Pontifícia Universidade Católica
do Rio de Janeiro

Gabriel de Quadros Ligneul

The Implementation of Records in Pallene

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Roberto Ierusalimschy

Rio de Janeiro
May 2019

Pontifícia Universidade Católica
do Rio de Janeiro

Gabriel de Quadros Ligneul

The Implementation of Records in Pallene

Dissertation presented to the Programa de Pós–graduação em Informática da PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the undersigned Examination Committee.

Prof. Roberto Ierusalimschy
Advisor
Departamento de Informática – PUC-Rio

Prof. Luiz Fernando Bessa Seibel
Departamento de Informática – PUC-Rio

Prof. Luiz Henrique de Figueiredo
IMPA

Rio de Janeiro, May 10th, 2019

Gabriel de Quadros Ligneul

Bachelor in Computer Science at the Pontifícia Universidade Católica do Rio de Janeiro (2016).

Bibliographic data

de Quadros Ligneul, Gabriel

The Implementation of Records in Pallene / Gabriel de Quadros Ligneul; advisor: Roberto Ierusalimschy. – 2019.

v., 74 f: il. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Lua;. 3. Tipagem Estática;. 4. Scripting;. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I would like to express my gratitude to my advisor Roberto Ierusalimschy for his patience, wisdom, and guidance; to Hugo Gualandi for the partnership during this research project; to André Maidl, Fábio Mascarenhas, and Hisham Muhammad for the initial ideas for this project; to my family, friends, and Karol Silva for the support during the project; and to my colleges from LabLua, Tecgraf, and CUJO AI.

I would also like to thank PUC-Rio and the Departamento de Informática for making this research project viable.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) Finance Code 001.

Abstract

de Quadros Ligneul, Gabriel; Ierusalimschy, Roberto (Advisor). **The Implementation of Records in Pallene**. Rio de Janeiro, 2019. 74p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The dynamic features of scripting languages introduce significant overhead in execution time when compared to system languages. The scripting architecture can be used to improve the poor performance of scripting languages. The programmer should use a system language for resource-intensive tasks, and a scripting one for flexibility. However, this architecture has two significant flaws when used to improve the performance of scripting languages. First, there is a conceptual gap between both languages; so migrating from the scripting language to the system language may require enormous effort. Second, there is a hidden overhead when manipulating the scripting-language data structures from the system language. Pallene is a system language designed particularly for Lua that aims to solve these two issues. Pallene is a statically-typed subset of Lua, which facilitates the migration process. Moreover, Pallene manipulates Lua's data structures directly without introducing overhead. In this work, we propose two types of records for Pallene, and we present the implementation of the Pallene compiler. We benchmarked our compiler to compare it to standard Lua, LuaJIT, and C programs using the Lua-C API. Our experiments show that Pallene is competitive with the existing solutions to improve Lua's performance.

Keywords

Lua; Static Typing; Scripting;

Resumo

de Quadros Ligneul, Gabriel; Ierusalimschy, Roberto. **A Implementação de Registros em Pallene**. Rio de Janeiro, 2019. 74p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

As características dinâmicas de linguagens de scripting introduzem um gargalo significativo no tempo de execução quando comparadas a linguagens de sistemas. A arquitetura scripting pode ser usada para melhorar o desempenho ruim de linguagens de scripting. O programador deve usar a linguagem de sistemas para tarefas que consomem muitos recursos, e a de scripting para flexibilidade. Entretanto, essa arquitetura tem duas falhas significativas quando usada para melhorar o desempenho de linguagens de scripting. Primeiro, existe uma lacuna conceitual entre as duas linguagens, logo migrar da linguagem de scripting para linguagem de sistemas pode exigir enorme esforço. Segundo, existe um gargalo escondido ao manipular as estruturas de dados da linguagem de scripting a partir da linguagem de sistemas. Pallene é uma linguagem de sistemas projetada particularmente para Lua que almeja resolver essas duas falhas. Pallene é um subconjunto estaticamente tipado de Lua, o que facilita o processo de migração. Além disso, Pallene manipula diretamente as estruturas de dados de Lua sem introduzir gargalo. Neste trabalho, nós propomos dois tipos de registros para Pallene, e nós apresentamos a implementação do compilador de Pallene. Nós avaliamos o desempenho do nosso compilador para compará-la com Lua padrão, LuaJIT, e programas C que utilizam a API C de Lua. Nossos experimentos mostram que Pallene é competitiva com as soluções existentes para melhorar o desempenho de Lua.

Palavras-chave

Lua; Tipagem Estática; Scripting;

Table of contents

List of figures

List of tables

1
Introduction

Scripting languages have several high-level features that make them easier to use when compared to system languages (OUSTERHOUT, 1998). Dynamic typing is one of the key features in scripting languages. Programmers can write more generic and concise code since there is not a static type system restricting the language. It is common for scripting languages to be interpreted instead of compiled, making them highly portable. Interpreters also enable a shorter development and test cycle. Scripting languages usually are also garbage collected and memory safe. Ultimately, all these features allow programmers to write code faster.

1.1
Problem

Although scripting languages have several benefits, these languages come with a high runtime performance cost. Scripting languages can be up to two orders of magnitude slower than statically-typed languages (MARR et al., 2016). However, programmers that seek the benefits of scripting languages sometimes neglect the performance overhead. These programmers start writing their programs in scripting languages without worrying about the runtime performance. Eventually, they realize that their programs perform poorly because of that design decision. This realization may happen only when the program code base becomes large. At this point, it is no longer viable to rewrite the whole program in another more efficient language. Then, these programmers are inclined to search for ways to improve the performance of programs written in a slow scripting language.

It is not a trivial task to improve the performance of scripting languages. Their dynamic nature hinders the usage of standard optimization techniques. Without a static type system, it is very difficult for an ahead-of-time (AOT) compiler to optimize a scripting language. During our research, we implemented an AOT compiler for the Lua scripting language. Our Lua AOT compiler runs twice as fast as in the standard Lua interpreter. However, the performance of our Lua AOT compiler still is in the same order of magnitude of the Lua interpreter. The code generated by compilers of static languages is

considerably faster than the code that our Lua compiler can generate.

One way to improve the performance of scripting languages is migrating part of the program to a system language. The scripting architecture combines two languages in the same program (OUSTERHOUT, 1998). The programmer uses a scripting language for expressiveness and a system language for performance-sensitive parts. Initially, this architecture was conceived to increase flexibility by introducing a scripting language to programs written in system languages. Programmers that seek better performance in scripting languages can use the same approach inversely, by migrating part of their programs from a scripting language to a system one.

Programmers face two issues when attempting to migrate from a scripting language to a system one. These issues are related, but they affect the programmer at distinct levels. The first issue is the large amount of effort necessary to migrate to a different language, as there is a conceptual gap between both languages. Usually, system languages do not contemplate high-level features such as dynamic data structures, closures, and polymorphism. Programmers might face difficulties when migrating code that uses these features to a low-level system language.

A well-designed scripting language provides an API for the system language to manipulate the dynamic data structures of scripting languages. This API provides abstractions that hide the implementation details of these data structures. However, the API leads to the second issue of migrating to a system language, which is a runtime performance overhead caused by the API itself. In some cases, the scripting-language API can negatively impact the performance of the system language, canceling the benefits of the migration process. In Chapter 6, we show examples where the Lua interpreter is faster than C code using the Lua-C API when manipulating Lua tables.

The process of migrating from a scripting language to a system one becomes risky when considering both issues. The migration process requires a considerable amount of effort due to the conceptual gap between the languages. In some cases, all the effort can be fruitless due to the runtime overhead caused by the scripting-language API.

Besides scripting, there are at least other two ways of improving scripting languages: optional typing (TOBIN-HOCHSTADT; FELLEISEN, 2006) and just-in-time compilers (AYCOCK, 2003). Optional type systems introduce static type annotations to the scripting language. An AOT compiler can use these type annotations to generate efficient machine code. Just-in-time (JIT) compilers automatically optimize scripting languages during program execution. A JIT compiler generates specialized machine-code using informa-

tion that only is available at runtime. Like scripting, there are issues when using these approaches to improve the performance of scripting languages. We discuss these issues in depth in Chapter 2.

1.2 The Companion Language Approach

Yet another way to improve the performance of scripting languages is using a companion language (GUALANDI; IERUSALIMSCHY, 2018). A companion language is a system language designed to interoperate with a particular scripting language. The companion language should have a static type system so that an AOT compiler can generate efficient code. Like scripting, programmers should use both the scripting and the companion languages in the same architecture. To improve the performance of the scripting language, programmers should migrate parts of their program to the companion language.

Unlike usual system languages, the companion language should encourage a smooth migration from the scripting language. To reduce the conceptual gap between the companion and the scripting languages, companion languages take inspiration from gradually-typed languages (SIEK; TAHA, 2006). Gradually-typed languages have optional type systems that encourage the gradual migration from dynamic to static code. In the place of an optional type system, the companion language is another language based on the scripting language but with a static type system.

Gradually-typed languages have a property called *gradual guarantee* which states that removing types from a correct program does not change its behavior (SIEK et al., 2015). Companion languages should apply the gradual guarantee property to scripting languages. If the programmer removes the type annotations from a correct program written in a companion language, the result should be a program in the scripting language that behaves exactly the same as the original one.

The gradual guarantee also provides a derivative property in the opposite direction, going from static code to dynamic code. Adding types annotations to dynamic code has one of two outcomes: either the program behaves the same, or it triggers a type error. When we apply this property to companion languages, migrating from the scripting language it is just a matter of adding the correct type annotations to the program. However, there are several programs that the companion-language type system cannot represent. The companion-language type system should restrict the dynamism of the scripting language to be amenable to the optimizations of an AOT compiler.

Besides the gradual guarantee, the companion language should manipu-

late the data structures of the scripting language directly. The manipulation should be transparent to the programmer; an API to manipulate these data structures should not be necessary. Instead, the companion-language type system should have types that are built over the dynamic data structures of the scripting languages. For instance, a companion language for Lua should have an array type that is built over Lua tables. Retrieving an element from an array should be equivalent to retrieving an entry with an integer key from a Lua table.

Bypassing the API of the scripting language allows the companion-language compiler to generate more efficient code. The companion-language compiler should access the internals of the scripting-language interpreter. A programmer would face difficulties if he tried to manually write code that bypasses the scripting-language API. However, the companion-language compiler should be carefully written so it does not violate the invariants of the scripting-language interpreter. Bypassing the API solves the runtime overhead present when other system languages manipulate the scripting-language data structures.

To materialize the idea of companion languages, we developed Pallene[1] (GUALANDI; IERUSALIMSCHY, 2018): a companion language for Lua. Pallene is a subset of Lua with static type annotations. The Pallene type system characterizes the main differences between both languages. This type system is simple; it contains only booleans, numbers, strings, arrays, and records. Values of arrays and records types are built over Lua tables. Pallene does not support high-level Lua features such as polymorphism to guarantee that its compiler can generate efficient code.

1.3 Objectives

This dissertation presents the implementation of our compiler for Pallene, which is an ahead-of-time compiler that generates portable C code. By generating C code, our compiler leverages the portability and low-level optimizations from a mature C compiler. Additionally, the code generated by our compiler manipulates the data structures of the Lua interpreter directly, bypassing the Lua-C API. We hypothesize that Pallene can archive performance similar to the existing methods that aim to improve Lua's performance. To confirm this hypothesis, we measure the performance of Pallene and compare it to the Lua interpreter, LuaJIT, and C programs that use the Lua-C API.

[1]Pallene is a fork of the Titan programming language (TITAN, 2018) and a collaborative project of the LabLua research laboratory.

In this dissertation, we also propose two types of records for Pallene: L-records and C-records. Records are data structures that map a set of fields to values of given types (like C structs). We use Lua tables to represent L-records, even though tables are dynamic data structures and Pallene records have static types. We use C-structs to represent C-records; to expose C-records to Lua, we use the Lua userdata type. We implemented these two types of records in our compiler and evaluated their performance.

In Chapter 2, we discuss optional typing and just-in-time compilers. In Chapter 3, we describe the Pallene language, explaining the reasoning behind the design decisions we made. In Chapter 4, we overview the internals of Lua interpreter to clarify how Pallene manipulates them. In Chapter 5, we describe the implementation of the Pallene compiler, especially the implementation of Pallene's data structures. In Chapter 6, we evaluate the implementation of the Pallene compiler by comparing it to other ways of improving the performance of Lua. Finally, in Chapter 7, we conclude our discussion and suggest avenues for future research.

2
Optimizing Scripting Languages

In Chapter 1, we described how programmers can use the scripting architecture to improve the performance of scripting languages. Another way to improve the performance of scripting languages is with optional type systems. These type systems introduce static-type annotations into a dynamically typed language. A compiler may use the static type annotations to generate optimized code. Besides optional typing, programmes can also use just-in-time (JIT) compilers to improve the performance of scripting languages. These compilers use pieces of information only available at runtime to optimize the dynamic code. In this chapter, we discuss optional type systems and JIT compilers.

2.1
Optional Typing

Optional type systems introduce static typing annotations into dynamic languages. Compilers and tools may use type annotations to check for type errors and generate optimized code. The type annotations also serve as a light machine-checked documentation. Optional typing is a broad area; moreover, each optional type system has its specific objectives. For instance, TypedLua (an optional type system for Lua) aims to catch type errors before execution; it does not try to improve Lua's performance (MAIDL, 2015).

Common Lisp (GRAHAM, 1995) is a dynamic language that allows programmers to add type annotations in their code. Type annotations go along with the *optimize* declaration, which tells the compiler to optimize the given function. Furthermore, programmers can hand-tune the safety and speed of their code. Unsafe code is implementation-defined and may skip runtime type checks, leading to untrapped errors. Figure 2.1 illustrates the usage of the optimize declaration.

Gradual typing is a version of optional typing that aims to provide a smooth transition from dynamic to static typing (SIEK; TAHA, 2006). Tobin-Hochstadt and Felleisen encourage programmers to migrate from dynamic to static when scripts evolve into full-blown programs (TOBIN-HOCHSTADT; FELLEISEN, 2006). They argue that sound

```
(defun mul (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (float x y))
  (the float (* x y)))
```

Figure 2.1: Function declaration in Common Lisp with static type annotations and optimize declaration.

statically-typed programs are more robust since they cannot trigger runtime type errors. When compared to the scripting approach, gradual typing facilitates language migration by reducing the boundaries between static and dynamic. Since gradual typing includes dynamic and static typing in a single language, both worlds can share the same runtime values. So, instead of writing a compatibility layer to combine different languages, the programmer only needs to rewrite part of his code with static types.

The gradual guarantee is a formal property of sound gradual typing systems (SIEK et al., 2015). As presented in the previous chapter, the gradual guarantee states that when removing types from a well-typed program, the untyped version should behave the same. This property requires the gradually typed language to be the union of both a dynamically typed and a statically typed version of the language. If the type system is sound, the gradual guarantee ensures runtime type errors can only occur in the dynamic side.

Even though gradual type systems benefit from a well-established theoretical framework, it might be challenging to implement a sound type system in practice. Language creators have to balance soundness, usability, and performance. For instance, TypeScript is a gradually typed language based on JavaScript. Typescript creators purposely designed its type system to be unsound (BIERMAN et al., 2014). This design decision allows programmers to type common dynamic idioms while keeping the type system simple. A sound design would either require a sophisticated type system or restrict the language too much. Nevertheless, TypeScript still helps its users to find bugs, and it also empowers auto-completion tools.

