Pallene: A companion language for Lua

Hugo Musso Gualandi, Roberto Ierusalimschy

Abstract

The simplicity and flexibility of dynamic languages make them popular for prototyping and scripting, but the lack of compile-time type information makes it challenging to generate efficient executable code.

Inspired by ideas from scripting, just-in-time compilers, and optional type systems, we have developed Pallene, a typed companion language to the Lua scripting language, intended for writing lower-level libraries and extension modules. It plays the role of a system language in the scripting paradigm, but one that has been explicitly designed to interoperate with Lua. Pallene is designed to be efficient, to interoperate seamlessly with Lua (even sharing its runtime), and to be familiar to Lua programmers. It should also be simple, easy to understand, and easy to implement, so it can be as portable as maintainable as Lua itself.

In this paper, we discuss the rationale for Pallene's design and present a description of its syntax, type system, and semantics. We also compare the performance of Pallene extension modules against pure Lua (both interpreted and JIT-compiled), against C extension modules (operating with Lua data structures), and also against programs fully written in C, which provide a familiar baseline. The results corroborate our hypothesis that Pallene has the potential to be a low-level counterpart to Lua. The performance of Lua extension modules written in Pallene can be better than that of equivalent modules written in C and it is competitive with the performance from a JIT compiler, despite the vastly simpler implementation.

This is a revised and extended version of a SBLP 2018 paper with a similar title [1].

Keywords: Dynamic Languages, Scripting Languages, Compilers, Just-in-time compilers, Type Systems, Gradual Typing

1. Introduction

Dynamic languages are widely used for prototyping and scripting, due to their simplicity and flexibility. However, the lack of compile-time type information poses a challenge for the generation of efficient executable code. There are at least three approaches to improve the performance of software which uses dynamically typed languages: scripting, just-in-time compilation, and optional type systems.

The scripting approach advocates combining two separate languages to write a program: a low-level *system language* for the parts of the program that need good performance and that interact with the underlying operating system, and a high-level *scripting language* for the parts that need flexibility and ease of use [2]. The main advantage of this approach is that the programmer can choose the programming language most suited for each particular task. The main disadvantages come from the large differences between the languages. It can be hard to rewrite a piece of code from one language to the other and also the conversion between distinct data representations introduces run-time overhead, which can cancel out some or all the benefit of rewriting the code in the first place.

Just-in-time (JIT) compilers translate high-level dynamically-typed programs into fast machine code [3]. They are the state of the art in optimizing dynamic languages but are also complex and challenging to implement. The performance gains are also not guaranteed: in some cases programmers might need to rewrite their programs into a form that is more amenable to optimization by the JIT compiler [4, 5]. Additionally, JIT compilation may not be an option in certain mobile or embedded platforms due to portability issues or requirements that executable code must reside in read-only memory (either due to limited RAM or by platform mandate).

As the name suggests, optional type systems allow programmers to partially add types to programs. These systems combine the static typing and dynamic typing disciplines in a single language. From static types they seek better compile-time error checking, machine-checked lightweight documentation, and run-time performance. From dynamic typing they seek flexibility and ease of use. One of the selling points of optional type systems is that they promise a smooth transition from small dynamically typed scripts to larger statically typed applications [6, 7, 8]. For this to be true these type systems attempt to accommodate common idioms of the dynamic language, which introduces additional complexities and implementation challenges. It is hard to design a type system that is at the same time simple, correct, and fast.

Both the scripting approach and optional types assume that programmers need to restrict the dynamism of their code when they seek better performance. Although in theory JIT compilers do not require this, in practice programmers also need to restrict themselves to achieve maximum performance. Realizing how these self-imposed restrictions result in the creation of vaguely defined language subsets, and how restricting dynamism seems unavoidable, we asked ourselves: what if we accepted the restrictions and defined a new programming language based on them? By focusing on this "well behaved" subset and making it explicit, instead of trying to optimize or type a dynamic language in its full generality, we would be able to drastically simplify the type system and the compiler.

To study this question, we have developed Pallene, a low-level counterpart to Lua. Pallene is intended to act as a system language counterpart to Lua scripting, but with better interoperability than existing static languages like C or Java. For efficiency, simplicity and portability, we designed Pallene to be compatible with standard ahead-of-time compilation techniques. To minimize the run-time mismatch, Pallene uses Lua's data representation and garbage collector. To minimize the conceptual mismatch, Pallene is syntactically and semantically similar to Lua.

In the next section of this paper, we overview the main approaches currently used to tackle the performance problems of dynamic languages, while keeping the design simple, maintainable and portable. In Section 3, we describe how we designed Pallene, aiming to combine desirable properties from those approaches. In Section 4 we provide a deeper explanation of Pallene's syntax, type system, and semantics. In Section 5, we discuss how our goals for Pallene affected its implementation. In Section 6, we evaluate the performance of our prototype implementation of Pallene on a set of micro benchmarks. We compare it against pure Lua (both the reference interpreter and LuaJIT [9]), against C extension modules (which operate on Lua data structures through the Lua–C API [10]), and against pure C code, which acts as a performance baseline. We also measure the overhead of run-time tag checking in Pallene. In Section 7 we compare Pallene with related research in type systems and optimization for dynamic languages. Finally, in Section 8 we state our conclusions and discuss avenues for future work.

A previous version of this paper appeared in SBLP 2018 [1]. The main additions to this version are a more detailed description of the language and an in-depth performance analysis of the cost of run-time tag checking.

2. Optimizing Scripting Languages

In this section we discuss the existing approaches to optimizing dynamic languages that we mentioned previously, namely scripting, JIT compilers, and optional types.

2.1. Scripting

The archetipical application for a scripting language is a multi-language system, where the high-level scripting language is used in combination with a low-level statically-typed *system language* [2]. In this architecture, the system language tends to be used for performance sensitive tasks, while the more flexible scripting language is more suited for configuration, orchestration, and situations where ease-of-development is at a premium.

Lua has been designed from the start with scripting in mind [11] and many applications that use Lua follow this approach. For instance, a computer game like Grim Fandango has a basic engine, written in C++, that performs physical simulations, graphics rendering, and other machine intensive tasks. The game designers, who are not professional programmers, wrote all the game logic in Lua [12].

From a performance point of view, the scripting architecture is pragmatic and predictable. Each language can be used where it is more adequate and the software architect can be relatively confident that the parts written in the system language will have good performance. Moreover, scripting languages are often implemented by small interpreters, which facilitates maintainability and portability. Lua's reference interpreter, for instance, has successfully been ported to a wide variety of operating systems and hardware architectures, from large web-servers to micro-controllers.

The fundamental downside of a multi-language architecture is the conceptual mismatch between the languages. Rewriting a module to use a different language is difficult. A common piece of advice when a Lua programmer seeks better performance is to "rewrite it in C", but this is easier said than done. In practice, programmers only follow this advice when the code is mainly about low-level operations that are easy to express in C, such as doing arithmetic and calling external libraries. Another obstacle to this suggestion is that it is hard to estimate in advance both the costs of rewriting the code and the performance benefits to be achieved by the change. Often, the gain in performance is not what one would expect. As we show in Section 6, the overhead of the language interface can sometimes cancel out the inherent performance advantage of the system language.

2.2. Just-in-time Compilers

Just-in-time (JIT) compilers are the state of the art in dynamic language optimization. A JIT compiler initially executes the program without any optimization, observes its behavior, and then, based on this, generates highly specialized and optimized executable code. For example, if it observes that some code is always operating on values of type *double*, the compiler will optimistically produce a version of this code that is specialized for that type. It will also insert tests (guards) that revert back to the generic version of the code in case one of the values is not of type *double* as expected.

