

From specification to testing: semantics engineering for Lua 5.2

Mallku Soldevila · Beta Ziliani · Bruno Silvestre

the date of receipt and acceptance should be inserted later

Abstract We provide formal semantics for a large subset of the Lua programming language, in its version 5.2. The semantics is a major part of an ongoing effort to construct reliable tools to analyze Lua code. In this work, we present the details of several key aspects of the language, like the semantics of its only structured data-type (*tables*), its meta-programming mechanism (*metatables*), error handling, and how these mechanisms are used to define a complex dynamic semantics that must deal with several possible erroneous situations during run-time, given the nature of the language.

The semantics is mechanized in Redex, a DSL specially designed to specify and debug operational semantics. We validated the mechanization in two ways: first, by executing within Redex the test suite of the reference interpreter of Lua; second, by specifying and performing random testing of its fundamental properties using the `redex-check` tool. Together, they evidence that our model soundly captures the semantics of the selected fragment of the language.

Additionally, we address some of the performance problems that typically arise when testing a mechanization in Redex by using a simple implementation of a reachability-based garbage collector that captures key aspects of Lua's. By collecting syntactic garbage we reduce the size of configurations during run-time.

Keywords Semantics · Operational Semantics · Imperative languages · Domain specific languages · Lua · Reduction semantics · Randomized testing

M. Soldevila
FAMAF, UNC and CONICET (Argentina)
E-mail: mes0107@famaf.unc.edu.ar
ORCID: 0000-0002-8653-8084

B. Ziliani
FAMAF, UNC and Manas.Tech (Argentina)
E-mail: beta@mpi-sws.org
ORCID: 0000-0001-7071-6010

B. Silvestre
INF, UFG (Brazil)
E-mail: brunoos@inf.ufg.br
ORCID: 0000-0002-5774-1948

CR Subject Classification 10011311 · 10010134 · 10011010 · 10011017

Acknowledgements We thank Dr. Daniel Fridlender and Dr. Fabio Mascarenhas for their useful insights and support during the development of this research.

Declarations

Funding This work was funded by the following projects: Consolidar II 33620180101063CB, UNC SECyT (Argentina) PICT D 2017-3315, ANPCyT (Argentina).

Conflicts of interest/Competing interests Not applicable.

Data transparency The claims made in the present work are supported by the provided mechanization. With it, it is possible to reproduce the results shown in the paper.

Code availability The mechanization is publicly available at <https://github.com/Mallku2/lua-gc-redex-model>.

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 (Ierusalimschy et al, 1996). 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 (Ierusalimschy et al, 2001a) but also in mobile games and game frameworks, plugin development (for example, in the photo editing software Adobe Photoshop Lightroom (Adobe, 2019) and the type-setting system LuaTeX (LuaTeX, 2018)), web application firewalls (Graham-Cumming, 2013), and embedded systems (Lua Developers, 2017).

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 as well as code linters and static analyzers can be found in the wild (Lua Developers, 2018, 2014). However, the informal nature of the specification means that developers of these tools must resort to their intuition, formed by study of the reference manual, inspection of the source code of the interpreter, and experimentation.

Even if Lua is considered a small language, it is not unusual to stumble across some puzzling behaviors that may not be easily inferred from the official documentation. As a simple example consider the following program:

Example 1 (To return or not to return.)

```

1 function f() return nil end
2
3 function g() end
4
5 print(f(), f(), f()) -->> nil  nil  nil
6 print(g(), g(), g()) -->> nil  nil

```

In this example, we have `f` returning Lua’s null value (`nil`), and `g` not returning anything. Then, when printing three identical calls to `f` and `g`, respectively, we see a difference: in the former case three `nil`s are printed, while in the latter only two are (we write `-->>` to indicate the output of the interpreter; `--` is Lua’s token for comments). The different outputs can be explained by two related aspects of Lua: *tuples* and *vararg* arguments.

In this work we present an extensive formalization—mechanized in Redex (Felleisen et al, 2009)—of a large subset of Lua 5.2, explaining in detail its different mechanisms, and providing sound grounds for developing alternative implementations, extensions, and analysis tools for the language. In fact, the present work refines, extends, and greatly improves Soldevila et al (2017), which was the basis to formalize Lua’s garbage collector (Soldevila et al, 2020), including the development of a small tool to verify simple programs making use of Lua’s *weak tables* (akin to weak references).

In order to gain confidence in our model, we took three measures: firstly, we developed a semantics that (mostly) contains each programming concept from the reference manual, in a one-to-one correspondence, instead of focusing on an idealized core of the language. Secondly, we successfully run the test suite of the reference interpreter of the language (Lua Dev. Team, 2013) *directly within Redex*, checking that our semantics is correct to the extent of what is tested within the suite. That is, unlike many other works in the literature (*e.g.*, Guha et al (2010); Politz et al (2013)), we did not pay the cost—confidence-wise—of building a new interpreter. Lastly, we used Redex’s support for formal systems and random testing (Klein, 2009) to random test soundness properties of our semantics. This allowed us to detect a plethora of errors and omissions that were not caught during the development of the semantics, even after successfully passing the tests of the test suite.

In overall, we obtain evidence that our semantics is sound and correspond to that of the selected subset of the language’s features, including:

- Every type of Lua value, except *coroutines* and *userdata* (see below);
- Metatables (Lua’s configurable dynamic dispatch);
- Identity of closures;
- Dynamic execution of source code;
- Error handling;
- A large collection of services from the standard library;
- Garbage collection (presented in Soldevila et al (2020)).

We purposely excluded 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;
- 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.

Contributions: As a summary, we present:

1. A formalization of a large portion of Lua, in its version 5.2.
2. A mechanization of said formalization in Redex.
3. An interpreter based on the mechanization that successfully passes all relevant tests cases from Lua’s interpreter test suite.
4. A randomly-tested proof of progress of the semantics.

The mechanization can be downloaded from

<https://github.com/Mallku2/lua-gc-redex-model>

Quick tour:

- §2 presents a brief description of Lua, with emphasis on particular features that we formalize in later sections. Its reading is non-essential for the Lua *connoisseur*.
- §3 presents the basic concepts that our model uses, via a formalization of a very small subset of Lua. Its reading is important to understand the following sections.
- §4 expands §3 with the formalization of the most interesting parts of the semantics. Of special interests are the formalization of metatables (§4.4) and tuples (§4.2.3).
- §5 presents some relevant properties of the language.
- §6 presents the software artifact (the mechanization in Redex of the language), and the results of the two types of testing performed. It is an essential part of the work; its reading being most recommended.
- §7 discusses related work.
- §8 reflects on the experience of formalizing and mechanizing Lua, and discusses future avenues of research.

2 Lua, an extensible scripting language

In this section we present a quick tour to some of Lua’s most salient characteristics and features captured in our model. The Lua expert can safely skip this section and come back to it only if necessary.

2.1 First-class closures and tables

We start with a simple memoization function,¹ listed in Example 2, 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). Tables are, in essence, associative arrays indexed by any Lua value except the null value (`nil`). They also

¹ Taken from <http://lua-users.org/wiki/FuncTables>.

Example 2 (Memoization in Lua.)

```

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)

```

Example 3 (The environment `_ENV`.)

```

1 x = 10
2 print(_ENV['x'])  -->> 10
3 _ENV['x'] = 0
4 print(x)  -->> 0
5 _ENV = {}
6 print()  -->> error, print is not anymore in _ENV

```

come with syntax sugar and metaprogramming facilities that can greatly extend their functionality beyond simple associative arrays, as we will see in §2.3 below.

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

It is important to note that all procedures 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.

2.2 `local` definitions and the environment `_ENV`

Note that the definitions of `memoize`, `t`, and `memsum` are prefixed by the keyword `local`. As its name suggests, `local` creates a local variable, initialized with the *rvalue* (what comes after the ‘=’ operator). If we omit this keyword, a declaration is considered an *assignment*, and if the variable was declared local before, then it performs an update of its value, as usual. More interestingly, if there is no such variable in scope, then Lua looks up for the variable in a table called the *environment*, which is bound to the identifier `_ENV`, using the variable’s identifier as the

Example 4 (OOP based on Lua’s metatable mechanism.)

```

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

```

key. Similarly, reading the value of a variable that is not within scope triggers the search for the value associated with that name in the environment. Abusing the notation, this means that any occurrence of a variable x that is *not* in scope, is just syntax sugar for `._ENV["x"]`. Variables in the environment are called *global* variables. Services from the standard library (like `print`) are accessible through global variables, and are, therefore, bound by this environment.

Since `._ENV` is in itself a variable, the programmer can, at any time, change the environment in which a program is executed by simply assigning another table to `._ENV`. Example 3 shows some simple interactions with the environment to illustrate this concept.

2.3 Metatables

The most notable feature of Lua is its metaprogramming mechanism, *metatables*, that lets the programmer adapt the language to specific domains. Thanks to metatables, Lua can maintain its original design decision to “*keep the language simple and small*” (Ierusalimsky et al, 2001b), while still being able to cope with a variety of programming concepts (Section “Code Structure / Programming Paradigms” of Lua Developers, 2020).

As an example, we show the implementation of some basic concepts of object-oriented programming. The example² is listed in Example 4, and it models classes and objects by combining tables, first-class functions—which we already saw in

² Taken from <http://lua-users.org/wiki/ObjectOrientationTutorial>.

Section 2.1—and the metatable mechanism. It also introduces 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 modeled with tables, containing in their fields their attributes.

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 implicit first parameter for these functions, named `self`.

In the last lines of Figure 4 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)` (remember that the first, implicit, parameter of `set.value` is `self`).

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 table provided as argument (the empty table `{}`). The (empty) table returned is our `self` object.

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 can be modified through a *metamethod* that, despite its name, is a *value* associated with a specific key in the metatable. In this case, the key is `__index` (line 2), and the value is the table `MyClass`. The overall effect will be that a miss when looking up for a key in the table (`mc`) will trigger a look up for the same key in the metatable (`MyClass`). This is how `mc["set.value"]` results in the method `set.value` from `MyClass`.

Lua also allows metamethods for an indexing with a non-existing key, for a function call over a value that is not a closure, for unexpected circumstances involving 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.

2.4 Tuples

A Lua function may return several values in what we call a *tuple*. Tuples are particular in the sense that several of Lua constructs treat them specially. Take for instance Example 5 (Ierusalimsky, 2003, Chp. 5.1). The function `maximum` takes an array `a` and returns its maximum element, together with its index. It uses the *iterator*, obtained from the function `ipairs` of the standard library (line 4), which, in a first call, returns a tuple with three values: an iterator function, the table to be iterated and 0. By calling repeatedly this iterator, supplying the table to be iterated and a position, we can traverse the table following its numeric integer indexes, beginning at 1. Accordingly, the **for-in** construct knows how to use the

Example 5 (Taking the maximum element of an array, together with its index.)

```

1 function maximum (a)
2   local mi = 1          -- maximum index
3   local m = a[mi]       -- maximum value
4   for i, val in ipairs (a) do
5     if val > m then
6       mi = i
7       m = val
8     end
9   end
10  return m, mi
11 end
12
13 print(maximum({8,10,23,12,5})) --> 23 3

```

Example 6 (Converting an array of elements into a tuple.)

```

1 function unpack (t, i)
2   i = i or 1
3   if t[i] ~= nil then
4     return t[i], unpack(t, i + 1)
5   end
6 end
7
8 a1, a2 = unpack({1,2,3}) -- a1 = 1 and a2 = 2
9
10 b1, b2, b3, b4 = unpack({1,2,3}) -- b1 = 1, b2 = 2, b3 = 3, b4 = nil

```

iterator to obtain, at each iteration, an index to the table being iterated and the value at that position. When the for-loop ends, we have in *m* the maximum value, and in *mi* its index in the array. These are returned by **maximum** in line 10.

The size of a tuple does not need to be statically determined. An example of dynamically-constructed tuples is presented in Example 6, where function **unpack** takes an array of elements and produces a tuple containing such elements. There are a couple of niceties in the code that needs some explanation. To begin with, in Lua a function can be called with fewer or extra arguments. Every argument that is not passed by the caller of the function has value **nil**, and every extra argument is simply discarded. This feature allows, for example, to avoid having an auxiliary function performing recursion, and then the main function calling it with the base case. Instead, we can combine the two. In **unpack**, the index *i* used to index the array can be omitted, in which case we ensure it starts at 1 (arrays are 1-based indexed) thanks to the **or** operator (line 2). If *i* is **nil**, then **or** will return its second argument, 1.

Another aspect of tuples worth discussing is that we can choose to ignore elements from a tuple, or even *take* more elements than actually present in the tuple. The first case is illustrated in line 8, where we assign only the first two elements from the tuple returned by **unpack**, discarding the third. The second is

Example 7 (Converting a list of arguments into a table.)

```

1 function pack ( ... )
2   return { ... }
3 end
4
5 t = pack(1,2,3)
6 print(t [1], t [2], t [3], t [4]) -->> 1  2  3  nil

```

illustrated in line 10, where the first three variables gets the three elements from the tuple, while the fourth gets value **nil**.

2.5 Vararg arguments

It is possible to specify a *variadic* function, *i.e.*, a function that can receive an arbitrary quantity of parameters, by appending the *vararg* parameter to the list of formal parameters of a function. Noted with `...`, this argument collects all the extra arguments that are provided to a function. In Figure 7 we see a function dual to `unpack`: it receives a list of arguments of arbitrary length, and returns a *vector* containing those arguments: a table whose elements are indexed by natural numbers.

