Gradual C Programming for Typed Lua

by

Rafi Shan Turas

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The work presented in this thesis deals with the problem of enhancing the performance of dynamically-typed programming languages by integrating features from statically-typed programming languages. Statically-typed languages focus on security and performance, but dynamically-typed languages focus on flexibility and automation. Optional type-systems and gradual type-systems realize some of the security benefits offered in statically-typed languages by adding a static type-system to a dynamically-typed language. However, these approaches generally do not provide the performance advantages of statically-typed languages.

In this thesis, a programming language named Poseidon Lua is proposed. It extends Typed Lua, an optionally-typed programming language, with language features that are only available from statically-typed languages. A Poseidon Lua program is able to use manual memory management to bypass the performance costs related to automatic garbage collection. A Poseidon Lua program is also able to use direct memory programming using its C pointers to avoid the performance overhead of using Lua tables. Note that Lua does not allow a program to directly manipulate raw memory. This thesis presents an extension of the compiler and virtual machine of Lua, named Modified Lua, that does allow the direct manipulation of raw memory. All Poseidon Lua programs are translated to Modified Lua programs before execution. In addition, for calling external C functions, a Modified LuaFFI library is provided for Poseidon Lua. The Modified LuaFFI library is an extension of the `luaffifb` library that avoids the performance overhead of the extra dynamic typechecking that is carried out by the `cdata` values. Poseidon Lua, Modifed Lua, and the Modified LuaFFI library are implemented by modifying the compiler of Typed Lua, the compiler and the virtual machine of Lua, and the `luaffifb` library, respectively.

Poseidon Lua is tested using a benchmark suite and a feature test suite. In the benchmark suite, Poseidon Lua programs achieve a speedup of 0.98X with respect to corresponding Lua programs and a speedup of 6.82X with respect to corresponding `luaffifb` programs, which is a Lua program that uses the `luaffifb` library. One Poseidon Lua program of the suite is able to achieve a maximum speedup of 10.76X with respect to the corresponding `luaffifb` program. In the feature test suite, relative to a Lua program, a Poseidon Lua program is able to achieve a speedup of 4.18X and 1.31X due to manual memory management and direct memory programming, respectively. A Poseidon Lua program that uses the Modified LuaFFI library is able to achieve a speedup of 10.32X over a `luaffifb` program. Poseidon Lua along with its components achieves significant performance advantages over the dynamically-typed language Lua using features from the statically-typed programming language C.

Acknowledgements

I would like to thank my supervisor, Prof. Gregor Richards, for his guidance, support, and encouragement throughout the duration of my research work. His excellent advice and direction was indispensable with regards to the completion of this thesis.

I wish to thank my readers, Prof. Peter Buhr and Prof. Werner Dietl, for their constructive feedback. Their feedback greatly improved my ability to clearly communicate some of the more subtle technical aspects of my thesis.

I would like to thank Prof. Dan Berry, Prof. David Toman, Prof. Robin Cohen, and Prof. Michael Godfrey, whose graduate courses I attended as part of my graduate studies. I appreciate their willingness to take the time, inside and outside the classroom, to discuss with me the various topics from their courses that I found interesting.

I wish to thank Prof. Joanne Atlee, Adjunct Prof. Nick Maes, and Continuing Lecturer Corey Van De Waal for their help in my pursuit of graduate studies.

I would like to thank my fellow students, teachers, and staff of the David R. Cheriton School of Computer Science.

I am thankful to my mother, Mrs. Salma Kader, for her support during the length of my studies. I am grateful to my father, Dr. Mohammed Ali, for his encouragement to pursue graduate studies. I highly appreciate the companionship of my little brother, Shwapneel, during my graduate studies.

Dedication

I dedicate this thesis to my parents and my little brother.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Most modern programming languages come with a type system. A type system is a technology from lightweight formal methods [22]. A type represents a set of values that is defined by a programming language. The type system of a programming language consists of a set of types as well as a set of type derivation rules. The type derivation rules can be used to derive a type for an expression of the programming language.

Typechecking is the act of enforcing the derived type for each expression of the programming language. Static typechecking refers to typechecking that is performed at compile-time and dynamic typechecking refers to typechecking that is performed at run-time.

A statically-typed programming language performs all typechecking at compile-time. Usually, such a programming language requires the programmer to supply static type-annotations for this purpose. This static typechecking restricts the run-time behavior of an expression of the language to guarantee that the expression always evaluates to a value that belongs to its derived type at run-time. As a result, a statically-typed programming language is able to provide strong security and maintainability assurances to the programmer.

A dynamically-typed programming language performs all typechecking at run-time. This dynamic typechecking guarantees that an expression of the language always evaluates to a value that belongs to its derived type at run-time. There are no restrictions placed on the expression at compile-time. The programmer is not required to provide any static type-annotations and all dynamic typechecking is automatically performed. As a result, a

dynamically-typed programming language is able to provide a great deal of flexibility to the programmer.

At the initial stage of a software project, the flexibility and automation of a dynamically-typed programming language is of high value to the programmer and enhances productivity greatly. However, as a software project matures, security and maintainability become more desirable qualities in a programming language. These qualities are already available in statically-typed languages. They can also be provided for a dynamically-typed programming language by attaching a static type system on top of it. This approach allows some static typechecking to take place for an otherwise dynamically typechecked program. Two popular methods for achieving this capability are optional type-systems and gradual type-systems.

An optional type-system adds a static type-system on top of a dynamically-typed programming language. The programmer supplies as much type annotations as they wish. These type annotations are used to perform as much static type checking as possible. The key characteristic of an optional type system is that it is not allowed to modify the run-time semantics of the underlying dynamically-typed program [3]. Thus, an optional type system cannot guarantee type soundness because at run-time, it cannot stop a statically-typed variable from being assigned a value that has a different type than the static type of the variable.

A gradual type-system also places a static type-system on top of a dynamically-typed programming language. It allows the programmer to provide as much type annotations as desired for the purpose of static typechecking. It is able to guarantee type soundness by inserting run-time checks between statically-typed code and dynamically-typed code to enforce static types at run-time [24]. Thus, a gradual type system is able to overcome the shortcomings of an optional type system. However, the extra run-time checks that are inserted by a gradual type system incurs a significant performance cost [25]. It is ultimately for the programmer to decide if the type soundness guarantee provided by a gradual type sytem is worth the extra cost in performance.

1.2 Motivation

Dynamically-typed programming languages have gained a great deal of popularity among software developers over the recent years. This popularity is principally due to the flexible programming style and extensive automation that is offered by these languages. However, as discussed earlier, security and maintainability become more of a concern than flexibility and automation when a software project matures and its codebase stabilizes.

Security and maintainability guarantees can be obtained through the addition of an optional type-system or a gradual type-system to a dynamically-typed programming language. Both optional and gradual type systems generate a program in the underlying dynamically-typed programming language as a final product. This means that their performance can only be as good as the performance of their underlying dynamically-typed language.

Research in this area has primarily focused on the security and maintainability benefits that could be gained through the use of increased static typechecking. However, the addition of a static type system on top of a dynamically-typed programming language also creates a new opportunity to improve performance as well.

Statically-typed programming languages tend to run faster than dynamically-typed programming languages [27]. This benefit is mainly due to the fact that statically-typed languages provide a variety of language features that help to cut down on performance costs. Dynamically-typed languages generally do not offer these features because they would undermine the flexibility of use and automation that is provided by these languages.

The work presented in this thesis investigates the integration of some of the features that are traditionally provided by statically-typed languages into an optionally-typed language. This investigation is done for the purpose of circumventing the performance limit that is imposed upon the optionally-typed language by its underlying dynamically-typed language.

This thesis explores the performance impact of integrating two such language features into an optionally-typed programming language: *manual memory management* and *direct memory programming*. Manual memory management gives the programmer control over garbage collection activities. Direct memory programming allows the programmer to directly manipulate data that is stored in memory. Both of these language features are available in the statically-typed programming language named C [12].

These language features are integrated into the optionally-typed programming language named Typed Lua that provides an optional type system for an underlying dynamically-typed programming language named Lua. Typed Lua allows the programmer to add static type annotations to an underlying Lua program, which are used to perform static typechecking. Typed Lua outputs the underlying Lua program as the final product.

Most of the dynamically-typed languages, including Lua, provide an automatic garbage collector to deallocate blocks of memory that are no longer used by a program at runtime. The automatic garbage collector performs its tasks without the involvement of the programmer. Therefore, it is able to avoid any memory errors that could have been introduced by the programmer. It keeps track of the blocks of memory that are in use by the program and the blocks of memory that are not in use by the program. Thus, it runs

periodically to update the status of the blocks of memory and to decide when to deallocate the unused blocks of memory. As a result, it incurs a performance overhead.

In contrast, manual memory management allows the programmer to decide when to deallocate blocks of memory. Manual memory management is implemented in the C programming language through two critical functions named malloc() and free(). The programmer is able to use the malloc() function to allocate a block of memory. The programmer is able to use the free() function to deallocate a block of memory.

Although there is a chance that the programmer may introduce memory errors, the performance overhead related to automatic garbage collection can be avoided. Manual memory management for Typed Lua is implemented using a similar mechanism to that used in C.

In this thesis, the term "direct memory programming" is used to refer to the ability of a programmer to directly manipulate memory without the need to go through an intermediate mechanism. One such intermediate mechanism is provided by Typed Lua in the form of the Lua table. It is the main data structure provided by the Lua programming language and it is implemented as a hash table by the Lua virtual machine (VM); consequently, a member access into a Lua table requires a hash table lookup [11]. As all the memory elements have to be accessed through Lua tables indirectly, it imposes extra performance overhead.

In contrast, the C programming language allows its programmer to directly manipulate the data that is stored at an arbitrary memory location without having to go through the interface of an intervening mechanism. This language feature is implemented through a special data type called a pointer. A pointer refers to a particular location in memory. A pointer can be dereferenced to access the data that is stored at the memory location that is referred to by the pointer. A pointer can also be used to store data at the memory location referred to by the pointer. In this way, a C programmer is able to use pointers to directly manipulate memory without having to go through an intervening mechanism. We intend to integrate a pointer mechanism along with other types from C into Typed Lua to implement direct memory programming in order to avoid the performance overhead of using Lua tables for memory manipulation.

When the new pointer mechanism and other types from C are integrated into Typed Lua, it also provides an opportunity to reduce the performance overhead related to dynamic typechecking of values returned from an external C function call done through a Lua foreign function interface (FFI) library.

1.3 Statement of Problem

We propose a programming language named Poseidon Lua that extends Typed Lua with manual memory management and direct memory programming features.

We implement manual memory management for Poseidon Lua in a similar manner to the way it is implemented in the C programming language. We add a `malloc` operator to Poseidon Lua, which can be used by the programmer to allocate a block of memory. We also add a `free` operator to Poseidon Lua, which can be used by the programmer to deallocate a block of memory. In this way, a Poseidon Lua programmer is able to make use of manual memory management when it is necessary.

We attempt to offer a more efficient alternative to Lua tables for memory manipulation. We do this through the implementation of direct memory programming for Poseidon Lua. This alternative is achieved by extending the static type system of Typed Lua with a subset of the type system of the C programming language including pointers. This modification allows a Poseidon Lua programmer to use C pointers to manipulate memory directly without having to go through an intermediate mechanism such as a Lua table. Therefore, our solution avoids the performance overhead related to the use of Lua tables.

A Typed Lua program is statically typechecked and then translated into a corresponding Lua program. Similarly, a Poseidon Lua program is statically typechecked and then translated into a corresponding Modified Lua program. Modified Lua is our extension of the Lua compiler and virtual machine. Modified Lua augments Lua with special C Semantics (CS) operators that allow C values to be handled from within a Lua program.

Note that a program that is written in a dynamically-typed programming language such as Lua is ultimately incapable of matching the performance of a program that is written in a statically-typed programming language such as C. This restriction is due to the various automated features that are provided by the dynamically-typed language and the way in which the implementation of the language is built around these features. For this reason, these languages generally find libraries that are written purely in C to be indispensable from a performance standpoint. As a result, these languages tend to provide a Foreign Function Interface (FFI) library that can be used by the programmer to call external C functions from within the dynamically-typed programming language. The `luaffifb` library is one such FFI library for Lua.

The main problem with the approach taken by the `luaffifb` library is in the way that it treats the value that is returned by the external C function. If the returned C value is of a primitive C type, `luaffifb` typechecks the returned value and converts it to a Lua value at run-time. Otherwise, if the returned C value is of a composite type such as a

C struct or a C pointer, `luaffifb` stores the returned C value within a `cdata` value. At each use-site of the returned C value, the `cdata` value performs dynamic typechecking to ensure the correct usage of the underlying C value, which incurs a recurring performance overhead. This is necessary because Lua is a dynamically-typed programming language and, as a consequence, does not accept static type annotations from the programmer.

Poseidon Lua does accept static type annotations for its C types. Thus, it can ensure that the value that is returned by an external C function always behaves in accordance with its C type by performing static typechecking only. Thus, it can avoid the recurring performance overhead at each use-site of the returned C value. In this way, Poseidon Lua is able to reduce the performance cost of using an FFI library.

Poseidon Lua comes equipped with the Modified LuaFFI library, which enables a C pointer that is returned from an external C function to be used within Poseidon Lua code without incurring any further performance costs from extra dynamic typechecking.

In this thesis, we explore the performance impact of integrating two language features, manual memory management and direct memory programming, which are normally only found in statically-typed programming languages such as C, into our proposed optionally-typed programming language named Poseidon Lua. Furthermore, we study the performance impact of using the C types of Poseidon Lua in conjunction with an FFI library.

The novel approach proposed in this thesis allows programmers of dynamically-typed languages to enjoy some of the performance benefits as offered by statically-typed languages.

1.4 Organization of the Thesis

The rest of the thesis is structured as follows.

Chapter 2 provides a review of the research work that is relevant to the work presented in this thesis. The language features of dynamically-typed programming languages and optionally-typed programming languages are covered. Next, we discuss the concept of type soundness. Then, the language features of gradually-typed programming languages are described. In addition, the language features of the dynamically-typed programming language Lua is given along with the language features of the optionally-typed programming language Typed Lua.

Chapter 3 provides an overview of our proposed programming language named Poseidon Lua. Poseidon Lua introduces 3 new operators: `sizeof`, `malloc`, and `free`. Poseidon

Lua introduces 4 primitive C types: `char`, `int`, `double`, and `bool`. Poseidon Lua also introduces 3 composite C types: C pointer type, C struct type, and C array type. In addition, Poseidon Lua includes a modification of the assignment operator of Typed Lua in order to work properly with the C types that are provided by Poseidon Lua.

In Chapter 4, we introduce Modified Lua, which is our extension of the Lua compiler and virtual machine. Modified Lua provides special operators to allow the manipulation of C values from within Lua.

In Chapter 5, we introduce the Modified LuaFFI library of Poseidon Lua. This library allows a Poseidon Lua program to call an external C function through an underlying FFI library and use the C pointer that is returned by the C function.

In Chapter 6, we show the performance impact of Poseidon Lua programs relative to Lua programs that use Lua tables and Lua programs that use a FFI library.

In Chapter 7, we provide the contributions of this thesis and possible directions for future work.

Chapter 2

Literature Review

2.1 Introduction

This chapter establishes the background context from which our work draws influence. We begin with a description of dynamically-typed programming languages. The overarching goal of these languages is to offer the programmer the maximum amount of flexibility and automation that is possible. They achieve this by automatically carrying out a lot of the tasks that would normally require the involvement of the programmer in statically-typed programming languages.

For example, two prominent features of dynamically-typed languages are dynamic typechecking and automatic garbage collection [27]. In general, statically-typed languages require the programmer to supply static type annotations for the purpose of performing static typechecking at compile-time. Some statically-typed programming languages do not require any static type annotations because they perform type inferencing. However, any type error that is reported by these languages may be time consuming for the programmer to track down and resolve due to the lack of explicit type annotations and the non-trivial nature of the type inferencing algorithm that is used. On the other hand, dynamically-typed languages avoid all of these issues altogether by forgoing static typechecking in favor of dynamic typechecking.

Some statically-typed languages, such as the C programming language [12], require the programmer to manually deallocate the memory blocks that are no longer used by the program. Dynamically-typed languages remove the involvement of the programmer from memory management matters by providing a garbage collector that automatically deallocates memory blocks that are no longer needed by the program at run-time.

It is common knowledge that the flexibility and automation provided by dynamically-typed languages is advantageous for programmers in the early stages of software development when the requirements are constantly changing and fast prototyping is desirable. However, in the later stages of software development, the requirements and the corresponding software become more stable. This makes maintainability and security very attractive qualities in a programming language. These qualities are more prevalent in statically-typed languages. Therefore, several techniques have been developed to provide static type systems for dynamically-typed languages. However, these techniques tend to emphasize security and maintainability. In contrast, our research focuses on improving the performance of dynamically-typed languages by integrating features from statically-typed languages.

For convenience of discussion, we have organized the contents of the chapter according to the following topics: Dynamic typing, Optional typing, Type soundness, Gradual typing, Lua, and Typed Lua.

2.2 Dynamic Typing

Dynamically-typed languages offer a different set of language features than those offered by statically-typed languages. Each of these language features comes with benefits as well as shortcomings when compared to its counterpart in statically-typed languages. Some of these language features facilitate a way of programming that is currently not possible in a statically-typed language.

Laurence Tratt [27] identifies the major defining features of dynamically-typed languages and traces the history of dynamically-typed languages. The author recognizes Lisp and Smalltalk as the earliest incarnations of dynamically-typed languages. Tratt states that Lisp, a functional programming language, is succeeded by Scheme, which has seen wide adoption in the research community. The author notes that Smalltalk, an object-oriented programming language, is succeeded in modern times by Python and Ruby. The author also notes that the class-based object model of Smalltalk has motivated the development of the prototype-based object model of the Self programming language. Tratt states that outside of Lisp and Smalltalk, the biggest influence on the development of dynamically-typed languages comes from the text processing languages. According to the author, Sed, an early text processing language, is succeeded by AWK and AWK is in turn succeeded by Perl. The author observes that these languages continue to enjoy much popularity due to their specialization for a particular application domain.

Moreover, Tratt identifies the advantages and disadvantages of dynamically-typed languages compared to statically-typed languages. The author states that dynamically-typed

languages generally provide highly optimized built-in data types such as lists and dictionaries. Tratt observes that statically-typed languages tend to require these to be implemented separately in libraries. The author notes that dynamically-typed languages provide automatic garbage collection. The author acknowledges that statically-typed languages are also starting to adopt this technology. The author states that dynamically-typed languages provide facilities for meta-programming. Tratt states that Lisp allows compile-time meta-programming via its macro definition feature. Tratt also states that Smalltalk allows run-time meta-programming through reflection-based programming. The author points out that statically-typed languages do not provide run-time meta-programming facilities of equivalent power. The author states that dynamically-typed languages offer a special feature named `eval` that allows a program to execute a string as a code fragment at run-time. The author notes that statically-typed languages generally do not provide this feature. Tratt observes that an important disadvantage for dynamically-typed languages in comparison to their statically-typed counterparts is slower performance.

Tratt also identifies the similarities and dissimilarities within different dynamically-typed languages. According to the author, one source of dissimilarities is object-orientation. Tratt observes that some dynamically-typed languages can be classified as Object-Oriented and others can be classified as Non-Object-Oriented. However, Tratt acknowledges that there also exists languages such as Python that started off as Non-Object-Oriented but added in Object-Oriented features over time. The author states that another source of dissimilarities is optional typing. Tratt observes that some dynamically-typed languages include an optional type system that is able to perform some compile-time typechecking depending on the amount of static type annotations provided by the programmer. Tratt states that optional type systems differ according to how much type annotation they consider to be mandatory. The author notes that aside from these differences, dynamically-typed languages tend to be very similar in terms of the general facilities that they provide.

Tratt notes that several features such as JIT-compilation, automatic garbage collection, and meta-programming were first pioneered in dynamically-typed languages and a lot of these features were later absorbed by statically-typed languages. The author also notes that it is reasonable to expect that many of the features that are unique to dynamically-typed languages today may be integrated into statically-typed languages at some point in the future.

A distinguishing feature of a programming language is its object model. There are two well-known object models: class-based object model and prototype-based object model. Class-based object models are generally found in statically-typed languages and prototype-based object models are generally found in dynamically-typed languages. Usually, the prototype-based object model is more flexible.

Ungar and Smith [28] identify the advantages of prototype-based object-oriented programming over class-based object-oriented programming. The authors present the design of the Self programming language. They state that Self uses the concept of prototypes for inheritance and object creation instead of classes. They note that Self also uses named slots in place of variables and procedures. They observe that Self removes the distinction between closures, procedures, and objects by allowing prototypes to model activation records.

Ungar et al. observe that objects are created in Self by copying a prototype instead of instantiating a class because Self does not have classes and the subclass relationship between classes. The authors note that this simplifies the relationship between objects. They state that Self can express object-oriented idioms as well as one-of-a-kind objects. They observe that Self also has facilities for inline objects as well as active values.

