Lua++: A language for modern times
Project of the National Contest for Swiss Youth in Science

Max Prihodko
max@prihodko.com

March 2023

Abstract

Lua++ is a fully-featured new programming language for a wide range of applications that is fast, lightweight, and energy-efficient. The author of this paper created Lua++ with the goal of increasing software efficiency, decreasing barriers to adoption on low-cost hardware platforms, and providing a tool for learning advanced computer science concepts.

Borrowing from several modern languages, Lua++ adds proper object-oriented programming with a class hierarchy and encapsulation, generics, type annotations, events, attributes, and much more. Overcoming the limitations of its popular predecessor, Lua++ is suitable for use by teams of developers in large cross-platform projects.

The author designed the syntax, built a state-of-the-art optimizing compiler, composed an instruction set, and implemented a fast, compact virtual machine to run Lua++ bytecode in end-user applications. Based on performance tests consisting of several commonly used benchmarks, Lua++ is four to 25 times faster than classic Lua.

This paper gives an overview of the language, discusses rationales for feature choices made, and showcases Lua++'s advantages.

Contents

1 Introduction

The Lua programming language was created in 1993 by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes at the Pontifical Catholic University of Rio de Janeiro [6]. It was designed to be lightweight, high-level, and embedded in applications. At the time of its creation, Lua had very few competitors so it quickly became the top choice for some of the world's top tech companies including Waze, World of Warcraft, Adobe, and Roblox. Today, Lua can be found virtually everywhere: in games, operating systems, mobile apps, web servers, embedded systems, hardware, databases, and statistical software.

Although Lua remains a very practical language, by design it lacks productivity features common in most of today's industrial strength programming languages like C++, Java, C#, or Rust. It also suffers from poor code maintainability and slow execution speed, both of which have become more important with Lua's scale across many different types of applications. Consequently, it is difficult to develop large applications using Lua and it remains a niche solution.

Some heavy Lua users have implemented solutions to some of these problems. To handle over 1 million lines of Lua code, Roblox developed their own flavor called Luau with upgraded performance and a strict type system that makes it easier to write robust code [8]. However, Luau's application is very much limited to game development, specifically to features that benefit the Roblox gaming platform. It does not adequately address the plethora of additional Lua applications outside of the game industry. Another popular example is LuaJIT, a tracing just-in-time compiler, that takes advantage of the performance benefits of just-in-time compilation to enhance Lua [11]. Yet, while it improves Lua's execution speed, it does nothing to address the language's outdated syntax and simplified set of features.

2 Why Lua++?

Recognizing the shortcomings and limitations of Lua, the author of this paper felt the time was ripe to create Lua++ for general application development. The impetus behind this was threefold:

Software efficiency As the exponential growth in hardware advancements flattens, producing faster and more efficient software has never been more important [10]. An efficient and fast virtual machine paired with a powerful optimizing compiler can help produce next-generation software without major advancements in hardware performance.

Empowerment Low computational requirements will allow use on affordable hardware devices and thereby make advanced programming accessible to historically disadvantaged populations in both developed and developing countries.

Educational opportunities In 2020, more than half of American children under the age of 16 played Roblox [4], a game platform that encourages players to create their own three-dimensional first person games using Lua as a scripting language. Even though writing primitive games in Python and assembling robotics kits are all the rage, chances are that the first language that a child will encounter and relate to will actually be Lua. State-of-the-art programming techniques should be easily available to them at the opportune teaching moment.

As the rest of the paper will demonstrate, Lua++ retains all the benefits of Lua while making it possible to run code close to the speed of today's fastest programming languages, while drastically improving code maintainability and enabling effective development of large code bases.

3 What is Lua++?

To explain the benefits of Lua++ over Lua, it may be helpful to start with what Lua++ is not. Lua++ is not a transpiler. It does not take the script and convert it to a Lua binary to run using Lua's runtime library. Instead, it converts the script into its unique binary code, which is sent to the Lua++ virtual machine (VM) embedded in user devices. The Lua++ compiler differs from the Lua compiler in that it takes the script and breaks it up into an abstract syntax tree, an object-based representation of the code. The objects are then cleaned up and converted to an intermediate, and therefore, simpler representation of the machine code, which can be easily optimized and converted to bytecode. As a result, the VM is extremely compact. Since its only value is to execute each instruction, it does not have to worry about runtime errors as the compiler has already caught many of them. In this way, Lua++ resembles some of today's top programming languages, such as C and Rust, whose newer compilation techniques produce extremely fast programs which can easily defeat any Lua script. Lua++ is therefore a clean, modern, and truly object-oriented version of Lua with many performance upgrades. For backward compatibility, it implements a familiar type interface into the Lua syntax while optimizing the backend of the language. Restructuring the entire Lua syntax does not inflate the compiled programs because the compiler and virtual machine were designed to give developers an easily embeddable interface while keeping the language machine code as compact as possible.

Moreover, the Lua++ virtual machine is so specific that it does not require any calculations or checks while running the Lua++ bytecode. Each instruction is straightforward and clarifies to the VM, which operation needs to occur. The compiler performs as many of these checks which make the VM more lightweight. This way Lua++ is comparable to some of today's fastest interpreted languages.

4 Language Improvements

The examples below will illustrate the many syntax improvements that Lua++ offers. Through them, the reader should be able to see how Lua++ makes it easier to design complex programs quickly and efficiently. Each improvement to the language provides a more comfortable programming environment for Lua++ developers while minimizing the lines of code of a script. The implications of specific improvements offered by Lua ++ are described in more detail in the following paragraphs.

4.1 Compound Assignments

When writing programs, it is generally necessary to increment a variable. The problem with Lua and many other high-level scripting languages is that they don't provide an efficient way to do such a task. To increment a variable in Lua, the code would look like this:

```
a = a + 13
```

Although this method is simple and straightforward, it becomes inefficient when dealing with large-scale programs and longer variable names. Additionally, the VM needs to reference the value of `a`, add `13` to it, and update the original value of `a`. An operator for actions like this would decrease execution time and improve developer productivity. The following code block shows an example using the `+=` compound assignment.

```
a += 13
```

Like C++ and Java, Lua has a compound assignment operator for each arithmetic and string operation, such as (`+=`, `-=`, `/=`. `*=`, `^=`, `%=`, and `..=`). Each operation assigns a variable the value of itself with the operation performed on the original value.

