checked when new values are inserted, preventing the need for the mutator to trigger collection.

Rust-GC

Rust-GC [Goregaokar, 2015] is library for Rust which provides optional single-threaded mark-sweep GC with the `Gc<T>` type. The API for Rust-GC is similar to Alloy, with the notable exception that `Gc` in Rust-GC does not implement the `Copy` trait. This means that in order to obtain additional pointers to garbage-collected objects, the `Gc` must be cloned.

Rust-GC is implemented as a hybrid form of reference counting and tracing GC. There is no mechanism for scanning the stack for roots as in traditional GC, so roots are tracked using reference counting, with a mark-sweep then performed from these roots. Like Alloy, `Gc` references in Rust-GC point to an underlying `GcBox`. However, in Rust-GC, this `GcBox` maintains a count of all of its roots. Listing 5.6 shows how this count is updated as references are used.

```rust
1  #[derive(Trace)]
2  struct S<T> {
3      a: Gc<T>,
4      // ‘U‘ never points to a ‘Gc‘, so we can avoid tracing
5      // it entirely.
6      #[unsafe_no_trace]
7      b: U,
8  }
9
10 struct U {
11     a: u64,
12     b: String,
13 }
```

Listing 5.7: An example using the `unsafe_no_trace` annotation. There is no `Gc` reachable from the struct `U`, so it can be excluded from the derived `Trace` implementation (line 1) with the annotation on line 6. This annotation is unsafe because if `U` is later modified to include a field of type `Gc`, it would not be traced, and the collector would miss a reference.

During a collection, the `GcBox`’s on the heap are enumerated, and those with a non-zero root count are used as roots to begin marking. Like bacon-rajan-cc, Rust-GC traces through objects by requiring types used in `Gc` to implement a `Trace` trait, which has a `trace` method called during marking to traverse and mark objects during a collection.

Rust-GC makes implementing `Trace` easy by providing a macro implemention, where types can be annotated with `#[derive(Trace)]` and have it implemented automatically. It also provides an optimisation on tracing where if the programmer is certain that a field does not contain a reference (transitively or directly) to another `Gc` type, they can annotate that field with `#[unsafe_no_trace]` to opt-out of tracing (shown in Listing 5.7).

Listing 5.8 shows an example of a cyclic data structure implemented in Rust-GC. It uses its own type, `GcCell` in order to support interior mutability (Section 4.2.1) as a `RefCell` cannot be used with Rust-GC. The `GcCell` provides additional support for rooting and unrooting objects across a borrow as they are mutated inside the `Gc`. It provides a similar API to the user as `RefCell`.

Unlike Alloy, objects are finalised by implementing a special `Finalize` trait. This reduces much of the complexity that Alloy needed in order to support calling `T::drop` from a finaliser or destructor context, which was a major implementation challenge (discussed in Chapter 6). However, Rust-GC requires the programmer to ensure that a finaliser implementation is present for any type that may need to call `Drop` on any of its component types. It is not easy to know which of these component types may need dropping, and forgetting to do so can cause memory leaks.

```
 1 use gc::{Finalize, Gc, Trace, GcCell};
 2
 3 #[derive(Trace)]
 4 struct Node {
 5     name: &'static str,
 6     next: GcCell<Option<Gc<Node>>>,
 7 }
 8
 9 // Types inside 'Gc<T>' cannot have a 'Drop' implementation.
10 impl Finalize for Node {
11     fn finalize(&self) {
12         println!("Finalizing␣{}", self.name)
13     }
14 }
15
16 fn main() {
17     let a = Gc::new(Node {
18         name: "a",
19         next: GcCell::new(None),
20     });
21     let b = Gc::new(Node {
22         name: "b",
23         next: GcCell::new(None),
24     });
25
26     *a.next.borrow_mut() = Some(b);
27     *a.next.borrow_mut().as_ref().unwrap().next.borrow_mut() = Some(Gc::clone(&a));
28 } // Prints:
29   //  "Finalizing b"
30   //  "Finalizing a"
```

Listing 5.8: An example of creating a small cyclic graph using Rust-GC. This example creates two nodes, `a` and `b`, before creating cyclic references between them using interior mutability (lines 26-27). The trait `Trace` is automatically derived for `Node` (line 3), informing Rust-GC that it must traverse the `next` field during a collection. In addition, deriving `Trace` implements an empty `Drop` trait on `Node` to prevent users from providing their own `Drop` implementation on types used inside a `Gc`. For "drop"-like behaviour to be run when a value is collected, we must instead implement the `Finalize` trait (lines 10-14). Before the program exits, Rust-GC explicitly triggers a collection, causing the objects pointed to by `a` and `b` to be collected and the finalisers to be run.

```
 1 fn make_vec() -> Vec<GcRef<u64>> {
 2     let mut v = Vec::new();
 3     for i in 0..12 {
 4         v.push(GcRef::new(i));
 5     }
 6     v
 7 }
 8
 9 fn main() {
10     let v = make_vec();
11
12     // Create a new ‘GcRef‘, which will invoke a collection due to
13     // allocation pressure. Bronze will not consider ‘v‘ a root which
14     // transitively points to managed objects.
15     let gc = GcRef::new(String::from("Hello␣World"));
16
17     // The following causes a use-after-free.
18     // The managed object pointed to by the ‘GcRef‘ at ‘v[1]‘ was GC’d.
19     println!("Vec[1]:␣{}", v[1])
20 }
```

Listing 5.9: An example of unsoundness using Bronze because it does not trace GC objects which are pointed to transitively via other objects. In this example, a vector `v` is created containing many garbage collected references to `u64`s. The `u64`s are allocated on the heap and managed by the Bronze collector. A collection in Bronze is triggered in `GcRef::new` when the Bronze allocation threshold reaches a certain size (in bytes). This is artificially triggered by the allocation on line 15, as we filled `v` with enough `GcRef`s to come just under the threshold. Later, when trying to access a value from inside `v` on line 19, we violate memory safety in the form of a use-after-free.

The Bronze collector

The *Bronze* collector is an optional GC implementation for Rust which was designed address usability concerns with Rust's borrow semantics [Coblenz et al., 2022]. It was designed alongside an empirical study which measured how long it took students to complete a variety of Rust tasks by using standalone Rust, and Rust with Bronze for managing memory.

Bronze bases much of its implementation on Rust-GC but with two key differences. First, it tracks roots to GC objects by using a modified version of the Rust compiler. Bronze's rustgc implementation inserts calls to LLVM's `gc.root` intrinsic at function entries in order to generate stackmaps. When a GC call is requested, Bronze iterates over the stackmaps generated its current call stack in order to locate the roots for garbage collection. However, Bronze does not implement this for transitive references from arbitrary objects. In other words, if a `Gc<T>` exists as a field inside another object instead of directly on the stack, it would not be tracked as a root for garbage collection, shown in Listing 5.9.

The second major difference between Rust-GC and Bronze is that Bronze's `Gc<T>` type allows the programmer to dereference its underlying type `T` mutably more than once.

```rust
 1 fn main() {
 2     let mut gr1 = GcRef::new(vec![1u16,2,3]);
 3     let mut gr2 = gr1.clone();
 4
 5     let ref1 = gr1.as_mut();
 6     let ref2 = gr2.as_mut();
 7
 8     // ref1 and ref2 now reference the same object:
 9     ref1.push(4);
10     ref2.push(5);
11     ref1.push(6);
12
13     let ref1elem0 = ref1.get_mut(0).unwrap();
14     // Force reallocation of the underlying vec
15     ref2.resize(1024, 0);
16     // Now this writes to deallocated memory
17     *ref1elem0 = 42;
18 }
```

Listing 5.10: An example of unsoundness in Bronze based on its ability to allow aliased mutable references. Here, we obtain two mutable references to the same underlying vector (lines 5-6), before using the second reference to resize the vector, which forces its backing store to be reallocated in memory (line 15). Later, when we try to access an element through the first reference, it no longer points to valid memory (line 15).

Coblenz et al. [2022] describes this as beneficial, because it makes it easier to use than other Rust shared ownership. However, this is fundamentally unsound, and allows programs which violate memory safety to be written in safe Rust using Bronze. Listing 5.10 shows an example of how this can violate memory safety by causing a write from deallocated memory.

Shifgrethor

Shifgrethor [withoutboats, 2018] is an experimental GC API for Rust which investigated a way for potential GC implemenations to precisely identify and trace roots to GC'd objects. Shifgrethor is therefore not a GC, but instead an experimental design for how a GC could interface with the language.

The basic idea is that in order to create a `Gc` object, it must be created by, and exist alongside a corresponding `Root<'root>` type on the stack. The `Root<'root>` can then dish out references to the underlying `Gc` which are tied to `Root<'root>`'s lifetime. This is similar in nature to how handle scopes work in V8 (Section 3.2).

GC arena

GC-ARENA [West, 2019] is another experimental approach at sound GC design in Rust. It was originally developed as part of the *luster* VM [West, 2018]: an experimental Lua VM written in Rust. Unlike ALLOY and the other approaches to GC in Rust seen so far, GC-ARENA does not retrofit Rust with a GC. Instead, it provides limited garbage collection in isolated garbage collected *arenas*. Arenas carefully guard mutator access to their objects through closures, which, when executing, prevent the collector from running. This solves the difficult problem of finding roots which reside on the stack: when an arena is *closed* to the mutator, no stack roots exist, so a collection can be safely scheduled. A single arena may contain several garbage collected objects, but they cannot be transferred between other arenas.

Because GC-ARENA is so different in nature to the other GCs described in this chapter, it is difficult to compare it ergonomically to other approaches.

Chapter 6

Finalisation in Alloy

A major challenge for Alloy is to ensure that finalisers are both sound and perform well. This chapter first explains some of the implementation problems that any GC with finalisers must consider, before explaining considerations that are unique to Rust. I then explore the different ways that finalisers could be implemented in Alloy, and why I decided on the approach that was used.

This chapter is structured as follows. Section 6.1 explains some of the common pitfalls and of implementing finalisers. In Section 6.2 I explain in detail how Rust's existing destructors work. Section 6.4 introduces Alloy's approach to finalisation. Finally, Section 6.5 to Section 6.8 are part of the main contributions in this thesis. They explain how Alloy deals with the various soundness and performance problems caused by finalisers in a Rust GC.

6.1 Finalisers background

Finalisers are a common component of most tracing GCs which are used to run code for cleanup once an object dies (e.g. closing a file handle or a database connection). Unfortunately, finalisation is fraught with problems, many of which are subtle and difficult to detect. In this section, I outline these general issues across languages and implementations, and how existing GCs have mitigated them. This is necessary to understand the design decisions that Alloy has made to ensure that its own finalisers are sound.

```
 1 class Count {
 2     public static Count globalCount = null;
 3     int count = 0;
 4 
 5     @Override
 6     protected void finalize() {
 7         // Resurrect object by keeping a reference to it alive
 8         globalCount = this;
 9     }
10 }
```

Listing 6.1: An example of a Java program with object resurrection. The finaliser for `Count` places a reference to its instance in a static, causing the collector to keep it alive (line 8). If Java allowed `finalise` to be called more than once, this object would never be reclaimed, as it would keep being resurrected.

6.1.1 Finalisers are not guaranteed to run

Most GC specifications are usually cautious about making guarantees about finaliser behaviour, often for correctness reasons. In fact, in any GC, finalisers are never guaranteed to be called at all. While this is true for destructors in languages such as C++ and Rust too, its more likely to happen with finalisers for several reasons. First, the window between when an object is last used by the mutator and when it is finalised is greater, so the opportunity for missed finalisers is greater if the program exits within this window.

Second, a program may accidentally hold onto a reference to an object which the programmer expects to be freed, thus preventing its finaliser from being called. In some cases, this type of memory leak can be due to programmer error, and is not a problem exclusive to finalisers. However, because a GC can over-approximate the root-set when marking, this can happen by means outside of the programmer's control. For example, a conservative GC may retain objects from a previous collection (known as *floating garbage*) because of a lingering reference in a disused stack slot. This would prevent those objects' finalisers from ever being called.

Finally, note that in Section 6.1.3, a GC which guarantees finalisation order will not finalise objects with finalisation cycles because it cannot determine a safe order in which to run them.

6.1.2 Object resurrection

Object resurrection is where a finaliser method stores a reference to its object in a global data structure or the field of another object, preventing it from being reclaimed. Object resurrection can lead to objects being kept alive longer than expected and is often difficult

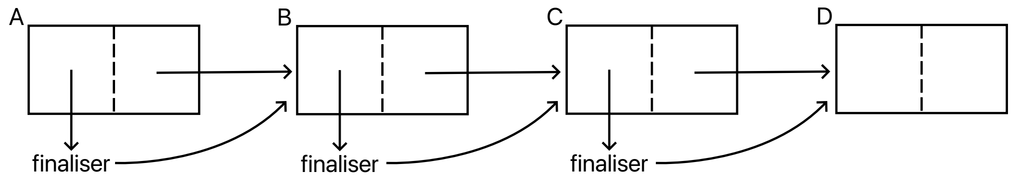
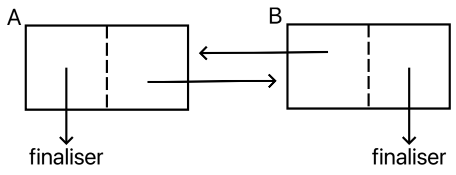
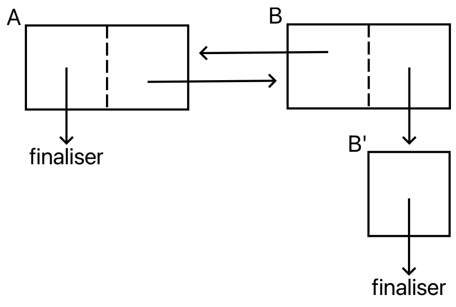


Figure 6.1: An example of ordered finalisation where the finalisers reference their inner objects. This object graph will be finalised in order from the outermost layer (**A**) to the innermost (**D**). Finalising this object graph requires four finalisation cycles because each inner object is reachable from the outer layer.



(a) Original order



(b) Refactored to handle cycles

Figure 6.2: An example from Jones et al. [2016, p. 218] which shows how objects can be restructured to force finalisation order in cyclic object graphs.

to detect. If an object's finaliser could run more than once, a resurrected object would be re-finalised and thus resurrected again ad infinitum (Listing 6.1). To prevent this, most GC'd languages such as Java and C# guarantee that an object's finaliser will be run at most once. This is also true of Alloy.

6.1.3 Finaliser ordering

A language implementation has a choice as to whether to guarantee that finalisers will run in a certain order. Some GCs implementations guarantee a finalisation order, because for some applications, this is important if one resource must be cleaned up before another.

For example, consider the objects in Figure 6.1. The finalisers for the "outer" objects all reference the object in the layer to the right of them, so they must be finalised "outside-in".

However, this guaranteed finalisation order has two disadvantages. First, finalising all the objects in a chain of floating garbage happens over multiple finalisation cycles because an object can only be finalised if it is not reachable from other unreachable objects. In Figure 6.1, this would require four finalisation cycles before the floating garbage is reclaimed. Such a delay in the eventual reclamation of objects can cause *heap drag*, where unreachable objects are kept alive longer than necessary.

Second, and most significantly, this approach is not able to finalise cycles of objects where more than one object needs finalising, which can lead to resource leaks. This is because the collector cannot know which (if any) object is safe to finalise first: if an object references another object which has already been finalised, this is unsound. Boehm proposes a workaround for this where programmers can refactor the objects in order to break the finalisation cycle [Boehm, 2003] (Figure 6.2 from Jones et al. [2016, p. 218]). Another workaround is to allow users to use weak references to break cycles (similar to breaking reference count cycles). Unfortunately this is often difficult to implement correctly [Jones et al., 2016].

Some GCs such as Blink's Oilpan GC [Ager et al., 2013] do not specify a finalisation order. This permits the GC to finalise cycles of objects where more than one object needs finalising with an important restriction placed on the programmer: an object's finaliser must not reference any other object. This caveat is necessary to prevent dereferencing an already finalised object. This approach is used in V8: finalisers are not able to dereference through to other JavaScript objects. In the rare cases where this is too restrictive, Oilpan also has pre-finalisers, which are run during the GC's stop-the-world pause, and allow object fields to be dereferenced [Ager et al., 2013]. The compromise here is performance: pre-finalisers increase the mutator pause time and cannot be enqueued to run later.

6.1.4 Finalisers and deadlocks

It is common in many languages for finalisers to access fields from other objects or even global state. Since an object's finaliser is run at some unknown point in time once it is considered unreachable by the collector, it must be able to safely access such state without racing with the mutator.

It is common in many languages for finalisers to access fields from other objects or even global state. Since an object's finaliser is run at some unknown point in time once it is considered unreachable by the collector, it must be able to safely access such state without racing with the mutator. If a finaliser runs on a separate thread to the mutator, it is clear that data accessed from a finaliser must be performed in a way which is thread-safe. However, what is less obvious is that for correctness reasons, finalisers cannot be scheduled to run on the mutator thread.

Listing 6.2 shows how finalisers run on the same thread as the mutator can cause a deadlock. When the finaliser for an instance of `Counter` is called, it tries to acquire the lock for the object's `count` member. If this happens while the mutator already holds the lock, the program will deadlock. While one might consider using re-entrant locks to solve this problem, it can in fact worsen the problem by turning an obvious failure into a race

```
 1 class X {
 2     Count count;
 3 
 4     @Override
 5     protected void finalize() {
 6         count.increment();
 7     }
 8 
 9     public static void main(String[] args) {
10         Count count = new Count();
11         X x = new X(count);
12 
13         ...
14 
15         count.increment();
16 }
17 
18 class Count {
19     Mutex mutex = new Mutex();
20     int count = 0;
21 
22     public void increment() {
23         try {
24             mutex.acquire();
25             count++;
26         }
27         catch (InterruptedException e) {
28             e.printStackTrace();
29         } finally {
30            mutex.release();
31         }
32     }
33 }
```

Listing 6.2: An example Java program showing how a finaliser could deadlock if run from the same thread as the mutator. The `Count` object has a method `increment` which acquires a mutex (line 22). A new instance of `Count` is created and a reference to it passed to a new instance of `X` (lines 10-11). If the collector decided to invoke `X.finalize()` while the mutator executes `count.increment()` on line 15, then the program will deadlock as they both try to acquire the mutex on the same thread. This can be solved by ensuring that finalisers run on a separate thread to the mutator: the finaliser thread would simply spin or sleep while it waits for the mutator to release the mutex.

where two logically separate operations mutate data in a way which should be performed atomically. This can cause intermittent incorrect execution [Boehm, 2003].

For this reason, most languages run finalisers on a separate thread from the mutator [Boehm, 2003]. Though programmers must still ensure they access shared state via some form of synchronisation such as mutexes, they are able to spin or sleep when blocked, allowing the mutator to progress without deadlocking.

6.1.5 Finalisers can run earlier than expected

Unlike RAII-like destructors found in languages such as C++ and Rust, finalisers are called by a garbage collector non-deterministically. They can run at the collector's leisure; often this means that they run later than desired, however, in rare cases, an object can be finalised while it is still being used by the mutator! This is because compiler optimisations – unaware of the presence of a GC – can remove the single reference to an object which is keeping it alive. An outer object can therefore be considered unreachable while its inner object is still in use. An unfortunately timed GC cycle could end up finalising the outer object, and run its finaliser. This can lead to subtle races in programs where the finaliser interleaves execution with the mutator.

For this reason, VM specifications do not commit to running finalisers at a specific time. This includes allowing an object's finaliser to be run while the mutator is potentially still using it. For the reasons outlined in Section 6.1.4, GC implementations must synchronise access to objects inside a finaliser. Jones et al. [2016, p .218] suggests this can be used to defer finalisation until later if a finaliser attempts to acquire an object lock which is already held by the mutator.

More generally, however, the fundamental problem is that the compiler optimises away the reference to the object too soon. C#'s .NET runtime provides a `gc.KeepAlive` function as a solution to this. `gc.KeepAlive` is an opaque empty function which the compiler cannot optimise away. The idea is that a reference to an object can be passed to `gc.KeepAlive`, ensuring it lives long enough so that the collector does not deem it unreachable and finalise it too soon. This mitigation is limited, however, as it is up to the user to call `gc.KeepAlive` when they require it.

6.2 Rust destructors

To understand the design decisions that Alloy makes surrounding finalisation, some background is needed on Rust's destructors.

Destructors in Rust were briefly introduced in Section 4.1.1 as a way of running cleanup code when an initialized variable or temporary goes out of scope. A destructor is run automatically at the end of the scope for values which implement the `Drop` trait. Consider the following example, which uses a Rust destructor to close a file descriptor:

```rust
struct FileDescriptor {
    fd: u64
}

impl Drop for FileDescriptor {
    fn drop(&mut self) {
        self.close();
    }
}

fn main() {
    let f = FileDescriptor { fd: 1 };
} // FileDescriptor::drop called.
```

The file descriptor `f` is destructed (or *dropped*) at the end of the `main` function, where it is no longer in scope. The ability to drop objects is a key component of Rust's ownership semantics, and is used extensively in the standard library.

A struct which has a drop implementation may have fields which also need dropping. For example, consider a `FileBuffer`, which has a field of type `FileDescriptor`:

```rust
struct FileBuffer {
    descriptor: FileDescriptor,
}

impl Drop for FileBuffer {
    fn drop(&mut self) {
        self.flush();
    }
}
```

Here, both the `FileBuffer` and its field `FileDescriptor` have drop methods which need running. Rust will automatically insert calls to drop them both when a `FileBuffer` value goes out of scope. In Rust terminology, a value is considered dropped once its drop method, and all drop methods belonging to its fields, have been dropped.

Rust drop order

Drop methods are used by Rust programmers for situations such as releasing locks. In such cases, the order in which values are dropped is vital for program correctness.

Variables and temporaries are dropped in reverse declaration order. For example:

```rust
fn main() {
    let s1 = String::from("s1");
    let s2 = String::from("s2");
    let s3 = String::from("s3");
}
```

At the end of `main`, `s3` would be dropped first, followed by `s2`, and finally `s1`.

Rust specifies that fields are dropped in declaration order. For example, consider the following struct definition:

```rust
impl Drop for S {
    fn drop(&mut self) {
        println!("Dropping␣S");
    }
}

struct S  {
    a: String,
    b: u64, // u64 does not implement the ‘Drop‘ trait.
    c: Vec<bool>,
}
```

`S` contains two fields (`a` and `c`) which also need dropping. Rust will drop `S` first, followed by the field `a`, and then the field `c`.

If any component of a type implements `Drop`, Rust will drop them when they go out of scope. For example, consider an enum `E` (a tagged union), where one variant needs dropping:

```rust
enum E  {
    A(String),
    B(bool),
}
```

Even though `E` does not have a drop method, when it goes out of scope, Rust will still insert a drop call because the variant `E::A` contains a droppable type, `String`. It is not possible to know at runtime which variant of the enum is active, so Rust inserts some additional code which checks dynamically which variant (if any) to drop.