Ungar et al. claim that Self provides a fresh understanding of the object-oriented programming paradigm due to its expressive and simple language features.

Many dynamically-typed programming languages provide the facilities to inter-operate with statically-typed languages for performance reasons. Usually, the statically-typed language of choice is the C programming language and the inter-operation happens over a C API. However, care must be taken in order to avoid unintended errors during inter-operation.

Muhammad and Ierusalimschy [21] deal with the tradeoffs that can be made in the design of the C API for a scripting language to make it more efficient as an embeddable language (where the C code uses the scripting language) or as an extensible language (where the scripting language uses the C code). The authors identify data transfer, garbage collection, and function registration and calls as the key dimensions for tradeoffs in the design of the C API for a scripting language and describe these dimensions. The authors note that a scripting language represents data in a different way than C. Thus, they emphasize that the C API of the scripting language must provide facilities to manipulate data safely when data is transferred between the two languages. The authors note that a scripting language likely uses automatic garbage collection whereas C uses manual memory management. They observe that the C API of the scripting language must provide facilities for each language to handle data in a way that avoids memory errors. Finally, they assert that the C API of a scripting language must offer facilities to register and call functions from one language to another, which includes facilities to send parameters properly for a function call and receive results from function calls.

Muhammad et al. perform analysis and comparison of the design of the C APIs of 4 scripting languages: Python, Perl, Ruby, and Lua. They also consider the C API for

Java in order to understand the impact of static typing on the design of C APIs. They note that the C API that is provided by a scripting language must allow data to be manipulated in an error-free way. They observe that Python, Ruby, and Perl offer direct handles to their objects to C for manipulation. They also observe that Lua only allows indirect manipulation of its objects through stack indices. The authors note that Java uses the correspondence between its own static types and the static types of C to reduce the amount of data conversions that need to take place.

Muhammad et al. recognize that it is important for a scripting language to hide the implementation details of its garbage collection algorithm so as to prevent the C code from inadvertently introducing memory errors. The authors find that Python and Perl use reference counting algorithms and they are unable to adequately hide the use of their reference increment and reference decrement operations in their C APIs. The authors observe that Ruby uses a mark-and-sweep algorithm but it is also unable to hide this when certain kinds of Ruby objects are created from within C. They point out that Lua uses an incremental garbage collection algorithm and is able to hide it relatively well. They note that Java succeeds in completely hiding the implementation details of its garbage collection algorithm.

Muhammad et al. are of the opinion that the ease with which function calls can be made has a direct impact on the ability of a scripting language to inter-operate with C code. They find that Python, Perl, and Lua allow functions to be represented as objects that can be readily called. They observe that Java and Ruby do not consider methods to be first-class objects, and these languages provide special C types to represent methods.

Muhammad et al. implement a C library called LibScript to study and compare the C APIs of the 4 scripting languages. They find that Python's C API provides more functions geared towards higher convenience whereas Lua's C API provides less functions but is simpler to use. They also observe that Python and Lua as well as Java provide C APIs that define their function and C type names with prefixes in order to avoid C namespace pollution; however, Perl and Ruby provide C APIs that define their function and C type names in an arbitrary manner.

Muhammad et al. claim that when the C API of a scripting language aims to support both embedding and extension, it should focus more on embedding since the needs of extension support is covered by the needs of embedding support as well. They also note that a variety of tools already exist to deal with issues related to extension support. Therefore, they argue that putting more of an emphasis on embedding support should not be a problem.

When a dynamically-typed language uses its C API, it is unable to use the static type

information from the C code. This information could potentially be used to detect a larger number of type errors at compile-time than would otherwise be possible.

Klint, Roosendaal, and van Rozen [13] address the problem of ensuring high quality for Lua scripts in the absence of proper static analysis tools. The authors note that the data types and function signatures of the interface to a game engine can be utilized to enable better static analysis of embedded Lua scripts. They observe that the static types of the interface to the game engine restrict the ways in which Lua code may use the facilities that are made available to it by the game engine. They report that the functions that are provided by the game engine may only be called with arguments that are of certain types and these functions are also only allowed to return results that are of certain types. Therefore, the authors argue that the static types of these functions could be utilized by a static analysis tool to detect errors in Lua scripts.

Klint et al. propose a new framework called Lua AiR (Analysis in Rascal) that works alongside the Eclipse IDE to create an abstract syntax tree (AST) for a Lua script in order to perform various types of analysis on the AST. They note that the integration of Lua AiR into the Eclipse IDE creates an opportunity for other pre-existing tools to use the information that is generated by Lua AiR to enhance the quality of their own services. They also note that type inferencing could be used to expand Lua AiR's ability to catch errors in Lua scripts.

Klint et al. intend to use Lua AiR on existing game-related codebases on a larger scale. They hope to provide a clearer sense of the magnitude of the impact that Lua AiR can have on the productivity of programmers. They state that it may inspire the development of other tools that could also make use of the static type information from the interface to the game engines.

A crucial feature of dynamically-typed languages is that they automatically perform dynamic typechecking at run-time without the involvement of the programmer. However, it is necessary to insert these dynamic typechecks into the program in a systematic manner in order to avoid introducing more dynamic typechecks than are actually needed.

Henglein [8] proposes an explicitly dynamically typed language for an implicitly dynamically typed language. The author presents the dynamically typed λ-calculus as an extension of the statically typed λ-calculus. The author describes that the dynamically typed λ-calculus includes a new type named `Dyn`, which represents values that have been tagged with a type constructor, and dynamic type coercions, which represent operations to tag and untag (after checking) a value with a type constructor. The author also presents the concept of completions as a way to translate a term of the untyped λ-calculus to a corresponding term of the dynamically typed λ-calculus through the insertion of dynamic

type coercions as well as type annotations for variables where appropriate.

We now consider a different perspective on typechecking. Normally, statically-typed languages perform all their typechecking at compile-time. However, there may be situations where the type of a value may be unknown at compile-time. In such situations, a statically-typed language may have to perform dynamic typechecking at run-time to avoid type errors.

Abadi, Cardelli, Pierce, and Plotkin [1] address the problem of adding dynamically-typed values to a statically-typed language for the purpose of providing strong typing guarantees in an environment where the type of the values may be statically unknown. The authors present an extension of the Simply Typed Lambda Calculus that includes a new type named `Dynamic`, which represents dynamically-typed values, where a certain value that belongs to the `Dynamic` type is a pair `(VAL, TAG)` whose first member is a value `VAL` and whose second member is the type tag `TAG` for the type of `VAL`.

Abadi et al. introduce 2 new constructs in relation to the `Dynamic` type. They propose a `dynamic` construct, which can be used to create new values of the `Dynamic` type. They also propose a `typecase` construct, which can be used to analyse the content of a value of the `Dynamic` type and select an expression to evaluate based on that content. The authors present an operational semantics as well as a denotational semantics for the calculus. The authors anticipate that future programming languages will include types that are similar to the proposed `Dynamic` type as a standard feature.

2.3 Optional Typing

In statically-typed languages, the programmer has to resolve all reported type errors at compile-time. This is inconvenient for the programmer, but it provides strong security and maintainability guarantees. On the other hand, in dynamically-typed languages, the programmer only needs to resolve the type errors that are reported at run-time because dynamically-typed languages perform all typechecking at run-time automatically. Thus, dynamically-typed languages are flexible and provide enhanced productivity.

In order to provide better security and maintainability guarantees in dynamically-typed languages, static typechecking at compile-time can be added as an option on top of the dynamic typechecking that already occurs at run-time. This arrangement is known as an optional type system.

Gilad Bracha [3] tackles the issue of enhancing the expressibility and security of programming languages using pluggable optional type systems instead of mandatory type

systems. The author proposes that instead of focusing on the difference between static and dynamic type systems, everyone should focus on the difference between mandatory and optional type systems. The author observes that mandatory type systems are brittle and not secure due to there being no formal proof of correctness for most of them; therefore, these systems can pose a security problem when their assumptions fail. The author describes the problem of mandatory type systems. The author observes that statically-typed programming languages normally use mandatory type systems. However, the author states that if the internal logic of one of these type systems is faulty, then a security guarantee cannot be provided by that type system. Bracha notes that without a proof of correctness, the internal logic of any of these type systems cannot be shown to be consistent. Therefore, according to the author, all the time and effort spent towards making a program conform to a mandatory type system may be wasted.

Bracha proposes that programmers use multiple optional type systems that each check for a different type system as opposed to one mandatory type system. The author describes the characteristics of an optional type system. Bracha observes that an optional type system allows a programmer to choose which type annotations to provide. The author states that an important characteristic of an optional type system is that it may not alter the dynamic semantics of the underlying dynamically-typed programming language. Bracha explains that this requirement means that each optional type system can be plugged on top of each other to form a pluggable type system. Therefore, according to the author, a pluggable type system composed of multiple optional type systems can provide a richer typechecking service than a mandatory type system.

Bracha claims that a pluggable type system is capable of supplying most of the benefits that a programmer would expect from a mandatory type system.

An optional type system does not add any new dynamic typechecks at run-time. As a result, there is no way to prevent a statically-typed variable from being assigned a value that is inconsistent with its static type at run-time. Thus, an optional type system cannot guarantee type soundness. However, such a type violation should be detected in order to give the programmer a chance to rectify the error for the proper functioning of the program.

Lehtosalo and Greaves [14] address the problem of monitoring runtime type errors of optional pluggable type systems. The authors propose an optional runtime type error monitoring system that inserts runtime checks that log type violations but otherwise preserve the original dynamic semantics of the program. The authors explain that the runtime checks are implemented using wrapper objects or wrapper functions that track type violations that may occur when values cross the boundary between statically typed code and dynamically typed code.

Lehtosalo et al. present a formal model for an extension of Featherweight Java (FJ) named FJ$^?$, where their model incorporates support for a pluggable type system as well as optional monitoring of runtime type errors.

An optional type system can be used to provide static typechecking facilities for programming languages from any programming paradigm. Moreover, an optional type system can integrate types from different programming paradigms.

Bonnaire-Sergeant, Davies, and Tobin-Hochstadt [2] explore the issue of providing an optional type system for a functional programming language that has support for interoperability with an object-oriented language. The authors present an optional type system for Clojure named Typed Clojure and construct a formal model for Typed Clojure. They describe the features of Typed Clojure. The authors note that Typed Clojure includes an adaptation of the occurrence typing feature from Typed Racket. The authors explain that Typed Clojure also provides facilities to support the ability of Clojure to inter-operate with Java, which is an object-oriented language. They elaborate that for the purpose of Java interoperability, Typed Clojure provides exception-based control flow analysis to avoid null-pointer exceptions. They point out that it also provides heterogeneous dictionary types and multimethods.

Bonnaire-Sergeant et al. claim that Typed Clojure is already being used broadly in the Clojure community. They conduct a case study where they analyse a software system that is developed by a third-party using Clojure and Typed Clojure.

Bonnaire-Sergeant et al. plan to extend Typed Clojure to implement gradual typing in the future. The authors intend to make use of language features that are already available from Java and Clojure for this purpose.

2.4 Type Soundness

In a sound type system, a type error cannot be left uncaught. Statically-typed programming languages ensure type soundness by typechecking at compile-time. On the other hand, dynamically-typed programming languages ensure type soundness by typechecking at run-time. The problem of type soundness arises when a static type system is imposed on top of a dynamic type system while keeping the dynamically-typed language unaware of the static type system because the static type system does not perform the required dynamic typechecking related to its static types at run-time.

Mezzetti, Moller, and Strocco [20] address the problem of empirically validating the use of unsoundness in the type system of programming languages. The authors conduct an

experiment to investigate the unsoundness of the type system for the Dart programming language. They identify 10 sources of unsoundness and implement corresponding sound alternatives. First, they test whether the type system outputs a higher volume of warnings when they switch from each source of unsoundness to its sound alternative. Next, they perform a qualitative analysis of the warnings that are output. Finally, they test the impact of the set of sound alternatives that are selected on the runtime errors. They find that unsoundness in Dart is not justified in the case of bivariant function subtyping and method overriding.

Mezzetti et al. claim that some sources of unsoundness in Dart can be justified, but not all of them. The authors state that it may be beneficial to investigate how the productivity of programmers can be affected by alternative design choices for the type system of Dart. The authors also wish to evaluate different designs for the standard library of Dart to make it more type sound.

Most of the dynamically-typed languages support higher-order functions, which are functions that can take another function as an input argument. Since higher-order functions can take as an input argument another higher-order function, ensuring type soundness in languages that allow these kinds of functions can be challenging.

Findler and Felleisen [5] examine the issue of providing assertion-based contracts for higher-order functions in programming languages. The authors present a typed lambda calculus named λ^{CON} that includes support for assertion-based contracts for higher-order functions. The authors prove various properties of the calculus including type soundness. They also use examples to demonstrate their assertion monitoring system in DrScheme.

Findler et al. claim that assertions for higher-order functions allow programmers to specify contracts that they are unable to express using existing type systems. The authors hope that further research into assertions will lead to more practical type systems.

A sound type system has the ability to detect and locate type violations at run-time. However, the location in the code where the type violation occurs is usually different from the location in the code where the type violation is ultimately detected. Thus, there should be a system to link the two locations. This is called a blame tracking system.

Wadler and Findler [31] tackle the problem of providing blame tracking for a language with explicit casts. The authors present their blame calculus. They explain that the blame calculus includes a dynamic type named `Dyn` that represents values whose type may be unknown at compile-time. They further elaborate that it also includes subset types that augment a base type with a refinement predicate to narrow down the set of possible values to a subset of the set of values represented by the base type. They note that the calculus also includes explicit casts that have blame labels attached to them. They point out that

when a cast fails at run-time, blame is allocated to the blame label that is attached to that cast.

Wadler et al. present the concepts of positive blame and negative blame: positive blame occurs when a cast fails due to a violation from the term being casted, and negative blame occurs when a cast fails due to a violation from the surrounding context. The authors also present the concepts of a positive subtyping relation and a negative subtyping relation. They show that casting from a positive subtype to its supertype cannot cause a positive blame. They also show that casting from a negative subtype to its supertype cannot cause a negative blame.

Wadler et al. utilize the positive subtyping relation and the negative subtyping relation to show that well-typed terms of the blame calculus cannot generate blame. Therefore, the authors observe that in the case of an interaction between a well-typed term and a dynamically-typed term, a cast failure always results in the dynamically-typed term being blamed.

2.5 Gradual Typing

Like an optional type system, a gradual type system also provides a static type system for a dynamically-typed language. However, gradual type systems are introduced in order to overcome the lack of type soundness of optional type systems. In a gradual type system, typechecking happens both at compile-time and at run-time. The run-time portion of the typechecking for the static types is performed at the boundary between statically-typed code and dynamically-typed code to check for static type violations.

Siek and Taha [24] address the issue of providing a type system for a functional language that allows the programmer to control the amount of static typechecking versus dynamic typechecking. The authors present an extension of the simply-typed λ-calculus named the gradually-typed λ-calculus. The authors explain that the gradually-typed λ-calculus includes a dynamic type that is represented by the symbol "?". The authors further elaborate that the type system for the calculus includes a type-consistency relation that allows a function application to pass static typechecking even when the type of the function or the type of the argument is unknown. The authors state that the dynamic semantics for the calculus is provided through a translation to an intermediate language with explicit cast insertion.

Siek et al. prove that for a fully annotated term, the type system of the gradually-typed λ-calculus assigns the same type as the type system of the simply-typed λ-calculus. Next,

they prove that for a fully annotated term, the translation to the intermediate language results in no casts being inserted. They also prove that when a program terminates, it either generates a cast error or it generates a value that belongs to the proper type.

Siek et al. plan to integrate their gradual type system into a dynamically-typed language. They hope to measure the magnitude of its impact on programmer productivity.

Programming languages from many different paradigms can be appended with a gradual type system. However, each paradigm poses its own set of challenges when it comes to the implementation of a gradual type system.

Siek and Taha [23] tackle the issue of providing a gradual type system for an object-oriented language that allows the programmer to control the amount of static typechecking to be performed in contrast to the amount of dynamic typechecking to be performed. The authors present an object-oriented calculus named $\mathbf{Ob}^{?}_{<:}$. They state that the $\mathbf{Ob}^{?}_{<:}$ calculus incorporates a dynamic type, "?", to indicate a value whose type is unknown at compile-time. The authors explain that this calculus also includes the Consistent-Subtyping relation that allows method invocations to pass static typechecking in the case where the type of the method or the type of the argument may be unknown. The authors note that the Consistent-Subtyping relation combines the Subtyping relation with the Type-Consistency relation in a manner that avoids placing the dynamic type, "?", at the top of the type hierarchy. According to the authors, this approach allows the detection of some subtle errors during static typechecking that would have originated from the up-casting mechanism of the Subtyping relation. The authors give the dynamic semantics for the calculus via translation to an intermediate language with explicit casts.

Siek et al. prove that no program belonging to the $\mathbf{Ob}^{?}_{<:}$ calculus causes any type errors at run-time. They also prove that no fully annotated program belonging to the $\mathbf{Ob}^{?}_{<:}$ calculus causes any type errors or any cast errors at run-time.

Siek et al. plan to explore the manner in which `Hindley-Milner` inference would work in conjunction with gradual typing. They also wish to study whether the amount of dynamic typechecks could be further decreased using static analysis.

As stated earlier, gradually-typed languages use run-time checks to enforce the static type requirements of the programmer. The main problem is to identify how and where to provide the run-time checks. Each choice has its own advantages and disadvantages.

Vitousek, Kent, Siek, and Baker [29] address the issues related to the semantics of the run-time casts in gradually-typed languages. The authors introduce a gradually-typed version of Python named Reticulated Python. They state that Reticulated Python is a source-to-source translator that accepts type annotated Python code and performs static

typechecking and outputs Python 3 code with run-time casts inserted for dynamic typechecking.

Vitousek et al. implement a Python library that encapsulates the behavior of the run-time casts. They implement 3 forms of run-time cast semantics for Reticulated Python: the `guarded` semantics uses proxies and does not preserve object identity, the `transient` semantics does preserve object identity but is unable to perform proper blame tracking in case of an error, and the `monotonic` semantics also preserves object identity but requires the type of an object to be monotonically locked to the most precise type that has ever been ascribed to that object.

Vitousek et al. perform a case study where they type annotate several pieces of third-party software and run them using Reticulated Python. They find several bugs in the third-party software. They observe that the `transient` and `monotonic` cast semantics show acceptable performance; however, the `guarded` semantics perform poorly due to its use of proxies that do not preserve object identity. The authors claim that they would have been able to statically type a larger proportion of Python code if they had implemented support for generics into the type system for Reticulated Python.

The main similarity between optional and gradual type systems is that both perform typechecking at compile-time. On the other hand, the main dissimilarity between the two is that only gradual type systems perform run-time typechecking. Therefore, it may be more efficient to create a gradual type system by extending a pre-existing optional type system rather than to develop the gradual type system from scratch.

Vitousek and Siek [30] address the issue of transforming an optionally-typed language cleanly into a gradually-typed language by adding run-time casts in an uncomplicated manner. The authors observe that many dynamically-typed languages already have optional type systems that are available for them; however, these optional type systems do not insert the run-time casts that are necessary for the implementation of a gradual type system. The authors discuss two different semantics for the run-time casts that could be deployed as part of a gradual type system: the `guarded` semantics uses proxies to encapsulate run-time casts, and the `transient` semantics uses explicit casts instead of proxies. The authors share the results from implementing the two casting semantics for Reticulated Python, a framework that can be used to test gradual type systems for Python. They observe that it is easier to implement the `transient` semantics for Python than to implement the `guarded` semantics for Python because it is difficult to add support for proxies to Python.

Vitousek et al. note that the optional type systems for many of the dynamically-typed languages already implement the static typechecker and other facilities that are required

for a gradual type system. Due to this, they argue that it should be relatively easy to add support for the actual run-time casts that use the `transient` semantics. They implement CheckScript by adding run-time checks with the `transient` semantics to TypeScript, an optionally-typed language that compiles to JavaScript. The authors expect that it will be possible to transform other optionally-typed languages into gradually-typed languages through the addition of run-time casts with `transient` semantics as well.

As a gradual type system for a dynamically-typed programming language ensures type soundness through the use of run-time checks, it introduces extra performance costs. For a gradual type system to be useful, it must keep these performance costs under an acceptable limit.

Takikawa, Feltey, Greenman, New, Vitek, and Felleisen [25] tackle the issue of evaluating the performance of gradually-typed languages. The authors propose a new method to measure the performance of gradually-typed languages and describe the method in their paper. First, they divide each benchmark program into its constituent modules. Next, they create Typed and untyped versions of each module. Then, they form each individual configuration by combining different typed and untyped modules. Then, they organize all the configurations into a lattice with the fully untyped configuration at the bottom and the fully typed configuration at the top. Finally, they calculate the performance metrics for each configuration relative to the fully untyped configuration.