JIT compilers are broadly classified as either method-based or trace-based [13], according to their main unit of compilation. In method-based JITs, the unit of compilation is one function or subroutine. In trace-based JITs, the unit of compilation is a linear trace of the program execution, which may cross over function boundaries. Trace compilation allows for a more embeddable implementation and is better at compiling across abstraction boundaries. However, it has trouble optimizing programs which contain unpredictable branch statements. For this reason, most JIT compilers now tend to use the method-based approach, with the notable exceptions of LuaJIT [9] and RPython-based JITs [14, 15].

Implementing a JIT compiler is a significant undertaking. Firstly, the system must combine both an interpreter and a compiler. In addition to normal execution, the interpreter also has to profile the execution of the code. Furthemore, the implementation must be able to switch between generic and optimized versions of the code, while the code itself is running. This on-stack replacement can be particularly challenging to implement. Furthermore, the overall performance is heavily influenced by compilation and profiling overheads in the warmup phase, so there is a powerful incentive to specialize the implementation to the target platform and to the language being compiled, trading away portability and reusability.

There is ongoing research in the area of JIT development frameworks to simplify JIT development—such as the *metatracing* of the previously mentioned RPython framework and the partial evaluation strategy of Truffle [16]—but unfortunately these tools currently do not support the wide variety of environments where Lua runs. For example, neither RPython nor Truffle can run on the Android or iOS operating systems. The large size of their generated binaries also precludes their use in low-memory environments. For reference, Lua can be used in devices with as low as 300KB of RAM.

From the point of view of the software developer, the most attractive aspect of JIT compilers is that they promise increased performance without needing to modify the original dynamically typed program. However, these gains are not always easy to achieve, because the effectiveness of JIT compiler optimizations can be inconsistent. Certain code patterns, known as *optimization killers*, may cause the whole section around them to be de-optimized, resulting in a dramatic performance impact. Therefore, programmers must accept that good performance depends on adapting the code to avoid these optimization killers, by following advice from the official documentation or from folk knowledge [17, 5].

Since there may be an order of magnitude difference in performance between JIT optimized and unoptimized code, there is a powerful incentive to write programs in a style that is more amenable to optimization.

```
-- Slower
local function mytan(x)
  return math.sin(x) / math.cos(x)
end

-- Faster
local sin, cos = math.sin, math.cos
local function mytan(x)
  return sin(x) / cos(x)
end
```

```
-- Faster (!)
local function hello()
    C.printf("Hello, world!")
end

-- Slower (!)
local printf = C.printf
local function hello()
    printf("Hello, world!")
end
```

Figure 1: LuaJIT encourages programmers to cache imported Lua functions in local variables. However, the recommendation for C functions called through the foreign function interface is surprisingly the other way around.

```
function increment_numbers(text)
  return (text:gsub("[0-9]+", function(s)
    return tostring(tonumber(s) + 1)
  end))
end
```

Figure 2: This function cannot be optimized by LuaJIT because it calls the `gsub` method and because it uses an anonymous callback function. These optimization killers negatively affect the performance not only of the `increment_numbers` function but also of any trace that calls it.

This often encourages unintuitive programming idioms. For example, LuaJIT's documentation recommends caching Lua functions from other modules in a local variable [18], as is shown in Figure 1. However, for C functions accessed via the foreign function interface the rule is the other way around.

Another example from LuaJIT is the function in Figure 2, which runs into several LuaJIT optimization killers (which the LuaJIT documentation calls "Not Yet Implemented" features). As of LuaJIT 2.1, traces that call string pattern-matching methods such as `gsub` are not compiled into machine code by the JIT. That is also the case for traces that define anonymous functions, even if the anonymous function does not close over any outer variables.

These patterns are not unique to LuaJIT, as every JIT has its own set of quirks. For example, until 2016 the V8 JavaScript implementation could not optimize functions containing a try-catch statement [5]. Encouraging different coding styles is not the only way that JIT behavior affects the software development process, either. Programmers resort to specialized debugging tools to discover which optimization killer is causing the performance problems [19]. This may require reasoning at a low level of abstraction, involving the intermediate representation of the JIT compiler or its generated machine code [20].

Another aspect of JIT compilers is that before they can start optimizing, they must run the program for

many iterations, in interpreter mode, collecting run-time information. During this initial warm-up period the JIT will run only as fast or even slower than a non-JIT implementation. Sometimes the warm-up time can even be erratic, as was observed by Barrett et al [21].

2.3. Optional Type Systems

Static types can serve several purposes. They are useful for error detection, can act as a form of lightweight documentation, and help the compiler generate efficient code. As a result, there are many projects aiming to bring these benefits to dynamic languages, using optional type systems to combine the benefits of static and dynamic typing [22].

A recurring idea to help the compiler produce more efficient code is to allow the programmer to add type annotations to the program. Compared with a more traditional scripting approach, optional typing promises a single language instead of two different ones, making it easier for the static and dynamic parts of the program to interact with each other. The pros and cons of these optional type system approaches vary from case to case, since each type system is designed for a different purpose. For example, the optional type annotations of Common LISP allow the compiler to generate extremely efficient code, but without any safeguards [23]. Meanwhile, in Typed Lua the type annotations are only used for compile-time error detection, with no effect on runtime performance.

A research area deserving special attention is Gradual Typing [24], which aims to provide a solid theoretical framework for designing type systems that integrate static and dynamic typing in a single language. However, gradual type systems still face difficulties when it comes to run-time performance. On the one hand, systems that fully check types as they cross the boundary between the static and dynamic parts of the code are often plagued with a high verification overhead cost [25]. On the other hand, type systems that perform type erasure at run-time usually give up on the opportunity to optimize the static parts of the program.

To facilitate a smooth transition from untyped programs to typed ones, optional type systems are typically designed to accommodate common idioms from the dynamically typed language. However, this additional flexibility usually leads to a more complex type system, which is more difficult to use and, more importantly to us, to optimize. For example, in Lua out of bound accesses result in `nil` and removing an element from a list is done by assigning `nil` to its position. Both cases require Typed Lua's type system to accept `nil` as a valid element of all Lua lists [26]. Array elements in Typed Lua are always nullable.

3. The Design of Pallene

Our inspiration for Pallene came from the problem of high-performance Lua programs. Over the years we have seen an increase in the number of applications that use Lua in performance-sensitive code paths, often in combination with LuaJIT, an alternative just-in-time compiler for Lua [9]. For example, the OpenResty framework embeds LuaJIT inside the nginx web server, and enables the development of high-performance web applications written mostly in Lua [27]. However, problems related to the high complexity and the difficulty of maintaining a just-in-time compiler such as LuaJIT have emerged. LuaJIT has diverged from the reference PUC-Lua implementation, as certain language features introduced in PUC-Lua would not get added to LuaJIT. Additionally, we observed that the complex performance landscape of the JIT compiler led programmers to adopt unusual programming idioms that eschewed language features that were deemed "too slow".

With the insight that programmers are willing to restrict their use of dynamic language features when performance matters, we have decided to explore a return to the traditional scripting architecture through Pallene, a system programming language that we have designed specifically to complement Lua. Since Pallene has static types, it can obtain good performance with an ahead-of-time compiler, avoiding the complexities of just-in-time compilation; and since it is designed specifically to be used with Lua, we hope Pallene will be attractive in situations where using C would be too cumbersome.

