CONSTRUCTOR UNIVERSITY

Lama memory manager development

by

Egor Sheremetov

Bachelor Thesis in Computer Science

Submission: May 15, 2023

Supervisor: Prof. Kirill Krinkin

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Sheremetov, Egor
Matriculation number	30007021
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

.....

Date, Signature

Abstract

This bachelor thesis presents the development of an efficient memory management system for the Lama programming language. Lama, primarily used for teaching compilers to university students, is developed by JetBrains Research. The existing automatic memory management system in Lama exhibited inefficiencies in memory usage and object ordering. Additionally, the stop-the-world collection algorithm used in the previous implementation limited performance and efficiency. This thesis aims to address these challenges and improve memory consumption efficiency while preserving object ordering.

Drawing inspiration from the article "Reimplementing the Wheel: Teaching Compilers with a Small Self-Contained One" by Daniil Berezun and Dmitry Boulytchev, which explores the pedagogical aspects of teaching compiler construction, the significance of Lama as a teaching tool for compiler education is recognized. The article highlights the importance of understanding compilers for software engineers and the complexities involved in compiler design and construction. It emphasizes the need for a balance between comprehensive coverage of compiler topics and the robustness of a reference compiler implementation.

In line with the principles outlined in the article, this thesis focuses on building a complete and efficient memory management system for the Lama language. While the primary objective is the development of the memory management system, the relevance of Lama as a teaching tool in compiler education is acknowledged. By implementing a garbage collection algorithm that addresses the inefficiencies of the previous system and preserves object ordering, this research contributes to the advancement of memory management techniques in programming languages, particularly in the context of the Lama language.

The research methodology involves analyzing the limitations of the existing memory management system, reviewing various garbage collection algorithms, and designing a novel garbage collection algorithm tailored to Lama's requirements. The implementation prioritizes preserving object ordering, which is essential for maintaining the integrity of other runtime components in the Lama language. Furthermore, the efficiency of memory consumption is enhanced to optimize the overall performance.

To ensure the correctness and robustness of the new garbage collector, a comprehensive unit test suite is developed. The testing process encompasses both simplistic test cases and larger, more realistic programs, aligning with the principles outlined in the article. By thoroughly validating the garbage collector, this research strives to provide a reliable and efficient memory management system for the Lama language.

Contents

1 Introduction

Memory management plays a crucial role in programming languages and runtime environments, ensuring efficient and effective allocation and deallocation of memory resources. Over the last few decades, significant advancements have been made in the field of memory management, driven by the ever-increasing demands of modern software systems. This introduction delves into the background of memory management, highlighting its evolution and the challenges faced in achieving optimal memory utilization.

The efficient management of memory resources is essential for both performance and reliability in software systems. In early programming languages and systems, memory management was largely manual, requiring programmers to explicitly allocate and deallocate memory for their data structures. This manual approach was error-prone and often led to issues such as memory leaks and dangling references, causing stability and security concerns.

The advent of automatic memory management brought about a paradigm shift in software development. Automatic memory management, commonly known as garbage collection, relieved programmers from the burden of manual memory management by automating the process of memory allocation and deallocation. Garbage collectors employ various algorithms and strategies to identify and reclaim memory that is no longer in use, freeing programmers from the responsibility of explicitly releasing memory.

Over the years, researchers and practitioners have developed a multitude of garbage collection techniques to improve memory management efficiency. Initially, simple algorithms like reference counting were prevalent, which tracked the number of references to an object and deallocated it when the count reached zero. However, reference counting had limitations, such as its inability to handle cyclic references and the overhead of maintaining reference counts.

As the complexity of software systems increased, more sophisticated garbage collection algorithms emerged. These algorithms aimed to minimize memory overhead, reduce pause times, and improve throughput. Some notable garbage collection techniques include mark-and-sweep, generational garbage collection, concurrent garbage collection, and incremental garbage collection. Each technique introduced novel approaches to memory management, addressing specific challenges encountered in different application domains.

In recent years, there has been a growing focus on memory management in programming languages designed for educational purposes. Teaching compilers and programming language design has become an integral part of computer science curricula in universities worldwide. Providing students with a comprehensive understanding of compilers involves exposing them to memory management concepts and techniques.

