Decoding Lua: Formal Semantics for the Developer and the Semanticist

Mallku Soldevila
FAMAF, UNC and CONICET
Argentina
mes0107@famaf.unc.edu.ar

Beta Ziliani
FAMAF, UNC and CONICET
Argentina
bziliani@famaf.unc.edu.ar

Bruno Silvestre
INF, UFG
Brazil
brunoos@inf.ufg.br

Daniel Fridlender
FAMAF, UNC
Argentina
fridlend@famaf.unc.edu.ar

Fabio Mascarenhas*
DCC, UFRJ
Brazil
fabiom@dcc.ufrj.br

Abstract

We provide formal semantics for a large subset of the Lua programming language, in its version 5.2. We validate our model by mechanizing it and testing it against the test suite of the reference interpreter of Lua, obtaining evidence that our model accurately represents the language.

We target both a PL semanticist —not necessarily versed in Lua—, and a Lua developer —not necessarily versed in semantic frameworks. To the former, we present the peculiarities of the language, and how we model them in a *modular* small-step operational semantics, using concepts from Felleisen-Hieb's *reduction semantics* with evaluation contexts. Moreover, we mechanize and test the model in PLT Redex, the *de facto* tool for reduction semantics.

To the reader unfamiliar with such concepts, we provide a gentle introduction to the model. It is our hope that developers of the different Lua implementations and dialects understand the model and consider it both for testing their work and for experimenting with new language features.

CCS Concepts • **Theory of computation** → **Operational semantics**; • **Software and its engineering** → **Semantics**;

Keywords Lua, Operational Semantics, PL Formalization.

ACM Reference Format:
Mallku Soldevila, Beta Ziliani, Bruno Silvestre, Daniel Fridlender, and Fabio Mascarenhas. 2017. Decoding Lua: Formal Semantics for the Developer and the Semanticist. In *Proceedings of 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3133841.3133848

1 Introduction

Lua is a lightweight imperative scripting language, featuring dynamic typing, automatic memory management, data description facilities, and metaprogramming mechanisms to adapt the language to specific domains [15]. The typical use case of a Lua application is as an extension library embedded in a *host* application, commonly written in C or C++. In that setting, Lua offers the possibility to add scripting facilities to the host application, combining the flexibility and rapid prototyping of a dynamic language within the static guarantees and optimizations of stricter programming languages.

Lua is extensively used in many diverse applications, ranging from game development, most notably by "AAA" games [5] but also in mobile games and game frameworks, plugin development (for example, in the photo editing software Adobe Photoshop Lightroom[1], and the type-setting system LuaTex[2]), web application firewalls[3], and embedded systems[4].

Lua is informally specified by both its reference manual and its reference interpreter, developed and maintained by the core Lua authors. Thanks to Lua's success, several alternative implementations[5], as well as code linters and static analyzers[6] can be found in the wild. However, the informal nature of the specification means that developers of those tools must resort to their intuition, formed by study of the reference manual, inspection of the source code of the interpreter, and experimentation.

In this work, we present a comprehensive formalization of (most of) Lua 5.2, which we argue will facilitate the development and testing of these alternative implementations

*Supported by a grant from CNPq (Universal 447727/2014-1).

[1] http://www.adobe.com/devnet/photoshoplightroom.html
[2] http://www.luatex.org/languages.html
[3] https://blog.cloudflare.com/cloudflares-new-waf-compiling-to-lua
[4] https://www.lua.org/uses.html
[5] http://lua-users.org/wiki/LuaImplementations
[6] http://lua-users.org/wiki/ProgramAnalysis

DLS'17, October 24, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5526-1/17/10…$15.00
https://doi.org/10.1145/3133841.3133848

and analysis tools, as well as the prototyping of new features and extensions to the Lua language.

The formalism that we use to express the semantics of Lua is mainly a small-step operational semantics with evaluation contexts. Evaluation contexts, taken from Felleisen-Hieb's reduction semantics (FH) [2], are used for the specific purposes of modularization, for providing a concise description of the context sensitive semantics, and to define the execution order. In this, we follow the path taken by [3, 13, 14], where FH is similarly applied for successfully formalizing real programming languages. However, from a technical point of view, we depart from the aforementioned works and introduce the following particularities:

- We emphasize the distinction between what constitutes the language we want to model, and what are the *run-time constructs*; the extra pieces of information required to model the semantics. Maintaining this distinction eases the presentation of the model, in particular for readers familiar with FH as presented in [2].
- For similar reasons, we separate the notion of a store, as presented in the text, from its mechanization, following a more traditional textbook approach.
- For better understanding and *trusting* the model, we reduce the complexity of the *desugaring* process, by staying as close as possible to the source language. See §6 for more on this topic.

While providing the semantics on paper of the most interesting parts of the language is an important contribution, it does not suffice to ensure that our characterization of the language is correct. For this reason we mechanize the semantics in PLT Redex [2], following the success of previously mechanized semantics for other scripting languages [3, 14].

We tested the mechanization of our formal semantics of Lua against the test suite of the reference interpreter, successfully passing every test within the scope of the formalization. We take this as strong evidence to support the claim that our semantics is a sound representation of the selected subset of the language's features, including:

- Every type of Lua value, except *coroutines* and *userdata* (see below);
- Metatables;
- Identity of closures;
- Dynamic execution of source code;
- Error handling;
- A large collection of the services of the standard library.

We purposely left out the following features for future work:

- Coroutines, in essence single-shot delimited continuations;
- Userdata, opaque handles to data from the host application and native libraries;
- Garbage collection;
- The **goto** statement;
- Services from the standard library that interface with the operating system, such as file manipulation, or have large complex C implementations, such as string pattern matching.

```
1  local function memoize(fn)
2    local t = {}
3    return function(x)
4      local y = t[x]
5      if y == nil then y = fn(x); t[x] = y end
6      return y
7    end
8  end
9
10 local memsum = memoize(function(x)
11   local a = 1
12   for i = 1,x do a = a + i end
13   return a
14 end)
```

Figure 1. Memoization in Lua.

The mechanization can be downloaded from https://github.com/Mallku2/lua-redex-model.

The rest of the paper is organized as follows: §2 presents a very brief description of Lua, with emphasis on some of the features that we formalize in later sections; §3 presents the basic concepts that our formalization uses, via a formalization of a very small subset of Lua; §4 expands §3 with the formalization of most important and original parts of our complete semantics; §5 briefly discusses the mechanization and its tests; §6 discusses related work; finally, §7 summarizes our contributions and discusses future avenues of research.

2 Lua, an Extensible Scripting Language

We organize our presentation of Lua around the examples of memoization and object-oriented programming, shown in figures 1 and 2, respectively. They serve to introduce several characteristics of Lua: its syntax, the versatility of its single data structure (*tables*), its metaprogramming mechanisms and some aspects of its scoping rules.

2.1 Memoization

The code[7] shown in Figure 1 implements a memoization function, memoize, which takes a function fn as argument and returns its *memoized* version. The values of fn already computed will be stored in a *table* (t in line 2). At their core, tables are associative arrays that can be indexed with any Lua value except **nil**. We will show later in this section that tables also come with syntax sugar and metaprogramming facilities that can greatly extend their functionality beyond simple associative arrays.

Line 3 is where the memoized version of fn is returned through an anonymous function. This function takes x as argument and, before computing fn(x), performs a look-up in

[7] Taken from http://lua-users.org/wiki/FuncTables.

```
1  local  MyClass = {}
2  MyClass.__index = MyClass
3
4  function MyClass.new(init)
5    local  self  =  setmetatable  ({},   MyClass)
6     self . value  =  init
7    return  self
8  end
9
10 function MyClass:set_value(newval)
11    self . value  =  newval
12 end
13
14 function MyClass:get_value()
15   return  self . value
16 end
17
18 local  mc = MyClass.new(5)
19 print (mc:get_value () )     >> 5
20 mc:set_value (6)
21 print (mc:get_value () )     >> 6
```

Figure 2. OOP based on Lua's metatable mechanism.

the table for value x (line 4). If the result of the look-up is **nil** it means no result was found, so it proceeds to compute fn(x) and store it in the table (line 5). The resulting value, either computed or retrieved from the table, is returned in line 6. The function memoize is used in lines 10–14 to improve the performance of a function that performs a sum from 1 to x.