Takikawa et al. apply their new method to 12 selected Typed Racket benchmark programs. They find that their selected Typed Racket benchmark programs perform very poorly, even under very loose restrictions. The authors claim that better performance evaluation gained from the use of their method will result in future improvements to gradually-typed languages.

2.6 Lua

The work presented in this thesis involves the integration of the language features of C into Lua, a dynamically-typed programming language. Understanding the motivation behind the development of Lua makes it easier to comprehend the design of its features.

Ierusalimschy, de Figueiredo, and Celes [10] survey the history of the development of Lua. According to the authors, the development of Lua originated in the Computer Graphics Technology Group (Tecgraf) at PUC-Rio. The authors report that Tecgraf had created 2 languages, DEL and SOL, for Petrobras. They state that DEL was developed for data-entry purposes and SOL was developed for configuration purposes. They note that

as the need for more powerful language features became apparent, the decision was made in 1993 to replace DEL and SOL with a new language named Lua.

Ierusalimschy et al. observe that Lua has been successful in the gaming industry. The authors identify the main reason for this success as the ease with which Lua can be embedded in game engines.

As stated earlier, the work presented in this thesis involves adding language features from C to Lua that perform better than the pre-existing features of Lua. The implementation of Lua provides some indication of how this performance boost can be achieved.

Ierusalimschy, de Figueiredo, and Filho [11] survey the implementation of the main features of Lua 5.0. The authors identify a register-based virtual machine, an algorithm to implement tables as arrays, and the implementation of closures and coroutines as new features in Lua 5.0 when compared to Lua 4.0.

Ierusalimschy et al. state that Lua has had a stack-based virtual machine since it was first released. They observe that Lua has switched to a register-based virtual machine starting with Lua 5.0 because a register-based virtual machine avoids executing extra stack manipulation instructions that are prevalent in stack-based virtual machines, which leads to a smaller code size. The authors claim that Lua is the first widely used language that has adopted a register-based virtual machine.

Ierusalimschy et al. state that the implementation of Lua tables has been changed to include an array component in addition to a hash table component in order to accommodate consecutive Lua `integer` keys, if any. The authors state that this change of implementation saves space since the array component does not store its keys and is more efficient to use since there is no need to hash the keys for a Lua table lookup.

2.6.1 Typed Lua

Typed Lua provides an optional type system for Lua.

Maidl, Mascarenhas, and Ierusalimschy [18] address the issue of adding a static type system to a dynamically-typed programming language. The authors present the design of an optional type system for Lua named Typed Lua. According to the authors, Typed Lua preserves some of the idioms that are commonly used by Lua programmers. The authors state that Typed Lua does not insert run-time checks as most gradual type systems do. However, the authors note that Typed Lua does contain the mechanisms to be transformed from an optional type system to a gradual type system in the future.

Maidl et al. describe the types provided by Typed Lua. They explain that Typed Lua provides first-class types for normal Lua values and second-class types for expression lists that are used in multiple assignment and function application. The authors state that all first-class types are subtypes of the `value` type. They point out that there is also an `any` type that represents dynamically-typed code. According to the authors, Typed Lua includes `table` types to represent Lua tables. They further elaborate that Typed Lua provides specialized syntax for indicating `table` types that represent maps, arrays, and records. The authors note that Typed Lua provides the facility to declare a named interface that represents a `table` type that represents a record.

Typed Lua provides types that are intuitive to use for Lua programmers.

Maidl, Mascarenhas, and Ierusalimschy [19] tackle the issue of providing an optional type system for a dynamically-typed language, that contains types for the most commonly used idioms of the target language. The authors perform a survey of codebases from the LuaRocks repository to determine the idioms that are most commonly used by Lua programmers. They collect data regarding the use of tables, the use of object-oriented programming, the use of modules, and the use of overloaded functions. Then, they incorporate types corresponding to these idioms into an optional type system for Lua named Typed Lua.

Maidl et al. present a formal description of Typed Lua. The authors state that Typed Lua provides support for incremental construction of records and objects. They also observe that Typed Lua provides support for projection types to deal with functions that can return multiple values. The authors claim that the features of Typed Lua can be integrated into the type systems for other languages.

2.7 Conclusions

In this chapter, we give an overview of the methods that investigate how static typechecking can be integrated into a dynamically-typed programming language. We start with a description of the language features that are common to dynamically-typed programming languages and note that they differ significantly from the features that are common to statically-typed programming languages. For example, these two categories of programming languages differ in their handling of garbage collection and run-time metaprogramming.

We follow two historically distinct strategies for integrating static typechecking into a dynamically-typed language: optional type systems and gradual type systems. An optional

type system performs static typechecking only [3]. In contrast, a gradual type system performs static typechecking as well as dynamic typechecking to enforce static types at run-time [24]. The goal of these type systems is to import some of the security benefits of statically-typed languages to dynamically-typed languages.

In this thesis, we expand on this line of research with a focus towards importing some of the performance benefits of statically-typed languages to dynamically-typed languages.

In the rest of the thesis, we discuss how some of the language features of C, a statically-typed language, can be integrated into Typed Lua to provide some performance benefits. Specifically, we describe how the language of Typed Lua can be extended to include support for manual memory management. We also describe how the static type system of Typed Lua can be extended with C types to allow direct memory programming using C pointers. In this way, we aim to bring some of the performance benefits of C to Typed Lua, and, by extension, to Lua as well.

Chapter 3

Poseidon Lua

3.1 Introduction

Statically-typed programming languages make use of a variety of language features to minimize run-time performance overhead. One such feature is manual memory management, which allows the programmer to deallocate memory created by the program. This feature allows the programming language to avoid the use of an automated garbage collector and its associated performance costs.

Another feature of note is direct memory programming. We define the term "direct memory programming" to mean that a program is able to manipulate the memory directly without having to go through an intermediate mechanism such as a Lua table. This feature allows the programming language to bypass the performance costs related to the interaction with the intermediate mechanism.

The utilization of such features confers performance benefits upon statically-typed programming languages that are simply not available to dynamically-typed programming languages. As a result, these benefits are also not available to the optionally-typed programming languages that are built on top of dynamically-typed programming languages.

In this chapter, we introduce our proposed programming language named Poseidon Lua, and study the various language features that this language offers. Poseidon Lua extends the language of Typed Lua, an optionally-typed programming language, with several special operators that can be used to perform manual memory management. Poseidon Lua also augments the type system of Typed Lua with C types. Included among these C types are C pointer types that can be used for the purpose of direct memory programming.

Typed Lua provides an optional type system for the dynamically-typed programming language named Lua. Thus, Poseidon Lua is an optionally-typed programming language that allows the programmer to start programming in Lua and then gradually transition towards programming in C.

In Section 3.2, the language features of Poseidon Lua and their implementation are described. The grammar rules of Poseidon Lua are given in Section 3.3. From Section 3.4 to Section 3.9, we discuss the C types of Poseidon Lua. Finally, we conclude the chapter in Section 3.10.

3.2 Poseidon Lua Language

Poseidon Lua extends Typed Lua by adding a number of language features from the C programming language [12] that are required to enable manual memory management and direct memory programming.

3.2.1 Language Description

The Poseidon Lua language has two distinct features: operators and C types. In addition to those, a modification of the assignment operator, (`=`), of Typed Lua is included in Poseidon Lua. The language features are listed as follows:

1. Operators: `malloc`, `free`, and `sizeof`
2. Primitive C types: `char`, `int`, `double`, and `bool`
3. C pointer types
4. C struct types
5. C array types
6. Inter-operation between primitive C types and Typed Lua types
7. Modified semantics for assignment operator (`=`)

Poseidon Lua supports manual memory management through the `malloc` and `free` operators. The `malloc` operator allocates a block of memory that is of a specified size in bytes. The `free` operator deallocates a block of memory that was previously allocated by the `malloc` operator. The `malloc` operator and the `free` operator of Poseidon Lua have the same functionality as the `malloc` function and the `free` function of the C programming language, respectively.

Poseidon Lua provides 4 primitive C types: `char`, `int`, `double`, and `bool`. However, a variable cannot be declared to have a primitive C type directly in a Poseidon Lua program. Only a member of a C struct, an element of a C array, or the base type of a C pointer can be declared to have a primitive C type. Three of the primitive C types are the same as their counterparts in the C programming language. These are: `char`, `int`, and `double`. Poseidon Lua adds the `bool` type to make it easier to inter-operate with values of the the `boolean` type from Typed Lua.

Poseidon Lua supports inter-operation between values of primitive C types and values of Typed Lua types. At every use-site for a value that belongs to a primitive C type, Poseidon Lua automatically converts that value to another value that belongs to a corresponding type from Typed Lua. The primitive C types `char`, `int`, `double`, and `bool` correspond to the `string`, `integer`, `number`, and `boolean` types from Typed Lua, respectively. Note that Poseidon Lua does not use these Typed Lua types directly as the primitive C types in order to allow the primitive C types to interact with each other according to their own independent semantics which may conflict with those of the Typed Lua types.

Poseidon Lua provides C pointer types, which are conceptually the same as the pointer types from the C programming language. A C pointer type consists of a base type and a pointer depth. The base type for a C pointer type can be a primitive C type or a C struct type. A C pointer type can have an arbitrary pointer depth. The set of C pointer types includes a special C pointer type called '`ptr void`'. A variable of '`ptr void`' type can hold the value of any other C pointer type and vice versa. Moreover, Poseidon Lua offers a singleton C pointer value named `cs.NULL` that corresponds to the NULL pointer of the C programming language.

Poseidon Lua provides C struct types that can be used to represent composite objects. The C struct types are conceptually the same as the struct types from the C programming language. A member of a C struct may belong to a primitive C type, a C pointer type, or a C array type. A C struct type can be defined such that it has a member that belongs to a C pointer type with a base type that is the same C struct type that is being defined. Alternatively, a C struct type can be defined such that it has a member that belongs to a C pointer type with a base type that is some other previously defined C struct type.

Note that a C struct type cannot have a member with a type that is a C struct type. This feature is not needed because a member that belongs to a C pointer type whose base type is a C struct type can be used to achieve the same functionality instead.

Poseidon Lua provides C array types, which are conceptually the same as the array types from the C programming language. A C array could be used to structure a set of C values into a single-dimensional or multi-dimensional array. Note that a C array can only be declared as a member of a C struct. A C array type has a base type and a dimension depth. The base type indicates the type of values that a C array can store as elements. The base type can be a primitive C type or a C pointer type. The dimension depth states the number of dimensions in a single-dimensional or multi-dimensional C array. Each dimension must have a fixed number of elements. A C array is physically stored as a flat array of elements. Thus, a C array can be converted to a C pointer that points to the beginning of the flat array and that C pointer can be assigned to a variable of a C pointer type.

Poseidon Lua provides the `sizeof` operator, which can be used to determine the size, in bytes, of any of the C types. This applies to C array types as well. The sizes of the primitive C types are set as follows: the size of `char` is 1 byte, the size of `int` is 4 bytes, the size of `double` is 8 bytes, and the size of `bool` is 4 bytes. For example, `sizeof(char)` will return 1 byte. The size of all C pointer types are 8 bytes irrespective of the pointer depth. For example, the size of '`ptr int`' is 8 bytes, which will be returned by `sizeof(ptr int)`. The size of a C struct type is the sum of the sizes of the types of its members. Note that Poseidon Lua does not use padding for alignment. All the members are laid out sequentially on an array of bytes. The size of a C array type is obtained by multiplying the size of the base type with the size of each of the dimensions.

Poseidon Lua modifies the semantics of the assignment operator, (`=`), that is provided by Typed Lua. This modified semantics of the assignment operator is used whenever a value that belongs to a type from Typed Lua is assigned to a variable that belongs to a primitive C type. Before proceeding with the assignment, Poseidon Lua automatically converts the value to a corresponding value that belongs to the type of the variable. There are some subtleties involved for the case of the assignment of a value that belongs to the `string` type of Typed Lua to a variable of the primitive C type `char`. For the assignment from `string` to `char`, only the first character of the `string` value is represented in the converted `char` value.

The modified semantics of the assignment operator is also used whenever a value that belongs to the `string` type of Typed Lua is assigned to a variable that belongs to the '`ptr char`' C type and vice versa. When a `string` value is assigned to a '`ptr char`' variable,

the `string` value is automatically converted to a '`ptr char`' value that contains the null-terminated C string representation of the original `string` value. This null-terminated C string value is then stored in memory at the location that is pointed to by the '`ptr char`' variable. Similarly, when a '`ptr char`' value is assigned to a `string` variable, the '`ptr char`' value is automatically converted to a `string` value. The original '`ptr char`' value is assumed to be a null-terminated C string. Thus, all the characters until the first null-termination are represented in the converted `string` value. This converted `string` value is then assigned to the `string` variable.

Typed Lua provides the facility to perform multiple assignments in a single statement. However, the semantics for multiple assignments introduces subtle complexities when the same variable appears on both the left hand side (LHS) as well as the right hand side (RHS) of the assignment operator. For example, let us assume that we have 2 variables with the following initial values: '`local y = 2`', and '`local z = 3`'. Then, the multiple assignment statement '`y, z = z, y`' proceeds as follows. First, Typed Lua evaluates the value of each of the expressions on the RHS of the assignment operator, (=). Thus, those expressions have the following values: '`z == 3`', '`y == 2`'. Next, Typed Lua performs the multiple assignment. After the multiple assignment, the variables have the following values: '`y == 3`', and '`z == 2`'. We call this the Multiple Assignment (MA) semantics.

The MA semantics for multiple assignment is confusing at times as the final result differs from what a programmer would normally expect if the assignments were carried out individually. Breaking up the multiple assignment statement would lead to the following single assignment statements: '`y = z`', '`z = y`'. Since '`z == 3`', the first assignment evaluates to '`y = 3`'. Since '`y == 3`' after the first assignment, the second assignment evaluates to '`z = 3`'. Thus, the value of each variable after the assignments is as follows: '`y == 3`', '`z == 3`'. We call this the Single Assignment (SA) semantics. Note that in the MA semantics, '`z == 2`' but in the SA semantics, '`z == 3`'.

Ultimately, an assignment statement that involves values of a C type has to be implemented through a translation to the invocations of the special operators of Modified Lua. Implementing the MA semantics for such an assignment statement would have introduced too much complexity and too much performance costs. In this situation, it is possible to implement the SA semantics with less of a performance overhead. Thus, Poseidon Lua executes multiple assignment statements with the SA semantics when the LHS of the assignment operator involves dereferencing a C pointer, accessing the member of a C struct, or indexing into a C array. This is also the case when a value that belongs to the `string` type of Typed Lua is assigned to a variable of the C type '`ptr char`' or vice versa.

With the language features described in this section, Poseidon Lua introduces gradual C

programming in Typed Lua programs by bringing the C programming paradigm to Typed Lua programs. Moreover, these Poseidon Lua language features help to achieve the 2 main objectives in this thesis: manual memory management and direct memory programming. The implementations of the Poseidon Lua language features are considered in the next sub-section.

3.2.2 Language Implementation

We integrate the features of Poseidon Lua into Typed Lua by modifying the compiler of Typed Lua. The compiler of Typed Lua is written in the Lua programming language. The source language of the modified compiler is Poseidon Lua and the target language is Modified Lua, which is introduced in Chapter 4. The main modifications are discussed as follows.

The lexer of Typed Lua uses the LPeg library to recognize the keywords of the language as lexical tokens. We use the lexer to recognize the following new lexical tokens of Poseidon Lua: `char`, `int`, `double`, `bool`, `ptr`, `struct`, `malloc`, and `sizeof`.

The parser of Typed Lua also uses the LPeg library to implement the grammar of the language. The parser parses a program to construct an abstract syntax tree (AST) according to the specified grammar. We modify the parser of Typed Lua to extend the grammar of Typed Lua with new production rules to add support for the language features of Poseidon Lua using the LPeg library as well. The actual grammar rules that are added to the grammar of Typed Lua are presented in Section 3.3.

The new production rules enable the parsing of primitive C types, C pointer types, and C array types as Typed Lua types. We also add new production rules to enable the parsing of the `struct` definition construct, which is used to define a new C struct type.

Furthermore, we add new production rules to enable the parsing of the `malloc` and `sizeof` operators. We need to modify the parser to be able to parse these operators because they both take C types as inputs, whereas a function in Typed Lua is only able to take Typed Lua values as inputs. Thus, these operators are not automatically detected as Typed Lua functions by the parser. Since the `free` operator does not take C types as input, it does not need any additional support from the parser because it is parsed as a normal Typed Lua function.

We modify the AST implementation to include new kinds of AST nodes for the primitive C types, C pointer types, and C array types. Furthermore, a new helper function is implemented to process C struct type definitions. This helper function processes a C

struct type definition and generates the AST node for Typed Lua interfaces. The total size, in bytes, of the C struct type as well as the offset of each of its members is calculated and stored in the AST node. New helper functions are also implemented to process the `malloc` and `sizeof` operators. The `malloc` and `sizeof` operators are processed such that they use the AST node for normal Typed Lua function calls.

We modify the static typechecker of Typed Lua to enable the static typechecking of the operators and the C types that are provided by Poseidon Lua. Additionally, we use the static typechecking process to perform a type-directed translation of a Poseidon Lua program to a Modified Lua program. To allow a Poseidon Lua program to be properly typechecked, we implement our own helper functions to typecheck AST nodes that represent Typed Lua function call expressions, Typed Lua table indexing expressions, and Typed Lua assignment statements.

The helper function that we provide to typecheck Typed Lua function call expressions proceeds as follows. It detects whether the function being called is one of the following operators of Poseidon Lua: `sizeof`, `malloc`, or `free`. If so, its arguments are typechecked and the invocation of the Poseidon Lua operator is translated to an invocation of the appropriate C Semantics (CS) operator from Modified Lua, if necessary. Modified Lua is introduced in Chapter 4. An invocation of the `sizeof` operator of Poseidon Lua is translated to a Typed Lua `integer` value that represents the size, in bytes, of the C type that is sent as the argument of the `sizeof` operator. An invocation of the `malloc` operator of Poseidon Lua is translated to an invocation of the the `CS_malloc` operator of Modified Lua. An invocation of the `free` operator of Poseidon Lua is translated to an invocation of the the `CS_free` operator of Modified Lua.

The helper function that we provide to typecheck Typed Lua table indexing expressions proceeds as follows. In Poseidon Lua, we reuse the same syntax as the Typed Lua table indexing expressions for the purpose of dereferencing a C pointer, accessing the member of a C struct, and indexing into a C array. Thus, our helper function begins by detecting whether the expression being typechecked is indeed dereferencing a C pointer, accessing the member of a C struct, or indexing into a C array. If so, the expression is typechecked and translated to an invocation of the appropriate CS operator from Modified Lua.

The helper function that we provide to typecheck Typed Lua assignment statements proceeds as follows. At first, note that in Typed Lua, the assignment statement is always a multiple assignment statement. Our helper function begins by detecting whether any of the expressions on the left hand side (LHS) of the assignment operator is dereferencing a C pointer, accessing the member of a C struct, or indexing into a C array. If so, this expression as well as its corresponding expression on the right hand side (RHS) is typechecked and

the RHS expression is translated to an invocation of the appropriate CS operator from Modified Lua. If the LHS expression is of type '`ptr char`' and the RHS expression is of type `string` or vice versa, both expressions are typechecked and the RHS expression is translated to an invocation of the appropriate CS operator from Modified Lua. This step is done to support the automatic conversion of a `string` value to a '`ptr char`' value through assignment, and vice versa. If any of the required kinds of expressions is found on the LHS of the assignment operator, then the multiple assignment statement is broken into a series of single assignment statements. These single assignment statements are placed inside a `do/end` statement. This implements the change of semantics for the multiple assignment statement from MA semantics to SA semantics.

The source code for Poseidon Lua is available at GitHub [15].

3.3 Grammar

```
1  <C_Type> ::= <C_BaseType>
2             | <PtrType>
3             | <C_ArrayType>
4
5  <C_BaseType> ::= "char"
6                 | "int"
7                 | "double"
8                 | "bool"
9
10 <C_VoidType> ::= "void"
11
12 <PtrType> ::= "ptr" { "ptr" }
13               ( <C_VoidType> | <C_BaseType> | <VariableType> )
14
15 <C_ArrayType> ::= "[" Number "]" { "[" Number "]" }
16                   ( <C_BaseType> | <PtrType> )
17
18 <VariableType> ::= Name
```

Figure 3.1: Grammar for the C types of Poseidon Lua

```
1  <Struct> ::= "struct" <Struct_TypeDec>
2
3  <Struct_TypeDec> ::= Name <Struct_IdDecList> "end"
4
5  <Struct_IdDecList> ::= <Struct_IdDec> { <Struct_IdDec> }
6
7  <Struct_IdDec> ::= <IdList> ":" <C_Type>
```

Figure 3.2: Grammar for the `struct` definition construct

```
1  <MallocExp> ::= "malloc" "(" ( <PtrType> "," <Expr>
2                               | <PtrType>
3                               | <Expr> ) ")"
4
5  <SizeofExp> ::= "sizeof" "(" ( <C_Type> | <VariableType> ) ")"
```

Figure 3.3: Grammar for the `malloc` and `sizeof` operator expressions

Poseidon Lua extends the grammar of Typed Lua to include support for C types, the `struct` definition construct, and the `malloc` and `sizeof` operator expressions. In Figures 3.1 - 3.3, we use curly braces '{}' to represent a repetition of 0 or more times of the elements inside and parenthesis '()' to represent a grouping of the elements inside.