4.2 Prefix and Postfix Operators

Another addition to the Lua++ language is prefix and postfix operators. These can be used to increment and decrement a specific variable within your program. These types of operators are frequently found in C-based programming languages like C++ [9], C# [7], or Java [2]. The following example demonstrates how to use a postfix increment operator to print numbers from 0 to 9.

```
local v = 0

while v < 10 do
    print(v)
    v++          -- increment value of v by 1
end
```

Listing 1: Postfix incrementation go print 0 to 9

The `++` operator will first return the value of that variable and then increment it by 1. If you want to increment the value prior to returning it, you can prefix your variable reference with the `++` operator.

```
local v = 1

print(v++)   -- prints 1
print(++v)   -- prints 3
```

Listing 2: Prefix and postfix variable incrementation

To decrement variables the `---` operator is used which functions exactly like the `++` operator. Lua uses `--` to indicate the beginning of a comment, which conflicts with the conventional `--` decrement operator of many other programming languages. Therefore, Lua++ provides a workaround that disables standard Lua comments using an attribute that can be found in section 4.8.1.

4.3 The continue statement

Another feature shortcoming of Lua is that it lacks the `continue` statement. It can be found in various programming languages but was first introduced in C. This statement is essentially the opposite of the `break` statement, forcing the next iteration to begin skipping any code in between. While the Lua creators have implemented a `goto` statement in more recent versions of Lua it often leads to messy, unmanageable code.

In the below comparison of Lua's new `goto` statement to Lua++'s `continue` statement, one can see the added convenience gained by having the `continue` statement in the language.

```
local a = 1

initial:
    while a < 10 do
        if a % 2 == 0 then
            goto initial
        end

       print("odd number: ", a)
       a = a + 1
    end
```

Listing 3: Lua's messy goto statement

```
local a: number = 1

while a < 10 do
    if a % 2 == 0 then
        continue
    end

   print("odd number: ", a++)
end
```

Listing 4: The new continue statement

However, while convenient Lua++'s `continue` statement does have some restrictions. For example, if continue is used in a `repeat` . . . `until` loop, it may not skip the local variable used in the loop condition. Code like this is invalid and will throw an error at compile time:

```
repeat
   continue      -- Compiler catches this!
   local a = 1
until a > 0
```

Listing 5: Invalid use of continue in repeat loops

4.4 Constant Variables

Constant variables are variables that are immutable after their first assignment, meaning they can **never** be reassigned or modified. While they provide no performance benefit, as the compiler will propagate unchanged variables, constant variables are a safety feature for large projects where other developers might not realize that the author meant for the variable to stay constant. Inspiration for this feature came from Java, where the `const` keyword serves the same purpose.

Although immutable values can be simulated in Lua with tables, such implementations are often messy, require external installation, and are performed at runtime, making Lua programs even slower. In Lua++ no 3rd party libraries are needed and execution speeds will not suffer because, as previously mentioned, the compiler is responsible for such checks.

Here is a simple example of constant variables in action:

```
const a = 12

print(a)

a = 1    -- Compiler error!
```

Listing 6: Constant variables are immutable

4.4.1 Constant Functions

Constant functions serve a similar purpose as constant variables do but have a slightly different syntax. If you have a function that performs a specific calculation and you don't want to create a new one, you can do the following in Lua++:

```
const function add(x, y): number ->
  return x + y

print(add(1, 2))
```

Listing 7: Constant function

With this syntax, you are telling the compiler that you don't need an entirely new environment and that the content of this function can be replaced with all the references in your code. Just like constant variables, the compiler can already identify whether functions like this are defined as constant or not, so from a performance standpoint, it doesn't matter, but as a developer, this can make your code much cleaner and easier to understand.

4.5 Type Annotations

Unlike any of today's Lua versions, Lua++ can restrict variables to be of a certain type. As projects grow in size, it becomes harder for developers to remember of what type each variable is. Type annotations help prevent insidious bugs that can take hours to discover. In addition, they help enforce interfaces to classes in object-oriented programming.

When adding this feature there were many different ways of type annotation to choose from, but in order to make Lua syntax backward compatible with Lua++ so that programmers would have direct control over their development environments, the decision was made to inherit Typescript's way of noting types.

Lua++ supports five variable primitive types: `boolean`, `string`, `number`, `Table`, and `Array`, which are described in detail in section 5. To assign a type to a variable, use the : separator after the variable name. The annotation can also be used on constant variables. If a variable is assigned a value whose type is not equal to the type of the annotation, then a compiler error will be thrown.

```
local a: number = 12
const b: string = "Hello, world!"
```

Listing 8: Variable type annotations

4.5.1 Function Annotations

Function annotations serve a similar purpose to type annotations, in that they tell the compiler what type of function to assign to this variable. This can be done either by assigning the type to a local variable or in the function itself.

```
local f: (number, string): boolean

-- Valid assignment
f = function(a: number, b: string): boolean
    return true
end
```

Listing 9: Function type annotation

The code above demonstrates how to utilize function annotations correctly. Note that primitive types are interchangeable. If the types of functions are not identical to the definition, then the compiler will throw an exception. Below is another example of how to annotate a function with types:

```
local function f(a: number, b: number): boolean
    return a == b
end

-- Error: should return boolean not string
local function g(x: number): boolean
    return "Hello"
end
```

Listing 10: Another function type annotation

4.6 Classes

Lua++ has been designed to support real object-oriented programming by implementing classes. They function similarly to Lua tables but eliminate the need for simulated solutions using metamethods.