The cost of runtime type checks is another potential issue with gradual type systems. Typed Racket is a mature implementation of sound gradual typing that suffers from the cost of runtime type checks (TAKIKAWA et al., 2016). When mixing dynamically and statically typed code, the execution time can be over 100 times slower than plainly dynamic code. To solve this performance issue, the programmer would need to rewrite all modules with type annotations. On the other hand, TypeScript only checks types statically to avoid this cost; its compiler generates plain

JavaScript without any type verifications.

2.2 Just-in-time Compilers

Just-in-time (JIT) compilation (AYCOCK, 2003) is another way to improve the performance of dynamic languages. Usually, JIT compilers start executing the program in an interpreter and profile its performance. When the JIT compiler identifies a piece of code that the interpreter frequently executes (hotspot), it generates equivalent machine code during runtime. The JIT compiler takes advantage of runtime information that the interpreter obtained previously. With information like the types and executed code paths, the JIT compiler can generate optimized machine-code that is highly specialized. Then, the JIT compiler executes this code in the place of the interpreter to improve the program performance.

Scripting languages rarely provide enough information for ahead-of-time (AOT) compilers to generate efficient code. Since these languages are dynamic, variables do not have type annotations, and data structures are not statically defined. Moreover, instructions in scripting languages tend to be polymorphic. They have a different implementation depending on the type of input values. An AOT compiler would have to handle all possible cases, not being able to optimize the code entirely. With these limitations, it should not be worth to change from an interpreter to an AOT compiler.

Figure 2.2 illustrates the polymorphic behavior in the virtual machine of Lua interpreter. When adding up two values in Lua, the interpreter first needs to verify the type of each of them. Depending on the types, the code might have to perform either an integer or a floating-point addition. It is even possible that there is a metamethod call or a runtime type error. Without type annotations, an AOT compiler would have to generate code similar to this one.

JIT compilers tackle the lack of type information through speculation. They generate code specialized for cases observed during profiling. Type specialization is a typical optimization in JIT compilers for scripting languages. In this optimization, the compiler detects variable types during execution to generate very specialized code. In the Lua *add* instruction described previously, a JIT compiler would first try to detect what are the types of input values that occur during runtime. Then, the JIT compiler might replace the code in figure 2.2 with just a few lines of specialized machine code for the observed types.

The JIT compiler adds guard instructions that check whether the incoming types are the ones expected. Since the language is still dynamic, the JIT

```
vmcase(OP_ADD) {
  TValue *rb = vRB(i);
  TValue *rc = vRC(i);
  lua_Number nb; lua_Number nc;
  if (ttisinteger(rb) && ttisinteger(rc)) {
    lua_Integer ib = ivalue(rb); lua_Integer ic = ivalue(rc);
    setivalue(vra, intop(+, ib, ic));
  }
  else if (tonumberns(rb, nb) && tonumberns(rc, nc)) {
    setfltvalue(vra, luai_numadd(L, nb, nc));
  }
  else
    Protect(luaT_trybinTM(L, rb, rc, ra, TM_ADD));
  vmbreak;
}
```

Figure 2.2: Add instruction in Lua virtual machine (defined in line 1089 of `lvm.h` in lua5.4.0-work1).

speculation might fail. If the code receives a type different from what it expects, the compiled code hands the execution control back to the interpreter. Eventually, the JIT compiler can adapt and recompile the corresponding code.

Method-based JIT compilers generate code for full functions. Trace-based JIT compilers is another technique that generates code for execution traces instead (GAL et al., 2006). Traces can start at any place of the function and may even go through other function calls. Tracing JIT compilers profile backward jumps in the code (usually loops) to detect hotspots. After detecting a hotspot, the interpreter records each instruction executed during an iteration of the loop. Like other JIT compilers, a tracing JIT gathers runtime information, like types, to generate specialized code. Since the JIT compiler records instructions for a single loop iteration, the trace covers only the executed path and the compiler does not compile branches that it did not traverse. If the compiled code reaches one of these branches during the generated trace, it hands back the control to the interpreter.

Both method-based and trace-based JIT compilers have proven to improve scripting languages performance considerably. Being the state-of-art of dynamic language optimization, JIT compilers can achieve the performance of system languages in some cases (MARR et al., 2016). However, archiving this level of performance requires considerable engineering effort since JIT compilers are hard to maintain. There are several factors responsible for the difficulty of writing a JIT compiler. The JIT compiler needs to gather runtime information to generate speculative machine code. The compilation has CPU and memory constraints, differently from an AOT compiler.

The requirement to execute code in two different stages (interpretation

and compilation) is one of the reasons that JIT compilers are hard to maintain. Some JIT compilers have even three stages: an interpreter, a baseline compiler, and an optimizing compiler (MCILROY, 2016). All these stages add up complexity to the JIT source code compared to an interpreter. LuaJIT 2.0 (PALL, 2005a), a JIT compiler for Lua, has 96k lines of code, which include 24k lines of non-portable hand-written assembly. On the other hand, the reference implementation of Lua 5.3 has only 24k lines of portable C code. The V8 JavaScript engine has 2 million lines of code, and a hello world program in it consumes 17 MB of memory[1].

The constraints in CPU and memory also impede JIT compiler to use compilation frameworks. For instance, LLVM (LLVM, 2019a) is a compiler framework designed for AOT compilers. WebKit's FTL JIT compiler used LLVM as a backend, but eventually, its creators replaced LLVM with a new backend called B3 (PIZLO, 2016). Even though LLVM is highly customizable, FTL creators concluded that they would need to replace it to reduce compile time and memory utilization.

There exist frameworks especially designed for implementing JIT compilers, such as PyPy (RIGO; PEDRONI, 2006) and Truffle (WÜRTHINGER et at., 2013). The programmer needs only to implement an interpreter using the provided framework, and the framework automatically optimizes the interpreter. PyPy is a *meta-tracing* JIT compiler that executes and optimizes the interpreter itself. Truffle provides tools to implement an AST interpreter; during execution, the interpreter leverages the partial-evaluation optimization technique of Truffle. While these approaches are promising, they are not widely used yet.

Like AOT compilers, JIT compilers capitalize on heuristics to optimize the source code. JIT compilers have to cover all language features, but not all of them can be optimized appropriately. Some dynamic features hinder the optimization in all code around it. The JIT compiler might not even trigger the JIT compiler, and it only interprets the code. When an optimization fails in an AOT compiler, the performance is only slightly affected. On the other hand, if the JIT compiler does not trigger the compiler, the performance gap that can reach two orders of magnitude. These features are called optimization killers because they significantly impact the performance of the program. Features that hinder optimizations can be primitive, like iterating over a collection of values (PALL, 2014, GIRALDEZ, 2017b, ANTONOV, 2013).

Due to the difficulties of implementing a JIT compiler, it is a hard task to change the JIT implementation to remove the optimization killers.

[1]Data obtained by the author using version 7.2 of V8.

The performance gain from a JIT compiler becomes unpredictable since the gap between optimized and not-optimized code is enormous. Hence, to avoid the slow-down, programmers are obliged to modify their code based on implementation details of the JIT compiler. It is not always clear where these modifications should be since they depend on knowing the internals of the compiler. There are even tools to disassemble the code generated by the compiler to optimize it (GIRALDEZ, 2017b). A language like JavaScript may have several implementations of JIT compilers, each them having different optimization problems.

3
The Pallene Language

Pallene is a companion language for Lua, which means that it should have a low conceptual gap to Lua and it should also manipulate Lua's data structures. In this chapter, we present how Pallene was designed to fulfill these requirements.

3.1
The Syntax of Pallene

Pallene's syntax is a subset of Lua's syntax augmented with static type annotations. An ahead-of-time compiler for Pallene uses these type annotations to generate efficient code. Pallene's type system limits several aspects of Lua; some of these limitations reflect on Pallene's syntax. For instance, Pallene's syntax does not contemplate multiple returns, variadic functions, closures, generic iteration, and a few other features. Nevertheless, when removing the type annotations from Pallene code, the code is still valid in Lua.

Figure 3.1 presents the grammar of Pallene using the EBNF notation. Unlike Lua, Pallene does not accept statements in the outermost scope of the program. The outermost scope of a Pallene program forms a Pallene module, which is a set of function declarations and type aliases. These functions can be called from Lua.

Pallene features the following types: booleans, floats, integers, strings, arrays, L-records, and C-records. In this section, we only describe the syntactic aspects of these types; we describe the semantics in the following sections. Pallene's syntax denotes booleans, floats, integers, and strings types as reserved words with the respective names. Array types are denoted by enclosing curly braces and the type of the elements. For instance, `{{float}}` is an array of arrays of floats.

Pallene's syntax denotes both types of records with enclosing curly braces and a list of fields. Each field is denoted by a static identifier followed by a type. Record declarations that have the modifier `C` before the opening curly braces are C-records. Record declarations without this modifier are L-records. For instance, `C{x: float, y: float}` is a C-record with two fields named x

```
module ::= { typealias | funcdecl }

typealias ::= 'type' Identifier '=' type

funcdecl ::= 'function' Identifier '(' paramlist ')' [':' type]
             chunk 'end'

type ::= Identifier | 'boolean' | 'float' | 'integer' | 'string'
       | '{' type '}' | 'C'? '{' paramlist '}'

paramlist ::= Identifier ':' type {',' paramlist}

chunk ::= {stat [';']} [laststat [';']]

stat ::= var '=' exp
       | call
       | 'do' chunk 'end'
       | 'while' exp 'do' chunk 'end'
       | 'repeat' chunk 'until' exp
       | 'if' exp 'then' chunk
         {'elseif' exp 'then' chunk}
         ['else' chunk] 'end'
       | 'for' Identifier ':' type '=' exp ',' exp [',' exp]
         'do' chunk 'end'
       | 'local' Identifier ':' type '=' exp

laststat ::= 'return' [exp] | 'break'

var ::= Identifier | prefixexp '[' exp ']' | prefixexp '.' Identifier

exp ::= 'nil' | 'false' | 'true' | Number | String
      | '{}' | '{' explist '}' | '{' fieldlist '}'
      | prefixexp | unop exp | exp binop exp

prefixexp ::= var | call | '(' exp ')'

call ::= prefixexp '(' [explist] ')'

explist ::= exp {',' explist}

fieldlist ::= Identifier '=' exp {',' fieldlist}

binop ::= '+' | '-' | '*' | '/' | '^' | '%' | '..'
        | '<' | '<=' | '>' | '>=' | '==' | '~='
        | 'and' | 'or'

unop ::= '-' | 'not' | '#'
```

Figure 3.1: Pallene grammar described in EBNF.

and y that are floating-points. The type `{x: float, y: float}` is an L-record with the same fields.

Programmers can use a type alias to avoid repeating the same type declaration in Pallene code. Pallene's syntax denotes a type alias as an identifier followed by a type. For instance, the following type alias bounds a C-record with two fields to the identifier `student`:

```
type student = C{
    name: string,
    grades: {
        course: string,
        grade: float
    }
}
```

After the declaration of this alias, Pallene code can refer to the identifier `student` as the declared C-record. It is worth noting that Pallene does not support recursive types.

Variables in Pallene have their types annotated during compile time. Programmers should annotate the type of the variable after its name. For instance, the following statement declares an array of integers:

```
local xs: {integer} = {1, 2, 3}
```

Functions in Pallene also have their types annotated. All parameters of the function should have a respective type annotation. Omitting the resulting type in a function means that the function does not return a value. Unlike in Lua, Pallene functions can not be reassigned like variables. The following snippet presents an example of a Pallene function that computes the average of an array of floats:

```
function average(arr: {float}): float
    if #arr == 0 then return 0 end
    local sum: float = 0.0
    for i: integer = 1, #arr do
        sum = sum + arr[i]
    end
    return sum / #arr
end
```

By removing the type annotations of this function, the code becomes valid in Lua.

3.2 The Pallene Type System

As we discussed in Chapter 1, Pallene follows the gradual guarantee property of gradually-typed languages. Unless there is a type error, removing the types from a Pallene program results in a Lua program with exactly the same behavior. Type errors in Pallene may happen either in compile time or in execution time. The Pallene type system is what defines these type errors.

We removed some dynamic Lua features from Pallene by omitting them in the syntax, such as the generic for-loop and methods. Nonetheless, these restrictions come from the type system. We can use the type system to explain the differences between both languages. It is worth noting that the type system was not designed to give compile-time guarantees to the programmer. Instead, the type system restricts Lua dynamism to guarantee that an ahead-of-time compiler can generate efficient code.

Even though Pallene has types, the representation of values is the same in both languages. For instance, strings in Pallene are the same strings of Lua. However, not all Lua values have a representation in the Pallene type system. For instance, currently Pallene does not have types that represent nil, maps, and objects.

Lua has the type *number*, which is the union of both integers and floating-point subtypes. During execution, the Lua interpreter converts between these subtypes automatically when necessary. In Pallene, we opted not to include *number*, but these subtypes as separate types. This design decision removes from Pallene the requirement to make runtime conversions between the number subtypes. It also gives more room for optimizations, since the Pallene compiler knows the internal representations of numbers during compile time.

Pallene triggers a runtime type error when the programmer passes the wrong numeric type to a function. Consider the following Pallene function that receives a float:

```
function double(x: float): float
   return x * 2.0
end
```

If Lua calls this function passing an integer, Pallene triggers a type error rather than of coercing the value. By removing the type annotations, the Lua version produces the same result that Pallene does. Pallene could produce a different result from Lua if it were to coerce the integer to a floating-point. For instance, if Lua passes the integer 4611686018427387903, the result of the Lua function would be 9223372036854775806. By converting this integer to a float, Pallene

would lose precision and return 9223372036854775808. Hence, Pallene triggers a type error instead.

In the place of coercion, numeric operations in Pallene are polymorphic as in Lua. For instance, Pallene automatically converts integers when there is an addition to a floating-point. However, the current version of Pallene restricts some operations that are polymorphic in Lua. The concatenation operator only accepts strings, and comparison expressions only accept values of the same type.

Control statements (if, while, repeat) and logical expressions (and, or, not) only accept boolean expressions. Due to this limitation, the Pallene type system does not accept some common Lua idioms. For instance, there is an idiom that uses the or-expression to assign a default value to a variable. The following line presents this idiom in Lua:

```
x = x or 10
```

If the value of the variable x is nil or false then the expression returns 10, otherwise it returns x. The type system does not accept this statement because the expressions are not booleans and because there is no optional type. In Pallene, the variable `x` could not be neither nil nor false.

3.3 Typing Lua tables

Lua tables are versatile. Programmers use them to represent arrays, records, maps, sets, objects, modules, and even the global environment. Using the same data structure for all these applications is only viable in a dynamic language. It is even possible to represent two structures at once with a single table. The following line shows a common idiom where a table represents both an array and a record simultaneously:

```
local line = {
    {x = 1.3, y = 3.2},
    {x = 4.2, y = -1.7},
    {x = 8.1, y = 7.5},
    color = "red"
}
```

Pallene does not try to mimic the dynamic behavior of tables since it would be too complicated to describe it with a static type system. A static type that covers all table applications would also hinder optimizations that an AOT compiler can do. Instead, Pallene defines stricter types that are built over Lua tables: arrays and L-records. Programmers may create an array or an L-

record in Pallene, and return it to Lua as a plain table. Likewise, programmers can create a table in Lua and pass it to Pallene where it expects an array or an L-record. Pallene may trigger a runtime type error if the entries in the table do not comply with the corresponding type annotation.