Overall, our goals for Pallene are that it should be safe, predictably efficient, seamlessly interoperable with Lua, and easy for Lua programmers to learn. Furthermore, it should have a simple and maintainable

```
function sum(xs: {float}): float
    local s: float = 0.0
    for i = 1, #xs do
        s = s + xs[i]
    end
    return s
end
```

Figure 3: A Pallene function for adding up the numbers in a Lua array. It is also a valid Lua program, except for the type annotations, which start with a colon.

implementation that is as portable as Lua itself. To achieve these goals, we designed Pallene as an ahead-of-time-compiled, typed subset of Lua, which can directly manipulate Lua data structures, which shares Lua's runtime and garbage collector, and which uses run-time tag checks to enforce type safety. In this section, we describe how these design choices accomplish our goals, and how all of them play a part in enabling good performance.

3.1. Pallene is a subset of Lua

Inspired by optional and gradual typing, Pallene is very close to a typed subset of Lua. For example, the Pallene program for computing the sum of an array of floating-point numbers that is shown in Figure 3 is also a valid Lua program, other than the type annotations. Furthermore, the behavior of Pallene programs is the same as that of Lua, except that Pallene may raise a run-time type error if it receives a value from Lua that does not have the expected type. This syntactical and semantical similarity enables the seamless interoperability between Lua and Pallene, and also makes Pallene easier for Lua programmers to learn.

Incidentally, this similarity also means that when it is desired to speed up a Lua module it should be easier to rewrite it in Pallene than it would be to rewrite it in a wholly different system language like C. That said, this transition might not be a simple matter of inserting type annotations, since Pallene's type system is designed for performance first and is not flexible enough for many common Lua idioms. This sets Pallene apart from gradual type systems such as Typed Lua [26]. Typed Lua's type system is designed to accommodate a wide variety of Lua programs but this flexibility also means that it is not able to offer better performance than plain Lua.

3.2. Pallene is typed, and uses tag checking to enforce type safety

One of the major sources of performance for Pallene when compared to Lua is that Pallene programs are annotated with types, and the compiler uses this type information to generate efficient, type-specialized code. However, there are many situations where Pallene will manipulate values that originate from Lua, in which case it must introduce a run-time type check to ensure that the value has the declared type. These run-time type checks inspect the type tags that are already present in dynamically-typed Lua values. As we will show in section 6, the overhead of this tag checking is modest and is more than compensated by the substantial performance gains from type specialization.

3.3. Pallene shares the runtime and garbage collector with Lua

Pallene offers first-class support for manipulating Lua objects and data structures that adhere to its type system. Not only can Pallene programs use objects such as Lua arrays, but Pallene modules also share the same runtime and garbage collector as Lua. This is in contrast to other system languages, which are only able to indirectly manipulate Lua values through the Lua–C API [10]. As we will show in Section 6, bypassing the traditional Lua–C API in this manner can significantly improve performance.

3.4. Pallene is ahead-of-time compiled

The presence of static type information in Pallene programs allows it to be implemented with a straightforward ahead-of-time compiler, which is faster than an interpreter, and simpler than a JIT.

Guided by the type annotations, Pallene's compiler can produce efficient, type-specialized code that is much faster than the Lua equivalent. This type specialization is similar to the one that JIT compilers perform following run-time guards, but much simpler to implement. Since a failed tag check in Pallene is a run-time type error, Pallene does not need to worry about deoptimizing speculatively-optimized code. There is also no need for a separate interpreter or a warmup profiling phase.

The Pallene compiler generates C source code and uses a conventional C compiler as a backend. This is simple to implement and is also useful for portability. The generated C code for a Pallene module is standard C code that uses the same headers and system calls that the reference Lua does. This means that Pallene code should be portable to any platforms where Lua itself can run. Ensuring this portability would have been more difficult if we would have needed to implement our own compiler backend, or if we had pursued a JIT-compilation strategy.

4. The Pallene Language

This section contains a more detailed description of Pallene's syntax, type system, and semantics. As Pallene is intentionally not attempting to innovate through novel type system or language semantics, the point we want to emphasize is that the main difference between Pallene and Lua—and what sets them apart as system and scripting languages, respectively—is the presence of a type system which effectively restricts what language features and idioms can be used.

In the interest of brevity, we limit this description of Pallene to the parts that concern primitive operations, arrays, and records, which are sufficient to demonstrate the behavior of Pallene's run-time type checks, to highlight the cooperation with the Lua runtime system, and to perform initial performance experiments. To simplify the presentation, we assume that that variable declarations always carry a type annotation. The full version of Pallene uses a form of bidirectional type inference to infer the types of most local variables.

4.1. Syntax

A Pallene module consists of a sequence of function definitions, which are to be exported to Lua. The language syntax is summarized in Figure 4. It is essentially the same as Lua's syntax except for the requirement that variables and functions must be typed, and that language features incompatible with the type system cannot be expressed. (We will list these features when describing the type system.) For brevity, we omit some primitive operators (e.g. bitwise operators), as well as syntactical niceties like optional semicolons.

The syntax should be familiar to those who have previously programmed in a statically typed imperative language, and immediately recognizable by Lua programmers. A prefixed `#` is the length operator and an infixed `..` is string concatenation. For clarity, we represent the empty statement as `nop`.

Curly braces are used both for arrays and records, following the notation used by Lua's table constructors. Empty record types are not allowed, to avoid an ambiguity in the meaning of `{}`.

4.2. Type System

Pallene's type system is a conventional imperative-language type system, extended with a dynamic type called `any`. Figure 5 describes the typing rules for expressions and Figure 6 does the same for statements. Figure 7 lists the types for primitive operators.

Variables of type `any` can hold any Lua value. Under the hood they are just tagged Lua values, which is the internal representation of values in the Lua runtime. Upcasting to `any` is always allowed, as is downcasting from `any` to another type. Downcasts are checked at run time. This kind of variable must be present in Pallene to implement values that come from Lua, before they are tag checked. Exposing this feature to the programmer is a simple way to improve interoperation with Lua. It also adds flexibility to

τ	:=		TYPES
		`nil` | `boolean` | `integer` | `float` | `string`	*primitive types*
	|	`{`τ`}`	*array type*
	|	`{` l_i `:` $\tau_i^{\,i\in 1..n}$ `}`	*record type*
	|	$\tau_i^{\,i\in 1..n} \to \tau$	*function type*
	|	`any`	*dynamic type*
e	:=		EXPRESSIONS
	|	`nil`	*nil literal*
	|	`true` | `false`	*boolean literals*
	|	*int*	*integer literals*
	|	*flt*	*floating-point literals*
	|	*str*	*string literals*
	|	`{` $e_i^{\,i\in 1..n}$ `}`	*array constructor*
	|	`{` $l_i = e_i^{\,i\in 1..n}$ `}`	*record constructor*
	|	x	*variable*
	|	op_1 e	*primitive unary operations*
	|	e op_2 e	*primitive binary operations*
	|	e`[`e`]`	*array-read*
	|	e`.`l	*record-read*
	|	e`(`$e^{i\in 1..n}$`)`	*function call*
	|	e `as` τ	*type cast*
op_1	:=	`-` | `not` | `#`	UNARY OPERATORS
op_2	:=	`+` | `-` | `*` | `/` | `..` | `and` | `or` | `==` | `<` | `>`	BINARY OPERATORS
stmt	:=		STATEMENTS
		`nop`	*empty statement*
	|	*stmt* `;` *stmt*	*sequence*
	|	`while` e `do` *stmt* `end`	*while loop*
	|	`for` x `=` e`,` e `do` *stmt* `end`	*for loop*
	|	`if` e `then` *stmt* `else` *stmt* `end`	*conditional*
	|	`local` x`:`τ `=` e`;` *stmt*	*variable declaration*
	|	x `=` e	*assignment*
	|	e`[`e`]` `=` e	*array-write*
	|	e`.`l `=` e	*record-write*
	|	e`(`$e^{i\in 1..n}$`)`	*function call*
	|	`return` e	*function return*
fun	:=	`function` f`(`x_i`:`$\tau_i^{\,i\in 1..n}$`)``:`$\tau$ *stmt* `end`	FUNCTION DEFINITION
prog	:=	$fun_i^{\,i\in 1..n}$	PALLENE MODULE