3 Basics of the formalization

In this section we gently introduce the semantic framework used throughout the paper by providing semantics to `Lua0`, a toy subset of Lua. We write the semantics in a small-step operational semantics with evaluation contexts. Evaluation contexts, taken from Felleisen-Hieb’s Reduction Semantics (RS) (Felleisen et al, 2009), 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 Guha et al (2010); Politz et al (2013, 2012), where RS is similarly applied for successfully formalizing real programming languages. However, we choose a different path with regard to the presentation of our formal semantics and the relation with its mechanization:

1. We emphasize the distinction between what constitutes the language we want to model, and what are the *run-time constructs*: concepts proper of the dynamic semantics required to provide an operational semantics.
2. We reduce the complexity of the *desugaring* process, by staying as close as possible to the source language.
3. We maintain a short-yet-noticeable distance between what is the formal model presented in the paper and its corresponding Redex mechanization.

The first two respond to the need of building the model as comprehensible as possible, mainly for the developers of the language and tools: Point 1 helps distinguishing the syntax developers care about from the constructs that are not visible to them when executing programs, and Point 2 reduces the gap between

Grammar	
$s ::= \text{if } e \text{ then } s \text{ else } s \mid ;$	$\text{binop} ::= \text{and} \mid \text{or}$
$e ::= v \mid e \text{ binop } e \mid \text{unop } e$	$\text{unop} ::= \text{not}$
$v ::= \text{nil} \mid \text{bool_literal}$	
Semantics	
$\frac{v \notin \{\text{nil}, \text{false}\}}{\text{if } v \text{ then } s_1 \text{ else } s_2 \xrightarrow{s/e} s_1} \text{ (IF-T)}$	$\frac{v \in \{\text{nil}, \text{false}\}}{\text{if } v \text{ then } s_1 \text{ else } s_2 \xrightarrow{s/e} s_2} \text{ (IF-F)}$
$\text{not } v \xrightarrow{s/e} \delta(\text{not}, v) \text{ (NOT)}$	$v \text{ binop } e \xrightarrow{s/e} \delta(\text{binop}, v, e) \text{ (BINOP)}$
$\delta(\text{and}, v, e) = \begin{cases} v & \text{if } v \in \{\text{nil}, \text{false}\} \\ e & \text{otherwise} \end{cases}$	
$\delta(\text{or}, v, e) = \begin{cases} v & \text{if } v \notin \{\text{nil}, \text{false}\} \\ e & \text{otherwise} \end{cases}$	
$\delta(\text{not}, v) = \begin{cases} \text{true} & \text{if } v \in \{\text{nil}, \text{false}\} \\ \text{false} & \text{otherwise} \end{cases}$	

Fig. 1: Pure fragment of Lua₀.

the code shown in the rules and actual Lua code, which helps building *trust* in the model (also noted in Bodin et al, 2014; Maffeis et al, 2008).

The motivation behind Point 3—discussed in depth in Section 6—is to present the rules of the language in a more natural form (as seen in many books on semantics, *e.g.*, Pierce (2002)), instead of presenting Redex code directly.

For the small subset of the Lua language presented in this section, we have two fragments: *pure* statements and expressions (following Lua’s distinction of statements and expressions), and *stateful* (*i.e.*, memory changing) statements. We describe each of them in isolation, each with their own *relation*. Then, we compose the two using a third relation which will also deal with execution order, providing semantics to entire programs.

3.1 The pure fragment of Lua₀

We show the grammar and semantics for *stateless* programs in Figure 1. As statements (s) we only include conditional branching and skip ($;$). The condition is an expression e , which can be a *value* (v), a fully-applied and infix binary operator (binop), or a fully-applied and prefix unary operator (unop). Values are **nil** or boolean literals (**true** or **false**). Operators are the logical connectives **and**, **or**, and **not**. Of course, with this language we are not able to write any useful program, but in the coming sections we will grow the language until we reach Lua.

The operational semantics for the pure fragment is modeled with the $\rightarrow^{s/e}$ relation between statements and expressions. Rule IF-T states that if the conditional

Grammar	Run-time terms
$s ::= \dots \mid \text{local } x = e \text{ in } s \mid x = e$	$e ::= \dots \mid r$
Semantics	
$\frac{\sigma' = (r, v), \sigma}{\sigma : \text{local } x = v \text{ in } s \rightarrow^\sigma \sigma' : s[x \backslash r]} \text{ (LOCAL-DECL)}$	
$\frac{\sigma' = \sigma[r := v]}{\sigma : r = v \rightarrow^\sigma \sigma' ;} \text{ (LOCAL-ASSGN)} \quad \sigma : r \rightarrow^\sigma \sigma : \sigma(r) \text{ (LOCAL-DEREF)}$	

Fig. 2: Stateful fragment of Lua_0 .

of the **if** is any value different from **nil** and **false**, then it is considered **true**, and therefore the **then** branch is taken. Note that we write above the line the conditions in which the rule applies. We omit the line in the case that no condition is required. Rule IF-F states that, for **false** or **nil**, the **else** branch is taken.

The reader may wonder why we have not put **true** instead of v in the first rule, since there are no other values. The reason is that we will grow the language to have more values, like numbers, and we will still take this rule as is. Indeed, in Lua, a value like the number 0 is considered **true**, unlike in languages like C.

As for the semantics of expressions, instead of providing operational rules for each operator, we follow Guha et al (2010) and use a function, called *the interpretation function* δ , which provides meaning to operators using a *declarative* style of semantics. That is, we put the emphases on what is the result obtained from using an operator, rather than describing how each operator produces a result step-by-step. The benefits of using this style are more clearly seen in coming sections when providing semantics to more complex operations than arithmetic operators. To maintain cohesion in our semantics, we will use δ to provide semantics to every primitive operator and service.

In this section, δ provides meaning to the shortcut boolean operators, in which the right operand of a binary operator is not expected to be a value. As with the conditional, a value that is distinct from **false** and **nil** is considered **true**.

A final note regarding the model introduced so far: to present a familiar grammar to the Lua programmer we maintain the syntactic distinction between statements and expressions. However, we do not let it to scale up to the level of the semantics relations. Firstly, this distinction is merely a concrete grammar issue (Lua Developers, 2009). Secondly, previous experience with the formulation of a formal semantics for Lua (Soldevila et al (2017)) shows that defining semantics relations following that distinction does not result in an improved understanding of the concepts, but it does result in an unnecessary proliferation of relations.

3.2 The stateful fragment

We extend the language presented so far with imperative features, namely imperative variables (Figure 2). Statements are extended with local variable definitions

and assignment. There are two things to note about a variable declaration: first, as mentioned in §2.2, they must be preceded with **local**; otherwise, in Lua, they are global. Second, we make here the first of a few modifications to Lua’s syntax: we explicitly declare the scope of a variable with the **in** keyword, absent in real Lua. This minor change in the grammar allows us to have a very simple representation of state that, as will become clear, brings several benefits.

The operational semantics of imperative features is commonly understood in terms of state and state change, therefore, we enrich our semantics with a model of state: 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 the domain of σ , referred as to $\text{dom}(\sigma)$, we do not force any specific representation and just require it to be a finite set, with elements that must be syntactically represented and different from any other syntactic object in the language. We further assume it is always possible to obtain a fresh reference from the store. We will write $(r, v), \sigma$ to mean the extension of store σ with reference r pointing to value v , and we will assume that r is *fresh*, *i.e.*, not in the domain of σ .

For the semantics of the new constructs we use a new relation \rightarrow^σ , which maps a pair of a store σ and a term t (either a statement or an expression) to another pair of a new store σ' and the resulting term t' . The first rule (LOCAL-DECL) models the declaration of a local variable. When the definition is a value v , we put it in the store mapped with a fresh reference r . Then, we replace each occurrence of variable x in the program s with r , denoted as $s[x \setminus r]$.

By substituting every occurrence of the variable with a fresh reference (*e.g.*, Felleisen, 1987), we obtain two benefits: we avoid having to carry around the *environment* (a mapping between variables and references), and we obtain a simple representation for closures (*c.f.*, §4.2.3). As a little downside, it forces us to add a semantic component to the language, making references be part of the grammar of expressions. We call a *run-time term* such an extension to the language needed to express its semantics. In more advanced sections they will also serve us to cleanly modularize the semantics.

As for variable assignment (LOCAL-ASSGN), we update the store using the notation $\sigma[r := v]$ to mean a new store in which the value of r is replaced with v (formally, a σ' such that $\text{dom}(\sigma') = \text{dom}(\sigma)$, where $\sigma'(r) = v$ and $\forall r' \in \text{dom}(\sigma'), r' \neq r \Rightarrow \sigma'(r') = \sigma(r')$). Note that the assignment reduces to a skip, indicating that there is nothing else to do for this particular statement. Finally, rule LOCAL-DEREF shows that references are implicitly dereferenced.

3.3 Executing entire programs

We have already defined two different relations, each of them computing a bit of a program: $\rightarrow^{s/e}$ relates stateless statements or expressions and \rightarrow^σ stateful ones. Now we are ready to combine the two relations to perform the execution of a complete program. To that effect we define the \mapsto relation. This relation selects the next term to be executed within a program, called the *redex*, and forwards it to the corresponding relation. We are aiming at a deterministic semantics, so there should be only one redex in every program.

Evaluation contexts

$$E ::= [] \mid \text{if } E \text{ then } s \text{ else } s \mid \text{local } x = E \text{ in } s \\ \mid x = E \mid E \text{ binop } e \mid \text{unop } E$$

Standard relation

$$\frac{t \xrightarrow{s/e} t'}{\sigma : E[t] \mapsto \sigma : E[t']} \text{ (FWD-PURE)} \quad \frac{\sigma : t \xrightarrow{\sigma} \sigma' : t'}{\sigma : E[t] \mapsto \sigma' : E[t']} \text{ (FWD-}\sigma\text{)}$$

Fig. 3: Semantics of Lua₀ programs.

σ	t	E
\emptyset	local $b = \text{false}$ in	$[]$
$\mapsto (r1, \text{false})$	if not b then $b = \text{true}$ else $b = \text{false}$	if not $[]$ then ... else ...
$\mapsto (r1, \text{false})$	if not $r1$ then $r1 = \text{true}$ else $r1 = \text{false}$	if $[]$ then ... else ...
$\mapsto (r1, \text{false})$	if not false then $r1 = \text{true}$ else $r1 = \text{false}$	$[]$
$\mapsto (r1, \text{false})$	if true then $r1 = \text{true}$ else $r1 = \text{false}$	$[]$
$\mapsto (r1, \text{false})$	$r1 = \text{true}$	$[]$
$\mapsto (r1, \text{true})$	$;$	

Fig. 4: Trace of execution of a simple program.

We capture the notion of redex syntactically with an *evaluation context* (Figure 3): a program with a *hole* $[]$ specifying where the redex is to be found. For every constructor of a term from our language, and for a given position into the term where the next redex must be found, there will be one evaluation context's constructor labeling that position with $[]$. The determinism of the semantics emanates then from the fact that, for every program, either it is a *final* computation, or there is only one way of decomposing it into an evaluation context and a term, and that term can be executed using only one rule (see §5).

We can observe from the definition of E the expected evaluation order: in an **if** statement, the guard must be evaluated first. For defining local variables, we evaluate the rvalues first. In a binary operation, we evaluate the left operand first.³ Note that, unlike with the grammar presented so far, for simplicity we are using a single category of evaluation contexts for statements and expressions.

As stated above, the semantics of programs is given with the \mapsto relation. $E[t]$ denotes an evaluation context where its hole has been filled with the term t , yielding a program. This relation simply forwards execution to the relations defined in previous sections.

3.4 Example

Figure 4 presents the reduction of a simple program that flips boolean variable *b*. In the right-most column we write the corresponding evaluation context considered for the step, leaving to the reader the other two components of a reduction step: the fragment of the program filling the hole and the rule being used. Note how in the first step the variable *b* is replaced with reference *r1*.

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 completes the semantics with execution order, to be able to explain the execution of complete programs, and error handling.

4.1 Stateless Lua

We extend the stateless subset of Lua_0 presented in §3 with new statements (while loops, breaks, composition of statements) and new values and expressions (numbers and strings, with their corresponding operations). To ease the presentation we present each addition separately, starting with the new statements.

4.1.1 Statements

Figure 4a shows the new statements. As mentioned, these are composition of statements (essentially, one statement after the other) and **while** loops (with their **break** operation). We purposely do not include **for** loops and instead translate them into **while** loops, following Lua’s reference manual definition (Lua Dev. Team, 2015, 3.3.5).

As for the semantics, we start with the composition of statements (SEQ): once the left-most statement has been executed to an end (;), the execution continues with what follows (*s*).

More interesting is the semantics of while loops. First, a while loop is labeled with a BREAK label (WHILE-START), renaming **while** to **Siter**. The labeling and **Siter** are new run-time statement created for the purpose of executing whiles: the label marks the point from which a **break** should continue the execution, and **Siter** is necessary to avoid repeatedly unfolding a while, stacking up labels.

A loop marked with **Siter** is then unfolded as usual, using an **if-then-else** to check the guard and perform a new iteration (WHILE-ITER). We took from Guha et al (2010) the idea of labeling the while loop to mark the exit point of a **break**;

³ Our definition enforces left-to-right evaluation of expressions. Even if this is left unspecified in Lua’s reference manual, that’s how expressions are evaluated in the two most popular implementations of Lua, the reference interpreter and LuaJIT (lua-jit.org).