Figure 3.1 shows the grammar for the C types of Poseidon Lua. Note that `Number` and `Name` are classes of tokens that are defined in the lexer of Typed Lua. `Number` represents any valid numerical literal and `Name` represents any valid identifier for variables, functions, and so on. `<VariableType>` is a pre-existing non-terminal symbol from the grammar of Typed Lua that is recycled by Poseidon Lua to represent the name of a C struct type in the context of `<PtrType>`. The rest is self-explanatory.

Figure 3.2 shows the grammar for the `struct` definition construct that is used to define a C struct type in Poseidon Lua. Note that `<IdList>` is a pre-existing non-terminal symbol from the grammar of Typed Lua that represents a valid identifier or a comma separated list of valid identifiers for variables. The rest is self-explanatory.

Figure 3.3 shows the grammar for the `malloc` and `sizeof` operator expressions of Poseidon Lua. Note that `<Expr>` is a pre-existing non-terminal symbol from the grammar of Typed Lua that represents an arbitrary expression in the language of Typed Lua. The

rest is self-explanatory.

3.4 The C types and the `sizeof` operator

Poseidon Lua provides a `sizeof` operator that can be used to determine the size of a C type. It takes as input a single C type and returns a value that represents the number of bytes in memory that are used to store a value of that C type. The returned value belongs to the type `integer` of Typed Lua. The `sizeof` operator does not take padding into account because Poseidon Lua does not use padding for alignment. Everything is placed sequentially in an array of bytes.

The `sizeof` operator of Poseidon Lua is similar to the `sizeof` operator that is provided by C. The `sizeof` operator that is provided by the C programming language is able to take as input either a type or an object of the C programming language, which can be an array, a structure, or a variable [12]. However, the `sizeof` operator that is provided by Poseidon Lua is able to take as input only a C type, including a C array type.

Next, we introduce the C types that are provided by Poseidon Lua. We also demonstrate the use of the `sizeof` operator to calculate the size of each of these C types. The C types that are introduced in this section are covered in more detail in later sections.

3.4.1 The primitive C types

Poseidon Lua provides `char`, `int`, and `double` as primitive C types. These correspond to the `char`, `int`, and `double` types that are provided by C. In addition, Poseidon Lua provides `bool` as a primitive C type for boolean values. We choose to include a special type for boolean values to make it easier to interact with values of the `boolean` type of Typed Lua.

We fix the size of each of the primitive C types. We set `char` to 1 byte in size, `int` to 4 bytes in size, `double` to 8 bytes in size, and `bool` to 4 bytes in size. Therefore, `sizeof(char)` evaluates to `1`, `sizeof(int)` evaluates to `4`, `sizeof(double)` evaluates to `8`, and `sizeof(bool)` evaluates to `4`.

3.4.2 The C pointer type

Poseidon Lua provides a C pointer type. A C pointer represents a reference to a location in memory and can be dereferenced, using the index operator '`[]`', to yield the value that

is stored in that location in memory. In Poseidon Lua, a variable named '`b`' of a C pointer type can be declared as follows: '`local b : ptr ptr ptr int`'. This is equivalent to the declaration '`int ***b;`' in C. Note that the number of instances of '`*`' in the C declaration matches the number of instances of '`ptr`' in the Poseidon Lua declaration.

Each C pointer type consists of a pointer depth and a base type. For the variable '`b`', the pointer depth is 3 as indicated by the 3 instances of the '`ptr`' keyword in the '`ptr ptr ptr`' part of the declaration. If a C pointer is dereferenced the same number of times as its pointer depth, then the result is a value that belongs to the base type of the C pointer. If the base type of the C pointer is a primitive C type, then the original value is automatically converted to a value that belongs to a corresponding type from Typed Lua.

For the variable '`b`', the base type of the C pointer is `int`. Therefore, dereferencing the variable '`b`' three times, as in '`b[0][0][0]`', yields a final value of type `int`. Since `int` is a primitive C type, the original value is automatically converted to a value that belongs to the `integer` type of Typed Lua. This conversion is performed because the primitive C type `int` corresponds to the `integer` type of Typed Lua.

The base type of a C pointer type can be one of the following: a primitive C type (`char`, `int`, `double`, `bool`), `void`, or a C struct type. The base type cannot be a Typed Lua type such as `integer`.

We choose to fix the size of a C pointer type to 8 bytes. Thus, `sizeof(ptr int)`, `sizeof(ptr ptr int)`, and `sizeof(ptr ptr ptr int)` all evaluate to `8`.

3.4.3 The C struct type

Poseidon Lua provides a C struct type. A C struct encapsulates members of different types into a single object. Poseidon Lua also provides a `struct` definition construct that can be used to define a C struct type. A C struct type named '`Pen`' can be defined using the `struct` definition construct as follows: '`struct Pen length : int width : int end`'. This is equivalent to the declaration '`struct Pen { int length; int width; };`' in C.

In Poseidon Lua, a C struct can only be used through a C pointer to that C struct. This restriction is necessary because Poseidon Lua always stores C struct values on the heap, leaving the stack untouched. As a result, only a C pointer to the location of the C struct value on the heap is ever made available for manipulation to the Poseidon Lua program. Thus, a variable of type '`ptr Pen`' must be used to manipulate a C struct of type `Pen`. Such a variable named '`b`' can be declared as follows: '`local b : ptr Pen`'.

The members of '`b`' can be accessed only using the member-access (`.`) operator. Thus, the '`length`' member can be accessed as follows: '`b.length`'. The '`width`' member can be accessed as follows: '`b.width`'.

A member of a C struct can belong to any of the following primitive C types: `char`, `int`, `double`, `bool`. A member can also belong to a C pointer type, such as '`ptr int`'. Moreover, a member can also belong to a C type that is a C pointer to a C struct such as '`ptr Pen`'. A member can also belong to a C array type, such as '`[5][2] int`'.

We calculate the size of a C struct type by adding up the size of the type of each of its members. Note that Poseidon Lua does not use any padding for alignment. Everything is laid out sequentially on an array of bytes. Since the C struct type `Pen` has 2 members of type `int`, the size of the C struct type `Pen` is `4 + 4 = 8` bytes. Therefore, `sizeof(Pen)` evaluates to `8`.

3.4.4 The C array type

Poseidon Lua provides a C array type. A C array can have multiple dimensions with a fixed number of elements for each dimension. A C array can only be declared as a member of a C struct. This restriction is necessary because Poseidon Lua always stores C arrays on the heap, leaving the stack untouched. As a result, a C pointer to the location of the C array on the heap would have to be made available for manipulation to the Poseidon Lua program. However, allowing a C pointer to point to a C array would add too much complexity to the type system of Poseidon Lua. In order to avoid this extra complexity, all C arrays are required to be placed inside a C struct. Thus, a C pointer to the C struct containing the C array is made available for manipulation to the Poseidon Lua program.

In Poseidon Lua, a member named '`b`' of a C struct that is of a C array type can be declared as follows: '`b : [2][2][2] int`'. This is equivalent to the following declaration in C: '`int b[2][2][2];`'.

Each C array type consists of a dimension depth and a base type. For '`b`', the dimension depth is 3 as indicated by the 3 dimension sizes that are specified in the '`[2][2][2]`' part of the declaration. In this case, each dimension has 2 elements. If a C array is indexed, the same number of times as its dimension depth, then the result is a value that belongs to the base type of the C array. If the base type of the C array is a primitive C type, then the value is automatically converted to a value that belongs to a corresponding type from Typed Lua.

For '`b`', the base type of the C array is `int`. Therefore, indexing the variable '`b`' three times, as in '`b[0][0][0]`', yields a value of type `int`. Since `int` is a primitive C type, the

original value is automatically converted to a value that belongs to the `integer` type of Typed Lua. This conversion is performed because the primitive C type `int` corresponds to the `integer` type of Typed Lua.

The base type of a C array type can be any of the following primitive C types: `char`, `int`, `double`, `bool`. The base type can also be a C pointer type.

The size of a C array type is calculated by multiplying the size of the base type with each of the dimension sizes. Therefore, for '`b`', the size of its type would be calculated as `4 * 2 * 2 * 2 = 32` bytes. Therefore, `sizeof([2][2][2] int)` evaluates to `32`.

3.5 The `malloc` and `free` operators

Poseidon Lua provides a `malloc` operator that can be used to allocate a block of memory for use by the program. It takes as input the number of bytes to allocate and returns a C pointer to the allocated block of memory. Poseidon Lua also provides a `free` operator that can be used to deallocate a block of memory that is no longer needed by the program. It takes as input a C pointer to a block of memory that was previously allocated using the `malloc` operator and proceeds to deallocate that block of memory. The `malloc` operator and the `free` operator allow the programmer to perform manual memory management.

The `malloc` operator of Poseidon Lua is inspired by the `malloc` function that is provided by C. Similarly, the `free` operator of Poseidon Lua is inspired by the `free` function that is provided by C. The main difference between the operators that are provided by Poseidon Lua and the functions that are provided by C is that Poseidon Lua allows the `malloc` operator to be used in a variety of different configurations to make programming easier.

In this section, we demonstrate the proper use of the `malloc` operator and the `free` operator in Poseidon Lua. We show how these operators can be used to allocate and deallocate memory blocks.

In Figure 3.4, lines 1 - 5 define the C struct type named `FruitBasket`. Note that `FruitBasket` has 3 members and each member is of type `int`. The primitive C type `int` is 4 bytes in size. Thus, the total size of the C struct type `FruitBasket` is `4 + 4 + 4 = 12` bytes. The rest of this section makes use of the C struct type defined in Figure 3.4.

3.5.1 The number of bytes as input

In one configuration, the `malloc` operator takes as input the number of bytes to allocate. The `malloc` operator allocates a block of memory of the required number of bytes and

```
1 struct FruitBasket
2         num_oranges : int
3         num_grapes : int
4         num_strawberries : int
5 end --end FruitBasket
```

Figure 3.4: malloc.tl, Part 1 of 7

returns a C pointer to that block of memory. We choose to include this configuration to allow the programmer to easily allocate a block of memory of any arbitrary size by simply specifying the required size in terms of the number of bytes in the memory block.

```
1 --[[ Malloc: bytes ]]
2 local basket_1 : ptr FruitBasket = malloc( 12 )
3
4 basket_1.num_oranges = 101
5 basket_1.num_grapes = 102
6 basket_1.num_strawberries = 103
```

Figure 3.5: malloc.tl, Part 2 of 7

In Figure 3.5, line 2 invokes the `malloc` operator with an input of `12`. For this particular configuration, the input value has to be of the `integer` type of Typed Lua. The `malloc` operator allocates a block of memory that is `12` bytes in size. Note that the size of the allocated block is large enough to hold a C struct of type `FruitBasket`. The `malloc` operator returns a C pointer to the allocated block of memory. This returned value is of type '`ptr void`'. The returned value is assigned to a local variable named `basket_1` of type '`ptr FruitBasket`'. The type of the returned value is automatically converted from '`ptr void`' to '`ptr FruitBasket`' in the process. The ultimate effect of this statement is similar to that of the C statement: '`struct FruitBasket *basket_1 = malloc( 12 );`'.

In Figure 3.5, lines 4 - 6 assign values to the members of `basket_1`.

3.5.2 The result of the `sizeof` operator as input

In this section, we explore an interesting consequence of the use of the configuration of the `malloc` operator that has been already discussed in Section 3.5.1. Since the `malloc` operator expects an input of type `integer` of Typed Lua, the programmer is able to use the result of applying the `sizeof` operator to a particular C type as input to the `malloc` operator instead of supplying the number of bytes directly. We know from Section 3.4 that the `sizeof` operator does indeed return a value of type `integer` of Typed Lua.

```
1 --[[ Malloc: sizeof ]]
2 local basket_2 : ptr FruitBasket = malloc( sizeof( FruitBasket ) )
```

Figure 3.6: malloc.tl, Part 3 of 7

In Figure 3.6, line 2 invokes the `malloc` operator with `sizeof( FruitBasket )` as an input. The ultimate effect of this statement is similar to that of the C statement: '`struct FruitBasket *basket_2 = malloc( sizeof( struct FruitBasket ) );`'.

3.5.3 The C pointer type as input

In one configuration, the `malloc` operator takes as input a C pointer type and returns a value of the same C pointer type. The `malloc` operator automatically calculates what would be the result of applying the `sizeof` operator to the C type that is pointed to by the input C pointer type and allocates a block of memory of that size. The `malloc` operator returns a C pointer to the allocated block of memory. The type of the returned C pointer is the same as the input C pointer type. For example, '`malloc( ptr int )`' has the same effect as '`malloc( sizeof( int ) )`', but the type of the returned value would be '`ptr int`' and not '`ptr void`'. We choose to include this configuration to allow the programmer to easily allocate a block of memory without having to perform complex calculations to determine its size and without having to use the `sizeof` operator directly.

```
1 --[[ Malloc: shortcut ]]
2 local basket_3 : ptr FruitBasket = malloc( ptr FruitBasket )
```

Figure 3.7: malloc.tl, Part 4 of 7

In Figure 3.7, line 2 invokes the `malloc` operator with an input of '`ptr FruitBasket`'. The `malloc` operator determines that the result of '`sizeof( FruitBasket )`' would be `12` because the size of a C struct of type `FruitBasket` is `12` bytes. Thus, the `malloc` operator allocates a block of memory that is `12` bytes in size. The `malloc` operator returns a C pointer to the allocated block of memory. This returned value is of type '`ptr FruitBasket`', the same type that was the input to the `malloc` operator. The returned value is assigned to a local variable named `basket_3` of type '`ptr FruitBasket`'. The ultimate effect of this statement is similar to that of the C statement: '`struct FruitBasket *basket_3 = malloc( sizeof( struct FruitBasket ) );`'.

An interesting aspect of this particular configuration for the `malloc` operator is that it allows the programmer to set the type of the C pointer that is returned by the `malloc` operator. The return type is always the type that is used as the input to the `malloc` operator. This is a valuable feature in situations where the C pointer that is returned by the `malloc` operator has to be assigned to a variable that cannot be given a static type annotation. One such situation is when the members of a Lua table have to be assigned initial values inside a Lua table constructor expression. We now give an example of this situation.

```
1 --[[ Malloc: tables ]]
2 local table_2 = { basket = malloc( ptr FruitBasket ) }
```

Figure 3.8: malloc.tl, Part 5 of 7

In Figure 3.8, line 2 creates a Lua table and assigns it to a local variable named `table_2`. A Lua table constructor expression is used to initialize the `basket` member of `table_2`. In the Lua table constructor expression, `basket` is assigned the C pointer that is returned by the `malloc` operator. Since the input to the `malloc` operator is '`ptr FruitBasket`', the type of the C pointer that is returned by the `malloc` operator is also '`ptr FruitBasket`'. Thus, the type of `table_2.basket` is automatically inferred to be '`ptr FruitBasket`'.

3.5.4 The C pointer type and the number of bytes as inputs

In one configuration, the `malloc` operator takes as its first input a C pointer type and as its second input the number of bytes to allocate. The `malloc` operator allocates a block of memory. The size of the block of memory is specified by the second input to the `malloc` operator. The `malloc` operator returns a C pointer to the allocated block of memory. The

C pointer that is returned has the same type as the C pointer type that is the first input to the `malloc` operator. We choose to include this configuration to allow the programmer to easily allocate a block of memory and to set the type for the C pointer that is returned at the same time.

We also choose to include this configuration to allow the programmer to implement functionality that is similar to that of polymorphism. For example, let us assume that we have defined a C struct type named `FruitBasket_2` that has the same members as `FruitBasket` as well as an additional member named `num_blueberries` that belongs to the type `int`. In Poseidon Lua, there is no way to indicate that the C struct type `FruitBasket_2` is a specialized version of the C struct type `FruitBasket`. However, it is possible to use '`malloc( ptr FruitBasket, sizeof( FruitBasket_2 ) )`' to allocate a block of memory that is large enough to hold all the members of `FruitBasket_2` as well as to get back a C pointer to that block of memory that has the type '`ptr FruitBasket`'. The returned C pointer can then be used anywhere that expects a C pointer of type '`ptr FruitBasket`'.

An interesting aspect of this particular configuration for the `malloc` operator is that it also allows the programmer to set the type of the C pointer that is returned by the `malloc` operator. This behavior is similar to the configuration discussed in Section 3.5.3. The return type is always the type that is used as the first input to the `malloc` operator. This is a valuable feature in situations where the C pointer that is returned by the `malloc` operator has to be assigned to a variable that cannot be given a static type annotation. One such situation is when the members of a Lua table have to be assigned initial values inside a Lua table constructor expression. We now give an example of this situation.

In Figure 3.9, lines 1 - 9 define a C struct type named `FruitBasket_2`. Lines 12 - 13 create a Lua table and assign it to a local variable named `table_3`. A Lua table constructor expression is used to initialize the `basket` member of `table_3`. In the Lua table constructor expression, `basket` is assigned the C pointer that is returned by the `malloc` operator. Since the first input to the `malloc` operator is '`ptr FruitBasket`', the type of the C pointer that is returned by the `malloc` operator is also '`ptr FruitBasket`'. Thus, the type of `table_3.basket` is automatically inferred to be '`ptr FruitBasket`'. The second input to the `malloc` operator is `sizeof( FruitBasket_2 )`, which evaluates to '`4 + 4 + 4 + 4 = 16`' bytes. Thus, the `malloc` operator allocates a memory block that is 16 bytes in size.

```
1  struct FruitBasket_2
2
3          num_oranges : int
4          num_grapes : int
5          num_strawberries : int
6
7          num_blueberries : int
8
9  end --end FruitBasket_2
10
11
12 local table_3 = { basket = malloc( ptr FruitBasket,
13                                     sizeof( FruitBasket_2 ) ) }
```

Figure 3.9: malloc.tl, Part 6 of 7

3.5.5 The `free` operator

The `free` operator can be used to deallocate a block of memory. The `free` operator takes as its first input a C pointer that points to a block of memory that was previously allocated using the `malloc` operator. The `free` operator proceeds to deallocate the block of memory that is pointed to by the input C pointer.

```
1  --[[ Malloc: bytes ]]
2  local basket_1 : ptr FruitBasket = malloc( 12 )
3
4  free( basket_1 )
```

Figure 3.10: malloc.tl, Part 7 of 7

In Figure 3.10, line 4 applies the `free` operator to `basket_1`, which holds a C pointer to a block of memory that was previously allocated using the `malloc` operator in line 2. Note that the ultimate effect of this statement is similar to that of the C statement: '`free( basket_1 );`'.

3.6 The assignment operator

Poseidon Lua modifies the semantics of the assignment operator (=) according to the C types of the expressions that are encountered within the containing assignment statement. In Typed Lua, it is possible to use the assignment operator to perform a multiple assignment where multiple variables are assigned values at the same time. The semantics of multiple assignment in Typed Lua is such that the expressions on the right hand side (RHS) of the assignment operator, (=), are evaluated first and then all the values are assigned to the corresponding variables on the left hand side (LHS) of the assignment operator, (=). We refer to this assignment semantics as the multiple assignment (MA) semantics.

Note that a value of a Typed Lua type can be assigned to a variable of a C type. Similarly, a value of a C type can also be assigned to a variable of a Typed Lua type.

When the LHS of the assignment operator, (=), consists of a dereferencing of a C pointer, a member access on a C pointer to a C struct, or an indexing of a C array, Poseidon Lua switches the semantics of a multiple assignment statement from the MA semantics to the single assignment (SA) semantics. Poseidon Lua also uses the SA semantics if the multiple assignment statement contains an assignment from the Typed Lua type `string` to the C type '`ptr char`' or vice versa. In these situations, the switch to SA semantics is needed because enforcing the original MA semantics would have required too much complex maneuvering with a corresponding increase in performance cost.

In SA semantics, the multiple assignment statement is split into a series of single assignment statements internally by Poseidon Lua. Each single assignment statement consists of the corresponding LHS variable and the RHS expression of the multiple assignment statement in order from left to right. The expression on the RHS of the assignment operator, (=), of the first split assignment statement is evaluated and the resulting value is assigned to the variable on the LHS of the assignment operator, (=), before the expression on the RHS of the assignment operator, (=) of the next split assignment statement is evaluated and so on.

The final result of a multiple assignment statement is different depending on which assignment semantics is used by Poseidon Lua.