Pallene arrays are tables where all keys are integers, and all values have the same type. Pallene triggers a runtime type error if the code accesses an element that has a type different from the annotation. (Since the absence of entries in tables results in nil, Pallene also triggers this same type error when the field is absent.) For instance, suppose Lua calls the function `average` presented in the previous section with the following table:

```
average({1.2, 3, 4.6})
```

This function expects an array of floats but the value in the second entry is an integer. Pallene verifies that the type of the element is not the one expected and triggers a type error.

In Pallene, arrays grow automatically when the program adds new elements to it. The program can add a new element in any position of the array; the only limit is the memory available to the program. When doing so, Pallene does not automatically initialize any other elements in the array. The code may initialize an array in the reverse order, but it needs to assign all elements to make it a valid sequence again.

L-records and C-records map static fields to values of a given type. These fields are represented by string keys in Lua tables. Just like arrays, Pallene verifies the type of the fields when accessing them.

Pallene does not trigger a type error if tables contain other keys in addition to the ones expected. Hence, an array coming from Lua may have string keys and L-records may have integer keys. However, Pallene code has no way to access these extra fields. For instance, suppose the following Pallene function that receives an L-record that represents a point:

```
type point = {
    x: float,
    y: float
}
function dot(a: point, b: point): float
    return a.x * b.x + a.y * b.y
end
```

Pallene does not trigger a type error if Lua passes the following table to the function *dot*, even though this table has an extra field:

```
dot({x = 1.0, y = 2.0, z = 3.0})
```

Pallene has another type of records called C-records which has the same set of operations of L-records. However, C-records are not built over Lua tables. Instead, C-record values are represented by userdata objects in Lua. Programmers can create C-records in Pallene and pass them to Lua. Lua code can write and read the fields of C-records using the usual dot notation. In chapter 5, we describe in more depth the differences between the two types of records.

3.4 Type Verification

We want the Pallene compiler to have as much room for optimization as possible. Nevertheless, Pallene still is a type-safe language. Pallene must verify the type of values it receives from Lua before manipulating them. Lua can pass values that are incompatible with the types Pallene expects since Lua is a dynamic language.

To give more room for optimization, Pallene allows its compiler to trigger a runtime error at any point of a function. If Lua only passes values that Pallene expects, the program should behave as expected. Otherwise, the Pallene compiler has the freedom to trigger a runtime error at any point. The runtime error might happen at the beginning of the function or just before Pallene uses the value. There are no rules about when the error happens.

Going even further in this direction, Pallene does not need to verify the type of values it does not use. For instance, if there is an execution path that does not use one of the parameters in a function, Pallene can either verify its type or not. The following snippet presents a record with three fields and a function that receives this record:

```
type person = {
    firstname: string,
    lastname: string,
    age: integer
}

function fullname(p: person): string
    return p.firstname .. ' ' .. p.lastname
end
```

Observe that this function does not access the field *age*, so Pallene does not need to verify whether this field is present in the table. The Pallene compiler can choose what is more efficient to do.

As another example, consider the following Pallene function:

```
function copy(arr: {float}): {float}
    local new: {float} = {}
    for i = 1, #arr do
        new[i] = arr[i]
    end
    return new
end
```

If the received parameter is indeed an array of floats, the function behaves as expected. The code generated by the Pallene compiler can trigger a type error in the middle of the procedure if it finds an incompatible element. However, since Pallene does not use the value directly, the compiler is free to generate another version of this function. In this version, the code copies the elements from one array to the other without checking their type first. Even without checking the type, the language is still safe because it does not manipulate the value directly.

4
The Internals of the Lua Interpreter

Lua provides an API so C programs can manipulate Lua values. However, this API has abstractions that introduce a runtime performance overhead. We want to obtain the best performance possible when manipulating Lua values in Pallene. Therefore, the code that the Pallene compiler generates bypasses the C API, accessing the internals of the Lua interpreter directly. In this chapter, we give a brief overview of the Lua-C API and we also present the internal data structures of the Lua interpreter that Pallene manipulates. (We cover the version 5.4.0-work1 for the Lua interpreter.) Pallene uses C closures to represent Pallene functions, Lua tables to represent arrays and L-records, and full userdata objects to represent C-records. Knowledge of the Lua interpreter's internals is important to understand the Pallene implementation that we describe in the next chapter.

4.1
Lua-C API

The Lua-C API creates an abstraction layer that helps C programmers. Instead of manipulating the interpreter structures directly, there is a set of API functions to help them. Moreover, the C API is mostly stable among different versions of Lua. When a new version of Lua is released, the programmer has only to make minor changes to his code, if any. Meanwhile, the internals of the interpreter are free to change in different versions of Lua.

The Lua-C API does not expose any internal data structure of the interpreter to the C programmer. The programmer must use the functions of the C API to manipulate Lua values. All Lua values that the C code manipulates must stay in an abstract stack of values. The Lua stack is opaque to programmers using the C API, so they must use the API functions to manipulate the stack. For instance, the function `lua_tonumber` obtains a floating-point number in a given position of the Lua stack. The function `lua_pushnumber` inserts a number into the top of the Lua stack.

Figure 4.1 exemplifies the use of the Lua stack when manipulating a Lua table. First, the code in the figure obtains the table `t` from the global environment by calling the function `lua_getglobal`. This function pushes the

```
lua_getglobal(L, "t");
if (lua_getfield(L, -1, "x") != LUA_TNUMBER)
    luaL_error(L, "x must be a number");
lua_Number id = lua_tonumber(L, -1);
```

Figure 4.1: Snippet that shows how C code obtains a field from a table using the Lua-C API.

value of the global variable into the top of the Lua stack. Like all API functions that manipulate the Lua stack, `lua_getglobal` expects the Lua state as the first argument (represented by the variable `L` in the figure).

In the following line, the code calls the function `lua_getfield` to obtain the entry indexed by the string `"x"`. This function expects as the second argument the position of the table in the Lua stack. A positive integer indicates that the position starts from the base of the stack and a negative integer indicates that it starts from the top of the stack. The code passes -1 to the function `lua_getfield` since the table is at the first position starting from the top of the stack. The function `lua_getfield` pushes the value indexed by the passed string in the top of the stack and returns the type of the value.

The code of the figure uses the return of `lua_getfield` to verify if the type of the value is a number. If the value is not a number, the code calls the function `luaL_error` passing an error message. This function triggers a Lua error and throws an exception (via long jump). In the last line, the code calls the function `lua_tonumber` to obtain the number from the Lua stack. This function also expects the position of the value in the Lua stack. The position of the value is -1 since the function `lua_getfield` pushed it into the top of the stack.

The Lua-C API allows programmers to manipulate Lua values in C. However, the C API also introduces a runtime overhead when compared to manipulating the values of the interpreter directly. Part of the overhead is caused by the computation the real position of the value in the stack based on a relative position. A compiler cannot optimize this kind of computations since they are inside the API function. Even the extra call to the API function introduces a small overhead that does not exist when accessing the Lua values directly.

4.2 Representation of Lua Values

The Lua interpreter represents the Lua dynamic values as tagged values. A tagged value is a structure that contains both the actual value and an integer

```
typedef union Value {
  struct GCObject *gc; /* collectable objects */
  void *p; /* light userdata */
  int b; /* booleans */
  lua_CFunction f; /* light C functions */
  lua_Integer i; /* integer numbers */
  lua_Number n; /* float numbers */
} Value;

/* ... */

#define TValuefields Value value_; lu_byte tt_

typedef struct TValue {
  TValuefields;
} TValue;
```

Figure 4.2: Declaration of Lua tagged values (defined in `lobject.h`).

tag representing the type of the value. Figure 4.2 presents the declaration of the struct that defines Lua tagged values. The union Value contains the different C types that a tagged value can store.

Depending on the type, the Lua interpreter stores the Lua value directly in the tagged value. The types of these values can be nil, boolean, number (integer and floating-point), and light *userdata*. The interpreter dynamically allocates memory in the heap to store values of other types; we call these values heap-objects. For such types, the tagged value stores only a pointer to the heap object. Heap-objects can be strings, tables, functions, threads, and full userdata. The Lua garbage collector keeps track of heap-objects and frees them when there are no more references to them.

4.3 C Functions

Programmers can expose C functions to Lua using the C API. These functions should follow the Lua calling convention so the interpreter can call them. In C, these functions receive just one parameter which is an opaque data structure called Lua state. The Lua states allow the programmers to manipulate the Lua stack. The interpreter places the actual arguments to the function in the Lua stack. It is worth noting that Lua supports multiple returns. The C function should place the return values in the Lua stack and return in C the number of values it is returning in the Lua stack.

Figure 4.3 presents a C function that can be exposed to Lua with the

```
int swap(lua_State *L)
{
    lua_pushvalue(L, 2);
    lua_pushvalue(L, 1);
    return 2;
}
```

Figure 4.3: C function that uses the Lua-C API to swap two values.

C API. The function `swap` receives two Lua values and returns them in the inverse order. `Swap` uses the function `lua_pushvalue` to rearrange the Lua stack. The function `lua_pushvalue` receives a stack position and pushes a copy of the value in that position into the top of the stack. The function `swap` uses `lua_pushvalue` to place the second and then the first arguments into the top of the stack in that order. Finally, `swap` returns 2 in C, indicating that it is returning to Lua the two values at the top of the stack.

When creating a C function value in Lua, programmers can associate a list of Lua values to it, forming a C closure. The values associated with the C closure are called *upvalues*. C code may access these upvalues by using the C API during the closure call. Internally, a C closure is a heap object that contains a pointer to the C function and the array of upvalues. (This array has a fixed size.)

Programmers can also create C functions that do not have any upvalues associated with them. These functions are called light C functions. Light C functions just contain the pointer to the actual function. Hence, the interpreter does not need to create a heap object to represent them. Instead, the interpreter stores light C functions directly in the tagged value.

4.4 Tables

Tables are the sole data-structure mechanism in Lua. They are associative arrays that map keys of any type to any values of any type. The name “table” comes from *hash tables*, which was how older versions of Lua implemented them. For performance reasons, hash tables are not the best representation for arrays of values. Starting from Lua 5.0, Lua tables contain a part specialized for arrays in addition to the hash table (IERUSALIMSCHY et al., 2005). Lua hides the hybrid implementation from the programmer. The interpreter does not allow programmers to manually place entries in a specific part. Instead, the Lua interpreter places the entries with integer keys automatically.

Lua has this hybrid implementation exclusively for optimization pur-

poses. The hybrid implementation improves Lua's performance in two ways. First, it accelerates the access to fields in the array part, since there is no need to compute the hash of the key. Second, it also reduces the memory footprint since the array keys are implicit; the array part only stores the values of the entries.

The array part stores values indexed by positive integer keys. The position of the entry in the array part represents the key. Entries indexed by negative integers or keys of other types stay in the hash part. The Lua interpreter also has an optimization for sparse arrays and entries with large integer keys. The interpreter does not allocate a huge array if it were to be mostly empty; instead, it places these entries in the hash part too.

The Lua interpreter keeps the array part capacity to the largest power of 2 such that more than half of the entries are not nil. Entries with integer keys larger than the array capacity stay in the hash part. The capacity of the hash part is the smallest power of 2 that fits all remaining entries. For instance, consider a Lua table that contains entries with the keys 1, 2, 3, 5. The capacity of the array part of this table is 4 and the capacity of the hash part is 1. The interpreter places the first three entries in the array part and the entry with the key 5 in the hash part. The interpreter does not place all the entries in the array part since the capacity would be 8 and the number of empty entries would be 4.

When the programmer inserts new entries to the Lua table, the interpreter grows the table automatically if there is not space for the entry. When growing the table, the interpreter performs a rehash operation. The first step of this operation is to find the capacity of the array part according to the largest power-of-2 rule. Then, the interpreter computes the capacity of the hash part to fit the remaining entries. After knowing the capacities of both parts, the interpreter rearranges the entries accordingly.

Figure 4.4 illustrates the rehash process. In this figure, there is a Lua table with integer keys being initialized in the reverse order. The Lua interpreter first inserts the entries with keys 5, 4, 3 in the hash part, since there are not enough entries to fill more than half of an array multiple of two. At this point, the hash part has capacity 4 and the array part capacity 0. The entry with the key 2 also goes to the hash part since there is space for it in the hash part. Finally, the interpreter performs a rehash operation when it inserts the entry with the key 1. Then, it transfers all entries from the hash part to the array part; the capacity of the hash part becomes 0 and the capacity of the array part becomes 8.

When the programmer creates a table, the Lua interpreter allocates

```
local a = {}
for i = 5, 1, -1 do
    a[i] = i
end
```

Figure 4.4: Lua array being initialized in the reverse order.

the exact amount of memory necessary to fill its capacity. Only when the programmer inserts a new entry to the table that the interpreter follows the largest power-of-two rule. This behavior avoids waste of resources if the array does not increase. For instance, suppose a Lua table with being initialized with the following list: `{'a', 'b', 'c', 'd', 'e'}`. The initial size for the array part in this table is 5. If the programmer inserts a new entry to the table, the array size should become 8. Even if Lua places the new entry in the hash part, the insertion triggers the rehash operation that resizes the array part as well.

Lua tables also do not shrink automatically. The Lua interpreter only resizes the table in the rehash operation. In its turn, the interpreter only triggers the rehash operation when there is not space for a new entry. Hence, if a programmer removes an entry from a table, the interpreter does not reduce its size. For instance, a programmer can create an array with 1000 entries and later assign nil to all of them. The array part stays with 1024 slot and all of these slots are empty.

4.5 Userdata

Programmers can create values of the type *userdata* with the C API. These values represent C pointers and are opaque in Lua; Lua code cannot manipulate a C pointer directly. Lua code simply holds a reference to the pointer and sends it back to C.

There exist two subtypes of userdata in Lua: light userdata and full userdata. Just like light C functions, a light userdata is just a C pointer. The interpreter stores light userdata values directly in the tagged value. A full userdata is a heap object and the tagged value stores a pointer to it.

Programmers can use the C API to create a full userdata with the function `lua_newuserdata`. This function receives the wanted size for the memory chunk and returns a pointer to the chunk. The function `lua_newuserdata` also pushes the userdata into the top of the Lua stack. Since the full userdata is a heap object, the Lua garbage collector is responsible for freeing it.

A common mistake that Lua programmers make is accessing a userdata after removing it from the Lua stack. Figure 4.5 presents a C snippet that

```
lua_getglobal(L, "f");
struct data *d = lua_newuserdata(L, sizeof(struct data));
fill_data(d);
lua_call(L, 1, 0);
print_data(d);
```

Figure 4.5: C snippet that illustrates a common mistake when using the C-API.

illustrates this mistake. The first line of the code obtains the global `f` with the function `lua_getglobal`. The C code expects that this value is a Lua function. Then, the code creates a userdata with the function `lua_newuserdata` and initializes the userdata with the function `fill_data`. At this point, the Lua stack top values are the function `f` and the userdata.

In the following line, the code calls the function `lua_call`. This function calls a Lua function in the stack passing arguments that are also in the stack. The second parameter of `lua_call` indicates the number of arguments and the third parameter indicates the number of expected return values. Before calling `lua_call`, the programmer should push into the top of the stack the function followed by the arguments. After the call, `lua_call` removes the function and the arguments from the stack leaving just the return values. Since the code in Figure 4.5 passed the integer 1 as the second paramter, `lua_call` calls the function `f` passing the userdata as the argument.