Grammar	Evaluation contexts
$s ::= \dots \mid s \ s \mid \mathbf{while} \ e \ \mathbf{do} \ s \mid \mathbf{break}$	$E_{ff} ::= [] \mid \mathbf{if} \ E_{ff} \ \mathbf{then} \ s \ \mathbf{else} \ s$ $\mid \mathbf{local} \ x = E_{ff} \ \mathbf{in} \ s \mid x = E_{ff}$ $\mid E_{ff} \ \mathbf{binop} \ e \mid \mathbf{unop} \ E_{ff} \mid E_{ff} \ s$
Run-time terms	
$s ::= \dots \mid \mathbf{Siter} \ e \ \mathbf{do} \ s \mid \langle s \rangle_{label}$	$E ::= \mathbf{extend} \ E_{ff} \ \mathbf{with} \ \langle E \rangle_{label}$
$label ::= \mathbf{BREAK}$	
Semantics	
$\begin{aligned} & ; s \rightarrow^{s/e} s \ (\mathbf{SEQ}) \qquad \mathbf{while} \ e \ \mathbf{do} \ s \rightarrow^{s/e} \langle \mathbf{Siter} \ e \ \mathbf{do} \ s \rangle_{\mathbf{BREAK}} \ (\mathbf{WHILE-START}) \\ & \mathbf{Siter} \ e \ \mathbf{do} \ s \rightarrow^{s/e} \mathbf{if} \ e \ \mathbf{then} \ s ; \mathbf{Siter} \ e \ \mathbf{do} \ s \ (\mathbf{WHILE-ITER}) \\ & \qquad \qquad \qquad \mathbf{else} \ ; \\ & \langle E_{ff} \llbracket \mathbf{break} \rrbracket \rangle_{\mathbf{BREAK}} \rightarrow^{s/e} ; \ (\mathbf{WHILE-BREAK}) \qquad \langle ; \rangle_{\mathbf{BREAK}} \rightarrow^{s/e} ; \ (\mathbf{WHILE-END}) \end{aligned}$	

Fig. 4a: The stateless subset of Lua (statements).

however, our label is inserted at run-time, while in the aforementioned work it is inserted when *desugaring* the code. Our take reduces the complexity of the desugared code, but at the expense of requiring the extra run-time construct **Siter** and the labeling step. While in this particular case the gain is little, in the case of function calls (§4.2.3) the gain is more evident. For consistency, thus, we decided to do the same for while loops.

When the execution finds a **break** inside a labeled block (**WHILE-BREAK**), the whole program contained in the inner-most **BREAK** label is discarded, effectively modeling the jump out of the body of the loop. To model the program within the label we use a new category of evaluation context, E_{ff} , which represents a program in which no other labeled term or **while** loop occurs. In order to avoid repetition, we decided to split the evaluation context in two: first, E_{ff} is like the evaluation context from §3.3, also adding the case for composition of statements; second, we define the new E context as an extension of E_{ff} . Extending an evaluation context like E_{ff} to create a new evaluation context E has the effect of renaming each occurrence of E_{ff} with E .

When the execution of the loop ends normally (**WHILE-END**), we just discard the label.

4.1.2 Expressions

We turn our attention to the evaluation of expressions, which we break in two: evaluation of equality and the application of the remaining binary operators. We omit the evaluation of boolean expressions and of the unary minus operator since they follow a similar pattern.

Binary operators, except equality, are treated in Figure 4b. As for the grammar, we define the class *strictbinop* to differentiate binary operators that requires the two values to be evaluated prior to execution, unlike with the boolean operators.

<div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">Grammar</div> $ \begin{aligned} v &::= \dots \mid \textit{number_literal} \mid \textit{string_literal} \\ e &::= \dots \mid e \textit{ strictbinop } e \\ \textit{strictbinop} &::= + \mid - \mid * \mid / \mid ^ \mid \% \mid .. \\ &\quad \mid < \mid \leq \mid > \mid \geq \mid == \\ \textit{unop} &::= \dots \mid - \mid \# \end{aligned} $	<div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">Run-time terms</div> $ \begin{aligned} e &::= \dots \mid \langle e \rangle_{\textit{label}} \\ \textit{label} &::= \dots \mid \text{BINOPWO} \mid \text{EQFAIL} \end{aligned} $ <div style="border: 1px solid black; padding: 2px; margin-bottom: 5px;">Evaluation contexts</div> $ \begin{aligned} E_{\textit{lf}} &::= \dots \mid E_{\textit{lf}} \textit{ strictbinop } e \\ &\quad \mid v \textit{ strictbinop } E_{\textit{lf}} \end{aligned} $
<div style="border: 1px solid black; padding: 2px; margin: 5px auto; width: 80%;">Semantics of binary operators</div>	
$ \frac{op \in \{+, -, *, /, ^, \%, .., <, <= \} \quad \delta(op, v_1, v_2) \in v}{v_1 \textit{ op } v_2 \rightarrow^{s/e} \delta(op, v_1, v_2)} \quad (\text{BINOP}) $	
$ \frac{op \in \{+, -, *, /, ^, \%, .., <, <= \} \quad \delta(op, v_1, v_2) \notin v}{v_1 \textit{ op } v_2 \rightarrow^{s/e} \langle v_1 \textit{ op } v_2 \rangle_{\text{BINOPWO}}} \quad (\text{BINOP-WO}) $	

Fig. 4b: The stateless subset of Lua (binary operators).

Note how we extend the evaluation context $E_{\textit{lf}}$, and assume that E is extended likewise.

Turning to their semantics, BINOP states that the actual semantics of the application of op over operands v_1 and v_2 is captured by the function $\delta(op)$. The semantics of the binary operators shown is mostly standard, the only exception being an internal step of coercion done between numbers and strings: in every Lua's primitive operator and library service, each time that the semantics dictates that a number is expected, it is possible to provide a string representation of a number, and internally Lua tries to coerce it to a number, following certain lexical conventions. The same happens when a string is expected and a number is provided. These details, together with the remaining aspects of the standard semantics of binary operations, are all captured in δ . The only role of the semantics relations is to interface with δ , checking that it is actually possible to apply op over operands v_1 and v_2 , *i.e.*, the application returns a value v , and not, for example, an error object.

On the other hand, rule BINOP-WO shows the case when the binary operator cannot be applied successfully ($\delta(op, v_1, v_2) \notin v$). This could mean that we tried to perform arithmetic over values that cannot be interpreted as numbers, or tried to compare values for which there is no standard order relation built-in in Lua. In this case we label the expression with information about what has gone wrong (BINOPWO, where WO stands for Wrong Operands). We therefore extend the class of expressions to include labeled expressions, and extend the set of labels with the new labels, in Figure 4b.

At this point, execution is stuck here. If we were not to extend the semantics any further, these error labels will only serve the purpose of informing the user what caused her program to fail. Later on, we will see how metatables try to resolve the situation, either by providing an alternative value, or by *throwing* an error (§4.4). Such errors can be caught with the primitive `xpcall` (§4.5).

$$\begin{array}{c}
\frac{\delta(==, v_1, v_2) = \mathbf{true}}{v_1 == v_2 \rightarrow^{s/e} \mathbf{true}} \text{ (EQ-TRUE)} \\
\\
\frac{\delta(==, v_1, v_2) = \mathbf{false} \quad \delta(\text{type}, v_1) \neq \text{"table"} \vee \delta(\text{type}, v_2) \neq \text{"table"}}{v_1 == v_2 \rightarrow^{s/e} \mathbf{false}} \text{ (EQ-FALSE)} \\
\\
\frac{\delta(==, v_1, v_2) = \mathbf{false} \quad \delta(\text{type}, v_1) = \text{"table"} \quad \delta(\text{type}, v_2) = \text{"table"}}{v_1 == v_2 \rightarrow^{s/e} (\Downarrow v_1 == v_2 \Downarrow_{\text{EQ_FAIL}})} \text{ (EQ-FAIL)}
\end{array}$$

Fig. 4c: The stateless subset of Lua (equality).

In case of equality, the behavior is a bit more intricate (Figure 4c). As with previous operators, we abstract into $\delta(==)$ the corresponding details of equality comparison, but we perform some extra processing of the result.

In essence, $\delta(==)$ compares numbers and strings by value, and tables and closures by reference—with values of different types being always distinct. That is, unlike other dynamic languages like JavaScript, $0 == "0"$ evaluates to **false**.

When the objects are equal (EQ-TRUE), we simply return **true**. But if the objects are distinct (for instance, they have distinct types), we have two options: if the operands involved are not tables (EQ-FALSE), equality comparison reduces to **false**. Otherwise, if they are tables (EQ-FAIL), instead of returning **false** we label the expression and defer its result to the metatable mechanism (§4.4). The metatable mechanism will then look up for user-defined functions (*metamethods*) implementing a comparison method for non-equal table values. In that way, Lua allows for the definition of arbitrary (reflexive) relations over table values.

4.2 Imperative Lua

In this section we grow the imperative fragment presented in section 3 to include tables (§4.2.1), multiple local variables definition and assignment (§4.2.2), and functions (§4.2.3). The latter is presented here for reasons that will become clear in time.

4.2.1 Tables

Tables are formalized in Figure 5a. Essentially, they are mutable associative arrays: data structures where we can store key/value pairs, where keys cannot be **nil** or **nan** (a value in the set of numbers used for undefined or not representable numeric results), and values cannot be **nil**.

Tables are manipulated through references, so, to that end, we create a new mapping between tables' identities (*tid*) and tables, called θ . The elements from the domain of θ are considered values and satisfy the same properties as references in σ , with the reasonable addition that former should be syntactically distinguishable from the latter. The image of θ only contains tables and any table-related meta-data. We capture a new store θ resulting from, either, extending a previous store

Syntax	Evaluation context
$e ::= \dots \mid \{ field, \dots \}$ $\quad \mid x [e]$ $field ::= e \mid [e] = e$ $efield ::= v \mid [v] = v$	$E_{lf} ::= \dots \mid E_{lf} [e] \mid v [E_{lf}]$ $\quad \mid \{ efield, \dots, [E_{lf}] = e, field, \dots \}$ $\quad \mid \{ efield, \dots, [v] = E_{lf}, field, \dots \}$ $\quad \mid \{ efield, \dots, E_{lf}, field, \dots \}$
Run-time terms	
$v ::= \dots \mid tid$ $label ::= \dots \mid \text{WRONGKEY} \mid \text{NONTABLE}$	
Semantics	
$\frac{\theta_2 = (tid, (\text{addkeys}(\{efield_1, \dots\}), \mathbf{nil})), \theta_1}{\theta_1 : \{efield_1, \dots\} \rightarrow^\theta \theta_2 : tid} \text{ (TABLE-CONSTR)}$	
$\frac{\delta(\text{rawget}, tid, v_1, \theta_1) \neq \mathbf{nil} \quad \theta_2, tid = \delta(\text{rawset}, tid, v_1, v_2, \theta_1)}{\theta_1 : tid [v_1] = v_2 \rightarrow^\theta \theta_2 : ;} \text{ (TABLE-UPDATE)}$	
$\frac{\delta(\text{rawget}, tid, v_1, \theta) = \mathbf{nil}}{\theta : tid [v_1] = v_2 \rightarrow^\theta \theta : (tid [v_1] = v_2) \mathbb{W}\text{RONGKEY}} \text{ (TABLE-UPDATE-WK)}$	
$\frac{\delta(\text{type}, v_1) \neq \text{"table"}}{\theta : v_1 [v_2] = v_3 \rightarrow^\theta \theta : (v_1 [v_2] = v_3) \mathbb{N}\text{ONTABLE}} \text{ (TABLE-UPDATE-NT)}$	

Fig. 5a: Formalization of tables (excerpt).

with a new pair *tid*/table or changing the content of a previously stored table, using the same devices and notation as with σ (§3.2).

The construction of a table is formalized in Rule TABLE-CONSTR (Figure 5a), which acts when all of its fields have been evaluated. Fields are evaluated from left to right, with each key being evaluated before each value. Note the new syntax non-terminal *efield* to denote evaluated fields, which can be just a value or a key/value pair (denoted $[v] = v$). The meta-function *addkeys* adds absent keys—namely, consecutive natural numbers—to fields of the form v . Note that we are actually mapping a fresh table id *tid* with an ordered pair: the first component contains the actual table and the second one the id of the metatable or **nil**, if absent, as is in this case. Metatables are discussed in depth in §4.4.

Assignment to table fields has three cases: the successful update of the field (TABLE-UPDATE), and two special scenarios. The first comes from trying to update a non-existing key (TABLE-UPDATE-WK); while the second comes from trying to update an object which is not a table (TABLE-UPDATE-NT). The semantics of these special scenarios can be meta-programmed, so, as we did with similar situations, we just label the term with information about what has occurred, to delegate the execution to the metatables mechanism.

$$\begin{aligned}
\delta(\text{rawget}, v_1, v_2, \theta) &= \delta(\text{error}, \dots) \quad \text{if } \delta(\text{type}, v_1) \neq \text{"table"} \\
\delta(\text{rawget}, tid, v_i, \theta) &= \begin{cases} v_j & \text{if } \theta(tid) = (\{\dots, [v'_i]=v_j, \dots\}, \dots) \\ & \text{and } \delta(==, v_i, v'_i) = \text{true} \\ \text{nil} & \text{otherwise} \end{cases} \\
\delta(\text{rawset}, tid, v_1, v_2, \theta_1) &= (\theta_2, tid), \\
\text{where } \begin{cases} v_1 \notin \{\text{nil}, \text{nan}\} \\ \theta_1(tid) = (\{\dots, field_{i-1}, [v'_1]=v_3, field_{i+1}, \dots\}, \dots) \\ \delta(==, v_1, v'_1) = \text{true} \\ \theta_2 = \begin{cases} \theta_1[tid := (\{\dots, field_{i-1}, field_{i+1}, \dots\}, \dots)] & \text{if } v_2 = \text{nil} \\ \theta_1[tid := (\{\dots, field_{i-1}, [v'_1]=v_2, field_{i+1}, \dots\}, \dots)] & \text{if } v_2 \neq \text{nil} \end{cases} \end{cases} \\
\delta(\text{rawset}, tid, v, v', \theta_1) &= (\theta_2, tid), \\
\text{where } \begin{cases} v \notin \{\text{nil}, \text{nan}\} \\ \theta_1(tid) = (\{[v_1]=v'_1, \dots, [v_n]=v'_n\}, \dots) \\ \forall k \in \{v_1, \dots, v_n\}, \delta(==, v, k) = \text{false} \\ \theta_2 = \begin{cases} \theta_1 & \text{if } v' = \text{nil} \\ \theta_1[tid := (\{[v]=v', [v_1]=v'_1, \dots, [v_n]=v'_n\}, \dots)] & \text{if } v' \neq \text{nil} \end{cases} \end{cases} \\
\delta(\text{rawset}, v_1, v_2, v_3, \theta) &= (\theta, \delta(\text{error}, \dots)), \text{ if } \delta(\text{type}, v_1) \neq \text{"table"} \vee v_2 \in \{\text{nil}, \text{nan}\}
\end{aligned}$$

Fig. 5b: Formalization of tables (primitives for table assignment and indexing).