Figure 4: Abstract syntax for Pallene. It is a subset of Lua, but with added type annotations, and the restriction that the toplevel must consist of a sequence of function definitions.

$$\Gamma \vdash \texttt{nil} : \texttt{nil} \qquad \Gamma \vdash \texttt{true} : \texttt{boolean} \qquad \Gamma \vdash \texttt{false} : \texttt{boolean}$$

$$\Gamma \vdash \mathit{int} : \texttt{integer} \qquad \Gamma \vdash \mathit{flt} : \texttt{float} \qquad \Gamma \vdash \mathit{str} : \texttt{string}$$

$$\frac{\Gamma \vdash e_i : \tau^{\,i \in 1..n}}{\Gamma \vdash \texttt{\{}\ e_i^{\,i \in 1..n}\ \texttt{\}} : \texttt{\{}\tau\texttt{\}}} \qquad \frac{\Gamma \vdash e_i : \tau_i^{\,i \in 1..n}}{\Gamma \vdash \texttt{\{}\ l_i = e_i^{\,i \in 1..n}\ \texttt{\}} : \texttt{\{}\ l_i : \tau_i^{\,i \in 1..n}\ \texttt{\}}} \qquad \Gamma \vdash (e\ \texttt{as}\ \tau) : \tau \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \text{optype1}(op_1, \tau_1) = \tau}{\Gamma \vdash (op_1\ e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \text{optype2}(op_1, \tau_1, \tau_2) = \tau}{\Gamma \vdash (e_1\ op_2\ e) : \tau}$$

$$\frac{\Gamma \vdash a : \texttt{\{}\tau\texttt{\}} \qquad \Gamma \vdash i : \texttt{integer}}{\Gamma \vdash a\texttt{[}i\texttt{]} : \tau} \qquad \frac{\Gamma \vdash e : \texttt{\{}\ l_i : \tau_i^{\,i \in 1..n}\ \texttt{\}}}{\Gamma \vdash e.l_j : \tau_j} \qquad \frac{\Gamma \vdash f : \tau_i^{\,i \in 1..n} \to \tau \qquad \Gamma \vdash e_i : \tau_i^{\,i \in 1..n}}{\Gamma \vdash f(e_i^{\,i \in 1..n}) : \tau}$$

Figure 5: Pallene typing rules for expressions. $\Gamma \vdash e : \tau$ means that expression e has type τ under the environment Γ

$$\Gamma \vdash \texttt{nop} \qquad \frac{\Gamma \vdash \mathit{stmt}_1 \qquad \Gamma \vdash \mathit{stmt}_2}{\Gamma \vdash \mathit{stmt}_1\ \texttt{;}\ \mathit{stmt}_2}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash \mathit{stmt}}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ \mathit{stmt}\ \texttt{end}} \qquad \frac{\Gamma \vdash e_1 : \texttt{integer} \qquad \Gamma \vdash e_2 : \texttt{integer} \qquad \Gamma, x : \texttt{integer} \vdash \mathit{stmt}}{\Gamma \vdash \texttt{for}\ x = e_1, e_2\ \texttt{do}\ \mathit{stmt}\ \texttt{end}}$$

$$\frac{\Gamma \vdash e : \texttt{boolean} \qquad \Gamma \vdash \mathit{stmt}_1 \qquad \Gamma \vdash \mathit{stmt}_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{end}} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma, x : \tau \vdash \mathit{stmt}}{\Gamma \vdash \texttt{local}\ x : \tau\ \texttt{=}\ e\ \texttt{;}\ \mathit{stmt}} \qquad \frac{(x : \tau) \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x\ \texttt{=}\ e}$$

$$\frac{\Gamma \vdash a\texttt{[}i\texttt{]} : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash a\texttt{[}i\texttt{]}\ \texttt{=}\ e} \qquad \frac{\Gamma \vdash r.l : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash r.l = e} \qquad \frac{\Gamma \vdash f(e_i^{\,i \in 1..n}) : \texttt{nil}}{\Gamma \vdash f(e_i^{\,i \in 1..n})}$$

$$\frac{\Gamma\ ,\ (f : \tau_i^{\,i \in 1..n} \to \tau)\ ,\ (x_i : \tau_i)^{i \in 1..n}\ ,\ (\$ret : \tau) \vdash \mathit{stmt}}{\Gamma \quad \vdash \quad \begin{array}{l}\texttt{function}\ \ f\texttt{(}x_i\texttt{:}\tau_i^{\,i \in 1..n}\texttt{)}\texttt{:}\tau \\ \qquad \mathit{stmt} \\ \texttt{end}\end{array}} \qquad \frac{\Gamma \vdash e : \tau \qquad (\$ret : \tau) \in \Gamma}{\Gamma \vdash \texttt{return}\ e}$$

Figure 6: Pallene typing rules for statements and functions. $\Gamma \vdash \mathit{stmt}$ means that the statement *stmt* is well-typed under the environment Γ. The special entry $\$ret$ in the environment corresponds to the return type for the surrounding function. $\$ret$ is not a valid Pallene identifier, and only appears in these typing rules.

optype1

op_1	operand	result
`-`	`integer`	`integer`
`-`	`float`	`float`
`not`	`boolean`	`boolean`
`#`	$\{\tau\}$	`integer`
`#`	`string`	`integer`

optype2

op_2	first operand	second operand	result
`+ - * /`	`integer`	`integer`	`integer`
`+ - * /`	`integer`	`float`	`float`
`+ - * /`	`float`	`integer`	`float`
`+ - * /`	`float`	`float`	`float`
`..`	`string`	`string`	`string`
`and or`	`boolean`	`boolean`	`boolean`
`< == >`	`boolean`	`boolean`	`boolean`
`< == >`	`integer`	`integer`	`boolean`
`< == >`	`integer`	`float`	`boolean`
`< == >`	`float`	`integer`	`boolean`
`< == >`	`float`	`float`	`boolean`
`< == >`	`string`	`string`	`boolean`

Figure 7: Typing relations for primitive operators. Arithmetic operators work on either integers or floating point numbers. The logic operators like `and`, `or` and `not` operate on booleans. Comparison operators work on non-nil primitive types.

```
function greet(name)
    name = name or "world"
    return "Hello " .. name
end

print(greet("friend"))
print(greet())
```

Figure 8: This Lua program cannot be directly converted to Pallene by just adding type annotations. Pallene functions must have fixed arity, and the typed version of the `or` operator would reject the string operands.

the type system, since this dynamic typing can be used to emulate unions and parametric polymorphism. (This is similar to the role that the Object type played in Java before version 5, which introduced generics.)

Generally speaking, the Pallene type system isn't innovative. The interesting discussion concerns which Lua features and idioms the type system can and cannot express, as this is what fundamentally positions Pallene as a system language instead of as a scripting one. For example, Pallene does not support the common default-parameter idiom exemplified in Figure 8, as Pallene functions have a fixed arity and its `or` operator only accepts boolean operands. Similarly, Pallene forbids functions with multiple return values and it rejects dynamically typed operands to primitive operators. For example, an `any` must be explicitly downcasted to either `integer` or `float` before it can be passed to the `+` operator.

Another, more substantial restriction imposed by Pallene concerns tables. The Lua table is a versatile datatype that maps arbitrary keys to arbitrary values, and is the building block for most data structures in Lua. Two particular use-cases are arrays, which in Lua are just tables with integer keys, and records, which are tables with string keys that are known at compile time. Pallene supports these use cases with its array and record types. However, Pallene is not able to describe other uses of tables, such as associative arrays.