This brings us to the Lama programming language, developed by JetBrains Research, which serves as a teaching tool for compiler education. Lama has an existing automatic memory management system; however, it suffers from inefficiencies in terms of memory usage and object ordering. Additionally, the previous garbage collection algorithm employed a stop-the-world collection cycle running in a single thread, which limited its sophistication and efficiency. Preserving the order of objects, an important invariant for other runtime components in the Lama language, was also a concern in the existing

implementation.

Therefore, the primary goal of this thesis is to implement a more efficient and ordered garbage collection system for the Lama programming language. By addressing the shortcomings of the current memory management system, this research aims to improve memory consumption efficiency while maintaining object ordering integrity. Furthermore, the development of a comprehensive unit test suite ensures the correctness and robustness of the new garbage collector.

By delving into the background and evolution of memory management, exploring various garbage collection techniques, and addressing the specific challenges faced in the Lama programming language, this thesis contributes to the advancement of memory management techniques in programming languages and compiler education.

The subsequent chapters of this thesis will delve into a detailed analysis of the existing memory management system in Lama, review relevant literature on garbage collection algorithms, present the design and implementation of the new garbage collector, and evaluate its performance and effectiveness. The research findings will not only benefit the Lama programming language but also contribute to the broader field of memory management in programming languages.

In summary, this thesis endeavors to enhance the memory management capabilities of the Lama programming language, combining theoretical knowledge and practical implementation to achieve an efficient and ordered garbage collection system. By doing so, it aims to improve the overall performance, reliability, and usability of the Lama language, particularly in the context of compiler education for university students.

Throughout the thesis, we will explore the evolution of memory management techniques, considering the challenges and advancements that have shaped the field. By examining the state-of-the-art garbage collection algorithms and their applicability to the Lama language, we aim to identify the most suitable approach for improving memory utilization and object ordering.

The implementation of a new garbage collection system involves not only designing efficient algorithms but also considering the specific requirements and constraints of the Lama language. We will focus on preserving the order of objects, as it serves as an important invariant for other runtime components in Lama. This necessitates careful consideration of memory allocation strategies, object traversal techniques, and the handling of cyclic references.

To ensure the accuracy and effectiveness of the developed garbage collector, a comprehensive testing process will be carried out. This will involve creating various tests, ranging from simple cases to more complex real-world scenarios, to thoroughly evaluate the performance of the memory management system. This testing approach aligns with the principles highlighted in the article ”Reimplementing the Wheel: Teaching Compilers with a Small Self-Contained One” by Daniil Berezun and Dmitry Boulytchev, emphasizing the importance of comprehensive testing in compiler education.

The insights and discoveries gained from this research extend beyond the Lama programming language. They have broader implications for memory management in different programming languages and compiler design. The knowledge and techniques acquired through this thesis will empower students and developers to design more efficient and reliable memory management systems across various programming languages.

By incorporating the concepts discussed in the aforementioned article, this research will contribute to the ongoing advancement of memory management techniques and best practices.

In conclusion, the primary objective of this thesis is to enhance the memory management capabilities of the Lama programming language by implementing an optimized and well-ordered garbage collection system. This involves exploring the historical background of memory management, analyzing existing techniques, and addressing specific challenges within the Lama language. Through these efforts, the aim is to improve the overall performance and usability of the language. The outcomes of this research will not only benefit the Lama programming language but also have a broader impact on the academic community and industry practitioners, driving advancements in memory management techniques in programming languages.

2 Statement and Motivation of Research

2.1 Statement

The present research aims to address the inefficiencies and limitations of the existing memory management system in the Lama programming language, developed by JetBrains Research. Lama's current automatic memory management system suffers from deficiencies in memory usage and object ordering, which adversely affect the language's performance and usability. This research project aims to implement a new garbage collection algorithm that not only rectifies these issues but also significantly improves testability, design, and performance of the memory management system.

2.1.1 Testability

One fundamental aspect that demands attention in the context of this research is the enhancement of testability specifically for the developed memory management system in Lama. The existing implementation suffered from a lack of proper unit testing, resulting in an unreliable and error-prone system. To address this limitation, a comprehensive testing framework will be developed, tailored to the specific requirements and characteristics of the implemented garbage collector. This testing framework aims to ensure the correctness and robustness of the memory management system developed for Lama. By establishing a systematic and standardized approach to unit testing, the performance, reliability, and efficiency of the garbage collector can be thoroughly evaluated. This rigorous testing approach not only serves to detect and eliminate potential bugs and performance bottlenecks but also provides a solid foundation for further extension and improvement of the memory management system, as highlighted in the undertaken research.