All procedures and functions in Lua, anonymous or named, are first-class values, and form lexically-scoped closures. The anonymous function that memoize returns will effectively capture into its scope the table t, as expected.

Note that the definitions of memoize, t, and memsum are prefixed by the keyword **local**. Without it, all of these declarations are simple assignments, and do not introduce new names in the current scope. In an assignment, if there is no variable in scope with that name, then the variable is global: the assignment will actually store its rvalue in a table called the *environment*, with a string containing the variable's name as the key. Using a variable that is not in scope also looks up the variable in the environment.

The environment is available to the programmer through a *variable* _ENV, which is always in scope. This means that any occurrence of a variable x that is *not* in scope is just syntax sugar for _ENV["x"]. Since it is a variable, the programmer can change the environment at will by simply assigning another table to _ENV.

2.2 Simple OOP in Lua

Another interesting example[8] is listed in Figure 2. It presents the implementation of some basics concepts of object-oriented programming, namely classes and objects, by combining tables, first-class functions, and the *metatable* mechanism. It also presents some syntax sugar provided by Lua to better support OOP.

In Lua, a class is essentially implemented as a dictionary (*e.g.*, table), in which the method names form the keys of the dictionary, and the method implementations are the associated values. Objects are also modelled with tables, containing the fields and their values.

In the example, we have a class MyClass with its corresponding constructor (line 4) and only one field value with its setter (line 10) and getter (line 14). The function declarations in these lines are actually syntax sugar for assignments, where the left-hand sides are, respectively, MyClass["new"], MyClass["set_value"], and MyClass["get_value"]. For the two methods on line 10 and line 14 the use of : instead of . also includes an extra first parameter for these functions, named self.

In the last lines of Figure 2 we show how to create an instance of MyClass (line 18), and how to invoke the methods. In line 20 we can observe the invocation of set_value with yet another syntax sugar: mc:set_value(6) is equivalent to mc["set_value"](mc, 6).

If classes contain methods, and objects contain fields, how is mc["set_value"] looking up the set_value method? The answer is the *metatable* mechanism, used in lines 2 and 5. In line 5, the call to setmetatable assigns MyClass as the metatable of the empty table {} passed as argument, and then returns this empty table.

A metatable can modify the behavior of a table with regards to most of Lua's operations. For this example, the behavior we are modifying is look-up of non-existing keys. Each behavior that can be modified has an associated handler. For look-up of non-existing keys the handler is called __index (line 2). A handler is usually a function, but in the case of __index it can be another table, in this case MyClass. A non-existing key then will be looked up in this table, and this is how mc["set_value"] results in the method set_value from MyClass.

Lua also specifies handlers for setting a non-existing key, for calling a value as if it were a function, for most of the binary and unary operators, for setting finalizers, and even for some functions in the standard library. Lua programmers typically use metatables for object-oriented programming (including more elaborated object models than class-based single inheritance), for operator overloading, and for proxies.

3 Basics of the Formalization

In this section, we gently introduce the semantic framework used throughout the paper by providing semantics to a small subset of Lua. Essentially, we mix classical ideas from operational semantics based on abstract machines —the description of a programs' execution by abstractly representing run-time constructions and their evolution during an

[8] Taken from http://lua-users.org/wiki/ObjectOrientationTutorial .

s ::= **if** e **then** s **else** s **end** | ;

v ::= **nil** | *bool_literal*

e ::= v | e *binop* e | *unop* e

binop ::= **and** | **or**

unop ::= **not**

Figure 3. Syntax of simple statements and expressions

$$\frac{v \notin \{\textbf{nil}, \textbf{false}\}}{\textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{end} \rightarrow^{s} s_1}$$

$$\frac{v \in \{\textbf{nil}, \textbf{false}\}}{\textbf{if}\ v\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{end} \rightarrow^{s} s_2}$$

Figure 4. Semantics of the conditional statement.

$$\textbf{not}\ v \rightarrow^{e} \delta(\textbf{not}, v) \qquad \frac{op \in \{\textbf{and}, \textbf{or}\}}{v\ op\ e \rightarrow^{e} \delta(op, v, e)}$$

Figure 5. Semantics of expressions.

$$\delta(\textbf{and}, v, e) = \begin{cases} v & \text{if } v = \textbf{false} \lor v = \textbf{nil} \\ e & \textbf{otherwise} \end{cases}$$

$$\delta(\textbf{or}, v, e) = \begin{cases} v & \text{if } v \neq \textbf{false} \land v \neq \textbf{nil} \\ e & \textbf{otherwise} \end{cases}$$

$$\delta(\textbf{not}, v) = \begin{cases} \textbf{true} & \text{if } v = \textbf{false} \lor v = \textbf{nil} \\ \textbf{false} & \textbf{otherwise} \end{cases}$$

Figure 6. δ function: boolean operators.

execution— together with reduction semantics with evaluation contexts [2], formalism from which we take the tools for modeling continuations, and to obtain a modular description of the semantics from simple computations to the execution of complete programs.

An interesting aspect of Felleisen-Hieb's reduction semantics is the possibility of defining the semantics of a language by decomposing it into fragments, describing the fragment's semantics in isolation with a separate relation. For our small subset of the Lua language, we describe three fragments: *pure* statements, pure expressions (following Lua's distinction of statements and expressions), and *stateful* (*i.e.,* memory changing) statements. Then, we compose the three using a fourth relation, thus providing the semantics for entire programs.

We show the grammar for *stateless* programs in Figure 3. The statements are conditional branching and skip (denoted with ;). The expressions are **nil** (the absence of a useful value), boolean constants, and logical operators. Of course, we are not able to write any useful program. In the coming sections we will grow our language until we reach Lua.

Figure 4 introduces the typical operational semantics for the conditional statement, modeled with the $\rightarrow^{s}$ relation between stateless statements. The first rule states that, in a boolean context (the conditional of the **if**), any value different from **nil** and **false** is considered **true**, and therefore the **then** branch is considered. Note that we write above the line the conditions in which the rule applies. When no condition is required, the line will be omitted. The second rule states that, for **false** or **nil**, the **else** branch is considered.

$$\frac{\sigma' = (r, v), \sigma}{\sigma : \textbf{local}\ x = v\ \textbf{in}\ s\ \textbf{end} \rightarrow^{s\_\sigma} \sigma' : s[x \backslash r]}$$

$$\frac{\sigma' = \sigma[r := v]}{\sigma : r = v \rightarrow^{s\_\sigma} \sigma' : ;} \qquad \sigma : r \rightarrow^{e\_\sigma} \sigma : \sigma(r)$$

Figure 7. Semantics of variables and references.

Figure 5 gives the semantics of expressions using a separate $\rightarrow^{e}$ relation. We use an *interpretation* function δ, as seen in the literature, which provides meaning to operators using *denotational* style of semantics. In contrast to the relations over terms presented so far, denotational semantics are not tied to single computation steps. Figure 6 shows (a simplified version of[9]) the δ-equations for boolean operators.

We proceed now to extend the language with imperative features: (local) variables. Statements are enlarged with variable definition and assignment:

$$s ::= \ldots \mid \textbf{local}\ x = e\ \textbf{in}\ s\ \textbf{end} \mid x = e$$

In order to describe its operational semantics, we must introduce a model of the memory store. We model it as a partial function from a set of references to values, denoted as σ. We refer to σ as the "*values' store*", or simply *store.*

As for references, we will not force any specific representation, just ask them to satisfy some simple properties to ensure the relation modeling the semantics of variables stays decidable. More specifically, we ask the domain of σ (referred as to $\text{dom}(\sigma)$) to be a finite set, with elements that must be syntactically represented, but different from any other syntactic object in the language. We further assume it is always possible to obtain a fresh reference from the store. Whenever we write $(r, v), \sigma$ we assume r to be fresh for σ.