Under the hood, classes and tables are represented identically in the VM, but the compiler is able to optimize class behaviors by evaluating its properties and functions. This is something that will not come when using tables with the Lua++ compiler because as of now it is unable to match table metamethods with their associative behaviors. This is something that we have plans for in the future but as of now, classes are significantly faster than tables.

Just like many features in the language, classes in Lua++ are similar to those in Typescript, but the members of a class are separated by commas like a list. A quick breakdown of Lua++ classes is provided below:

```
class [tag [template-spec] : [base-list]] {
  member-list
}
```

Listing 11: Generic class definition in Lua++

Table 1: Parts of class definition

Token	Description
tag	The type name given to the class.
template-spec	Optional template specifications.
base-list	Optional list of classes this class will derive members from.
member-list	List of members.

To demonstrate just how superior classes are to the legacy tables, consider code fragments that calculate the area of a triangle.

```
class Triangle {
  b: number,
  h: number,

  constructor(b: number, h: number)
    self.b, self.h = b, h
  end,

  const function area(): number ->
    return (self.b * self.h) / 2.0
}

print(Triangle(3, 4).area())
```

Listing 12: Area of triangle calculation in Lua++

Contrast code above with the following implementation in classic Lua:

```
local Triangle = { width = 0, height = 0 }

function Triangle:set( fWidth, fHeight )
    self.width = fWidth
    self.height = fHeight
end

function Triangle:get()
    return {
        width = self.width,
        height = self.height
    }
end

function Triangle:area()
```

```
    return self.width * self.height / 2.0
end

Triangle:set(100, 300)
print(Triangle:area())
```

Listing 13: Area of triangle calculation in Lua

As illustrated, Lua++ classes provide a simple and straightforward interface for getting and setting member variables while Lua's tables perform a similar function but are awkward and inconvenient.

4.6.1 Constructors

Lua++ supports two different kinds of constructors by default: `implicit` and `explicit`. Explicit constructors require the parameters to be wrapped in parentheses, while `implicit` allow the value to be directly passed as shown in the following examples:

```
class ExplicitConstructor {
    value: number,

    constructor(value: number)
        self.value = value
    end
}

local exp: ExplicitConstructor = ExplicitConstructor(1)
```

Listing 14: Explicit constructor example

```
class ImplicitConstructor {
    value: number,

    implicit constructor(value: number)
        self.value = value
    end
}

local imp: ImplicitConstructor = 1
```

Listing 15: Implicit constructor example

4.6.2 Generics

Generic classes are commonly found in type-oriented programming languages like TypeScript [3] or C++ [9]. While class generics are sometimes seen as having limited applications, they allow the creation of type-flexible classes that significantly reduce repetitive code.

Class generics in Lua++ are quite simple and efficient. They work like so: `tag<T>` where `T` is a type. To allow more than one type, one would separate each type name with a comma, such as: `tag<T1, T2, ...>`. A simple example of a generic class is shown below:

```
class Pair<T1, T2> {
  first: T1,
  second: T2,

  implicit constructor(f: T1, s: T2)
    self.first, self.second = f, s
  end
}

local pair: Pair<number, string> = { 3, "Three" }
```

Listing 16: Generic class example

4.6.3 Inheritance

Like all object-oriented programming languages, Lua++ supports full class inheritance. To inherit all the heritable properties of a class in Lua++, one would separate the class name from the base class using a colon in this way: `tag : base-class`, and for multiple class inheritance, one would separate each base class with a comma: `tag: base-1, base-2, ...` as shown in the following example:

```
class Person {
    firstName: string,
    lastName:  string,

    constructor(firstName: string, lastName: string)
        self.firstName = firstName
        self.lastName = lastName
    end,

    function getName(): string
        return self.firstName .. " " .. self.lastName
    end,
    function getDescription(): string
        return "This is " .. self.getName() .. "."
    end
}
```

Listing 17: Parent class implementation

Here we define a base class `Person`, with the `firstName` and `lastName` properties. This class has two public methods, `getName()`, and `getDescription()`. As mentioned earlier, to inherit a class you use the : operator. For example, the following `Employee` class inherits properties and methods from the `Person` class:

```
class Employee : Person {
    -- ...
}
```

Listing 18: Child class Employee inherits from Person

Since the `Person` class has a constructor that initializes the `firstName` and `lastName` properties, you need to initialize these properties in the constructor of the `Employee` class by calling its parent class' constructor. This can be done by using the `base()` keyword.

```
class Employee : Person {
    job: string,

    constructor(firstName: string, lastName: string,
                job: string)
        self.job = job

        -- Call the constructor of the person class
        base(firstName, lastName)
    end
}
```

Listing 19: Calling a parent class' constructor

The following creates an instance of the `Employee` class which inherits all the methods and properties of the `Employee` class:

```
local employee: Employee = Employee("Max", "Prihodko",
                                    "Programmer")
```

Listing 20: Instance of a class with inherited properties and methods

Lua++ also allows you to override methods inherited from a base class. Refer to the example below where `Employee` overrides the `describe()` method inherited from the `Person` class.

```
class Employee : Person {
    job: string,

    constructor(firstName: string, lastName: string,
                job: string)
        self.job = job

        -- Call the constructor of the person class
        base(firstName, lastName)
    end,

    function describe(): string
        return base.describe() .. " I am a " ..
                                    self.job .. "."
    end
}

local employee: Employee = Employee("Max", "Prihodko",
                                    "Programmer")

print(employee.describe())
```

Listing 21: Overriding methods of the base class

The output of this fragment of code will be:

```
This is Max Prihodko. I am a Programmer.
```

4.6.4 Encapsulation

One of the biggest problems with Lua's simulated OOP is that it does not provide a safe method of limiting access to member variables and methods. One can't create "black-box" style interfaces to classes and libraries to prevent other programmers from modifying things they shouldn't. In Lua++ all members of a class are public by default, but they can be hidden from being accessed from anywhere but within the class using the `private` keyword.

```
class Person {
    -- This variable can't be accessed outside the class
    private age: number,
    constructor(age: number)
        self.age = age
    end,

    function getAge(): string
        return "This person is " ... age ... " years old."
    end
}

local p: Person = Person(100)
print(p.getAge())  -- OK
print(p.age)       -- Error, since age is private
```

Listing 22: Example usage of the private keyword.