The function `lua_call` pops the userdata object from the Lua stack before it returns. In the next line, the code calls the function `print_data` that accesses the contents of the userdata to print it in the screen. However, if the Lua code of the function `f` did not maintain any references to the userdata, the Lua garbage collector might have freed it. Accessing the userdata after popping it from the stack might lead to a use-after-free memory error. To solve this issue, the programmer should push into the Lua stack another reference to the userdata with `lua_pushvalue` before calling `lua_call`.

Full userdata objects can keep references to other Lua values. Programmers cannot store heap objects in the memory part of the userdata since the C API does not provide pointers to them. Instead, full userdata objects have a separate area for Lua values. Starting from Lua 5.4, a userdata can have an array of values just like C closures. (Before version 5.4 of Lua, a userdata could only store a single Lua value.) The Lua values stored in a userdata are called user values. Full userdata objects contain the memory part accessible through a pointer and the array of user values.

Another mistake is keeping a reference to another userdata in the memory part. The Lua garbage collector can only reach the values in the user values

array. If the reference to the other userdata is only in the memory part, the garbage collector does not reach this reference. Hence, the other userdata might be collected and the pointer to it in the memory part would become invalid. All heap objects should be stored in the user values array.

5
Pallene Implementation

We wrote an ahead-of-time compiler for Pallene in Lua[1]. Our compiler receives a single Pallene source file as input. The output of the compiler is an extension library for Lua. An unmodified version of the Lua interpreter can load this library like any other extension library. After loading the Pallene library, Lua code can call Pallene functions. In this chapter, we describe the implementation of the Pallene compiler.

5.1
The Architecture of the Pallene Compiler

Our compiler's first step is to parse the input file with the help of the LPeg pattern-matching library (IERUSALIMSCHY, 2009). Hence, we converted the CFG presented in Chapter 3 to a valid PEG. The compiler also uses the LPegLabel library in addition to LPeg to handle syntax errors (MEDEIROS; MASCARENHAS, 2018).

During the parsing step, our compiler builds an abstract syntax tree (AST) based on the Pallene input code. After the compiler builds the AST, it checks whether the static type annotations comply with the Pallene type system. The compiler stops the compilation process if it catches any type errors during this step. The compiler also binds the variables names to its declarations during the type checking step.

After the type checking step, the Pallene compiler generates C code by traversing the AST. Generating C code has three main advantages when compared to generating assembly code. First, it is easier to write a compiler that generates C since it is higher-level than assembly. One line of C code can be equivalent to several lines of assembly. Furthermore, it is easier to read C than assembly, which facilitates the debugging process during the development of our compiler.

The second advantage of generating C code is portability. C is a portable language; there are several C compilers available for a wide range of architectures. By generating C, our compiler does not need to perform low-level tasks,

[1]The source code of the Pallene compiler for this dissertation is available at `https://github.com/pallene-lang/pallene/tree/gabriel-masters`.

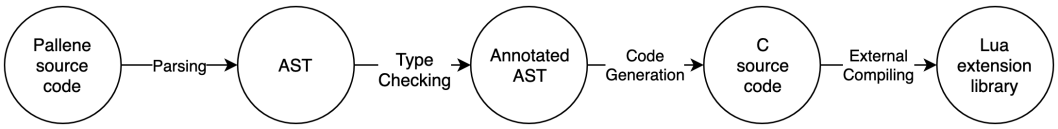

Figure 5.1: Data flow diagram presenting the steps of the Pallene compiler.

such as allocating registers and generating code for the platform calling convention. The Pallene compiler does not generate any platform-specific code; the code that our compiler generates is plain C99. Any C compiler that complies with the standard can compile the C code that our compiler generates.

The third advantage of generating C code is leveraging the optimizations that a C compiler already performs. A mature C compiler like *GCC* already performs several low-levels optimizations over the C code. Our compiler generates code that is amenable to these optimizations. It is important for our compiler to perform such optimizations before generating the final library. However, it would require a considerable amount of effort to implement these optimizations in our compiler. Generating C allows us to leverage the optimizations already done by mature C compilers.

We considered using the compiler framework LLVM for our compiler instead of generating C code. LLVM provides the same advantages of a C compiler when compared to generating assembly. However, C is still more portable than LLVM; GCC by itself supports more platforms than LLVM (LLVM, 2019b). We chose to generate C since we want the Pallene compiler to be as portable as possible.

After generating the C code, our compiler invokes a C compiler (e.g. GCC) to generate the final Lua extension library. This process is transparent to the programmer; our compiler does not expose the temporary C code. Figure 5.1 presents a data flow diagram with all the steps that our compiler performs.

The Lua extension library that our compiler generates follows the ABI that the Lua interpreter expects. An unmodified version of the interpreter can load this extension library using Lua's `require` function. Like other Lua extension libraries, the generated Pallene library exports a Lua table with Pallene functions. From the point of view of the Lua interpreter, Pallene functions are just C closures. These functions follow the Lua calling convention; hence, the Lua code can call these functions normally.

Even though Pallene functions are C closures, the code that our compiler generates bypasses the Lua-C API. Instead of using the Lua-C API, the code manipulates the data structures of the Lua interpreter directly. This allows Pallene to overcome the runtime overhead caused by some of the abstractions

in the Lua-C API. Moreover, accessing the internals of the interpreter directly gives more room for the C compiler to optimize the code generated by our compiler.

However, there is a disadvantage when bypassing the Lua-C API. The Lua-C API is stable among different releases of the same Lua version. For example, Lua 5.3.1, 5.3.2, and 5.3.3 are three different interpreter releases for the version 5.3 of the language. All these releases have the same Lua-C API, but the internal data structures of the Lua interpreter can be different. The code that our compiler generates is therefore bound to a specific release since it bypasses the Lua-C API. During the library initialization, Pallene code verifies that the release of the Lua interpreter is compatible with the library.

5.2 Manipulating Lua Tables in Pallene

One of the key features of Pallene is being able to manipulate Lua values. Pallene can manipulate Lua tables, even though they are dynamic data structures. To manipulate a dynamic data structure in a static language, Pallene introduces two types that are built over Lua tables: arrays and L-records. When Pallene creates a data structure of these types, the code creates a Lua table. When Lua passes a table to Pallene, the code expects either an array or an L-record. To manipulate both types of data structures, the code that our compiler generates accesses the table structure directly.

5.2.1 Arrays

When looking for an element in a Pallene array, the code first verifies the tag of the Lua value to check whether it is a table. In Section 5.4 and Section 5.5, we describe when our compiler perform these verifications. After verifying the tag, the code performs a bounds checking. If the key is smaller than the capacity of the array part then the code accesses the array part of the table directly. Otherwise, the code uses the function `luaH_getint` to look up for the entry in the hash part of the table. Once the code finds the entry, it performs a type verification and obtains the expected element.

Figure 5.2 illustrates the code that our compiler generates when looking up for an element in a Pallene array of floats. In this figure, `i` is the integer key and `t` is the pointer to the Lua table. In the first line, the code converts the 1-based Lua index to a 0-based index. Then, the code performs the bounds checking; the code accesses the array part directly if the index is smaller than the capacity of the array part. Otherwise, the code uses the function

```
unsigned int ui = i - 1;
TValue *slot;
if (ui < t->sizearray) {
    slot = &t->array[ui];
}
else {
    slot = luaH_getint(t, i);
}
if (!ttisfloat(slot)) {
    array_type_error(L, LUA_TNUMFLT);
}
lua_Number v = fltvalue(slot);
```

Figure 5.2: Code that obtains a float element at the position `i` of the table `t`.

`luaH_getint` to look up for the entry in the hash part. After looking up for the entry, the code verifies whether the type of the value in the slot is a float by using the macro `ttisfloat`. If the value is not a float as Pallene expects, the code triggers a Lua error calling the function `array_type_error`. Finally, the function obtains the actual float value from the slot by calling the macro `fltvalue`.

The code for inserting an element in a Pallene array is similar to the code for retrieving an element. After verifying the tag, the code looks up whether the table already contains a slot with the given key. The code performs a bounds checking and tries to obtain the element from the array part if it is smaller than the capacity. Otherwise, the code looks up for the entry in the hash part with `luaH_getint`. If the code does not find the entry in the hash part, it uses the function `luaH_newkey` to create a new entry for the given key. The function `luaH_newkey` grows the Lua table if necessary. Once the code has the slot of the entry, it writes the value to the slot using the macro `setfltvalue`.

Notice that the code for array manipulation has a fast path when the element is in the array part. In this fast path, the code avoids calling functions. Instead, the code either access Lua's data structures directly or calls macros defined by the Lua interpreter. This fast path avoids the unnecessary overhead of calling functions for the case that we expect to be the most common one.

5.2.2 L-records

Fields of L-records stay in the hash part of Lua tables because they are indexed by strings. To manipulate these fields, our compiler needs to compute the hashes of these strings. However, it is not possible to pre-compute these hashes at compile time because the Lua interpreter randomizes string hashes

```
TString *key = tsvalue(&f->upvalue[1]);
static int pos = 0;
TValue *slot = pallene_getstr(t, key, &pos);
lua_Number v = fltvalue(slot);
```

Figure 5.3: Code that reads a float from the table `t`. The string that represents the field is stored in the second entry of the upvalues array of the function `f`.

during runtime. Hence, a given string has a different hash for each program run. The interpreter performs this randomization for security purposes.

To avoid repeating the computation of the string hashes, the Lua interpreter stores the hash inside the string object. The interpreter computes the hash only once when the string is created. Later, the interpreter recovers the hash from the string to look up an entry in a table.

Our compiler also uses the hashes stored in strings to look up fields of L-records. The code initializes the strings that represent the fields of L-records when the module is loaded. The code stores these strings in the upvalues arrays for each Pallene function of the module. When accessing a field, the code recovers the respective string from the upvalues array.

Besides recovering the hashes from strings, our compiler uses inline caching (DEUTSCH; SCHIFFMAN, 1984) to further optimize the manipulation of L-records. In this optimization, the code creates a cache of the last position that the field was found in the hash table. When accessing the field, the code first checks whether the entry at the cached position is the one being looked for. If the entry is not the one expected then the code performs the look up normally and updates the cached position.

Figure 5.3 presents the code that our compiler generates to obtain a floating-point field from a L-record. In the first line, the code obtains from the upvalues array the string that represents the field. In the following line, the code declares the static variable that caches the last position that the field was found. Then, the code calls the function `pallene_getstr` that obtains the slot indexed by the given string key. This function first checks the cached position and updates it if necessary. Finally, the code obtains the actual floating-point with the macro `fltvalue`.

Writing a field to a L-record is similar to reading a field. The code also has a inline cache and uses the function `pallene_getstr` to obtain the slot of the field. If there is not a slot for the field in the table, the code uses the function `luaH_newkey` to create a new one. Then, the code calls a macro to write the given value to the slot of the field.

Figure 5.4 presents the function `pallene_getstr`. Notice that this is a

inline function so the C compiler can better optimize the code. First, the code checks whether the cached position is smaller than the size of the hash part, which is given by the `sizenode` macro. If the position is smaller, the code obtains the node at the cached position by calling the macro `gnode`. Then, the code checks whether the key of the node is a string by calling the macro `keyisshrstr`. Finally, the code checks whether the key of the node is equals to the passed key by calling the macro `eqshrstr`. If the keys are equal then the function returns the slot that was found. This is a fast path that uses the position previously cached.

If the inline caching fails, the function `pallene_getstr` looks up the wanted slot like the Lua interpreter. The code computes the modulo of the string's hash by the size of the hash table to start the lookup process. Inside a loop, the code checks whether the key in the current position is equals to the passed key. If the node with the passed key is found then the code updates the cache and returns the found slot. If the key is different, the code checks whether there is a conflict by obtaining the offset to the next node. If the offset is zero then there is no conflict and the function returns a empty object to indicate it did not find the slot. Otherwise, the code updates the current position and continues the lookup process.

5.3 C-Records

As we explained in Chapter 3, C-records provide the same functionality of L-records. Both types of records support the same set of operations in Pallene. The difference between these two types of records is the underlying representation and how Lua interprets them. C-records are not Lua tables like L-records. Instead, the Pallene compiler should use C structs as the underlying representation for C-records. Nonetheless, Pallene can send and receive C-records from Lua and Lua should be able to manipulate the fields of C-records.

The code that our generates uses the Lua full userdata type to represent C-records. Native fields stay in the memory part of the full userdata. Our compiler creates a C struct to arrange these fields in the memory part. For instance, assume a C-record with the following type definition:

```
C{
    a: integer,
    b: float
}
```

Our compiler declares the following C struct to manipulate the fields in the memory part of the userdata:

```
static inline TValue *pallene_getstr(
    Table *t, TString *key, int *pos)
{
    if (*pos < sizenode(t)) {
       Node *n = gnode(t, *pos);
       if (keyisshrstr(n) && eqshrstr(keystrval(n), key))
          return gval(n);
    }
    int currpos = lmod(key->hash, sizenode(t));
    Node *n = gnode(t, currpos);
    for (;;) {
        if (keyisshrstr(n)
            && eqshrstr(keystrval(n), key)) {
            *pos = currpos;
            return gval(n);
        }
        else {
            int nx = gnext(n);
            if (nx == 0)
                return (TValue *)luaH_emptyobject;
            currpos += nx;
            n += nx;
        }
    }
}
```

Figure 5.4: Function that obtains the slot of the passed key in the hash table `t`. This function uses the inline-cached position.

```
struct {
    lua_Integer a;
    lua_Number b;
}
```

Since our compiler bypasses the Lua-C API, the code has access to the internal pointers of the interpreter data structures. However, our compiler cannot store pointers to heap objects in the memory part of the userdata since the Lua garbage collector would not be able to reach these objects. Eventually, the collector would free these objects and the pointers in the memory part would become invalid. Therefore, the code places the fields that store heap objects in the user values array. The garbage collector traverses this array so there is no risk of the values being freed.

Figure 5.5 presents a C-record that contains both native and heap-objects fields. The fields `id` and `age` are native since they are plain integers. The fields `name` and `grades` are heap objects since they store a Lua string and a Lua

```
type Student = C{
    id: integer,
    name: string,
    age: integer,
    grades: {integer},
}
```

Figure 5.5: A C-records that contains both native and heap-object fields.

table respectively. Our compiler stores the native fields in the memory part of the userdata and the heap-object fields in the user values array. For this particular C-record, our compiler declares the following C struct for arranging the native fields:

```
struct {
    lua_Integer id;
    lua_integer age;
}
```

The fields `name` and `grades` are stored in the positions 1 and 2 of the user values array.

5.3.1 Manipulating C-Records in Lua

Lua code cannot manipulate the fields of C-records directly since userdata values are opaque in Lua. It is possible to modify this behavior with Lua *metatables*, which allow programmers to extend the behavior of the Lua interpreter. A metatable is a Lua table that contains a set of *metamethods*. When the Lua interpreter can not perform an operation, the interpreter looks up whether there is a metamethod in the metatable of the object for this operation. Then, the interpreter calls the *metamethods* to perform the operation. Programmers can associate metatables with different kind of Lua values, but we are interested only to associate metatables with userdata objects.

For instance, the interpreter can not perform an addition over two userdata objects that do not have a metatable. However, the interpreter can perform the addition if one of the objects has a metatable with the metamethod `add`. The Lua interpreter calls the metamethod `add` passing the two userdata objects as arguments. In this case, the result of the addition is the return value of this metamethod.

Lua provides several hooks that call metamethods that extends the behavior of the interpreter. In the case of C-records, we are interested in

the `index` and `newindex` metamethods. When Lua tries to access a key in a userdata, the Lua interpreter looks for the `index` metamethod. Assume the Lua code `u[k]`, where `u` is a userdata and `k` is a value that represents the key. When performing this operation, the Lua interpreter looks up for the metamethod `index` in the userdata's metatable. If the interpreter finds the metamethod, the interpreter calls it passing the userdata and the key as arguments. The return of the operation is the return of the metamethod.