We extend the grammar of expressions with references:

$$e ::= \ \ldots \mid r$$

References, in contrast to all the language constructs we mentioned so far, do not belong to the Lua source language, *i.e.,* they cannot be written down by a developer. They are *run-time constructs*: constructions not expressible in the source language, which are related to run-time concepts and are made explicit for the purpose of obtaining an operational semantics of the language. We will see other examples of such constructs in the coming sections.

Figure 7 describes the semantics of the definition and assignment of local variables. We use a new $\rightarrow^{s\_\sigma}$ relation, which maps a pair of a store σ and a statement s to another pair of a new store σ' and the resulting statement s$'$.

As shown in the rule for the introduction of local variables, when the right side of the definition is a value v we put it in the store with a fresh reference r. Then, we replace each

[9] The actual equations use Lua's *parenthesized expressions*, introduced in 4.1.

$$\begin{aligned} E ::= {} & [\ ] \mid \textbf{if } E \textbf{ then } s \textbf{ else } s \textbf{ end} \\ & \mid \textbf{local } x = \mathrm{E} \textbf{ in } s \textbf{ end} \mid \\ & \mid x = E \mid E\ binop\ e \mid unop\ E \end{aligned}$$

Figure 8. Evaluation contexts.

$$\frac{e \ \rightarrow^{\mathrm{e}} \ e'}{\sigma : E[\![e]\!] \ \mapsto \ \sigma : E[\![e']\!]} \qquad \frac{s \ \rightarrow^{\mathrm{s}} \ s'}{\sigma : E[\![s]\!] \ \mapsto \ \sigma : E[\![s']\!]}$$

$$\frac{\sigma : s \ \rightarrow^{\mathrm{s\_}\sigma} \ \sigma' : s'}{\sigma : E[\![s]\!] \ \mapsto \ \sigma' : E[\![s']\!]} \qquad \frac{\sigma : e \ \rightarrow^{\mathrm{e\_}\sigma} \ \sigma' : e'}{\sigma : E[\![e]\!] \ \mapsto \ \sigma' : E[\![e']\!]}$$

Figure 9. Semantics of programs.

occurrence of variable x in the scope of the **local** statement by the new reference r.

An important property of this semantics is that variables are never free, as substitution will always replace them by references right before they would become free. This will have an impact on closure creation (see 4.2).

Returning to Figure 7, the second rule shows variable assignment, with $\sigma[r := v]$ denoting a store σ' such that $\mathrm{dom}(\sigma') = \mathrm{dom}(\sigma)$, where $\sigma'(r) = v$ and $\forall r' \in \mathrm{dom}(\sigma'), r' \neq r \Rightarrow \sigma'(r') = \sigma(r')$. Note that the assignment reduces to an empty statement ;, indicating that there is nothing else to do for this particular statement. The third and final rule shows that references appearing in expressions are always implicitly dereferenced.

We have already defined three different relations, each of them computing a bit of a program: $\rightarrow^{\mathrm{e}}$ computes an expression, $\rightarrow^{\mathrm{s}}$ an stateless statement, and $\rightarrow^{\mathrm{s\_}\sigma}$ an stateful statement. Now we are ready to combine the three to perform the execution of a full program. To that effect we define the $\mapsto$ relation. This relation will say exactly when each of the previously defined relations will trigger, at the same time defining the order in which statement or expression must be executed next.

Here is where *evaluation contexts* play a central role. They describe the syntax of the language with the addition of a new construction: a *hole*, usually denoted as [ ]. Evaluation contexts will play different roles in later sections, but for the moment the (only) hole in a program will be filled in with the next statement or expression to be executed.

Figure 8 defines the evaluation context E for the small subset of Lua we described so far. We can see from the definition the order we expect evaluation to take place: in an **if**, the guard must be evaluated first. In the definition of variables we evaluate the rvalue for the definition first. In a binary operation, we evaluate the left operand first[10].

Figure 9 defines the $\mapsto$ relation. Both $E[\![e]\!]$ and $E[\![s]\!]$ denote an evaluation context where the hole is filled with the respective expression or statement, if this yields a syntactically valid term of the language. If the evaluation context is well-defined, together with the relations that describes computation steps, there is a unique decomposition of a valid term into an evaluation context and a subterm, and this subterm will match one and only one of the semantic rules. The subterm that is filling the hole gives the current focus of the computation.

$$\begin{aligned} s &::= \ldots \mid \textbf{while } e \textbf{ do } s \textbf{ end} \mid \textbf{break} \mid s\ s \\ v &::= \ldots \mid number\_literal \mid string\_literal \\ binop &::= \ldots \mid strictbinop \\ strictbinop &::= + \mid - \mid * \mid / \mid \hat{} \mid \% \mid .. \mid < \mid \leq \mid > \mid \geq \mid == \\ unop &::= \ \ldots \mid - \mid \# \end{aligned}$$

Figure 10. Syntax of the remaining stateless subset.

$$\begin{aligned} s &::= \ldots \mid \textbf{\$iter } e \textbf{ do } s \textbf{ end} \mid \llparenthesis\ s\ \rrparenthesis_{label} \\ label &::= \textsc{Break} \end{aligned}$$

Figure 11. Run-time statements for **while** and **break**.

$$\begin{aligned} \textbf{while } e \textbf{ do } s \textbf{ end} &\rightarrow^{\mathrm{s}} \llparenthesis\ \textbf{\$iter } e \textbf{ do } s \textbf{ end}\ \rrparenthesis_{\textsc{Break}} \\ \textbf{\$iter } e \textbf{ do } s \textbf{ end} &\rightarrow^{\mathrm{s}} \textbf{if } e \textbf{ then } s \textbf{ \$iter } e \textbf{ do } s \textbf{ end} \\ &\qquad \textbf{else ; end} \\ ;\ s &\rightarrow^{\mathrm{s}} s \\ \llparenthesis\ E_{\mathrm{If}}[\![\ \textbf{break}\ ]\!]\ \rrparenthesis_{\textsc{Break}} &\rightarrow^{\mathrm{s}} ; \\ \llparenthesis\ ;\ \rrparenthesis_{\textsc{Break}} &\rightarrow^{\mathrm{s}} ; \end{aligned}$$

Figure 12. Semantics of stateless statements.

$$\begin{aligned} e &::= \ldots \mid \llparenthesis\ e\ \rrparenthesis_{label} \\ label &::= \ldots \mid \textsc{ArithWO} \mid \textsc{ConcatWO} \mid \textsc{OrdWO} \mid \ldots \end{aligned}$$

Figure 13. Run-time expressions for errors.

With all of the main ingredients in place, we are now ready to provide semantics to Lua.

4 A Formal Description of Lua

In this section, we describe the highlights of our formalization of the semantics of Lua, the main contribution of this work. §4.1 covers the stateless subset of the language, §4.2 covers the imperative subset, §4.3 describes the concepts added to support standard library services, §4.4 covers the semantics of metatables, and §4.5 wraps up with the semantics of complete programs and error handling.

4.1 Stateless Lua

We extend the *stateless* subset presented in §3 with while loops, breaks, composition of statements, and numbers and strings with their corresponding operations (Figure 10).