3.6.1 Conversions for primitive C types

Poseidon Lua supports the automatic conversion of a value that belongs to a type from Typed Lua to a value that belongs to a corresponding primitive C type during assignment

to a variable that belongs to the same primitive C type. The `string` type from Typed Lua corresponds to the primitive C type `char`. The `integer` type from Typed Lua corresponds to the primitive C type `int`. The `number` type from Typed Lua corresponds to the primitive C type `double`. The `boolean` type from Typed Lua corresponds to the primitive C type `bool`.

Note that a variable of a primitive C type can only be on the LHS of the assignment operator, (=), as a result of a dereferencing of a C pointer, a member access on a C pointer to a C struct, or an indexing of a C array. This restriction is necessary to avoid any confusion between primitive C types and their corresponding Typed Lua types when dealing with variables within a program. Under these circumstances, Poseidon Lua switches the semantics of a multiple assignment statement from the MA semantics to the single assignment (SA) semantics. The automatic conversion of the value on the RHS that belongs to a Typed Lua type to a value that belongs to a corresponding primitive C type is implemented as part of the switch to SA semantics.

Conversion to `char`

```
1 --[[ Assignment: primitive C types ]]
2 local producer_1 : string = "A"
3 local holder_1 : ptr char = malloc( ptr char )
4
5 holder_1[0] = producer_1
6
7 print( "producer_1 : " .. tostring( producer_1 ) )
8 print( "holder_1[0] : " .. tostring( holder_1[0] ) )
```

Figure 3.11: assignment.tl

In Figure 3.11, line 2 shows that the variable `producer_1` has type `string` from Typed Lua and its value is set to `"A"`. Line 3 shows that the variable `holder_1` has the C type '`ptr char`'. Thus, `holder_1[0]` has the primitive C type `char` because it dereferences the C pointer `holder_1` at offset `0`. Line 5 assigns the value of `producer_1` to `holder_1[0]`. Therefore, this is an assignment of a value of type `string` of Typed Lua to a variable of the primitive C type `char`. Thus, an assignment of the value of `producer_1` to `holder_1[0]` proceeds by converting the value of `producer_1` from a value of type `string` to a value of type `char` and assigning the converted value to `holder_1[0]`. Therefore, `holder_1[0]` will have the value of `'A'` after the assignment.

Lines 7 - 8 print the values of these variables. Note that when the result of dereferencing a C pointer is a value that belongs to a primitive C type, that value will always be automatically converted to a value of the corresponding Typed Lua type. Thus, in line 8, the value of `holder_1[0]` will be automatically converted from `'A'` to `"A"`.

The conversions from `integer` to `int`, from `number` to `double`, and from `boolean` to `bool` are carried out in a similar manner to the conversion from `string` to `char`.

Assignments between `ptr char` and `string`

Poseidon Lua allows automatic conversions of values between the C pointer type '`ptr char`' and the `string` type of Typed Lua. When a value of type `string` is assigned to a variable of type '`ptr char`', the value is converted to another value of type '`ptr char`' and then stored at the location in memory that is pointed to by the variable. When a value of type '`ptr char`' is assigned to a variable of type `string`, the value is converted to another value of type `string` and then assigned to the variable. We choose to include this feature to allow a Poseidon Lua program to interact seamlessly with both strings from Typed Lua and null-terminated C strings.

Moreover, note that in a multiple assignment statement, if there is an assignment from `string` to '`ptr char`' or an assignment from '`ptr char`' to `string`, Poseidon Lua will use the SA semantics.

3.7 The C pointer types

Poseidon Lua provides C pointer types. A C pointer can be used to allocate an array whose size is only known at run-time. This is in contrast to a multi-dimensional C array member of a C struct whose size has to be known at compile-time. Allowing the size of the array to be determined at run-time instead of at compile-time enables programs to be more flexible and make them capable of solving a wider range of problems within the same program. A C pointer would point to the first element of the array. An individual element of the array can be accessed by indexing the C pointer with the offset of the required element from the first element in the array.

The base type of a C pointer type can be any of the following primitive C types: `char`, `int`, `double`, `bool`. The base type can be `void`. The base type can also be a C struct type.

3.7.1 C pointer to `void`

Poseidon Lua provides a special C pointer type, '`ptr void`', which acts as a general C pointer type that serves a similar purpose to the '`void *`' type in the C programming language [12].

A value of any C pointer type can be assigned to a variable of type '`ptr void`' and vice versa. For example, a value of type '`ptr ptr ptr int`' can be assigned to a variable of type '`ptr void`'. Similarly, a value of type '`ptr void`' can be assigned to a variable of type '`ptr ptr ptr int`'. In both examples, the assignment is successful even though '`ptr ptr ptr int`' has a pointer depth of 3 and '`ptr void`' has a pointer depth of 1.

3.7.2 C pointer to an array of C structs

Poseidon Lua provides C pointers that can point to an array of C structs. A particular element of the array could be accessed by indexing the C pointer with the offset of the required element relative to the first element of the array. The result is a C pointer to the required element and the type of the result is the same as the type of the C pointer to the original array. This is necessary because Poseidon Lua only allows a C struct to be manipulated through a C pointer to the location of the C struct in the heap.

This is a different behavior than what occurs when the C pointer points to an array of elements that belong to a primitive C type such as `int`. In that case, the result of indexing into the C pointer is the required value itself and the type of the result is the base type of the C pointer to the array. Since the base type of the C pointer is a primitive C type, the accessed value is automatically converted to a value of a corresponding Typed Lua type.

Note also that in C, the result of dereferencing a pointer to a struct is a value that belongs to the type of the struct. In Poseidon Lua, the result is a value that belongs to the type of the C pointer to the C struct. This makes it easier to program in Poseidon Lua because the members of a C struct can only be accessed through a C pointer to that C struct.

3.7.3 Setting a C pointer to NULL value

In Poseidon Lua, a C pointer of any type can be assigned the value of `cs.NULL` to set its value to NULL. The value of a C pointer of any type may also be checked against the value of `cs.NULL` using the equality operators, (== and ~=), to determine if its value is already

set to NULL. Therefore, the equality operators can be used with `cs.NULL` in the same way that they can be used with any other C pointer value.

3.8 The C struct types

Poseidon Lua provides C struct types as a way to encapsulate members of different types into one type. In Poseidon Lua, a C struct type is defined using the struct definition construct.

A C struct type can be defined to be completely stand-alone in the sense that it does not contain a member that belongs to a C pointer type whose base type is another C struct type. Alternatively, a C struct type can be defined such that it does contain a member that belongs to a C pointer type whose base type is another C struct type. In addition, a C struct type can also be defined such that it contains a member that belongs to a C pointer type whose base type is the containing C struct type. We refer to this last kind of C struct types as recursively defined C struct types.

Note that Poseidon Lua does not allow a C struct to be embedded inside another C struct as a member. This feature is not needed because a member that is a C pointer to a C struct can be used instead to achieve the same functionality. This keeps the type system of Poseidon Lua simple.

3.8.1 The type of a member

Poseidon Lua provides C struct types that encapsulate members of different types. These members may belong to any of the following primitive C types: `char`, `int`, `double`, `bool`. The members may also belong to a C pointer type such as the following: '`ptr int`'. The members may also belong to a C array type such as the following: '`[2] int`'. Therefore, a C struct can be used to represent a wide variety of objects which may be composed of different kinds of components.

3.9 The C array types

Poseidon Lua also provides C array types. A C array is a single-dimensional or multi-dimensional array with a base type. For a C array, the size of each dimension must

be determined purely at compile-time. We choose to include this feature to enable the programmer to make use of multi-dimensional arrays. In certain circumstances, organizing a large set of values into a multi-dimensional array will simplify code. Therefore, this feature is beneficial for programmers when a simple structuring mechanism is required for a large set of values.

Note that each element of a multi-dimentional C array is itself a C array. Assume `x` is a C array of `N` dimensions where `N > 1`. Each element of `x` is a C array `y` of `N-1` dimensions. If `N - 1 > 1`, each element of `y` is a C array `z` of `N-2` dimensions and so on. Each element of a C array can be accessed by indexing the C array with the offset of the required element from the first element of the C array. Indexing into a C array of a single dimension returns a value of the base type.

The base type of a C array type can be one of the following primitive C types: `char`, `int`, `double`, `bool`. The base type of a C array type can also be a C pointer type such as '`ptr ptr int`'.

3.9.1 Assignment to a C pointer type

Poseidon Lua allows the automatic conversion of a value of a C array type to a value of a C pointer type during assignment. This is possible because a multi-dimensional C array is physically stored as a flat one-dimensional array. All the elements of the C array are stored in order from the first element to the last element in the flat array. If an element of the C array is also a C array, all of its elements are also stored in order from the first element to the last element and so on.

For example, if a C array is declared as follows: '`a : [2][2][2] int`', then the values of the C array are stored in the flat array in the following order: `a[0][0][0]`, `a[0][0][1]`, `a[0][1][0]`, `a[0][1][1]`, `a[1][0][0]`, `a[1][0][1]`, `a[1][1][0]`, `a[1][1][1]`. Since a C pointer can point to an array of elements, it is possible for a C pointer to point to the flat array. Thus, a value of a C array type can be assigned to a variable of an appropriate C pointer type.

If a value is of a C array type with base type `BASE_TYPE`, then the value can be assigned to a variable of a C pointer type that points to `BASE_TYPE`. For example, a value of type '`[10] int`', where `BASE_TYPE` is '`int`', can be assigned to a variable of type '`ptr int`'. Similarly, a value of type '`[10] ptr int`', where `BASE_TYPE` is '`ptr int`', can be assigned to a variable of type '`ptr ptr int`'.

Moreover, a value of any C array type can be assigned to a variable of type '`ptr void`'.

For example, a value of type '`[10] ptr int`' can be assigned to a variable of type '`ptr void`'.

3.10 Conclusions

In this chapter, we give an overview of our proposed programming language named Poseidon Lua. Poseidon Lua is designed to bring the C programming paradigm to the Typed Lua programming language. Poseidon Lua imports the concepts of manual memory management and direct memory programming from the C programming language to Typed Lua.

Poseidon Lua extends Typed Lua using language features inspired by the C programming language. Poseidon Lua is implemented by modifying the compiler of Typed Lua.

Poseidon Lua introduces the operators `malloc`, `free`, and `sizeof`. These operators serve similar purposes to their counterparts in the C programming language. `malloc` and `free` are used for manual memory management.

Poseidon Lua also introduces C types: Primitive C types, C pointer types, C struct types, and C array types. The primitive C types are `char`, `int`, `double`, and `bool`. Poseidon Lua allows inter-operation between primitive C types and Typed Lua types. C pointers can be used to perform direct memory programming. C structs can be used to create composite objects. The members of a C struct can belong to a variety of C types. A member of a C struct can also be a C pointer to a C struct. C arrays can be used to organize C values into multi-dimensional arrays. C arrays can interoperate with C pointers. Poseidon Lua performs static typechecking for the C types.

Note that an individual variable cannot be declared to have a primitive C type directly. Only a member of a C struct, an element of a C array, or the base type of a C pointer can be declared to have a primitive C type. In addition, a C array can only be declared as a member of a C struct.

Poseidon Lua uses the assignment operator, (`=`), to automatically convert Typed Lua values to C values. This feature is essential to the inter-operation of C values with a Typed Lua codebase. It facilitates the gradual introduction of C programming into a program that is otherwise written in an optionally-typed programming language.

A Poseidon Lua program is ultimately translated to a Modified Lua program. Modified Lua is our modification of the Lua compiler and virtual machine. We explore the features of Modified Lua in the next chapter.

In Chapter 5, we introduce our proposed Modified LuaFFI library that allows a Poseidon Lua program to call an external C function.

In Chapter 6, we compare the performance of a program written in Poseidon Lua with the performance of a program written in Lua.

Chapter 4

Modified Lua

4.1 Introduction

In Chapter 3, we have proposed the Poseidon Lua programming language. Poseidon Lua extends Typed Lua [18] with operators for manual memory management and C types. Typed Lua is an optionally-typed programming language that provides a static type system for the dynamically-typed programming language named Lua [11].

In this chapter, we propose our extension of Lua named Modified Lua. Modified Lua augments Lua with a set of special operators that can be used to manipulate C values.

A Poseidon Lua program carries static type annotations. These static type annotations are used to perform static typechecking on the program. Then, the Poseidon Lua program is translated to a Modified Lua program.

Modified Lua provides the CS operators, where CS stands for C Semantics. The CS operators allow the manipulation of C values from within the dynamically-typed programming language named Lua. The `CS_malloc` and the `CS_free` operators allow manual memory management through the allocation and deallocation of memory blocks for use by the program.

There are individual CS load and CS store operators that enable the manipulation of C values that belong to any of the following primitive C types: `char`, `int`, `double`, and `bool`. The `CS_loadPointer` and `CS_storePointer` operators enable the manipulation of C pointers. The `CS_loadOffset` operator enables the manipulation of C arrays. The `CS_loadString` and `CS_storeString` operators enable the manipulation of null-terminated C strings.

We explore in this chapter the CS operators that are provided by Modified Lua and how a Poseidon Lua program is translated to make use of them. In Section 4.2, the features of Modified Lua and their implementations are described. Modified Lua is implemented by augmenting the compiler and the virtual machine (VM) of Lua.

The rest of this chapter is structured as follows. In Section 4.3, we discuss how Modified Lua calculates the size, in bytes, of a C struct as well as the offset, in bytes, of each member of a C struct. In Section 4.4, we discuss the workings of the `CS_malloc` operator and the `CS_free` operator. Next, we discuss the `CS_loadChar`, `CS_storeChar`, `CS_loadInt`, `CS_storeInt`, `CS_loadDouble`, `CS_storeDouble`, `CS_loadBool`, and `CS_storeBool` operators in Section 4.5. Then, we discuss how the `CS_loadPointer` and `CS_storePointer` operators enable the manipulation of C pointers in Section 4.6. In Section 4.7, we discuss how the `CS_loadOffset` operator enables the manipulation of C arrays. In Section 4.8, we discuss how the `CS_loadString` and `CS_storeString` operators enable the manipulation of null-terminated C strings. In Section 4.9, we discuss the use of `cs.NULL` as the NULL pointer. Finally, we conclude the chapter in Section 4.10.

4.2 Modified Lua Features

Poseidon Lua provides operators that allow a Poseidon Lua program to perform manual memory management. Poseidon Lua also provides the facilities to perform assignment (`=`), member-access (`.`), and pointer-dereference (`[]`) with respect to C values. The extension of the static type system of Typed Lua with C types enables the use of C pointers for direct memory programming. However, a Poseidon Lua program cannot be compiled to a corresponding Lua program the same way that a Typed Lua program can be compiled to a corresponding Lua program. This is due to the fact that Lua does not provide the language facilities that are necessary to implement manual memory management and direct memory programming. Therefore, a Poseidon Lua program must be compiled to a different target programming language that does provide these facilities.

Hence, Modified Lua extends the Lua programming language with CS operators. These CS operators allow a Modified Lua program to perform manual memory management. Other CS operators enable the manipulation of C values. In particular, the ability to manipulate C pointers allows a Modified Lua program to perform direct memory programming. Thus, a Poseidon Lua program is always compiled to a Modified Lua program.

Note that Typed Lua only provides a static typechecker for Lua. It is not integrated with any Lua VM. Since Poseidon Lua extends Typed Lua, it is also not integrated with

any Lua VM. As a result, Poseidon Lua and the Modified Lua VM are necessarily two separate and distinct components.

4.2.1 Feature Description

As stated earlier, Modified Lua provides CS operators. Modified Lua also provides the NULL pointer. The relevant features are listed below.

1. Operators for manual memory management: `CS_malloc` and `CS_free`
2. Operators for inter-operation between primitive C types and their corresponding Lua types:
   (a) For `char`: `CS_loadChar` and `CS_storeChar`
   (b) For `int`: `CS_loadInt` and `CS_storeInt`
   (c) For `double`: `CS_loadDouble` and `CS_storeDouble`
   (d) For `bool`: `CS_loadBool` and `CS_storeBool`
3. Operators for C pointer types: `CS_loadPointer` and `CS_storePointer`
4. Operator for C array types: `CS_loadOffset`
5. Operators for null-terminated C strings: `CS_loadString` and `CS_storeString`
6. NULL pointer: `cs.NULL`

The Lua programming language provides a `light-userdata` value [11]. It contains a pointer, of type '`void *`', to a block of memory that is not automatically garbage collected by the Lua VM. Modified Lua uses the `light-userdata` value to represent a C pointer to a block of memory that is manually allocated and deallocated by the programmer.

Modified Lua enables manual memory management through the `CS_malloc` and the `CS_free` operators. The `CS_malloc` operator takes a Lua `integer` value as an argument. Modified Lua uses the Lua `integer` value to represent the size, in bytes, of a block of memory. The `CS_malloc` operator allocates a block of memory of the given size and returns a C pointer to that block of memory.

The `CS_free` operator takes a C pointer as an argument and deallocates the block of memory that is pointed to by the C pointer. This block of memory must have been previously allocated by the `CS_malloc` operator.

Modified Lua supports inter-operation between values of primitive C types and values of Lua types. In order to achieve this, Modified Lua automatically converts a value of a primitive C type to a value of a corresponding Lua type and vice versa. For this purpose, primitive C types `char`, `int`, `double`, and `bool` correspond to the Lua types `string`, `integer`, `number`, and `boolean`, respectively.

Modified Lua provides CS operators for retrieving a primitive C value from within a block of memory. The CS operators that are used to retrieve values of the primitive C types `char`, `int`, `double`, and `bool` from a block of memory are `CS_loadChar`, `CS_loadInt`, `CS_loadDouble`, and `CS_loadBool`, respectively. Each of these CS operators behaves in a similar fashion. Each of them takes 2 arguments. The first argument to the CS operator is a Lua `light-userdata` value that represents a C pointer to a block of memory. The second argument is a Lua `integer` value that represents the offset, in number of bytes, from the beginning of the block of memory to the location of the primitive C value that is to be retrieved. The CS operator retrieves the primitive C value from the block of memory and automatically converts the value to its corresponding Lua value and returns the Lua value as the final result.

Modified Lua also provides CS operators for storing a primitive C value within a block of memory. The CS operators that are used to store values of the primitive C types `char`, `int`, `double`, and `bool` within a block of memory are `CS_storeChar`, `CS_storeInt`, `CS_storeDouble`, and `CS_storeBool`, respectively. Each of these CS operators behaves in a similar fashion. Each of them takes 3 arguments. The first argument is a Lua `light-userdata` value that represents a C pointer to a block of memory. The second argument is a Lua `integer` value that represents the offset, in number of bytes, from the beginning of the block of memory to the location where the primitive C value is to be stored. The third argument is a Lua value that represents a primitive C value. The CS operator automatically converts the third argument to its corresponding primitive C value and stores the converted value to the given offset within the block of memory. Note that in case of `CS_storeChar`, only the first character of the third argument, which is a Lua `string` value, is converted to a `char` value. For each of these CS operators, the first argument is returned as the final result.

Modified Lua provides 2 CS operators that enable the manipulation of C pointers. These are the `CS_loadPointer` and the `CS_storePointer` operators. These CS operators behave in an analogous manner to the CS operators for the primitive C types. However, they do not perform automatic conversions between C and Lua values. The `CS_loadPointer` operator retrieves a C pointer value from within a block of memory. It takes 2 arguments. The first argument is a Lua `light-userdata` value that represents a C pointer to a block of memory. The second argument is a Lua `integer` value that represents

the offset, in number of bytes, from the beginning of the block of memory to the location of the C pointer value that is to be retrieved. The CS operator retrieves the C pointer value from the block of memory and returns it encapsulated within a Lua `light-userdata` value.

The `CS_storePointer` operator stores a C pointer value within a block of memory. It takes 3 arguments. The first argument is a Lua `light-userdata` value that represents a C pointer to a block of memory. The second argument is a Lua `integer` value that represents the offset, in number of bytes, from the beginning of the block of memory to the location where the C pointer value is to be stored. The third argument is a Lua `light-userdata` value that represents the C pointer that is to be stored. The CS operator stores the C pointer value that is represented by the third argument to the given offset within the block of memory. The first argument is returned as the final result.

Modified Lua provides the `CS_loadOffset` operator for the purpose of allowing the manipulation of C arrays. It takes 2 arguments. The first argument is a Lua `light-userdata` value that represents a C pointer to a block of memory. The second argument is a Lua `integer` value that represents the offset, in number of bytes, from the beginning of the block of memory to the location of a C array. The CS operator returns a C pointer to the beginning of the C array encapsulated within a Lua `light-userdata` value.

Note that the C pointer returned by the `CS_loadOffset` operator can be passed in as the first argument to the appropriate CS load operator to retrieve an element of the C array that belongs to a primitive C type or a C pointer type. Similarly, the C pointer can be passed in as the first argument to the appropriate CS store operator to set the value of an element of the C array that belongs to a primitive C type or a C pointer type.