The `newindex` metamethod is similar to `index`. When Lua assigns a value to a key of an userdata, the interpreter looks up for the `newindex` metamethod. Assume the Lua code `u[k] = v`, where `u` is a userdata, `k` is the key, and `v` is the value being assigned. The interpreter looks up for the metamethod `newindex` and calls it passing these three values. This operation has no return values.

Our compiler creates a unique metatable for each C-record type declaration in a Pallene module. The code creates the metatables when the interpreter loads the Pallene module. Later, the code associates the metatable to C-record object when the object is constructed. The C-record metatables contain the `index` and `newindex` metamethods so Lua code can manipulate the fields in the C-record. These metamethods are C functions that the Lua interpreter can call.

When the Lua interpreter calls the metamethods `index` and `newindex`, it passes a dynamic value representing the key of the entry to the metamethod. The C-record metamethods have to map this dynamic value to the actual position of the field in the C-record. It is even possible that the passed key is not a field in the C-record.

The C-record metamethods use a Lua table as a dynamic dispatcher. This table maps the string with the name of the field to a function that performs the expected action for the given field. In the `index` metamethod, the action is retrieving the value from the field. In the `newindex` metamethod, the action is setting the value to the field.

Both `index` and `newindex` metamethods contain an unique dispatch table. When the interpreter calls one of these metamethods, the metamethod looks up in the table for the function that performs the operation for the given field. If the table does not contain a function for the given key, it means that the Lua code is trying to access a field that does not exist in the C-record. If that is the case, the metamethod triggers a Lua error. Finally, the metamethod calls the function to perform the expected action for the given field.

Figure 5.6 presents the index metamethod for Pallene's C-records. In the first line of the metamethod, the code retrieves the userdata object from the Lua stack. Then, the code retrieves the key from the stack and verifies whether

```
static int crecord_index(lua_State *L)
{
    Udata *u = uvalue(s2v(L->ci->func + 1));
    TValue *v = s2v(L->ci->func + 2);
    if (!ttisstring(v)) {
        crecord_nonstr_error(L, rawtt(v));
    }
    Table *t = hvalue(u->metatable->array);
    const TValue *f = luaH_getstr(t, tsvalue(v));
    if (!ttisfunction(f)) {
        crecord_index_error(L, svalue(v));
    }
    void (*getfield)(Udata *, TValue *) =
        (void (*)(Udata *, TValue *))fvalue(f);
    getfield(u, v);
    return 1;
}
```

Figure 5.6: Index metamethod for C-records.

the key is a string by calling the macro `ttisstring`. If the key is not a string then the code triggers a Lua error by calling `crecord_nonstr_error`. Later, the code obtains the dispatcher table at the first position in the array part of the userdata's metatable. The metamethod calls the function `luaH_getstr` to look up the specific function in the dispatcher table. If the code does not find the function then the code triggers an error by calling `crecord_index_error`. Finally, the code obtains the pointer to the specific function and calls it passing the userdata and the slot that will receive the return value.

Figure 5.7 shows the functions that our compiler generates for the field `id` of the C-record `student`. In the top, the function `index_student_id` reads the integer value from the field `id`. In the first line of the function, the code obtains the pointer to the memory area of the userdata. Then, the code sets the output Lua value by calling the function `setivalue`. In the bottom, the function `newindex_student_id` writes an integer value to the field `id`. First, it checks if the type of the passed value `v` is an integer by calling the function `ttisinteger`. If the value is not an integer, the code calls the function `c_record_type_error` which triggers a Lua error. Finally, the code obtains the memory part and assigns to the field the integer value obtained with the function `ivalue`.

5.4 Possible Type-Verifications Approaches For a Pallene Compiler

The functions that Pallene export to Lua follows the Lua calling convention. When calling a Pallene function, the Lua interpreter pushes the ar-

```
static void index_student_id(Udata *u, TValue *out)
{
    struct student *s = getudatamem(u);
    setivalue(out, s->id);
}

static void newindex_student_id(
        lua_State *L, Udata *u, TValue *v)
{
    if (!ttisinteger(v)) {
        c_record_type_error(L, "id");
    }
    struct student *s = getudatamem(u);
    s->id = ivalue(v);
}
```

Figure 5.7: Generated functions that perform the index and newindex operations for the field `id`.

guments to the Lua stack and expects the results in the stack. Since Lua is a dynamic language, the values in the Lua stack have their types defined only during runtime. Pallene has to verify whether the values that Lua passed comply with the static type annotation before using them. If the type does not comply with the type annotation, Pallene triggers a runtime type error.

Pallene cannot perform manipulate the values it receives from Lua without verifying their types first. Otherwise, Pallene would not be a type-safe language. For instance, if Pallene code expects an array then the code has to verify whether the value is a table first. If Pallene does not perform this verification and Lua passes an integer, Pallene would interpret this integer as a pointer to a table. Most likely, interpreting this integer as a pointer would result in an invalid-memory-access error.

For verifying primitive types such as booleans, numbers, and strings, the code just needs to compare the tag of the value to the one it expects. If the tag does not correspond to what Pallene expects, the code triggers a runtime type error. However, verifying the types of table-based data structures is more complicated than verifying the types of primitive types. The first step is to verify whether the value is a table but this not enough to verify the whole type. Eventually, the code should also verify whether the tags of the entries comply with the type annotation.

As we explained in Chapter 3, the code can verify the types of the values at any point of the Pallene function. There is a range of possibilities for the moment that the code can verify the type of Lua dynamic values. One extreme

is eager verification: the code verifies the whole type of all Lua values in the entry of the Pallene function. The other extreme is lazy verification: the code verifies the type of the Lua value on demand, just before the code uses the value.

5.4.1 Eager Verification

With eager verification, the code just needs to verify the type of a tagged value once during the whole function call. After the code verifies the type of all arguments, they should remain well-typed until the Pallene function returns. Pallene by itself cannot produce a value that does not correspond to the type annotation of the variable. Runtime type errors only occur if Lua passes a value with the incompatible type to Pallene.

The Pallene compiler can take advantage when the code verifies the types eagerly. Once Pallene verifies the types, Pallene code can move all variable from the Lua stack to the C variables. After the code moves the values to C variables, the C compiler can better optimize the code it generates. For instance, the C compiler can assign these variables directly to machine registers. By using machine registers, the code becomes more efficient since there is no need to retrieve these variables from memory each time they are used.

However, eager verification is not an efficient approach because of table-based types such as arrays and L-records. The code would have to verify the types of all entries of the table to confirm whether the value complies to the respective type annotation. Parts of these type verifications are unnecessary if the Pallene function does not access all the table entries. Verifying all the elements of a table can increase the asymptotic complexity of the function.

For example, Figure 5.8 presents a Pallene function that performs a binary search. If the array contains the expected element, the function returns its position. The algorithm in this function has complexity $O(\log n)$, where n is the size of the array. However, Pallene code with eager type verification would traverse the whole array to verify whether each entry is indeed a float. The complexity of this implementation would be $O(n)$, which is much worse than the complexity of the algorithm.

Besides being able to verify the type of values at any moment, Pallene does not need to verify the type of values it does not use. The code can skip type verifications that are not necessary. In the binary search function of Figure 5.8, the code does not need to verify all the elements of the array. The code just needs to verify the $\log n$ entries of the array it accesses. This line of thought leads to the other extreme implementation of type verification.

```
function binary_search(xs: {float}, v: float): integer
    local lo: integer = 1
    local hi: integer = #xs + 1
    while lo < hi do
        local mid = (lo + hi) // 2
        if xs[mid] < v then
            lo = mid + 1
        else
            hi = mid
        end
    end
    return lo
end
```

Figure 5.8: Pallene function that performs a binary search over an array of floats.

It is worth noting that the eager verification is not possible for type systems that are more complex than the current Pallene type system. For instance, suppose a type system with a function type for Lua tables. Knowing the actual type of the function is not computable since Lua functions are dynamic. Even after calling the function, there might exist another input that makes the function return a value of invalid type.

5.4.2 Lazy Verification

With lazy verification, the code verifies the type of values just before they are used. Therefore, if Pallene does not use the value, the code does not verify the type of the value. More importantly, Pallene does not completely verify the type of table-based data structures. Instead, the code just verifies whether the value is a table and the types of the elements the code retrieves from the table. For instance, in the binary search function with lazy verification, the code only verifies the type of $\log n$ elements that it accesses.

Notice that with lazy verification, Pallene code only verifies the types of tables partially. If Lua passes an array with mixed floats and integers to the binary search function, Pallene might not trigger a type error. Even though the type of the array is wrong, Pallene code might not access the elements with integer types. This behavior is not an issue because it does not break the gradual guarantee property of Pallene.

Differently from eager verification, lazy verification is a viable implementation for our compiler. However, lazy verification brings some performance drawbacks to the code that the Pallene compiler generates. The code verifies

the type of values more than once if Pallene uses the value again. Moreover, lazy verification requires Pallene code to keep variables in the Lua stack instead of in C variables. The C compiler would not be able to map the Lua values to machine registers like C variables.

5.5 Type Verification in The Pallene Compiler

The code generated by our compiler combines both lazy and eager verification approaches. The code performs the first step of type verification eagerly. In this step, the code compares the tag of the value with the one it expects from the type annotation. For values of primitive types, comparing the tag is enough to verify whether the value complies with the type annotation. The code moves the value to C variables after verifying the tag.

For table-based types, the code verifies the tags of the entries lazily. Only when Pallene accesses the entry that the code verifies whether the tag corresponds to the annotation. After verifying the tag, the code moves the value in the entry to a C variable. Notice that the code does not verify the whole type of the entry, it just verifies the tag. If the value in the entry is another table, the code still has to verify the tags of the entries in this other table.

The code moves the value to C variables even if Pallene does not manipulate the value directly. For instance, if Pallene just copies an entry of an array to another array, the value is not actually used. Nonetheless, the code moves the value to a C variable before storing it the other array.

The code does not verify the types of entries it does not access since the type verification is lazy. Lua may pass a table with entries that do not convey with a given type annotation. If Pallene does not access these elements, the code does not trigger a runtime error. However, the programmer should not rely on this behavior since this is an implementation detail of our compiler. The Pallene language specification states that the code might trigger a type error even if the code does not access the entry.

Lua returns nil when the programmer tries to access an entry that is not present in a table. In Pallene, the behavior of the code is similar to Lua when trying to access an absent entry of a table. If the expected entry is not present in the table, the type of the entry should be *nil*. The code triggers a type error since nil should be different from the expected type. Notice that Pallene's type system does not contain the type nil.

The type verification for C-records is different from L-records. When Lua writes to a field of the C-record, the `newindex` metamethod verifies the tag of

the value being written. Lua code cannot write a value with an incompatible tag to a field of a C-record. Therefore, the code does not need to verify again the tags of the fields inside a C-record. However, Lua can still write a table with an incompatible type to a record field. The code has to verify the tags of the entries of this table.

Verifying the tag of a C-record is not enough to verify whether its type conveys with the type annotation. The tag of a C-record just informs that the value is a userdata but not what is inside this userdata. All userdata objects share the same tag so this verifying it not enough to identify which C-record the object is. To verify whether the userdata correspond to the expected C-record, the code compares the metatable of the userdata with the metatable of the expect C-record. Each C-record type has its own metatable, so we can use these metatables to verify the type of the userdata. The code stores the metatable of expected C-records in the upvalues arrays of the Pallene functions. Then, when the code needs to verify the type of a C-record, the code recovers the metatable from the upvalues array.

Our compiler obtains the benefits of both verification approaches by combining them. Comparing the tag eagerly allows the code to move the values from the Lua stack to C variables. However, verifying eagerly the whole type of tables is not efficient. For that reason, the code verifies the type of the elements of tables lazily.

Once our compiler moves the Lua variables to C variables, part of the code generation is straight forward. Our compiler translates most of Pallene statements and expressions over primitive values to the equivalent ones in C. For instance, an addition in Pallene becomes an addition in C; an if-then-else statement in Pallene becomes an if-then-else statement in C. The major challenge in our compiler is generating C code that manipulates Lua data structures.

5.6 Pallene Internal Calling Convention

Once Pallene verifies the tag of a variable, the code does not need to perform this task again. The code cannot assign a value with an incompatible tag to a variable in C variables. We can apply this same principle to function calls that Pallene performs. The code cannot call a function passing arguments with incompatible tags. Hence, a Pallene function does not need to verify the tag of the arguments if the callee is another Pallene function.

The code that our compiler generates uses a different calling convention for calls between Pallene functions. Instead of using the Lua stack, the code

calls other Pallene functions using C parameters. When calling a Pallene function, the code verifies the tag of these parameters before the call.

Pallene functions still need to support the Lua calling convention so the interpreter can call these functions. To feature both calling conventions simultaneously, the functions that our compiler generates have two entry points. One entry point uses the Lua calling convention and the other uses the Pallene calling convention. In the generated C code, each entry point becomes a different C function.

The Lua entry point is a C function that follows the Lua calling convention. This function verifies the tags of the received parameters and moves these parameters from the Lua stack to C variables. Then, the Lua entry point calls the Pallene entry point passing the parameters in C. The Pallene entry point receives the parameters in C and executes the actual code for the function. Other Pallene functions can call the Pallene entry point directly to avoid unnecessary tag verifications.

Figure 5.9 presents the C code that our compiler generates for a function that adds two integers. In the top, the function `add_pallene` represents the Pallene entry point of the function `add`. This function receives two integers, performs the addition, and returns the result through C. In the bottom, the function `add_lua` represents the Lua entry point for the function `add`. In the first line of the function, the code obtains the pointer to the base of the Lua stack. The based of the stack contains the function (Pallene C Closure) followed by the Lua parameters. In the following line, the code obtains the pointer to the first parameter in the Lua stack. Then, the code calls `ttistinger` to verify whether the parameter is an integer as the function expects. If the value is not an integer, the code calls the function `argument_type_error` that triggers a Lua error. The code repeats this process to the second parameter. After verifying the types, the code moves the integer values from the Lua stack to C variables by calling the macro `ivalue`. Later, the code calls the Pallene entry point passing the parameters through C. With the return of the Pallene entry point, the code sets the Lua value above the top of the Lua stack using the macro `setivalue`. The function `api_incr_top` increases the stack size by 1, incorporating the value that was above to the stack. Finally, the Lua entry point returns 1 in C, which is the number of values it left in the top of the Lua stack.

There is a consideration that our compiler has to make when moving values out of the Lua stack. The garbage collector cannot reach objects that are neither in the Lua stack or in the Lua global environment. If the garbage collector performs a step and the object is unreachable, the GC might collect

```
static lua_Integer add_pallene(lua_State *L,
    lua_Integer x1, lua_Integer x2)
{
    return x1 + x2;
}

static int add_lua(lua_State *L)
{
    StackValue* x1 = L->ci->func;
    TValue* x2 = s2v(x1 + 1);
    if (!ttisinteger(x2)) {
        argument_type_error(L, "a", LUA_TNUMINT);
    }
    TValue* x3 = s2v(x1 + 2);
    if (!ttisinteger(x3)) {
        argument_type_error(L, "b", LUA_TNUMINT);
    }
    lua_Integer x4 = ivalue(x2);
    lua_Integer x5 = ivalue(x3);
    lua_Integer x6 = add_pallene(L, x4, x5);
    setivalue(s2v(L->top), x6);
    api_incr_top(L);
    return 1;
}
```

Figure 5.9: The Pallene entry point and the C entry point for a function that add two integers.

the object. The code cannot access objects after the GC collects them. The code needs to be aware of this interaction with the garbage collector to behave correctly.