Correspondingly, we extend the relation $\rightarrow^{\mathrm{s}}$ with the semantics of the new statements (Figure 12). First, a while loop begins by wrapping the whole loop in a Break label, changing also the name from **while** to **$iter**. The purpose of the label is to mark the point in which a **break** should continue

[10] Our definition enforces left-to-right evaluation of expressions. Even if this is left unspecified in Lua’s reference manual, the two most popular implementations of Lua, the reference interpreter and LuaJIT (luajit.org), both evaluate expressions left-to-right.

$$\frac{op \in \{+, -, *, /, \hat{}\,, \%, <, \leq\} \quad v_1, v_2 \in number}{v_1\ op\ v_2 \rightarrow^{e} \boldsymbol{\delta}(op, v_1, v_2)}$$

$$\frac{\begin{array}{c} op \in \{+, -, *, /, \hat{}\,, \%\} \\ v_1 \notin number \vee v_2 \notin number \\ v_1 = \boldsymbol{\delta}(\textbf{tonumber}, v_1, 10) \in number \\ v_2 = \boldsymbol{\delta}(\textbf{tonumber}, v_2, 10) \in number \end{array}}{v_1\ op\ v_2 \rightarrow^{e} \boldsymbol{\delta}(op, v_1, v_2)}$$

$$\frac{\begin{array}{c} op \in \{+, -, *, /, \hat{}\,, \%\} \\ v_1 \notin number \vee v_2 \notin number \\ (\boldsymbol{\delta}(\textbf{tonumber}, v_1, 10) \notin number \vee \\ \boldsymbol{\delta}(\textbf{tonumber}, v_2, 10) \notin number) \end{array}}{v_1\ op\ v_2 \rightarrow^{e} \llparenthesis\, v_1\ op\ v_2 \,\rrparenthesis_{\textsc{ArithWO}}}$$

Figure 14. Semantics of stateless expressions.

the execution, and the renaming is necessary to avoid repeatedly unfolding a while and piling up labels. Labels and **$iter** are new run-time statements (Figure 11). Then, a loop marked with **$iter** is unfolded as usual, using the conditional to check the guard and perform a new iteration. In [3], the Break label is inserted when *desugaring* the code. Instead, we opt to execute code that is closer to what the developer wrote.

In a composition of two statements, when the one on the left is a skip (;), we continue with the second. More interestingly, when the execution finds a **break** inside a (labeled) block, the whole code is replaced with a skip, to signal the execution of the break has exited. This is achieved by using a new evaluation context E_{lf}, which represents a program in which no other labeled term occur (its definition is elided for brevity). By not having other labels, we know the one surrounding this context is the one we have to break. The last rule removes the label once the execution of a loop reached skip.

Having defined the semantics for statements, we turn our attention to expressions (Figure 14). For brevity we focus only on arithmetic operators, but similar rules exists for strings. The first rule state that, if operands v_1 and v_2 are both numbers, and the operation is relevant to numbers, we delegate the result to the δ function already introduced in 3. The second rule covers the case where one or both of the operands are not numbers, but can be coerced into a number by the external function **tonumber**. In that case, we coerce the operands and do the operation. There is similar rule for concatenation, elided for brevity, when one of the operands is a string and the other a number. Finally, the last rule applies when the operands cannot be coerced into numbers. In this case we label the expression with ArithWO (some labels are listed in Figure 13, where WO stands for Wrong Operands), to signal the error. At this point, execution is stuck here, but in §4.4 we show how the metatable mechanism handles this erroneous situation.

$$\begin{array}{l} v ::= \ldots \mid \textbf{function}\ l\ (\ x\ ,\ldots\ )\ s\ \textbf{end} \\ \quad\quad \mid \textbf{function}\ l\ (\ x\ ,\ \ldots\ ,\ldots)\ s\ \textbf{end} \\ s ::= \ \ldots \mid e\ (\ e\ ,\ldots\ ) \mid e : x\ (\ e\ ,\ldots\ ) \mid \textbf{return}\ e \\ \quad\quad \mid \textbf{local}\ \ x, \ldots = \ e, \ldots\ \textbf{in}\ \ s\ \textbf{end} \mid var\ , \ldots = e\ , \ldots \\ var ::= x \mid e\ [\ e\ ] \\ e ::= \ldots \mid (e) \mid \{field\ , \ldots\} \mid e(e, \ldots) \mid e\!: x\ (e\ , \ldots) \\ field ::= e \mid [\ e\ ] = e \end{array}$$

Figure 15. Syntax of the remaining imperative subset.

$$\begin{array}{l} v ::= \ldots \mid objr \\ e ::= \ldots \mid <\ e\ , \ldots\ > \\ label ::= \ldots \mid \textsc{Return} \mid \textsc{Index} \mid \textsc{NewIndex} \mid \textsc{WFunCall} \end{array}$$

Figure 16. Store-related run-time terms.

$$\frac{\begin{array}{c} \boldsymbol{\delta}(\mathrm{rawget}, objr, v_1, \theta_1) \neq \textbf{nil} \\ \theta_2 = \boldsymbol{\delta}(\mathrm{rawset}, objr, v_1, v_2, \theta_1) \end{array}}{\theta_1 : objr\ [v_1] = v_2 \rightarrow^{\mathrm{s\_}\theta} \theta_2 : ;}$$

$$\frac{\boldsymbol{\delta}(\mathrm{rawget}, objr, v_1, \theta) = \textbf{nil}}{\theta : objr\ [v_1] = v_2 \rightarrow^{\mathrm{s\_}\theta} \theta : \llparenthesis\, objr\ [v_1] = v_2 \,\rrparenthesis_{\textsc{NewIndex}}}$$

$$\frac{\boldsymbol{\delta}(\mathrm{type}, v_1) \neq \text{"table"}}{\theta : v_1\ [v_2] = v_3 \rightarrow^{\mathrm{s\_}\theta} \theta : \llparenthesis\, v_1\ [v_2] = v_3 \,\rrparenthesis_{\textsc{NewIndex}}}$$

Figure 17. Field update.

$$\frac{v_2 = \boldsymbol{\delta}(\mathrm{rawget}, objr, v_1, \theta) \quad v_2 \neq \textbf{nil}}{\theta : objr\ [\ v_1\ ] \rightarrow^{\mathrm{e\_}\theta} \theta : v_2}$$

$$\frac{\boldsymbol{\delta}(\mathrm{rawget}, objr, v, \theta) = \textbf{nil}}{\theta : objr\ [\ v\ ] \rightarrow^{\mathrm{e\_}\theta} \theta : \llparenthesis\, objr\ [\ v\ ] \,\rrparenthesis_{\textsc{Index}}}$$

$$\frac{\boldsymbol{\delta}(\mathrm{type}, v_1) \neq \text{“table”}}{\theta : v_1\ [\ v_2\ ] \rightarrow^{\mathrm{e\_}\theta} \theta : \llparenthesis\, v_1\ [\ v_2\ ] \,\rrparenthesis_{\textsc{Index}}}$$

Figure 18. Field indexing.

4.2 Imperative Lua

The *imperative* subset is made up of functions, function application, tables, field indexing, and field update. Despite being values, Lua functions are in the imperative subset because parameters are mutable variables, so they are allocated in the σ store. Tables are mutable objects, and we allocate them in a separate store, denoted with θ. Object references, the domain of θ, are considered values, so are in the image of σ. We ask form them to satisfy the same properties as asked for references σ, together with the possibility of distinguishing syntactically between each kind of reference. The image of θ only contains tables.