Moreover, preserving the order of objects during the garbage collection process is of paramount importance in terms of enhancing the performance of user code in Lama. The previous memory management system in Lama suffered from a lack of object order preservation, resulting in unpredictable object addresses and a significant impact on the runtime behavior of the language. By implementing a garbage collection algorithm that effectively maintains the order of objects, the new memory management system developed as part of this research ensures that objects allocated earlier always have lower addresses, preserving this crucial invariant for other runtime components of Lama. This preservation of object order enables more efficient memory access, improves cache locality, and ultimately optimizes the overall performance of user code. By effectively addressing this issue, the undertaken research aims to provide a memory management system in Lama that not only ensures correctness but also maximizes the performance of user programs, directly aligning with the goals and objectives of the conducted work.

The significance of robust and predictable testing cannot be understated when it comes to low-level software components like garbage collection, which inherently impacts the performance and stability of the overall system. In the case of the implemented garbage collector for Lama, the importance of thorough testing is even more pronounced, given the complexities involved in memory management and the critical role it plays in the language's functionality. With the new testing framework developed as part of this research, it becomes possible to evaluate the correctness and efficiency of the garbage collector in a controlled and systematic manner. By simulating various scenarios and edge cases, including object allocation, references, lifetimes, and garbage collection cycles, the testing framework validates the behavior and performance of the memory management system.

This meticulous testing approach not only helps in identifying and rectifying potential bugs and corner cases but also ensures that the system behaves as intended, ultimately contributing to the development of a reliable and efficient garbage collector for the Lama programming language.

Furthermore, testing plays a vital role in evaluating the performance and efficiency of the garbage collector. Memory management systems, including garbage collection algorithms, need to strike a delicate balance between minimizing memory consumption and maximizing execution speed. In the context of this research, the developed testing framework enables the measurement and analysis of different garbage collection algorithms and strategies. By carefully assessing their impact on memory usage, execution time, and other performance metrics, it becomes possible to identify potential bottlenecks, inefficiencies, and areas for optimization. The insights gained from testing facilitate the fine-tuning and refinement of the garbage collection algorithm, leading to improved performance and efficiency in terms of memory consumption. This optimization process directly contributes to the overall effectiveness and usability of the memory management system developed for the Lama programming language, further enhancing the achievements and objectives of the conducted research.

2.1.2 Design

Another critical aspect of this research is the improvement of the memory management system's design. The current implementation suffers from strong coupling between independent components, resulting in a lack of modularity and maintainability. To address this issue, it is crucial to decouple runtime functions from the internal workings of the garbage collector. Runtime functions should not have unnecessary knowledge about the garbage collector's internals and auxiliary information stored in object headers. By establishing a well-defined and modular architecture, we can enhance the system's maintainability, facilitate future improvements, and enable better code reuse.

Furthermore, as the garbage collection algorithm may evolve and adapt over time, it is essential to establish a universal mechanism for traversing object fields and objects themselves. This mechanism should be flexible and extensible to accommodate potential changes in the garbage collection algorithm. By implementing a standardized approach to object traversal, we can avoid code duplication and ensure that modifications to the garbage collector do not require extensive changes throughout the codebase. This design improvement will not only enhance the system's maintainability but also promote a more efficient and adaptable memory management system in Lama.

The research questions addressed in this project revolve around evaluating the impact of the new garbage collection algorithm on memory consumption and performance. Specifically, we seek to determine the extent to which memory savings can be achieved through optimized memory management techniques. Additionally, we aim to explore the feasibility of building an abstraction layer for the garbage collector, allowing for easy extensibility and potential parallelization of collection cycles. By addressing these research questions, we can gain valuable insights into the effectiveness and efficiency of the memory management system in Lama.

In conclusion, this research project aims to improve the memory management capabilities of the Lama programming language. By implementing a more efficient and ordered garbage collection system, the research seeks to address the existing limitations and in-

efficiencies in memory usage and object ordering. The project also focuses on enhancing testability, design, and overall performance of the memory management system.

2.2 Motivation

By developing a comprehensive unit testing framework, the research project aims to ensure the correctness and effectiveness of the new garbage collector. Proper testing of the memory management system will not only help identify and resolve potential bugs but also provide a reliable foundation for future extensions and improvements. This emphasis on testability contributes to the overall stability and robustness of the memory management system.