Modified Lua provides 2 CS operators that enable the manipulation of null-terminated C strings. These are the `CS_loadString` operator and the `CS_storeString` operator. These CS operators behave in an analogous manner to the CS operators for the primitive C types. The `CS_loadString` operator retrieves a null-terminated C string value from within a block of memory. It takes 2 arguments. The first argument is a Lua `light-userdata` value that represents a C pointer to a block of memory. The second argument is a Lua `integer` value that represents the offset, in number of bytes, from the beginning of the block of memory to the location of the null-terminated C string value that is to be retrieved. The CS operator retrieves the null-terminated C string value from the block of memory and automatically converts it to a corresponding Lua `string` value. This converted value is returned as the final result.

The `CS_storeString` operator stores a null-terminated C string value within a block of memory. It takes 3 arguments. The first argument is a Lua `light-userdata` value that

represents a C pointer to a block of memory. The second argument is a Lua `integer` value that represents the offset, in number of bytes, from the beginning of the block of memory to the location where the null-terminated C string value is to be stored. The third argument is a Lua `string` value. The CS operator automatically converts the third argument to its corresponding null-terminated C string value and stores the converted value to the given offset within the block of memory. The first argument is returned as the final result.

Modified Lua provides a NULL pointer in the form of `cs.NULL`. It can be assigned to a C pointer variable to set its value to NULL. It can also be used to check if the value of a C pointer variable is NULL.

4.2.2 Feature Implementation

We implement the features of Modified Lua by modifying the compiler and the virtual machine (VM) of Lua. The Lua compiler is modified so that it may recognize the CS operators provided by Modified Lua. In addition, the Lua virtual machine is modified to integrate new CS opcode instructions in order to implement the CS operators. Note that the Lua compiler and the Lua VM are both implemented in C.

There are 15 different CS operators in Modified Lua. Each of these CS operators is compiled to a corresponding CS opcode instruction by the Modified Lua compiler. Thus, there are 15 different CS opcode instructions. The correspondence between CS operators and CS opcode instructions is given below.

1. The CS operators `CS_malloc` and `CS_free` are compiled to the CS opcode instructions `OP_CS_MALLOC` and `OP_CS_FREE`, respectively
2. The CS operators `CS_loadChar` and `CS_storeChar` are compiled to the CS opcode instructions `OP_CS_LOAD_CHAR` and `OP_CS_STORE_CHAR`, respectively
3. The CS operators `CS_loadInt` and `CS_storeInt` are compiled to the CS opcode instructions `OP_CS_LOAD_INT` and `OP_CS_STORE_INT`, respectively
4. The CS operators `CS_loadDouble` and `CS_storeDouble` are compiled to the CS opcode instructions `OP_CS_LOAD_DOUBLE` and `OP_CS_STORE_DOUBLE`, respectively
5. The CS operators `CS_loadBool` and `CS_storeBool` are compiled to the CS opcode instructions `OP_CS_LOAD_BOOL` and `OP_CS_STORE_BOOL`, respectively

6. The CS operators `CS_loadPointer` and `CS_storePointer` are compiled to the CS opcode instructions `OP_CS_LOAD_POINTER` and `OP_CS_STORE_POINTER`, respectively

7. The CS operator `CS_loadOffset`, for C array type, is compiled to the CS opcode instruction `OP_CS_LOAD_OFFSET`

8. The CS operators `CS_loadString` and `CS_storeString` are compiled to the CS opcode instructions `OP_CS_LOAD_STRING` and `OP_CS_STORE_STRING`, respectively

The Lua compiler performs lexical analysis, parsing, and code generation simultaneously in the same pass of the compiler instead of performing each task separately. We modify the parser so that whenever it parses a function call expression, the name of the function being called is checked against the names of the CS operators. If there is a match, the function call expression is parsed as a CS operator call expression instead. First, each expression that denotes an argument of the CS operator is parsed and the relevant opcode instructions are generated to evaluate the expression and produce its value. Next, the CS opcode instruction that corresponds to the CS operator being parsed is generated for the purpose of evaluating the CS operator call expression and returning its final value.

We have implemented the code generation for a CS opcode instruction in such a way that the modified Lua compiler treats the CS opcode instruction as if it is the Lua `OP_CALL` opcode instruction that is usually generated for a function call expression. As a result, the rest of the compilation process treats a CS operator call expression as if it is a normal function call expression.

However, we have to deal with a subtle problem with this approach to code generation. When a Lua return statement is parsed, the original Lua parser code checks whether the value being returned is the result of a function call and whether that is the only value being returned. If so, the Lua `OP_CALL` opcode instruction for the function call is replaced by a Lua `OP_TAILCALL` opcode instruction for a tailcall to the same function. In the situation where one function tailcalls another function, the run-time context of the function doing the tailcalling is replaced by the run-time context of the function that is being tailcalled. In this case, a CS opcode instruction does not work the exact same way as a Lua `OP_CALL` opcode instruction. Thus, when the original Lua parser code attempts to replace a CS opcode instruction with a Lua `OP_TAILCALL` opcode instruction, it introduces errors. Thus, we have modified the Lua parser so that during the code generation for a tailcall from a CS opcode instruction, the modified Lua parser does not replace the CS opcode instruction with a Lua `OP_TAILCALL` opcode instruction.

The Lua virtual machine (VM) runs as follows. The Lua VM executes Lua opcode instructions sequentially using a long interpreter loop and keeps the Lua values that are

manipulated by the Lua opcode instructions on a VM stack. The interpreter loop identifies the current Lua opcode instruction. Then, the implementation code for the current Lua opcode instruction is executed. The implementation code manipulates the Lua values on the VM stack. After this, the next Lua opcode instruction is fetched and the interpreter loop is executed again. We modify the interpreter loop to include support for the CS opcode instructions.

The input arguments for a CS opcode instruction are pushed onto the VM stack. Any Lua value that is returned by the CS opcode instruction is also pushed onto the VM stack.

The `OP_CS_MALLOC` opcode instruction takes as argument a Lua `integer` value that specifies the size, in bytes, of a block of memory to be allocated. This size information is then popped from the VM stack and used with the `malloc` function of the C programming language to allocate a block of memory of the required size. As an aside, note that the Lua VM is itself implemented in the C programming language. A pointer to the allocated block of memory is then stored inside a Lua `light-userdata` value that is pushed onto the VM stack to be returned.

The `OP_CS_FREE` opcode instruction takes as argument a Lua `light-userdata` value that contains a pointer to a block of memory. The Lua `light-userdata` value is popped from the VM stack. The pointer is extracted form the Lua `light-userdata` value and used with the `free` function of the C programming langauge to deallocate the block of memory. A Lua `light-userdata` value that contains a NULL pointer is pushed onto the VM stack to be returned. This is done to ensure that the CS operator does return a value and behaves in a predictable manner if its return value is used elsewhere in the program. Note that the NULL pointer in the Lua `light-userdata` value is set from the underlying C programming language code that implements `OP_CS_FREE`.

The CS load opcode instructions for primitive C types are as follows: `OP_CS_LOAD_CHAR`, `OP_CS_LOAD_INT`, `OP_CS_LOAD_DOUBLE`, and `OP_CS_LOAD_BOOL`. These CS load opcode instructions can be used to retrieve the value of a primitive C type which may be a member of a C struct or an element of a C array. These CS load opcode instructions retrieve a value from within a block of memory. Each of these CS opcode instructions takes 2 input arguments. The first argument is a Lua `light-userdata` value that contains a pointer to a block of memory. The second argument is a Lua `integer` value that represents an offset, in number of bytes, from the beginning of the block of memory. These argument values are popped from the VM stack and used to perform pointer arithmetic in the C programming language to obtain a pointer to the location of the value to be retrieved from within the block of memory. The value is retrieved and stored in a corresponding Lua value and the Lua value is pushed onto the VM stack to be returned. The CS

load opcode instructions `OP_CS_LOAD_CHAR`, `OP_CS_LOAD_INT`, `OP_CS_LOAD_DOUBLE`, and `OP_CS_LOAD_BOOL` ultimately return a Lua `string` value, a Lua `integer` value, a Lua `number` value, and a Lua `boolean` value, respectively.

The CS store opcode instructions for primitive C types are given as `OP_CS_STORE_CHAR`, `OP_CS_STORE_INT`, `OP_CS_STORE_DOUBLE`, and `OP_CS_STORE_BOOL`. These CS store opcode instructions store a value within a block of memory. Each of these CS opcode instructions takes 3 input arguments. The first argument is a Lua `light-userdata` value that contains a pointer to a block of memory. The second argument is a Lua `integer` value that represents an offset, in number of bytes, from the beginning of the block of memory. The third argument is a Lua value. The CS store opcode instructions `OP_CS_STORE_CHAR`, `OP_CS_STORE_INT`, `OP_CS_STORE_DOUBLE`, and `OP_CS_STORE_BOOL` take as the third argument a Lua `string` value, a Lua `integer` value, a Lua `number` value, and a Lua `boolean` value, respectively. The argument values are popped from the VM stack. The first two argument values are used to perform pointer arithmetic in the C programming language to obtain a pointer to a location within the block of memory. The third argument value is stored at that location within the block of memory. Then, the first argument value is pushed onto the VM stack to be returned.

There are 2 CS opcode instructions that manipulate C pointers: `OP_CS_LOAD_POINTER` and `OP_CS_STORE_POINTER`. The `OP_CS_LOAD_POINTER` opcode instruction takes 2 input arguments. The first argument is a Lua `light-userdata` value that contains a pointer to a block of memory. The second argument is a Lua `integer` value that represents an offset, in number of bytes, from the beginning of the block of memory. These argument values are popped from the VM stack and used to perform pointer arithmetic in the C programming language to obtain a pointer to the location of the value to be retrieved from within the block of memory. The value is retrieved and stored in a Lua `light-userdata` value and the Lua value is pushed onto the VM stack to be returned.

The `OP_CS_STORE_POINTER` opcode instruction takes 3 input arguments. The first argument is a Lua `light-userdata` value that contains a pointer to a block of memory. The second argument is a Lua `integer` value that represents an offset, in number of bytes, from the beginning of the block of memory. The third argument is a Lua `light-userdata` value that contains a pointer to another block of memory. The argument values are popped from the VM stack. The first two argument values are used to perform pointer arithmetic in the C programming language to obtain a pointer to a location within the block of memory that is pointed to by the first argument. The third argument value is stored at that location. Then, the first argument value is pushed onto the VM stack to be returned.

The CS opcode instruction `OP_CS_LOAD_OFFSET` is provided for the manipulation of C

arrays. It takes 2 input arguments. The first argument is a Lua `light-userdata` value that contains a pointer to a block of memory. The second argument is a Lua `integer` value that represents an offset, in number of bytes, from the beginning of the block of memory. These argument values are popped from the VM stack and used to perform pointer arithmetic in the C programming language to obtain a pointer to a location within the block of memory. The obtained pointer value is stored in a Lua `light-userdata` value and the Lua value is pushed onto the VM stack to be returned.

There are 2 CS opcode instructions that manipulate null-terminated C strings as follows: `OP_CS_LOAD_STRING` and `OP_CS_STORE_STRING`. The CS opcode instruction for loading strings `OP_CS_LOAD_STRING` takes 2 input arguments. The first argument is a Lua `light-userdata` value that contains a pointer to a block of memory. The second argument is a Lua `integer` value that represents an offset, in number of bytes, from the beginning of the block of memory. These argument values are popped from the VM stack and used to perform pointer arithmetic in the C programming language to obtain a pointer to the location of the null-terminated C string value to be retrieved from within the block of memory. The null-terminated C string value is retrieved and converted into a Lua `string` value and the Lua `string` value is pushed onto the VM stack to be returned.

The `OP_CS_STORE_STRING` opcode instruction takes 3 input arguments. The first argument is a Lua `light-userdata` value that contains a pointer to a block of memory. The second argument is a Lua `integer` value that represents an offset, in number of bytes, from the beginning of the block of memory. The third argument is a Lua `string` value. The argument values are popped from the VM stack. The first two argument values are used to perform pointer arithmetic in the C programming language to obtain a pointer to a location within the block of memory. The third argument value is converted to a null-terminated C string value and stored at that location within the block of memory. Then, the first argument value is pushed onto the VM stack to be returned.

Modified Lua provides a NULL pointer in the form of `cs.NULL` for two purposes: 1) to set the value of a C pointer to NULL and 2) to check whether the value of a C pointer is NULL or not. The value of `cs.NULL` can be sent as the third argument to the CS operator `CS_storePointer` to store a NULL pointer. Using the equality operator (`==`), we can check whether a Lua `light-userdata` value that is returned by the CS operator `CS_loadPointer` has the same value as `cs.NULL`.

We implement `cs.NULL` by modifying the part of the Lua VM that loads the Lua standard libraries before running a Lua program. Generally, these libraries contain functions and constants that are implemented in the C programming language. However, in a Lua program, these libraries appear as Lua tables whose members represent the functions and

constants that are contained within the libraries. In the same manner as these pre-existing standard libraries, we implement a new library named `cs` that has only one constant member named `NULL`. The value of `cs.NULL` is set to a Lua `light-userdata` value that contains the NULL pointer from the C programming language. The Lua VM is modified so that the `cs` library is loaded and made accessible to a Lua program the same way that the standard libraries are loaded and made accessible.

The source code for the Modified Lua VM is available at GitHub as a component of Poseidon Lua [15].

4.3 Size and offset

```
1 struct House
2         unit : char
3         num_rooms : int
4         area_feet2 : double
5         on_sale : bool
6         num_windows_in_room : ptr int
7         num_rooms_in_floor : [2] int
8 end --end House
```

Figure 4.1: implementation.tl, Part 1 of 6

In Figure 4.1, lines 1 - 8 show the definition of the C struct type named `House`, which is used in the rest of the examples for demonstrative purposes. The size of the C struct type `House` is the sum of the sizes of all its members. Therefore, the size of `House` is `1 + 4 + 8 + 4 + 8 + ( 2 * 4 ) = 33` bytes.

The offset of each member of the C struct type `House` is the sum of the sizes of all the previous members. Thus, the C struct members `unit`, `num_rooms`, `area_feet2`, `on_sale`, `num_windows_in_room`, and `num_rooms_in_floor` have offsets 0, 1, 5, 13, 17, and 25 bytes, respectively.

4.4 The `CS_malloc` and `CS_free` operators

Poseidon Lua translates an invocation of its `malloc` operator to an invocation of the `CS_malloc` operator of Modified Lua. The `CS_malloc` operator of Modified Lua takes as

its input a Lua `integer` value that specifies the number of bytes that should be allocated. It allocates a block of memory of the required size and returns a Lua `light-userdata` value that points to this block of memory.

Poseidon Lua translates an invocation of its `free` operator to an invocation of the `CS_free` operator of Modified Lua. The `CS_free` operator of Modified Lua takes as its input a Lua `light-userdata` value that points to the block of memory that is to be deallocated. It deallocates this block of memory and returns a Lua `light-userdata` value that points to NULL.

```
1 --[[ struct ]]
2 local house_1 : ptr House = malloc( ptr House )
3
4 free( house_1 )
```

Figure 4.2: implementation.tl, Part 2 of 6

```
1 local house_1 = CS_malloc(33)
2
3 CS_free(house_1)
```

Figure 4.3: Part of implementation.lua translated from implementation.tl Part 2 of 6

In Figure 4.2, the invocations of the `malloc` and `free` operators of Poseidon Lua are shown. Note that '`malloc( ptr House )`' is the same as '`malloc( sizeof( House ) )`', with the exception that the returned value has type '`ptr House`' instead of '`ptr void`'.

In Figure 4.3, the invocations of the `CS_malloc` and `CS_free` operators of the translated Modified Lua program are shown.

4.5 The primitive C types

Modified Lua provides CS load and CS store operators to manipulate C values belonging to each of the following primitive C types: `char`, `int`, `double`, `bool`. Each of the CS load operators can be used to retrieve a C value that is stored at a particular location in a block of memory and convert it to a corresponding Lua value and return that Lua value. Each

of the CS store operators can take a Lua value as its third argument and convert it to a corresponding C value and store that C value at a particular location in a block of memory. Note that the values of the primitive C types `char`, `int`, `double`, and `bool` correspond to the values of the Lua types `string`, `integer`, `number`, and `boolean`, respectively.

```
--[[ struct ]]
local house_1 : ptr House = malloc( ptr House )

house_1.unit = "A"
house_1.num_rooms = 4
house_1.area_feet2 = 1000.25
house_1.on_sale = true

local str = ""
str = str .. "house_1.unit : "
str = str .. tostring( house_1.unit )
str = str .. "\n"
str = str .. "house_1.num_rooms : "
str = str .. tostring( house_1.num_rooms )
str = str .. "\n"
str = str .. "house_1.area_feet2 : "
str = str .. tostring( house_1.area_feet2 )
str = str .. "\n"
str = str .. "house_1.on_sale : "
str = str .. tostring( house_1.on_sale )
print( str )
```

Figure 4.4: implementation.tl, Part 3 of 6

In Figure 4.4, a Poseidon Lua program is shown. In this program, values are assigned to variables of primitive C types. Then, the values of these variables are accessed.

In Figure 4.5, the translated Modified Lua program is shown. In this program, the assignment of values to variables of primitive C types is translated to invocations of the following CS store operators: `CS_storeChar`, `CS_storeInt`, `CS_storeDouble`, `CS_storeBool`. The accessing of the values of these variables is translated to the invocations of the following CS load operators: `CS_loadChar`, `CS_loadInt`, `CS_loadDouble`, `CS_loadBool`.

```
1  local house_1 = CS_malloc(33)
2  do
3  CS_storeChar(house_1,0,"A") end
4  do
5  CS_storeInt(house_1,1,4) end
6  do
7  CS_storeDouble(house_1,5,1000.25) end
8  do
9  CS_storeBool(house_1,13,true) end
10
11 local str = ""
12 str = str .. "house_1.unit : "
13 str = str .. tostring(CS_loadChar(house_1,0))
14 str = str .. "\n"
15 str = str .. "house_1.num_rooms : "
16 str = str .. tostring(CS_loadInt(house_1,1))
17 str = str .. "\n"
18 str = str .. "house_1.area_feet2 : "
19 str = str .. tostring(CS_loadDouble(house_1,5))
20 str = str .. "\n"
21 str = str .. "house_1.on_sale : "
22 str = str .. tostring(CS_loadBool(house_1,13))
23 print(str)
```

Figure 4.5: Part of implementation.lua translated from implementation.tl, Part 3 of 6

4.6 The C pointer types

Modified Lua provides 2 CS operators to manipulate C pointers: `CS_loadPointer` and `CS_storePointer`. The `CS_loadPointer` operator retrieves a C pointer value that is stored at a particular location in a block of memory. The `CS_storePointer` operator stores a C pointer value at a particular location in a block of memory.

In Figure 4.6, a Poseidon Lua program is shown. In this program, a value is assigned to `house_1.num_windows_in_room`, a variable of a C pointer type, in line 6. Then, this variable is indexed at different offsets to assign values to those locations in memory.

In Figure 4.7, the translated Modified Lua program is shown. In this program, the assignment of a value to a variable of a C pointer type is translated to an invocation of the `CS_storePointer` operator. The indexing of this variable is translated to an invocation of

```
--[[ struct ]]
local house_1 : ptr House = malloc( ptr House )

local num_rooms = 4
local pointer_1 : ptr int = malloc( num_rooms * sizeof( int ) )
house_1.num_windows_in_room = pointer_1
house_1.num_windows_in_room[0] = 1
house_1.num_windows_in_room[1] = 2
house_1.num_windows_in_room[2] = 3
house_1.num_windows_in_room[3] = 4
```

Figure 4.6: implementation.tl, Part 4 of 6

```
local house_1 = CS_malloc(33)

local num_rooms = 4
local pointer_1 = CS_malloc(num_rooms * 4)
do
CS_storePointer(house_1,17,pointer_1) end
do
CS_storeInt(CS_loadPointer(house_1,17),4 * 0,1) end
do
CS_storeInt(CS_loadPointer(house_1,17),4 * 1,2) end
do
CS_storeInt(CS_loadPointer(house_1,17),4 * 2,3) end
do
CS_storeInt(CS_loadPointer(house_1,17),4 * 3,4) end
```

Figure 4.7: Part of implementation.lua translated from implementation.tl, Part 4 of 6

the `CS_loadPointer` operator.