Functions are labeled so each function in the source program has a unique label l. How labels are represented is not important; as long as they are comparable. This reproduces the correct semantics of function equality in Lua, where two identical functions are not equal if they are defined in

$$\frac{\begin{gathered}\forall\ 1 \leq \mathrm{i}, \mathit{field}_{\mathrm{i}} = v \vee \mathit{field}_{\mathrm{i}} = [\ v\ ] = v' \\ \theta_2 = \mathbf{(}\ \mathit{objr},\ \mathbf{<}\ \mathrm{addkeys}(\{\mathit{field}_1, \ldots\})\ \mathbf{,\ nil}\ \mathbf{>}\ \mathbf{)},\ \theta_1\end{gathered}}{\theta_1 : \{\mathit{field}_1, \ldots\} \ \rightarrow^{\mathrm{e\_}\theta}\ \theta_2 : \mathit{objr}}$$

Figure 19. Object creation.

$$\frac{\begin{gathered}\sigma' = (r_1, v'_1), \ldots, (r_{\mathrm{n}}, v'_{\mathrm{n}}), \sigma \\ \mathrm{i} \leq \mathrm{m} \Rightarrow v'_{\mathrm{i}} = v_{\mathrm{i}} \quad \mathrm{i} > \mathrm{m} \Rightarrow v'_{\mathrm{i}} = \mathbf{nil}\end{gathered}}{\begin{gathered}\sigma : (\mathbf{function}\ l\ (x_1, \ldots, x_{\mathrm{n}})\ s\ \mathbf{end})\ (v_1, \ldots, v_{\mathrm{m}}) \ \rightarrow^{\mathrm{funcall}} \\ \sigma' : \llparenthesis\ s\ [x_1 \backslash r_1, \ldots, x_{\mathrm{n}} \backslash r_{\mathrm{n}}]\ \rrparenthesis_{\textsc{Return}}\end{gathered}}$$

$$\frac{\begin{gathered}\sigma' = (r_1, v_1), \ldots, (r_{\mathrm{n}}, v_{\mathrm{n}}), \sigma \\ \mathrm{i} \leq \mathrm{m} \Rightarrow v'_{\mathrm{i}} = v_{\mathrm{i}} \quad \mathrm{i} > \mathrm{m} \Rightarrow v'_{\mathrm{i}} = \mathbf{nil} \\ \mathit{tuple} = \mathbf{<}\ v_{\mathrm{n+1}}, \ldots, v_{\mathrm{m}}\ \mathbf{>}\end{gathered}}{\begin{gathered}\sigma : (\mathbf{function}\ l\ (x_1, \ldots, x_{\mathrm{n}}, \mathbf{...})\ s\ \mathbf{end})\ (v_1, \ldots, v_{\mathrm{m}}) \\ \rightarrow^{\mathrm{funcall}} \sigma' : \llparenthesis\ s\ [x_1 \backslash r_1, \ldots, x_{\mathrm{n}} \backslash r_{\mathrm{n}}, \mathbf{...} \backslash \mathit{tuple}]\ \rrparenthesis_{\textsc{Return}}\end{gathered}}$$

$$\frac{\boldsymbol{\delta}(\mathrm{type}, v) \neq \text{“function”}}{\sigma : v\ (v_1, \ldots, v_{\mathrm{n}}) \ \rightarrow^{\mathrm{funcall}} \sigma : \llparenthesis\ v\ (v_1, \ldots, v_{\mathrm{n}})\ \ \rrparenthesis_{\textsc{WFunCall}}}$$

$$\sigma : v{:}\mathit{name}\ \mathbf{(}e_1,\ldots,e_{\mathrm{n}}\mathbf{)} \ \rightarrow^{\mathrm{funcall}} \sigma : v\mathbf{[}\text{“}\mathit{name}\text{"}\mathbf{]}\ \mathbf{(}v,e_1,\ldots,e_{\mathrm{n}}\mathbf{)}$$

Figure 20. Function and method calls.

$$\begin{gathered}\sigma : \llparenthesis\ ;\ \rrparenthesis_{\textsc{Return}} \rightarrow^{\mathrm{funcall}}\ \sigma : ; \\ \sigma : \llparenthesis\ E_{\mathrm{lf}}[\![\ \mathbf{return}\ \ < v, \ldots >\ ]\!]\ \rrparenthesis_{\textsc{Return}} \ \rightarrow^{\mathrm{funcall}} \\ \sigma :\ < v, \ldots > \\ \sigma : \llparenthesis\ E_{\mathrm{lf}}[\![\ \mathbf{return}\ \ < v, \ldots >\ ]\!]\ \rrparenthesis_{\textsc{Break}} \ \rightarrow^{\mathrm{funcall}} \\ \sigma : \mathbf{return}\ \ < v, \ldots >\end{gathered}$$

Figure 21. Semantics of **return**.

different parts of the source file, as shown in the following interaction with the reference interpreter:

```
> f = function() end
> g = function() end
> print(f == g)
false
```

A source function may evaluate to different values during the evaluation of the program, due to different substitutions of their free variables. Our use of substitution and references means that we do not need to have explicit *closures*, a function definition is itself a closure once the focus of evaluation has reached it.

Figure 16 adds *tuples*, used for returning multiple values from function application, and for functions that can handle a variable number of arguments (*vararg* functions) through the **...** *vararg* operator. Wrapping an expression in parenthesis has a semantic effect in Lua: if the expression evaluates to a tuple the parenthesis discards all but the first value of the tuple (if the tuple is empty the parenthesized expression evaluates to **nil**).

Besides being "truncated" to their first value, these tuples can also be concatenated with another tuple, depending on their syntactical place in the program: in an expression list $e_1, \ldots, e_n$ the tuples of expressions e_1 to e_{n-1} evaluate to their first value, or **nil** for the empty tuple (the same behavior as parenthesized expressions). These $n-1$ values then form a tuple of their own, which is concatenated with the tuple of e_n. Semantically, this is done through reductions between tuples that "flattens" tuples of tuples until reaching a tuple where none of the values are another tuple.

The new statements also include multiple variable definition and assignment, which generalizes the single-variable versions introduced in §3. The reduction rules for these statements are not shown here for reasons of brevity, but are a straightforward extension of the simpler versions: in case of multiple assignment, the evaluation contexts assure that all lvalues are evaluated before rvalues, and the tuples for both sides are flattened, then lvalues are paired with their corresponding rvalue, with any lvalues that do not have a corresponding rvalue paired with **nil**.

Figure 17 describes assignment to table fields. It uses some services modeled by the δ function: $\delta(\mathrm{type}, \mathrm{v})$ is the type of the value; $\delta(\mathrm{rawget}, \mathit{objr}, v, \theta)$ is primitive table indexing, yielding either the value associated with v in $\theta(\mathit{objr})$ or **nil** if there is no associated value; $\delta(\mathrm{rawset}, \mathit{objr}, v_k, v, \theta)$ is primitive table update, yielding a new θ where the table referenced by *objr* associates v with value v_k.

The rules show field update under 3 different circumstances: when the operation is made over an actual table with an existing key; when the operation is made over an actual table but with an unknown key; and when the assignment is carried over a non-table value. The last two cases just tag the expression with NewIndex, which will be handled by the metatable mechanism explained in Section 4.4. Field access (Figure 18) have similar rules, but tagging exceptional situations with Index.

Figure 19 provides meaning to table constructors. Its complete semantics actually depends upon the meta-function addkeys, which adds absent keys in the constructor (see Figure 15 for the syntax of table constructors). It works by supplying consecutive natural numbers as keys, starting with 1.

Figure 20 shows function and method application, described with a new relation $\rightarrow^{\mathrm{funcall}}$. Formal parameters are mutable variables, so a fresh reference is allocated for each parameter. The first rule covers all the cases involving the application of a non-vararg function: when it is applied to the same number of arguments as formal parameters, when it is applied to fewer arguments, with unpaired parameters receiving **nil**, and when it is applied to more arguments, with extra arguments silently ignored. Similar to what we did with while loops in §4.1, we label the body with Return, to indicate the point in which a **return** statement must jump

$$\frac{\mathsf{l} \in \{\text{type}, \text{assert}, \text{error}, \text{pcall}, \text{select}, \ldots\}}{\theta : \textbf{\$builtIn}\ \mathsf{l}\ (v_1, \ldots, v_n) \rightarrow^{\text{builtIn}} \theta :\ \delta(\mathsf{l},\ v_1, \ldots, v_n)}$$

$$\frac{\mathsf{l} \in \{\text{ipairs}, \text{next}, \text{pairs}, \text{getmetatable}, \ldots\}}{\theta : \textbf{\$builtIn}\ \mathsf{l}\ (v_1, \ldots, v_n) \rightarrow^{\text{builtIn}} \theta : \delta(\mathsf{l}, v_1, \ldots, v_n, \theta)}$$

$$\frac{\begin{array}{c}\mathsf{l} \in \{\text{rawset}, \text{setmetatable}\}\\ \theta_2 = \delta(\mathsf{l}, v_1, \ldots, v_n, \theta_1)\end{array}}{\theta_1 : \textbf{\$builtIn}\ \mathsf{l}\ (v_1, \ldots, v_n) \rightarrow^{\text{builtIn}} \theta_2 : v_1}$$

Figure 22. Interface with the δ function.

to. It is, roughly speaking, the syntactic equivalent to the return address saved in an activation frame. In [3], as with while loops, the body of every function is put into a labeled block when desugaring.