Preserving the order of objects in the garbage collection process is another key objective of this research. The previous memory management system in Lama lacked this essential feature, leading to unpredictable object addresses and impacting the performance of user code. By implementing a garbage collection algorithm that maintains the order of objects, we can optimize memory access, improve cache locality, and ultimately enhance the execution speed of user programs. This preservation of object order ensures a more efficient and predictable runtime behavior, benefiting the overall performance of the Lama programming language.

Improving the design of the memory management system is crucial to overcome the issues of strong coupling between independent components. The current implementation suffers from a lack of modularity and proper separation of concerns, hindering its maintainability and scalability. By decoupling runtime functions from the internal workings of the garbage collector and establishing a clear abstraction layer, we can enhance the system's design and promote better code organization. This design improvement enables easier maintenance, facilitates future enhancements, and contributes to the overall stability of the memory management system.

In addition, the research project recognizes the need for a universal mechanism for traversing object fields and objects themselves. This mechanism should provide a flexible and extensible solution that accommodates potential changes in the garbage collection algorithm. By implementing a standardized approach to object traversal, we can eliminate code duplication and ensure that modifications to the garbage collector do not introduce unnecessary complexity or impact the overall system performance. This design enhancement promotes a more maintainable and adaptable memory management system in Lama.

Through this research project, we aim to address the gap in the existing memory management system in Lama. The implementation of an efficient and ordered garbage collection algorithm, along with improvements in testability and design, will contribute to the overall performance and usability of the language. Furthermore, the findings and insights gained from this research will have broader implications for memory management in programming languages and compiler design.

The research questions to be explored include the evaluation of memory savings achieved through optimized memory management techniques and the feasibility of building an abstraction layer for the garbage collector. By investigating these questions, we can assess the impact and effectiveness of the new memory management system in Lama. Additionally, this research project aims to provide valuable knowledge and techniques that

empower students and developers to design more efficient and reliable memory management systems for a wide range of programming languages.

In conclusion, this research project seeks to enhance the memory management capabilities of the Lama programming language by implementing an efficient and ordered garbage collection system. By addressing the deficiencies in memory usage and object ordering, improving testability and design, and exploring research questions related to memory consumption and performance, we aim to contribute to the advancement of memory management techniques in programming languages. The outcome of this research will benefit both the academic community and industry practitioners, paving the way for more efficient and reliable memory management systems.

3 Description of the Investigation

3.1 Existing GC algorithms analysis

The investigation section of this thesis delves into the research and evaluation of existing garbage collection (GC) algorithms to determine the most suitable approach for our specific purpose in developing an efficient memory management system for the Lama programming language. This section focuses on understanding the properties and characteristics of different GC algorithms and their applicability to our project requirements. In particular, we place significant emphasis on preserving the order of objects during the garbage collection process, as it plays a crucial role in various aspects of Lama's runtime behavior and overall performance.

Preserving the order of objects holds paramount importance for our memory management system. There are several reasons why this property is critical to our goals. Firstly, it greatly influences the cache-friendliness of user code. By preserving object order, we can enhance cache locality, reducing cache misses and improving overall execution speed. Additionally, other runtime components within Lama rely on the invariant that the order of objects remains unchanged after garbage collection cycles. Therefore, maintaining this order is essential to ensure the correctness and integrity of the language's runtime environment.

To determine the most suitable GC algorithm for our purposes, we conducted a comprehensive comparison of different classical GC algorithms, including mark-sweep, copying collector, and reference counting. Each algorithm possesses its own strengths and merits, but we carefully evaluated their compatibility with our specific requirements using various criteria.

One important criterion for comparison is the allocation overhead for a mutator thread. The mutator thread is responsible for allocating new objects during program execution. In this context, the mark-sweep algorithm exhibits low allocation overhead since it does not require additional bookkeeping information during object allocation. However, it suffers from heap fragmentation due to its inability to compact memory after garbage collection cycles. The copying collector, on the other hand, provides efficient memory allocation by copying live objects to a separate space, but it incurs additional costs due to object copying and potentially disrupts the object order. Reference counting, while offering immediate reclamation of objects, introduces significant overhead in terms of maintaining reference counts for each object, impacting allocation performance.