To make sure that the Lua garbage collector can reach the values in Pallene, our compiler copies the values from C variables to the Lua stack. With this approach, the code can still keep all heap objects in C variables. The code copies the unreachable objects back to the Lua stack before the garbage collector performs a step. Then, the garbage collector can reach all objects that the code uses. After the garbage-collection step is over, the code removes the objects from the Lua stack so it does not need to keep track of two different stacks at the same time.

6
Performance Evaluation

As we discussed throughout this dissertation, the primary goal of Pallene is to aid programmers that seek better performance in Lua. Our hypothesis is that Pallene's performance speedup to be on par with the speedup that LuaJIT achieves. We use LuaJIT as a target because it is the state of the art in optimizing Lua's performance. To test this hypothesis, we measured the running time of eight benchmark programs. In this chapter, we present and analyze the results of this experiment.

A secondary objective of this chapter is to compare the performance of the two types of Pallene records (L-records and C-records). These two types of records offer a similar functionally but they feature two different representations. We expect C-records to be faster than L-records in Pallene, given that C-records are represented by static data structures. On the other hand, in the Lua interpreter, we expect C-records to be slower than Lua tables due to the overhead of metatables.

For each benchmark, we implemented a C version that uses the Lua-C API to manipulate Lua's data structures. Then, we measured the running time of these C programs to compare to Pallene. In this chapter, we use these measurements to analyze the overhead present in the API. We anticipate that these C programs are strictly slower than Pallene because Pallene bypasses the API to access the interpreter internals. In some cases, the overhead of the API makes the C program slower even than Lua-equivalent program executed by the interpreter.

We performed all our experiments in a Linux machine with a processor Intel Core i5-7600 (4.1GHz) and 8 GB of RAM. We used the version 5.4-work1 of the Lua interpreter, the version 2.1.0-beta3 of LuaJIT, and the version 7.2.1 of GCC. In all of our experiments, we measured the running time of the benchmark program 10 times and present the arithmetic mean of these measurements. The standard deviation observed in our experiments was at most 1% of the mean for a given benchmark. Because the deviation was small, we decided to suppress it from the tables we present in this chapter. In our analysis, we disregard differences that are in the same order of magnitude of the deviation.

6.1 Benchmarks Specification

For our main experiment, we selected eight benchmark programs that implement common algorithms in a straight-forward manner. We did not tweak any of the programs to perform better in a particular execution engine (the Lua interpreter, LuaJIT, or Pallene). The major focus of this experiment is measuring the performance of manipulating Lua's data structures. For the benchmarks that use records, we also tested a Pallene version that uses C-records instead of L-records. The following list describes the benchmark programs that we used in our experiment:

- **Binsearch** performs an iterative binary search over an array of random integers. The size of the array is 10^6 and the program repeats the search 10^6 times.
- **Centroid** computes the centroid of an array of 2D points. Each point is represented by a record. The size of the array is 10^5 and the program repeats the computation $5 \cdot 10^5$ times.
- **Conway** simulates the Conway's Game of Life, which is a cellular automaton. The canvas is represented by an array of boolean arrays. The size of the canvas is 40 by 80 and the program performs 2000 simulation steps.
- **Mandelbrot** computes the Mandelbrot set and generates an image. Each complex number is represented by a record. The size of the generated image is 256 by 256.
- **Matmul** multiplies two matrices of floats. Each matrix is represented by an array of arrays of floats. The size of both matrices is 800 by 800.
- **N-body** performs an N-body simulation, which is a simulation of a dynamic system of particles. Each body is represented by a record with the body's coordinates, velocity, and mass. This simulation has 5 bodies and the program performs 10^7 simulation steps.
- **Queens** solves all the possible configurations of a variant of the 8-queens puzzle. In this variant, the chessboard size is 13 by 13. The chessboard is represented by a single array of integers where each array's position contains the column where the queen was placed for that line.
- **Sieve** computes a list of primes with the sieve of Eratosthenes algorithm. The array of primes is represented by an array of integers. The benchmark program also uses a temporary array of booleans to represent the sieve. The benchmark goes through the first 10^5 numbers and repeats the computation 10^3 times.

For each benchmark, we chose a large enough input so the program takes more than 100 milliseconds to run. Programs with linear and sub-linear complexity run extremely fast even with a large input. For these programs, we repeated the execution of the algorithm several times inside a loop. By increasing the running time, we reduce the impact of the noise caused by external factors, such as the operating system.

Notice that by increasing the running time, the warm-up of LuaJIT also becomes negligible. The warm-up is the period of time that the JIT compiler profiles the code and compiles it into machine code. However, LuaJIT is well known for having an extremely fast warm-up (PALL, 2005b). For our benchmarks, the warm-up was less than a millisecond which is in the same order of magnitude of the noise generated by external factors. Hence, we would not be able to measure the warm-up overhead even with a small input size.

6.2 Comparing Pallene to LuaJIT and Lua

Table 6.1 presents the results of our experiments. Each line presents the mean running time for a given benchmark and execution engine. Notice that the last column of the table shows the time of the given execution engine normalized by the time Lua took for that benchmark. Figure 6.1 aggregates these normalized results in a single chart.

By observing the results, we noticed that Pallene's speedup was in the same order of magnitude of LuaJIT's speedup for five benchmarks. For Binsearch and Conway, Pallene performed significantly better than LuaJIT. On the other hand, LuaJIT performed significantly better than Pallene for Mandelbrot. In the following paragraphs, we discuss the results of each program individually.

Binsearch For this benchmark, Pallene took 35% of the time of LuaJIT. The binary search algorithm has an unpredictable branch that leads to two distinct paths, depending on whether the current element is greater or smaller than the element being searched for. This unpredictable branch is particularly bad for LuaJIT because of the nature of tracing-JIT compilation.

By using the Loom debugging tool (GIRALDEZ, 2017a), we could inspect the traces that the tracing-JIT compiler generated. If the current element is smaller than the expected one then the code sticks to the main trace; otherwise, the code jumps to a side-trace. Jumping to a side-trace is an expensive operation since the compiler only optimizes the code inside of a trace. The

code has to jump to the side-trace as it is the only way to perform branching in a tracing-JIT.

The slowdown caused by an unpredictable branch problem is very particular to tracing-JIT compilers. Pallene does not suffer from this issue because it optimizes the loop as a whole; branching is not an expensive operation in Pallene. Method-based JIT compilers do not suffer from this kind of issue as well.

Centroid For this benchmark, LuaJIT took 53% of the time of the Pallene version with L-records. By inspecting the code generated by LuaJIT, we observed that LuaJIT performs an optimization similar to the inline caching in the Pallene compiler. LuaJIT code looks up a specific position first when accessing a field from a record. However, this cached position becomes a constant in the code instead of a variable. Moreover, LuaJIT also hard-codes the pointer to the string key instead of retrieving it in every access. When accessing a field in a table, Pallene still have to access variables to obtain the cached position and the pointer to the string key. Pallene is slower than LuaJIT because an ahead-of-time compiler cannot perform the optimizations that a JIT compiler does. On the other hand, the Pallene version that uses C-records took 71% of LuaJIT's time. When obtaining a field from a C-record, Pallene already knows the exact position of the field during compile time.

Conway For this benchmark, the speedup of Pallene is not in the same order of magnitude of the speedup for the previous benchmarks. By measuring each part of the program separately, we discovered that the bottleneck of this benchmark is string concatenation. Each string concatenation generates a temporary string object that soon becomes garbage. Since Pallene uses the Lua garbage collector, there was not room for Pallene to improve the performance of Lua for this benchmark.

We were initially surprised to find that LuaJIT's performance was worse than the Lua interpreter for Conway. Our main hypothesis is the fact that LuaJIT uses an incremental garbage collector and Lua 5.4 uses a generational garbage collector. The performance gain of the generational garbage collector most likely compensated the performance gain of LuaJIT. Table 6.2 presents a comparison with Lua using the incremental mode for the garbage collector. When compared to Lua with the incremental mode, LuaJIT also improves the performance of the program.

Benchmark Program	Execution Engine	Time in seconds	Time normalized by Lua
Binsearch	Lua	0.71	1.00
	LuaJIT	0.26	0.37
	Pallene	0.09	0.13
Centroid	Lua	8.53	1.00
	LuaJIT	1.27	0.14
	Pallene (L-records)	2.37	0.27
	Pallene (C-records)	0.91	0.10
Conway	Lua	1.63	1.00
	LuaJIT	1.89	1.16
	Pallene	0.85	0.52
Mandelbrot	Lua	0.84	1.00
	LuaJIT	0.013	0.01
	Pallene (L-records)	0.54	0.64
	Pallene (C-records)	0.29	0.35
Matmul	Lua	7.53	1.00
	LuaJIT	0.56	0.07
	Pallene	1.07	0.14
N-body	Lua	15.6	1.00
	LuaJIT	1.48	0.09
	Pallene (L-records)	2.55	0.16
	Pallene (C-records)	1.12	0.07
Queens	Lua	10.7	1.00
	LuaJIT	1.29	0.12
	Pallene	1.09	0.10
Sieve	Lua	3.75	1.00
	LuaJIT	0.95	0.25
	Pallene	0.90	0.24

Table 6.1: Results of the main experiments that compare the Lua interpreter, LuaJIT, and Pallene.

Execution Engine	Time in seconds	Time normalized by Lua incremental
Lua incremental	2.24	1.00
Lua generational	1.63	0.72
LuaJIT	1.89	0.84

Table 6.2: Comparison of Lua with incremental GC, Lua with generational GC, and LuaJIT for Conway.

Execution Engine	Time in seconds	Time normalized by Pallene
Pallene	1.07	1.00
LuaJIT	0.56	0.52
Pallene (BCE)	0.82	0.77

Table 6.3: Comparison of LuaJIT and a hand-optimized version of the Pallene code for Matmul.

Mandelbrot For this benchmark, Pallene did not improve the performance of Lua as we would like. Just like Conway, the bottleneck of this benchmark is memory allocation and garbage collection. Mandelbrot performs several operations over complex numbers represented by records. The result of each operation is a new complex number that soon becomes garbage. Since the bottleneck in Mandelbrot is the creation of temporary objects, Pallene's speedup was not as significant as in the other benchmarks.

On the other hand, LuaJIT took only 1% of the time of the Lua interpreter. By inspecting the code that LuaJIT generates, we discovered that it archived this performance by using an allocation-sinking optimization. In this optimization, the compiler allocates temporary objects in the stack instead of allocating them in the heap. This optimization is only possible when the compiler detects that the object is only used locally. This is an optimization that the Pallene compiler can perform but we did not have time to implement.

Matmul For this benchmark, Pallene's speedup was in the same order of magnitude of LuaJIT's speedup. However, LuaJIT took half of the time of Pallene. By inspecting the LuaJIT code, we noticed that LuaJIT performs a bound-checking-elimination (BCE) optimization. Instead of verifying whether each element is in the array part in every loop iteration, the code verifies whether the loop limit is smaller than the size of the array part. If the loop limit is greater than the array size then the trace handles the execution back to the interpreter.

To test the impact of BCE, we performed this optimization by hand over the C code that the Pallene compiler generates. In the hand-optimized version, the code verifies the size of the array part only once in a loop. If the size of the array part is smaller than the loop limit then the code grows the array part and moves all the elements from the hash part to the array part. (We are currently developing this optimization in the Pallene compiler.) Table 6.3 presents the running time of the hand-optimized version of the Pallene code for Matmul.

Matrices Size	Repetitions	Pallene-LuaJIT Ratio	Execution Engine	Time	LLC miss
1000 by 1000	1	1.66	Pallene	1.78	51.8%
			LuaJIT	1.07	22.6%
800 by 800	1	1.46	Pallene	0.82	34.3%
			LuaJIT	0.56	15.2%
600 by 600	2	1.15	Pallene	0.54	14.9%
			LuaJIT	0.46	6.1%
400 by 400	8	0.93	Pallene	0.50	1.2%
			LuaJIT	0.53	0.6%

Table 6.4: Comparison of LuaJIT and Pallene when varying the sizes of the matrices for Matmul.

Even when performing the BCE optimization, Pallene was still slower than LuaJIT. By investigating further this performance difference, we noticed that the last-level-cache (LLC) miss of Pallene was above 51% while it was 22% for LuaJIT. (We measured the LLC miss using the Linux's `perf` tool (DE MELO, 2010).) Our hypothesis is that the performance difference between Pallene and Lua was due to the LLC-miss rate.

To test our hypothesis, we performed another experiment using matrices of different sizes. Given that smaller matrices require less memory, the LLC miss should be lower for these matrices. In this experiment, we used the hand-optimized version of the Pallene code. For matrices of size smaller than 800 by 800, we repeated the multiplication in a loop to maintain the running time above 100 milliseconds.

Table 6.4 presents the results for this experiment. As expected, the LLC-miss rate is proportional to the size of the matrices. Moreover, the results of this experiment indicate that our hypothesis was correct. When the LLC-miss rate is near 1%, the performance of Pallene is better than LuaJIT's performance.

The LLC miss was smaller in LuaJIT for larger matrices because LuaJIT uses a compact representation for Lua values called *NaN-tagging* (PALL, 2009). With this representation, a Lua value is represented solely by a double floating-point number. LuaJIT encodes values that are not floating-point numbers as NaNs (not a number); the value itself is stored in the mantissa bits of the floating-point. To access non-number values, LuaJIT uses a bitmask to extract the value from the mantissa.

Even though NaN-tagging reduces memory usage, this representation also has a major limitation. It is only possible to store non-floating-point values that are smaller than the size of the mantissa, which is 53 bits for doubles of the IEEE 754 standard. Due to this limitation, LuaJIT does not support 64 bit integers, breaking the compatibility with Lua 5.3 and onwards.

N-body This benchmark is very similar to Centroid; both benchmarks perform arithmetic operations over arrays of records. For N-body, LuaJIT took 58% of the time of the Pallene version with L-records. One of the reasons that LuaJIT was faster is the one presented for Centroid: LuaJIT hard-codes the cached positions and the pointers to the string keys. Besides optimizing the field access, LuaJIT also unrolls the inner loop that iterates over the bodies' array. Instead of performing a loop, LuaJIT creates a linear trace that executes the code of the loop five times (one for each element of the array). This kind of optimization is only possible in a JIT compiler since the size of the array is only known during runtime. Even with the loop unrolling, the Pallene version with C-records was faster than LuaJIT, taking 76% of the time of LuaJIT.

Queens For this benchmark, Pallene took 84% of the time of LuaJIT. To explain why LuaJIT was slower, we need first to explain what the benchmark does to compute the possible solutions of the queen puzzle. The benchmark iterates over every line of the chessboard and it tries to insert a queen in one of the positions of that line. When trying to insert a queen, the benchmark loops through the previous positions to checks whether the current position is safe from another queen attack. For every previous position, the code checks whether there is a queen in it and whether the position is in one of the two diagonals or in the same column. All these checks performed in the innermost loop are branches that become side-traces in LuaJIT. Like Binsearch, these branches are unpredictable, leading to several jumps between side-traces during the benchmark execution. Pallene performs better because it optimizes the loop as a whole.