Returning to Figure 20, the second rule shows the case of a vararg function call: the difference just resides on what is done with surplus arguments: in this case, they are put into a tuple expression, which replaces the vararg expression (**...**) in the body of the function.

The third rule has to do with one of the exceptional situations that can be managed by the metatable mechanism: a function call over a non-function value. Again, at this point we just label the whole expression with a tag that indicates what happened. The last rule shows how the method invocation is translated into a table look-up, with the object being injected as the first argument of the function.

Figure 21 shows the semantics of the **return** statement as well as implicitly returning by reaching the end of the function. The ideas used in this rules are analogous to the ones expressed when defining the semantics of the **break** statement, in Section 4.1.

4.3 Built-in Services

In Lua, built-in services offered by Lua's standard library are stored in the execution environment, a table named _ENV, where the keys are the names of the services and the values are their definitions. For instance, when we access the table field named "type", we access the function that given an element provides its type (as a string):

```
> print(type({}))
table
```

(Remember from §2: using an identifier not in scope is equal to accessing _ENV.) We can override its definition and obtain a different behavior:

```
> type = function () return 'not a type' end
> print(type({}))
not a type
```

$$\delta(\text{pairs}, objr, \theta) = \begin{array}{l}(\textbf{function}\ \$\text{getIter}\ \ ()\\ \quad \textbf{local}\ \text{v1, v2, v3} = \text{h(objr)}\ \textbf{in}\\ \quad\quad \textbf{return} < \text{v1, v2, v3} >\\ \quad \textbf{end})()\end{array}$$

where $h = \text{indexmetatable}(objr, “\_\_\text{pairs}”, \theta)$
and $h \neq \textbf{nil}$

$$\delta(\text{pairs}, objr, \theta) = \begin{array}{l}< \textbf{function}\ \$\text{next}\ (\text{table}, \text{index})\\ \quad\ \textbf{return}\ \textbf{\$builtIn}\ \text{next}(\text{table}, \text{index})\\ \quad \textbf{end}, \text{objr}, \textbf{nil}>\end{array}$$

if $\text{indexmetatable}(objr, “\_\_\text{pairs}”, \theta) = \textbf{nil}$

$\delta(\text{pairs}, v, \theta) = \textbf{\$builtIn}\ \text{error}(msg\ ..\ \textbf{\$builtIn}\ \text{type}(v))$
if $\delta(\text{type}, v) \neq$ “table”
where msg = “table expected, got "

Figure 23. Basic functions of the standard library: pairs.

However, the original type function is still accessible from other services in the library. We can see this when we call next, the function that iterates over the fields in a table:

```
> next(1)
stdin:1: bad argument (table expected, got number)
```

In order to model this behavior, prior to the execution of a program the _ENV table must be populated with the functions from the standard library. But these functions are just wrappers for a special (run-time) expression **$builtIn**. When evaluated, this expression calls the δ function with the actual definition of the function. Built-in services, like next, might call other services through the **$builtIn** term instead of ordinary function application, effectively reproducing the early binding that is required.

While it might sound a bit intricate, this design gives the formalization several desirable properties: compliance with the semantics as defined in the reference manual and the reference interpreter, and a modular way of tackling the formalization of built-in services. And at the level of the mechanization, it allows us to experiment and test against different implementations of these services with minimal changes in the rest of the formalization.

Figure 22 gives the semantics of **$builtIn** using three rules, corresponding to three different kinds of services: services that do not operate on tables, so do not need to access the object store θ, services that read from tables, and services that update existing tables, yielding a new θ. The antecedents of the first two rules show just some of the services that are in each category; the actual list of services includes almost all the built-in basic functions of the Lua language, together with services from the libraries math, string and table.

The δ function defines, in a denotational way, the actual fundamental details of the semantics of the built-in services and the primitive operators of the language. In the rest of this section we discuss an interesting example: the pairs built-in function (Figure 23).

The built-in service pairs is used to iterate a table using a **for** loop. It must return three values: an iterator function, the object to index, and the first index. According to the equations in the figure, there are three different scenarios: In the first case, when the table *objr* has a custom handler *h* in the __pairs key of its metatable, calls this handler to get the iterator triplet. The metafunction indexmetatable queries the metatable (metatables are discussed further in §4.4). Also note that we let δ yield not only values but any valid expression.

It could look odd to create a function whose body calls *h*, instead of directly returning it. The reason is twofold: First, according to the manual, we must only return the first three values returned by the indexed function. This is achieved by creating three variables, one for each value, and return only those. If there are more values, they are discarded. Second, since **local** and **return** are not valid *expressions*, and in this case, $\boldsymbol{\delta}$ must return an expression (not necessarily a value!), we wrap this code in a closure.

In the second case, when the table has no metatable or no handler for __pairs, the reference manual indicates that pairs(t) returns "*the* `next` *function, the table* `t`*, and **nil***". This case models this behavior by wrapping a call to **$builtIn** next, as mentioned earlier in this section. The label of this function guarantees that it will be the same function that is bound to next in our initial environment.

The third and final case of pairs constructs an expression that will assemble an error message and then throw an error using the error built-in primitive. As with next, we cannot look-up the error built-in function in the environment, as it could have been rebound by the programmer.

Before concluding this section, we note that we let the interpretation function define the meaning of every primitive operator and library service using a denotational approach. Several of these primitives could also be given an operational semantics, however, we decided to prioritize cohesion and modularity.

4.4 Metatables

The most notable feature of Lua is its metaprogramming mechanism, *metatables*, that lets the programmer adapt the language to specific domains. With metatables Lua can maintain its original design decision to "*keep the language simple and small*"[4], while still being able to cope with a variety of programming concepts[11].

Briefly, metatables let the programmer specify *fallbacks* for certain operations: arithmetic over non-numeric values, concatenation over non-string values, equality between objects that do not have the same identity, application over values that are not functions, indexing or updating a field over a value that is not a table, etc.

Metatables are plain tables, and the fallbacks that a particular metatable supports are typically functions associated with a unique string key for each operation (e.g. __add for the fallback to the plus operator, or __newindex for the fallback to field update). Lua libraries are free to extend this mechanism with their own fallbacks (like the pairs built-in function of the previous section, which can look up __pairs in the metatable, if it exists). Each Lua table can have its own metatable, while values of other types share a single metatable for each type.

We have shown in previous sections that regular semantics of operations just labels the expression or statement involved when it reaches a case where a fallback in a metatable could be used. This approach simplifies the regular semantics, and improves the modularity of the formalization. The relations $\rightarrow^{\mathrm{e\_metatable}}$ and $\rightarrow^{\mathrm{s\_metatable}}$ that we show in this section take these labeled terms and act accordingly.

Figure 24 shows how $\rightarrow^{\mathrm{e\_metatable}}$ resolves arithmetic operations over operands of unexpected type, a condition labeled with ArithWO. The metafunction getbinhandler is analogous to those described in the Lua reference manual: it looks for a handler first in the metatable of the left operand v_1, then the metatable of the right one v_2, by looking into the corresponding field in those metatables. We also abstract the mapping between binary operators and their metatable keys with the metafunction binopeventkey.

Looking up a fallback in a metatable is guaranteed to either return the fallback or return **nil** because of two invariants: a metatable is always a table, and the metatable of a metatable, it if exists, is ignored for this look-up. This means that abstracting this look-up with a metafunction does not compromise the small-step nature of our semantics.