To know if a key exists in a table and to update the field we use two services of Lua, `rawget` and `rawset`, formalized in Figure 5b. $\delta(\text{rawget}, tid, v, \theta)$ yields either the value associated with v in $\theta(tid)$ or `nil`, if there is no associated value. Note that, to determine which field is being indexed with the given key, we call the equality comparison operator over keys, instead of syntactically requiring the same values.

$\delta(\text{rawset}, tid, v_1, v_2, \theta)$ yields a new θ where the table referenced by `tid` associates v_2 with value v_1 . In the first equation we consider an existing key, in which case we remove or update the field, depending of the value v_2 . Note how `nil` and `nan` cannot be the keys of a table field. As with `rawget`, the keys are compared using the equality operator. The second equation of `rawset` considers the case when the key does not exist in the table. In this case, the pair is added, only if the value is not `nil`. The last equation considers the error cases.

Accessing a field of a table follows a similar pattern so we omit the rules of this operation.

4.2.2 Local variables

We introduce a small change in the language in order to support multiple variable definition and assignment (Figure 6). Essentially, we can have arbitrary lists of variables and terms as the lvalues and rvalues of a local declaration, respectively. Similarly, we can have arbitrary lists in the assignment, with the addition that we can update variables *and* tables' fields.

Grammar	Evaluation context
$s ::= \dots \mid \text{var}, \dots = e, \dots$ $\quad \mid \text{local } x, \dots = e, \dots \text{ in } s$ $\text{var} ::= x \mid e [e]$ $\text{eval} ::= r \mid v [v]$	$E_{ff} ::= \dots$ $\quad \mid \text{local } x, \dots = v, \dots, E_{ff}, e, \dots \text{ in } s$ $\quad \mid \text{eval}, \dots, E_{ff} [e], \dots = e, \dots$ $\quad \mid \text{eval}, \dots, v [E_{ff}], \dots = e, \dots$ $\quad \mid \text{eval}, \dots = v, \dots, E_{ff}, \dots$
Semantics	
$\frac{k \geq 1 \quad v_{n-k+1} = \text{nil}, \dots, v_n = \text{nil}}{\text{eval}_1, \dots, \text{eval}_n = v_1, \dots, v_{n-k} \xrightarrow{s/e} \text{eval}_1, \dots, \text{eval}_n = v_1, \dots, v_{n-k}, v_{n-k+1}, \dots, v_n} \text{ (ASSGN-FEWER)}$	
$\frac{k \geq 1}{\text{eval}_1, \dots, \text{eval}_n = v_1, \dots, v_n, \dots, v_{n+k} \xrightarrow{s/e} \text{eval}_1, \dots, \text{eval}_n = v_1, \dots, v_n} \text{ (ASSGN-MORE)}$	
$\frac{n \geq 2}{\text{eval}_1, \dots, \text{eval}_n = v_1, \dots, v_n \xrightarrow{s/e} \text{eval}_1 = \text{eval}_1 \quad \text{eval}_2, \dots, \text{eval}_n = v_2, \dots, v_n} \text{ (ASSGN-SPLIT)}$	
$\frac{\sigma' = (r_1, v_1), \dots, (r_n, v_n), \sigma}{\sigma : \text{local } x_1, \dots, x_n = v_1, \dots, v_n \text{ in } s \xrightarrow{\sigma} \sigma' : s[x_1 \setminus r_1, \dots, x_n \setminus r_n]} \text{ (LOCAL-DECL)}$	

Fig. 6: Multiple variables declaration and assignment (excerpt).

For multiple-variable assignment, the following steps are performed. First, the execution evaluates all lvalues before the rvalues, as indicated by the evaluation contexts. Second, the quantity of rvalues is adapted to match the lvalues, either adding extra **nil** rvalues when there are fewer rvalues than lvalues (ASSGN-FEWER), or discarding the extra rvalues when there are more rvalues than lvalues (ASSGN-MORE). Once the number of lvalues matches that of rvalues, a multiple assignment is decomposed into a sequence of single assignments. Note that we need not to make any further step: each assignment will be handled by the corresponding rules (LOCAL-ASSGN, from §3.2, and TABLE-UPDATE, from §4.2.1).

The introduction of local variables follows the same steps, which we omit for brevity. The creation of new references is modeled with a simultaneous substitution operation (LOCAL-DECL, which subsumes the previously defined rule with the same name in §3.2).

4.2.3 Functions, function applications, and tuples

It might at first look suspicious to consider Lua's functions and application as belonging to the imperative subset. However, Lua's functions are first-class values internally represented with closures closing over their external variables' references, and, also, the parameters of a function are modeled as imperative variables.

Grammar

$$\begin{aligned}
v &::= \dots \mid \mathbf{function} \ell (x, \dots) s \mid \mathbf{function} \ell (x, \dots, \dots) s \\
s &::= \dots \mid \mathbf{\$statFcall} e (e, \dots) \mid \mathbf{\$statFcall} e : x (e, \dots) \\
&\quad \mid \mathbf{return} e, \dots \\
e &::= \dots \mid e (e, \dots) \mid e : x (e, \dots) \mid \dots \mid (e)
\end{aligned}$$

Run-time terms

$$\begin{aligned}
e &::= \dots \mid \langle e, \dots \rangle \mid \langle s \rangle_{label} \\
label &::= \dots \mid \mathbf{RETSTAT} \mid \mathbf{RETEXP} \mid \mathbf{WFUNCALL}
\end{aligned}$$

Evaluation context

$$\begin{aligned}
E_{el} &::= v, \dots, E_{lf}, e, \dots \\
E_{lf} &::= \dots \mid \mathbf{\$statFcall} E_{lf} (e, \dots) \mid \mathbf{\$statFcall} v (E_{el}) \mid \mathbf{return} E_{el} \\
&\quad \mid E_{lf} (e, \dots) \mid v (E_{el}) \mid E_{lf} : x (e, \dots) \mid (E_{lf}) \\
&\quad \mid \langle E_{lf} \rangle_{\mathbf{RETEXP}} \mid \langle E_{lf} \rangle_{\mathbf{RETSTAT}}
\end{aligned}$$

Fig. 7a: Functions and function calls (grammar).

Therefore, we need the state to define functions and function's applications. Tuples are also considered here because they are required to model functions.

Figure 7a shows the grammar involved in the formalization of function definitions and calls. The first thing to note is that functions are values, and that each function is labeled with a new countable set of elements ℓ , which should not to be confused with a function's name: the label ℓ —inserted at *desugaring*—is used for closure comparison (§4.2.4), and in practice it is akin to the line number, in the source code. As an example, **function** $f()$ **end** is desugared as $f = \mathbf{function} \ell_1 () \mathbf{end}$.

Functions have two constructors; the difference being the *vararg* arguments ... (not to be confused with ellipses, also used in the grammar). This special token refers to the extra arguments passed to a function beyond of those explicitly named in its definition. Within the function body, ... is treated as a *tuple*. Tuples are run-time expressions, noted as $\langle e, \dots \rangle$, and cannot be constructed by the user.

Then, we have the **return** statement, which allows the return of multiple values; function calls, with a special syntax for calling an object's method; and parenthesized expressions, which takes the first value of a list of values.

As for function calls, we need to distinguish their occurrences: they can appear in a statement, as in $f(); \mathbf{print('hi')}$, or in an expression, as in $\mathbf{print}(f())$. A function call used as an expression can return none, one or more values, while a function call used as a statement is useful just for its side-effects, with any value returned from such function being discarded. Our parser is in charge of distinguishing and compiling each case into syntactically different objects, tagging with **\\$statFcall** the latter case.⁴

⁴ This distinction helped simplify the model, as we do not need to define new notions of evaluation contexts to distinguish among either case—a task that proved to be cumbersome and difficult to maintain through the evolution of the model.

$$\begin{array}{c}
\frac{1 \leq i \leq \min(m, n) \Rightarrow v'_i = v_i \quad i > m \Rightarrow v'_i = \mathbf{nil} \quad \sigma' = (r_1, v'_1), \dots, (r_n, v'_n), \sigma}{\sigma : (\mathbf{function} \ell (x_1, \dots, x_n) s) (v_1, \dots, v_m) \rightarrow^\sigma \quad \sigma' : \llbracket s [x_1 \setminus r_1, \dots, x_n \setminus r_n] \rrbracket_{\text{RETEXP}}} \text{(E-CALL)} \\
\\
\frac{i > m \Rightarrow v'_i = \mathbf{nil} \quad \text{tuple} = \langle v_{n+1}, \dots, v_m \rangle \quad \sigma' = (r_1, v'_1), \dots, (r_n, v'_n), \sigma}{\sigma : (\mathbf{function} \ell (x_1, \dots, x_n, \dots) s) (v_1, \dots, v_m) \rightarrow^\sigma \quad \sigma' : \llbracket s [x_1 \setminus r_1, \dots, x_n \setminus r_n, \dots \setminus \text{tuple}] \rrbracket_{\text{RETEXP}}} \text{(E-CALLVARG)} \\
\\
\frac{\delta(\text{type}, v) \neq \text{"function"}}{v (v_1, \dots, v_n) \rightarrow^{s/e} \llbracket v (v_1, \dots, v_n) \rrbracket_{\text{WFCALL}}} \text{(E-CALLWRONG)} \\
\\
v : \text{name} (e_1, \dots, e_n) \rightarrow^{s/e} v["\text{name}"] (v, e_1, \dots, e_n) \text{(E-MCALL)}
\end{array}$$

Fig. 7b: Functions and function calls (semantics, 2nd step).

Function calls are not plain β -contractions: they allow us to call a given function with an arbitrary quantity of parameters, regardless of what is specified in its signature, discarding or completing with extra **nil** values as required, like the following simple example shows:

Example 8 (Calling a function with different number of parameters.)

```

function f(x1, x2) return x1, x2 end
print(f())      -->> nil nil
print(f(1))     -->> 1  nil
print(f(1,2))   -->> 1  2
print(f(1,2,3)) -->> 1  2

```

Additionally, the result of a call might be in a position where the returned values must be appended to existing ones or trimmed. For instance, consider the following short example, in which **maximum** is the function from Example 5:

Example 9 (Returned values are appended to the argument list or trimmed.)

```

m = maximum({1, maximum{1,2,2,3}})  -- m: 4

```

Remember that **maximum** returns the greatest number of a list together with its position. To understand this result, first note that the inner call returns 3 and 4. These values are then appended to the outer list, resulting in {1, 3, 4}. Therefore, the second call to **maximum** yields values 4 and 3, the second value being trimmed out and the first one being assigned to **m**.

In order to specify these behaviors, we take the following actions:

1. Evaluate the function position and then the arguments, from left-to-right.
2. Adjust the list of actual parameters and perform the function call.
3. According to the place of the call (statement or expression), discard the returned values or put them in a tuple.
4. In the case of an expression, distinguish if the tuple must be appended to other values or trimmed.

$$\begin{aligned}
(\cdot)_{\text{RETSTAT}} &\rightarrow^{s/e} ; (\text{S-RETSKIP}) & (E_{\text{f}}[\text{return } v, \dots])_{\text{RETSTAT}} &\rightarrow^{s/e} ; (\text{S-RETURN}) \\
(\cdot)_{\text{RETEXP}} &\rightarrow^{s/e} < > (\text{E-RETSKIP}) \\
(E_{\text{f}}[\text{return } v, \dots])_{\text{RETEXP}} &\rightarrow^{s/e} < v, \dots > (\text{E-RETURN}) \\
(E_{\text{f}}[\text{return } v, \dots])_{\text{BREAK}} &\rightarrow^{s/e} \text{return } v, \dots (\text{S-RETURNBREAK})
\end{aligned}$$

Fig. 7c: Functions and function calls (semantics, 3rd step).

The *first step* is specified with the help of the evaluation contexts, as can be seen in Figure 7a. As a shorthand we introduce a new non-terminal E_{el} to mean a list of arguments whose elements to the left of the evaluation point are fully evaluated.