Class methods can also be made `private` to hide those that should only be called internally by other class functions.

4.7 Events

The event keyword is used to declare an event in a publisher class. The following code segment displays how to declare and raise an event.

```
class Publisher {
    event SampleEvent: (sender: any, text: string),

    function RaiseSampleEvent(text: string)
        SampleEvent.Invoke(self, text)
    end
}
```

Listing 23: Declare and raise an event.

Events can only be invoked from within a class, so they can be extremely useful when implementing a customizable API or library. One can assign a function to be called whenever the publisher class raises the event by doing the following:

```
function OnInvoked(sender, text: string)
  print("SampleEvent has been invoked!")
end

local publisher: Publisher = Publisher()

publisher.SampleEvent += OnInvoked
```

Listing 24: Declare and raise an event.

4.8 Attributes

Attributes are commands within the code that allow for customization of the Lua++ compiler's behavior. To use an attribute one would use the `--!` prefix followed by the name of the attribute. These attributes must be the first statement of the program. Otherwise, the compiler will throw an error. Here are some examples:

```
--!<attribute>
print("hello")
```

Listing 25: Correct usage of an attribute.

The following segment will not compile successfully as there is a statement preceding the attribute definition:

```
local v = 1 -- NO
--!<attribute>
print("hello")
```

Listing 26: Incorrect usage of an attribute.

4.8.1 The "Lenient" Attribute

Developers simply interested in using the Lua++ compiler can use `--!lenient` to disable the strict type checker. Although using this attribute is strongly discouraged as it will make code management much more difficult, it could be helpful when looking to compile basic Lua scripts that can be run on the Lua++ VM.

4.8.2 The "Comment" Attribute

As mentioned in the postfix section, traditional Lua comments prevent the implementation of the decrementation operator (`--`). To disable the traditional single-line Lua comment, one can use `--!comment`. Once the compiler sees that this attribute is specified, single-line comments will transition over to the `#` sign and the `--` operator will replace `---` for decrementation.

Below is a code segment that would yield an error at compilation after applying the `--!comment` attribute.

```
--!comment

-- this is an invalid comment (and code)
local v = 1
```

```
print(v---)
```

Listing 27: Incorrect usage of the –!comment attribute.

The following code segment demonstrates the correct usage of the `--!comment` attribute and the syntax change it produces.

```
--!comment

# Valid new comment
local v = 1
print(v--)
```

Listing 28: Correct usage of the –!comment attribute.

An alternative to this approach of managing the conflicting comment and decrement operator issue would be to use reassignments (`var = var - 1`) or compound assignments (`var -= 1`).

5 Semantics

Despite it being a type-oriented programming language, Lua++ has exactly the same number of primitive types as Lua. However, the key benefit of Lua++ is that it separates arrays from tables so that code is much easier to follow. Lua++ has six primitive types: `number`, `string`, `boolean`, `Table`, `Array`, and `Function`. The last three primitive types `Table`, `Array`, and `Function` are capitalized unlike `number`, `string`, and `boolean` because they all have class-like behaviors.

5.1 Number

`Number` type can hold double-precision floating-point values of up to 15 digits taking up a space of 8 bytes (64 bits) in memory. The range of values that can be stored in this primitive is $1.79 \cdot 10^{-308}$ to $1.79 \cdot 10^{308}$. To optimize execution times, the compiler will identify whether the value assigned to a number primitive can be represented as a smaller value and will omit the respective instruction. Below is a simple example of the number primitive in use:

```
local intMax: number = 2,147,483,647

print("MAX_INT: " .. intMax)
```

Listing 29: Usage of the number primitive type.

5.2 String

The string primitive remains functionally unchanged from the Lua version, as it still represents a string of characters. It is unlimited in size and can be concatenated using the `..` operator mentioned.

```
local myString: string = "Lua++ is the best language ever."

print("A random fact: " .. myString)
```

Listing 30: Usage of the string primitive type.

5.3 Boolean

Just like the string primitive, the boolean primitive remains unchanged from the Lua version. A boolean represents a value that can either be `true` or `false`.

```
local myBool: boolean = true
print("Lua++ is the best language: ", myBool)
```

Listing 31: Usage of the boolean primitive type.

5.4 Table

The table primitive has undergone a significant change from Lua, which is used for both arrays and dictionaries. In addition, since objects are not primitive types in Lua, tables were used to emulate classes and inheritance. In Lua++, tables are simply dictionaries of key and value pairs. To retrieve a value, you simply index the table instance like you would with an array: `["<your key>"]`.

```
-- KEY: Name, VALUE: Id
local users: Table<string, integer> = {
  { "John Doe", 17362 },
  { "Jane Doe", 99999 }
}

print("Name: John Doe", "Id:", users["John Doe"])

users["Jane Doe"] = 0
print("Name: Jane Doe", "Id:", users["Jane Doe"])
```

Listing 32: Usage of the Table primitive type.

The output for the above code:

```
Name: John Doe Id: 17362
Name: Jane Doe Id: 0
```

To improve performance, Lua++ uses one of the fastest non-cryptographic hash table algorithms, SpookyHash, which is twice as fast as lookup3, the algorithm used by Luau and LuaJIT. The original Lua uses the simplest hash function, called modulo, which suffers from key collisions and in some cases can even lead to failed implementations [5].

5.5 Array

As previously mentioned, Lua has no distinct type for arrays as they are approximated with tables. In Lua++, arrays are an independent data structure that consists of a list of values. To get a value within this array, you can index the instance with the zero-based position of that value in the list:

```
local list: Array<number> = { 1, 2, 3 }

-- You can easily iterate through them, there is no need to
-- call ipairs/pairs
for n in list do
 print("List item: " .. n)
end

print()

for i = 0, 3 do
 print("List item: " .. list[i])
end
```

Listing 33: Usage of the Array primitive type.