Since Rust ensures that drop methods are called automatically, it is not possible to call the drop method for a value (or any of its fields) directly. This ensures that a value is only dropped once, an important protection against double-freeing resources. This can be restrictive, because sometimes it is useful to drop a value earlier than at the end of its scope. Consider a common example, where a `Mutex`'s lock is released from its drop method:

```rust
fn main() {
    let mutex = Mutex::new(123); // A mutex which guards a u64 value.
    let data = mutex.lock().unwrap();
    println("locked␣value:␣{}", data);

    // Code that shouldn't belong in the critical section
    ...
} // lock is released as part of drop.
```

By unlocking the mutex at the end of main, sometimes we can execute more code than is necessary in the critical section. Rust provides a standard library helper function, `std::mem::drop` which can accept values of any type in order to drop them early:

```rust
1  fn main() {
2      let mutex = Mutex::new(123); // A mutex which guards a u64 value.
3      let data = mutex.lock().unwrap();
4      println("locked␣value:␣{}", data);
5
6      // Release the lock early
7      std::mem::drop(data);
8
9      // Code that shouldn't belong in the critical section
10     ...
11 } // lock is released as part of drop.
```

`std::main::drop` is implemented as an empty function. Since ownership of `data` is transferred (line 7), Rust will insert a call to `data`'s drop method immediately afterwards.

Drop methods are not guaranteed to run

Destructors in Rust are guaranteed to run at most once — but they may not be run at all. This is for three reasons.

First, consider the example back in Chapter 4 (Listing 4.7) where a cycle is created between two `Rc` values. A reference cycle such as this introduces a memory leak and thus the values in this data structure are never dropped.

Second, values are only dropped if they are initialized. It is not always possible to know whether a value is initialised so Rust can sometimes end up performing dynamic checks to know whether a value should be dropped. The details of this are not relevant to the rest of the thesis.

Third, one can explicitly prevent a value from being dropped by passing it to the `std::mem::forget<T>` function. This is commonly used when the underlying resource originated from non-Rust code, and therefore destruction of it should happen outside of Rust.

What types can be dropped?

In short, Rust will automatically call drop for any type which implements the `Drop` trait when it goes out of scope. However, copiable types (those which implement the `Copy` trait, mostly primitive types such as `bool`s, `char`s, numeric types and so on) cannot implement Drop because doing so would mean that when values are copied they would be dropped multiple times. This would violate Rust's guarantee that drop is called at most once.

Rust also supports C-like union types, which in contrast to enums do not use runtime tags to denote the active variant. Union types are not automatically dropped because there is no way for Rust to know which variant to insert a drop method for.

6.3 Design choices for finalisers in Rust

Before explaining finalisation in ALLOY, we should ask an over-arching design question: what should a finaliser in a Rust GC look like? Other approaches to GC in Rust, such as RUST-GC and BRONZE, define a custom `Finalize` trait, which types can implement to specify finaliser behaviour when they are used in a `Gc` (shown in Listing 6.3).

The benefit of this approach is that it creates a logical separation between destructors expected to run in an RAII based context, and GC finalisers. This allows finalisers, which

```
 1 struct S;
 2
 3 impl Drop for S {
 4     fn drop(&mut self) {
 5         println!("Dropping␣S");
 6     }
 7 }
 8
 9 impl Finalize for S {
10     // Run before collection when value used in a ‘Gc‘.
11     fn finalize(&mut self) {
12         println!("Finalizing␣S");
13     }
14 }
15
16 fn main() {
17     let s1 = S;
18     let s2 = S;
19
20     let gc1 = Gc::new(s2);
21 } // Dropping s1
```

Listing 6.3: An example from Rust-GC, where a custom `Finalize` trait is used for finalisation semantics. In this scenario, before `s2` is collected, Rust-GC calls `S`'s `finalize` method (line 11).

have subtly different rules to destructors, to be correctly specified by the user (as we will see in Section 6.5, there are specific restrictions that need to be placed on finalisers in Rust in order to guarantee soundness).

Alloy takes a different approach, however, as separating destruction and finalisation in this way has unfortunate consequences. First, for most types that already implement `Drop`, their destruction logic must be duplicated in a finaliser. This is, at least, significant extra effort; it also offers many opportunities for copy and paste errors.

Second, a separate `Finalize` trait has as a major ergonomic cost because it is not possible to implement `Finalize` on code from external libraries. This is because Rust enforces *trait coherence*, a property in the language which ensures that every type has at most one implementation of a given trait. This coherence rule is fundamental to the language, because it removes ambiguity in trait method resolution, ensuring there is only one implementation of a trait method to choose from.

Trait coherence is a problem for programs that use external compilation units known as *crates* (roughly speaking, ‘libraries’), because if two unrelated crates provide separate implementations for the same trait, then those crates cannot be imported together.

To address this, traits in Rust must adhere to something called the *orphan rule*. The rule is simple: it is not possible to implement a trait for a type where both the trait

```
use a::{MyType, MyTrait};

impl MyTrait for MyType {
    fn method1() {
        ...
    }

    fn method2() {
        ...
    }
}
```

(a) Invalid trait implementations

```
error[E0117]: only traits defined in the current
              crate can be implemented for types
              defined outside of the crate
 --> src/lib.rs:3:1
3 | impl MyTrait for MyType {}
  | ^^^^^^^^^^^^^^^^^------
  | |                 |
  | |                 ‘MyType‘ is not defined in
  | |                 the current crate
  | impl doesn’t use only types from inside the
  | current crate
```

(b) Compiler error

Listing 6.4: Here, we try to provide an implementation of the externally defined trait, `MyTrait` for the externally defined type, `MyType`. This results in a compile error in Rust because it violates the orphan rule.

and the type are defined in separate crates. This prevents multiple conflicting trait implementations from existing across crates. Listing 6.4 shows how the orphan rule is enforced at compile-time in Rust.

The problem with the orphan rule is that it would become a major source of ergonomic frustration for Alloy if it defined a separate `Finalize` trait. It would not be possible to implement `Finalize` for any type which was not defined in the user’s current crate. If types from external crates do not provide their own implementations for `Finalize`, then those types may cause resource leaks when used in a `Gc`.

A workaround for the orphan rule is to use the *new-type idiom*, where the current crate defines a wrapper type for an external type. Unfortunately, this workaround can be cumbersome to write and makes types in Rust harder to read. Listing 6.5 shows how the new-type idiom can be used to add a finaliser to a type defined outside of the current crate. This can be used in other GC designs for Rust which use a separate `Finalize` trait such as Rust-GC.

6.4 Finalisers in Alloy

Alloy takes a novel approach to finalisation compared to previous Rust GCs in that it uses existing drop methods as garbage collection finalisers, saving users the potentially error-prone task of manually writing both desctructor and finaliser methods for GC managed objects. Despite this, Rust’s semantics do not expect GC, so Alloy must also consider soundness issues that other GC’d languagess have not had to face. In the rest of this chapter, I explain how Alloy addresses these issues. Finally, as destructors in Rust are used frequently, methods are pervasive in Rust, this implies that finalisers must also

```rust
use a::MyType;

struct Wrapper(MyType);

impl Finalize for Wrapper {
    fn finalize(&mut self) {
        println!("Finalizing␣MyType␣via␣Wrapper");
    }
}

fn main() {
    let a = Gc::new(Wrapper(MyType::new()));
}
```

Listing 6.5: A workaround the orphan rule using the *new type idiom*. Here, a new `Wrapper` type is defined for which we define a finaliser. To garbage collected `MyType` objects, one could then use `Gc<Wrapper>` instead of `Gc<MyType>` to ensure that its finaliser is called.

be pervasive. Since finalisers impose significant performance costs, one only wants to run them when strictly necessary. Alloy thus introduces a number of static analyses that are able to remove most finalisers at compile-time.

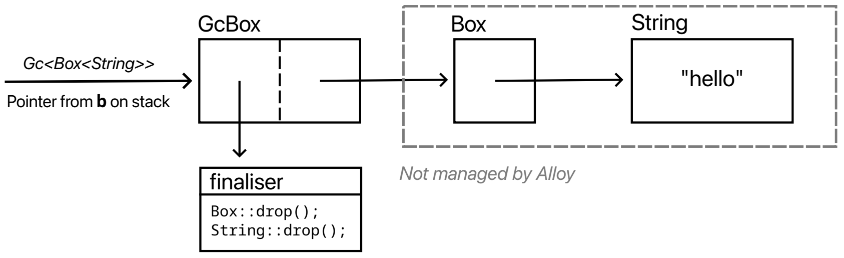
In Alloy, a `Gc<T>` will call `T::drop` when `T`'s value is unreachable. For example:

```rust
struct S;

impl Drop for S {
    fn drop(&mut self) {
        println!("Dropping␣S");
    }
}

fn main() {
    let s1 = S;
    let s2 = S;

    let gc = Gc::new(s2);
} // Dropping s1
```

Two values of type `S` are created; `s1` is dropped normally at the end of `main`. `s2`, however is moved inside a `Gc`, so in order for its drop method to be called, Alloy finalises it once it becomes unreachable. Consider another example, where a `Gc` box stores an owning reference to other, non-GC'd values on the heap:

```rust
let a = Box::new(String::from("Hello"));
let b = Gc::new(a);
```

The sole owning reference to the heap allocation `Box<String>` is moved into `Gc::new`, which creates a `Gc` object containing the reference to the `Box<String>`. This has the following representation in memory:



When the `Gc`'s underlying allocation (called GcBox) becomes unreachable, Alloy will call its finaliser, which means that `drop` is called on all the component types (in the same way that Rust automatically calls drop in Section 6.2). If, for whatever reason, the finaliser is not run, then the allocations for the `Box` and the `String` will leak (i.e. their heap allocation will never be reclaimed). I thus define a finaliser in Alloy as calling drop on the contents of a `Gc` (including its field types). Therefore a type `Gc<T>` has a finaliser if type `T` needs dropping.

6.4.1 Omitting finalisers

Finalisation is not always desirable. For example, consider a `FileDescriptor` which uses its drop method to close the descriptor:

```
struct FileDescriptor {
    fd: u64
}

impl Drop for FileDescriptor {
    fn drop(&mut self) {
        self.close();
    }
}
```

Here, objects of type `Gc<FileDescriptor>` would use a finaliser to call the `FileDescriptor`'s drop method. However, if we were to close the descriptor in the mutator once we are finished with the object, the finaliser is no longer necessary:

```
let stdout = FileDescriptor { fd: 1 };
let descriptor = Gc::new(stdout);
...
descriptor.close()
```

To allow for this, Alloy provides a special wrapper type, `NonFinalizable<T>`, which can be used to create a `Gc` which omits finalisers on an individual basis:

```
let descriptor = Gc::new(NonFinalizable::new(stdout));
```

Here, when the `Gc<NonFinalizable<FileDescriptor>>` is collected, it will not be finalised. The `NonFinalizable<T>` type has no additional storage costs, and at runtime is represented as a bare `FileDescriptor`.

This is only intended to be used in exceptional circumstances where performance is a concern: It can easily lead to resources leaks if not used carefully.

6.5 Safely finalising objects off-thread

In Section 6.1.4, I explained that running finalisers on the same thread as the mutator can cause deadlocks. Listing 6.6 shows how this could happen in a hypothetical version of Alloy where finalisers are scheduled to run from the mutator thread.

As with any GC, Alloy can finalise objects at its leisure, with no way for the programmer to know when this will happen. Rust does not prevent users from writing code which deadlocks, but it does not make the situation worse. However, if Alloy were to perform on-thread finalisation, it would open the possibility of previously non-deadlocking code deadlocking.

One possible solution might seem to be to prohibit finalisers from acquiring locks, however this can cause race-like bugs because of how finalisers can interleave asynchronously with the mutator [Matsakis, 2013].

6.5.1 Off-thread finalisation

Alloy finalises unreachable objects on a separate finalisation thread. This means that finalisers cannot deadlock simply by acquiring a lock held by the mutator: the finalisation thread will simply wait and attempt to re-acquire the lock later. However, this now means that shared data, or other objects accessed from a finaliser must be done in a thread-safe way. The problem with this is that in Alloy, a finaliser calls a type's existing drop methods. Since `Drop` was not originally defined in expectation of being called on a separate thread, it does not guarantee thread safety.

The most obvious solution to this is to ensure that only thread-safe types can be used inside a garbage collected container. In other words, a type `T` could not be placed inside a

```
 1 use std::gc::Gc;
 2 use std::rc::Rc;
 3 use std::sync::Mutex;
 4
 5 struct CounterWrapper {
 6     value: Rc<Mutex<usize>>
 7 }
 8
 9 impl Drop for CounterWrapper  {
10     fn drop(&mut self) {
11         let mut count = self.value.lock().unwrap();
12         *count += 1;
13     }
14 }
15
16 fn main() {
17     // Create a reference counted counter object protected by a mutex.
18     let counter = Rc::new(Mutex::new(0));
19
20     {
21         // Create a new garbage collected object which contains a reference
22         // counted pointer to the counter. When this object is collected, the
23         // drop methods for ‘CounterWrapper‘ and ‘Rc‘ will be called as
24         // finalizers.
25         let gc = Gc::new(CounterWrapper { value: Rc::clone(&counter) });
26     }
27
28     // Assume GC can happen here because ‘gc‘ is unreachable.
29
30     // If the finalizer for the ‘Gc<CounterWrapper>‘ is asynchronously called
31     // while ‘counter.lock()‘ below is already acquired, a deadlock will occur.
32     let lock = counter.lock().unwrap();
33     ...
34
35     assert_eq!(*counter, 0);
36 }
```

Listing 6.6: An example showing how a potential deadlock can be caused if finalisers are run on the mutator thread. A shared counter is created using a reference counted container (line 18), a reference to this is then placed inside a garbage collected container (line 25). This is potentially short-lived, as it is only reachable by the `gc` variable until the end of the inner scope (line 26). If Alloy decides to schedule a collection after this (e.g. on line 28) then the `CounterWrapper` could be considered garbage, where a finaliser would run its drop method (line 10). If this happens while the main mutator thread already holds `counter.lock()` (line 32-36) then this program can deadlock. When finalisers are run on the same thread, there is nothing the programmer can do in this situation to guarantee that this kind of deadlock does not occur.

```
 1 use std::cell::Cell;
 2 use std::gc::Gc;
 3 use std::thread;
 4
 5 struct Counter(Cell<usize>);
 6
 7 // ‘Counter‘ is not ‘Sync‘ by default because it contains a ‘Cell‘. However,
 8 // it doesn’t have a drop method, so it’s safe to use inside a ‘Gc‘. In order to
 9 // do this, we must explicitly mark it as ‘Sync‘ so that ‘Gc‘ accepts it.
10 unsafe impl Sync for Counter {}
11
12 fn main() {
13     let gc = Gc::new(Counter(Cell::new(0)));
14     let gc2 = gc;
15
16
17     // By explicitly implementing ‘Sync‘ for ‘Counter‘, we’ve accidentally
18     // allowed it to be used across threads in ways unrelated to finalisers
19     // which are not thread-safe. E.g. the following can race, as the counts are
20     // not updated atomically.
21     thread::spawn(move || {
22         for i in 0..1000 {
23             gc2.0.set(gc.0.get() + 1);
24         }
25     });
26
27     for i in 0..1000 {
28         gc.0.set(gc.0.get() + 1);
29     }
30 }
```

Listing 6.7: An example of the `Send` + `Sync` dilemma if Alloy required `T: Send + Sync` for `Gc<T>` to provide finaliser safety. Such a restriction on `T` is overly strict, and could lead to users explicitly marking types as `Send` or `Sync` so Alloy considers them safe for finalisation. This example shows how this can lead to inadvertantly bypassing Rust's concurrency safety when such types are used elsewhere. By default, `Counter` cannot be used inside a `Gc` because it contains a field `Cell` which does not implement `Sync`. To allow the construction of `Gc<Counter>`, `Sync` can be explicitly implemented on `Counter` (line 10). However, this now allows objects of this type to be used across threads without any synchronisation (lines 21-29), which can cause surprising races.

`Gc` unless `T` implements both `Send` and `Sync` – Rust's builtin traits for concurrency safety (see Section 5.2.5). This solution would prevent programs from compiling if an object without synchronisation is placed inside a `Gc` container. While this would ensure that finalisers are thread-safe, it is less than ideal for two reasons.

First, it would restrict a `Gc` from managing many valid types: a non-`Send` and non-`Sync` type would be prevented from being used in a `Gc` even if it does not have a drop method (and therefore, never needed finalising in the first place!).

Second, for `T` to be `Send` and `Sync`, all of `T`'s component types must be `Send` and `Sync` too. This presents a dilemma: either every field of `T` must be thread-safe (even those which

```rust
use std::rc::Rc;
use std::gc::Gc;

fn main() {
    let rc1 = Rc::new(123);
    let rc2 = Rc::clone(&rc1);
    let gc1 = Gc::new(rc2);
}
```

```
error: ‘rc2‘ cannot be safely finalized.
    --> src/main.rs:9:22
     |
9    |     let gc = Gc::new(rc2);
     |                      ^^^ has a drop method
     |                          which cannot be
     |                          safely finalized.
     |
    ::: /rust/library/alloc/src/rc.rs:1559:13
     |
1559 |            self.inner().dec_strong();
     |            ------------
     |            |
     |            caused by the expression in
     |            ‘fn drop(&mut)‘ here because
     |            it uses a type which is not
     |            safe to use in a finalizer.
     |
     = help: ‘Gc‘ runs finalizers on a separate
     | thread, so drop methods must only use values
     | whose types implement ‘FinalizerSafe‘.
```

Listing 6.8: A `Gc` object stores a reference counted integer (`Rc<u64>`). The `drop` method of an `Rc` decrements the underlying object's reference count, which is called as a finaliser before collecting the `Gc<Rc<u64>>` object. This is not thread-safe because decrementing a reference count in `Rc` is non-atomic, and the finaliser will be run on a separate thread, which would race if the finaliser ran while the mutator also updated the count. FSA detects this, and prevent the program from compiling, with the error message in **(b)** displaying the exact line of code that FSA rejected. In this example, as is often the case in practice, the offending line is not *in* the finaliser but is *called* by the finaliser.

are never used in a finaliser); or, the user, certain in the knowledge that `T`'s `drop` method is thread-safe, forcibly `unsafe` implements `Send` and `Sync` on `T`. In the case of the latter approach, `T` can then be accidentally be used in concurrency contexts unrelated to garbage collection, bypassing an important part of the type system in order to keep Alloy happy. Listing 6.7 shows how this could cause a leaky abstraction which introduces bugs in non-GC related code.

6.5.2 Finaliser safety analysis

Alloy uses a novel technique to ensure that finalisers are sound called *Finaliser Safety Analysis* (FSA). The basic idea is to encode finalisation rules into Rust's type system, and then perform a conservative static analysis to ensure a type's `drop` method does not use any types which would be non-thread safe in a finaliser. Listing 6.8 shows how this can prevent a potential race condition due to thread-unsafety.

FSA permits a `Gc` to contain values with non-thread-safe fields provided they are not used in the `drop` method. Consider the following example:

```
 1 struct Wrapper(RefCell<String>);
 2
 3 impl Drop for Wrapper {
 4     fn drop(&mut self) {
 5         println!("Dropping␣Wrapper");
 6     }
 7 }
 8
 9 impl Wrapper {
10     fn swap_string(&self) {
11         *self.0.borrow_mut() = String::from("b");
12     }
13 }
14
15 fn main() {
16     let gc = Gc::new(Wrapper(RefCell::new(String::from("a"))));
17     gc.swap_string();
18 }
```

The `Wrapper` uses a `RefCell` to swap the value of underlying string (line 11). A `RefCell` provides a form of interior mutability which is not thread-safe (because its `borrow()` / `borrow_mut()` methods are non-atomic). In this example, a `Wrapper` can safely be placed inside a `Gc`, because the `RefCell` is not used in `Wrapper`'s finaliser (line 4). This is checked by FSA at compile-time.

Automating finaliser safety analysis

Finaliser safety analysis is performed automatically without needing to do anything manually. First, I introduce a new auto trait used as a marker for finaliser safety called `FinalizerSafe` (an introduction to auto traits is provided in Section 5.2.5). As an auto trait, `FinaliserSafe` is implemented for all types by default in Rust, so in the Rust standard library, I explicitly remove the implementation of `FinalizerSafe` from types which do not already implement `Send` and `Sync`.

The Rust compiler is then extended to perform FSA. The basic idea is that whenever a type is used in a `Gc`, that type's `drop` method needs to be checked to ensure it doesn't access a field which is not `FinalizerSafe`. Performing this check only when such types are used in `Gc` is important as it prevents FSA from breaking existing Rust programs:

`drop` methods with unsound finalisation behaviour are not a problem if they are never used in a `Gc`.

Implementation

The Rust compiler pipeline lowers a Rust program into various different intermediate representations (IR) which it performs its analysis on. FSA is performed on Rust's mid-level IR (MIR), which represents a control-flow graph of a Rust program. MIR represents a Rust program as a collection of *MIR bodies*, which map to a single Rust function. A *MIR body* consists of a set of *basic blocks* connected by edges known as *terminators*. Basic blocks represent a list of straight-line statements, where terminators represent the control flow in the program.

FSA is a flow-sensitive analysis, so MIR is the most natural representation of a Rust program to perform its analysis on. FSA is implemented as a new *MIR pass* – a traversal over the MIR where each MIR body can be individually processed. We describe the algorithm for FSA in stages using pseudocode as follows.

The first stage of FSA is to identify calls to `Gc::new`:[1]

```
function FinaliserSafetyAnalysis($prog$)
    for each $mir\_body \in prog$ do
        for each $basic\_blocks \in mir\_body$ do
            for each $block \in basic\_blocks$ do
                if IsCallToGcConstructor($block.terminator$) then
                    CheckCallsiteForDropImpl($block.terminator$)
```

This checks every statement in the MIR for a call to `Gc<T>::new` constructor. If found, we check if `T` implements `Drop`. If it does, then `Gc<T>` needs finalising, so the MIR body for `T`'s `drop` method is checked for soundness violations. FSA only considers a `drop` method sound if fields which are dereferenced implement the `FinalizerSafe` trait. In Rust's MIR terminology, such a field access would constitute a *place projection*, where a place is a memory location (or lvalue), and a projection is a field access. A single MIR statement can contain more than one field projection (e.g. `self.foo.bar.baz`).

[1]In Alloy, a `Gc` object can only be created through the `Gc::new` constructor. We mark the definition of this function with a special label, known as a *diagnostic label*, so that it can be easily referred to during the FSA phase of Rust compilation.

This part of the analysis happens in the `CheckCallsite` function:

```
function CheckCallsiteForDropImpl(callsite)
    arg_ty ← GetTypeOfFirstArg(callsite)
    if not Impls(arg_ty, Drop) then
        return
    drop_body ← GetDropMirBody(arg_ty)
    for each basic_blocks ∈ drop_body do
        for each block ∈ basic_blocks do
            for each statement ∈ block do
                if HasPlaceProjection(statement) then
                    for each projection ∈ statement do
                        CheckProjection(projection)
```

If a projection is found, its type is checked for an implementation of the `FinalizerSafe` trait:

```
function CheckProjection(projection)  ▷ A projection elem is the RHS of a field
access.
    projection_ty ← GetType(projection.elem)
    if not Impls(arg_ty, FinalizerSafe) then
        return Error
```

If `CheckProjection` discovers a field access of a field which does not implement `FinalizerSafe`, it will throw a compiler error. This does not halt the analysis, so multiple lines in `drop` methods which perform unsound field accesses will be caught in a single FSA pass.