The first rule of Figure 24 shows how the operation is rewritten as an application of the handler on the two operands as arguments. If the handler is not a function this may trigger yet another fallback. The second rule shows what happens when no handler is found: an error is thrown using the error built-in service. We also abstract the construction of error messages with the #errmessage metafunction.

Figure 25 describes how the metatable mechanism works for field updates over a non-table value or a missing key. Again, we make use of metafunctions that abstracts the inner workings of the metatable mechanism: indexmetatable, which looks for the metatable of its first argument and looks up the fallback with the key passed as its second argument.

The first two rules of Figure 25 shows how this case resolves differently depending on whether the handler is a function or not (typically, in the second case the handler will be a table). The last two rules show how the absence of handler has different results depending on whether the original value is a table or not.

[11] See section "Code Structure / Programming Paradigms" at lua-users.org/wiki/LuaDirectory

$$\frac{v_3 = \text{getbinhandler}(v_1, v_2, \text{binopeventkey}(op), \theta) \qquad v_3 \neq \textbf{nil}}{\theta : (\!|\ v_1\ op\ v_2\ |\!)_{\textsc{ArithWO}} \rightarrow^{\text{e\_metatable}} \theta :\ v_3\ \textbf{(}v_1, v_2\textbf{)}}$$

$$\frac{\text{getbinhandler}(v_1, v_2, \text{binopeventkey}(op), \theta) = \textbf{nil} \qquad t_1 = \boldsymbol{\delta}(\text{type}, v_1) \qquad t_2 = \boldsymbol{\delta}(\text{type}, v_2)}{\theta : (\!|\ v_1\ op\ v_2\ |\!)_{\textsc{ArithWO}} \rightarrow^{\text{e\_metatable}} \theta : \textbf{\$builtIn}\ \text{error}\ (\ \textbf{\#errmessage}(\textsc{ArithWO}, t_1, t_2)\ )}$$

Figure 24. Metatable mechanism for arithmetic binary expressions.

$$\frac{v_4 = \text{indexmetatable}(v_1, \text{“\_\_newindex”}, \theta) \qquad \boldsymbol{\delta}(\text{type}, v_4) = \text{“function”}}{\theta : (\!|\ v_1\ \textbf{[}v_2\textbf{]}\ \texttt{=}\ v_3\ |\!)_{\textsc{NewIndex}} \rightarrow^{\text{s\_metatable}} \theta : v_4\ \textbf{(}v_1, v_2, v_3\textbf{)}}$$

$$\frac{v_4 = \text{indexmetatable}(v_1, \text{“\_\_newindex”}, \theta) \qquad v_4 \neq \textbf{nil} \qquad \boldsymbol{\delta}(\text{type}, v_4) \neq \text{“function”}}{\theta : (\!|\ v_1\ \textbf{[}v_2\textbf{]}\ \texttt{=}\ v_3\ |\!)_{\textsc{NewIndex}} \rightarrow^{\text{s\_metatable}} \theta : v_4\ \textbf{[}v_2\textbf{]}\ \texttt{=}\ v_3}$$

$$\frac{\text{indexmetatable}(\text{objr}, \text{“\_\_newindex”}, \theta_1) = \textbf{nil} \qquad \theta_2 = \boldsymbol{\delta}(\text{rawset}, \text{objr}, v_1, v_2, \theta_1)}{\theta_1 : (\!|\ \text{objr}\ \textbf{[}v_1\textbf{]}\ \texttt{=}\ v_2\ |\!)_{\textsc{NewIndex}} \rightarrow^{\text{s\_metatable}} \theta_2 : \texttt{;}}$$

$$\frac{\text{indexmetatable}(v_1, \text{“\_\_newindex”}, \theta) = \textbf{nil} \qquad t = \boldsymbol{\delta}(\text{type}, v_1) \qquad \text{t} \neq \text{“table”}}{\theta : (\!|\ v_1\ \textbf{[}v_2\textbf{]}\ \texttt{=}\ v_3\ |\!)_{\textsc{NewIndex}} \rightarrow^{\text{s\_metatable}} \theta : \textbf{\$builtIn}\ \text{error}\ \textbf{(}\ \textbf{\#errmessage}(\textsc{NewIndex}, t)\ \textbf{)}}$$

Figure 25. Metatable mechanism for field update.

$$\begin{aligned}
\sigma : \theta : Enp\llbracket\ \textbf{\$err}\ v\rrbracket &\mapsto \sigma : \theta : \textbf{\$err}\ v \\
\sigma : \theta : E\llbracket\ (\!|\ Enp\llbracket\ \textbf{\$err}\ v\rrbracket\ |\!)_{\textsc{ProtMd}}\ \rrbracket &\mapsto \sigma : \theta : E\llbracket\ \textbf{<false}, v\textbf{>}\rrbracket \\
\sigma : \theta : E\llbracket\ (\!|\ \texttt{;}\ |\!)_{\textsc{ProtMd}}\ \rrbracket &\mapsto \sigma : \theta : E\llbracket\ \textbf{<true>}\rrbracket \\
\sigma : \theta : E\llbracket\ (\!|\ \textbf{<}v, \ldots\textbf{>}\ |\!)_{\textsc{ProtMd}}\ \rrbracket &\mapsto \sigma : \theta : E\llbracket\ \textbf{<true}, v, \ldots\textbf{>}\rrbracket
\end{aligned}$$

Figure 26. Errors.

4.5 Semantics of Programs and Error Handling

The definition of the ↦ reduction relation that describes the full semantics of Lua is essentially a straightforward extension of the simpler relation given in Figure 10. The domain now includes θ, and maps the relations that were described in previous sections. Each of these relations is extended with the θ and σ stores as needed. We omit these definitions for brevity.

More interestingly, in order to model the semantics of Lua's exception handling, we must extend this relation. Lua's exception handling consist of two built-in functions: error, which throws an error (any Lua value, usually a string), and pcall, which executes a function in *protected mode.* As any other built-in function, this behavior can be override by a developer.

Normally an error aborts the program, but if it is thrown in the context of a pcall, it is caught. In that case, pcall returns **false** and the error, otherwise, it returns true and the values returned by the function called.

Figure 26 describes the part of the ↦ relation that models error propagation and handling. For it, two new run-time constructs are added: **\$err** to denote an error, and $(\!|\ s\ |\!)_{\textsc{ProtMD}}$ to denote code that must be executed in protected mode. In the first rule, the evaluation context Enp is identical to E, except that there are no occurrences of $(\!|\ E\ |\!)_{\textsc{ProtMD}}$. The rule essentially aborts the whole program if there is no protected context around the error. The second rule aborts up to the first occurrence of a protected mode label, if there is one. The other three rules transition out of protected mode whether an error occurred or not.

The reader might wonder why we are modeling error handling here, and not in its own relation as we did with all the other parts of the semantics. The reason is merely technical: the rule that aborts the whole program, if isolated in its own relation, would break the *unique decomposition* property of evaluation contexts, in which there is a single way for decomposing a term into an evaluation context and the contents of its hole. We could have put just this rule explicitly in ↦ while having the others in an hypothetical $\rightarrow^{\text{error}}$ relation, but decided to keep all aspects of a feature together.

5 Mechanization

The formalization of the semantics was carried in parallel with its mechanization in PLT Redex [2]. This tool helped us recognize problems in our first attempts at formalizing Lua, and allowed us to experiment with new ideas before adding them to the formalization. It also allowed us to execute part of test suite of the reference interpreter of the language[12], providing evidence that our semantics is in compliance with it.

We could not use the whole test suite, for the following reasons:

- Language features not covered by our formalization: coroutines, the **goto** statement, garbage collection, some standard library functions (mostly related with

[12] Available at https://www.lua.org/tests/.