Sieve For this benchmark, the results of Pallene and LuaJIT were very close; Pallene took 95% of the time of LuaJIT. However, by inspecting the LuaJIT code, we noticed that LuaJIT also performs the BCE optimization for Sieve. We performed this optimization by hand over C code that the Pallene compiler generates to see how further we can push Pallene performance. In this experiment, the hand-optimized version took 0.77 seconds, which is 81% the time of LuaJIT.

6.3 Comparing C-Records with Lua Tables

To compare Pallene C-Records with Lua tables, we use Centroid, Mandelbrot, and N-body benchmarks from the previous section. Table 6.1 already presents the running time of Pallene using C-records and Lua tables (repre-

Benchmark Program	Time in seconds	Time normalized by Lua with tables
Centroid	26.9	3.16
Mandelbrot	1.44	1.71
N-body	70.6	4.52

Table 6.5: Comparison of Lua using C-records for Centroid, Mandelbrot, and N-body benchmarks.

sented as L-records). This table also presents the running time of Lua using tables to represent the records. What is missing for our analysis is the measurement of a Lua version of these benchmarks using C-records.

We performed another experiment with a version of these benchmarks where Lua uses C-records instead of tables. In this experiment, the Lua code performs all the computations and Pallene is only called to create the C-records. Table 6.5 presents the running time of each benchmark for this experiment. The third column of the table presents the running time of each benchmark normalized by the Lua version using tables.

Figure 6.2 presents two charts that aggregate the results of Table 6.1 and Table 6.5. As expected, manipulating C-records in Pallene is faster than manipulating tables, and manipulating tables in Lua is faster than manipulating C-records. For all benchmarks, the overhead of using C-records in Lua is similar to the overhead of using tables in Pallene. For Mandelbrot, the overhead was less significant since the bottleneck was in memory allocation and garbage collection.

Since C-records are represented by static data structures (userdata), the Pallene compiler can generate more efficient code for them. When manipulating a C-record, the Pallene compiler already knows the offset of the field during compile time. Moreover, the code does not need to verify the tag of the field. On the other hand, when manipulating L-records, Pallene has to recover the key from the closure, look up the entry in the hash table, and verify the tag retrieved from the table. All these operations add up making L-records slower than C-records in Pallene.

In Lua, C-records are slower than tables because of the metatable indirection. When manipulating a C-record, the Lua interpreter has to look up the respective metamethod, call this metamethod, and then perform another look up in the dispatcher table. Moreover, when writing to a field in a C-record, the metamethod verifies whether the tag of the value conveys to the type of the field. When manipulating a Lua table, the interpreter just needs to perform a single lookup.

Benchmark Program	Time in seconds	Time normalized by Lua
Binsearch	0.24	0.34
Centroid (tables)	14.7	1.72
Centroid (userdata)	16.7	1.96
Conway	1.47	0.90
Mandelbrot (tables)	1.16	1.37
Mandelbrot (userdata)	0.58	0.69
Matmul	9.61	1.27
N-body (tables)	24.0	1.53
N-body (userdata)	6.09	0.38
Queens	2.98	0.27
Sieve	3.58	0.95

Table 6.6: Results of C programs using the Lua-C API for the benchmarks of Section 6.1.

6.4 Evaluating the Performance of the Lua-C API

We expect the Lua-C API to introduce a runtime performance overhead when manipulating Lua's data structures. To test this hypothesis, we performed an experiment with C programs that use the Lua-C API. For each benchmark program of Section 6.1, we implemented an equivalent version in C that uses the Lua-C API to manipulate Lua's data structures. All these C programs use Lua tables to represent arrays. For the benchmarks that use records, we implemented two versions of each: one using tables and another using userdata objects to represent the records.

Table 6.6 presents the results we obtained for the C programs that use the Lua-C API. The second column presents the running time of the given program and the last column presents the time normalized by the time the Lua interpreter took. The time the Lua interpreter took is the one presented on Table 6.1.

Figure 6.3 presents a chart comparing the running time of C-API, Pallene and Lua. As expected, all programs using the C-API are strictly slower than Pallene. Compared to the Lua interpreter, the performance of the C-API varies greatly. There are programs where migrating to C provided a speedup compared to the Lua interpreter. However, there are programs that the performance of C is equivalent or even worse than the interpreter.

For Binsearch, the C-API performed better than the Lua interpreter, taking 34% of the time of the interpreter. However, Pallene was still considerably faster than the C program, taking 38% of the time of the C program. We observe a similar pattern for the Queens benchmark, where the Lua-C

API is faster than the interpreter but Pallene is faster than the Lua-C API. For Conway and Sieve, migrating from Lua to C improved the performance of the program only by 10% and 5% respectively. For Matmul, the C program was even slower than the Lua interpreter, taking 127% of the time of the interpreter.

Obtaining an entry with an integer key from a table is slow with the Lua-C API because of its abstractions. When manipulating arrays in Pallene, the code accesses the pointer to the table directly. In C, the API functions that manipulate tables receive the index of the table in the Lua stack. For every API call, the code verifies whether the value in the stack is indeed a Lua table before the operation. Moreover, the function call itself introduces an overhead. All theses overheads make the program using the API slower than the Lua interpreter.

All benchmarks that use the C-API to manipulate tables that represent records are slower than the Lua interpreter. The C code uses the API functions `lua_getfield` and `lua_setfield` to manipulate records represented as tables. These functions receive a string constant with the name of the field the code wants to access. Then, these functions have to look up for the respective object in the Lua string cache. The Lua-C API does not expose this string object to C, so it is not possible for the C code to store a pointer to this object. On the other hand, the Lua interpreter stores the pointer to the string constant in the function object, making it faster than the C code.

For the benchmarks where the C programs use userdata objects to represent records, the performance varied greatly. After the C code obtains the pointer to the userdata, the code that the Pallene compiler generates is equal to the C code for our benchmarks. However, both C and Pallene have to verify the metatable of the userdata before obtaining its pointer. In Pallene, the code stores the metatable as a upvalue of the Pallene closure. In C, the code uses the function `luaL_checkudata` to verify the metatable of the userdata. This function receives a string with the name of the metatable and looks up the metatable in the global registry. Retrieving the metatable from the global registry is slower than retrieving a upvalue from a closure, making the metatable-verification process in C slower than in Pallene.

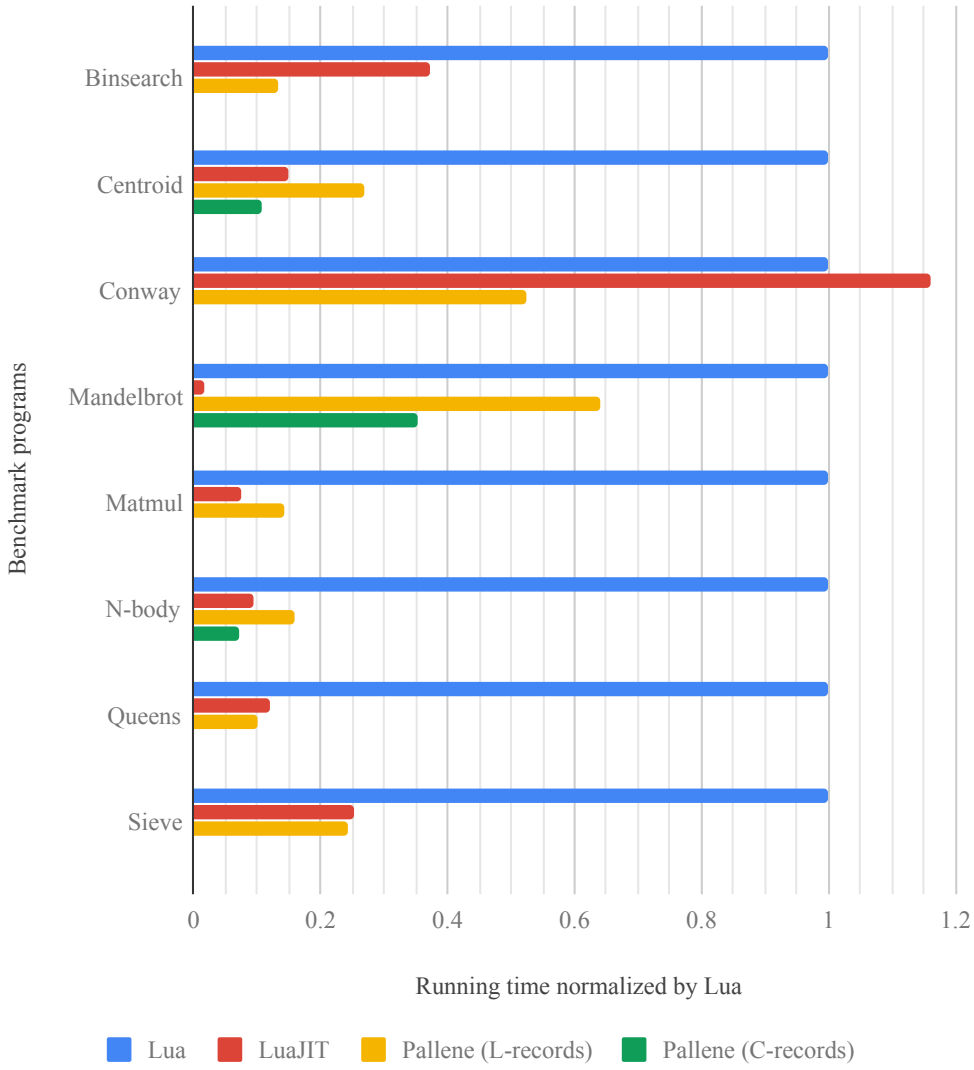

Figure 6.1: Chart comparing of the running time of the experiments of Table 6.1.

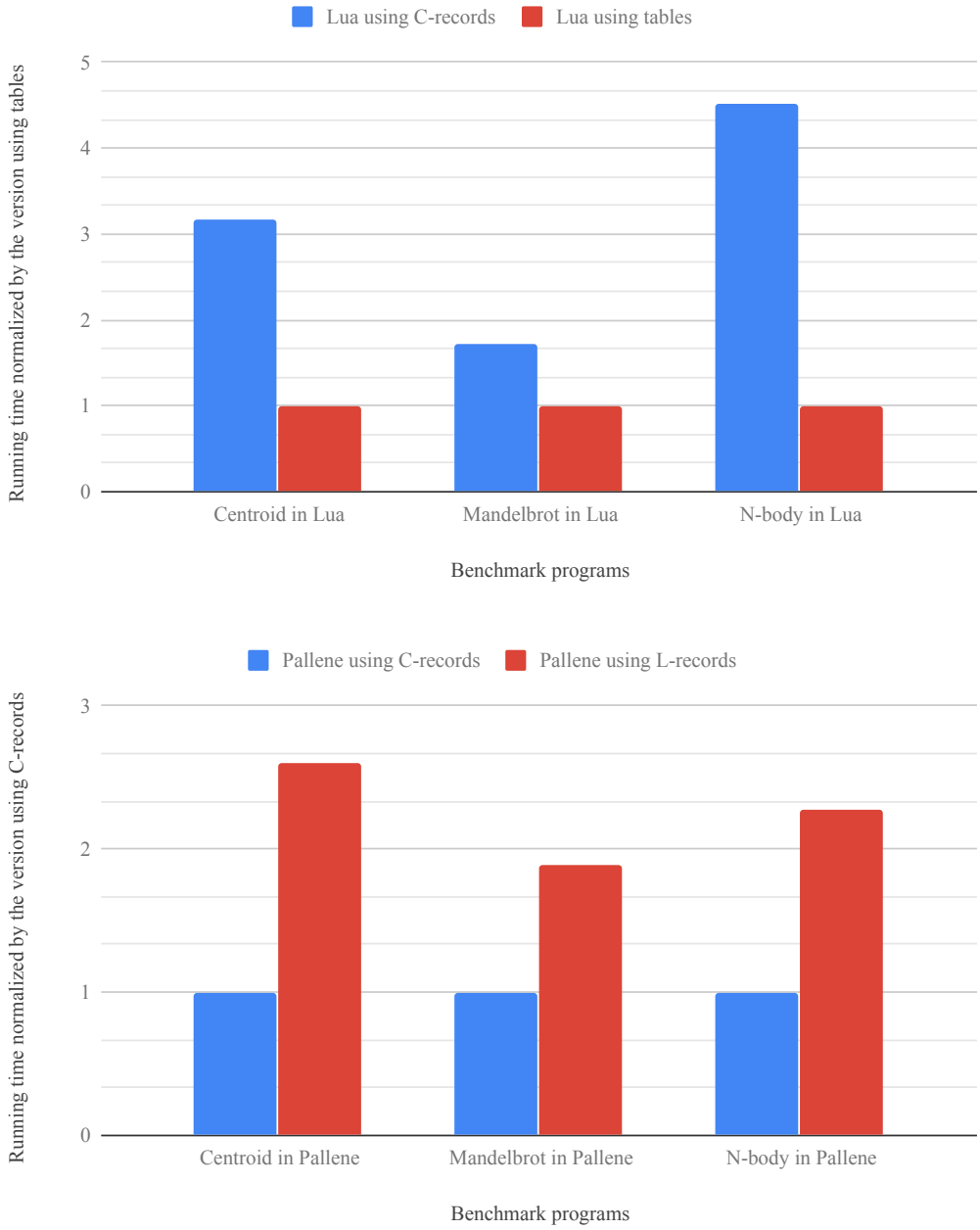

Figure 6.2: On the top, the comparison of using C-records and Lua tables in Lua. On the bottom, the comparison of using C-records and Lua tables in Pallene.

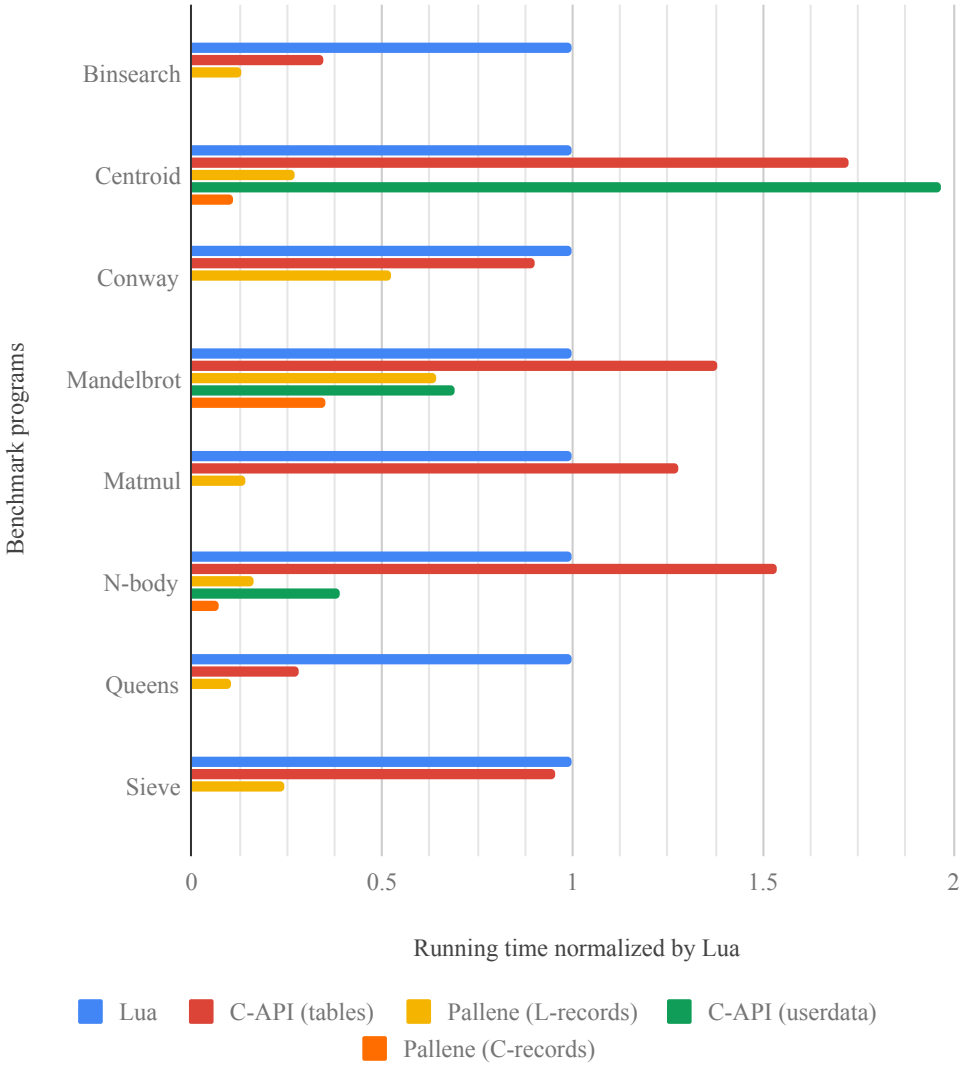

Figure 6.3: Comparison of C programs using the C-API, Pallene, and the Lua interpreter.