Limitations

As with any static analysis, FSA is inherently conservative: some `drop` methods are impossible to analyse at compile-time so in these cases Alloy will err on the side of caution and reject potentially safe programs. This is most likely to happen in two situations.

First, a `drop` method may contain a call to an opaque (i.e. externally linked) function for which the compiler does not have the MIR. If a reference to a field which would be unsafe to use in a finaliser is passed to this function, then FSA would reject the program. This function call *could* be safe, but FSA has no way of knowing, and will reject it.

Second, if a finaliser is used on a trait object which is called using dynamic dispatch in Rust. At compile-time, it is not possible to know the concrete type of a trait object, so FSA does not know which `drop` method to check.

In both cases, the user has the option of explicitly informing the compiler that a particular `drop` method is safe to use as a finaliser. We observe that this is rare in practice.

```
 1 struct Wrapper<'a>(&'a u64);
 2
 3 impl<'a> Drop for Wrapper<'a> {
 4     fn drop(&mut self) {
 5         println!("Dropping␣{}", self.0);
 6     }
 7 }
 8
 9 fn main() {
10     {
11         let b = Box::new(123);
12         let gc1 = Gc::new(Wrapper(b.as_ref()));
13     } // b is dropped
14
15     // GC happens here, calling 'Wrapper::drop' as a finaliser.
16     // This causes a use-after-free.
17 }
```

Listing 6.9: The `Gc` now stores a reference to the boxed `u64` using an intermediate `Wrapper` struct which has a `drop` method which will be run as a finaliser (line 4). The boxed `u64`, and the `Gc` are created in an inner scope. At the end of this scope, `b` is dropped. Later, when the `Gc<Wrapper>` is collected (e.g. at line 16) a use-after-free will occur because its finaliser (line 4) dereferences a reference to a value which was already dropped (at line 14).

6.6 Finalisers and Rust references

Like other smart pointers in Rust, ALLOY's `Gc` container can store ordinary Rust references. Without finalisers, this is perfectly safe, because Rust's borrow rules prevent the references from outliving their referent. Consider the following Rust program:

```
1 fn main() {
2     let b = Box::new(123);
3     let gc1: Gc<&u64> = Gc::new(b.as_ref());
4
5     foo(b);
6     // println!("Boxed value: {}" gc1); // ERROR: use-after-move
7 }
8
9 fn foo(b: Box<u64>) {}
```

A `u64` is boxed on the heap, and an immutable reference to it is stored in a `Gc`. This is valid as long as `gc1` does not outlive `b`. If line 6 is uncommented, this program would not compile, because Rust identifies that printing the `gc1` would try to dereference the boxed `u64` after it has been moved (line 5). This would seem to suggest that Rust's borrow rules are enough to allow references to be used inside `Gc` soundly.

Unfortunately, using references stored in a `Gc` from a finaliser can be unsound. The collector is free to schedule a finaliser to run at any point after a garbage collected

```
 1 use std::gc::Gc;
 2 use std::sync::Arc;
 3
 4 struct Wrapper<'a>(Arc<&'a u64>);
 5
 6 impl<'a> Drop for Wrapper<'a> {
 7     fn drop(&mut self) {
 8         println!("Dropping␣{}", self.0);
 9     }
10 }
11
12 fn main() {
13
14     {
15         let val = 0;
16         let a = Arc::new(&val);                    // Arc ref count = 1
17         let gc = Gc::new(Wrapper(Arc::clone(&a))); // Arc ref count = 2
18     } // ‘a‘ dropped, Arc ref count = 1
19
20     // GC happens here, calling ‘Wrapper::drop‘ as a finaliser.
21     // This is unsound, because ‘Wrapper‘’s ‘Arc‘ keeps the ‘&'a u64‘ alive,
22     // but not the data it points to.
23
24 }
```

Listing 6.10: A reference to `val` is stored in an `Arc`, which is then shared inside a `Gc`. While both the `Gc` and `Arc` go out of scope at the same time as the referent `val`, the finaliser for `Wrapper` can be called at some pointer later, each in lines commented below (20-22), causing a dangling pointer dereference when `&u64` is invalid.

object is unreachable, which could mean running after a value it references is dropped. Listing 6.9 shows how a program which uses a Rust reference can cause a use-after-free.

To make matters worse, even references reachable from other shared ownership types are unsafe to access from a finaliser. Consider a slightly modified example in Listing 6.10, where instead of storing a `&u64` reference directly, the `Gc` contains an atomic reference counted field (`Arc`), which stores a `&u64` reference.

6.6.1 Preventing dangling references with the borrow-or-finalise rule

For a `Gc<T>` with a finaliser to be sound, it cannot be used with an object which contains ordinary Rust references. To guarantee this, Alloy imposes a restriction called the *borrow-or-finalise rule*. This rule states that that a type `T` cannot be placed in a `Gc` if both of the following conditions are true:

Condition 1. *`T` (or any component type of `T`) is of some type `&U` or `&mut U`.*

Condition 2. *`T` has a finaliser. In other words, `T` (or any component type of `T`) has a `drop` method.*

```
struct S<'a> {
    a: &'a str
}

struct T<'a> {
    a: u64,
    b: S<'a>,
}

enum U<'a> {
    A(u64),
    B(String),
    C(S<'a>)
}
```

Listing 6.11: We assume each definition has a `drop` implementation. Values of the first struct `S` cannot be passed to a `Gc` because its field, `a` contains an immutable reference to a string literal (`&str`). The struct `T` cannot be passed to a `Gc` either because it contains a transitive reference, through field `b`.

Finally, an object of type `Gc<U>` is not possible because the `U::C` variant contains a reference. The active variant cannot be known statically, so Alloy disallows it entirely.

As with references, this same unsoundness can be caused by storing raw pointers inside a `Gc` (either immutable `*const` or mutable `*mut`) and then dereferencing them in a finaliser. However, Alloy is sound even without enforcing this rule for raw pointers because dereferencing raw pointers is already not possible in safe Rust code, and programmers must ensure the reference is valid for each dereference anyway. Listing 6.11 shows an example of some types which do not adhere to the borrow-or-finalise rule.

Automating the borrow-or-finalise rule

Alloy checks that programs adhere to the borrow-or-finalise rule at compile-time, throwing an error for those programs which violate it. Attempting to compile the earlier example would result in the following error message:

```
error: 'Wrapper(b.as_ref())' cannot be safely constructed.
  --> src/main.rs:19:26
   |
19 |         let gc = Gc::new(Wrapper(b.as_ref()));
   |                  --------^^^^^^^^^^^^^^^^^^^-
   |                  |       |
   |                  |       contains a reference (&) which may no longer
                             be valid when it is finalized.
   |                  'Gc::new' requires that a type is reference free.

warning: 'ref_soundness' (bin "ref_soundness") generated 1 warning
```

```
 1 struct Wrapper<'a>(&'a u64);
 2
 3 impl<'a> Drop for Wrapper<'a> {
 4     fn drop(&mut self) {
 5         println!("Dropping␣Wrapper");
 6     }
 7 }
 8
 9 fn main() {
10     {
11         let a = 123;
12         let gc = Gc::new(Wrapper(&a));
13     } // a is no longer in scope
14
15     // GC happens here, calling 'Wrapper::drop' as a finaliser.
16     // 'Wrapper''s drop method does not deref the box pointer,
17     // so it does not violate the properties we care about,
18     // but it does violate the borrow-or-finalise rule.
19 }
```

Listing 6.12: A `Gc` is created, storing a reference to `a` inside a `Wrapper` whose `drop` method gives it a finaliser. This does not compile, since it violates the borrow-or-finalise rule. However, the `drop` method for `Wrapper` (line 4) does not use the reference `&a` at all, which makes it perfectly safe.

To check whether a type passed to `Gc` contains a reference, ALLOY defines a marker trait with no methods called `ReferenceFree` which is used by the compiler to indicate that a value of a type does not contain references.

`ReferenceFree` is defined as an *auto trait* (see Section 5.2.5), so it is implemented on every type in Rust by default. ALLOY then explicitly unimplements `ReferenceFree` on all reference types:

```
impl !ReferenceFree for &T {}
impl !ReferenceFree for &mut T {}
```

This means that if a type contains a `&T` or a `&mut T` for any `T` in one of its component types, then it will not implement `ReferenceFree` either.

Whether or not a type implements `ReferenceFree` can then be checked at compile-time: if `T` implements `ReferenceFree`, then it is safe to use inside a `Gc`. If `T` does not implement `ReferenceFree`, and `T` has a `drop` method, then it cannot be used inside a `Gc`. This is implemented with an extension to finaliser safety analysis, where the type of `T` is checked for an implementation of `ReferenceFree` if it has a finaliser during the `CheckCallsite` phase.

Limitations

The borrow-or-finalise rule can be restrictive: in particular, a `Gc<T>` object which both contains references and has a finaliser, but where the finaliser never uses the reference would be deemed invalid, even though it is sound. Listing 6.12 shows an example program which is sound, but fails to compile due to the borrow-or-finalise rule.

To allow the program in Listing 6.12 to compile, one can explicitly implement `ReferenceFree` on `Wrapper`:

```
unsafe impl<'a> ReferenceFree for Wrapper<'a> {}
```

This will now compile and `Gc<Wrapper>` will be finalised before collection. However, it is unsafe as the onus is now on the programmer to ensure `Wrapper`'s `drop` method never uses `&a`.

6.6.2 Finalisation order

In Section 6.1.3 I explained how a GC can guarantee ordering on finalising floating garbage. The main disadvantage of this approach is that objects with finalisation cycles will not be finalised. The alternative is for the collector to not specify any order. This allows objects with cycles to be finalised but with a heavy constraint: they must not reference other objects from inside their finalisers. For many GCs, this is too restrictive. Boehm [2003] makes the case that in languages such as Java, the usefulness of finalisers depends on them being able to interact with other objects. As such, it is common to see VMs for managed languages such as Java and .NET specify an ordering for finalisers.

However, our requirements for GC are unique in this respect: ALLOY is not intended to replace Rust's RAII approach to memory management, instead, it provides optional GC for objects where the RAII approach is difficult or impossible. It is not uncommon to see Rust programs which use ALLOY with a mix of GC'd and non-GC'd objects. In such cases, it is safe for a finaliser to access a field of a non-GC'd object because there is no danger of them being finalised already. Listing 6.13 shows an example of how dereferencing another `Gc` can be unsound, with Figure 6.3 showing its representation in memory.

In addition, one of ALLOY's main goals is to make it easier to work with data structures that have cycles in Rust. It is suggested that finalisation cycles are rare in GC'd languages [Jones et al., 2016]. However, this is different for ALLOY, since destructors in Rust are common, and mapping them to finalisers means that it is not uncommon to see finalisation cycles in Rust programs using ALLOY.

```
 1 impl Drop for Node {
 2     fn drop(&mut self) {
 3         // println!("Dropping {}", self.a) // Unsound when ‘Node‘ used in Gc!
 4         println!("Dropping␣{}", self.b)
 5     }
 6 }
 7
 8 struct Node {
 9     a: Gc<u64>,
10     b: Box<u64>,
11 }
12
13 let gc1 = Gc::new(Node{ a: Gc::new(1), b: Box::new(2) });
```

Listing 6.13: Here, it is safe for `Node`'s finaliser to reference its field `a` (line 4) because it points to an non-GC'd object. In contrast, if line 3 was uncommented this program would be unsound. Unlike in languages such as Java, where every object is managed, the restriction of dereferencing through fields of other GC'd objects is less of an issue because non-GC'd objects can still be accessed.

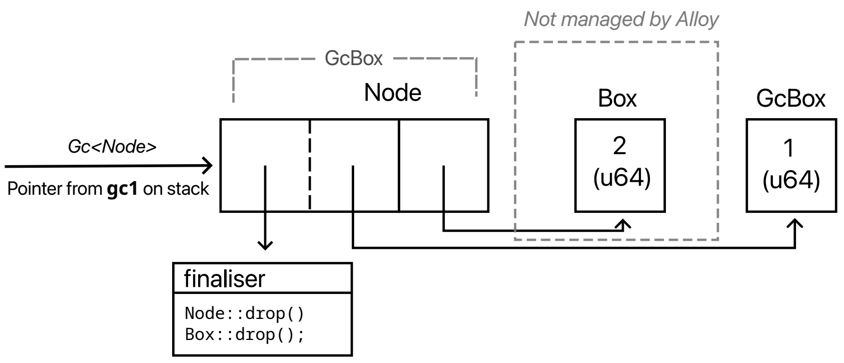


Figure 6.3: Listing 6.13 representation in memory.

For these reasons, I decided that by default, Alloy does not guarantee a finalisation order. This allows cyclic data managed by Alloy to be finalised, preventing potential resource leaks. In Section 6.6.2 I describe how this is made sound by preventing a `Gc`'s finaliser from dereferencing a field to another `Gc`.

Forcing ordered finalisation

If users find the constraint this imposes on finalisers too restrictive, they may instead choose to build Alloy with ordered finalisation. This can be done with the `-DENABLE_TOPOLOGICAL_FINALIZATION` build flag, which uses the BDWGC's topological finalisation order. In this configuration, cyclic data structures such as that shown in Figure 6.4 would not finalise, leading to a memory leak.

```
 1 use std::gc::Gc;
 2 use std::sync::Mutex;
 3
 4 struct Node {
 5     name: String,
 6     next: Option<Gc<Mutex<Node>>>,
 7 }
 8
 9 impl Drop for Node {
10     fn drop(&mut self) {
11         println!("Dropping␣{}", self.name);
12     }
13 }
14
15 fn main() {
16     let a = Gc::new(Mutex::new(Node {
17         name: String::from("a"),
18         next: None,
19     }));
20
21     let b = Gc::new(Mutex::new(Node {
22         name: String::from("b"),
23         next: None,
24     }));
25
26     a.lock().unwrap().next.insert(b);
27     b.lock().unwrap().next.insert(a);
28 }
```

Figure 6.4: An example of a small cyclic graph using Alloy. This example creates two nodes, `a` and `b`, before creating cyclic references between them (lines 26-27). If Alloy used ordered finalization, then neither `a` nor `b` would be finalised. This would cause a memory leak, because `Node.name` contains an owned `String`. A `String`'s `drop` method is used to deallocate its backing store, which never be called.

Alloy can therefore only represent this cyclic data structure without memory leaks if it runs finalisers without a specified ordering. This way, the collector can decide at its leisure whether to finalise `a` or `b` first.

While ordered finalisation would allow finalisers to dereference fields to other `Gc`'s, they must still only access fields which use the `FinalizerSafe` trait for the soundness reasons explained in Section 6.5.1.

Soundness issues with unordered finalisation

The problem with finalising `Gc`s in an unspecified order is that any other `Gc` object which they reference may have already been freed. Consider the example in Figure 6.5, which allocates `Gc`s which reference each other in a cycle. A cycle such as this one will always lead to unsoundess with finalisers which access other `Gc` objects. This is also a problem for

```rust
struct Node {
    name: String,
    next: Option<Gc<Mutex<Node>>>,
}

impl Drop for Node {
    fn drop(&mut self) {
        match self.next {
            Some(n) => {
                println!("The␣next␣node␣is␣{}", self.next.unwrap());
            },
            None => println!("No␣next␣node!"),
        }
        println!("Dropping␣{:?}", self.next);
    }
}

fn main() {
    let a = Gc::new(Mutex::new(Node {
        name: String::from("a"),
        next: None,
    }));

    let b = Gc::new(Mutex::new(Node {
        name: String::from("a"),
        next: None,
    }));

    a.lock().unwrap().next.insert(b);
}
```

Figure 6.5: An example of a potentially unsound Rust program. Two `Gc<Mutex<Node>>` objects (`a` and `b`) are created where `a` contains a reference to `b`. When these objects are garbage collected, Alloy will schedule their finalisers to run in a non-specified order. If `b` is finalised before `a`, then `a`'s drop method will access a dropped value.

non-cyclic data structures in Alloy's non-ordered configuration as there is no guarantee that non-cyclic data structures will be finalised 'outside-in'.

Making unordered finalisation sound

When Alloy is compiled with unordered finalisation, it prevents `drop` methods from dereferencing fields which point to other `Gc` objects. Listing 6.14 shows the error message that is displayed when the example in Figure 6.5 is compiled. It has identified that the user is trying to access field `b`, which contains a `Gc` type. I extend Alloy's *finaliser safety analysis* (described in Section 6.5.2) to detect this.

This is implemented by first adding a negative implementation of `FinalizerSafe` to the `Gc<T>` type:

```
error: ‘Mutex::new(Node {
               name: String::from("a"),
               next: None,
           })‘ cannot be safely finalized.
  --> src/main.rs:30:21
    |
15  |              Some(n) => {
    |                   -
    |                   |
    |                   caused by the expression here in ‘fn drop(&mut)‘ because
    |                   it uses another ‘Gc‘ type.
...
30  |        let b = Gc::new(Mutex::new(Node {
    |   _____________________^
31  | |          name: String::from("a"),
32  | |          next: None,
33  | |      }));
    | |______^ has a drop method which cannot be safely finalized.
    |
   = help: ‘Gc‘ finalizers are unordered, so this field may have already been dropped.
     It is not safe to dereference.
```

Listing 6.14: The compile-time error message shown when attempting to compile the example in Figure 6.5. FSA identifies that another `Gc` is being dereferenced inside a finaliser. This is unsound when Alloy does not use ordered finalisation as it may have already been collected.

```rust
impl<T> !FinalizerSafe for Gc<T> {}
```

FSA then uses this to identify the specific field which was unsafely dereferenced and will generates an error message different from those to do with thread-safety.

However, as explained in Section 6.5.2, FSA is not complete. It is possible that a `drop` method could dereference a `Gc` field in a way that FSA could not detect, e.g. by doing so behind an opaque function call. In such cases where the MIR for the entire `drop` method cannot be checked, FSA will err on the side of caution, favouring soundness by refusing to compile the program. Figure 6.6 shows an example of this in practice.

Unordered finalisation and the borrow-or-finalise rule

Alloy's `Gc<T>` type provides an `as_ref` function for obtaining a reference to the underlying type (`&T`). This allows the `Gc<T>` to be used in Rust code which does not care about how the value is stored, but instead just expects a Rust reference to it (`&T`). Unfortunately, this creates a backdoor where a reference to another `Gc` object could be used in a finaliser. In such a case, FSA can not tell that dereferencing a `&T` accesses memory in another `Gc` object, because the "`Gc`" part of the type has been omitted. Listing 6.15 shows a potential example of a raw reference can leak into a finaliser.

```rust
use std::gc::Gc;

extern "C" {
    // A printer function written in "C" that is externally linked.
    // The compiler cannot see the contents of this function.
    fn node_printer(value: &Node);
}

struct Node {
    name: String,
    next: Option<Gc<Node>>,
}

impl Drop for Node {
    fn drop(&mut self) {
        unsafe {
            node_printer(self);
        }
    }
}

fn main() {
    let node = Gc::new(Node {
        name: String::from("a"),
        next: Some(Gc::new(Node {
            name: String::from("b"),
            next: None,
        })),
    });
}
```

Figure 6.6: An example program which is forbidden in Alloy. The `drop` method for `Node` calls `node_printer` on its value. This function is written in C and is externally linked, so the Rust compiler is unable to view its contents. Because of this, it cannot know whether or not `node_printer` accesses the `next` field. Alloy thus conservatively rejects the program. If the user is certain that `node_printer` does not access the `next` field, they can inform Alloy by implementing the `FinalizerSafe` trait on `Node`.

Fortunately, this problem is solved by the borrow-or-finalise rule (Section 6.6.1), and the example above would not compile with the error showing in Listing 6.16

6.7 Early finalisation

A fundamental assumption in Rust's destructor semantics is that dropping a value is the last thing to happen to it. The Rust compiler prevents using a value after it has been dropped as this would cause unsoundness. For `Gc` values in Alloy, the same must be true for finalisers. If a finaliser is able to run before the mutator has finished using it, this would also be unsound.

```rust
use std::gc::Gc;
use std::fmt::Debug;

struct Wrapper<T: Debug>(T);

impl<T: Debug> Drop for Wrapper<T> {
    fn drop(&mut self) {
        println!("Dropping␣{:?}", self.0);
    }
}

fn main() {
    let gc1 = Gc::new(123);
    let r1 = gc1.as_ref();

    // Pass a rust reference to ‘Wrapper‘, hiding the fact that it points to
    // another GC object, which would be unsound to deref in the finaliser.
    let gc2 = Gc::new(Wrapper(r1));
}
```

Listing 6.15: A reference to the `Gc` object on line 13 is passed as a Rust reference to another `Gc` (line 18). The `drop` method for `Wrapper` dereferences this, which is unsound because if the finalisation order is unspecified.

```
error: ‘Wrapper(r1)‘ cannot be safely constructed.
  --> src/main.rs:18:23
   |
18 |     let gc2 = Gc::new(Wrapper(r1));
   |               --------^^^^^^^^^^^-
   |               |       |
   |               |       contains a reference (&) which may no longer be
   |               |       valid when it is finalized.
   |               ‘Gc::new‘ requires that a type is reference free.

error: could not compile ‘bin‘ due to previous error; 1 warning emitted
```

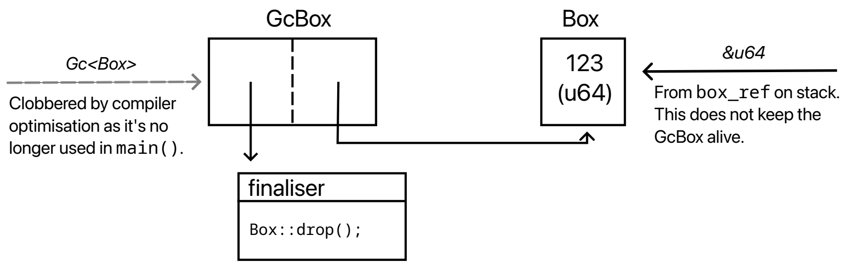
Listing 6.16: Compiler error for Listing 6.15 when run with ALLOY.

In Section 6.1.5, I explain how finalisers can run earlier than expected because compiler optimisations – unaware of the presence of a GC – can cause GC objects to become unreachable earlier than expected. If this is paired with an unfortunately timed GC cycle, the object's finaliser could run while the object is still in use by the mutator. Rust and ALLOY is no different: the Rust compiler is allowed to perform any optimisation that does not change the observable behaviour of the program, and such optimisations are not aware of the retro-fitted collector.

A finaliser which runs early can cause finalisation code to interleave unexpectedly with the mutator [Boehm, 2003]. But, even worse, in ALLOY early finalisation can even lead to a memory safety violation. Consider the following example, which shows how a finaliser which runs early could cause a use-after-free:

```
1  fn main() {
2      let a = Gc::new(Box::new(123));
3      let box_ptr: &usize = &**a;
4
5      ...
6
7      // Gc happens here causing
8      // a’s finaliser to drop the box.
9
10     println!("{}", box_ptr); // Potential use-after-free!
11 }
```

Assuming that the compiler clobbers the reference the `Gc` stored in variable `a`, this program can be represented as follows:


GcBox
Box
Gc<Box>
Clobbered by compiler optimisation as it's no longer used in main().
123 (u64)
&u64
From box_ref on stack. This does not keep the GcBox alive.
finaliser
Box::drop();

In this program, semantically, both `a` and `box_ptr` live until the end of `main`. However, the compiler may decide to reuse the register holding the reference at `a` any time after line 3 as it is no longer used. An unfortunately timed GC cycle which happens immediately afterwards would consider the `Gc` object unreachable. Its finaliser will then be run, freeing the Box. This would happen even though there is still a reference (`box_ptr`) to the inner `Box` value. This reference is now a dangling reference, and its use on line 10 would constitute a use-after-free.