Finally, Pallene does not allow operators to be overloaded by the programmer, a feature present in Lua which is provided by its metatable mechanism.

4.3. Semantics

To be compatible with Lua and to be familiar to Lua programmers, we have kept Pallene's semantics as close as possible to a strict subset of Lua. As a guiding principle, we try to follow the Gradual Guarantee

```
-- Pallene Code:
function add(x: float, y:float): float
    return x + y
end

-- Lua Code:
local big = 1 << 54
print(add(big, 1) == big)
```

Figure 9: An example illustrating why Pallene raises an error instead of automatically coercing integer numbers to float. If Pallene silently converted integer function parameters to float, this function would produce a different result than the equivalent Lua code that does not have any type annotations.

of Siek et al. [24], which dictates that removing the type annotations from a Pallene program and running it as a Lua module should produce the same result, except that Pallene may raise run-time type errors that Lua would not.

When Pallene manipulates values that come from Lua, it must use run-time type checks to ensure that the types are the ones it expects. For example, let's consider again the list-summing function from Figure 3. When Lua calls this function, it may pass it an ill-typed value that isn't an array, or an array with the wrong type of element. To protect itself against this possibility, Pallene checks if `xs` is actually an array before reading from it, and checks if `xs[i]` is a floating-point number before the floating-point addition in `s = s + xs[i]`. That said, we do not specify exactly when these run-time checks should occur. The main purpose of Pallene's type system is performance and this flexibility gives the compiler the freedom to move these type checks around the code and lift them out of loops. For instance, in the previously-mentioned example the type of `xs` would be checked before the loop, and the type of `xs[i]` would be checked as needed inside the loop. This sort of delayed or "superficial" checking is the norm for non-primitive types in Pallene. Pallene is type safe in the same way that a dynamically typed language is type safe: untrapped type errors and segfaults never happen, but trapped type errors are commonplace.

So far, keeping Pallene's semantics similar to a subset of Lua's semantics is mostly a matter of not allowing the type annotations to affect behavior (other than run-time type checking, of course). For instance, Pallene does not perform automatic coercions between numeric types when assigning values or calling functions, unlike most statically typed languages. To illustrate, consider the example in Figure 9. In Lua, as in many dynamic languages, the addition of two integers produces an integer while the addition of two floating-point numbers produces another floating point number. If we remove the type annotations from the Pallene function `add`, and treat it as Lua code, Lua will perform an integer addition and the program will print `false`. On the other hand, were Pallene to automatically coerce the integer arguments to match the floating-point type annotations, the program would have printed `true`. Pallene would have performed a floating-point addition and, since floating-point numbers cannot accurately represent $2^{54}+1$, it would be rounded down to 2^{54}. To avoid this inconsistency, Pallene does not perform automatic coercions between `float` and `integer`, when assigning a value. It instead complains that an integer was used where a floating-point value was expected.

Although Pallene's type annotations never introduce coercions between integers and floating point numbers, it is still possible to perform arithmetic operations on integer and floating point numbers—the result is a floating point number, just like it would be in Lua. For example, `1 + 2.0` evaluates to `3.0`. This is because the arithmetic operators are overloaded to accept any combination of integer and floating point parameters, as is indicated by the tables in Figure 7. This subtle distinction is specially relevant for the comparison operators. When comparing an integer and a floating point number, Lua and Pallene use accurate algorithms that always return the correct result. They don't just convert the integer to a floating point number and then perform a floating-point comparison, because that is a lossy conversion when large integers are involved.

5. Implementing Pallene

Our implementation of the Pallene compiler and runtime was driven by our goals of efficiency, seamless interoperation with Lua, simplicity, and portability. The compiler itself is quite conventional. After a standard parsing step, it converts the program to a high-level intermediate form and from that it emits C code, which is then fed into a C compiler such as gcc. The final executable complies with Lua's Application Binary Interface (ABI) and can be imported by Lua programs as a standard extension module.

The choice of C as a backend allowed us to simplify various aspects of the Pallene compiler. Firstly, we could rely on a mature compiler for machine code generation, register allocation, and several optimizations. Secondly, we could take advantage of the fact that the reference Lua interpreter is also written in C. Pallene shares many of its data structures and semantics with Lua, and by also using C we could include in our generated code many C snippets and macros lifted directly from the Lua interpreter source code.

The C backend is also useful for portability. Any platform that can run the Lua interpreter should also be able to run Pallene modules, since the generated C code for a Pallene module is standard C code that is similar to the code that is present in the reference Lua interpreter.

From the point of view of performance, the most important characteristic of Pallene is that it is typed. For instance, when we write `xs[i]` in Pallene (as in our running example from Figure 3), the compiler knows that `xs` is an array and `i` is an integer, and generates code accordingly. The equivalent array access in Lua (or in C code using the Lua–C API) would need to account for the possibility that `xs` or `i` could have different types or even that `xs` might actually be an array-like object with an `__index` metamethod overloading the `[]` operator.

Also important for performance is how Pallene directly manipulates private fields of Lua data structures and of the Lua runtime. Regular C extension modules for Lua must interact with Lua data structures through the Lua–C API [10]. The high-level stack-based design of this API is backwards compatible, and allows the Lua garbage collector to accurately keep track of live objects.[1] However, it necessarily introduces some overhead to the interaction between C and Lua. By bypassing the usual Lua–C API, Pallene can achieve better performance, as we show in Section 6. It is important to remember that Pallene can do this without sacrificing memory safety or garbage collection. The compiler is careful to generate code that respects the various invariants of the Lua runtime. For example, Pallene ensures that whenever the garbage collector is invoked, all pointers to garbage-collectable Lua objects are rooted in the Lua stack, where the garbage collector can see them. (Between runs of the garbage collector, some pointers might only be stored in local C variables.) These safety measures are easily enforced by a compiler, but enforcing them in hand-written C code would be a nightmare.

6. Performance Evaluation

In this section, we perform experiments to evaluate some of our assumptions and hypothesis about Pallene. That is to say:

- Pallene's performance is comparable with that of a good JIT compiler, despite the much simpler implementation.
- Rewriting parts of a Lua program in C does not always improve performance.
- The cost of run-time tag checking in Pallene is small and is more than compensated by gains due to type specialization.

We selected six micro benchmarks for this evaluation. All are array-focused benchmarks that have been commonly used by the Lua community. Based on known algorithms, and implemented in a straightforward manner, without any performance tuning for a particular implementation, the six benchmarks are the following:

[1] Lua's stack-based API contrasts with the lower-level C APIs of languages such as Python and Ruby, which offer fewer guarantees to the programmer. In Python the programmer is tasked with keeping track of object reference counts for memory management, and in Ruby the garbage collector is conservative regarding pointers held by the C stack.