File	Features tested	Coverage
calls.lua	functions and calls	77.83%
closure.lua	closures	48.5%
constructs.lua	syntax and short-circuit opts.	63.18%
events.lua	metatables	90.4%
locals.lua	local variables and environments	62.3%
math.lua	numbers and math lib	82.2%
nextvar.lua	tables, next, and for	53.24%
sort.lua	(parts of) table library	24.1%
vararg.lua	vararg	100%

Figure 27. Lua 5.2's test suite coverage.

file handling) and other standard libraries implemented in C (bit32, coroutine, debug, io, etc).

- Several other tests for implementation details of the interpreter, and not the language. According to the Lua authors, the goal of the test suite is to test their reference *implementation* of Lua, and not to serve as a conformance test for alternative implementations[13].

In practice, from the 25 .lua files present in the test suite, which actually test some feature of the language, we are able to port and run 9 against our PLT Redex mechanization. Figure 27 shows the percentage of LOCs actually tested from each of these remaining files, totaling 1256 LOCs successfully tested. Each file from the test suite is a sequence of assertions about the expected outcome of valid code as well as code that generates errors. It is important to remark that every file and line not tested is for the reasons explained above, and every line (in the 9 files tested) that fall within the scope of this work successfully passes the test. We take this as strong evidence that the mechanization of our formal semantics behaves exactly the same as the reference Lua interpreter.

Unfortunately, we do not have the space to discuss the code, which we plan to do in an extended version of this article. For the moment, we refer the reader to the documentation accompanying the code attached as supplementary material.

Dynamic Loading of Code By implementing our parser directly in Racket (the language upon which PLT Redex is based), we mechanized easily the load service: Lua's compiler available at runtime.

There are several details to mention related to the solution implemented, but for reasons of space we point out the most prominent:

- It covers the two modes: when the program to be compiled is passed as a string, or when it is a function from which the service obtains the program's string.
- It can handle the compilation of code on a modified global environment.
- For completion, we emulate the case of the compilation of *binary chunks* (that is, a pre-compiled version of the code). This feature is implemented in conjunction with the service string.dump, which returns a string containing a binary representation of a function, given as a parameter.

6 Related Work

As mentioned in the introduction and throughout the text, the major source of inspiration is the work done in [3, 13, 14]. At a broad view, we incorporated from these works the semantic model, together with its mechanization in PLT Redex. As for the differences, we already mentioned in the introduction that our presentation of the model is, arguably, more suitable for readers with a traditional background in operational semantics.

Also, the metatable mechanism, the tuples, the δ-function returning any expression of the language, and the evaluation of code at runtime are some of the aspects of the language and its formalization which distinguishes our work from the aforementioned works. And as mentioned in the introduction, we do not minimize the language into a core. As a result, we avoid the known complexities that the core language approach could introduce in the resulting model [1, 9]: verbose desugared code, a reduced confidence on the compliance of the given semantics with respect to the original language's specification and, in general, a non-trivial connection between the properties and phenomena observed in the core language and the original language (this is tied to the complexities of the translation process). Additionally, maintaining the proximity with the original language paves the way to a mechanization that also Lua developers could use and verify, as their intuition about the language's semantics is better expressed —one of the key goals of this project.

Following [3], one major step in formal semantics for JavaScript is JSCert[1]: a formalization of ES5 in the Coq proof assistant, together with an interpreter extracted from the formalization (JSRef). It presents a big-step semantics for the specification of ES5. In order to gain confidence about the compliance of their formalization with respect to the specification of ES5, the authors recognize the importance of the revision of their Coq model by people of different areas, ranging from developers of analysis tools for JavaScript, developers of JavaScript VMs and even ECMA authors. In our project, beside semanticist and Lua implementers, it is our hope to include Lua developers as well. This may be difficult to achieve over a Coq model because, as the authors from [1] recognize, using proof assistants requires considerable more learning than using, for example, tools specifically designed for mechanizing language specifications. While we pursue in the future the mechanization of proofs of properties of

[13] https://www.lua.org/wshop15/Ierusalimschy.pdf

our model, perhaps using Coq, we want to take advantage of the ease of use of PLT Redex.

Besides the aforementioned [3, 13], [9] introduces a small-step operational semantics for the full language specified in ECMA-262 Standard, 3rd Edition, including proofs of several properties of the model. The authors recognize that, by defining the semantics of each construction as described by the ECMA specification, it gives them the greatest likelihood that the model is correct. While this approach resulted in a model that inherited the size and complexities of the language being defined, the experience is interesting for our investigations on Lua, as we are dealing with a smaller, simpler language.

Specific to Lua's semantics, to the best of our knowledge, there is just one another experience: [8] presents an operational semantics for Lua, in the style of Featherweight Java [6]. It considers a subset of the features from the ones presented here and provides a reference implementation in Haskell. While being an interesting project, it suffers from the same limitations that we mentioned above when considering the study of a core language instead of the full language.

7 Conclusion

We give a small-step operational semantics for a large subset of the Lua programming language, as specified both by its informal reference manual and by its reference implementation.

The semantics tackles the majority of the complex features of Lua, such as its metatable mechanism, dynamic execution of source code, error handling and several other standard library functions. It is defined in a modular way, and could be extended to tackle absent features, such as coroutines [12] and garbage collection [11], without modifying the essence of what is already specified.

We provide a mechanization of the formal semantics in PLT Redex. The portion of the test suite for the reference interpreter, which is of interest for our model, has been successfully tested against this mechanization, providing evidence that we are successfully modeling the behavior of the language.

The development of the formal semantics, its mechanization, and its test suite make up a tool that both semanticists and Lua developers can use for understanding and extending the features of the language.

There are several further avenues for development: adding missing features as coroutines, **goto** statements, new operators, large integer types of version 5.3 and garbage collector. Among them, the **goto** statement would require using a slightly different notion of a concept already in use in our model: evaluation contexts that maintain the portions of the program already executed, as used in [7]. The small step style of the semantics of our model seems to be adequate for modeling non-local control and the interaction between goto statements and block-scoped variables, as argued in the cited work. It remains as a future work to cope with the remaining complexities of the semantics of the statement.

The formal semantics and its mechanization can provide a basis for specifying, implementing, and formally proving correct static analyses for Lua programs. There are already some tools and language extensions that do this[14], but they currently have no formal guarantee of correctness.

In proving desired properties of the model (fundamentally, progress property), we choose to let as a future work the developing of a tool for assisting in the translation of our PLT Redex model to a Coq model. In that way, we can offer a mechanization ready to use by a broader audience, while also having a quick escape to a proof assistant, to verify properties of each iteration of the PLT Redex model.

Acknowledgments

We are very grateful to the anonymous reviewers for their insightful feedback.

References

[1] M. Bodin, A. Chargueraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *POPL '14*, 2014.

[2] M. Felleisen, R. B. Finlder, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[3] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP '10*, 2010.

[4] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of an extension language: a history of Lua. In *Brazilian Symposium on Programming Languages*, 2001.

[5] R. Ierusalimschy, L.H. de Figueiredo, and W. Celes. The evolution of an extension language: a history of lua. In *Brazilian Symposium on Programming Languages*, 2001.

[6] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23:396–450, 2001.

[7] R. Krebbers and F. Wiedijk. Separation logic for non-local control flow and block scope variables. In *FOSSACS'13*, 2013.

[8] Hanshu Lin. Operational semantics for Featherweight Lua. Master's thesis, San JosÃľ State University, march 2015.

[9] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS '08*, 2008.

[10] A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy. A formalization of Typed Lua. In *DLS '15*, 2015.

[11] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *FPCA '95*, 1995.

[12] A. L. Moura and R. Ierusalimschy. Revisiting coroutines. *TOPLAS*, 31(2):6:1–6:31, February 2009.

[13] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *DLS '12*, 2012.

[14] J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty: A tested semantics for the Python programming language. In *OOPSLA '13*, 2013.

[15] L. H. de Figueiredo R. Ierusalimschy and W. Celes. Lua – an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[14] Such as Luacheck (https://github.com/mpeterv/luacheck), Ravi (http://ravilang.github.io), and Typed Lua [10].