The possibility of early finalisation has led many VMs to specify that finalisers can happen at any time – even earlier than when an object becomes unreachable (see Section 6.1.5). One way of preventing early finalisation in ALLOY would be to prevent `Gc` objects from owning non garbage collected objects, but this would render ALLOY almost unusable. Fortunately, we can do better.

6.7.1 Early finaliser prevention in Alloy

To understand how early finaliser prevention works in Alloy, lets revisit the example from the previous section:

```
1  fn main() {
2      let a = Gc::new(Box::new(123));
3      let box_ptr: &usize = &**a;
4
5      ...
6
7      // Gc happens here causing
8      // a's finaliser to drop the box.
9
10     println!("{}", box_ptr); // Potential use-after-free!
11 }
```

The problem is that the normal Rust compiler is not aware that a conservative GC exists, and that line 8 is dependent on the reference at `a` still being reachable.

To fix this, Alloy needs to ensure that the reference at `a` exists for the entire duration of `main`. The basis of a solution is to realise that inserting a *compiler barrier* – which prevents reads and writes of specified variables from being reordered before/after the barrier – at the end of `main`, followed by an artificial read of `a` keeps references around sufficiently long.

In other words, Alloy wants to convert the example above to look roughly as follows:

```
fn main() {
    let a = Gc::new(Box::new(123));
    ...

    COMPILER_BARRIER(a)
    *a;
}
```

Using drop to insert Compiler barriers

Alloy takes advantage of two observations: Rust already inserts calls to `drop` at the same point in a function where we want to insert compiler barriers; and we only need to insert barriers for variables of type `Gc`. However, since `Gc` is a `Copy` type, Rust prevents us from adding a `drop` method to `Gc`.

Fortunately, since Alloy already alters the Rust compiler, it is easy to add a further modification. I thus modify the Rust compiler to allow for simultaneous implementation of `Copy` and `Drop` for `Gc` types only, with the following drop implementation:

```
impl<T: ?Sized> Drop for Gc<T> {
    fn drop(&mut self) {
        unsafe {
            COMPILER_BARRIER(self)
        }
    }
}
```

The `COMPILER_BARRIER(a)` includes inline assembly using Rust's `asm!` macro to create a read of the `Gc`'s `self` reference after a compiler barrier. This is platform specific: for x86 it translates to the following:

```
asm("":::"memory")
```

Although the compiler barrier does not contain platform instructions, its format is still platform dependent: other platforms such as AArch64 may require a slightly different `asm` statement.

However, the compiler barrier by definition prevents the compiler from performing some of its normal optimisations — it is an expensive solution to a rare problem. In our performance analysis, this had roughly a 2-3% slowdown. In Section 6.9 I describe how I optimise this approach, removing barriers where it can be statically determined that they are unnecessary.

6.8 Optimising finalisers

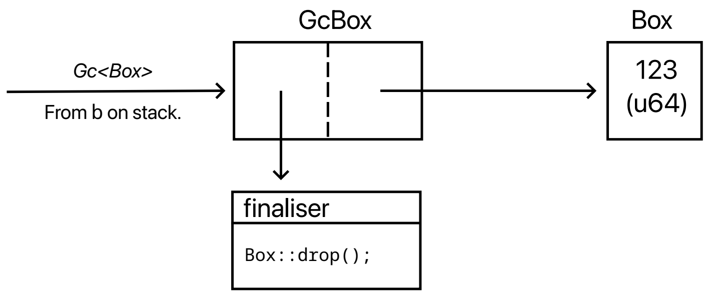
6.8.1 Finaliser elision

For performance reasons, many GC'd languages discourage programmers from using finalisers. In Rust, since `drop` is ubiquitous, mapping `drop` methods to finalisers therefore has a high performance overhead.

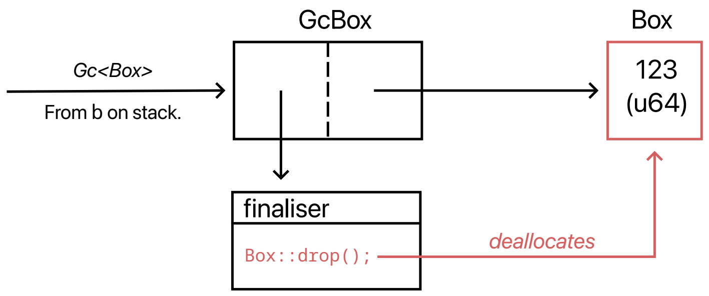
To claw back this performance hit, I implement a new optimisation called *finaliser elision*. This optimisation is based on the observation that sometimes only the top-most object in a `Gc` graph needs to have a finaliser. Consider the following example:

```
let a = Box::new(123);
let b = Gc::new(a);
```

Here `a` refers to a boxed integer on the heap which is not managed by the collector. It is placed inside a `Gc`, which has the following representation in memory:



Before the `Gc` referenced by `b` is collected, its finaliser is called, which calls `drop` on the non-GC'd box:

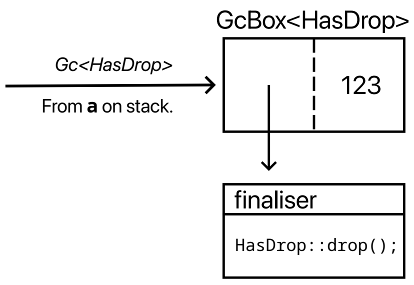


This is inefficient because the collector would have later reclaimed the `Box`, since all references to it would be lost. In other words, neither `a`'s finaliser nor `b`'s `drop` method need to be called for the memory to be reclaimed.

Since sweeping is cheaper than finalisation, finaliser elision aims to identify other cases where finalisation can be avoided. It is a powerful optimisation, but not applicable everywhere as this example shows:

```rust
struct HasDrop(u64);

impl Drop for HasDrop {
    fn drop(&mut self) {
        println!("Dropping␣HasDrop");
    }
}

let a = Gc::new(HasDrop(123));
```

Here `a` refers to a `Gc` containing an integer, which has the following representation in memory:



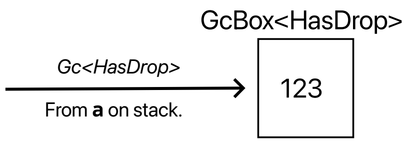
We are not able to elide the finaliser for this `Gc` because, unlike the previous example, the drop method does more than just drop another heap object in this case printing to stdout.

When can a finaliser be elided?

The foundation of finaliser elision lies with the `FinalizerOptional` trait, which is used to determine if a type needs finalising. If a type `T` implements `Drop`, but also implements `FinalizerOptional`, then `Gc<T>` will not be finalised. For example we can adjust our earlier example as follows:

```rust
struct HasDrop;

impl Drop for HasDrop {
    fn drop(&mut self) {
        println!("Dropping␣HasDrop");
    }
}

unsafe impl FinalizerOptional for HasDrop {}

let a = Gc::new(HasDrop(123));
```

This informs Alloy that `HasDrop` does not need a finaliser when placed inside a `Gc`, even though `HasDrop` implements the `Drop` trait. When a `Gc` is constructed with a value of this type, it no longer has a finaliser:



`FinalizerOptional` is particularly powerful when used with container types. For example, consider the standard Rust `Box<T>` type. Its heap memory can be automatically reclaimed by the allocator, but depending on the type `T`, we may need to call a finaliser. Thus we cannot simply always remove a `Box`'s finaliser. Fortunately we can easily tell Rust's type system that we want to make `Box<T>` be `FinalizerOptional` if `T` is also `FinalizerOptional` with:

```rust
unsafe impl<T> FinalizerOptional for Box<T> {}
```

This tells ALLOY that it is safe to elide the finaliser for `Box<T>` if `T` does not need finalising. When ALLOY discovers a type which implements `FinalizerOptional`, it will treat it as if it does not implement `Drop`, but continue on checking all its component types.

ALLOY implements `FinalizerOptional` on the following heap allocating standard library types: `Box<T>`, `Vec<T>`, `RawVec<T>`, `HashMap<T>`. This is enough to elide a significant amount of finalisers without any extra effort required by the user (see Section 7.4). If the user defines their own heap allocating types which use drop to deallocate, they can implement `FinalizerOptional` on it so that it can also benefit from finaliser elision.

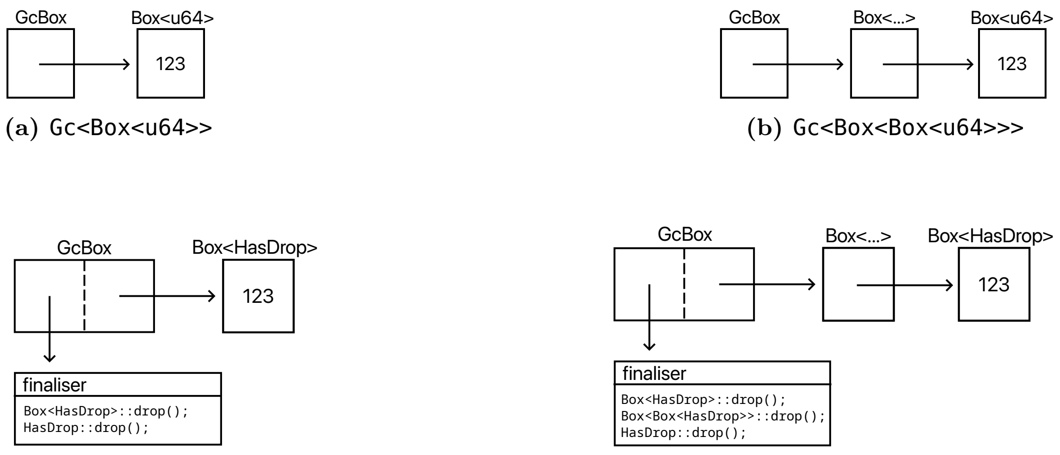
Finaliser elision algorithm

The exact algorithm for finaliser elision detection is defined as follows:

```
function NEEDSFINALISER(type)
    if IMPLS(type, Drop) and not IMPLS(type, FinalizerOptional) then
        return true
    for each component $\in$ type do
        if NEEDSFINALISER(component) then
            return true
    return false
```

Figure 6.7 shows various examples of when finaliser elision can remove a finaliser, and when it cannot. What is important to note here is that if a type `T` is marked `FinalizerOptional`, but has fields which need finalising, then `T` will still be finalised.

The algorithm for finaliser elision is implemented as a compiler intrinsic, `needs_finalizer<T>()` which returns true if `Gc<T>` needs a finaliser. This intrinsic is then called during the construction of new `Gc` objects (inside `Gc::new`):



(a) `Gc<Box<u64>>` **(b)** `Gc<Box<Box<u64>>>` **(c)** `Gc<Box<HasDrop>>` **(d)** `Gc<Box<Box<HasDrop>>>`

Figure 6.7: The memory layout of various `Gc` types. **(a)** and **(b)** do not need a finaliser because the `Box` implements `FinalizerOptional` and `u64` does not need finalising. **(c)** and **(d)** do need a finaliser because in each example, a `Box` contains a `HasDrop`, which needs finalising because it has a `drop` method. Notice that for **(c)** and **(d)**, the finaliser calls the `drop` methods for each component type – even the boxes which are marked `FinalizerOptional`. This is because finaliser elision does not do partial elision of finalisers.

```
1  impl Gc<T> {
2      pub fn new(value: T) -> Self {
3          ...
4          if needs_finalizer::<T>() {
5              Gc<T>::new_with_finalizer(value)
6          } else {
7              Gc<T>::new_without_finalizer(value)
8          }
9          ...
10     }
11 }
```

`needs_finalizer<T>` is marked as a special type of Rust function, called a *const* function, which means that it is evaluated at compile time, this means that the conditional on line 4 has no runtime cost.

Sweeping objects with elided finalisers

Once an object's finaliser has been elided, Alloy needs to be responsible for managing its memory. This is achieved with a modification to the global allocator in Rust. Every heap allocation in Rust programs compiled with Alloy is allocated using BDWGC's `GC_malloc` function — even those which are not part of a `Gc`. For values which are allocated using Rust's RAII approach, they are manually deallocated with calls to `GC_free`. For non-GC'd heap allocations, this is semantically equivalent to a regular `malloc` call: there will always be a reference preventing them from being collected, and RAII ensures that the corresponding `GC_free` is called the moment they go out of scope. However, when such allocations are owned by a `Gc`, they will be freed by the collector along with the `Gc`.

6.9 Optimising early finaliser prevention

Early finalisation prevention (Section 6.7.1) overapproximates the places where early finalisation can happen, which can have a significant impact on performance. Fortunately, the finaliser elision optimisation in Section 6.8.1 shows that many finalisers never need to be called, at which point we also no longer have to worry early finalisation! Where this is the case, we are able to remove the `drop` method for the `Gc<T>` pointers which contain compiler barriers during compilation.

All `Gc` values have `drop` methods with barriers by default. During compilation, barriers which we can prove are unnecessary are removed. This is done once the Rust compiler has generated its mid-level IR (MIR). Like finaliser safety analysis (Section 6.5.2), we perform an in-order traversal on the control flow graph represented by the MIR for each function with the following algorithm:

```
function BarrierRemoval($callsite$)
    for each $mir\_body \in prog$ do
        for each $basic\_blocks \in mir\_body$ do
            for each $block \in basic\_blocks$ do
                if CallsDrop($block.terminator$) then
                    $arg \leftarrow$ GetFirstArg($block.terminator$)
                    $arg\_ty \leftarrow$ GetType($arg$)
                    if IsGC($arg\_ty$) then
                        if not NeedsFinaliser($arg\_ty$) then
                            RemoveDrop($projection$)
```

This iterates over all `drop` methods in the entire program, identifying those which belong to a `Gc<T>`. If found, the `drop` call is removed if the Gc reference points to an object which does not need finalising. The `drop` method is removed by patching the terminator of the block which calls `drop` with the terminator at the end of the `drop` body:

```
function RemoveDrop($block$)
    $drop\_mir \leftarrow$ GetDropMirBody($block.terminator$)
    $last\_block \leftarrow$ GetLastBlock($drop\_mir$)
    $block.terminator \leftarrow last\_block.terminator$
```

After this pass, we call the Rust compiler's existing *simplify mir* pass, which tidies up the control flow graph by removing the empty blocks which were created as a result of `drop` removal.

Chapter 7

Alloy performance evaluation

This chapter examines the performance characteristics of Alloy. To do this, I present five performance studies:

1. An evaluation of the wall-clock time and memory usage of Alloy as a candidate for use in a single-threaded VM. This compares Alloy with Rust's single-threaded reference counting library on the SOM benchmark suite [Marr et al., 2016] when used to manage objects in the somrs VM (Section 7.2.1).
2. An evaluation of the wall-clock time and memory usage of Alloy as a candidate for use in a multi-threaded VM. This compares Alloy with Rust's atomic reference counting library using the WLambda VM (Section 7.2.2).
3. An evaluation of the effectiveness of the finaliser elision optimisation. This uses two configurations of the yksom VM – with and without finaliser elision – to measure its performance impact using the SOM benchmark suite (Section 7.3).
4. An evaluation of the slowdown introduced by the early finaliser prevention mechanism in Alloy (Section 7.4).
5. A comparison between Alloy and other GC implementations in Rust on the Binary Trees benchmark (Section 7.5).

7.1 Generic benchmarking methodology

The four performance studies share a common benchmarking methodology. All experiments were run on a Xeon E3-1240v6 3.7GHz with 4 CPU cores. Each core has an L1 and L2 cache size of 256KB and 1MB respectively. The 8MB L3 cache is shared between cores. The benchmarking machine has 32GB of RAM and runs Debian 8. I disabled turbo boost and hyper-threading in the BIOS and set Linux's CPU governor to *performance* mode. I observed the `dmesg` after each experiment and did not observe any oddities such as the machine overheating.

For Alloy I use an initial heapsize of 64KB and the heap is always expanded when the collector encounters insufficient space for an allocation. A collection is triggered whenever more than `heap_size / 6` bytes of allocation have taken place. This is the default behaviour in the BDWGC.

7.2 Alloy for language implementation

One of Alloy's main goals is to be usable for implementing other languages VMs. In order to evaluate the effectiveness of this, this section aims to answer the following two questions:

1. How much slower is a language VM written in Rust when using Alloy compared to Rust's reference counting library?
2. How much memory does a language VM written in Rust when using Alloy use when compared to Rust's reference counting library?

To answer these questions I conduct two studies, where I retrofit Alloy to two separate language VMs:

somrs A VM written in Rust for the SOM language (a cut down Smalltalk language) which uses Rust's single threaded reference counting library (`Rc`) to manage SOM objects.

WLambda A VM written in Rust for the WLambda language: a multi-threaded JavaScript-like scripting language which uses Rust's atomic reference counting library (`Arc`) to manage SOM objects.

This allows a performance comparison between VMs which use Alloy and both forms of reference counting in otherwise identical language implementations. The following subsections outline the methodology and results of each experiment.

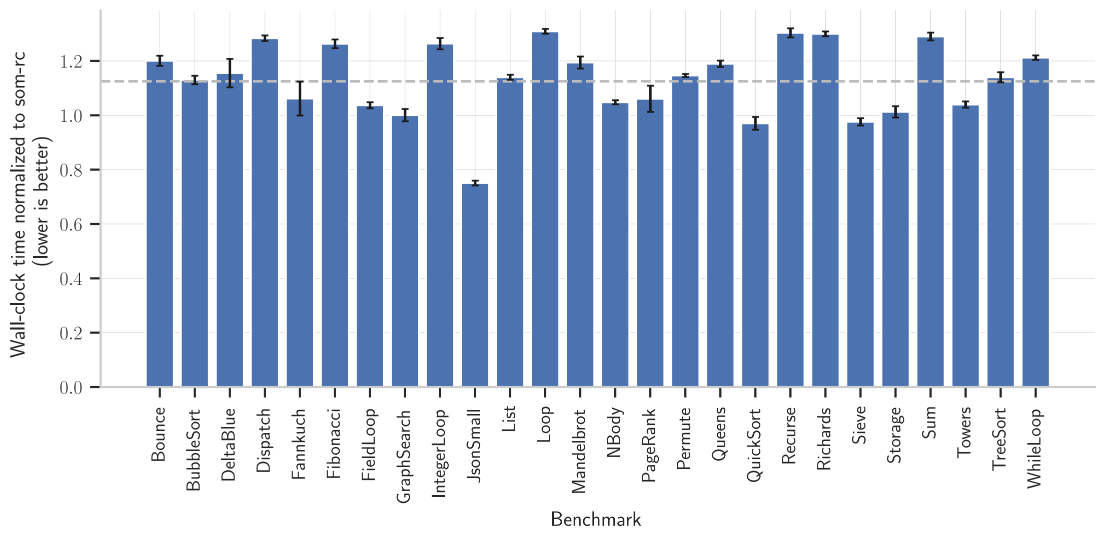


Figure 7.1: Results from the PERF experiment, where the SOM benchmark suite is run with both configurations: SOMRS-RC, and SOMRS-GC. The vertical bars represent the wall-clock time of SOMRS-GC, normalised to the wall-clock time of SOMRS-RC. The dashed grey line shows the geometric mean of the slowdown of SOMRS-GC across all benchmarks (1.12×). Each benchmark is run for 30 process executions, where the error bars represent 99% confidence intervals.

7.2.1 First study: the SOMRS VM

Methodology

The first study is a performance and memory comparison using SOMRS [Polomack, 2020] – a Rust implementation for the SOM language (a cut down version of Smalltalk). SOMRS does not support Just-in-Time (JIT) compilation and is a medium sized interpreter with roughly 10KLoC. I build two configurations of SOMRS as follows:

SOMRS-RC the standard implementation, where SOM objects are managed using `Rc` (Rust's single-threaded reference counting mechanism).

SOMRS-GC where SOMRS is modified so that SOM objects are managed using Alloy. This required changing 62 lines of code, most of which was replacing instances of `Rc` with Alloy's `Gc`. Both configurations are otherwise identical.

I use the SOM language's existing suite of synthetic benchmarks, adapted from the *are-we-fast-yet* benchmark suite [Marr et al., 2016]. The benchmarks are written using idiomatic SOM code.

The study is divided into two sub-experiments which measure performance and memory separately:

PERF In the performance experiment, I compare the performance between SOMRS-RC, and SOMRS-GC using the Rebench benchmarking suite [Marr, 2018]. Each SOM benchmark is run for 30 *process executions*, where each VM is closed down and started again. Rebench randomises the order in which each VM implementation is run after each process execution and records wall-clock times using Python's `time()` function.

MEM In the memory experiment, memory usage from both configurations is measured by running each benchmark individually while sampling the process's *resident set size* every 10 milliseconds as this was the smallest value where there were noticable differences between samples. From this, I present the peak and average real (i.e. physical, not virtual) memory usage for each benchmark as well as plotting the usage over time.

Results

Figure 7.1 shows the performance results for the PERF experiment. On average, SOMRS-GC performs 12% worse, when taking the geometric mean across all benchmarks. The worst performing benchmarks are *Loop*, *Recurse*, and *Richards* which each have a slowdown of 1.24×. One possible explanation for this is that for these benchmarks in particular, many of the allocated objects have a lifetime which extends for the duration of the benchmark. This means that frequent GC pauses by ALLOY free little memory but end up increasing the total wall-clock time of the benchmark. For example, during the Richards benchmark, a GC collection cycle was scheduled an average of 24,800 times for each iteration.

Table 7.1 shows the memory usage results from the MEM experiment. For most benchmarks, SOMRS-GC consumes slightly more memory. Though ALLOY will use additional memory for its internal data structures for bookkeeping and during collection, this does not sufficiently account for the additional memory used by SOMRS-GC. It is more likely that objects in SOMRS-GC are being retained for longer than in SOMRS-RC. It is not clear why this is the case, but one possible reason is that SOMRS uses a third party hash map implementation internally for locals and method storage, and once references are removed, they may not be properly zeroed, causing stale pointers to keep objects alive. Figure 7.2 highlights two cases in particular, *PageRank* and *GraphSearch*, where memory usage in SOMRS-GC is unusually high. The graphs showing memory usage over time for all benchmarks can be found in Appendix A.