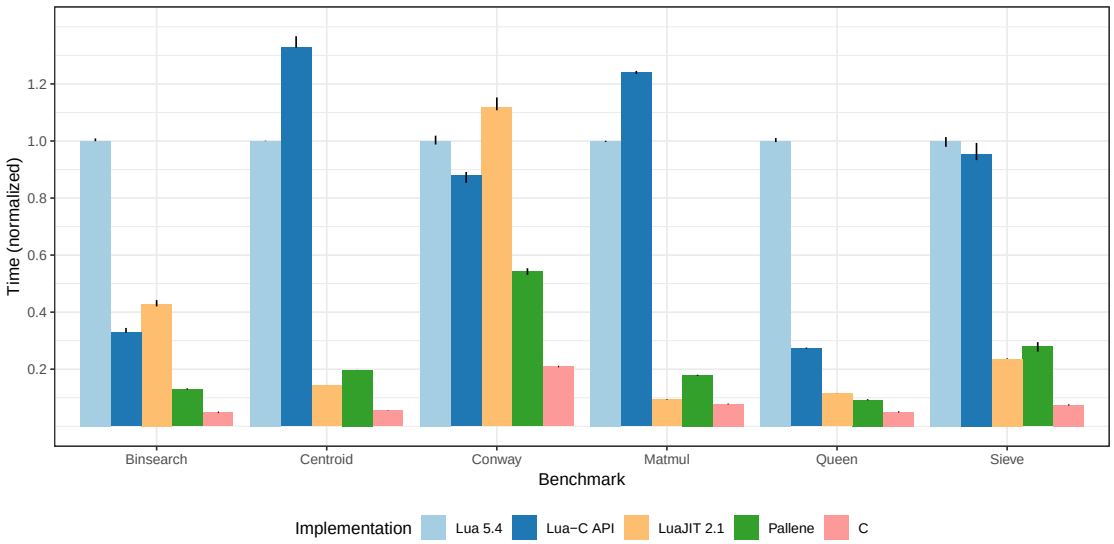

Figure 10: Comparison of Pallene with Lua, LuaJIT, the Lua–C API, and idiomatic C. Times are normalized by the Lua result. Lower is better. Lines represent 90% confidence intervals.

Matmul: multiplies two floating-point matrices represented as arrays of arrays.

Binsearch: performs a binary search over an array of integers.

Sieve: produces a list of primes with the sieve of Eratosthenes.

Queens: solves the classic eight-queens puzzle.

Conway: a cellular automaton for Conway's Game of Life.

Centroid: computes the centroid (average) of an array of points, each represented as an array of length 2.

For the main experiment, we ran each benchmark in five ways: Lua, LuaJIT, Lua–C API, Pallene and C. Lua means the standard Lua interpreter, version 5.4-work2. For LuaJIT we ran the same source code under the 2.1.0-beta3 version of the compiler. For Lua–C API we rewrote the benchmark code in C, but operating on standard Lua data structures through the Lua–C API. Pallene means our implementation, running Pallene programs that are similar to the Lua ones, except for the type annotations. To provide a familiar baseline, we also implemented the benchmarks in C, using C data structures.

We executed our experiments in a workstation with a 3.10 GHz Intel Core i5-4440 CPU, with 8 GB of RAM. The operating system was Fedora Linux 29, and our C compiler was GCC 8.2. For each benchmark implementation we performed forty measurements of the total wall-clock running time. To maximize the percentage of time spent running the benchmarks (as opposed to initializing them), we calibrated our benchmarks so each took at least one second to run. The Matmul, Queens, and Conway benchmarks feature algorithms with super-linear time complexity, so we simply increased the input size until it was large enough. For the Binsearch, Sieve, and Centroid benchmarks, we repeated the benchmark inside a loop to achieve the same effect. All the measurements we list are for total running time, including the ones for LuaJIT. In this set of benchmarks we observed that the LuaJIT warmup time was negligible so we did not calculate separate warmup and peak performance times, as is customarily done when evaluating just-in-time compilers.

Figure 10 shows the elapsed time for each benchmark. The bars represent the median running time for each benchmark, normalized by the Lua running time. The vertical lines represent a 90% confidence interval: the range of normalized running times encountered, excluding the 5% slowest and fastest results. Most results cluster near the median, with the exception of a small number of slower outliers. The exact

Benchmark	Lua	Lua–C API	LuaJIT	Pallene	Palene No Check	C
Binsearch	11.05	3.63	4.74	1.43	1.40	0.54
Centroid	18.62	24.73	2.64	3.67	3.44	1.03
Conway	4.44	3.90	4.97	2.41	2.37	0.93
Matmul	20.15	25.00	1.88	3.61	3.64	1.58
Queen	14.30	3.90	1.65	1.31	1.30	0.72
Sieve	11.22	10.72	2.65	3.14	3.02	0.84

Figure 11: Median running times for our benchmarks, in seconds. Pallene No Check is a memory-unsafe configuration of Pallene that skips run-time tag checks, which we created for this experiment.

N	M	Time ratio	Pallene time	LuaJIT time	Pallene LLC miss	LuaJIT LLC miss
100	1024	1.02	1.46	1.44	0.25%	0.26%
200	128	1.15	1.46	1.28	15.83%	2.26%
400	16	1.86	2.74	1.48	49.59%	37.34%
800	2	1.90	2.86	1.51	48.83%	38.81%

Figure 12: Median running time and cache misses for the Matmul benchmark on different input sizes. N is the size of the input matrices. M is how many times we repeated the multiplication. Time ratio is Pallene time divided by LuaJIT time. Times are in seconds. Last Level Cache (LLC) load misses are a percentage of all LL-cache hits.

times in seconds are summarized in Figure 11.

In these tests, Pallene running times are comparable with LuaJIT, at around half the speed of idiomatic C. The fact that it can achieve results comparable with a mature optimizing compiler suggests that Pallene may be suitable as system language for writing lower-level modules for Lua, at least in terms of performance.

The Lua–C results are all over the place. For Matmul and Centroid, benchmarks with more operations on Lua arrays, the Lua–C API overhead outweighs the gains from using C instead of Lua. This is because the operations for manipulating a Lua array through the Lua–C API pushing and for popping values from the Lua stack have to check at runtime whether the C code has pushed the correct number and the correct type of values to the stack.

Let us now analyze the Pallene versus LuaJIT situation in more detail. The only benchmark where LuaJIT is substantially faster than Pallene is Matmul. We have found that this difference is due to memory access. LuaJIT uses the *NaN-boxing* technique to pack arbitrary Lua values and their type tags inside a single IEEE-754 floating-point number [28]. In particular, this means that in LuaJIT an array of floating-point numbers consumes only 8 bytes per number, against the 16 bytes used by Lua and Pallene. This results in a higher cache miss rate and worse performance for Pallene. To confirm that this is the case, we also ran this benchmark under smaller values of N, measuring running times and cache-miss rates using the Linux `perf` tool. The results are summarized in Figure 12. For smaller values of N, the matrices always fit inside the machine cache and Pallene and LuaJIT are just as fast. For larger N, LuaJIT's more compact data representation leads to less cache misses and better performance. This NaN-boxing effect also explains the difference between LuaJIT and Pallene in the Centroid benchmark.

The NaN-boxing technique, however, is accompanied by other problems that usually do not show up in benchmarks. For example, NaN tagging is incompatible with unboxed 64-bit integers, which is one of reasons why this technique is not used in the standard Lua interpreter anymore.

The Binsearch benchmark highlights a particularly bad scenario for trace-based JITs, such as LuaJIT. The inner loop of the binary search features a highly unpredictable branch, forking the hot path. This is not an issue for Pallene and other ahead-of-time compilers.

The Sieve and Queens benchmarks need no further explanation as the results were quite expected. Both

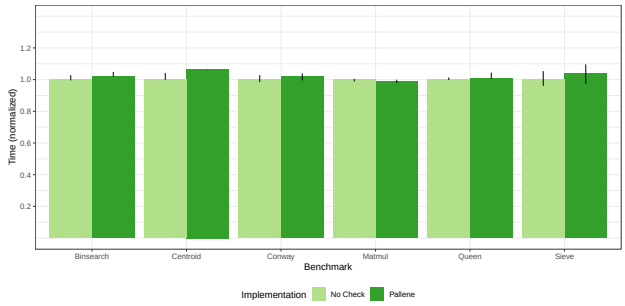

Figure 13: Pallene run-time tag checking overhead. Times are normalized by the No Check result. Lower is better. Lines represent 90% confidence intervals.