The *second step* is shown in Figure 7b, only for the case of function calls in expressions, as the other case is analogous. Rule E-CALL models a simple call, with the parameters of the function treated as mutable variables, so a fresh reference is used to model each parameter. If there are fewer arguments, the value **nil** is assigned to the remaining parameters of the function, and if there are more, they are simply silently ignored. The resulting term—the body of the function with the fresh references replacing the parameters—is labeled with a new run-time expression $(\cdot)_{\text{RETEXP}}$. This labeling serves two purposes: first, it allows to recognize the function call as, in this case, an expression, something needed when reasoning about tuples (see below); second, it marks the place where a **return** statement must jump to. In the case of a function call in a statement, the label is RETSTAT.

Rule E-CALLVARG shows the case of a vararg function call, with the difference residing on what is done with surplus arguments: instead of being ignored they are put into a tuple, which replaces the vararg expression (...) in the body of the function. Formally, we treat the vararg mark occurrence in the signature of a function as a binding occurrence, with special scoping rules in order to account for a particularity discussed in Manura (2007, Issue #1).

Rule E-CALLWRONG describes the exceptional situation that might be handled by the metatable mechanism: a function call over a non-function value. As with previous special situations, we just label the whole expression with a tag documenting what happened.

Finally, E-MCALL describes method invocation: it is simply translated into a table look-up, with the object being injected as the first argument of the function.

The *third step* is split in two. First, the body of the function is executed to an end, either a skip or a **return**. Second, as listed in Figure 7c, the values are processed as follow: a function call as a statement simply returns skip (S-RETSKIP), discarding the values if needed to (S-RETURN). Instead, in an expression, it returns a tuple, either empty (E-RETSKIP) or containing the values (E-RETURN). A **return** might occur inside the body of a while (S-RETURNBREAK), in which case we must jump outside of the loop.

Contexts for tuple processing (excerpt)

$$\begin{aligned}
E_{\text{tel}} &::= v, \dots, \llbracket \cdot \rrbracket, e_1, e_2 \dots \\
E_{\text{t}} &::= \text{if } \llbracket \cdot \rrbracket \text{ then } s \text{ else } s \mid \text{local } x, \dots = E_{\text{tel}} \text{ in } s \mid \text{eval}, \dots = E_{\text{tel}} \\
&\quad \mid (\llbracket \cdot \rrbracket) \mid \{ \text{efield}, \dots, \llbracket \cdot \rrbracket, \text{field}_1, \text{field}_2, \dots \} \mid \dots \\
E_{\text{ael}} &::= v, \dots, \llbracket \cdot \rrbracket \\
E_{\text{a}} &::= \text{local } x, \dots = E_{\text{ael}} \text{ in } s \mid \text{eval}, \dots = E_{\text{a}} \mid \{ \text{efield}, \dots, \llbracket \cdot \rrbracket \} \mid \dots
\end{aligned}$$

Appending function (excerpt)

$$\begin{aligned}
\text{append}(\text{eval}, \dots = v_1, \dots, \llbracket \cdot \rrbracket, < v_2, \dots >) &= \text{eval}, \dots = v_1, \dots, v_2, \dots \\
\text{append}(\text{eval}, \dots = v_1, \dots, \llbracket \cdot \rrbracket, < >) &= \text{eval}, \dots = v_1, \dots
\end{aligned}$$

Semantics

$$\begin{aligned}
E_{\text{t}} \llbracket < v_1, v_2, \dots > \rrbracket &\rightarrow^{s/e} E_{\text{t}} \llbracket v_1 \rrbracket \text{ (TUPLE-ONE)} \\
E_{\text{t}} \llbracket < > \rrbracket &\rightarrow^{s/e} E_{\text{t}} \llbracket \text{nil} \rrbracket \text{ (TUPLE-ZERO)} \\
E_{\text{a}} \llbracket < v_1, \dots > \rrbracket &\rightarrow^{s/e} E_{\text{a}} \llbracket v_1, \dots \rrbracket \text{ (TUPLE-APPEND)}
\end{aligned}$$

Fig. 7d: Functions and function calls (semantics, 4th step).

The *fourth step* kicks in once we have a tuple. As noted in Example 9, we need to know if values are appended to some existing list of values or trimmed to just obtain the first value. The process requires some machinery but is simple in essence. If the *context*, in which the tuple is immersed, requires a list of values, the tuple's values are appended to the values existing in the context, else, the first value is taken. To know the context, as the name suggests, we use the same type of contexts used for evaluation.

Figure 7d shows the details of the fourth step. Rule TUPLE-ONE replaces the tuple with its first value. For this rule to be executed, the tuple must be in a context E_{t} (the *t* stands for *truncate*), which for brevity is shown partially. This means that the tuple is, for instance, in the conditional of an **if-then-else**, or in a list of values (but not at the end), or in a parenthesized expression, *etc.* If the tuple is empty, under the same context, it is replaced with **nil** (TUPLE-ZERO).

Rule TUPLE-APPEND shows the case when the tuple must be appended to a list. This rule uses a new context, E_{a} (*a* for *append*), which only applies if the tuple is at the end of a list of values. Then, the result of the tuple is computed with the meta-function **append**, which is also shown in abridged form. This function takes the context and the tuple and simply insert the values of the tuple to the end of the list, or does nothing if the tuple is empty.

With the semantics shown in this section we can understand the example from the introduction (Example 1). We explain it in two bits. First, we have function **f** returning **nil**:

```
function f() return nil end
```



```
print(f(), f(), f())  $\longrightarrow$  nil nil nil
```

The execution of the **print** statement occurs after each call to **f** is resolved. Since they are in the positions of expressions, after E-CALL we get $\langle \text{return nil} \rangle_{\text{RETEXP}}$. With an empty E_f context, we perform one E-RETURN step and get a tuple with a **nil** inside. In the first two calls to **f**, the tuple is in a position where a value is expected, therefore after executing TUPLE-ONE we simply get **nil**. In the third call, instead, we append the tuple to the values computed so far (TUPLE-APPEND). Therefore, we end with **print(nil, nil, nil)**, which in turn prints the three **nils**.

Different is the case when we do not return nothing:

```
function g() end  
print(g(), g(), g())  $\longrightarrow$  nil nil
```

Note that the body of **g** is an implicit skip. Therefore, in the first two calls to **g** we obtain a skip that is converted to an empty tuple by E-RETSKIP, and then to **nil** by TUPLE-ZERO. In the last call, as in the previous case, TUPLEAPPEND applies and **append** discards the empty tuple. Therefore, we now end with **print(nil, nil)**.

4.2.4 Function comparison and caching

Lua 5.2 admits an unspecified caching of closures: when their definitions have *no observable difference* (Ierusalimsky et al, 2013, Chp. 8.1). We implement this caching in a way that is compatible with the official interpreter, correctly mimicking the following interactions:⁵

Example 10 (Caching of functions.)

```
1 f = function() end  
2 g = function() end  
3 print(f == g)  $\longrightarrow$  false  
4 h = function() return function() end end  
5 print(h() == h())  $\longrightarrow$  true
```

The first two functions are equal, however, Lua creates a closure for each of them, as line 3 shows. But when the function is created from the same function definition, it does cache its definition (lines 4-5).

As mentioned at the beginning of §4.2.3, our function's definitions include a label ℓ , which is what distinguishes different occurrences of definitions in the program. Observe the partially-unsugared code obtained from Example 10:

```
f = (function $1 () ; end)  
g = (function $2 () ; end)  
print(f == g)  
h = (function $3 () return (function $4 () ; end) end)  
print(h() == h())
```

Each function created from calling **h** shares its label (**\$4**), and therefore we can treat each instance as the same. But this is not enough to identify a function, as the following example shows:

⁵ Note that LuaJIT 2.0, the other major implementation of (part of) Lua 5.2, does not perform caching of closures.

Grammar

$\text{svc} ::= \text{assert} \mid \text{error} \mid \dots$
 $e ::= \dots \mid \text{\$builtin svc } (e, \dots)$

Semantics

$$\begin{array}{c}
\frac{\text{svc} \in \{\text{type}, \text{assert}, \text{error}, \text{pcall}, \text{select}, \dots\}}{\text{\$builtin svc } (v_1, \dots, v_n) \rightarrow^{s/e} \delta(\text{svc}, v_1, \dots, v_n)} \text{ (SVC-NOSTATE)} \\
\\
\frac{\text{svc} \in \{\text{ipairs}, \text{next}, \text{pairs}, \text{getmetatable}, \dots\}}{\theta : \text{\$builtin svc } (v_1, \dots, v_n) \rightarrow^\theta \theta : \delta(\text{svc}, v_1, \dots, v_n, \theta)} \text{ (SVC-CONST)} \\
\\
\frac{\text{svc} \in \{\text{rawset}, \text{setmetatable}\} \quad (\theta_2, v) = \delta(\text{svc}, v_1, \dots, v_n, \theta_1)}{\theta_1 : \text{\$builtin svc } (v_1, \dots, v_n) \rightarrow^\theta \theta_2 : v} \text{ (SVC-UPDATE)}
\end{array}$$

Fig. 8: Interface with the δ function.

Example 11 (Same function yet different closures.)

```

for i=1,10 do print(function() return i end) end
-->> function: 0x1c4d970 function: 0x1c4e3d0 ...

```

Note how at each iteration it prints a new address: the function is not cached. The reason is that its body refers to a variable, i , which is different at each iteration (**for** loops makes the iteration variable immutable by creating a fresh variable at each iteration). Therefore, caching the closure would be unsound. In our formalization we consider the whole function definition as its identification, therefore, including the environment of the closure where the variable i is captured.

4.3 Built-in services

As we saw throughout this work, the interpretation function δ defines the semantics of built-in services (**type**, **rawget**, ...) and operators (**+**, **..**, ...). Originally introduced in ISWIM (Felleisen et al (2009)) for the purpose of abstracting the semantics over a set of primitive operators, we use it here motivated by different purposes. As mentioned in §3, instead of describing *operationally* the semantics of built-in services and primitive operators, we use a declarative style; *i.e.*, we specify just the result of the call to a service or the application of an operator, instead of describing how we obtain such results. This approach reflects the style of the reference manual: there is no operational specification from which the results can be obtained, just a description of what is expected from calling a given service or applying a given operator. While there may be some services or operators for which it could be possible to extract a simple operational semantics, for purposes of cohesion we apply the same approach to the semantics of all of them.

Calling a service: There are two different types of calls to services: those that are internal of other services and operators (*e.g.*, the iterator function `next` calls the `type` service), and those that are invoked by the user. This distinction is important because it is possible to change the binding of a service, but only in the latter situation:

Example 12 (Overriding a built-in service.)

```
print(type({})) -->> table
type = function () return 'foo' end
print(type({})) -->> foo
next(1) -->> error: bad argument (table expected, got number)
```

As we can see from this example, the `next` function was unaffected by the change to the `type` function, while the user is now unable to call Lua's built-in service.

From the point of view of the formalization, we introduce a new expression **\$builtin** which simply acts as a dispatch to the δ function (Figure 8). Note that we cannot simply let a program refer directly to the δ function: δ is a meta-function, and it does not belong to the programming language. Therefore, we need an interface *within the language*. Then, prior to the execution of a program, the `.ENV` table is populated with functions from the standard library as wrappers to the **\$builtin** constructor.

The semantics of the new expression is simple: it just reduces to an application of δ , providing the necessary arguments and processing its output when needed. We distinguish three cases, according to the semantics of the service: if it does not use the store (SVC-NOSTATE); if it *inspects* the objects store (SVC-CONST); or if it *updates* the object store (SVC-UPDATE).⁶ Each rule lists some of the services that fit into each category. The actual list of services includes almost all the built-in basic functions of the Lua language, together with several services from the libraries `math`, `string` and `table`.

4.4 Metatables

We have seen in repeated occasions how several operations required the intervention of metatables; usually in special situations, like trying to add two values that are not numbers nor strings. Metatables are plain tables that let programmers specify *metamethods* for resolving such situations, associating specific string keys with—typically—functions. For instance, `"__add"` is the key for associating a handler to the addition operator.

We first explain how to obtain and set metatables. Then, we show with an example how the metatable mechanism triggers once a labeled term is found.

4.4.1 Setting and getting metatables.

For each Lua type except tables (and userdata, not included in our model), it is only possible to define one single metatable. That is, for example, the type of

⁶ We do not include the value store since our subset implemented so far does not require it, but this may change in the future.

$$\begin{aligned}
& \text{indexmeta}(tid, v, \theta) \doteq \delta(\text{rawget}, tid, v, \theta) \\
& \text{protmeta}(tid, \theta) \doteq \text{indexmeta}(tid, \text{"_metatable"}, \theta) \\
& \text{prot?}(tid, \theta) \doteq \text{protmeta}(tid, \theta) \neq \text{nil} \\
\\
& \delta(\text{setmetatable}, tid, v, \theta_1) = (tid, \theta_2) \quad \begin{array}{l} \text{if } \delta(\text{type}, v) \in \{\text{"table"}, \text{"nil"}\} \\ \wedge \neg \text{prot?}(tid, \theta_1) \\ \wedge \theta_2 = \theta_1[tid := (\theta_1(tid)(1), v)] \end{array} \\
\\
& \delta(\text{setmetatable}, v_1, v_2, \theta) = (\delta(\text{error}, \text{"..."}), \theta) \quad \begin{array}{l} \text{if } \delta(\text{type}, v_2) \notin \{\text{"table"}, \text{"nil"}\} \\ \vee \text{prot?}(tid, \theta_1) \\ \vee \delta(\text{type}, v_1) \neq \text{"table"} \end{array} \\
\\
& \delta(\text{getmetatable}, tid, \theta) = \begin{cases} \text{protmeta}(tid, \theta) & \text{if } \text{prot?}(tid, \theta) \\ \theta(tid)(2) & \text{otherwise} \end{cases} \\
\\
& \delta(\text{getmetatable}, v, \theta) = \begin{cases} \text{nil} & \text{if } tid_{\delta(\text{type}, v)} \notin \text{dom}(\theta) \\ tid_{\delta(\text{type}, v)} & \text{if } tid_{\delta(\text{type}, v)} \in \text{dom}(\theta) \\ & \wedge \neg \text{prot?}(tid_{\delta(\text{type}, v)}, \theta) \\ \text{protmeta}(tid_{\delta(\text{type}, v)}, \theta) & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 9: Primitives for setting and getting metatables.

numbers has one metatable ruling every possible interaction with numbers. For tables, instead, it is possible to define a metatable per value. To this effect, the basic library provides two primitives, `setmetatable`, to set the metatable of a given table, and `getmetatable`, to obtain the metatable associated with a given value (of any type). Being services of the basic library, we provide a declarative description of their behavior using δ . Their semantics is shown in Figure 9.