However, ALLOY's advantage of being able to cope with objects with cyclic references is shown clearly in Figure 7.3. *DeltaBlue* and *JsonSmall* when run with SOMRS-RC leak memory because these benchmarks contain objects with cycles. For these benchmarks,

Benchmark	Peak memory usage (MB)		Average memory usage (MB)	
	Reference Counting	Alloy	Reference Counting	Alloy
Bounce	3.94	4.50	3.94	4.50
BubbleSort	3.94	4.50	3.94	4.50
DeltaBlue	22.88	11.81	13.76	11.07
Dispatch	3.75	4.12	3.75	4.12
Fannkuch	3.94	4.50	3.94	4.46
Fibonacci	3.94	4.12	3.94	4.12
FieldLoop	3.75	4.12	3.75	4.12
GraphSearch	15.64	50.18	14.86	45.16
IntegerLoop	3.94	4.12	3.94	4.12
JsonSmall	520.12	8.25	261.40	8.17
List	3.94	4.50	3.94	4.50
Loop	3.75	4.31	3.75	4.31
Mandelbrot	3.75	4.31	3.75	4.31
NBody	4.12	4.50	4.12	4.50
PageRank	5.68	35.31	5.68	29.08
Permute	3.94	4.31	3.94	4.31
Queens	3.94	4.31	3.94	4.31
QuickSort	4.12	7.69	4.12	7.61
Recurse	3.94	4.31	3.94	4.31
Richards	4.31	4.69	4.14	4.60
Sieve	4.12	6.80	4.12	6.76
Storage	5.25	9.19	5.21	9.01
Sum	3.94	4.12	3.94	4.12
Towers	3.94	4.50	3.94	4.50
TreeSort	4.31	5.25	4.30	5.23
WhileLoop	3.94	4.12	3.94	4.12

Table 7.1: Peak and average memory usage (in megabytes) of each SOM benchmark for both som-rs configurations.

somrs-rc leaks memory so severely that they would cause cause out-of-memory errors if ran for just a few minutes, while somrs-gc maintains a steady heap profile over time.

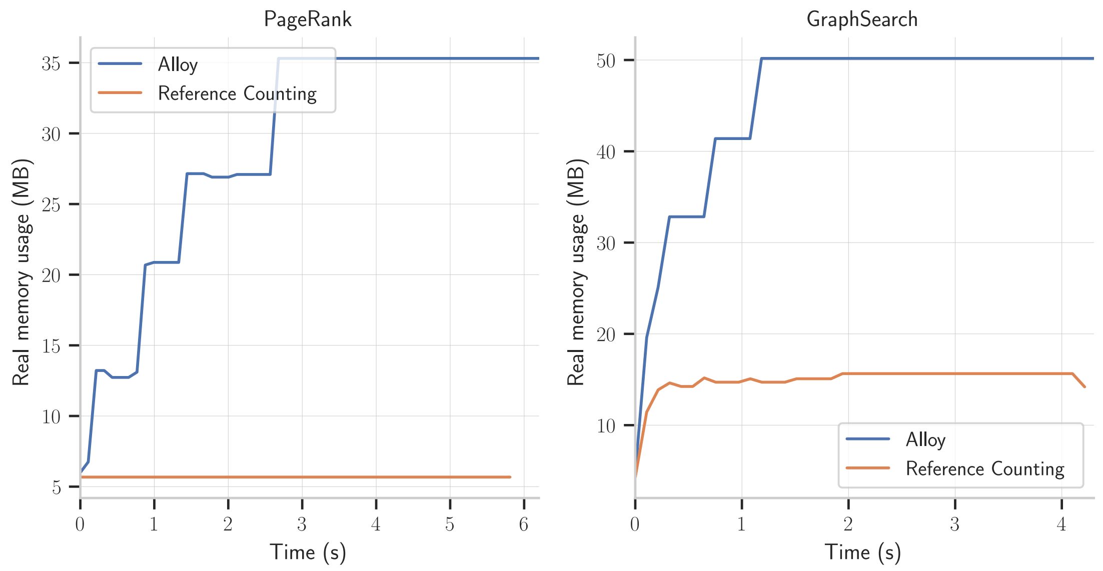


Figure 7.2: Real memory usage of the PageRank and GraphSearch benchmarks over time. In both of these benchmarks, SOMRS-GC uses significantly more memory. While this suggests there is a memory leak somewhere in SOMRS-GC, it is not yet clear why. One likely explanation which I have come across before during development is that during allocation heavy tight loops, stale references to heap objects may still exist in not-yet-overwritten stack slots which are then traced conservatively during marking.

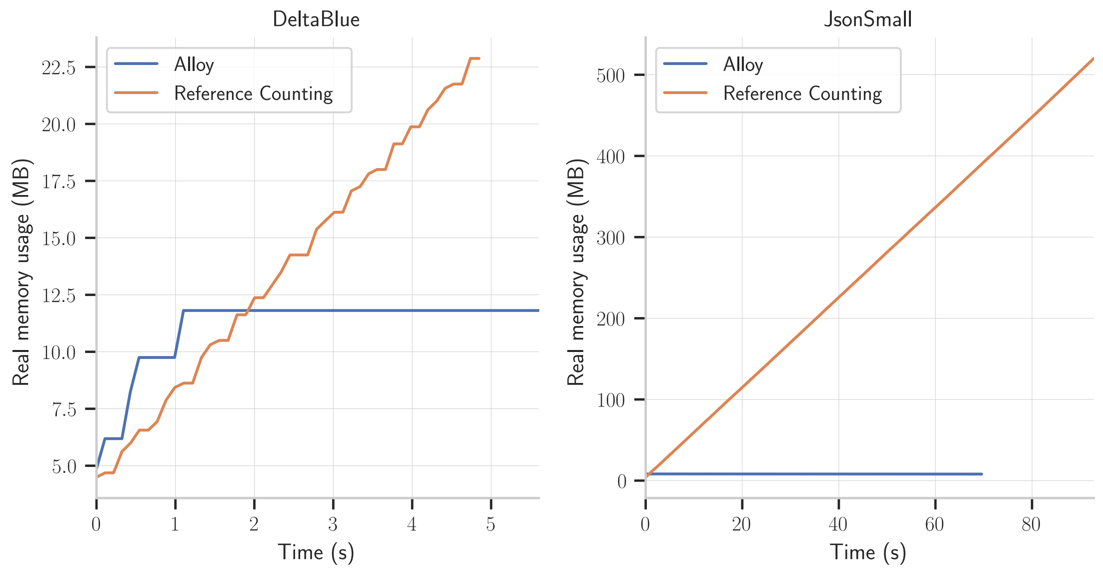


Figure 7.3: Real memory usage of the DeltaBlue and JsonSmall benchmarks over time. Both of these benchmarks allocate objects containing reference cycles, which SOMRS-RC is unable to free due to its use of reference counting. The resulting leak causes SOMRS-RC's real memory usage to grow continously over time. SOMRS-GC does not have this problem because Alloy is a tracing garbage collector.

7.2.2 Second study: the WLambda VM

The second study is a performance and memory comparison using WLambda [WeirdConstructor, 2019] – a Rust VM for an embeddable perl-like scripting language. WLambda is a medium-sized VM with approximately 30KLoC. It supports prototype based object orientation, and most WLambda objects are built upon vector and map data structures. Unlike somrs, WLambda is multi-threaded, so it allows for a comparison of a VM using Alloy and atomic reference counting. I build two configurations of WLambda as follows:

WLambda-arc the standard implementation, where VM objects are managed using `Arc` (Rust's atomic reference counting mechanism).

WLambda-alloy where WLambda is modified so that VM objects are managed using Alloy. This required changing roughly 700 lines of code, most of which was replacing instances of `Rc` with Alloy's `Gc`. WLambda makes use of Rust's trait-based dynamic dispatch, so each trait was manually audited to see whether it could be marked `FinalizerSafe` to satisfy FSA (Section 6.5.2), or `FinalizerOptional` to use finaliser elision (Section 6.8.1). Both configurations are otherwise identical.

WLambda comes with four benchmarks which I use to test each configuration: *Pattern Matching*, which stresses comparisons performed on objects; *Fibonacci*, which stresses function call overhead and recursion; *Vec Iter*, which creates and manipulates large lists and maps; and *Box*, which allocates many objects and stresses read and write operations.

Like the first somrs study, this study is also divided into two sub-experiments to measure performance and memory separately:

perf In the performance experiment, I compare the performance between WLambda-arc, and WLambda-alloy using the *multitime* tool: a Unix *time* utility extension which allows commands to be run multiple times for comparison [Tratt, 2014]. I run benchmarks for each configuration for 30 process executions. multitime randomises the order in which each benchmark and VM configuration is run, and records wall-clock time. I report the mean wall-clock time of each benchmark with 99% confidence intervals.

mem Similar to the yksom experiment, the memory usage is of both configurations is measured by running each benchmark individually while sampling the process's *resident set size* every 10 milliseconds as this was the smallest value where there were noticable differences between samples.

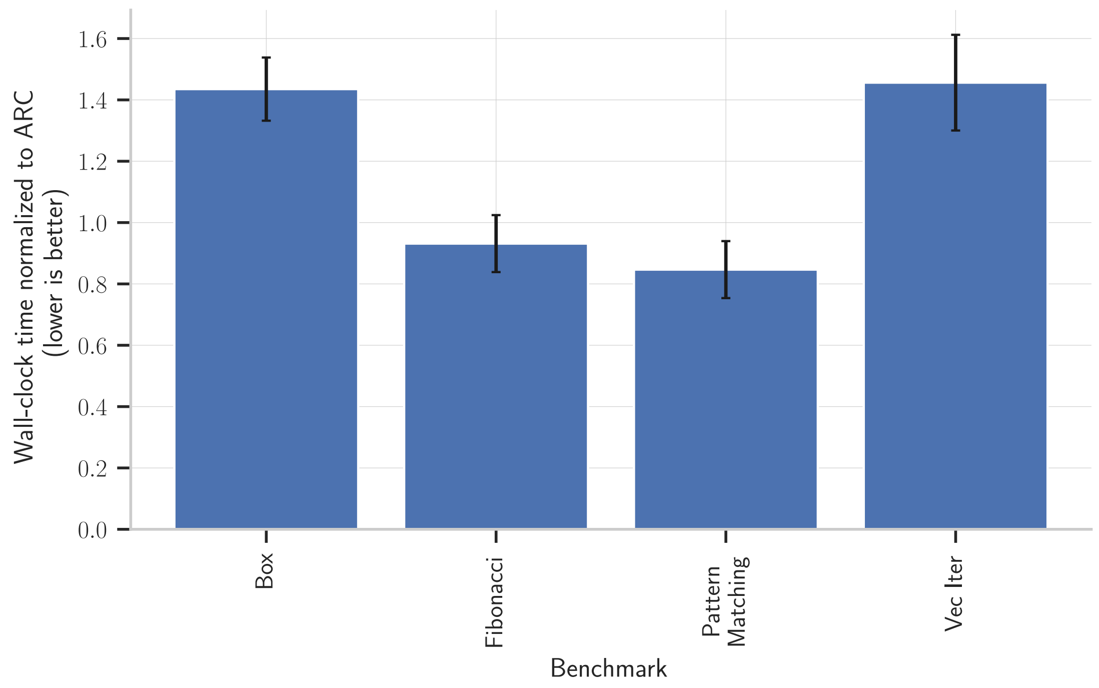


Figure 7.4: Results from the perf experiment, where the WLambda benchmarks are run with both configurations: WLambda-arc, and WLambda-alloy. The vertical bars represent the wall-clock time of WLambda-alloy, normalised to the wall-clock time of WLambda-arc. Each benchmark is run for 30 process executions, where the error bars represent 99% confidence intervals.

Results

Figure 7.4 shows the performance results for the perf experiment. The range in performance on this suite is large, with WLambda-alloy performing 18% better on *Pattern Matching* but 45% slower on *Box*.

Table 7.2 shows the memory usage results from the mem experiment while Figure 7.5 shows the memory usage of each benchmark over time. In all benchmarks, WLambda-alloy uses more memory. This is particularly apparent in the *Box* benchmark, where on average WLambda-alloy requires 4MB of additional memory suggesting that WLambda-alloy has a prominent memory leak for this benchmark. In the perf experiment, this is also the worst performing benchmark. *Box* allocates a total of 450,027,871 objects which results in 30% of the runtime spent in marking when profiling the benchmark in perf.

7.3 Measuring finaliser overhead

In my third study, I evaluate the effectiveness of Alloy's finaliser elision with a straightforward comparison between Alloy with and without finaliser elision. I use yksom: a

Benchmark	Peak memory usage (MB)		Average memory usage (MB)	
	Atomic RC	Alloy	Atomic RC	Alloy
Pattern Matching	10.94	11.49	10.94	11.49
Fibonacci	11.12	11.68	11.12	11.68
Vec Iter	10.94	11.68	10.94	11.65
Box	10.93	15.44	10.93	14.93

Table 7.2: Peak and average memory usage (in megabytes) of each WLambda benchmark for both configurations.

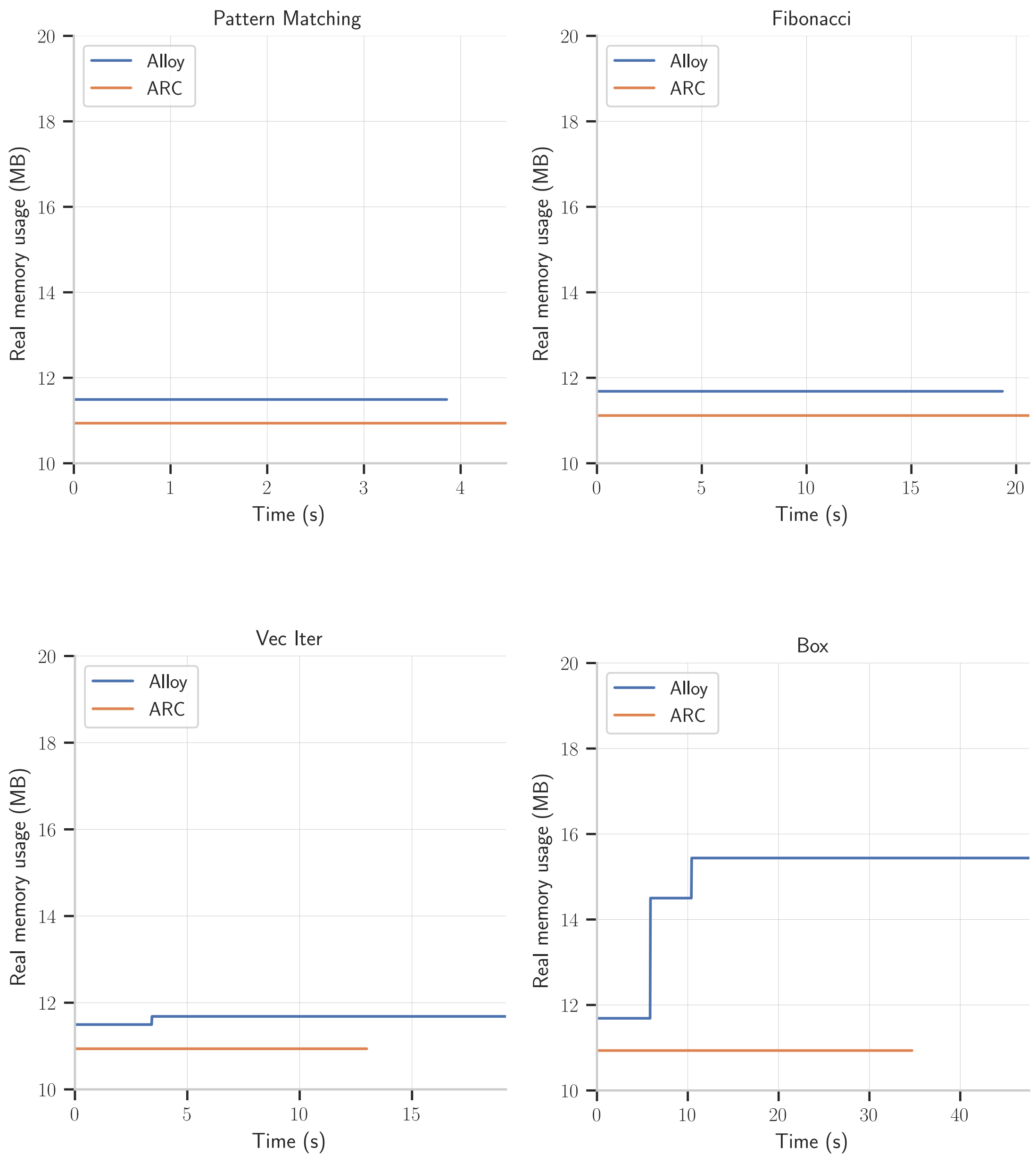


Figure 7.5: Real memory usage of the WLambda benchmarks over time.

another SOM interpreter written in Rust which uses Alloy to manage SOM objects. yksom is a lightweight interpreter with approximately 5700KLoC. It is suitable for measuring the impact of finaliser overhead because it makes frequent use of `drop` methods inside the VM. I build two configurations of yksom as follows:

YKSOM$_{\text{FinaliserElision}}$ The standard Alloy build which enables finaliser elision optimisation (Section 6.8.1).

YKSOM$_{\text{Naive}}$ An otherwise identical build but with Alloy's finaliser elision optimisation disabled.

I use the SOM benchmark suite using Rebench benchmarking suite using two versions of yksom for each Alloy variant. I run each SOM benchmark for 30 *process executions*, where each instance of yksom is closed down and started again. Rebench records wall-clock times using Python's `time()` function. I also run each SOM benchmark and record the number of finalisers that were executed for each Alloy variant.

7.3.1 Results

Table 7.3 shows the performance results for this study. Finaliser elision is able to remove over 99% of finalisers on every benchmark, leading to a 14.87% improvement in runtime speed when taking the geometric mean across all benchmarks. Such an improvement is clear in part due to yksom's internal VM object representation: yksom objects use Strings and HashMaps internally, which are allocated on the Rust heap and thus, without finaliser elision, require tens of thousands of calls to drop in order to deallocate.

However, despite significant improvements in wall-clock time on nearly every benchmark, *FieldLoop* regresses by 10.59%. The reason for this became clear once I recorded the peak memory usage of YKSOM$_{\text{Naive}}$ (3.25MB) and YKSOM$_{\text{FinaliserElision}}$ (4.31MB). The finalisers for 606,385 objects are elided, meaning that those objects become the responsibility of Alloy to GC. Therefore, based on the higher memory usage, the most likely reason for the resulting slowdown is that those objects are being erroneously kept alive. After analysing the program in perf, almost all the excess runtime is spent in marking.

Benchmark	YKSOM$_{\text{Naive}}$		YKSOM$_{\text{FinaliserElision}}$			
	Runtime (ms)	Finaliser count	Runtime (ms)	Finaliser count	Runtime slowdown	Finalisers elided
Bounce	224.329 ± 0.2697	1770542	178.654 ± 0.2646	853	-20.36%	99.95%
BubbleSort	95.525 ± 0.1243	795517	79.956 ± 0.1423	894	-16.30%	99.89%
DeltaBlue	1609.086 ± 18.9061	12381013	1369.114 ± 12.5874	1817	-14.91%	99.99%
Dispatch	81.540 ± 0.0874	601406	67.245 ± 0.0722	762	-17.53%	99.87%
Fannkuch	13.350 ± 0.0836	109721	11.363 ± 0.0884	821	-14.88%	99.25%
Fibonacci	80.245 ± 0.1068	876916	63.837 ± 0.0896	767	-20.45%	99.91%
FieldLoop	205.160 ± 0.1370	601385	226.884 ± 0.1300	791	10.59%	99.87%
GraphSearch	1077.769 ± 11.8820	7170437	1024.444 ± 18.6056	1062	-4.95%	99.99%
IntegerLoop	300.491 ± 0.1138	2401644	251.242 ± 0.2050	761	-16.39%	99.97%
JsonSmall	3135.514 ± 17.0954	18609339	2885.890 ± 29.5012	540483	-7.96%	97.10%
List	244.949 ± 0.2017	1784997	217.742 ± 0.2453	803	-11.11%	99.96%
Loop	196.561 ± 0.1462	1581627	161.965 ± 0.1405	771	-17.60%	99.95%
Mandelbrot	565.439 ± 0.1513	5191571	496.384 ± 0.3128	804	-12.21%	99.98%
NBody	5869.196 ± 4.0206	47102081	5301.500 ± 3.2877	1066	-9.67%	100.00%
PageRank	204.388 ± 0.2031	1703836	173.393 ± 0.7059	6037	-15.16%	99.65%
Permute	273.491 ± 0.2795	1649632	236.447 ± 0.1688	776	-13.54%	99.95%
Queens	217.204 ± 0.0963	1376985	191.803 ± 0.1144	794	-11.69%	99.94%
QuickSort	166.488 ± 0.1343	1235682	139.955 ± 0.1858	912	-15.94%	99.93%
Recurse	249.443 ± 0.1334	1967290	212.140 ± 0.1147	766	-14.95%	99.96%
Richards	4511.721 ± 4.1691	35824587	3803.452 ± 3.5815	1299	-15.70%	100.00%
Sieve	400.462 ± 0.5388	3118040	337.759 ± 0.6396	773	-15.66%	99.98%
Storage	90.836 ± 1.2021	779531	78.436 ± 1.3190	823	-13.65%	99.89%
Sum	152.987 ± 0.1092	1202148	123.737 ± 0.1181	763	-19.12%	99.96%
Towers	271.376 ± 0.1319	2296825	232.203 ± 0.1956	813	-14.43%	99.95%
TreeSort	179.148 ± 0.5050	1806430	142.640 ± 0.2324	928	-20.38%	99.95%
WhileLoop	358.626 ± 0.1872	2013973	320.426 ± 0.2700	766	-10.65%	99.96%
Geometric mean	298.406		257.122		-14.87%	

Table 7.3: Results from my finaliser elision experiment, where I evaluate the performance of yksom using two configurations: naive finalisation, and ALLOY's finaliser elision optimisation. Each configuration is compared using a subset of benchmarks on the Rebench benchmarking suite for 30 process executions, where I report 99% confidence intervals. I also count the number of finalisers which were run for each benchmark for a single process execution.