7
Conclusions

We start this chapter by comparing Pallene to other systems that introduce static type systems to scripting languages. Even though the companion language approach is unique to Pallene, these other languages served as inspiration for Pallene. Later, we review the key points of this dissertation to evaluate whether we reached our goals. We also review some points we did not have time to work on and suggest avenues for future research.

7.1
Related Works

There are other languages that introduce a static type system to scripting languages to gain performance. In the following paragraphs, we present these languages and explain why they are different from Pallene.

Crystal (CARDIFF, 2019) is a statically-typed programming language that features a syntax inspired by Ruby. By using a similar syntax, the authors of Crystal believe it has a low learning curve for Ruby programmers. Crystal also features a global type inference system so programmers can omit some type annotations. Differently from the relation between Pallene and Lua, Crystal does not aim to interoperate with Ruby; Crystal is a completely separate language. Moreover, there are no guarantees that removing the types of a Crystal program results in a valid Ruby program.

Cython (BEHNEL et al., 2010) is an extension of the Python language that features C type annotations. Like the relation between Pallene and Lua, programmers can write Cython code that is callable from Python. Compared to Pallene, Cython has narrower objectives, which are reducing the overhead for numerical computations and interfacing with existing C libraries. Cython does not feature types that are compatible with the dynamic data structures of Python. Due to this limitation, Cython can not provide large speedups for programs that operate over Python data structures.

RPython (ANCONA et al., 2007) is a restricted subset of Python which can be compiled to efficient machine code. Differently from Pallene, the programmer does not add static type annotations to RPython code. Instead, RPython uses a global type inferring system to generate efficient machine code.

RPython's objectives are also different from Pallene's objectives. While Pallene is a general-purpose language that interoperates with Lua, RPython is part of a framework to build interpreters. RPython code is compiled into standalone executables that do not interoperate directly with Python.

Terra (DEVITO, 2014) is a low-level statically-typed system language that features a syntax similar to Lua. Terra is designed to use Lua as a metaprogramming language, replacing other preprocessing techniques such as C++'s templates and the C preprocessor. Programmers can use Lua to generate high-performance Terra code during runtime. Differently from Pallene, Terra does not provide tools to manipulate the dynamic Lua's data structures.

7.2 Evaluating our Work

We started this work by describing how programmers can use the companion language approach to improve the performance of scripting languages. We compared this approach to the scripting architecture, optional typing and just-in-time compilers. Then, we presented Pallene, which is a companion language for Lua. After presenting the language, we described the ahead-of-time compiler we implemented for Pallene. Finally, we evaluated the performance of our compiler by performing benchmarks and comparing the results with the Lua interpreter, LuaJIT, and C programs that use the C API.

One of the main objectives of Pallene is allowing a smooth migration from Lua. Compared to C programs that use the Lua-C API, migrating from Lua to Pallene is significantly easier since the conceptual gap is small. When writing Pallene code, programmers do not need to write an interface layer to receive Lua values and they can manipulate Lua data structures directly. For the benchmark programs of Chapter 6, migrating to Pallene was just a matter of adding the correct type annotations to the Lua programs.

The Lua-C API has abstractions that help programmers by hiding the internals of the Lua interpreter. However, these abstractions introduce a runtime performance overhead. In our benchmarks, we presented several cases where the C programs using the C-API were slower than the Lua interpreter. By manipulating the Lua data structures directly, Pallene does not experience the runtime overhead of the C API. In all of our benchmarks, Pallene was faster than the Lua interpreter.

It is harder to develop a just-in-time (JIT) compiler than an ahead-of-time (AOT) compiler. By introducing a static type system, we were able to implement an AOT compiler for Pallene that generates C. Our compiler is

more portable than LuaJIT and its implementation is more straight-forward than a JIT compiler. Even though our compiler is simpler than LuaJIT, the performance it achieves is comparable to LuaJIT's performance. For most of our benchmarks, the performance of our Pallene compiler was in the same order of magnitude of LuaJIT's performance.

In this dissertation, we introduced two types of records to Pallene: L-records and C-records. Both record types offer similar functionality but they feature two completely different underlying representations. Since C-records are represented by static data structures, Pallene code can manipulate them faster than L-records. On the other hand, manipulating L-records is faster than C-records in Lua given the runtime overhead of metatables.

Programmers should consider the runtime performance of each type of record to decide which one is more suitable for their programs. If a program manipulates a given record in Lua often, then L-records should be used to avoid decreasing the performance. Otherwise, it is preferable to use C-records because they are faster in Pallene. We recommend programmers to benchmark their own programs with the two types of records to find which one is more suitable for each case.

The Pallene compiler achieved favorable performance results by leveraging the optimizations of a mature C compiler. However, there are optimizations that the C compiler can not perform for Pallene. For instance, the C compiler can not perform a bound-check elimination for Pallene arrays as we observed in Chapter 6. One avenue of research that we are currently exploring is implementing these optimizations in the Pallene compiler.

Besides the improvements in performance, there are other improvements to be made in the Pallene language itself. We could easily migrate the benchmark programs to Pallene because they were straight-forward programs that manipulate basic data structures. The Pallene type system is very simplistic and cannot represent more complex Lua programs. Another avenue of research is adding more features to the Pallene type system to facilitate the migration from Lua. In particular, we want to introduce polymorphism and closures to Pallene without impacting Pallene's performance.

Finally, a key feature of system languages is still missing from Pallene. System languages should be able to interface with other system libraries and with the operating system. To support this feature, we want to implement a foreign function interface (FFI) in Pallene. The FFI should allow Pallene to call C functions and manipulate C data structures. By interfacing with C, there will not be anything else preventing Pallene to be used as a system language.

Bibliography

[ANCONA et al., 2007] ANCONA, D.; ANCONA, M.; CUNI, A. ; MATSAKIS, N. D.. **RPython: A step towards reconciling dynamically and statically typed OO languages**. In: PROCEEDINGS OF THE 2007 SYMPOSIUM ON DYNAMIC LANGUAGES, DLS '07, p. 53–64. ACM, 2007.

[ANTONOV, 2013] ANTONOV, P.; OTHERS. **V8 optimization killers**, 2013. `https://github.com/petkaantonov/bluebird/wiki/Optimization-killers` [Retrieved April 2019].

[AYCOCK, 2003] AYCOCK, J.. **A brief history of just-in-time**. ACM Computing Surveys, 35(2):97–113, June 2003.

[BEHNEL et al., 2010] BEHNEL, S.; BRADSHAW, R.; CITRO, C.; DALCIN, L.; SELJEBOTN, D. ; SMITH, K.. **Cython: The best of both worlds**. Computing in Science Engineering, 13(2):31–39, Mar. 2011.

[BIERMAN et al., 2014] BIERMAN, G.; ABADI, M. ; TORGERSEN, M.. **Understanding TypeScript**. In: Jones, R., editor, 28TH EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP '14, p. 257–281. Springer Berlin Heidelberg, 2014.

[CARDIFF, 2019] CARDIFF, B. J.. **Crystal 0.28.0 released**, 2019. `https://crystal-lang.org/2019/04/17/crystal-0.28.0-released.html` [Retrieved April 2019].

[DE MELO, 2010] DE MELO, A. C.. **The new linux'perf'tools**. In: SLIDES FROM LINUX KONGRESS, volumen 18, 2010.

[DEUTSCH; SCHIFFMAN, 1984] DEUTSCH, L. P.; SCHIFFMAN, A. M.. **Efficient implementation of the Smalltalk-80 system**. In: PROCEEDINGS OF THE 11TH ACM SIGACT-SIGPLAN SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL '84, p. 297–302, New York, NY, USA, 1984. ACM.

[DEVITO, 2014] DEVITO, Z.. **Terra: Simplifying High-Performance Programming Using Multi-Stage Programming**. PhD thesis, Stanford University, Dec. 2014.

[GAL et al., 2006] GAL, A.; PROBST, C. W. ; FRANZ, M.. **HotpathVM: An effective JIT compiler for resource-constrained devices**. In: PROCEEDINGS OF THE 2ND INTERNATIONAL CONFERENCE ON VIRTUAL EXECUTION ENVIRONMENTS, VEE '06, p. 144–153, New York, NY, USA, 2006.

[GIRALDEZ, 2017a] GUERRA GIRALDEZ, J.. **The rocky road to MCode**. Talk at Lua Moscow conference, 2017, 2017. `https://www.youtube.com/watch?v=sz2CuDpltmM` [Retrieved April 2019].

[GIRALDEZ, 2017b] GUERRA GIRALDEZ, J.. **LuaJIT hacking: Getting next() out of the NYI list**. CloudFare Blog, Feb. 2017. `https://blog.cloudflare.com/luajit-hacking-getting-next-out-of-the-nyi-list/` [Retrieved April 2019].

[GRAHAM, 1995] GRAHAM, P.. **ANSI Common LISP**. Prentice-Hall, 1995.

[GUALANDI; IERUSALIMSCHY, 2018] GUALANDI, H. M.; IERUSALIMSCHY, R.. **Pallene: a statically typed companion language for Lua**. In: PROCEEDINGS OF THE XXII BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, SBLP 2018, SAO CARLOS, BRAZIL, SEPTEMBER 20-21, 2018, p. 19–26, 2018.

[IERUSALIMSCHY, 2009] IERUSALIMSCHY, R.. **A text pattern-matching tool based on Parsing Expression Grammars**. Software: Practice and Experience, 39(3):221–258, 2009.

[IERUSALIMSCHY et al., 2005] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; CELES, W.. **The implementation of lua 5.0**. Journal Universal Computer Science, 11(7):1159–1176, July 2005.

[LLVM, 2019a] LLVM. **The LLVM compiler infrastructure**, 2019. `https://llvm.org/` [Retrieved April 2019].

[LLVM, 2019b] LLVM. **Clang vs other open source compilers**, 2019. `https://clang.llvm.org/comparison.html` [Retrieved April 2019].

[MAIDL, 2015] MAIDL, A. M.. **Typed Lua: An Optional Type System for Lua**. PhD thesis, PUC-Rio, Apr. 2015.

[MARR et al., 2016] MARR, S.; DALOZE, B. ; MÖSSENBÖCK, H.. **Cross-language compiler benchmarking: Are we fast yet?** In: PROCEEDINGS OF THE 12TH SYMPOSIUM ON DYNAMIC LANGUAGES, DLS 2016, p. 120–131, 2016.

[MCILROY, 2016] MCILROY, R.. **Firing up the Ignition interpreter**, 2016. `https://v8.dev/blog/ignition-interpreter` [Retrieved April 2019].

[MEDEIROS; MASCARENHAS, 2018] MEDEIROS, S.; MASCARENHAS, F.. **Syntax error recovery in Parsing Expression Grammars**. In: PROCEEDINGS OF THE 33RD ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, SAC '18, p. 1195–1202, New York, NY, USA, 2018. ACM.

[OUSTERHOUT, 1998] OUSTERHOUT, J. K.. **Scripting: Higher-level programming for the 21st century**. Computer, 31(3):23–30, Mar. 1998.

[PALL, 2005a] PALL, M.. **LuaJIT, a just-in-time compiler for Lua**, 2005. `http://luajit.org/luajit.html` [Retrieved April 2019].

[PALL, 2005b] PALL, M.. **LuaJIT performance comparison**, 2005. `http://luajit.org/performance.html` [Retrieved April 2019].

[PALL, 2009] PALL, M.. **LuaJIT 2.0 intellectual property disclosure and research opportunities**. lua-l mailing list, nov 2009. `http://lua-users.org/lists/lua-l/2009-11/msg00089.html` [Retrieved April 2019].

[PALL, 2014] PALL, M.. **Not yet implemented operations in LuaJIT**. LuaJIT documentation Wiki, 2014. `http://wiki.luajit.org/NYI` [Retrieved April 2019].

[PIZLO, 2016] PIZLO, F.. **Introducing the B3 JIT compiler**, 2016. `https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/` [Retrieved April 2019].

[RIGO; PEDRONI, 2006] RIGO, A.; PEDRONI, S.. **PyPy's approach to virtual machine construction**. In: COMPANION TO THE 21ST ACM SIGPLAN SYMPOSIUM ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA '06, p. 944–953, New York, NY, USA, 2006. ACM.

[SIEK; TAHA, 2006] SIEK, J. G.; TAHA, W.. **Gradual typing for functional languages**. Scheme and Functional Programming Workshop, 6:81–92, 2006.

[SIEK et al., 2015] SIEK, J. G.; VITOUSEK, M. M.; CIMINI, M. ; BOYLAND, J. T.. **Refined criteria for gradual typing**. In: 1ST SUMMIT ON ADVANCES IN PROGRAMMING LANGUAGES, SNAPL '15, p. 274–293, Asilomar, California, USA, May 2015.

[TAKIKAWA et al., 2016] TAKIKAWA, A.; FELTEY, D.; GREENMAN, B.; NEW, M. S.; VITEK, J. ; FELLEISEN, M.. **Is sound gradual typing dead?** In: PROCEEDINGS OF THE 43RD ANNUAL ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, POPL '16, p. 456–468, 2016.

[TITAN, 2018] MAIDL, A. M.; MASCARENHAS, F.; LIGNEUL, G.; MUHAMMAD, H. ; GUALANDI, H. M.. Source code repository for the Titan programming language, 2018. `https://github.com/titan-lang/titan` [Retrieved April 2019].

[TOBIN-HOCHSTADT; FELLEISEN, 2006] TOBIN-HOCHSTADT, S.; FELLEISEN, M.. **Interlanguage migration: From scripts to programs**. In: COMPANION TO THE 21ST ACM SIGPLAN SYMPOSIUM ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA '06, p. 964–974, New York, NY, USA, 2006. ACM.

[WÜRTHINGER et at., 2013] WÜRTHINGER, T.; WIMMER, C.; WÖSS, A.; STADLER, L.; DUBOSCQ, G.; HUMER, C.; RICHARDS, G.; SIMON, D. ; WOLCZKO, M.. **One VM to rule them all**. In: PROCEEDINGS OF THE 2013 ACM INTERNATIONAL SYMPOSIUM ON NEW IDEAS, NEW PARADIGMS, AND REFLECTIONS ON PROGRAMMING & SOFTWARE, Onward! 2013, p. 187–204, 2013.