Another criterion for comparison is the difficulty of runtime support. Some GC algorithms require additional runtime support mechanisms, such as read and write barriers, to track object references and ensure correct garbage collection. The mark-sweep algorithm, for instance, relies on write barriers to track changes in object references, adding complexity to the runtime system. In contrast, the copying collector and reference counting algorithms have less demanding runtime support requirements, as they do not rely heavily on barriers. However, the copying collector introduces the overhead of copying objects between memory spaces, while reference counting has its limitations in dealing with cyclic references.

Heap fragmentation is also an important aspect to consider. Fragmentation refers to the distribution of free memory blocks in the heap, which can impact memory utilization and allocation efficiency. The mark-sweep algorithm suffers from fragmentation as it does

not compact memory after garbage collection, resulting in scattered free memory blocks. The copying collector effectively eliminates fragmentation by copying live objects to a separate memory space but may incur additional costs in terms of memory usage due to maintaining two memory spaces. Reference counting typically does not suffer from fragmentation, but cyclic references can lead to memory leaks if not handled properly.

Considering these criteria, the mark-compact algorithm emerged as the most favorable choice for our memory management system. It offers a balanced trade-off by preserving object order, minimizing allocation overhead, and addressing heap fragmentation. The mark-compact algorithm systematically scans and marks live objects, followed by a compaction phase that eliminates fragmentation and organizes objects in a contiguous manner, preserving their original order. This property aligns perfectly with our objective of maintaining object order and facilitates cache-friendly execution of user code.

In the forthcoming sections of this investigation, we will delve deeper into the technical aspects of the mark-compact algorithm, its implementation considerations, and how it aligns with the requirements and goals of our memory management system. We will explore the intricacies of the mark-compact algorithm, its parallel and concurrent variants, and its potential application as a collector for old objects in a generational garbage collection scheme.

Furthermore, the investigation will examine the challenges and considerations associated with integrating the mark-compact algorithm into the existing runtime environment of the Lama programming language. We recognize that incorporating a new garbage collection algorithm requires careful design and implementation to ensure compatibility with the language's runtime components and minimize any negative impact on performance and memory consumption.

To facilitate a comprehensive evaluation, we will conduct experiments and simulations to measure and analyze the performance characteristics of the mark-compact algorithm under various scenarios and workloads. We will assess its efficiency in terms of memory consumption, execution time, and the impact on user code performance. These evaluations will provide valuable insights into the suitability and effectiveness of the mark-compact algorithm within the context of Lama's memory management requirements.

Moreover, this investigation aims to address not only the technical aspects of selecting and implementing a garbage collection algorithm but also the broader implications for the Lama programming language and its users. By enhancing the memory management system, we strive to improve the overall performance, reliability, and usability of the language. Efficient memory management is crucial for the execution of complex programs, resource utilization, and providing a seamless development experience for programmers using Lama.

In conclusion, this section sets the stage for the investigation by discussing the importance of preserving object order and outlining the criteria for comparing different garbage collection algorithms. Through a thorough evaluation of classical GC algorithms and their alignment with our requirements, the mark-compact algorithm emerges as a promising choice. The investigation will delve deeper into the technical details, challenges, and performance evaluation of the mark-compact algorithm, ultimately aiming to enhance the memory management capabilities of the Lama programming language and benefit its users.

In the subsequent pages, we will explore the design of experiments, simulations, and

implementation details of the mark-compact algorithm, shedding light on its advantages and addressing the specific challenges encountered during its integration into Lama's runtime environment. The investigation will provide valuable insights into the performance, reliability, and efficiency of the memory management system, paving the way for improved memory management techniques in programming languages and benefiting both the academic and industry communities.

3.2 Overview of Lama object's structure

3.2.1 Lama object types

In the Lama programming language, there are five fundamental types of objects: number, S-expression, closure, string, and array.

The Number type requires no additional introduction as it represents numerical values.

S-expression is a functional way of creating structures in Lama. It consists of fields and a name associated with it. For instance, if we want to create an S-expression `Structure` with two fields storing the numbers 42 and 9, it would be represented as follows: `Structure(42, 9)`. Fields can be either numbers or other complex types, and in the latter case, a reference to the actual object is stored as a field.

Closures and arrays share many similarities. The main distinction is that closures hold a reference to the code that should be executed when the corresponding closure is invoked. However, in general, both closures and arrays are sequences of elements. These elements can be either numbers or references to complex objects. In the case of closures, they represent the set of objects that need to be captured.