Benchmark	BarriersNone	BarriersAll		BarriersOpt	
	Runtime (ms)	Runtime (ms)	Slowdown	Runtime (ms)	Slowdown
Bounce	177.685 ± 0.5710	200.267 ± 0.8618	12.71%	174.867 ± 0.6087	-1.59%
BubbleSort	76.489 ± 0.4761	88.817 ± 0.3675	16.12%	78.764 ± 0.3020	2.97%
DeltaBlue	1332.107 ± 31.0046	1492.094 ± 37.7451	12.01%	1419.885 ± 23.2784	6.59%
Dispatch	68.006 ± 0.1652	76.597 ± 0.1439	12.63%	69.342 ± 0.1457	1.96%
Fannkuch	11.196 ± 0.1818	12.764 ± 0.2088	14.01%	11.344 ± 0.2270	1.32%
Fibonacci	62.955 ± 0.1439	71.017 ± 0.2969	12.81%	61.234 ± 0.3129	-2.73%
FieldLoop	191.046 ± 0.2221	246.352 ± 0.1694	28.95%	198.393 ± 0.1252	3.85%
GraphSearch	983.338 ± 32.7477	1261.518 ± 57.6965	28.29%	987.944 ± 36.9464	0.47%
IntegerLoop	254.676 ± 0.3206	288.371 ± 0.3495	13.23%	250.040 ± 0.2444	-1.82%
JsonSmall	2846.510 ± 5.7309	3004.179 ± 88.3440	5.54%	3015.966 ± 63.0982	5.95%
List	212.349 ± 0.3258	247.745 ± 0.3761	16.67%	212.127 ± 0.2569	-0.10%
Loop	171.301 ± 0.2056	188.520 ± 0.1256	10.05%	163.770 ± 0.3270	-4.40%
Mandelbrot	471.427 ± 0.4220	508.122 ± 0.3002	7.78%	498.200 ± 0.4006	5.68%
NBody	5075.057 ± 4.4648	5529.958 ± 6.9342	8.96%	5149.974 ± 3.2279	1.48%
PageRank	164.693 ± 2.6170	185.957 ± 0.8159	12.91%	162.454 ± 0.3700	-1.36%
Permute	233.861 ± 0.5195	266.001 ± 0.3781	13.74%	236.824 ± 0.2042	1.27%
Queens	186.806 ± 0.2976	217.072 ± 0.3082	16.20%	191.837 ± 0.3830	2.69%
QuickSort	140.766 ± 0.4320	153.391 ± 0.5466	8.97%	141.896 ± 0.4044	0.80%
Recurse	203.637 ± 0.2076	234.638 ± 0.1775	15.22%	207.555 ± 0.3163	1.92%
Richards	3579.761 ± 3.1664	4240.377 ± 2.7827	18.45%	3626.458 ± 4.7520	1.30%
Sieve	327.594 ± 1.0237	378.351 ± 2.1365	15.49%	335.019 ± 0.7508	2.27%
Storage	80.075 ± 3.1399	86.691 ± 3.8760	8.26%	78.496 ± 2.9225	-1.97%
Sum	127.385 ± 0.2532	143.214 ± 0.1238	12.43%	122.600 ± 0.3598	-3.76%
Towers	230.223 ± 0.6778	265.052 ± 0.7693	15.13%	234.641 ± 0.3364	1.92%
TreeSort	143.269 ± 2.3865	158.499 ± 0.6312	10.63%	142.932 ± 0.6975	-0.24%
WhileLoop	298.890 ± 0.7961	414.810 ± 0.2913	38.78%	308.436 ± 0.6255	3.19%

Table 7.4: Results from my early finaliser prevention experiment, where I evaluate the performance of yksom using three configurations: BarriersNone, where there are no compiler barriers (which is unsound); BarriersAll, where every single `Gc` reference has a corresponding barrier; and BarriersOpt, where barriers can be optimised away where Alloy is certain they are unnecessary. Each configuration is compared using a subset of benchmarks on the Rebench benchmarking suite for 30 process executions, where I report 99% confidence intervals. The slowdown of BarriersOpt and BarriersAll is shown when compared with BarriersNone.

7.4 Early finaliser prevention

My fourth study aims to understand the performance impact of the early finaliser prevention technique Alloy uses to ensure finalisers are sound. I aim to answer two questions:

1. What is the impact of the compiler barriers that Alloy uses to prevent early finalisation?
2. How effective is the optimisation that alloy uses to remove barriers where it can guarantee they are unnecessary?

As with the previous study in Section 7.3, I use yksom to perform this evaluation. I build three configurations of yksom as follows:

BarriersNone This compiles yksom with a version of Alloy with no compiler barriers to prevent possible early finalisation. This is unsound.

BarriersAll This compiles yksom with a version of Alloy which inserts a compiler barrier for every single `Gc` reference.

BarriersOpt This compiles yksom with a version of Alloy which inserts a compiler barrier `Gc` reference which cannot be optimised away with the approach described in Section 6.9.

I use the SOM benchmark suite using Rebench benchmarking suite using two versions of yksom for each Alloy variant. I run each SOM benchmark for 30 *process executions*, where each instance of yksom is closed down and started again. Rebench records wall-clock times using Python's `time()` function. I also run each SOM benchmark and record the number of finalisers that were executed for each Alloy variant.

7.4.1 Results

Table 7.4 shows the performance results for this study. As one would expect, inserting barriers for every GC reference causes a slowdown on every benchmark. However, perhaps surprisingly, while slower on several benchmarks, BarriersOpt is faster than BarriersNone on the Bounce, Fibonacci, IntegerLoop, Loop, and Sum benchmarks (and statistically insignificant on List, PageRank, Storage, and TreeSort). While I am not certain why this is the case, I believe the most likely reason for this is that sometimes, barriers can be inserted in such a way that they keep GC objects alive for the duration of the shorter running benchmarks, thus preventing finalisation for those objects entirely. Furthermore, when I disable the scheduling of finalisers entirely for the BarriersOpt configuration, each previously of these benchmarks regressed from between 0.83% to 3.32%

7.5 Comparison between other GCs

My fifth and final experiment aims to understand the performance costs of Alloy against other garbage collected approaches in Rust.

The overall question I would like this experiment to answer is: how fast is Alloy when compared with the other garbage collected options available in Rust? While I have been able to provide a more detailed assessment of Alloy's performance when used to implement other languages, I am only able to provide a limited comparison of Alloy's

typed-arena	Alloy	Rust-GC	Rc
3.839 ±0.0037	9.190 ± 0.0315	60.448 ± 0.143	9.748 ±0.0216

Table 7.5: Results from my experiment comparing Alloy against three other garbage collection configurations on the Binary Trees benchmark for 30 process executions, where I report 99% confidence intervals. This clearly shows that except for the index-arena (which deallocates all its memory at once) Alloy is the fastest configuration.

performance relative to other collectors: converting benchmarks to Luster's unusual approach was prohibitively difficult; and Bronze crashed with many benchmarks. In addition, I had difficultly finding suitable Rust programs which were practical enough to modify to use the various tracing GC implementations while also performing enough heap allocations to be useful. Fortunately, and despite these restrictions, this experiment is still able to provide valuable insights.

In this experiment, I compare the performance of Alloy against three different approaches to managing memory: using a indexed-arena, Rust-GC (Section 5.5), and Rust's standard reference counting library. I run each configuration on the Binary Trees benchmark from the Computer Language Benchmark Game for 30 process executions. The wall-clock times are recorded before and after each process execution using the multitime tool.

7.5.1 Results

Table 7.5 shows the results for this experiment. The results shows that for Binary Trees (an allocation heavy benchmark) Typed-arena was the fastest as it never performs any deallocation during the benchmark run, it simply deallocates all memory at the end.

Rust-GC performs poorly for two reasons. First, it uses a form of reference counting to track the roots for each garbage collected object. Second, it has a naive implementation of the mark-sweep algorithm and does not use parallel collector threads.

Chapter 8

Conclusions

8.1 Summary

In this thesis, I first converted around half of the V8 JavaScript VM to use conservative GC so that direct references to garbage-collected heap objects can be used instead of handles. I also introduced ALLOY, an extension of the Rust language with optional tracing garbage collection. A key difficulty in retrofitting GC to an existing language is how one ensures finalisers are both sound and perform well. A major contribution of this thesis are the techniques I offer on how one overcomes these issues: finaliser safety analysis, early finaliser prevention, and finaliser elision. A more detailed summary is as follows.

In Chapter 3 I show that conservative GC can be used to make the VM code of an industrial strength GC more ergonomic. By this I mean that conservative GC allows the VM to be written using direct references to heap objects instead of via handles. This has the following ergonomic advantages: first, it reduces code size by removing handle scopes; second, it allows for more idiomatic usage of pointers in C++; and third, object reference lifetimes are now tied to lexical scope, making them easier to reason about. I show some of the practical concerns involved in migrating from handles to direct references, and offer a migration strategy that I hope other VM authors can learn from when converting industrial strength VMs of similar size.

I then showed that this undertaking has a promising impact on performance: while only around half of V8 was converted, there was a performance improvement of 1-10% in 10 of the benchmarks, while 5 regressed by 2-6%. Moreover, due to the considerable engineering effort involved, I have so far only migrated around half of V8's 1.75MLoc. This bodes well for a full migration since the current state can be considered the worst of both worlds: we must still keep some handle related GC code around; and there are performance bottlenecks around the boundaries where code is converted between handles and direct references.

Chapter 4 provides an overview of Rust and, in particular, its approach to memory management. It highlights that one of the main problems with Rust's affine type system is how difficult it is to implement cyclic data structures. This problem has motivated an active research area in retrofitting a form of opt-in tracing garbage collection for Rust.

In Chapter 5, I discuss some of the existing approaches to opt-in tracing GC in Rust, and introduce ALLOY, which attempts to address some of their shortcomings. I show how conservative GC can be retrofit onto a modern systems programming language, and that one of the major lessons in this endevour is the difficulty in ensuring that finalisation is sound and offers reasonable performance.

In Chapter 6, I offer solutions to these problems. I first introduce a novel form of static analysis, known as *finaliser safety analysis* (FSA) – one of my main contributions. FSA allows for sound GC finalisation to be retrofit to Rust by preventing programs which have potentially unsafe finalisers from compiling. I show that FSA can help make ALLOY much easier to use, because it even allows Rust objects with unsafe fields to be garbage collected provided those fields are never dereferenced inside the finaliser.

A lesson learned when developing ALLOY is that if GC is naively retrofit to Rust, finalisers will be pervasive so that they can call the destructors of other, non-GC'd objects. This can be a major source of slowdown in programs which use ALLOY, so I developed an optimisation for reducing the number of finalisers that need running in an ALLOY program, known as *finaliser elision*. Finaliser elision identifies which finalisers are used solely to deallocate non-GC'd objects, and then promotes those objects to be managed by the GC instead. In my evaluation I found that this can improve the runtime performance of a program by around 14%.

In Chapter 7, I undertake a detailed performance evalutation of ALLOY. I pay particular attention to the performance and memory costs of ALLOY when compared with Rust's reference counting library. This is done with two separate studies where ALLOY is used in reasonably complex languages VMs: SOMRS, and WLAMBDA. A key insight is that although SOMRS is on average 1.12x slower when using ALLOY, SOMRS had previously leaked memory in two of the benchmarks when using reference counting. ALLOY, on the other hand, is able to correctly manage the cyclic data structures in these benchmarks.

8.2 Future Work

8.2.1 V8

In Chapter 3 I explain how because of the considerable engineering effort required, only around half of V8's codebase was migrated to use direct references. A full migration would offer a more convincing picture of both the ergonomic benefits of using direct references as well as the overall performance impact. As of the writing of this thesis, this work is currently underway.

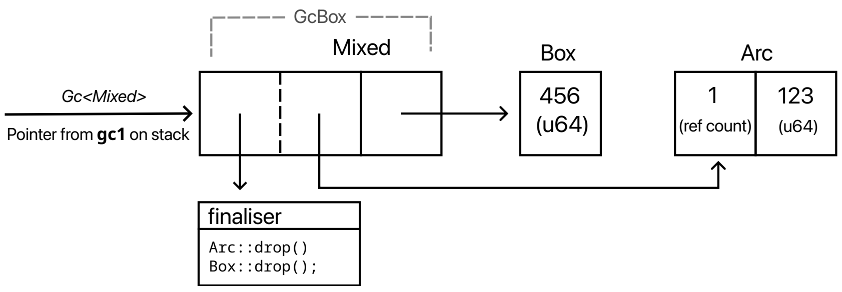
8.2.2 Alloy

Conservatively scanning through objects is slow. The ideal scenario would be for Alloy to use semi-conservative GC similar to V8, where heap objects are precisely traced during marking. While this would constitute a significant engineering effort, a possible intermediate solution would be to avoid tracing through objects which can statically be guaranteed not to contain pointers. This could be achieved by marking such types with a trait (e.g. `NoTrace`). When values of such types are allocated, the BDWGC could use the `GC_malloc_atomic` call to allocate such objects in a separate, non-tracing pool. I believe this could help improve the performance of Alloy by reducing marking time when many objects are allocated.

Alloy does not currently perform partial finaliser elision. This is where a `Gc<T>` will only elide `drop` methods for fields in `T` which are marked `FinalizerOptional`, but still finalise others. Consider an example of this where a struct contains two fields, one of which *could* be elided:

```
1 struct Mixed {
2     a: Arc<u64>,
3     b: Box<u64>,
4 }
5
6 let a = Mixed { a: Arc::new(123), b: Box::new(456) };
7 let b = Gc::new(a);
```

Here, the object is finalised in its entirety. The object `Gc<Mixed>` (line 7) has the following memory representation:



The field `b` is a `Box` type, which implements `FinalizerOptional`. However, `a`'s type (`Arc`) does not, as it is not the sole owner of the integer managed by the `Arc`. If the collector elided its finaliser then it would not decrement its reference count when the `Gc` becomes unreachable.

Support for partial finaliser elision would mean that before the `Gc` is collected, only the `Arc` is finalised. This would further reduce the number of finalisers that need running before a collection, improving ALLOY's performance.

To implement partial finalisation, the Rust compiler would need to be extended to conditionally remove drop calls of a value's fields depending on whether it is being finalised, or dropped regularly as part of RAII. I have not yet come up with a suitable way to implement this.

8.3 Extending finaliser safety analysis

Finaliser safety analysis does not currently perform def-use analysis on references so it cannot identify whether they came from the `self` reference on an object, or a reference to another field. An extension to FSA which provides this would allow the borrow-or-finalise rule to be relaxed and the drop method checked in the same way that it is for `FinalizerSafe` types.

While some of the individual soundness issues that FSA seeks to address are unique to Rust, the fundamental idea of FSA is not. Future work could explore whether FSA can be applied to retrofitting GC to other non-managed languages such as C++. The FSA analysis phase could be performed on the LLVM IR of a finaliser written in C++.

Bibliography

Ager, M., E. Corry, V. Egorov, K. Hara, G. Wibling, and I. Zerny (2013, December). Oilpan: tracing garbage collection for Blink. `https://docs.google.com/document/d/1y7_0ni0E_kxvrah-QtnreMlzCDKN3QP4BN1Aw7eSLfY`. Accessed on 2023-10-12.

Ainsworth, S. and T. M. Jones (2020, May). MarkUs: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 578–591.

Apple (1998). Webkit. `https://webkit.org`. Accessed: 2023-02-16.

Apple (2013). JavaScriptCore. `https://developer.apple.com/documentation/javascriptcore`. Accessed: 2023-04-03.

Bacon, D. F., P. Cheng, and V. Rajan (2004, October). A unified theory of garbage collection. In *OOPSLA*, pp. 50–68.

Bacon, D. F. and V. T. Rajan (2001, June). Concurrent cycle collection in reference counted systems. In *ECOOP*, pp. 207–235.

Baker, J., A. Cunei, F. Pizlo, and J. Vitek (2007, March). Accurate garbage collection in uncooperative environments with lazy pointer stacks. In *ETAPS*, pp. 64–79.

Baker Jr, H. C. and C. Hewitt (1977, August). The incremental garbage collection of processes. *ACM SIGART Bulletin* (64), 55–59.

Banerjee, S., D. Devecsery, P. M. Chen, and S. Narayanasamy (2020, November). Sound garbage collection for C using pointer provenance. *4*, 1–28.

Bartlett, J. F. (1988, April). Compacting garbage collection with ambiguous roots. *ACM Lisp Pointers 1*(6), 3–12.

Blackburn, S. M., P. Cheng, and K. S. McKinley (2004, June). Myths and realities: The performance impact of garbage collection. In *ICMMCS*, Volume 32, pp. 25–36.

Blackburn, S. M. and K. S. McKinley (2003, October). Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA*, Volume 38, pp. 344–358.

Boehm, H.-J. (1996, May). Simple garbage collector safety. In *PLDI*, Volume 31, pp. 89–98.

Boehm, H.-J. (2003, January). Destructors, finalizers, and synchronization. In *POPL*, pp. 262–272.

Boehm, H.-J. (2004, January). The space cost of lazy reference counting. In *POPL*, Volume 39, pp. 210–219.

Boehm, H.-J. and M. Weiser (1988, September). Garbage collection in an uncooperative environment. *SPE 18*(9), 807–820.

Brooks, R. A. (1984, August). Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LISP and Functional Programming*, pp. 256–262.

Caplinger, M. A. (1998, January). A memory allocator with garbage collection for C. In *USENIX*, pp. 325–330.

Cheney, C. J. (1970, November). A nonrecursive list compacting algorithm. *Commun. ACM 13*(11), 677–678.

Chiovoloni, T. (2015, August). typed-arena: A fast (but limited) allocation arena for values of a single type. `https://github.com/thomcc/rust-typed-arena`. Accessed: 2023-04-03.

Coblenz, M., M. Mazurek, and M. Hicks (2022, May). Does the Bronze garbage collector make Rust easier to use? A controlled experiment. In *ICSE*.

Collins, G. E. (1960, December). A method for overlapping and erasure of lists. *Communications of the ACM 3*(12), 655–657.

Degenbaev, U., J. Eisinger, M. Ernst, R. McIlroy, and H. Payer (2016, June). Idle time garbage collection scheduling. In *PLDI*, Volume 51, pp. 570–583.

Degenbaev, U., J. Eisinger, K. Hara, M. Hlopko, M. Lippautz, and H. Payer (2018, October). Cross-component garbage collection. In *OOPSLA*, pp. 1–24.

Demers, A., M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker (1989, December). Combining generational and conservative garbage collection: Framework and implementations. In *POPL*.

Dieckmann, S. and U. Hölzle (1999, November). A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *ECOOP*, pp. 92–115.

Dijkstra, E. W., L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens (1978, November). On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21*(11), 966–975.

Diwan, A., E. Moss, and R. Hudson (1992, June). Compiler support for garbage collection in a statically typed language. In *PLDI*, Volume 27, pp. 273–282.

Egorov, V. (Unknown year). Garbage collection in Dart. `https://mrale.ph/dartvm/gc.html`. Accessed: 2023-03-03.

Fitzgerald, N. (2015, August). bacon-rajan-cc: A reference counted type with cycle collection for Rust. `https://github.com/fitzgen/bacon-rajan-cc`. Accessed: 2023-04-03.

Flood, C. H., D. Detlefs, N. Shavit, and X. Zhang (2001, April). Parallel garbage collection for shared memory multiprocessors. In *JVM Research and Technology Symposium.*

Flood, C. H., R. Kennke, A. Dinn, A. Haley, and R. Westrelin (2016, August). Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *PPPJ*, pp. 1–9.

Galindo Salgado, P. (2022, July). Python garbage collector design. `https://devguide.python.org/internals/garbage-collector/index.html`. Accessed: 2023-04-03.

Google (2008). The V8 JavaScript engine. `https://v8.dev/`. Accessed: 2022-12-17.

Google (2019a, March). Garbage collection in the Dart VM. `https://dart.googlesource.com/sdk/+/refs/tags/2.15.0-99.0.dev/runtime/docs/gc.md`. Accessed: 2023-04-03.

Google (2019b, April). Getting started with embedding V8. `https://v8.dev/docs/embed#handles-and-garbage-collection`. Accessed: 2023-03-03.

Google (2020). Issue 1046776: base::Optional<T> when compiled with clang leads to memory layout that is incompatible with conservative stack scanning. `https://bugs.chromium.org/p/chromium/issues/detail?id=1046776`. Accessed: 2023-03-02.

Goregaokar, M. (2015). rust-gc: a simple tracing (mark and sweep) garbage collector for Rust. `https://github.com/manishearth/rust-gc/`. Accessed: 2022-12-17.

Goregaokar, M. (2016, August). GC support in Rust: API design. `https://manishearth.github.io/blog/2016/08/18/gc-support-in-rust-api-design/`. Accessed: 2023-04-03.

Goregaokar, M. (2018, December). elsa: append-only collections for Rust where borrows to entries can outlive insertions. `https://github.com/Manishearth/elsa/`. Accessed: 2023-04-03.

Goregaokar, M. (2021a, March). Arenas in Rust. `https://manishearth.github.io/blog/2021/03/15/arenas-in-rust/`. Accessed: 2023-04-03.

Goregaokar, M. (2021b, April). A tour of safe tracing GC designs in Rust. `https://manishearth.github.io/blog/2021/04/05/a-tour-of-safe-tracing-gc-designs-in-rust/`. Accessed: 2023-04-03.

Halstead, R. H. (1984, August). Implementation of MultiLisp: Lisp on a multiprocessor. In *LFP*, pp. 9–17.

Hanson, D. R. (1990, January). Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience 20*(1), 5–12.

Henderson, F. (2002, June). Accurate garbage collection in an uncooperative environment. In *ISMM*, pp. 150–156.

Huang, X., S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng (2004, October). The garbage collection advantage: Improving program locality. In *OOPSLA*, Volume 39, pp. 69–80.

Jones, R., A. Hosking, and E. Moss (2016, September). *The Garbage Collection Handbook: the Art of Automatic Memory Management.* Chapman and Hall/CRC.

Kalibera, T. and R. Jones (2011, November). Handles revisited: optimising performance and memory costs in a real-time collector. *ACM SIGPLAN Notices 46*(11), 89–98.

KDE (2013, May). heaptrack - a heap memory profiler for Linux. `https://github.com/KDE/heaptrack`. Accessed: 2023-04-03.

Klabnik, S. and C. Nichols (2018, June). *The Rust Programming Language.* No Starch Press.

Klock, F. S. (2015, November). GC and Rust: specifying the problem. `http://blog.pnkfx.org/blog/2015/11/10/gc-and-rust-part-1-specing-the-problem/`. Accessed: 2023-04-03.

Klock, F. S. (2016, January). GC and Rust: The roots of the problem. `http://blog.pnkfx.org/blog/2016/01/01/gc-and-rust-part-2-roots-of-the-problem/`. Accessed: 2023-04-03.

Krasner, G. (1983, December). *Smalltalk-80: Bits of history, words of advice.* Addison-Wesley.

Lin, Y., S. Blackburn, A. Hosking, and M. Norrish (2016, June). Rust as a language for high performance GC implementation. In *ISMM*, pp. 89–98.

Lin, Y., S. M. Blackburn, A. L. Hosking, and M. Norrish (2017, November). MMTk: a framework for the design and implementation of memory managers. `https://github.com/mmtk/mmtk-core`. Accessed: 2023-04-03.

Lins, R. D. (1992, December). Cyclic reference counting with lazy mark-scan. In *Information Processing Letters*, Volume 44, pp. 215–220.

Marr, S. (2018, August). ReBench: Execute and document benchmarks reproducibly. `https://github.com/smarr/rebench`. Accessed: 2022-12-19.

Marr, S., B. Daloze, and H. Mössenböck (2016, November). Cross-language compiler benchmarking: are we fast yet? *ACM SIGPLAN Notices 52*(2), 120–131.

Matsakis, N. (2013, January). Destructors and finalizers in Rust. `http://smallcultfollowing.com/babysteps/blog/2013/01/17/destructors-and-finalizers-in-rust/`. Accessed: 2023-04-03.

McCarthy, J. (1960, April). Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM 3*(4), 184–195.

Microsoft (2019, July). A proactive approach to more secure code. `https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/`. Accessed: 2023-04-03.

Mozilla (1995). Spidermonkey. `https://spidermonkey.dev`. Accessed: 2023-02-16.

Mozilla (2010, December). The Rust programming language. `https://rust-lang.org`.

Mozilla (2012). Servo. `https://servo.org`. Accessed: 2022-12-17.

Nethercote, N. and J. Seward (2007, June). Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices 42*(6), 89–100.

Novark, G., E. D. Berger, and B. G. Zorn (2009, June). Efficiently and precisely locating memory leaks and bloat. In *PLDI*, pp. 397–407.