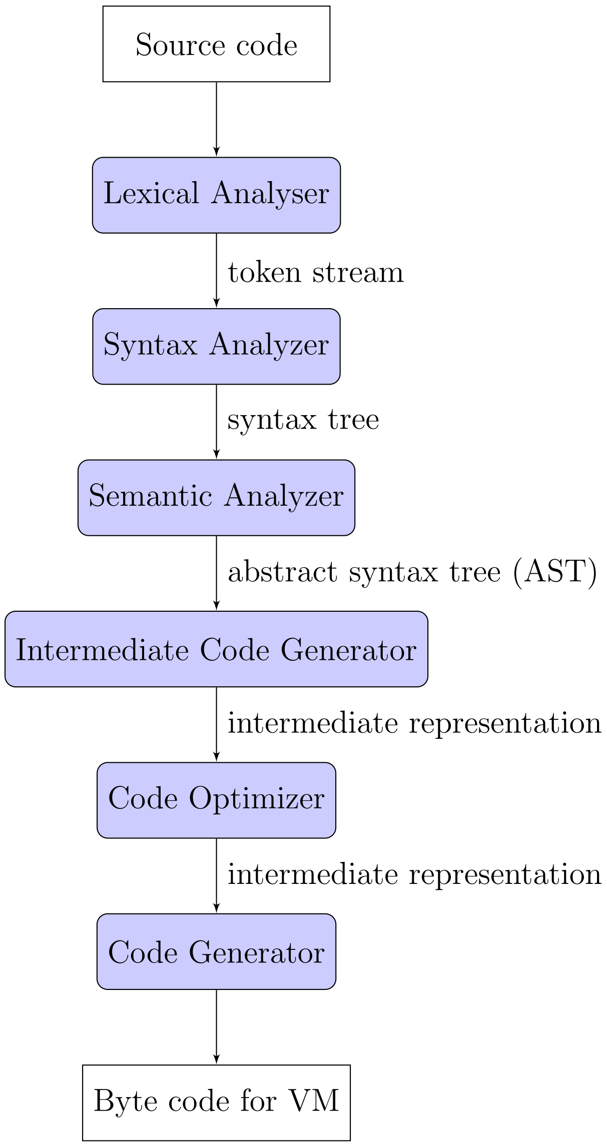
The output for the above code:

```
List item: 1
List item: 2
List item: 3

List item: 1
List item: 2
List item: 3
```

6 Compiler

`Luappc` (Lua++ compiler) is a multi-pass compiler designed to alleviate the VM of as many checks and calculations as possible. In this way, the Lua++ VM can be as simple as possible minimizing the size of the executable and optimizing the speed of a script. Unlike the single-stage Lua compiler, `luappc` compiles code in six different stages: lexical analysis, syntax analysis, semantic analysis, intermediate representation (IR) generation, code optimization, and code generation.



6.1 Lexical Analysis

In this stage, the script is converted into a sequence of lexical tokens. Each token is assigned a start and end location in the script (line:column). Take the following script and its output after passing it through the lexical stage as an example:

```
local v: number = 1

LOCAL_T                                     0001:0001-0001:0005
IDENTIFIER_T        v                       0001:0007-0001:0007
COLON_T                                     0001:0008-0001:0008
TNUMBER_T                                   0001:0010-0001:0015
EQUAL_T                                     0001:0017-0001:0017
NUMBER_T            1.000000                0001:0019-0001:0019
```

Listing 34: Lexical analysis example output.

6.2 Parsing

After the script has been passed through the lexer, the parser takes the list of tokens and builds an Abstract Syntax Tree (AST) of nodes containing statements and expressions found within the script. The parser also verifies that the script is "grammatically" correct by inspecting each token. Similar to the lexer, the parser has been made with a parser generator called bison (yet another compiler-compiler).

6.3 Abstract Syntax Tree (AST)

In an AST, each statement is a base node (independent) while expression nodes stem from the statement they were used in.

The flow graph in Figure 1 depicts the AST for a simple numerical assignment like (`local a: number = 1`). The root node labeled as program represents a list of statements. In this case, there is only one statement, but there can be infinitely many. Type annotations act as a pair, the first value contains the identifier, and the second value, highlighted in green, represents the type the identifier has been annotated with.

Abstract syntax trees can get complex very quickly. Consider a relatively simple function with two arguments returning a `boolean`. After the parsing stage, the compiler will construct an AST shown in Figure 2.

```
local function f(a: number, b: string): boolean
    print(a)
```

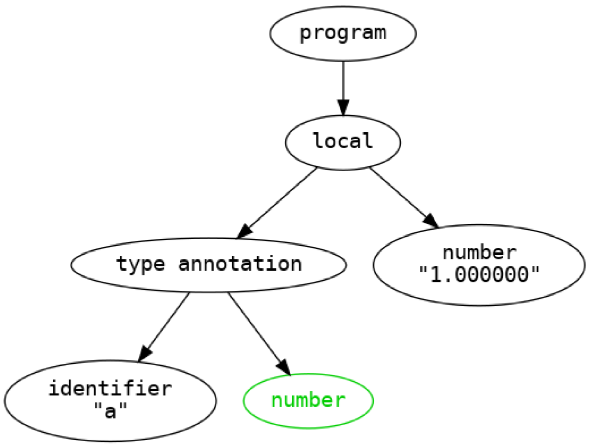


Figure 1: AST for a simple numerical assignment

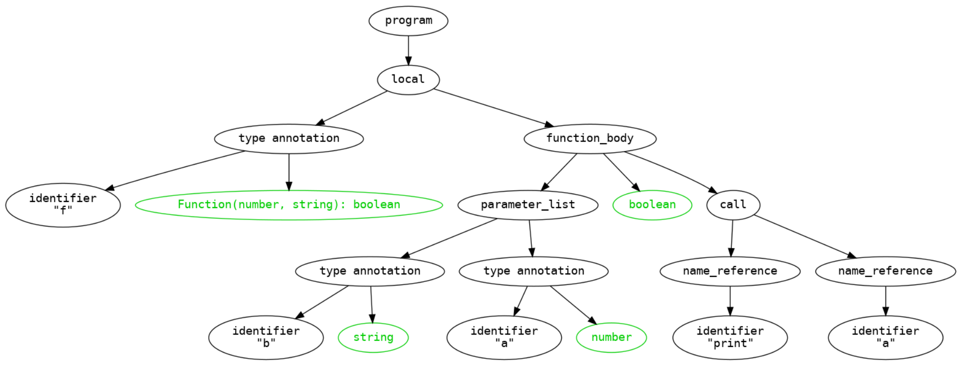


Figure 2: A more complex AST for a function

```
end
```

Listing 35: Simple function results in a larger AST

Since functions can be used as expressions, function definitions are processed as if they were local or global assignments. Therefore, we have a local definition of type `Function(number, string): boolean`.