Implementation	Time	Cycles	Instructions	Intructions per Cycle
Pallene	2.86	9.36×10^9	16.68×10^9	1.78
Pallene No Check	2.86	9.34×10^9	7.46×10^9	0.80

Figure 14: Run-time tag checks correspond to more than half of the x86-64 instructions executed in the Matmul benchmark. However, they do not impact the total running time, due to the Intel Core i5's instruction-level parallelism. Times in seconds, $N = 800$, $M = 2$.

LuaJIT and Pallene are around ten times faster than Lua.

The Conway benchmark results are surprising because LuaJIT performed worse than standard Lua. This unusual result is due to the new generational garbage collector introduced in Lua 5.4. It turns out that the bulk of the time in this benchmark is spent doing string manipulation and garbage collection and LuaJIT still relies on the incremental garbage collector that was used until version 5.3. In a separate experiment, we disabled the generational mode of the 5.4 garbage collector and its performance slowed down to become comparable to LuaJIT's. This difference highlights that JIT compilation is not a panacea and that sometimes other aspects can dominate the execution time.

6.1. The cost of run-time tag checks

We also investigated the effect of run-time tag checking in our implementation of Pallene arrays. Since Pallene arrays can be directly modified by untyped Lua code, Pallene must perform a runtime tag check whenever it reads from an array slot that might have been written to by untyped code. In this experiment we ran our benchmark suite under both the default Pallene compiler and No Check, an unsafe version of Pallene that skips all the run-time tag checks. Figure 13 shows the results of this experiment, normalized against the No Check implementation. The exact times are also summarized in Figure 11.

In all our benchmarks, the tag checks next to the array reads introduce only little overhead, ranging from 0% to 10%, which we consider surprisingly low. Many of our benchmarks perform run-time checks inside their innermost loops, and this kind of run-time tag checking often introduces greater overheads in other languages. For example, Reticulated Python also checks types when accessing lists and object fields, but in their case the average slowdown was by a factor of 2.5x [29].

To understand why our tag checks were cheaper than expected, we again resorted to Linux's `perf` tool. The extreme case of the Matmul benchmark is particularly illuminating. It had a 0% tag-checking overhead despite the presence of multiple run-time tag checks for the array reads in its innermost loop. In Figure 14 we show the Instructions per Cycle statistics for the Matmul that we collected with `perf`. Although the tag checks accounted for more than half of the machine-level instructions that were executed, the final running time is still the same with or without them. This happens because the Matmul benchmark is memory bound, as indicated by the low (≤ 1.0) Instructions per Cycle statistic for Pallene No Check. This gives plenty of

room for the pipelining in the Intel Core CPU to execute the extra tag-checking instructions in parallel, effectively for free.

In our other benchmarks we observed a similar pipelining benefit, albeit less pronounced than in Matmul. Naturally, the run-time overhead of tag checking may be higher for CPU-bound benchmarks, or on other kinds of CPUs. Nevertheless, these results suggest that run-time tag checking is not incompatible with efficient typed code, specially if the tag checks are compiled down to machine-level jumps (as opposed to source-level conditional statements, or bytecode-level jumps).

7. Related Work

In this section we briefly discuss how Pallene compares to related work in type systems and optimization for dynamic languages.

JIT compilers answer a popular demand to speed up dynamic programs without the need to rewrite them in another language. However, they do not evenly optimize all idioms of the language, which in practice affects programming style, encouraging programmers to restrict themselves to the optimized subset of the language. Pallene has chosen to be more transparent about what can be optimized, and made these restrictions a part of the language. Aditionally, developing a JIT compiler is a notably complex afair. Pallene, on the other hand, was intentionally designed to be implementable with a simple ahead-of-time compiler.

Like Gradual Typing systems, Pallene recognizes that adding static types to a dynamic language provides many benefits, such as safety, documentation, and performance. Pallene is also inspired by the Gradual Guarantee, which states that the typed subset of the language should behave exactly as the dynamic language, for improved interoperability. However, unlike gradually typed systems, Pallene can only statically type a restricted subset of Lua, and does not attempt to be a superset of Lua. This avoids the complexity and performance challenges that are common in many gradually typed systems.

Typed Lua [26] is a gradual type system for Lua. Its types can be used for documentation and compile-time error detection. Typed Lua aims to be flexible enough to type a wide variety of Lua programs and has a rich set of table types to model the many different uses of Lua tables. Similarly to TypeScript [30], Typed Lua is intentionally unsound, meaning its types cannot be used for program optimization. They are erased before the program runs, and have no effect at run-time.

Common Lisp is another language that has used optional type annotations to provide better performance. As said by Paul Graham in his ANSI Common Lisp Book [23], "Lisp is really two languages: a language for writing fast programs and a language for writing programs fast". Pallene and Common Lisp differ in how their sub-languages are connected. In Common Lisp, they live together under the Lisp umbrella, while in Pallene they are segregated, under the assumption that modules can be written in different languages. Common Lisp also has an option to unsafely disable run-time tag checks.

RPython [31, 14, 15] is a restricted subset of Python which can be compiled to efficient machine code, either through ahead-of-time compilation (to C source code) or just-in-time compilation (through the PyJitPl JIT). Unlike Pallene, RPython does not have explicit type annotations, and is not defined by a syntax-directed type system. The types are inferred by a whole-program flow-based analysis that uses a graph of live Python objects as starting point. Finally, one very important difference is that Pallene modules can be independently-compiled, and called from Lua code running in an unmodified version of the reference Lua interpreter. RPython-based executables, on the other hand, must be built as a single compilation unit. Compiled RPython modules are also incompatible with CPython, the reference Python interpreter.

Cython [32] is an extension of Python with C datatypes. It is well suited for interfacing with C libraries and for numerical computation, but its type system cannot describe Python types. Cython is unable to provide large speedups for programs that spend most of their time operating on Python data structures.

Terra [33] is a low-level system language that is embedded in and meta-programmed by Lua. Similarly to Pallene, Terra is also focused on performance and has a syntax that is very similar to Lua, to make the combination of the two languages more pleasant to use. However, while Pallene uses Lua as a scripting language, Terra uses it as stage-programming tool. The Terra system uses Lua to generate Terra programs

aimed at high-performance numerical computation and, once produced, these programs run independently of Lua. Terra uses manual memory management and features low-level C-like datatypes. There are no language features to aid in interacting with a scripting language at run-time.

8. Conclusion and Future Work

In this paper we presented Pallene, a companion language to Lua. Pallene is an ahead-of-time-compiled, typed subset of Lua, designed for situations where efficiency and interoperability with Lua are desired. From the point of view of the scripting paradigm, Pallene is a system-language counterpart to Lua. By being typed and ahead-of-time-compiled, Pallene can have a simple and portable implementation that is still very efficient. By being a subset of Lua and sharing its data-structures and runtime, Pallene can call and be called from Lua with low overhead. It is also much easier to convert an existing Lua module to Pallene than to a completely different language like C.

Our performance evaluation shows that Pallene can be an order of magnitude faster than Lua and is around as fast as LuaJIT, a state-of-the-art JIT compiler. It also shows that Lua extension modules written in Pallene can be faster than those written in C. This suggests that Pallene may be a viable choice for implementing performance-sensitive libraries and applications.

In future work we plan to study the impact of implementing traditional compiler optimizations such as common sub-expression elimination and loop invariant code motion. Our current implementation relies on an underlying C compiler for almost all optimizations, and it cannot be expected to understand Lua-level abstractions. For example, Pallene frequently calls into the Lua runtime to create or resize arrays, or to invoke the garbage collector. The C compiler cannot know precisely what are the various side effects that these operations perform, so it often assumes the worse. Therefore, we expect that implementing some optimizations at the Pallene level might result in more improvements.