Oracle (2007, December). The HotSpot runtime. `https://github.com/openjdk/jdk/blob/master/src/hotspot/share/runtime/handles.hpp#L36`. Accessed: 2023-03-03.

Pierce, B. C. (2004, December). *Advanced topics in Types and Programming Languages*. MIT press.

Pirinen, P. P. (1998, October). Barrier techniques for incremental tracing. In *ISMM*, Volume 34, pp. 20–25.

Pizlo, F. (2017, January). Introducing Riptide: WebKit's retreating wavefront concurrent garbage collector. `https://webkit.org/blog/7122/introducing-riptide-webkits-retreating-wavefront-concurrent-garbage-collector/`. Accessed: 2023-02-16.

Polomack, N. (2020, June). som-rs: An alternative implementation of the Simple Object Machine, written in Rust. `https://github.com/Hirevo/som-rs`. Accessed: 2023-04-03.

Powers, B., D. Tench, E. D. Berger, and A. McGregor (2019, June). Mesh: Compacting memory management for C/C++ applications. In *PLDI*, pp. 333–346.

Project, T. L. (2014, December). Garbage collection with LLVM. `https://llvm.org/docs/GarbageCollection.html`. Accessed: 2023-04-03.

Project, T. R. (2018, May). RustPython: a Python interpreter written in Rust. `https://github.com/RustPython/RustPython`. Accessed: 2023-04-03.

Reames, P. (2017, October). Falcon: an optimizing Java JIT. `https://llvm.org/devmtg/2017-10/slides/Reames-FalconKeynote.pdf`. Accessed: 2023-04-03.

Röjemo, N. and C. Runciman (1996, June). Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *ICFP*, Volume 31, pp. 34–41.

Saunders, R. A. (1974, March). *The LISP system for the Q-32 computer*. Berkeley and Bobrow.

Serebryany, K., D. Bruening, A. Potapenko, and D. Vyukov (2012, June). AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, pp. 309–318.

Shahriyar, R., S. M. Blackburn, and D. Frampton (2012, June). Down for the count? Getting reference counting back in the ring. In *ISMM*, pp. 73–84.

Shahriyar, R., S. M. Blackburn, and K. S. McKinley (2014, October). Fast conservative garbage collection. In *OOPSLA*, pp. 121–139.

Stichnoth, J. M., G.-Y. Lueh, and M. Cierniak (1999, May). Support for garbage collection at every instruction in a Java compiler. In *PLDI*, Volume 34, pp. 118–127.

Stroustrup, B. (1997, June). *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley.

SUN Microsystems (2006, April). Memory management in the Java HotSpot Virtual Machine. `https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`. Accessed: 2023-04-03.

Suzuki, M. and M. Terashima (1995, October). Time-and space-efficient garbage collection based on sliding compaction. *IPSJ 36*(4), 925–931.

Tene, G., B. Iyengar, and M. Wolf (2011, June). C4: The continuously concurrent compacting collector. In *ISMM*, pp. 79–88.

The Chromium Developers (2022, March). Chromium memory safety. `https://www.chromium.org/Home/chromium-security/memory-safety/`. Accessed: 2023-04-03.

Tratt, L. (2014, November). multitime. `https://tratt.net/laurie/src/multitime/`. Accessed: 2023-04-03.

Ungar, D. (1984, May). Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN notices 19*(5), 157–167.

Ungar, D., D. Grove, and H. Franke (2017, November). Dynamic atomicity: Optimizing swift memory management. *PLDI 52*(11), 15–26.

WebKit (2022). Speedometer2.1 benchmark suite. `https://browserbench.org/Speedometer2.1/`. Accessed: 2023-03-02.

WeirdConstructor (2019, May). WLambda: embeddable scripting language for Rust. `https://github.com/WeirdConstructor/WLambda/`. Accessed: 2023-04-03.

West, C. (2018). luster: An experimental Lua VM implemented in pure Rust. `https://github.com/kyren/luster/`. Accessed: 2022-12-17.

West, C. (2019, September). gc-arena: An experimental system for Rust garbage collection. `https://github.com/kyren/gc-arena/`. Accessed: 2023-04-03.

Williams, J. (2018, August). Boa: an experimental JavaScript lexer, parser and interpreter written in Rust. `https://github.com/boa-dev/boa`. Accessed: 2023-04-03.

withoutboats (2018). shifgrethor. `https://github.com/withoutboats/shifgrethor/`. Accessed: 2022-12-17.

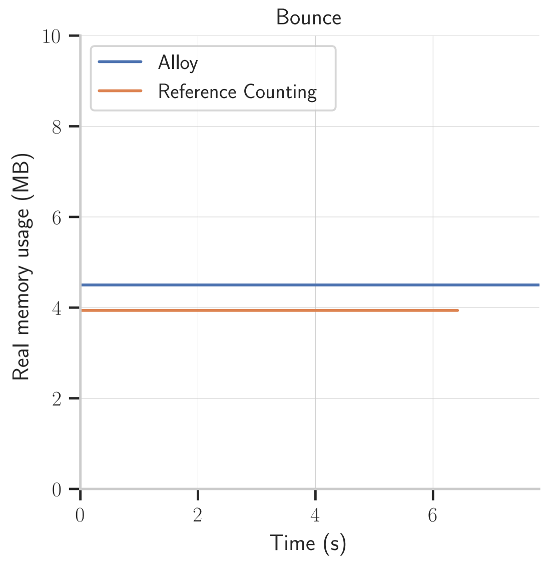
Xu, W., J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu (2015, October). From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *CCS*, pp. 414–425.

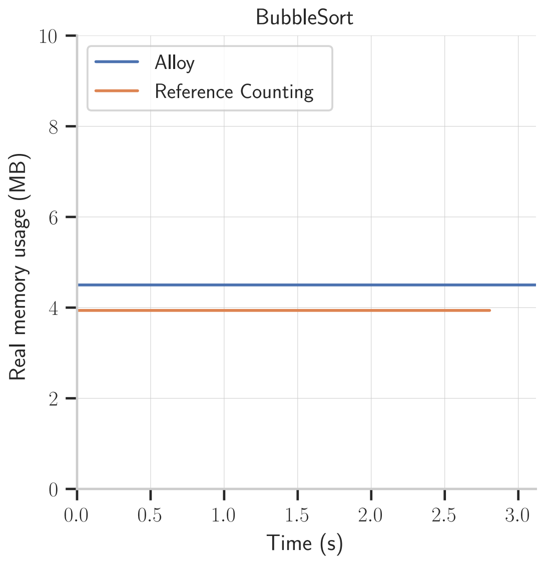
Zorn, B. (1993, July). The measured cost of conservative garbage collection. *Software: Practice and Experience 23*(7), 733–756.

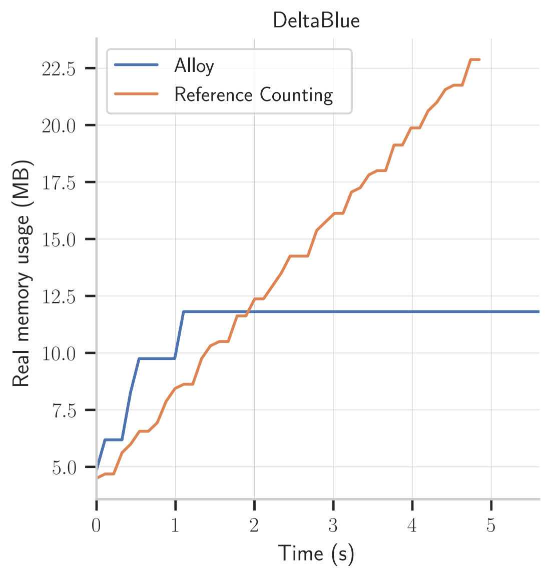
Appendix A

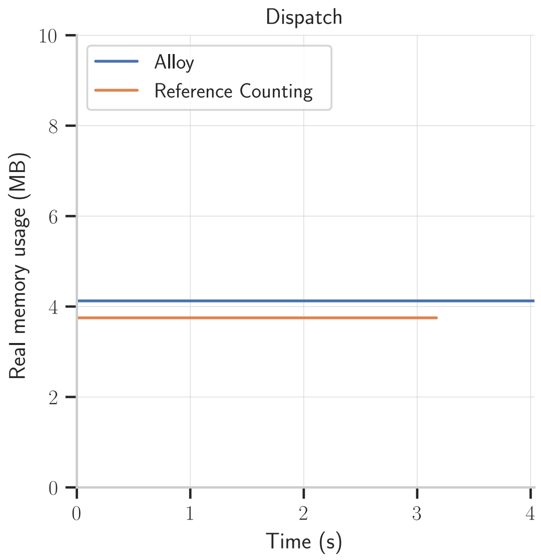
Alloy memory usage for the somrs study

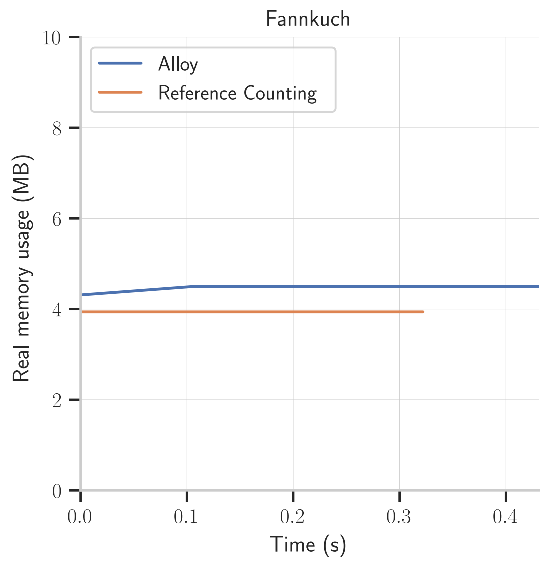
The following graphs show the memory usage over time for each benchmark in the somrs perf study from Section 7.2.1.

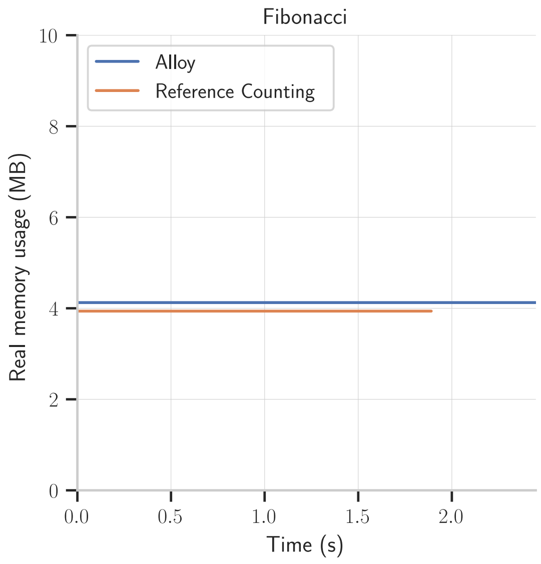


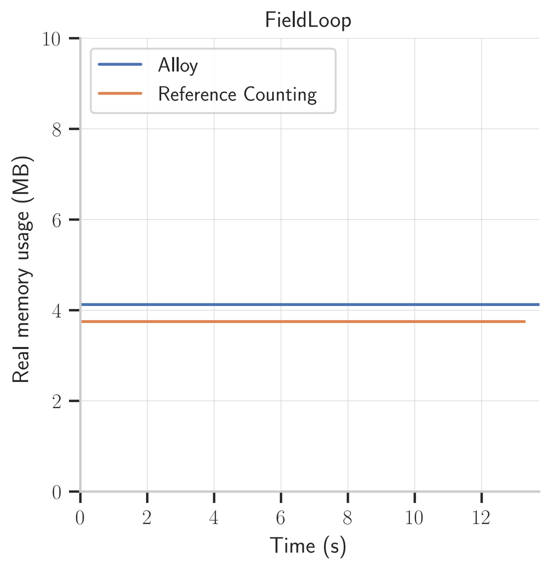


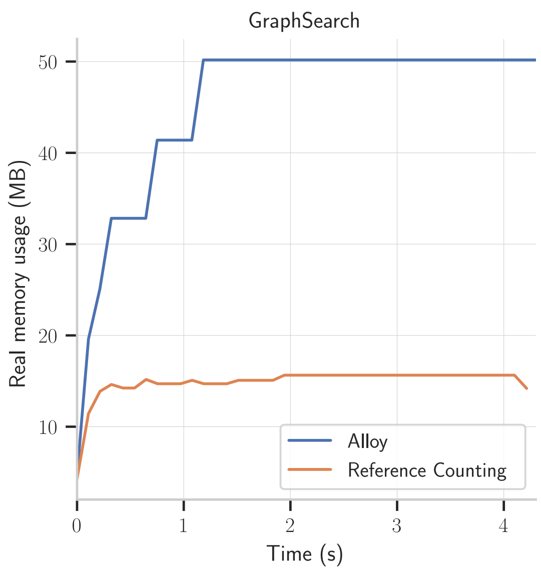

DeltaBlue
Alloy
Reference Counting
Real memory usage (MB)
Time (s)

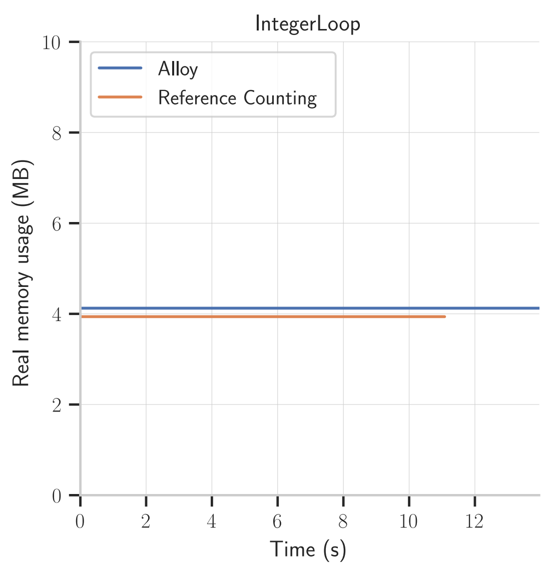

Dispatch
Alloy
Reference Counting
Real memory usage (MB)
Time (s)

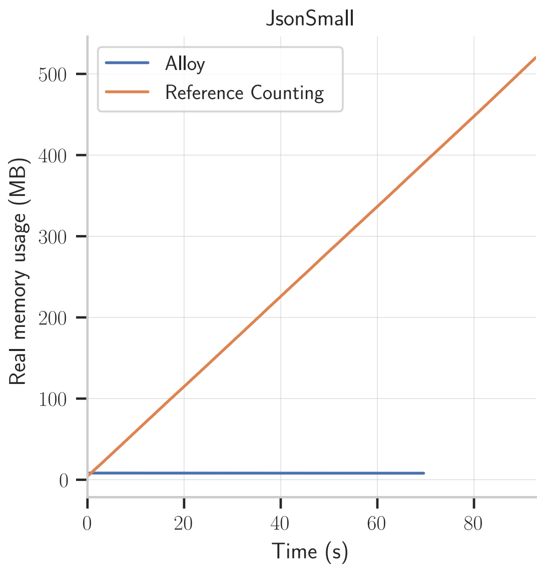

Fannkuch
Alloy
Reference Counting
Real memory usage (MB)
Time (s)

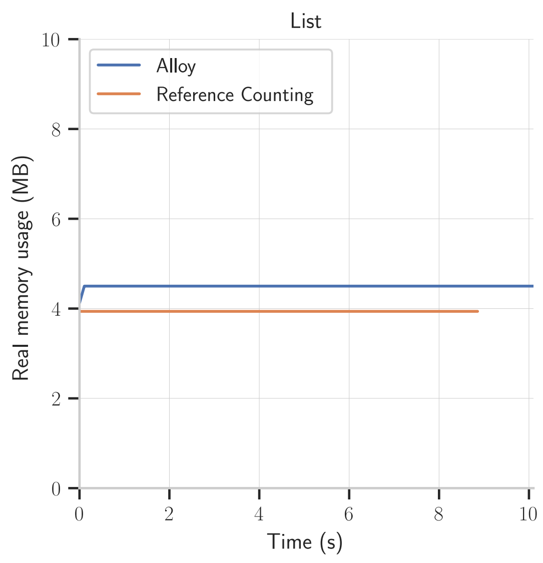

Fibonacci
Alloy
Reference Counting
Real memory usage (MB)
Time (s)

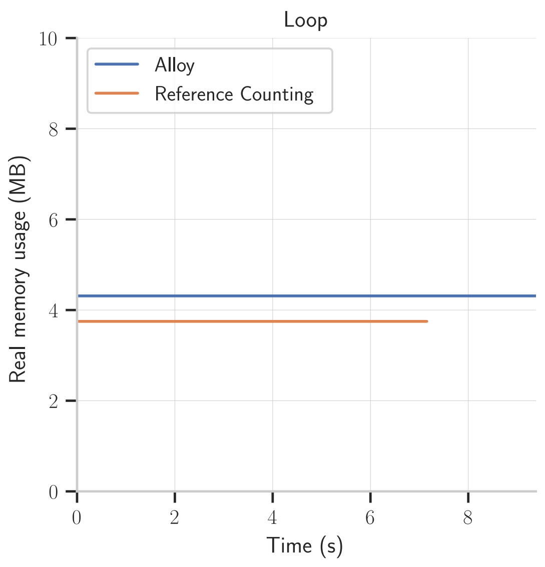

FieldLoop
Alloy
Reference Counting
Real memory usage (MB)
Time (s)

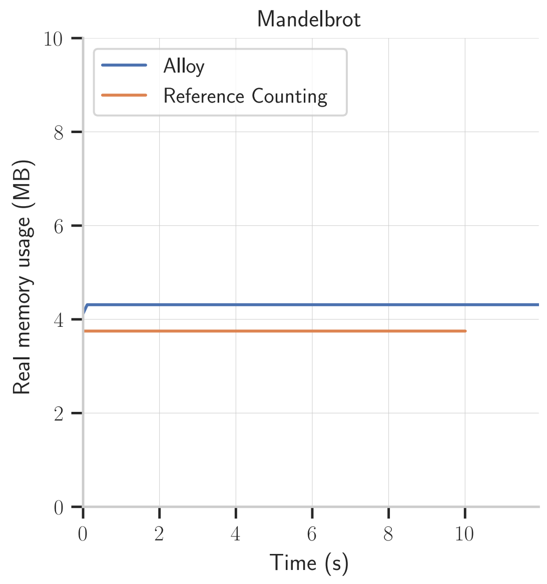

GraphSearch
Alloy
Reference Counting
Real memory usage (MB)
Time (s)

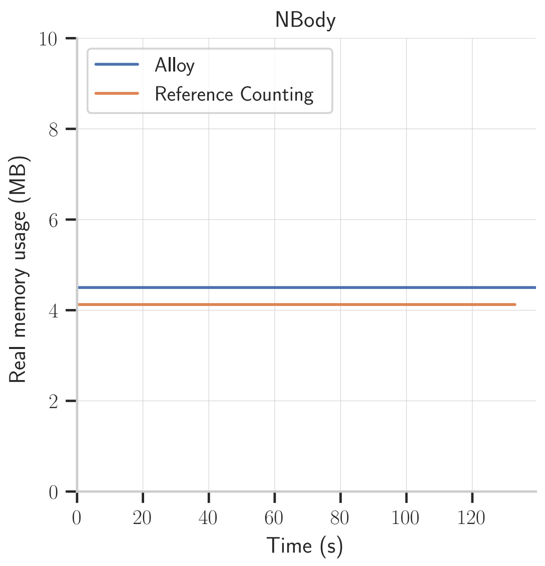

IntegerLoop
Alloy
Reference Counting
Real memory usage (MB)
Time (s)
0
2
4
6
8
10
12

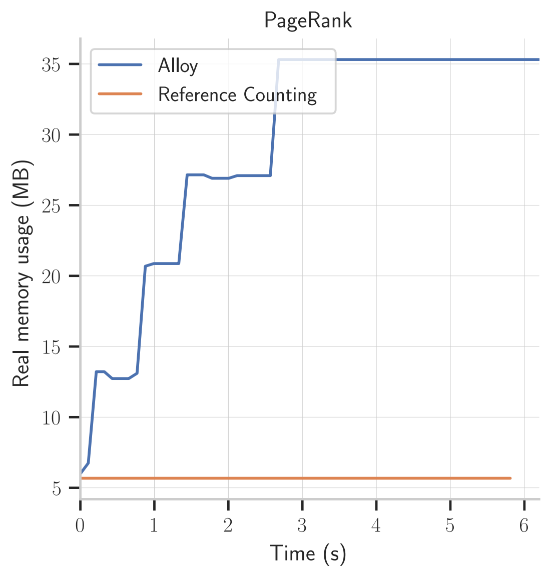

JsonSmall
Alloy
Reference Counting
Real memory usage (MB)
Time (s)
0
100
200
300
400
500
20
40
60
80

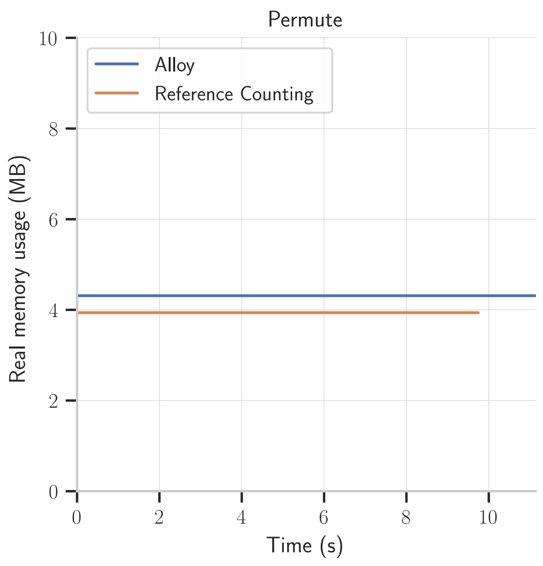

List
Alloy
Reference Counting
Real memory usage (MB)
Time (s)
0
2
4
6
8
10

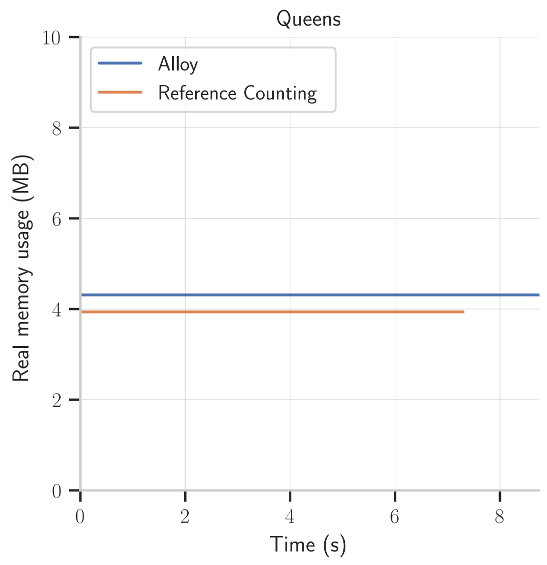

Loop
Alloy
Reference Counting
Real memory usage (MB)
Time (s)
0
2
4
6
8
10

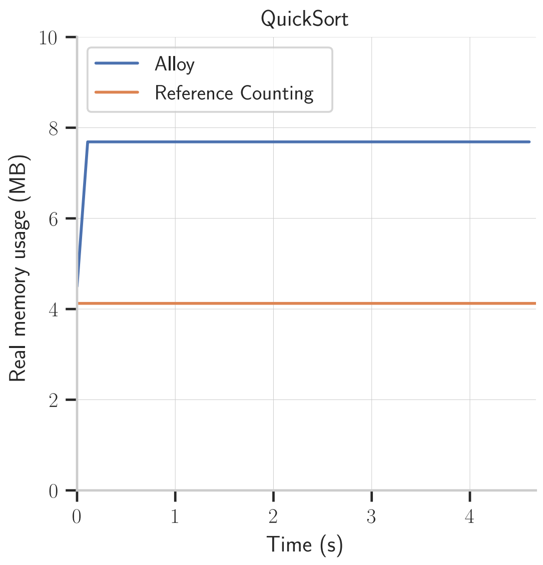

Mandelbrot
Alloy
Reference Counting
Real memory usage (MB)
Time (s)
0
2
4
6
8
10

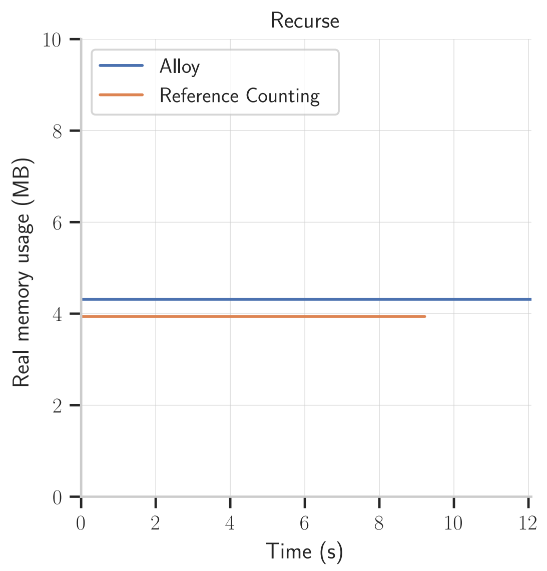

NBody
Alloy
Reference Counting
Real memory usage (MB)
Time (s)
0
2
4
6
8
10
20
40
60
80
100
120