6.4 Type Analysis

Since Lua++ is a type-oriented programming language, it was vital to implement an efficient system for comparing and assigning types. The type checker traverses through the Abstract Syntax Tree generated by the parser and ensures that each data type is used consistently with its definition. The compiler is also natively in strict mode, so the type checker will throw an error if the script does not use type annotations, inhibiting the compilation of vanilla Lua scripts. To disable this for backward compatibility with classic Lua, one can use the lenient attribute which is described in section 4.8.1.

6.5 Intermediate Representation (IR)

An intermediate representation (IR) is the data structure or code used internally by a compiler or virtual machine to represent source code. `luappc` converts the abstract syntax tree from the parser into a structural representation of the respective bytecode for the script. This representation is then processed and optimized. Here is the IR output of a simple `print("Hello, world!")` script:

```
symbol table:
  Hello, world          0
  print                 1

-----------------------------------------------------------------

proto->is_vararg          true
proto->parameters_size       0
proto->max_stack_size        2

[0001]     VARARGPREP         0 0 0
[0002]     GETENV             0 1
[0003]     LOADK              1 2
[0004]     CALL               0 2 1
[0005]     RETURN             0 1 0

constants:
[0]   string { 1 }
[1]   global { k(0) }
[2]   string { 0 }
-----------------------------------------------------------------
```

Listing 36: Sample output of the Intermediate Representation

Here we can see all the properties of the IR: symbol table, function prototype information, instruction list, and constant pool. As of the publication of this paper, significant optimizations have not been applied to the IR, although there are plans for it in the future.

6.6 Code Generation

Code generation is the simplest stage of the compiler. It takes the final version of the IR and converts it into a string of bytes, also known as bytecode, so that the VM or any other external program can pick up and process the information of the script [1].

Currently, the code generation stage only produces bytecode for the VM, but there are plans for native code generation. The goal is to implement code generation support for x86 and x64 assembly as this would allow developers to create Lua++ applications specific to popular operating systems. Other assembly formats will be implemented based on user demand.

Once enabled, the code generation stage will emit debug info, making it much easier to track your code flow and identify any issues preventing it from running successfully.

7 Virtual machine

Unlike the compiler, the Lua++ virtual machine is a heavily modified and optimized version of the standard Lua 5.1 virtual machine. It still lifts the bytecode into a structural representation and executes each instruction individually, but has been modified to support the new and improved instruction format.

The Lua VM was kept as a template in order to retain the Lua API for compatibility with existing C libraries.

7.1 Chunk

The “chunk” is a container of function prototypes to be executed in the code. A regular Lua chunk contains a header section with the Lua version and other useful information for the VM. A Lua++ chunk consists of a header section preceded by a global list of identifiers (symbol table) used within the code (see Figure 3). In basic Lua bytecode, the identifiers are stored with the constant sections in each function prototype. Adding a global list minimizes the bytecode size as there are no duplicate identifier entries.

7.2 Instructions

As in Lua, each Lua++ instruction is represented by a 32-bit unsigned integer with the first 8 bits representing the operation the instruction is trying to perform. Each instruction has one to three arguments containing values

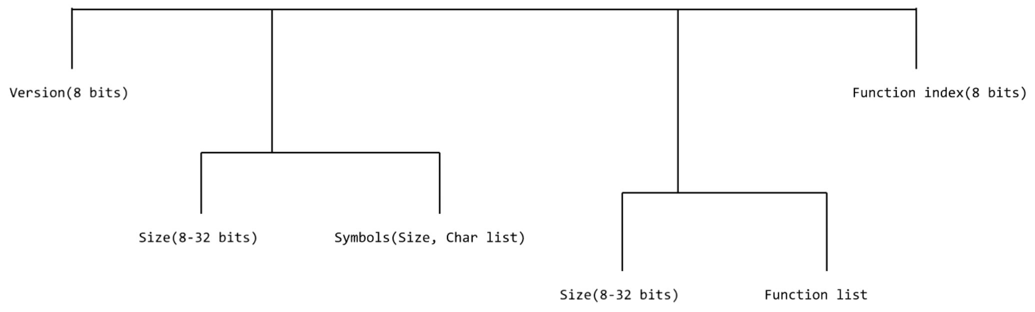


Figure 3: Chunks tree

or other important information about the operation. Lua++ has three instruction formats: `iABC`, `iAD`, `iE` where i represents the operation. The table below shows the structure of each different format:

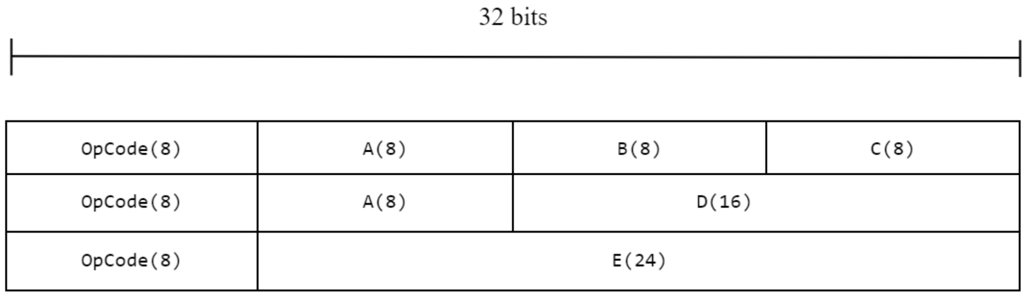


Figure 4: Three instruction formats

This variety in formats is needed because some operations require large operands. Examples include loading numbers into a register or making large-scale instruction jumps in the program. Sometimes, even the instruction format with the largest operand, `iE` will not suffice. Consequently, some instructions are followed by an extra instruction representing an additional 32-bit argument.