The first equation of `setmetatable` shows the normal behavior: we are setting v as the metatable of tid , and v is itself a table or `nil`; and tid is not *protected*. If these conditions apply, the output of `setmetatable` is the identifier of the table together with a new object store, resulting from updating θ_1 to include v as the second component of the value stored for tid . If v is `nil`, this has the effect of removing any previous association between the table and a metatable.

If a metatable has a field having key `"_metatable"`, it is said to be protected. The concept is specified by the predicate `prot?`. Semantically, a protected metatable cannot be overridden.

The second equation of `setmetatable` deals with different erroneous situations: trying to set a metatable for a value which is not a table; trying to set, as a metatable, a value which is not a table or `nil`; or trying to modify a protected metatable. In any of these cases, an error is issued. We write `...` to abbreviate the message, which depends on the condition that produced the failure.

The first equation for `getmetatable` shows its normal behavior: when applied over a table value tid , it either returns the value set in its `"_metatable"` field, if present, or the second component in the θ store associated with tid .

More interestingly is the situation when we want to obtain a metatable for a non-table value (the second equation of `getmetatable`). As it turns out, for values such as numbers or strings, it is possible to set the metatable by using the C API or, in the official interpreter, by calling the `debug.setmetatable` function. In

$$\begin{aligned}
\text{new_index}(\theta : \llbracket v_1 [v_2] = v_3 \rrbracket_{\text{LABEL}}) &= \theta : \mathbf{\$statFcall} \ v_4 \ (v_1, v_2, v_3) \\
&\quad \text{if } \begin{cases} \text{LABEL} \in \{\text{WRONGKEY}, \text{NOTABLE}\} \\ v_4 = \text{indexmeta}(v_1, \text{"_newindex"}, \theta) \\ \delta(\text{type}, v_4) = \text{"function"} \end{cases} \\
\\
\text{new_index}(\theta : \llbracket v_1 [v_2] = v_3 \rrbracket_{\text{LABEL}}) &= \theta : v_4 [v_2] = v_3 \\
&\quad \text{if } \begin{cases} \text{LABEL} \in \{\text{WRONGKEY}, \text{NOTABLE}\} \\ v_4 = \text{indexmeta}(v_1, \text{"_newindex"}, \theta) \\ v_4 \neq \mathbf{nil} \\ \delta(\text{type}, v_4) \neq \text{"function"} \end{cases} \\
\\
\text{new_index}(\theta_1 : \llbracket tid [v_1] = v_2 \rrbracket_{\text{WRONGKEY}}) &= \theta_2 : ; \\
&\quad \text{if } \begin{cases} \text{indexmeta}(tid, \text{"_newindex"}, \theta_1) = \mathbf{nil} \\ (\theta_2, tid) = \delta(\text{rawset}, tid, v_1, v_2, \theta_1) \end{cases} \\
\\
\text{new_index}(\theta_1 : \llbracket tid [v_1] = v_2 \rrbracket_{\text{WRONGKEY}}) &= \theta_2 : \mathbf{\$err...} \\
&\quad \text{if } \begin{cases} \text{indexmeta}(tid, \text{"_newindex"}, \theta_1) = \mathbf{nil} \\ (\theta_2, \mathbf{\$err...}) = \delta(\text{rawset}, tid, v_1, v_2, \theta_1) \end{cases} \\
\\
\text{new_index}(\theta_1 : \llbracket tid [v_1] = v_2 \rrbracket_{\text{NOTABLE}}) &= \theta_2 : \delta(\text{error}, \text{"error updating..."}) \\
&\quad \text{if } \text{indexmeta}(tid, \text{"_newindex"}, \theta_1) = \mathbf{nil} \\
\\
\frac{\text{new_index}(\theta_1 : \llbracket v_1 [v_2] = v_3 \rrbracket_{\text{LABEL}} = \theta_2 : s}{\theta_1 : \llbracket v_1 [v_2] = v_3 \rrbracket_{\text{LABEL}} \xrightarrow{\text{meta}} \theta_2 : s} &\quad (\text{M-UPD})
\end{aligned}$$

Fig. 10: Metatable mechanism for field update.

our model we assume that, if set, the metatables for these types are identified by fixed tables' ids: for a given value v , we assume that its metatable must be identified with $tid_{\delta(\text{type}, v)}$. Then, determining if such metatable is set reduces to ask if $tid_{\delta(\text{type}, v)} \in \theta$. Note that such metatables might be protected too.

4.4.2 Resolving labeled terms (excerpt).

In this section we introduce a new relation $\rightarrow^{\text{meta}}$ that takes a labeled term and acts according to the case that originated the exceptional situation. We do not show all the rules that apply to each different label; instead, we focus on one example rich enough to show how this mechanism works.

Figure 10 describes the $\rightarrow^{\text{meta}}$ relation for field updates. The inner workings of the metatables mechanism is handled by the meta-function `new_index`, while $\rightarrow^{\text{meta}}$ serves as an interface with it (M-UPD). Recall from §4.2.1 that we have two situations, each marked by its distinctive label: either the update is attempted on a value that is not a table (NOTABLE), or the value is a table, yet the key does not belong to it (WRONGKEY). In either case, if there is a function associated with the key `"_newindex"` in the metatable of the object (first equation of `new_index`), the

function is called with the arguments of the update. If, instead, the handler is not a function, nor **nil**, the update is performed on the handler (second equation).

If the handler is **nil**, we need to distinguish according to the label. For **WRONGKEY**, we try creating the field with **rawset**. If it succeeds (third equation), we return the modified θ store and the skip statement. If it fails (fourth equation), because the key is **nil** or **nan**, we propagate the error. Note that we need to inspect the returned error object **\$err**—the result of $\delta(\text{error})$ —which is introduced later in §4.5. Finally, if the label is **NOTABLE**, we just produce an error (last equation of **new_index**).

4.5 Semantics of programs and error handling

We concluded the presentation of the different relations used to provide semantics to Lua. Following §3, we must now provide meaning to entire programs, defining a new relation \mapsto that includes execution order and forwards to the specific relation the next piece of the program to be executed. Since its definition is mostly an extension of the relation shown in Figure 3, but now adding also the θ context, we will omit these definitions for brevity. More interestingly, we develop in this section the treatment of errors in Lua, whose propagation requires its formulation at the level of the \mapsto relation, for soundness reasons.

Errors and protected mode. Lua provides mechanisms to generate and catch errors. Errors are represented with *error objects* or *error messages*. They can be explicitly generated by user code, using the function **error**, and it is possible to attach information about the error using any Lua value. The semantics of error objects consist, essentially, in aborting the execution of the program where they are generated.

In order to catch an error, the user is provided with two functions, **pcall** and **xpcall**, the former being a specialization of the latter. They allow to execute a function under a *protected mode*: any error thrown by the function is caught, avoiding error propagation. For an erroneous function call, **pcall** returns **false** and the error object. Otherwise, it returns **true** together with the values returned by the function call. **xpcall** behaves as **pcall**, but instead of returning the error object in case of error, it forwards it to a user-provided handler function to continue the execution.

Formalization. In formalizing error propagation, we need to take into account that it is a context-sensitive computation, requiring to look at the whole remaining program in order to decide about program termination. Therefore, we need to be cautious if we want to obtain a deterministic model. To understand this, let us consider a program of the form $E \llbracket \text{\$err } v \rrbracket$, where E does not contain a protected mode. What we want for such a program is to completely halt in one step, ending as **\$err** v . This suggests to include a step:

$$E' \llbracket \text{\$err } v \rrbracket \rightarrow^{s/e} \text{\$err } v$$

But, since \mapsto closes over $\rightarrow^{s/e}$ as follows:

$$\frac{t \rightarrow^{s/e} t'}{\sigma : \theta : E'' \llbracket t \rrbracket \mapsto \sigma : \theta : E'' \llbracket t' \rrbracket}$$

Grammar

$s ::= \dots \mid \mathbf{\$err} \ v \mid \langle s \rangle_{\text{PROTMD}}^v \mid \langle s \rangle_{\text{PROTMD}}$
 $E ::= \dots \mid \langle E \rangle_{\text{PROTMD}}^v \mid \langle E \rangle_{\text{PROTMD}}$

Semantics

$$\begin{array}{c}
\frac{E_{np} \neq []}{\sigma : \theta : E_{np} [\mathbf{\$err} \ v] \mapsto \sigma : \theta : \mathbf{\$err} \ v} \text{ (E-TERMINATION)} \\
\\
\langle \langle v, \dots \rangle \rangle_{\text{PROTMD}}^{v'} \rightarrow^{s/e} \langle \mathbf{true}, v, \dots \rangle \text{ (E-PROTTRUE)} \\
\\
\frac{\delta(\text{type}, v_2) = \text{"function"}}{\langle E_{np} [\mathbf{\$err} \ v_1] \rangle_{\text{PROTMD}}^{v_2} \rightarrow^{s/e} \langle v_2(v_1) \rangle_{\text{PROTMD}}} \text{ (E-PROTHANDLER)} \\
\\
\frac{\delta(\text{type}, v_2) \neq \text{"function"}}{\langle E_{np} [\mathbf{\$err} \ v_1] \rangle_{\text{PROTMD}}^{v_2} \rightarrow^{s/e} \langle \mathbf{false}, \text{"error"} \rangle} \text{ (E-PROTHANDLERERR)} \\
\\
\langle v \rangle_{\text{PROTMD}} \rightarrow^{s/e} \langle \mathbf{false}, v \rangle \text{ (E-PROTFALSE)} \\
\\
\langle E_{np} [\mathbf{\$err} \ v_1] \rangle_{\text{PROTMD}} \rightarrow^{s/e} \langle \mathbf{false}, \text{"error"} \rangle \text{ (PROTERR)}
\end{array}$$

$$\delta(\mathbf{xpcall}, v_1, v_2, v_3, \dots) = \langle v_2(v_3, \dots) \rangle_{\text{PROTMD}}^{v_2}$$

Fig. 11: Error object and protected mode.

We have several possible ways of propagating an error, depending on the decomposition of context E between contexts E' and E'' , offending the intended determinism of the semantics.

The common workaround to this problem is to propagate errors directly from \mapsto . In absence of a protected mode to catch an error, it will make the whole program to halt.

Figure 11 shows the new constructions added to the model for formalising errors and protected mode. Error objects will be denoted with the construction $\mathbf{\$err} \ v$, which allows for the inclusion of any Lua value as information attached to the error. $\langle s \rangle_{\text{PROTMD}}^v$ and $\langle E \rangle_{\text{PROTMD}}^v$ represent a statement or a context that must run in protected mode, where v is the error handler passed to \mathbf{xpcall} . We also include the same labels but without the handlers. Error propagation will be decided in terms of the presence or absence of a protected mode. To define the absence, we use a new context E_{np} that is a subset of E but without protected modes (omitted for brevity).

With the given constructions we can formalize what it means for an error to propagate or to be caught in protected mode. The first rule (E-TERMINATION) aborts the whole program if there is no protected mode around the error object. This is the only rule from error handling that is defined in the \mapsto relation.

The remaining rules are defined in the $\rightarrow^{s/e}$ relation, and concerns the execution of code inside a protected mode. If the code executed in protected mode runs until completion without an error (E-PROTTRUE), then its values are appended to the tuple starting with **true**. Instead, if there is an error, we have two situations: if the handler is a function, then it is executed within a protected mode (without a new handler, E-PROTHANDLER). Note how the call is performed within parenthesis, to take just the first value. Also, note how it is not possible to call the handler if it is not a function (E-PROTHANDLERERR), that is, this scenario cannot be meta-programmed through metatables. In this case, we return a tuple starting with **false** and a descriptive error message.

Once the code of a handler is executed in protected mode (denoted by a `PROTMd` context without a handler), we find two options: in a successful execution of the handler, its only returned value is appended to a tuple starting with **false** (to signal that a handled error occurred, E-PROTFALSE), and, in an erroneous execution of the handler, an error is returned (E-PROTERR), exactly as in E-PROTHANDLERERR.

Here concludes our tour of the semantics of the language, proceeding to state its properties and discuss its mechanization in the coming sections.

5 Properties of the semantics

In this section we state two fundamental properties of our model: *soundness* (as the traditional combination of *progress* and *preservation*) and determinism. Their statements were tested using a lightweight approach enabled by Redex (Klein et al, 2012), in which we specify a property and use *random testing* to try to falsify it. Together with the successful execution of the code from the test suite of Lua, they help to build trust in the model. The details of the experience with the mechanization are presented in §6, but it is worth mentioning that we were able to catch, with little effort, several ambiguities and ill-defined cases.