Strings in Lama are straightforward and represent sequences of characters.

3.2.2 Object header

To ensure the correct functioning of the garbage collector, additional information associated with each allocated object needs to be stored. This information is commonly referred to as an object's header. The header contains vital metadata that the GC utilizes to manage memory effectively. Depending on the requirements and optimization goals of the GC algorithm, the object's header can be stored separately from the actual object or as part of its memory layout.

Various approaches exist for storing object headers in different programming languages and GC implementations. One common method is to include the header within the memory space allocated for the object itself. In this approach, the header is typically located at the beginning of the object's memory block. This design ensures that the header is easily accessible and tightly integrated with the object's data. Languages like C and C++ often adopt this approach, as it allows for direct and efficient access to the object's metadata.

Alternatively, some GC algorithms separate the object's header from the actual object data. In such cases, the header is stored in a separate data structure, such as a hash table or an array, maintaining a mapping between the object and its associated metadata. This approach offers flexibility in terms of the size and structure of the header, as it is decoupled from the object itself. Languages like Java and C# commonly employ this technique to support features like garbage collection, reflection, and dynamic dispatch.

In the pursuit of further algorithmic improvements and optimizations, it is crucial to consider the placement of the object's header. The paper titled "A Fully Parallel LISP2 Compactor with Preservation of Sliding Properties" by Xiao-Feng Li, Ligang Wang, and Chen Yang delves into the importance of incorporating the header as part of the object header. It provides valuable insights into the benefits and implications of this design choice, emphasizing the preservation of sliding properties in the context of a fully parallel LISP2 compactor.

By exploring different approaches to storing object headers in various languages and GC implementations, we have carefully considered the trade-offs and design considerations involved. After careful deliberation, it has been determined that for our specific garbage collector implementation in the Lama programming language, we will adopt the approach of storing the object's header before the actual object data.

This decision aligns with the desire for a tightly integrated memory layout, where the header resides at the beginning of the object's memory block. By placing the header in close proximity to the object, we ensure efficient access to the metadata and maintain a cohesive structure. This design choice facilitates direct and straightforward manipulation of the object's metadata, enabling effective memory management and garbage collection operations.

Although other approaches, such as storing the header separately in a dedicated data structure, offer flexibility and decoupling of the metadata from the object data, we have determined that the benefits of an integrated header outweigh the potential advantages of a separate storage mechanism in our context. By storing the header before the object data, we establish a clear and predictable memory layout that aligns with the goals and requirements of our GC algorithm for the Lama programming language.

3.3 Designing the Garbage Collector Interface for Testing

One critical aspect of developing a reliable GC is the ability to thoroughly test its functionality, performance, and correctness. In this section, we will delve into the design considerations and strategies employed to create convenient and capable GC interface for testing.

When designing the GC interface, we need to strike a balance between providing flexibility for testing and maintaining the integrity of the GC's behavior in real-world scenarios. On one hand, we aim to create an environment where certain components, such as the stack, can be mocked or simulated to facilitate controlled testing scenarios. This allows us to isolate specific aspects of the GC's functionality and evaluate its behavior under various conditions. By mocking the stack, for instance, we can simulate different stack configurations and test the GC's ability to correctly identify and collect unreachable objects.

On the other hand, it is crucial to ensure that the GC's behavior in the test environment closely reflects its behavior in the actual runtime environment of the Lama programming language. Modifying the GC's behavior too extensively for testing purposes may lead to a disconnect between the test results and the GC's performance in real-world scenarios. Therefore, we strive to strike a balance where the testing environment mimics the runtime environment as closely as possible, allowing us to evaluate the GC's performance and effectiveness in a representative setting.

To achieve these objectives, we employ various techniques and design choices. First, we create a well-defined and standardized GC interface that separates the GC implementation from the rest of the runtime system. This interface encapsulates the essential operations and interactions between the GC and other components of the language runtime, such as the memory allocator and object manager. By decoupling the GC from these components, we enable easy swapping of different GC implementations and facilitate testing of individual GC functionalities in isolation.

Furthermore, we introduce configurable parameters within the GC interface that enable fine-tuning and customization of the GC behavior during testing. These parameters allow us to control aspects such as garbage collection frequency, heap size, and collection strategies, enabling us to explore different scenarios and evaluate the GC's performance under varying conditions. By providing this flexibility, we can conduct comprehensive testing and analysis of the GC's behavior and performance characteristics.