8 Optimizations

The Lua++ framework has a series of optimizations that improve the execution speed of a typical Lua++ script. These include specific optimizations to the compiler, instruction formats, and virtual machine.

8.1 Compiler Optimizations

The majority of Lua++'s optimizations are installed in the compiler. After the Intermediate Representation is generated, a series of optimizations are applied to it to maximize performance. As of now, there is no SSA (single static assignment) [1], but there are plans to implement it in the future.

Unlike some of today's simpler compilers, Lua++ does not optimize the AST generated by the parser but instead analyzes the IR function prototypes. This way the compiler has the ability to optimize not only the control flow of a script but also the individual instructions, making it easier to perform complex optimizations.

8.1.1 Constant Propagation

Constant variables or variables whose values are never reassigned or modified are propagated through the flow graph and substituted at the use of the variable.

In the code segment below the value of `x` can be propagated with x:

```
local x: number = 1
print(x)
```

Listing 37: Constant propagation example

Below is the code segment above after undergoing constant propagation:

```
print(1)
```

Listing 38: Constant propagation example applied

8.1.2 Constant Folding

Expressions with constant operands can be evaluated at compile time, significantly improving runtime performance. Constant folding can be especially useful when paired with constant propagation.

In the code fragment below, the expression `(2 + 3)` can be evaluated as `5`.

```
local function f(): number
    return (2 + 3)
end
```

Listing 39: Constant folding example

Below is the code segment above after undergoing constant folding:

```
local function f(): number
    return 5
end
```

Listing 40: Constant folding example applied

8.1.3 Instruction Combining

At a source code level, this optimization takes two redundant statements and combines them into one. At the IR (intermediate representation) this optimization represents the combination of two instructions. This optimization is applied to all operators that actively change the value of a variable.

In the code fragment below, the two post-increment statements can be combined into one statement.

```
local i: number = 1

i++
i++
```

Listing 41: Instruction combining example

Below is the code segment above after combining the two post-increment statements:

```
local i: number = 1

i += 2
```

Listing 42: Instruction combining example applied

8.1.4 Dead Code Elimination

Arguably one of the most important optimizations for a modern-day compiler is dead code elimination. Code that is unreachable or does not affect the program can be eliminated.

In the code fragment below, the value assigned to `i` is never used and can be eliminated. The first assignment to `g` is dead and the last assignment is unreachable so they both can be eliminated.

```
local g: number = 1

local function f()
    local i: number = 1      -- Dead store
    g = 1                    -- Dead store
    g = 2
    return
    g = 13                   -- Unreachable
end
```

Listing 43: Dead code elimination example

Below is the code segment above after dead code elimination:

```
local g: number = 1

local function f()
    g = 2
end
```

Listing 44: Dead code elimination example applied

8.1.5 Integer Divide Optimization

Integer division is usually slower or much slower than other operations like addition and bit shifting. Divide expressions with power-of-two denominators or other special bit patterns can be replaced with faster instructions.

In the code segment below the instruction associated with the integer division can be replaced with a bit-shift instruction instead.

```
local function f(n: number): number
    return (n / 2)
end
```

Listing 45: Integer divide example

There is no optimized segment as there is no bitshift operator in Lua++.

8.1.6 Expression Simplification

Some expressions can be simplified by replacing them with an equivalent expression that is much more efficient. Expression simplification is often paired with constant folding.

The code fragment below contains several examples of expressions that can be simplified:

```
local function f(i: number): Array<number>
    return {
        i + 0,
        i * 0,
        i - i,
        1 + i + 1
    }
end
```

Listing 46: Expression simplification example

Below is the code fragment after expression simplification:

```
local function f(i: number): Array<number>
    return {
        i,
        0,
        0,
        i + 2
    }
end
```

Listing 47: Expression simplification example applied

8.1.7 Loop Fusion

Some adjacent loops can be fused into one loop to reduce loop overhead and improve run-time performance. Loop fusion is only applied when the two loops are independent of each other.

The two adjacent loops on the code fragment below can be fused into one loop.

```
local a: Array<number> = {}
local b: Array<number> = {}

for i = 0, 100 do
    a[i] += 1
end

for i = 0, 100 do
    b[i] += 1
end
```

Listing 48: Loop fusion example

Below is the code fragment after loop fusion:

```
local a: Array<number> = {}
local b: Array<number> = {}

for i = 0, 100 do
    a[i] += 1
    b[i] += 1
end
```

Listing 49: Loop fusion example applied

8.1.8 Hoisting

Loop invariant expressions are removed from loops, thus improving runtime performance by executing the expression only once, not every iteration of that loop.

In the code segment below, the expression `(x + y)` is loop invariant, meaning it can be removed or 'hoisted' out of the loop.

```
local function f(x: number, y: number): Array<number>
    local a: Array<number> = {}

    for i = 0, 100 do
        a[i] = x + y
    end

    return a
end
```

Listing 50: Hoisting example

Below is the code fragment after hoisting:

```
local function f(x: number, y: number): Array<number>
    local a: Array<number> = {}
    local v: number = x + y

    for i = 0, 100 do
        a[i] = v
    end

    return a
end
```

Listing 51: Hoisting example applied

8.2 Virtual Machine Optimizations

Just like in the compiler, the Lua++ VM also has a series of performance boosting optimizations. The majority of said optimizations are in the byte-code instruction formats, making it much easier to access values associated with certain instructions. These optimizations are not described here as they are out of the scope of this paper.