When we say *preservation*, we say that a *well-formed configuration* that can perform a step results in another well-formed configuration. As for *progress*, we say that a well-formed configuration represents the *final computation* of a program, or it can perform a step of computation. In essence, together they allow us to exclude *stuck terms*, computations that are incorrect from the point of view of the semantics, like a program with references not allocated in the store. We explicit all these terms next.

The most important ingredient is the predicate that captures well-formedness of configurations (bottom of Figure 12), which states the conditions that a given configuration $\sigma : \theta : t$ must hold to be well-formed, for the given stores σ and θ , and term t (a statement or expression). This predicate is parametrized by a new context C , whose excerpt can be found at the top of Figure 12, and which captures the local information required to check the well-formedness of t . In essence, C keeps track of local variables, while loops (to know if a **break** may or may not occur in t), *etc.*

A configuration $\sigma : \theta : t$ is well-formed for a context C , written $C \vdash_{wfc} \sigma : \theta : t$, if every term of σ and every table in θ is well-formed, and also t is. When the context is $\llbracket \rrbracket$, we omit it and simply write $\vdash_{wfc} \sigma : \theta : t$. In order to check when a given term is well-formed (w.r.t. the well-formed stores), we use a new relation \vdash_{wft} , and show only a representative set of rules.

Contexts for well-formedness of terms (excerpt)

$C ::= \llbracket \rrbracket \mid \text{while } e \text{ then } C \mid \langle C \rangle_{\text{BREAK}} \mid \text{local } x, \dots = e, \dots \text{ in } C$
 $\mid \text{function } \ell(x, \dots) C \text{ end} \mid \dots$

Well-formedness of terms (selected rules)

$$\frac{C \vdash_{\text{wft}} \sigma : \theta : e \quad C \vdash_{\text{wft}} \sigma : \theta : s_1 \quad C \vdash_{\text{wft}} \sigma : \theta : s_2}{C \vdash_{\text{wft}} \sigma : \theta : \text{if } e \text{ then } s_1 \text{ else } s_2} \text{ (WF-IF)}$$

$$\frac{r \in \text{dom}(\sigma)}{C \vdash_{\text{wft}} \sigma : \theta : r} \text{ (WF-REF)}$$

$$C' \llbracket \text{local } x_1, \dots, x, x_2, \dots = e, \dots \text{ in } C'' \rrbracket \vdash_{\text{wft}} \sigma : \theta : x \text{ (WF-VARLOCAL)}$$

$$C' \llbracket \text{function } x_1 (x_2, \dots, x, x_3, \dots) C'' \rrbracket \vdash_{\text{wft}} \sigma : \theta : x \text{ (WF-VARFUN)}$$

$$C \llbracket \text{while } e \text{ do } C' \text{ end} \rrbracket \vdash_{\text{wft}} \sigma : \theta : \text{break} \text{ (WF-BREAKWHILE)}$$

$$C \llbracket \langle C' \rangle_{\text{BREAK}} \rrbracket \vdash_{\text{wft}} \sigma : \theta : \text{break} \text{ (WF-BREAKUNFOLDED)}$$

$$\frac{C \vdash_{\text{wft}} \sigma : \theta : e \dots \quad C \llbracket \text{local } x, \dots = e, \dots \text{ in } \llbracket \rrbracket \rrbracket \vdash_{\text{wft}} \sigma : \theta : s}{C \vdash_{\text{wft}} \sigma : \theta : \text{local } x, \dots = e, \dots \text{ in } s} \text{ (WF-LOCAL)}$$

$$\frac{C \vdash_{\text{wft}} \sigma : \theta : e \quad C \llbracket \text{while } e \text{ do } \llbracket \rrbracket \rrbracket \vdash_{\text{wft}} \sigma : \theta : s}{C \vdash_{\text{wft}} \sigma : \theta : \text{while } e \text{ do } s} \text{ (WF-WHILE)}$$

$$\frac{C \vdash_{\text{wft}} \sigma : \theta : v_2 \quad C \vdash_{\text{wft}} \sigma : \theta : v_1 \quad v_1 \in \text{dom}(\theta) \quad v_2 \notin \text{dom}(\theta(v_1)(1))}{C \vdash_{\text{wft}} \sigma : \theta : \langle v_1[v_2] = v_3 \rangle_{\text{WRONGKEY}}} \text{ (WF-WRONGKEY)}$$

Well-formedness of configurations

$$C \vdash_{\text{wfc}} \sigma : \theta : t \doteq \forall r \in \text{dom}(\sigma), C \vdash_{\text{wft}} \sigma : \theta : \sigma(r) \quad (1)$$

$$\wedge \forall tid \in \text{dom}(\theta), C \vdash_{\text{wft}} \sigma : \theta : \theta(tid) \quad (2)$$

$$\wedge C \vdash_{\text{wft}} \sigma : \theta : t \quad (3)$$

Fig. 12: Well-formedness of configurations (excerpt).

\vdash_{wft} is defined recursively over the structure of the term t , as can be seen, for example, in WF-IF. Some rules require checking side conditions, like WF-REF, in which the reference must exist in the corresponding store, in order to check for the absence of *free occurrences* of references, that would render the corresponding term *stuck*. Additionally, some rules require specific local information: for a variable x , we need to check that the context contains a **local** or **function** construct binding x (WF-VARLOCAL and WF-VARFUN, respectively). Similarly, a **break** can only occur if

the context contains a yet-to-be-executed **while** (WF-BREAKWHILE) or an already-executed one (WF-BREAKUNFOLDED). The local information is extended each time we check certain constructs. For instance, WF-LOCAL shows how local variables are inserted in the context C prior to checking the inner statement s . Similarly, WF-WHILE shows how the **while** construct is inserted in the context C in order to check the body of the loop. Finally, for labeled expressions or statements we need to make sure the conditions that created the labeling holds. For instance, for WRONGKEY we need to check that the key indeed does not exist (WF-WRONGKEY).

Now that we have a precise meaning of what constitutes a well-formed configuration, and prior to state the main property of the formalization, we enunciate what *final computations* are:

Definition 1 We say that a statement s is *final* if it is one of the following, for some value v :

- (a) **Serr** v
- (b) **return** v, \dots
- (c) **;**

Lemma 1 (Soundness of \vdash_{wfc}) For given stores σ, θ , statement s , if $\vdash_{wfc} \sigma : \theta : s$, then one of the following situations holds:

- (a) s is final.
- (b) There exists a configuration $\sigma' : \theta' : s'$ such that $\sigma : \theta : s \mapsto \sigma' : \theta' : s'$ and $\vdash_{wfc} \sigma' : \theta' : s'$.

The previous property does not state that the step is unique. In order to have deterministic semantics there should always be at most one redex into a given term:

Lemma 2 (Unique decomposition) Given stores σ, θ , and statement s , such that $\vdash_{wfc} \sigma : \theta : s$, then one of the following situations holds:

- (a) s is final.
- (b) There exist only one evaluation context E and term t such that s matches against the pattern $E[t]$, in such a way that only one of the following is a redex:
 - (1) t
 - (2) $\sigma : t$
 - (3) $\theta : t$

From these properties we infer two relevant corollaries:

Corollary 1 A well-formed configuration runs until completion, or loops forever. In other words, every erroneous situation is handled properly by the error handling mechanism:

For every configuration $\sigma : \theta : s$, such that $\vdash_{wfc} \sigma : \theta : s$, just one of the following situations holds:

- (a) The execution diverges.
- (b) The execution ends with a configuration $\sigma' : \theta' : s'$, such that $\vdash_{wfc} \sigma' : \theta' : s'$ and s' is final.

Finally, the following corollary of soundness of \vdash_{wfc} justifies the representation of closures explained in section 4.2.3:

Corollary 2 (No free variables introduced) *For a given closed term s (i.e., without unbounded occurrences of variables), and well-formed configuration $\sigma : \theta : s$, if there exists σ' and θ' such that $\sigma : \theta : s \mapsto \sigma' : \theta' : s'$, then s' is a closed term.*

In other words, once the focus of execution has reached a closure, every external variable in its body has been replaced by the corresponding reference, *i.e.*, the environment of the closure is completely embedded into its body, avoiding stuck terms.

6 Mechanization

In this section we discuss the mechanization in Redex of the model presented in this paper. We begin by explaining the differences between the formal model and its mechanization, and then discuss the results of executing the tests from the test suite and of randomly testing fundamental properties.

6.1 Correspondence with the formal model

During the process of formalizing the language, we went back-and-forth between our pen-and-paper formalization and its mechanization in Redex. This tool was of great help, as the resulting mechanized formalization can be executed and traced; therefore, we could recognize problems in our first attempts at formalizing the language, experiment with new ideas before landing them on the formalization and testing the mechanization against the test suite for the official interpreter, prior to verifying the desired properties.

Given that we produced two documents—the formalization on paper and its mechanization—we did not follow closely the original philosophy of Redex, consisting in considering *semantic models as software artifacts* (Klein et al, 2012). Though it is indeed an interesting approach, with a real potential to optimize the time spent in a formalization effort, we still need to observe that reading the mechanization alone is at times a difficult task: some concepts from our model are not possible to express directly into Redex, forcing us to obtain a mechanization that loses some abstraction and clarity. This occurs, for example, when dealing with concepts specified in terms of a quantification over an arbitrary predicate: Lua function calls, operations that filter fields of a given table, *etc.*

This said, the formalization presented in this paper and its mechanization are remarkably similar. There are, however, four groups of differences that are worth mentioning:

Particularities of the tool: Redex fully supports the set of concepts of reduction semantics with evaluation contexts; namely, the definition of languages, evaluation contexts, meta-functions and reduction rules. In particular, reduction rules are described as relations over *terms* of a defined language. In practice, this means that we must represent some semantics elements of our model as phrases of a language. For instance, we presented the stores as partial functions in §3, but they must be defined as terms in the mechanization. This is observable for instance in the rule for updating a variable:

$$\frac{\sigma' = \sigma[r := v]}{\sigma : r = v \rightarrow^\sigma \sigma' ;;} \text{ (LOCAL-ASSGN)}$$

The same rule in Redex is written as:

```
[-->σ ((vsp_1 ... (r v_1) vsp_2 ...) : (r = v_2))
      ((vsp_1 ... (r v_2) vsp_2 ...) : \;)]
Local-Assgn]
```

where the σ store is now described in terms of a phrase, and **vsp** is a new syntactic class for describing a pair containing a reference and a value, something that we did not considered part of the language in the pen-and-paper formalization.⁷ Also, the substitution of the value associated with the reference **r** by the value **v_2**, is now performed using pattern-matching and a *term template* (Casey Klein and Findler, 2011), which builds a new σ storage that differs from the first one just in the value associated with the reference **r**: now, **v_1** is replaced by **v_2** (a pair of a reference and a value is noted with **(r v)**).

While in this simple example the distance is short, in some complex rules the difference becomes more noticeable.

Particularities of the use of the semantics: When using our model to build a tool to check misuses of weak tables (Soldevila et al, 2020), we had to introduce two changes in order to better emulate the behavior of a garbage collector. First, we change the definition of local variables (*e.g.*, LOCAL-DECL in §4.2.2) and the instantiation of formal parameters of functions (*e.g.*, E-CALL in §4.2.3) in a way that the variables are not discarded once they are replaced by references. Instead, we keep them within a **local** statement or function call, annexed to the code in which they are replaced. This allows for a simple representation of the set of variables that are in scope, regardless of their presence in what remains to be computed. Second, and for similar reasons, closures are manipulated through references, analogous to the manipulation of tables. In fact, this is closely as to how the official implementation of Lua works.

Particularities of the official interpreter: In order to stay close to the official interpreter, and its corresponding test suite, we had to enrich our specification with implementation details that might not be strictly considered as belonging to the semantics of the language. In many cases, these were just corner cases of the services of the language and its standard library (discussed in §6.3). But a more interesting example is how to deal with mutually recursive references between metatables and handlers, as the following program illustrates:

Example 13 (Mutually recursive references in metatables.)

```
1 local a = {}
2 a.__newindex = a
3 setmetatable(a, a)
4 a[1] = 2
```

⁷ The language of patterns from Redex is expressive enough to be able to impose, for example, the expected representation invariant for a term that describes a finite functional mapping.

From the point of view of the semantics of metatables (§4.4), the previous program is an infinite loop: the instruction in line 4 triggers the metatable mechanism, looking for the metatable of `a` to index it with the key `"_newindex"` (rule `M-UpdNotFun` in Figure 10), obtaining the handler. But this is `a` itself, as specified in line 2, repeating the assignment in an endless loop.

While the reference manual does not specify a particular way to handle these situations, the official interpreter implements some intelligence that is capable of recognizing these loops. In particular, the execution of the previous program results in an error with the message `"loop in settable"`.

Given that the previous behavior is tested in the official test suites, we include it into our semantics by maintaining, in the instruction term, the *tids* to the metatables being used while trying to solve a labelled term. In this way, we can recognize when the same metatable is being indexed repeatedly and raise the appropriate error.

*Particularities of the implementation: the **load** service.* Like many scripting languages, Lua includes a service to execute an arbitrary program coded in a string. This service, called **load**, has little interest from the point of view of its semantics: given a string `s`, it returns a function that, when called, executes the code written in `s`. The service returns **nil** if there are syntax errors or unbound identifiers in the resulting program.

From the point of view of its mechanization, however, it is interesting to mention that its inclusion was rather trivial: our parser and the desugaring process are both written in Racket, the underlying language of Redex.