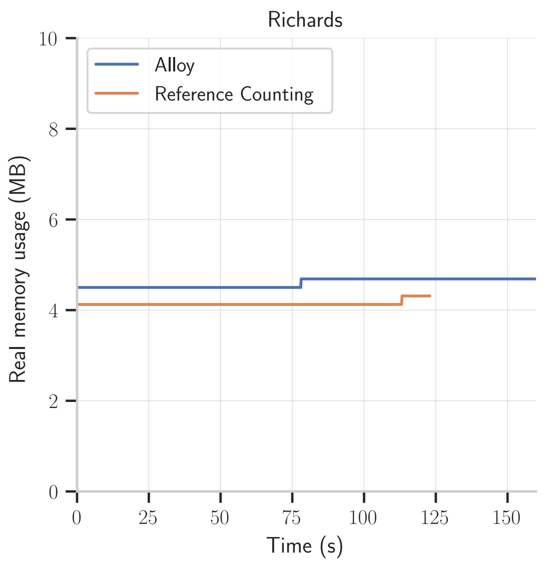

PageRank
Alloy
Reference Counting
Real memory usage (MB)
5
10
15
20
25
30
35
0
1
2
3
4
5
6
Time (s)

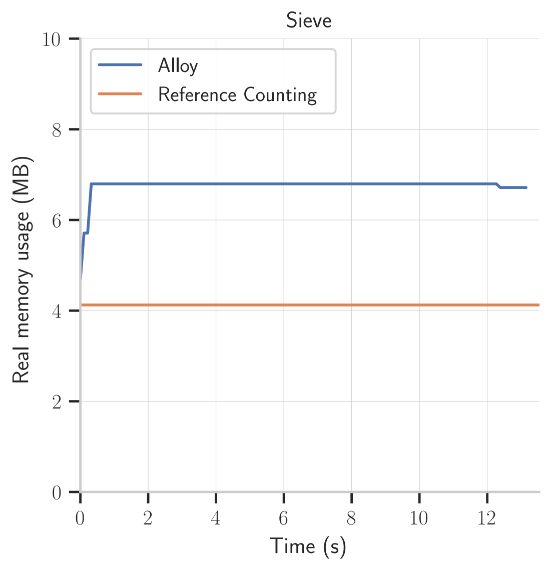

Permute
Alloy
Reference Counting
Real memory usage (MB)
0
2
4
6
8
10
0
2
4
6
8
10
Time (s)

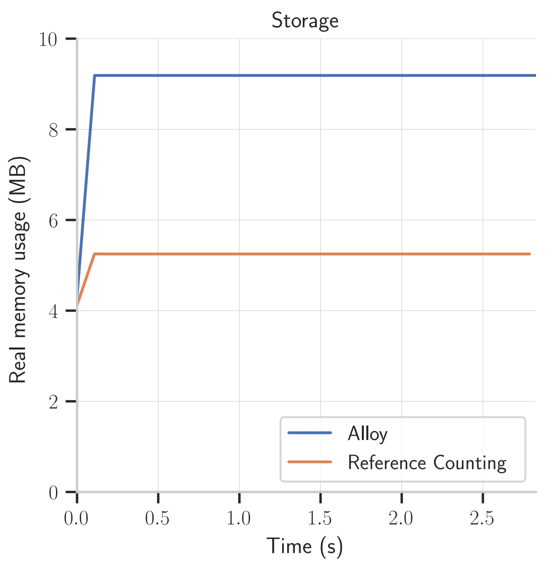

Queens
Alloy
Reference Counting
Real memory usage (MB)
0
2
4
6
8
10
0
2
4
6
8
Time (s)

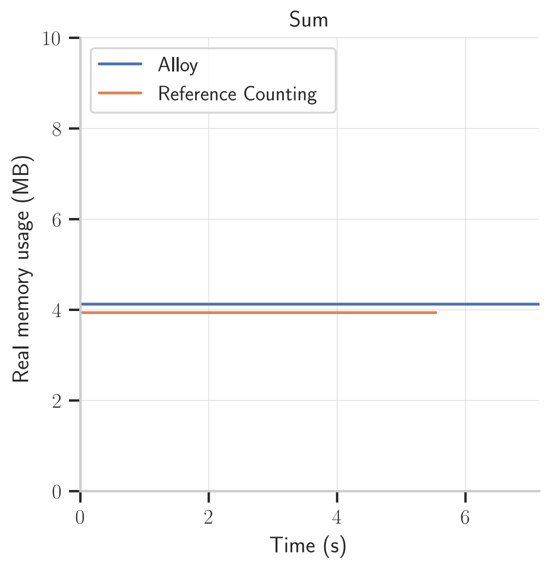

QuickSort
Alloy
Reference Counting
Real memory usage (MB)
0
2
4
6
8
10
0
1
2
3
4
Time (s)

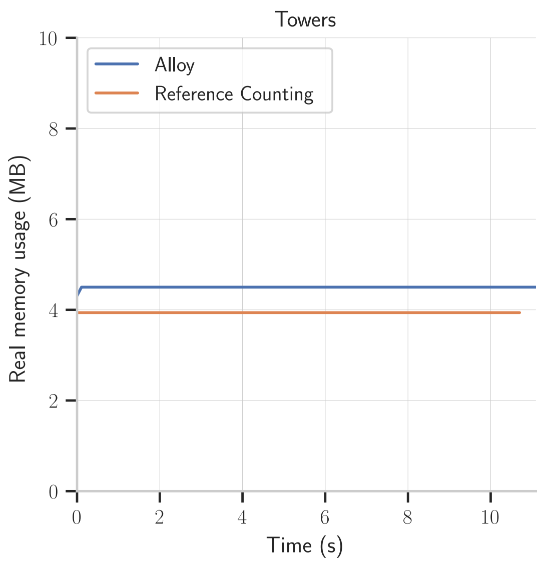

Recurse
Alloy
Reference Counting
Real memory usage (MB)
0
2
4
6
8
10
0
2
4
6
8
10
12
Time (s)

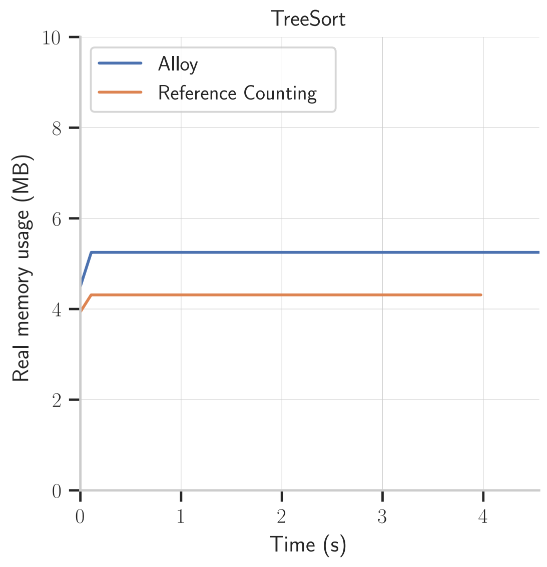

Richards
Alloy
Reference Counting
Real memory usage (MB)
0
2
4
6
8
10
0
25
50
75
100
125
150
Time (s)

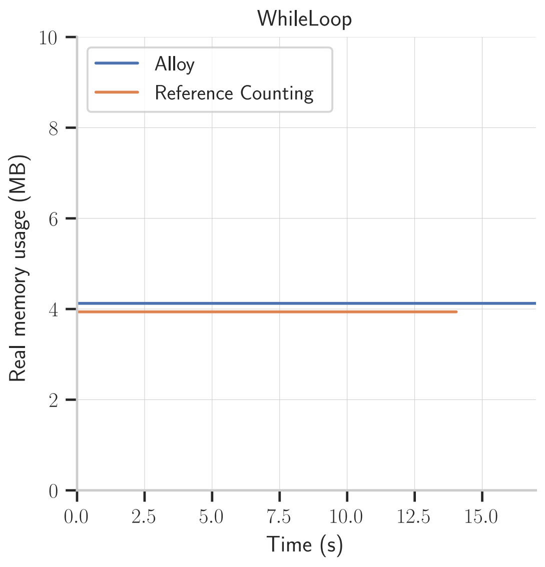

Sieve
Alloy
Reference Counting
Real memory usage (MB)
Time (s)


Storage
Alloy
Reference Counting
Real memory usage (MB)
Time (s)


Sum
Alloy
Reference Counting
Real memory usage (MB)
Time (s)


Towers
Alloy
Reference Counting
Real memory usage (MB)
Time (s)


TreeSort
Alloy
Reference Counting
Real memory usage (MB)
Time (s)


WhileLoop
Alloy
Reference Counting
Real memory usage (MB)
Time (s)

Appendix B

Finaliser safety analysis algorithm

The following shows the full finaliser safety analysis algorithm, implemented as a MIR transform pass in ALLOY's modified version of the Rust compiler.

```
use crate::MirPass;
use rustc_hir::lang_items::LangItem;
use rustc_middle::mir::visit::PlaceContext;
use rustc_middle::mir::visit::Visitor;
use rustc_middle::mir::*;
use rustc_middle::ty::subst::InternalSubsts;
use rustc_middle::ty::{self, ParamEnv, Subst, Ty, TyCtxt};
use rustc_span::symbol::sym;
use rustc_span::{Span, DUMMY_SP};
use rustc_trait_selection::infer::{InferCtxtExt, TyCtxtInferExt};

#[derive(PartialEq)]
pub struct CheckFinalizers;

impl<'tcx> MirPass<'tcx> for CheckFinalizers {
    fn run_pass(&self, tcx: TyCtxt<'tcx>, body: &mut Body<'tcx>) {
        let ctor = tcx.get_diagnostic_item(sym::gc_ctor);

        if ctor.is_none() {
            return;
        }

        let ctor_did = ctor.unwrap();
        let param_env = tcx.param_env_reveal_all_normalized(body.source.def_id());

        for block in body.basic_blocks() {
            match &block.terminator {
                Some(Terminator { kind: TerminatorKind::Call { func, args, .. }, source_info }) => {
                    let func_ty = func.ty(body, tcx);
                    if let ty::FnDef(fn_did, ..) = func_ty.kind() {
                        if *fn_did == ctor_did {
                            let arg = match &args[0] {
                                Operand::Copy(place) | Operand::Move(place) => {
                                    body.local_decls()[place.local].source_info.span
                                }
                                Operand::Constant(con) => con.span,
                            };
                            let arg_ty = args[0].ty(body, tcx);
                            let mut finalizer_cx =
                                FinalizationCtxt { ctor: source_info.span, arg, tcx, param_env };
                            finalizer_cx.check(arg_ty);
                        }
                    }
                }
                _ => {}
            }
        }
    }
}

struct FinalizationCtxt<’tcx> {
    ctor: Span,
    arg: Span,
    tcx: TyCtxt<’tcx>,
    param_env: ParamEnv<’tcx>,
}

impl<’tcx> FinalizationCtxt<’tcx> {
    fn check(&mut self, ty: Ty<’tcx>) {
        if self.is_finalizer_safe(ty) || !self.tcx.needs_finalizer_raw(self.param_env.and(ty)) {
            return;
        }

        // We must now recurse through the ‘Ty‘’s component types and search for
        // all the ‘Drop‘ impls. If we find any, we have to check that there are
        // no unsound projections into fields in their drop method body. More
        // specifically: if one of the drop methods dereferences a field which
        // is !FinalizerSafe, we must throw an error.
        match ty.kind() {
            ty::Infer(ty::FreshIntTy(_))
            | ty::Infer(ty::FreshFloatTy(_))
            | ty::Bool
            | ty::Int(_)
            | ty::Uint(_)
            | ty::Float(_)
            | ty::Never
            | ty::FnDef(..)
            | ty::FnPtr(_)
            | ty::Char
            | ty::GeneratorWitness(..)
            | ty::RawPtr(_)
            | ty::Ref(..)
            | ty::Str
            | ty::Foreign(..) => {
                // None of these types can implement Drop.
                return;
            }
            ty::Dynamic(..) | ty::Error(..) => {
                // Dropping a trait object uses a virtual call, so we can’t
                // work out which drop method to look at compile-time. This
                // means we must be more conservative and bail with an error
                // here, even if the drop impl itself would have been safe.
                self.emit_err();
            }
            ty::Slice(ty) => self.check(*ty),
            ty::Array(elem_ty, ..) => {
                self.check(*elem_ty);
            }
            ty::Tuple(fields) => {
                // Each tuple field must be individually checked for a ‘Drop‘
                // impl.
                fields.iter().for_each(|f_ty| self.check(f_ty));
            }
            ty::Adt(def, substs) if !self.is_copy(ty) => {
                if def.has_dtor(self.tcx) {
                    if def.is_box() {
                        // This is a special case because Box has an empty drop
                        // method which is filled in later by the compiler.
                        self.emit_err();
                    }

                    let drop_trait = self.tcx.require_lang_item(LangItem::Drop, None);
                    let drop_fn = self.tcx.associated_item_def_ids(drop_trait)[0];
                    let substs = self.tcx.mk_substs_trait(ty, substs);
                    let instance = ty::Instance::resolve(self.tcx, self.param_env, drop_fn, substs)
                        .unwrap()
                        .unwrap();
                    let mir = self.tcx.instance_mir(instance.def);
                    let mut checker = ProjectionChecker { cx: self, body: mir };
                    checker.visit_body(&mir);
                }

                for field in def.all_fields() {
                    let field_ty = self.tcx.bound_type_of(field.did).subst(self.tcx, substs);
                    self.check(field_ty);
                }
            }
            _ => (),
        }
    }

    fn is_finalizer_safe(&self, ty: Ty<’tcx>) -> bool {
        self.tcx.infer_ctxt().enter(|infcx| {
            self.tcx.get_diagnostic_item(sym::FinalizerSafe).map(|t| {
                infcx
                    .type_implements_trait(t, ty, InternalSubsts::empty(), self.param_env)
                    .may_apply()
            }) == Some(true)
        })
    }

    fn is_copy(&self, ty: Ty<’tcx>) -> bool {
        ty.is_copy_modulo_regions(self.tcx.at(DUMMY_SP), self.param_env)
    }

    fn is_gc(&self, ty: Ty<’tcx>) -> bool {
        if let ty::Adt(def, ..) = ty.kind() {
            if def.did() == self.tcx.get_diagnostic_item(sym::gc).unwrap() {
                return true;
            }
        }
        return false;
    }

    fn emit_err(&self) {
        let arg = self.tcx.sess.source_map().span_to_snippet(self.arg).unwrap();
        let mut err = self
            .tcx
            .sess
            .struct_span_err(self.arg, format!("‘{arg}‘␣cannot␣be␣safely␣finalized.",));
        err.span_label(self.arg, "has␣a␣drop␣method␣which␣cannot␣be␣safely␣finalized.");
        err.span_label(
            self.ctor,
            format!("‘Gc::new‘␣requires␣that␣it␣implements␣the␣‘FinalizeSafe‘␣trait.",),
        );
        err.help(format!("‘Gc‘␣runs␣finalizers␣on␣a␣separate␣thread,␣so␣‘{arg}‘␣must␣implement␣‘Finaliz
        err.emit();
    }
}

```

```
struct ProjectionChecker<’a, ’tcx> {
    cx: &’a FinalizationCtxt<’tcx>,
    body: &’a Body<’tcx>,
}

impl<’a, ’tcx> ProjectionChecker<’a, ’tcx> {
    fn emit_err(&self, ty: Ty<’tcx>, span: Span) {
        let arg = self.cx.tcx.sess.source_map().span_to_snippet(self.cx.arg).unwrap();
        let mut err = self
            .cx
            .tcx
            .sess
            .struct_span_err(self.cx.arg, format!("‘{arg}‘␣cannot␣be␣safely␣finalized.",));
        if self.cx.is_gc(ty) {
            err.span_label(self.cx.arg, "has␣a␣drop␣method␣which␣cannot␣be␣safely␣finalized.");
            err.span_label(span, "caused␣by␣the␣expression␣here␣in␣‘fn␣drop(&mut)‘␣because");
            err.span_label(span, "it␣uses␣another␣‘Gc‘␣type.");
            err.help("‘Gc‘␣finalizers␣are␣unordered,␣so␣this␣field␣may␣have␣already␣been␣dropped.␣It␣is
        } else {
            err.span_label(self.cx.arg, "has␣a␣drop␣method␣which␣cannot␣be␣safely␣finalized.");
            err.span_label(span, "caused␣by␣the␣expression␣in␣‘fn␣drop(&mut)‘␣here␣because");
            err.span_label(span, "it␣uses␣a␣type␣which␣is␣not␣safe␣to␣use␣in␣a␣finalizer.");
            err.help("‘Gc‘␣runs␣finalizers␣on␣a␣separate␣thread,␣so␣drop␣methods\nmust␣only␣use␣values␣
        }
        err.emit();
    }
}

impl<’a, ’tcx> Visitor<’tcx> for ProjectionChecker<’a, ’tcx> {
    fn visit_projection(
        &mut self,
        place_ref: PlaceRef<’tcx>,
        context: PlaceContext,
        location: Location,
    ) {
        for (_, proj) in place_ref.iter_projections() {
            match proj {
                ProjectionElem::Field(_, ty) => {
                    if !self.cx.is_finalizer_safe(ty) {
                        let span = self.body.source_info(location).span;
                        self.emit_err(ty, span);
                    }
                }
                _ => (),
            }
        }
        self.super_projection(place_ref, context, location);
    }

    fn visit_terminator(&mut self, terminator: &Terminator<'tcx>, location: Location) {
        if let TerminatorKind::Call { ref args, .. } = terminator.kind {
            for caller_arg in self.body.args_iter() {
                let recv_ty = self.body.local_decls()[caller_arg].ty;
                for arg in args.iter() {
                    let arg_ty = arg.ty(self.body, self.cx.tcx);
                    if arg_ty == recv_ty {
                        // Currently, we do not recurse into function calls
                        // to see whether they access `!FinalizerSafe`
                        // fields, so we must throw an error in `drop`
                        // methods which call other functions and pass
                        // `self` as an argument.
                        //
                        // Here, we throw an error if `drop(&mut self)`
                        // calls a function with an argument that has the
                        // same type as the drop receiver (e.g. foo(x:
                        // &Self)). This approximation will always prevent
                        // unsound `drop` methods, however, it is overly
                        // conservative and will prevent correct examples
                        // like below from compiling:
                        //
                        // ```
                        // fn drop(&mut self) {
                        //   let x = Self { ... };
                        //   x.foo();
                        // }
                        // ```
                        //
                        // This example is sound, because `x` is a local
                        // that was instantiated on the finalizer thread, so
                        // its fields are always safe to access from inside
                        // this drop method.
                        //
                        // However, this will not compile, because the
                        // receiver for `x.foo()` is the same type as the
                        // `self` reference. To fix this, we would need to
                        // do a def-use analysis on the self reference to
                        // find every MIR local which refers to it that ends
                        // up being passed to a call terminator. This is not
                        // trivial to do at the moment.
                        let span = self.body.source_info(location).span;
                        self.emit_err(arg_ty, span);
                    }
                }
            }
```

```
            }
        }
}
```

Appendix C

Early finaliser prevention optimisation

The following shows the full early finaliser prevention optimisation, which removes compiler barriers which we can guarantee are unneeded. It is implemented as a MIR transform pass in ALLOY's modified version of the Rust compiler.

```
use crate::MirPass;
use rustc_middle::mir::*;
use rustc_middle::ty::{self, TyCtxt};
use rustc_span::sym;

use super::simplify::simplify_cfg;

pub struct RemoveGcDrops;

impl<'tcx> MirPass<'tcx> for RemoveGcDrops {
    fn run_pass(&self, tcx: TyCtxt<'tcx>, body: &mut Body<'tcx>) {
        trace!("Running␣RemoveGcDrops␣on␣{:?}", body.source);

        let is_gc_crate = tcx
            .get_diagnostic_item(sym::gc)
            .map_or(false, |gc| gc.krate == body.source.def_id().krate);

        let did = body.source.def_id();
        let param_env = tcx.param_env_reveal_all_normalized(did);
        let mut should_simplify = false;

        for block in body.basic_blocks.as_mut() {
            let terminator = block.terminator_mut();
            if let TerminatorKind::Drop { place, target, .. } = terminator.kind {
                let ty = place.ty(&body.local_decls, tcx).ty;
                let decl = &body.local_decls[place.local];
                if !ty.is_gc(tcx) {
                    continue;
                }

                if let ty::Adt(_, substs) = ty.kind() {
                    if !tcx.sess.opts.unstable_opts.gc_no_early_finalizers
                        || is_gc_crate
                        || !decl.is_user_variable()
                        || !substs.type_at(0).needs_finalizer(tcx, param_env)
                    {
                        terminator.kind = TerminatorKind::Goto { target };
                        should_simplify = true;
                    }
                }
            }
        }
        // if we applied optimizations, we potentially have some cfg to cleanup to
        // make it easier for further passes
        if should_simplify {
            simplify_cfg(tcx, body);
        }
    }
}
```