The Lua++ VM does not implement any sort of JIT compilation, as the VM would become inflated which makes it impossible to implement on machines with restricted storage space. Although, it may come to future builds of the language, depending on the demand from future users.

9 Performance

As mentioned throughout the paper, Lua++ is significantly faster than Lua. To prove this, a series of common benchmarks were run on the Lua++ VM and the Lua interpreter. These results have shown that Lua++ is 4 to 25 times as fast as classic Lua.

The table below shows four benchmarks run to come to the conclusions mentioned above. The 'factor' column represents how many times faster Lua++ was on this benchmark then Lua.

Table 2: Lua++ performance versus classic Lua

Benchmark	**Description**	**Factor**
md5	The calculation of 20,000 MD5 hashes.	24
recursive fibonacci	Simulation of the Fibonacci sequence up to 40 nodes.	8
k-nucleotide	Analysis of 5,000,000 nucleotide strings	7
binary trees	Allocation and deallocation of 16 binary trees	4

The benchmarks were all run on an x86/x64 Intel processor architecture on Ubuntu 22.04.1. All of the benchmarks are single-threaded, meaning they were all run on one core.

One of the more extensive of these benchmarks is the *recursive Fibonacci* benchmark. It tests to see how fast your programming language can handle many recursive calls. Specifically, it measures the performance of your stack, which in the case of Lua++, is located in the VM. The code for the test is displayed below:

```
local function fib(n: number): number
    if n <= 1 then return n end
    return fib(n - 1) + fib(n - 2)
end

print(fib(47))
```

Listing 52: Recursive Fibonacci benchmark

The majority of the performance improvements come from virtual machine optimizations as the code has already been optimized by the programmer giving the compiler nothing else to optimize. What most likely benefited the benchmarks the most was the VM ”pre-loading” environment variables. Library functions like `print()` have their C functions looked up via a hash table when the bytecode for the current script is processed. This way the VM is much more efficient during execution.

10 Integration with other languages

Lua++ can be embedded in other programming languages thanks to its flexible API and libraries. We are presenting a few examples in C, however, integration with Python and Java works substantially the same way.

The frontend of the Lua++ API has been developed to be nearly identical to the Lua API, making it much easier to port previous integrations to Lua++.

To integrate Lua++ into your projects you will need to depend on the VM and it's API which are included in `luapp.h`. You will need to first have the bytecode of a program at hand, which can be loaded using `luapp_load()`, or if you are loading a file you could use `luau_loadfile()`. The code segment below displays an example of such an implementation:

```
lua_State *L = lua_newstate();

/* Load a file of bytecode */
if (luapp_loadfile(L, "script.bin") != LUAPP_OK) {
    lua_close(L);
    return 1;
}
```

Listing 53: Load bytecode into environment

For more details on how to use the API, refer to the Lua 5.1 documentation or any Lua 5 version. There have been no changes to the API other than a few backend performance tweaks which do not pertain to this section.

11 Status and Future Work

As of the publication of this paper, Lua++ is open source, hosted on GitHub: `https://github.com/luapp-org/luapp` and an official website for the project can be found here: `https://luaplusplus.org`. Pull requests are accepted and welcome. Also, contributions from the developer community would be very much appreciated.

Although close to completion, this project remains in development as not all the features mentioned in this paper have been implemented, like native code generation.

The objective is ensure that Lua++ is not limited it to a certain field of programming, but rather to ensure that it is applicable almost everywhere, much like Python. To achieve this goal, a series of libraries will be implemented including a complete HTTP and WebSocket library, providing a comfortable interface for network communication as well as a potential a WinForms library, allowing developers to create desktop applications for the Windows operating system.

Feedback and suggestions from potential users would be greatly appreciated and can be provided at the official Lua++ discussion board on GitHub: `https://github.com/orgs/luapp-org/discussions`.

12 Conclusion

In the development of Lua++, almost all significant shortcomings of classic Lua were addressed. Most importantly, by creating an efficient compiler and a compact virtual machine, we were able to achieve a four to 24-time increase in speed of comparable performance benchmarks. Should Lua++ achieve adoption on par with Lua today, deployment would result in significant reduction in power consumption of running Lua code.

Increased efficiency and small resource footprint also allow Lua++ to be successful on various low-power devices, such as household and industrial sensors (IoT), smartwatches, fitness trackers, various appliances and so on. Anything that has a small microprocessor is usually developed for using C, so such devices would benefit from the availability of a fast high-level language like Lua++.

On the other end of the spectrum, fully-fledged object-oriented programming features of Lua++, along with optional strict typing, make the language suitable for the development of much larger applications by larger teams. Therefore, Lua++ breaks out of a niche occupied by its predecessor and has appeal for a wide range of applications.

Consequently, Lua++ is the optimal application development solution for anyone wishing to retain all the benefits of Lua but also have the ability to run their code in a modern programming environment at the speed and efficiency close to that of today's fastest programming languages.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, August 2006.

[2] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[3] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard E. Jones, editor, *ECOOP*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 2014.

[4] Kellen Browning. Where has your tween been during the pandemic? on this gaming site. `https://www.nytimes.com/2020/08/16/technology/roblox-tweens-videogame-coronavirus.html`, August 2020. online; accessed on 2020-08-16.

[5] Luyu Huang. Beware of hash collisions in lua tables. `https://luyuhuang.tech/2021/07/30/hash-collision.html`. online; accessed on 2022-10-28.

[6] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.1 Reference Manual.* Lua.org, 2006.

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, N.J., 1988.

[8] Roblox. Luau. `https://luau-lang.org/`. online; accessed on 2022-10-28.

[9] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[10] Elie K. Track, Nancy Forbes, and George O. Strawn. The end of moore's law. *Comput. Sci. Eng.*, 19(2):4–6, 2017.

[11] Wikipédia. Jit — wikipédia, a enciclopédia livre, 2011. [Online; accessed on 10-May-2011].

List of Figures

List of Tables

Listings