4.7 The C array types

Modified Lua provides the `CS_loadOffset` operator to manipulate C arrays. A C array can only be declared as a member of a C struct. Thus, the memory space for a C array is

always allocated as part of the memory allocation for the C struct. The `CS_loadOffset` operator allows the program to refer to the location in memory of a particular element of the C array.

```
--[[ struct ]]
local house_1 : ptr House = malloc( ptr House )

house_1.num_rooms_in_floor[0] = 2
house_1.num_rooms_in_floor[1] = 2
```

Figure 4.8: implementation.tl, Part 5 of 6

```
local house_1 = CS_malloc(33)

do
CS_storeInt(CS_loadOffset(house_1,25),4 * 0,2) end
do
CS_storeInt(CS_loadOffset(house_1,25),4 * 1,2) end
```

Figure 4.9: Part of implementation.lua translated from implementation.tl, Part 5 of 6

In Figure 4.8, a Poseidon Lua program is shown. In this program, a variable of a C array type is indexed at different offsets in order to assign values to the appropriate memory locations.

In Figure 4.9, the translated Modified Lua program is shown. In this program, the indexing of a variable of a C array type is translated to an invocation of the `CS_loadOffset` operator.

4.8 The C strings

Modified Lua provides 2 CS operators, `CS_loadString` and `CS_storeString`, to manipulate null-terminated C strings. The `CS_loadString` operator retrieves a null-terminated C string value that is stored at a particular location in a block of memory and converts it to a Lua `string` value and returns the Lua `string` value. The `CS_storeString` operator

takes a Lua `string` value as its third argument and converts it to a null-terminated C string and stores that C string at a particular location in a block of memory.

```
1  --[[ string ]]
2  local hello_str : string = "Hello World"
3  local str_size : integer = string.len( hello_str ) + 1
4  local pointer_2 : ptr char = malloc( str_size * sizeof( char ) )
5
6  local left_1 : integer = 1
7  local right_1 : integer = 3
8
9  left_1, pointer_2, right_1 = right_1, hello_str, left_1
10
11 local received_str : string = "None"
12
13 local left_2 : integer = 1
14 local right_2 : integer = 3
15
16 left_2, received_str, right_2 = right_2, pointer_2, left_2
```

Figure 4.10: implementation.tl, Part 6 of 6

In Figure 4.10, a Poseidon Lua program is shown. In this program, a value of type `string` is assigned to a variable of type '`ptr char`' in a multiple assignment statement in line 9. Then, a value of type '`ptr char`' is assigned to a variable of type `string` in another multiple assignment statement in line 16.

In Figure 4.11, the translated Modified Lua program is shown. In this program, the assignment of a value of type `string` to a variable of type '`ptr char`' is translated to an invocation of the `CS_storeString` operator in line 9. The assignment of a value of type '`ptr char`' to a variable of type `string` is translated to an invocation of the `CS_loadString` operator in line 18. Note that for both multiple assignment statements, the single assignment (SA) semantics is used.

4.9 The `cs.NULL` pointer

Modified Lua provides a NULL pointer within a new standard library named `cs`. The NULL pointer can be accessed as `cs.NULL`. Thus, we do not need any special CS operators

```
local hello_str = "Hello World"
local str_size = string.len(hello_str) + 1
local pointer_2 = CS_malloc(str_size * 1)

local left_1 = 1
local right_1 = 3
do
left_1 = right_1
CS_storeString(pointer_2,0,hello_str)
right_1 = left_1 end

local received_str = "None"

local left_2 = 1
local right_2 = 3
do
left_2 = right_2
received_str = CS_loadString(pointer_2,0)
right_2 = left_2 end
```

Figure 4.11: Part of implementation.lua translated from implementation.tl, Part 6 of 6

for loading or storing this NULL pointer.

In Figure 4.12, a Poseidon Lua program is shown. In this program, a variable of a C pointer type is set to the NULL value. Then, the value of this variable is checked using the equality operator (`==`) to determine if its value has been successfully set to the NULL value. Finally, the program prints the value of the variable '`is_NULL`'.

In Figure 4.13, the translated Modified Lua program is shown. In this program, the value of `cs.NULL` is used to set the value of a variable to the NULL value and also to check that the value of the variable has been successfully set to the NULL value.

4.10 Conclusions

In this chapter, we give an overview of the different CS operators that are provided by Modified Lua and how a Poseidon Lua program can be translated to make use of them. Modified Lua extends Lua with CS operators that can be used to manipulate C values.

```
--[[ NULL pointers ]]
local pointer_23 : ptr int = malloc( ptr int )
pointer_23[0] = 400

pointer_23 = cs.NULL

local is_NULL : boolean = false
if pointer_23 == cs.NULL then
   is_NULL = true
end --end if

print( "is_NULL : " .. tostring( is_NULL ) )
```

Figure 4.12: pointers.tl

```
local pointer_23 = CS_malloc(4)
do
CS_storeInt(pointer_23,4 * 0,400) end

pointer_23 = cs.NULL

local is_NULL = false
if pointer_23 == cs.NULL then
  is_NULL = true
end

print("is_NULL : " .. tostring(is_NULL))
```

Figure 4.13: pointers.lua translated from pointers.tl

Modified Lua is implemented by augmenting both the Lua compiler and the Lua virtual machine (VM).

Modified Lua provides 2 operators for manual memory management: `CS_malloc` and `CS_free`. Modified Lua provides 4 operators to store a value of a primitive C type to a location in memory: `CS_storeChar`, `CS_storeInt`, `CS_storeDouble`, `CS_storeBool`.

Modified Lua provides 4 operators to load a value of a primitive C type from a location in memory: `CS_loadChar`, `CS_loadInt`, `CS_loadDouble`, `CS_loadBool`. Modified Lua provides 2 operators for C pointer manipulation: `CS_storePointer` and `CS_loadPointer`. Modified Lua provides the `CS_loadOffset` operator for C array manipulation. Modified Lua provides 2 operators for null-terminated C string manipulation: `CS_storeString` and `CS_loadString`.

Modified Lua also provides a NULL pointer in the form of `cs.NULL`. For this purpose, we provide a library named `cs` that is loaded by the Modified Lua VM along with all the other Lua standard libraries.

Even with the ability to manipulate C values using the CS operators that are provided by Modified Lua, it may be impossible to match the performance advantage that is provided by functions that are written purely in the C programming language. Thus, Poseidon Lua provides a Modified LuaFFI library that is able to call external C functions that are written in the C programming language. We cover the features of the Modified LuaFFI library in the next chapter.

In Chapter 6, we compare the performance of a program written in Poseidon Lua with the performance of a program written in Lua.

Chapter 5

Modified LuaFFI Library

5.1 Introduction

In this chapter, we introduce our proposed foreign function interface (FFI) library called Modified LuaFFI, which is also known as `luaffi_cs`. The Modified LuaFFI library represents our modification of the existing `luaffifb` FFI library [16][17]. The `luaffifb` library enables a Lua program to call an external C function. Similarly, the Modified LuaFFI library enables a Poseidon Lua program to call an external C function.

When a Lua program uses the `luaffifb` library to call an external C function, the value that is returned to the Lua program depends on the C value that is returned by the external C function. If the external C function returns a primitive C value, such as an `int` or `double`, the `luaffifb` library converts this primitive C value to a corresponding Lua value and returns it. However, If the external C function returns a composite C value, such as a C struct or a C pointer, the `luaffifb` library returns a `cdata` value back to the Lua program. The `cdata` value consists of a C value element and a type tag. The `cdata` value is needed to facilitate the interaction between the composite C value and other Lua values in the Lua program. When a component of the composite C value is accessed within the Lua program, it must be converted to a corresponding Lua value before it can be used in the Lua program. Similarly, when the content of a Lua value is to be placed inside the composite C value, it must be converted to a corresponding C value first. Either conversion step at a `cdata` use-site requires dynamic typechecking using its type tag.

The Modified LuaFFI library alters the functionality of the `luaffifb` library in such a way that whenever an external C function returns a C pointer value, it returns the C

pointer without encapsulating it inside a `cdata` value. It is possible to assign this value to a variable that belongs to a C pointer type in Poseidon Lua. From this point onwards, the static typechecking capabilities of Poseidon Lua can be harnessed to ensure the correct usage of this value inside the Poseidon Lua program. Moreover, Poseidon Lua is able to leverage its static typechecking capabilities to perform conversions between Lua values and C values without incurring the performance overhead of extra dynamic typechecking.

Note that when the return value of an external C function is a non-primitive C value other than a C pointer value such as a C struct value, our Modified LuaFFI library and the existing `luaffifb` library both return a `cdata` value.

The rest of this chapter is structured as follows. An overview of the existing Lua FFI library `luaffifb` is given in Section 5.2. In Section 5.3, the features and implementation details of the Modified LuaFFI library are provided. In Section 5.4, we show how the Modified LuaFFI library can be used by a Poseidon Lua program to call an external C function in order to verify that the Modified LuaFFI library is implemented correctly. Finally, we conclude the chapter in Section 5.5.

5.2 The Foreign Function Interface (FFI) Library

In general, an FFI library allows a program written in one programming language to make use of a library that is written in a different programming language. In particular, the `luaffifb` library enables a Lua program to call an external C function.

Note that the C programming language [12] is a statically-typed programming language. Thus, all typechecking is performed at compile-time. As a result, C values can be efficiently manipulated at run-time without the need for any extra run-time typechecking. Since Lua is a dynamically-typed programming language, it performs all of its typechecking at run-time. Thus, it may be preferable for a Lua program to call an external C function to carry out a particular task for the purpose of avoiding the performance costs related to dynamic typechecking. This capability is exactly the service that is provided by the `luaffifb` library.

A `cdata` value performs dynamic typechecking for the purpose of converting a C value to an appropriate Lua value and vice versa. However, the performance costs of these dynamic typechecks can negate any performance benefits gained through the call to the external C function. One possible remedy would be to use static typechecking to avoid the performance overhead from the dynamic typechecking that is carried out as part of the value conversions. However, the `luaffifb` library is unable to take advantage of

the performance benefits that stem from the use of static typechecking because Lua is a dynamically-typed programming language and does not require any static type annotations from the programmer.

The performance overhead of executing these dynamic typechecks could be avoided by utilizing the static typechecking facility for C types that is provided by Poseidon Lua. Poseidon Lua is able to perform static typechecking at compile-time because it does have access to static type annotations. Thus, Poseidon Lua is able to manipulate its C values without the need to perform any extra run-time typechecking. Furthermore, Poseidon Lua is able to use its static typechecking capabilities to perform automatic conversions between C values and Lua values without executing any dynamic typechecks.

In fact, when a Poseidon Lua program is translated to a Modified Lua program, that Modified Lua program uses type-specific operators to manipulate the C values. Therefore, using variables that belong to the C pointer types of Poseidon Lua to manipulate C values should be more efficient than using the `cdata` values that are provided by `luaffifb` to manipulate C values.

In addition, this approach reduces memory overhead because extra memory does not need to be allocated to hold the `cdata` value.

The Modified LuaFFI library facilitates the use of variables that belong to the C pointer types of Poseidon Lua to manipulate pointers that are returned by external C function calls.

5.3 Modified LuaFFI Library

The Modified LuaFFI library provides to a program written in Poseidon Lua the ability to call an external C function that returns a C pointer and use the returned C pointer in the rest of the program without incurring the performance costs of extra dynamic typechecking.

5.3.1 Modified LuaFFI Library Description

The Modified LuaFFI library extends the `luaffifb` library to provide features that enable the interoperation between Poseidon Lua code and external C functions. This makes it possible to take advantage of the static typechecking for C types that is provided by Poseidon Lua. In this way, the correct use of C values within Poseidon Lua code can be guaranteed without the need for any extra dynamic typechecking.

A value of a C pointer type is represented in Poseidon Lua by a Lua `light-userdata` value that contains the actual C pointer to memory. Thus, a pointer that is returned by an external C function can be placed inside a Lua `light-userdata` value, which can be easily assigned to a variable that belongs to a C pointer type. Then, the static typechecking facility of Poseidon Lua can be used to make sure that the C pointer is used correctly.

5.3.2 Modified LuaFFI Library Implementation

When an external C function returns a value, the `luaffifb` library checks the return type for the external C function. If the return type is a pointer type, a `cdata` value is created and pushed onto the Lua VM stack. The type tag of the `cdata` value is set to the return type and the returned pointer value is placed within the `cdata` value. Then, the `cdata` value is returned back to the Lua program.

The Modified LuaFFI library is implemented as follows. When an external C function returns a value, the Modified LuaFFI library checks the return type for the external C function. If the return type is a C pointer type, the `lua_pushlightuserdata` function of the Lua API is called. It pushes a Lua `light-userdata` value onto the Lua VM stack and places the returned pointer value within the Lua `light-userdata` value. Then, the Lua `light-userdata` value is returned back to the Lua program.

The source code for the Modified LuaFFI library is available at GitHub as a component of Poseidon Lua [15].

5.4 Calling an external C function

Modified LuaFFI gives a Poseidon Lua program the ability to call an external C function and use the returned value as a C value that belongs to a C type of Poseidon Lua. Specifically, if the returned value is a C pointer to a block of memory, then a Lua `light-userdata` value containing that C pointer is returned instead. Since in Poseidon Lua, C values belonging to a C pointer type such as '`ptr int`' are implemented as Lua `light-userdata` values containing a C pointer to a block of memory, the value that is returned by the C function can easily be treated as a C value that belongs to a C pointer type.

In Figure 5.1, the Modified LuaFFI library is loaded using '`require( ''ffi_cs'' )`'. Then, the types for the input values and output values for the external C function named `memcpy` is provided to the library using '`ffi_cs.cdef( def_str )`'.

```
1  local ffi_cs = require( "ffi_cs" )
2
3  local def_str : string = ""
4
5  def_str = def_str .. "void *memcpy ( "
6  def_str = def_str .. "void *dest, "
7  def_str = def_str .. "const void *source, "
8  def_str = def_str .. "size_t num_bytes "
9  def_str = def_str .. ") ;"
10
11 ffi_cs.cdef( def_str )
```

Figure 5.1: ffi.tl, Part 1 of 3

```
1  --[[ ffi ]]
2  local fruit_serving_A = ffi_cs.new( "int[2]" )
3  fruit_serving_A[0] = 101
4  fruit_serving_A[1] = 102
5
6  local sum_A : integer = fruit_serving_A[0] + fruit_serving_A[1]
```

Figure 5.2: ffi.tl, Part 2 of 3

In Figure 5.2, line 2 uses '`ffi_cs.new(``int[2]'')`' to create a `cdata` value that contains an array of 2 `int` elements. The assignment of values in lines 3 - 4 and the accessing of values in line 6 cause the `cdata` value to perform dynamic typechecking in order to convert Lua values to C values and vice versa.

In Figure 5.3, line 1 allocates memory for an array of 2 `int` elements. Lines 3 - 5 call the external C function named `memcpy` to copy values into the allocated array. Line 7 accesses the values that are stored in the array without performing any dynamic typechecking.

```
1 local fruit_serving_B : ptr int = malloc( 2 * sizeof( int ) )
2
3 fruit_serving_B = ffi_cs.C.memcpy( fruit_serving_B,
4                                    fruit_serving_A,
5                                    2 * sizeof( int ) )
6
7 local sum_B : integer = fruit_serving_B[0] + fruit_serving_B[1]
```

Figure 5.3: ffi.tl, Part 3 of 3

5.5 Conclusions

In this chapter, we give an overview of how Modified LuaFFI can be used by a Poseidon Lua program to call an external C function. If the return value of the external C function is a pointer, then Modified LuaFFI returns a C pointer value. This C pointer value can be assigned to a variable that belongs to a C pointer type. From this point onwards, this C pointer value can be manipulated by Poseidon Lua code. Since the use of values that belong to a C pointer type is statically typechecked by Poseidon Lua, there is no need to perform any run-time typechecking to enforce the static type of the C pointer value. Therefore, a Poseidon Lua program is able to use the Modified LuaFFI component to avoid the performance overhead that would otherwise have to be incurred by a Lua program.

In the next chapter, we compare the performance of a program written in Poseidon Lua with the performance of a program written in Lua. We also compare the performance of a program that uses the C pointer types of Poseidon Lua with the performance of a program that uses the `cdata` values of the `luaffifb` library.

Chapter 6

Performance

6.1 Introduction

In this chapter, we measure the performance advantage that can be attained by a Poseidon Lua program with respect to a Lua program. For convenience, a Lua program that does not use the `luaffifb` library is referred to as a `plain Lua` program and a Lua program that does use the `luaffifb` library is referred to as a `luaffifb` program. Furthermore, whenever a Poseidon Lua program uses the Modified LuaFFI library, it is referred to as a Modified LuaFFI program.

In contrast to a `plain Lua` program and a `luaffifb` program, a Poseidon Lua program should be able to use manual memory management to avoid the performance costs related to the use of an automatic garbage collector and it should be able to use direct memory programming to avoid the performance overhead related to the use of Lua tables.

A `cdata` value that is created by the `luaffifb` library performs dynamic typechecking to enforce the static type of the value that it contains. These dynamic typechecks introduce performance costs. Since Poseidon Lua is able to perform static typechecking for its C types, a Poseidon Lua program can avoid the use of these dynamic typechecks and the overhead associated with them with the use of the Modified LuaFFI library.

We run a benchmark suite to test the performance advantage of a Poseidon Lua program over Lua programs. In addition, we run a feature test suite to measure the performance benefit of the manual memory management and direct memory programming features of Poseidon Lua as well as the performance benefit of the Modified LuaFFI library of Poseidon Lua.

In Section 6.2, we give an overview of the test environment within which our benchmarks and feature test cases are run. In Section 6.3, we give a description of the individual benchmarks in our benchmark suite, outline the methodology that we use to run our experiment, and present the results of our experiment along with relevant explanations. Similarly, in Section 6.4, we describe our feature test suite along with results and discussion. In Section 6.5, we conclude the chapter.

6.2 Test Environment

In this section, we provide some information regarding the computer on which we ran our benchmarks and feature test cases. This computer has a GenuineIntel x86_64 CPU architecture and a total memory of 8052948 kB. This computer runs an Ubuntu 14.04.3 LTS operating system.

6.3 Benchmarks

There are four benchmarks in our benchmark suite to compare the performance of Poseidon Lua programs and Lua programs.

6.3.1 Introduction

For our benchmark suite, we borrow 4 benchmark programs that are `plain Lua` programs from `The Computer Language Benchmarks Game` website [6][7]. We create a version of each of these original programs that is written in Poseidon Lua and uses its C types. We also create a version of each of the original programs as a `luaffifb` program that uses the `cdata` values that are provided by the `luaffifb` library. We describe the characteristics of these programs and the problems that they solve as follows.

For our benchmark suite, we only use 4 of the original 10 benchmark programs that were available from the `The Computer Language Benchmarks Game` website [6][7]. The other 6 benchmark programs are unusable for a variety of reasons. Some of these programs execute program fragments that are encoded as Lua strings. These programs are unusable because Poseidon Lua cannot perform static typechecking for program fragments that are encoded as Lua strings. The other unused benchmark programs utilize data structures that are not easily representable using values that belong to the C types of Poseidon Lua.

The 4 programs in our benchmark suite make use of data structures that are easily representable using values that belong to the C types of Poseidon Lua.

Binary-trees (`b1`)

The goal of this benchmark is to create and traverse binary trees of various depths.

N-body (`b2`)

The goal of this benchmark is to track the movements of the different planets in the solar system.

Spectral-norm (`b3`)

The goal of this benchmark is to compute the spectral norm of a matrix.

Fannkuch-redux (`b4`)

The goal of this benchmark is to take an array of numbers and change the location of some of the numbers until a given condition is satisfied.

6.3.2 Methodology

We perform a total of 5 runs of all the programs in our benchmark suite. Each run executes the `plain Lua` programs from the benchmark suite in the following order: `b1`, `b2`, `b3`, `b4`. Next, the same sequence of execution is repeated for the Poseidon Lua programs. Finally, the same sequence of execution is repeated for the `luaffifb` programs.

6.3.3 Results

We process the result of running our benchmark suite using a program written in the R programming language. In Table 6.1, we provide the mean run time for the `plain Lua` program, Poseidon Lua program, and `luaffifb` program of each benchmark as well as the corresponding standard deviations. In Table 6.2, for each benchmark, we provide the

time saved by and the speedup achieved by the Poseidon Lua program with respect to the `plain Lua` program and the `luaffifb` program. Then, we provide the geometric mean of the speedup values for all the benchmarks.

In Table 6.2, for each benchmark, we provide the percentage of the run time of the `plain Lua` program that is saved by the Poseidon Lua program as well as the percentage of the run time of the `luaffifb` program that is saved by the Poseidon Lua program. For each benchmark, we also provide the speedup that is achieved by the Poseidon Lua program relative to the `plain Lua` program and the `luaffifb` program.

The 'Time Saved from `plain Lua` program' column contains the percentage of the run time of the `plain Lua` program that is saved by the Poseidon Lua program as calculated by ((('Mean of `plain Lua` program' - 'Mean of Poseidon Lua program') / 'Mean of `plain Lua` program') * 100) where 'Mean of `plain Lua` program' and 'Mean of Poseidon Lua program' are obtained from Table 6.1.