We also intend to investigate whether the type system simplicity and the good performance results we achieved in array-based benchmarks will be preserved as we add more language features, such as modules and a foreign function interface.

Another potential avenue for future work would be to formalize the semantics of Pallene. This might help to provide a precise description of the corner cases where Pallene might not behave as a strict subset of Lua. Soldevilla et al. have recently developed a formal model of Lua's semantics [34], which could potentially be used as a basis for formalizing the semantics of Pallene.

Acknowlegements

We would like to thank Hisham Muhammad, Gabriel Ligneul, Fábio Mascarenhas, and André Maidl for useful discussions about the initial ideas behind Pallene. We would also like to thank Sérgio Medeiros for assisting us with the development of Pallene's scanner and parser.

Pallene was born as a fork of the Titan [35] programming language, with a focus on researching the performance aspects of dynamic programming language implementation. Gabriel Ligneul has heavily contributed to current implementation of the Pallene compiler, as well as its continued design.

This study was financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico—Brasil (CNPQ)—Finance Codes 153918/2015-2 and 305001/2017-5, and by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior—Brasil (CAPES)—Finance Code 001.

References

[1] H. M. Gualandi, R. Ierusalimschy, Pallene: a statically typed companion language for Lua, in: Proceedings of the XXII Brazilian Symposium on Programming Languages (SBLP), 2018, p. 19–26. doi:10.1145/3264637.3264640.

[2] J. K. Ousterhout, Scripting: Higher-level programming for the 21st century, Computer 31 (3) (1998) 23–30. doi:10.1109/2.660187.

[3] L. P. Deutsch, A. M. Schiffman, Efficient implementation of the Smalltalk-80 system, in: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '84, 1984, p. 297–302. doi:10.1145/800017.800542.

[4] J. Guerra, LuaJIT hacking: Getting next() out of the NYIlist, CloudFare Blog (Feb. 2017).
URL https://blog.cloudflare.com/luajit-hacking-getting-next-out-of-the-nyi-list/
[5] P. Antonov, et al., V8 optimization killers, retrieved in 2017-01-08. (2013).
URL https://github.com/petkaantonov/bluebird/wiki/Optimization-killers
[6] S. Tobin-Hochstadt, M. Felleisen, Interlanguage migration: From scripts to programs, in: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA '06, 2006, p. 964–974. doi:10.1145/1176617.1176755.
[7] J. G. Siek, W. Taha, Gradual typing for functional languages, Scheme and Functional Programming Workshop 6 (2006) 81–92.
[8] G. Bracha, Pluggable type systems, OOPSLA04 Workshop on Revival of Dynamic Languages (2004).
[9] M. Pall, LuaJIT, a just-in-time compiler for Lua (2005).
URL http://luajit.org/luajit.html
[10] R. Ierusalimschy, L. H. De Figueiredo, W. Celes, Passing a language through the eye of a needle, Communications of the ACM 54 (7) (2011) 38–43. doi:10.1145/1965724.1965739.
[11] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, The evolution of Lua, in: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, HOPL III, 2007, p. 2–1–2–26. doi:10.1145/1238844.1238846.
[12] B. Mogilefsky, Lua in Grim Fandango, Grim Fandango Network (May 1999).
URL https://www.grimfandango.net/features/articles/lua-in-grim-fandango
[13] A. Gal, C. W. Probst, M. Franz, HotpathVM: An effective JIT compiler for resource-constrained devices, in: Proceedings of the 2nd International Conference on Virtual Execution Environments, VEE '06, 2006, p. 144–153. doi:10.1145/1134760.1134780.
[14] C. F. Bolz, A. Cuni, M. Fijalkowski, A. Rigo, Tracing the meta-level: PyPy's tracing JIT compiler, in: Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS '09, 2009, p. 18–25. doi:10.1145/1565824.1565827.
[15] The PyPy Project, RPython official documentation (2016).
URL https://rpython.readthedocs.io/
[16] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, M. Wolczko, 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, 2013, p. 187–204. doi:10.1145/2509578.2509581.
[17] M. Pall, et al., Not Yet Implemented operations in LuaJIT, LuaJIT documentation Wiki, retrieved 2017-01-08. (2014).
URL http://wiki.luajit.org/NYI
[18] M. Pall, LuaJIT performance tips, lua-l mailing list (nov 2012).
URL http://wiki.luajit.org/Numerical-Computing-Performance-Guide
[19] J. Guerra Giraldez, LOOM - a LuaJIT performance visualizer (2016).
URL https://github.com/cloudflare/loom
[20] J. Guerra Giraldez, The rocky road to MCode, Talk at Lua Moscow conference, 2017 (2017).
URL https://www.youtube.com/watch?v=sz2CuDpltmM
[21] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, L. Tratt, Virtual machine warmup blows hot and cold, in: Proceedings of the 32nd Annual Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '17, 2017, pp. 52:1–52:27. doi:10.1145/3133876.
[22] H. M. Gualandi, Typing dynamic languages – a review, Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio) (2015).
[23] P. Graham, ANSI Common LISP, Apt, Alan R., 1996.
URL http://www.paulgraham.com/acl.html
[24] J. G. Siek, M. M. Vitousek, M. Cimini, J. T. Boyland, Refined criteria for gradual typing, in: 1st Summit on Advances in Programming Languages, SNAPL '2015, 2015, p. 274–293. doi:10.4230/LIPIcs.SNAPL.2015.274.
[25] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, M. Felleisen, Is sound gradual typing dead?, in: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, 2016, p. 456–468. doi:10.1145/2837614.2837630.
[26] A. M. Maidl, F. Mascarenhas, R. Ierusalimschy, A formalization of Typed Lua, in: Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, 2015, p. 13–25. doi:10.1145/2816707.2816709.
[27] Y. Zhang, Openresty (2011).
URL https://openresty.org/en/
[28] M. Pall, LuaJIT 2.0 intellectual property disclosure and research opportunities, lua-l mailing list (nov 2009).
URL http://lua-users.org/lists/lua-l/2009-11/msg00089.html
[29] M. M. Vitousek, C. Swords, J. G. Siek, Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems, in: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, 2017, pp. 762–774. doi:10.1145/3009837.3009849.
[30] G. Bierman, M. Abadi, M. Torgersen, Understanding TypeScript, in: 28th European Conference on Object-Oriented Programming, ECOOP '14, 2014, p. 257–281. doi:10.1007/978-3-662-44202-9_11.
[31] D. Ancona, M. Ancona, A. Cuni, N. D. Matsakis, RPython: A step towards reconciling dynamically and statically typed OO languages, in: Proceedings of the 2007 Symposium on Dynamic Languages, DLS '07, 2007, p. 53–64. doi:10.1145/1297081.1297091.
[32] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, K. Smith, Cython: The best of both worlds, Computing in Science Engineering 13 (2) (2011) 31–39. doi:10.1109/MCSE.2010.118.

[33] Z. DeVito, Terra: Simplifying high-performance programming using multi-stage programming, Ph.D. thesis, Stanford University (Dec. 2014).

[34] M. Soldevila, B. Ziliani, B. Silvestre, D. Fridlender, F. Mascarenhas, Decoding lua: Formal semantics for the developer and the semanticist, in: Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, DLS 2017, 2017, p. 75–86. doi:10.1145/3133841.3133848.
URL https://arxiv.org/pdf/1706.02400.pdf

[35] A. M. Maidl, F. Mascarenhas, G. Ligneul, H. Muhammad, H. M. Gualandi, Source code repository for the Titan programming language (2018).
URL https://github.com/titan-lang/titan