In addition to designing the GC interface, it is important to develop a comprehensive testing framework that encompasses both unit tests and real-world program simulations. The unit tests focus on specific functionalities and edge cases of the GC, while the program simulations aim to replicate real-life usage scenarios and assess the GC's performance under practical conditions. This multi-faceted testing approach ensures that the GC is thoroughly evaluated and validated across a range of scenarios and usage patterns.

As the GC is a critical component of the Lama programming language, the design of the GC interface for testing requires careful consideration and attention to detail. By balancing flexibility, maintainability, and consistency with the runtime environment, we can develop a robust testing infrastructure that empowers us to validate the GC's correctness, efficiency, and scalability. Through comprehensive testing, we gain confidence in the reliability and effectiveness of the GC, ultimately enhancing the overall performance and usability of the Lama programming language.

3.4 Executing Lama Runtime Functions with Virtual Stack

During the development of the testing framework for the garbage collector (GC), a significant challenge emerged in ensuring the proper emulation of the stack and the correct execution of Lama runtime functions. The issue stemmed from the use of GC functions alongside object allocation functions provided by the Lama runtime, such as `Bstring`, `Bsexp`, `Bclosure`, and `Barray`. The problem arose from the fact that some values on the stack were arbitrary data stored by the C compiler, such as callee-saved registers. In order to accurately replicate the behavior of the actual Lama stack, it was crucial to eliminate any random data and ensure that all objects stored on the stack were valid Lama objects.

To address this challenge, a virtual stack was implemented as part of the testing framework. The virtual stack was designed to hold objects that were programmatically placed onto it, allowing for precise control and emulation of the Lama stack. By utilizing the virtual stack, it became possible to eliminate any unwanted random data and ensure that the objects stored on the stack accurately represented valid Lama objects.

In addition to addressing the stack emulation issue, it was also necessary to execute Lama runtime functions within the context of the virtual stack. This was essential for accurately scanning and evaluating the behavior of the GC. To achieve this, a specialized function was developed in assembly language with the following signature:

`size_t call_runtime_function(void *stack, void *fptr, int num_args, ...)`

This function accepted a pointer to the current top of the virtual stack, a pointer to the Lama runtime function to be invoked, the number of arguments expected by the function, and the arguments themselves using a variable argument list. The function dynamically adjusted the program stack to use the virtual stack, executed the specified runtime function with the provided arguments pushed onto the virtual stack, and then reverted the stack back to its normal state to ensure seamless continuation of program execution.

To illustrate the usage of the virtual stack and the execution of Lama runtime functions, consider the following example test case:

Listing 1: Example test case using virtual stack

```
void test_simple_array_alloc(void) {
    virt_stack* st = init_test();

    // Allocate array [ BOX(1) ] and push it onto the stack
    vstack_push(
        st,
        call_runtime_function(vstack_top(st), Barray, 2, BOX(1), BOX(1))
    );

    const int N = 10;
    int ids[N];
    size_t alive = objects_snapshot(ids, N);
    assert((alive == 1));

    cleanup_test(st);
}
```

In this test case, the virtual stack is initialized at the beginning of the test using the `init_test()` function, and cleaned up at the end using the `cleanup_test()` function. Within the test, a Lama runtime function `Barray` is invoked with arguments `BOX(1)` and `BOX(1)`, and the resulting array is pushed onto the virtual stack. The `objects_snapshot()` function captures a snapshot of the objects in the GC and stores their identifiers in the `ids` array, allowing for verification of the expected number of alive objects. This example demonstrates the integration of the virtual stack and the execution of Lama runtime functions within the testing framework.

By enabling the execution of Lama runtime functions with the virtual stack, the testing framework facilitates accurate evaluation and analysis of the GC's behavior. It ensures that the GC functions correctly and consistently interact with the Lama runtime

References

[1] Daniil Berezun, Dmitry Boulytchev. “Reimplementing the Wheel: Teaching Compilers with a Small Self-Contained One”. July 2022.

[2] Richard Jones, Antony Hosking, Eliot Moss. “The Garbage Collection Handbook: The art of automatic memory management”.

[3] Xiao-Feng Li, Ligang Wang, and Chen Yang. “A Fully Parallel LISP2 Compactor with preservation of the Sliding Properties”. June 2014.