On the other hand, the 'Time Saved from `luaffifb` program' column contains the percentage of the run time of the `luaffifb` program that is saved by the Poseidon Lua program as calculated by ((('Mean of `luaffifb` program' - 'Mean of Poseidon Lua program') / 'Mean of `luaffifb` program') * 100) where 'Mean of `luaffifb` program' and 'Mean of Poseidon Lua program' are obtained from Table 6.1.

The 'Speedup from `plain Lua` program' column contains the speedup achieved by the Poseidon Lua program with respect to the `plain Lua` program as calculated by ('Mean of `plain Lua` program' / 'Mean of Poseidon Lua program'). The 'Speedup from `luaffifb` program' column contains the speedup achieved by the Poseidon Lua program with respect to the `luaffifb` program as calculated by ('Mean of `luaffifb` program' / 'Mean of Poseidon Lua program'). In the last row, we provide the geometric mean of the speedup data of the columns 'Speedup from `plain Lua` program' and 'Speedup from `luaffifb` program'.

From the '`geometric mean`' row of Table 6.2, we observe that a Poseidon Lua program can be expected to achieve a speedup of 0.98X with respect to a corresponding `plain Lua` program. This means that a Poseidon Lua program and a corresponding `plain Lua` program are roughly equal in terms of performance.

From the '`geometric mean`' row of Table 6.2, we also observe that a Poseidon Lua program can be expected to achieve a speedup of 6.82X with respect to a corresponding `luaffifb` program. This means that a Poseidon Lua program can offer a significant performance advantage over a corresponding `luaffifb` program.

We observe from the 'Speedup from `luaffifb` program' column of Table 6.2 that for all benchmarks, Poseidon Lua programs achieve a significant speedup with respect to

Table 6.1: Benchmark Mean Table

Name	Mean of `plain Lua` program (Seconds)	Standard Deviation of `plain Lua` program (Seconds)	Mean of Poseidon Lua program (Seconds)	Standard Deviation of Poseidon Lua program (Seconds)	Mean of `luaffifb` program (Seconds)	Standard Deviation of `luaffifb` program (Seconds)
b1	22.0	0.70	18.8	0.44	202.4	2.96
b2	4.0	0.70	4.0	0	40.6	1.14
b3	105.6	0.89	108.0	0	270.8	2.58
b4	55.0	0	66.8	2.94	528.8	9.67

Table 6.2: Benchmark Speedup Table

Name	Time Saved from `plain Lua` program (%)	Time Saved from `luaffifb` program (%)	Speedup from `plain Lua` program	Speedup from `luaffifb` program
b1	14.54	90.71	1.17	10.76
b2	0	90.14	1.00	10.15
b3	-2.27	60.11	0.97	2.50
b4	-21.45	87.36	0.82	7.91
geometric mean			0.98	6.82

`luaffifb` programs. These speedups happen because the `luaffifb` programs use `cdata` values to represent relevant data structures in the programs. As a result, `luaffifb` programs have to perform dynamic typechecking which Poseidon Lua programs are able to avoid using static typechecking.

As per speedup values, Poseidon Lua programs outperform `plain Lua` programs in 1 out of 4 benchmarks and have the same performance in 1 out of 4 benchmarks. Poseidon Lua programs outperform `luaffifb` programs in all 4 benchmarks.

6.3.4 Discussion

A Poseidon Lua program can attain performance gains over a corresponding `plain Lua` program in two ways. One way is to use manual memory management to avoid the performance costs of an automatic garbage collector. Another way is to use direct memory programming to avoid the performance costs associated with using a Lua table to manipulate memory.

For 3 out of our 4 benchmarks, i.e., benchmarks `b2`, `b3`, and `b4`, the `plain Lua` program uses Lua tables with consecutive integer indices. In such situations, the Lua VM performs an optimization where it places all the values of the Lua table into an actual array instead of a hash table as is usually the case, which allows a `plain Lua` program to avoid the performance costs of hash-indexing as the array elements can be accessed directly [11]. This effectively allows a `plain Lua` program to perform direct memory programming in a similar manner to a Poseidon Lua program. This negates the ability of a Poseidon Lua program to outperform the use of Lua tables when it comes to memory manipulation.

For each of these 3 benchmarks, the `plain Lua` program uses Lua tables that are not made available for garbage collection until the end of the program. This negates the ability of Poseidon Lua to gain any performance advantage over the automatic garbage collector through the use of manual memory management.

Due to these features of the 3 benchmarks, we can see from Table 6.2 that Poseidon Lua programs can only achieve a speedup of 0.98X with respect to `plain Lua` programs. However, we contend that the features of these 3 benchmarks are anomalous and do not represent the real-world conditions for `plain Lua` programs. It is unlikely that `plain Lua` programs never produce any garbage for the automatic garbage collector to process or that they always use Lua tables as arrays.

The 4 `plain Lua` programs in our benchmark suite were selected because they used data structures that could be easily represented by values that belong to the C types of

Poseidon Lua. On the other hand, the programs in our benchmark suite are all micro-benchmarks. Thus, these are programs that solve a small problem within a small amount of code. As a result, these programs are vulnerable to having anomalous behavior that would not occur in larger real-world programs.

However, from Table 6.2 we can observe that for benchmarks `b2`, `b3`, and `b4`, the Poseidon Lua programs are able to attain significant performance gains over the `luaffifb` programs. Note that in these cases, Poseidon Lua programs are able to avoid dynamic typechecking of `cdata` values at each use-site.

On the other hand, benchmark `b1` (Binary-trees) does not exhibit the anomalous features of benchmarks `b2`, `b3`, and `b4`. For this benchmark, the `plain Lua` program uses Lua tables as hash tables and regularly produces garbage for the automatic garbage collector to process. This allows the Poseidon Lua program to use manual memory management and direct memory programming to gain a performance advantage over the `plain Lua` program. Indeed, we can see from Table 6.2, that for benchmark `b1`, the Poseidon Lua program is able to attain a speedup of 1.17X with respect to the `plain Lua` program and a speedup of 10.76X with respect to the `luaffifb` program. We argue that this is a more accurate reflection of the performance advantage that can be achieved by a Poseidon Lua program with respect to a `plain Lua` program under real-world conditions.

Since 3 out of 4 benchmarks show anomalous behavior for the `plain Lua` programs, we decided to carry out further testing to ascertain that the features of Poseidon Lua have indeed performance advantages over Lua. Our new tests represent the more realistic scenarios for which we have designed the features of Poseidon Lua.

6.4 Feature Testing

There are three test cases in our feature test suite to compare the performance of Poseidon Lua and Lua programs.

6.4.1 Introduction

Using our feature test suite, we test the performance of the various language features of Poseidon Lua. We test how the manual memory management feature of Poseidon Lua performs against the automatic garbage collector of Lua. We test the performance impact of using the direct memory programming feature of Poseidon Lua relative to the use of

Lua tables for memory manipulation. We also test the performance benefit of using the Modified LuaFFI, `luaffi_cs`, library of Poseidon Lua in place of the `luaffifb` library.

Manual memory management (`c1`)

The goal of this test case is to measure the performance benefits of using the manual memory management feature of Poseidon Lua relative to using the automatic garbage collection feature of Lua. For this purpose, each program in this test case executes 200 iterations. At the start of each iteration, 30 lists are created where each list has 1,000,000 integer elements. For each list, the elements that have even indices are populated with values. Then, the values of all the populated elements of all the lists are added together to obtain a sum. At the end of each iteration, the 30 lists are deleted. The `plain Lua` program and the Poseidon Lua program use a Lua table and a C pointer to a block of memory, respectively, to represent each list.

Note that the lists are created and deleted on every iteration. This emphasizes the difference in performance of the automatic garbage collection of Lua tables in the `plain Lua` program and the manual deallocation of the memory blocks in the Poseidon Lua program. In addition, only the elements of the lists that have even indices are assigned values so that Lua tables are implemented using hash tables and an optimization is not performed to implement Lua tables using arrays.

Direct memory programing (`c2`)

The goal of this test case is to measure the performance benefits of using the direct memory programming feature of Poseidon Lua relative to using the Lua table values of Lua. For this purpose, each program in this test case creates 30 lists where each list has 1,000,000 integer elements, executes 200 iterations to traverse over the elements of the 30 lists, and deletes the 30 lists at the end of the program. In each iteration, for each list, the elements of the list that have even indices are populated with values. Then, the values of all the populated elements of all the lists are added together to obtain a sum. The `plain Lua` program and the Poseidon Lua program use a Lua table and a C pointer to a block of memory, respectively, to represent each list.

Note that the behavior of `c2` is different than the behavior of `c1`. In `c2`, the lists are created and deleted only once and this is done outside the iterations. The iterations only manipulate the elements of the lists. This emphasizes the difference in performance of the data manipulation mechanism of Lua tables in the `plain Lua` program and the direct

memory programming facility of the Poseidon Lua program. In addition, only the elements of the lists that have even indices are assigned values so that Lua tables are implemented using hash tables and an optimization is not performed to implement Lua tables using arrays.

Foreign function interface (`c3`)

The goal of this test case is to measure the performance benefits of using the Modified LuaFFI library of Poseidon Lua relative to using the `luaffifb` library of Lua. For this purpose, each program in this test case executes 2000 iterations. At the start of each iteration, 30 lists are created where each list has 10,000 integer elements. All the elements of these lists are populated with values. Then, for each list, an external C function is called to set the value of one of its elements. Then, the values of all the populated elements of all the lists are added together to obtain a sum. At the end of each iteration, the 30 lists are deleted. The `luaffifb` program and the Modified LuaFFI program use a `cdata` and a C pointer to a block of memory, respectively, to represent each list.

The main purpose of `c3` is not to measure the difference in the performance of the two FFI libraries when they are used to call an external C function. The main difference between the two FFI libraries is that when an external C function call returns a pointer, the `luaffifb` library returns a `cdata` value containing the pointer, whereas the Modified LuaFFI library returns a C pointer value that can be used directly in a Poseidon Lua program. The goal is to measure the difference in the performance of using the `cdata` value that is returned by the `luaffifb` library and the C pointer value that is returned by the Modified LuaFFI library.

The lists are the data structures that are represented by the `cdata` values and the C pointer values. The lists are used as the inputs and outputs of the external C function calls. Manipulating the data stored in the lists after the call to the external C function causes dynamic typechecking to be performed by the `cdata` values.

The C pointer values should be able to avoid the overhead from the dynamic typechecking that is performed by the `cdata` values. The C pointer values should also be able to avoid the overhead from the automatic garbage collection of the `cdata` values through the use of manual memory management.

Table 6.3: Feature Testing Mean Table

Name	Mean of `plain Lua` program (Seconds)	Standard Deviation of `plain Lua` program (Seconds)	Mean of Poseidon Lua program (Seconds)	Standard Deviation of Poseidon Lua program (Seconds)
c1	378.4	4.61	90.4	0.54
c2	118.8	0.44	90.2	0.44
Name	Mean of `luaffifb` program (Seconds)	Standard Deviation of `luaffifb` program (Seconds)	Mean of Modified LuaFFI program (Seconds)	Standard Deviation of Modified LuaFFI program (Seconds)
c3	179.6	1.67	17.4	0.54

6.4.2 Methodology

We perform a total of 5 runs of all the programs in our feature test suite. Each run begins by executing the `plain Lua` program of test case `c1`. Then, the `plain Lua` program of test case `c2` is executed. Next, the `luaffifb` program of test case `c3` is executed.

This is followed by the execution of the Poseidon Lua program of test case `c1`. Then, the Poseidon Lua program of test case `c2` is executed. Next, the Modified LuaFFI program of test case `c3` is executed.

6.4.3 Results

We process the result of running our feature test suite using a program written in the R programming language. In Table 6.3, we provide the mean run time and the standard deviation for each program of each test case. In Table 6.4, for each test case, we provide the speedup achieved by the Poseidon Lua or Modified LuaFFI program with respect to the `plain Lua` or `luaffifb` program, respectively.

For the test case `c1`, the Poseidon Lua program is able to achieve a speedup of 4.18X with respect to the corresponding `plain Lua` program. For the test case `c2`, the Poseidon

Table 6.4: Feature Testing Speedup Table

Name	Time Saved from `plain Lua` program (%)	Speedup from `plain Lua` program
c1	76.10	4.18
c2	24.07	1.31

Name	Time Saved from `luaffifb` program (%)	Speedup from `luaffifb` program
c3	90.31	10.32

Lua program is able to achieve a speedup of 1.31X with respect to the corresponding `plain Lua` program.

For the test case `c3`, the Modified LuaFFI program is able to achieve a speedup of 10.32X with respect to the corresponding `luaffifb` program.

From the speedup values of Table 6.4, we observe that Poseidon Lua programs outperform `plain Lua` programs in the first 2 test cases (`c1` and `c2`). The Modified LuaFFI program outperforms the `luaffifb` program in the last test case (`c3`). Therefore, the Poseidon Lua or Modified LuaFFI programs outperform the `plain Lua` or `luaffifb` programs in all 3 test cases.

6.4.4 Discussion

Test case `c1` involves heavy dynamic memory allocation and deallocation activities. The `plain Lua` program uses automatic garbage collection to deal with the deallocation of memory. The Poseidon Lua program uses manual memory management to handle the deallocation of memory. For this reason, we observe from Table 6.4 that the Poseidon Lua program is able to achieve a notable speedup of 4.18X over the `plain Lua` program.

Test case `c2` involves mainly memory manipulation activities. The `plain Lua` program uses Lua tables to deal with memory manipulation. The Poseidon Lua program uses direct memory programming to handle memory manipulation. From the speedup value for test case `c2` in Table 6.4, we observe that the Poseidon Lua program is able to achieve a speedup of 1.31X over the `plain Lua` program.

In test case `c3`, both programs call an external C function which returns a C pointer. As a result, the `luaffifb` program uses a `cdata` value to represent the C pointer and carries out dynamic typechecking at each use-site of the return value. On the other hand, the Modified LuaFFI program uses static typechecking to avoid any extra dynamic typechecking. Moreover, test case `c3` also involves heavy dynamic memory allocation and deallocation activities. Therefore, the Modified LuaFFI program is also able to use manual memory management to avoid the performance overhead of the automatic garbage collection of `cdata` values. For these reasons, we observe from Table 6.4 that the Modified LuaFFI program is able to achieve a significant speedup of 10.32X over the `luaffifb` program.

Note that in the test case `c3`, the goal is to measure the total performance benefit of using the C pointer that is returned by the Modified LuaFFI library over using the `cdata` value that is returned by the `luaffifb` library. Thus, we are not just measuring the performance of the FFI libraries when it comes to calling the external C function. We are mainly measuring the performance gains from the avoidance of extra dynamic typechecking and automatic garbage collection.

6.5 Conclusions

In this chapter, we measure the performance advantage that can be attained by a Poseidon Lua or Modified LuaFFI program with respect to a `plain Lua` or a `luaffifb` program. For this purpose, at first, we run a benchmark suit and find that a Poseidon Lua program can be expected to achieve a speedup of 0.98X with respect to a corresponding `plain Lua` program and a speedup of 6.82X with respect to a corresponding `luaffifb` program. Thus, a Poseidon Lua program can offer a substantial improvement in performance with respect to a corresponding `luaffifb` program. However, a Poseidon Lua program is evenly matched with a corresponding `plain Lua` program with regard to performance.

To ascertain that the Poseidon Lua programs do indeed have a performance advantage over Lua programs in real life programming situations, we test the features of Poseidon Lua individually in our feature test suit which consists of 3 test cases.

A Poseidon Lua program is able to achieve a speedup of 4.18X with respect to a corresponding `plain Lua` program when heavy dynamic memory allocation and deallocation is involved. A Poseidon Lua program is able to achieve a speedup of 1.31X with respect to a corresponding `plain Lua` program when memory manipulation is involved.

A Modified LuaFFI program is able to achieve a speedup of 10.32X with respect to a corresponding `luaffifb` program when the C pointer value returned by the Modified

LuaFFI library is used instead of the `cdata` value that is returned by the `luaffifb` library.

Chapter 7

Conclusions and Future Work

7.1 Contributions

The novel features that are introduced in this thesis are as follows.

In Chapter 3, we introduce our proposed programming language named Poseidon Lua. Poseidon Lua extends Typed Lua with operators to perform manual memory management and C types, which include C pointer types that can be used for direct memory programming.

In Chapter 4, we introduce Modified Lua. Modified Lua extends Lua with special operators that can be used to manipulate C values. These special operators can also be used to perform manual memory management. A Poseidon Lua program is statically typechecked and translated to a Modified Lua program for the purpose of making use of its special operators. Modified Lua is interesting independently of Poseidon Lua because it effectively allows a program to take advantage of facilities such as pointer manipulation and manual memory management from within a dynamically-typed scripting language. Traditionally, these language facilities are only available from statically-typed programming languages such as the C programming language. These language facilities are generally not offered by dynamically-typed scripting languages such as Lua because these languages tend to strive for a greater degree of automation.

In Chapter 5, we introduce the Modified LuaFFI library. This library allows a Poseidon Lua program to call an external C function that returns a C pointer. Since Poseidon Lua is able to perform static typechecking, there is no need to perform any dynamic typechecking to enforce the static type of the C pointer at run-time.

7.2 Future Work

Our implementation of Poseidon Lua does not make use of a JIT compiler. A JIT compiler would be able to use run-time information to improve the performance of a Poseidon Lua program. There are situations where a Poseidon Lua program would need to perform run-time computations. For example, a Poseidon Lua program may need to compute the offset required to index a C array at run-time. A JIT compiler may be able to improve the performance of a Poseidon Lua program in situations such as this. This is a possible direction of future research work.

The C programming language is not a memory-safe programming language. It allows a program to access any location of memory regardless of whether the program should or should not access that location in memory. This poses a security concern for software that is developed in the C programming language. Poseidon Lua also allows a program to access any location in memory. A possible direction for future work is to develop language features that can mitigate the security risk from unrestricted memory accesses.

When the return value of an external C function is a non-primitive value other than a pointer value, such as a struct value, Poseidon Lua cannot efficiently handle this case using the Modified LuaFFI library. In this situation, the Modified LuaFFI library returns a `cdata` value that cannot be handled by Poseidon Lua without incurring the performance costs of extra dynamic typechecks. A direction for future work is to handle the cases where the return value is a non-primitive value other than a pointer with static typechecking only.

Another possible direction for future work is to investigate how to perform static typechecking for arguments to an external C function call instead of dynamic typechecking.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, April 1991.

[2] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure. In Thiemann [26], pages 68–94.

[3] Gilad Bracha. Pluggable type systems. In *OOPSLA workshop on revival of dynamic languages*, volume 4, 2004.

[4] Giuseppe Castagna, editor. *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*. Springer, 2009.

[5] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, September 2002.

[6] The Computer Language Benchmarks Game. Current website. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[7] The Computer Language Benchmarks Game. Original website. http://benchmarksgame.alioth.debian.org/.

[8] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

[9] Marc Herrlich, Rainer Malaka, and Maic Masuch, editors. *Entertainment Computing - ICEC 2012 - 11th International Conference, ICEC 2012, Bremen, Germany, September 26-29, 2012. Proceedings*, volume 7522 of *Lecture Notes in Computer Science*. Springer, 2012.

[10] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 2–1–2–26, New York, NY, USA, 2007. ACM.

[11] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. The implementation of Lua 5.0. *J. UCS*, 11(7):1159–1176, 2005.

[12] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[13] Paul Klint, Loren Roosendaal, and Riemer van Rozen. Game developers need Lua AiR - static analysis of Lua using interface models. In Herrlich et al. [9], pages 530–535.

[14] Jukka Lehtosalo and David J Greaves. Language with a pluggable type system and optional runtime monitoring of type errors. In *Proceedings of International Workshop on Scripts to Programs (STOP)*, 2011.

[15] Poseidon Lua. https://github.com/rt1000/Poseidon_Lua.

[16] luaffifb. Current website. https://github.com/facebookarchive/luaffifb.

[17] luaffifb. Original website. https://github.com/facebook/luaffifb.

[18] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed Lua: An optional type system for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 3:1–3:10, New York, NY, USA, 2014. ACM.

[19] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. A formalization of Typed Lua. *SIGPLAN Not.*, 51(2):13–25, October 2015.

[20] Gianluca Mezzetti, Anders Møller, and Fabio Strocco. Type unsoundness in practice: An empirical study of Dart. *SIGPLAN Not.*, 52(2):13–24, November 2016.

[21] Hisham Muhammad and Roberto Ierusalimschy. C APIs in extension and extensible languages. *J. UCS*, 13(6):839–853, 2007.

[22] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[23] Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2007.

[24] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[25] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? *SIGPLAN Not.*, 51(1):456–468, January 2016.

[26] Peter Thiemann, editor. *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*. Springer, 2016.

[27] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[28] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.

[29] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. *SIGPLAN Not.*, 50(2):45–56, October 2014.

[30] Michael M. Vitousek and Jeremy G. Siek. From optional to gradual typing via transient checks. In *5th Script To Program Evolution Workshop*, 2016.

[31] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Castagna [4], pages 1–16.