6.2 Tests coverage

From the 25 files present in Lua’s test suite (Lua Dev. Team, 2013) we successfully run 12 files (including garbage collection; see Soldevila et al (2020)), with varying degrees of coverage (Figure 13). Each file from the test suite is a sequence of assertions about the expected outcome of the code. Those files and lines not tested are due to the following reasons:

- Language features not covered by our formalization: coroutines, the **goto** statement, some standard library functions (mostly related with file handling) and other standard libraries implemented in C (`bit32`, `coroutine`, `debug`, `io`, *etc.*).
- Tests covering implementation details of the interpreter, and not relevant to the language’s semantics: tail call implementation, manipulation of large tables (not tested for performance reasons), generation of bytecode, and the like.

In pursuit of reasonable running times for each module of tests, we divided each `.lua` file in smaller modules. Also, when necessary, we reduced the iterations performed by loops and added explicit calls to the garbage collector (16 LOCs, affecting results in a few other lines). Additionally, while looking for more coverage, we changed a handful lines in modules `errors.lua` and `strings.lua` to remove services that are not yet included. In the case of `errors.lua`, the lines were in functions used throughout the code to test the error messages. By modifying them, we obtained weaker predicates for testing our formalization of the error mechanism, without testing details of the produced error messages.

File	Features tested	LOCs (total/tested)	Coverage
calls.lua	functions and calls	221/162	73.3%
closure.lua	closures	198/139*	70.2%
constructs.lua	syntax and short-circuit opts.	237/193*	81.4%
errors.lua	errors	317/137*	43.2%
events.lua	metatables	302/300	99.3%
gc.lua	garbage collection	445/181*	40.1%
locals.lua	local variables and environments	114/75	65.8%
math.lua	numbers and math lib	219/191	87.2%
nextvar.lua	tables, next, and for	355/226*	63.7%
sort.lua	(parts of) table library	133/46	34.9%
strings.lua	strings and string library	233/88*	37.8%
vararg.lua	vararg functions	95/95	100%
Total		2869/1833	63.9%

* These files were slightly modified in order to increase performance and coverage of tests. Counted LOCs do not include extra lines added for these purposes.

Fig. 13: Lua 5.2's test suite coverage.

With the addition of parallel threads of execution, enabled by modern-day processors, the whole selected fraction of Lua's test suite can take, roughly, 40 minutes to completion without testing garbage collection, and 1:50 hours including it.⁸ Therefore, it is possible to add the execution of these tests in CI to ensure that new features and fixes included in the language do not break existing ones.

The accompanying material includes the modified files with relevant information to inspect the changes and run the tests.

Improving performance thanks to semantics-preserving garbage collection. As mentioned, in order to reduce the timings of the test we introduce explicit calls to our mechanization of Lua's garbage collector. To understand the reasons, consider the execution of the following Lua program:

```
for i = 1, 100 do ; end
for i = 1, 100 do ; end
```

Without the intervention of the garbage collector, each iteration of both loops generates garbage that will persist in the configurations until the end of the execution. Given the computational model of reduction semantics—in which at each step the whole configuration is matched against the (possible) several patterns of

⁸ On a system running Ubuntu 18.04, Racket v8.0, and 6 parallel processes on an Intel Core i5-8265U CPU @ 1.60GHz × 8, with 16GB of ram.

the left side of each execution rule—increasing the size of the store impacts negatively on the performance of this task. Hence, if we reduce the size of the store during run-time, in a way that preserves the semantics of the program, we obtain some performance benefits without compromising soundness.

For instance, the execution of the previous program takes roughly 7 minutes (on a similar hardware). By adding a single call to the garbage collector, right after the first loop, the execution time is shortened by more than a half. This said, in smaller configurations the performance of the execution of the garbage collector can also impact negatively. For example, if we consider a modification of the previous program where both loops iterate 10 times, the program that performs garbage collection after the first loop is slower (by a couple of tenths of a second) than the one that runs until the end carrying all the garbage generated by both loops. We understand this as the cost of having to execute our garbage collector, which inherits some of the complexities of Lua’s—yet not its efficiency.

Together with the modularization of the code into smaller files, adding explicit calls to the garbage collector allowed us to mitigate the known problems with performance that comes with mechanizing an interpreter in Redex (Politz et al, 2013), and in a semantic-preserving way (Soldevila et al, 2020).

6.3 Random testing of properties

Redex’s features for implementing formal systems and random testing (`redex-check`(Klein, 2009)) allowed us to mechanize the desired check for well-formedness of programs and to stress test its soundness property, together with the expected determinism of the semantics.

Interestingly, the randomly generated terms showed unexpected errors and omissions in our mechanization, that were not caught during testing against the test suite. As it turns out, `redex-check` proceeds completely *uninhibited*, unaware of the semantics (Klein et al, 2012), providing intricate, hard-to-decipher terms that proved to be useful to uncover:

1. Missing border cases and implicit coercions of the parameters of some library services.
2. Erroneously handled cases of the table constructor, mostly related with the generation of numeric keys.
3. Misinterpretations from our part of the reference manual.
4. Errors in our first attempts at defining a suitable notion of *well-formed configuration* for our semantics.

With respect to 1, these omissions can be understood as behavior that is assumed or which does not belong to the essentials of the semantics. Naturally, the later cases cannot be eluded from our semantics, since we aim at an executable model. As an example, the results of the following invocations of library services cannot be inferred from the reference manual:

Example 14 (Special cases in library services.)

```

1 -- sub(s,i,j): substring of s from position i and with length j
2 print (string.sub("abc", 1, -3))    -->> "a"
3
```

```

4 -- rep(s,n,sep): repeat string s n times, each copy separated by sep
5 print (string.rep ("abc", 1.99999, ""))    -->> "abc"

```

In the call to **string.sub** the third argument is negative, a situation that—while mentioned in the reference manual—its semantics is not actually specified. In the call to **string.rep** the parameter *n* is not an integer. In this case the official implementation takes the floor, although this is not mentioned in the manual.⁹

Even though our first attempt at mechanizing the semantics contained several of these errors, it actually managed to pass the tests. We remark that it is not necessarily a problem of the tests of the interpreter, since several services are just wrappers around already tested C’s libraries.

In this scenario, using **redex-check**, for testing easily verifiable properties like progress, served as a lightweight approach to stress testing our implementation in search of omissions and errors, with a reasonable time overhead.

Preparation of test cases. Naturally, if run *inhibited*, most of the terms generated by **redex-check** are ill-formed, since the grammar does not enforce well-formedness. To alleviate this, it is possible to provide **redex-check** with a function that manipulates the generated term and turns it into a well-formed one, when possible.

Additionally, to obtain a better coverage of the rules, **redex-check** offers the possibility of generating terms following the pattern of the left-hand side of each of the rules. However, while we obtain full-coverage, this selection of patterns is too restrictive: the properties that we tested are predicates quantified over a well-formed configuration $\sigma : \theta : s$, *i.e.*, not just terms that matches against the left-hand side of the semantic rules. Therefore, to increase the confidence that the rules are complete, we generate just a fraction of the total cases following the rules, and ask **redex-check** to generate the remaining terms by just following the pattern $\sigma : \theta : s$.

Properties tested and results. For every well-formed term generated we tested progress and preservation of well-formedness, together with the determinism of our semantics. For a run of $5 * 10^4$ attempts we obtain, in an average of ten runs, $28.7 * 10^3$ well-formed configurations that successfully pass the tests, with a complete coverage of the rules.

7 Related work

As mentioned throughout the text, the major source of inspiration are the line of work by Guha et al (2010); Politz et al (2012, 2013). The first two of these works formalize JavaScript, and the third one formalizes Python. At a broad view, we share with these works several of the characteristics of the semantic models and their corresponding mechanization in Redex, although there are several differences important to note. First, our model includes particularly interesting features, like

⁹ Note about numeric representations: The official interpreter follows the IEEE 754 standard, while Racket has a richer set of numbers and behaviors, even including complex numbers. Therefore, it is required to convert to the required representation of numbers in order to correctly emulate the behavior of Lua.

Lua’s mechanism for meta-programming. While similar to Python’s metaclass construction, this feature is not covered in Politz et al (2013). Second, we purposely do not focus on identifying a core set of features, but instead model the language as a whole (approach enabled by Lua’s relatively small size). As mentioned in §6, we have a very thin desugaring process, staying very close to the surface language. Fourth, and related to the previous one, we tested our mechanization using the language’s own test suite directly within Redex, instead of writing another interpreter, as done for performance reasons in Guha et al (2010); Politz et al (2013). This increases the confidence in our mechanization, at the expenses of spending more testing time. Lastly, we pay special attention on the specification and random testing of the model’s properties.

By not following the core language approach we avoided known complexities in the resulting model (Bodin et al, 2014; Maffeis et al, 2008): verbose desugared code and a reduced confidence on the compliance of the given semantics with respect to the original language’s specification. 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. This goal is shared with WebAssembly, whose semantics are formally specified in its core documentation (Rossberg A, 2021).

Besides Guha et al (2010), one major step in formal semantics for JavaScript is JSCert (Bodin et al, 2014): 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, it is our hope to include Lua developers. This may be difficult to achieve over a Coq model because, as the authors from Bodin et al (2014) 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 our model, perhaps using Coq, we want to take advantage of the ease of use of Redex.

Besides the aforementioned Guha et al (2010); Politz et al (2012), Maffeis et al (2008) 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 for Lua 5.2: Lin (2015) presents an operational semantics for Lua, in the style of Featherweight Java (Igarashi et al, 2001), *i.e.*, with a strong focus on recognizing a core language. It considers a subset of the features from the ones presented here and provides a reference implementation in Haskell. Lua 5.1’s dynamic semantics has been studied in Mascarenhas de Queiroz (2009), together with type inference, to obtain an optimized compiler.

8 Final thoughts and future work

This project aims at understanding Lua to the point of being able to construct certified tools, being Soldevila et al (2020) our first step in this direction. Currently, we are at the point where it is possible to analyze realistic programs, perhaps being coroutines (Moura and Ierusalimschy, 2009) the most salient missing feature.

With respect to the mechanization, Redex proved to be a great tool for the semanticist: it is flexible and it is equipped with a well-suited collection of tools. However, as is expected with a dynamic language as Redex, this flexibility is at times a double-edge sword, as it gives path to several simple bugs to crawl in, bugs that are easily avoided with a type system. This said, we ended up eliminating a large amount of bugs by random-testing the semantics.

The interpreter resulting from the mechanization is significantly slow, but it is of no surprise taking into account that the whole mechanization takes about 8k LOCs (without counting comments and tests). Some researchers (Guha et al, 2010; Politz et al, 2013) decided to construct another interpreter by hand, testing this interpreter instead of the mechanization in Redex. In our case, we were still able to execute files with hundreds of lines of code within Redex, by splitting the files and/or adding specific calls to the garbage collector.

An interesting possibility to explore is to port and formally prove the properties in a proof assistant. Then, one could extract a verified interpreter based on abstract machines, that should, in principle, run faster than executing the model within Redex, following the computation model of reduction semantics. Also, this automated step should help in building trust about the correspondence of the obtained interpreter and the original formal semantics.

References

- Adobe (2019) Adobe Lightroom®. <https://www.adobe.io/apis/creativecloud/lightroom.html>, accessed: 2020-05-04
- Bodin M, Chargueraud A, Filaretti D, Gardner P, Maffei S, Naudziuniene D, Schmitt A, Smith G (2014) A trusted mechanised JavaScript specification. In: POPL '14
- Casey Klein SJ Jay McCarthy, Findler RB (2011) A semantics for context-sensitive reduction semantics. In: APLAS'11
- Felleisen M (1987) The calculi of lambda-v-cs conversion: a syntactic theory of control and state in imperative higher-order programming languages. PhD thesis, Indiana University
- Felleisen M, Findler RB, Flatt M (2009) Semantics Engineering with PLT Redex. The MIT Press
- Graham-Cumming J (2013) CloudFlare's new WAF: compiling to Lua. <https://blog.cloudflare.com/cloudflares-new-waf-compiling-to-lua>, accessed: 2020-05-04
- Guha A, Saftoiu C, Krishnamurthi S (2010) The essence of JavaScript. In: ECOOP '10
- Ierusalimschy R (2003) Programming in Lua. Lua.org
- Ierusalimschy R, de Figueiredo LH, Celes W (1996) Lua – an extensible extension language. *Software: Practice and Experience* 26(6):635–652

- Ierusalimsky R, de Figueiredo LH, Celes W (2001a) The evolution of an extension language: a history of lua. In: Brazilian Symposium on Programming Languages
- Ierusalimsky R, de Figueiredo LH, Celes W (2001b) The evolution of an extension language: a history of Lua. In: Brazilian Symposium on Programming Languages
- Ierusalimsky R, de Figueiredo LH, Celes W (2013) Lua 5.2 Reference Manual. Available at www.lua.org/manual/5.2/manual.html
- Igarashi A, Pierce BC, Wadler P (2001) Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS* 23:396–450
- Klein C (2009) Randomized testing in PLT Redex. In: Proc. Scheme and Functional Programming, pp 26–36
- Klein C, Clements J, Dimoulas C, Eastlund C, Felleisen M, Flatt M, McCarthy JA, Raskind J, Tobin-Hochstadt S, Findler RB (2012) Run your research: On the effectiveness of lightweight mechanization. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, POPL '12, pp 285–296, DOI 10.1145/2103656.2103691, URL <http://doi.acm.org/10.1145/2103656.2103691>
- Lin H (2015) Operational semantics for Featherweight Lua. Master's thesis, San José State University
- Lua Dev Team (2013) Lua 5.2 test suite. <https://www.lua.org/tests/>, accessed: 2020-05-04
- Lua Dev Team (2015) Lua 5.2 reference manual. <https://www.lua.org/manual/5.2/manual.html>, accessed: 2020-05-04
- Lua Developers (2009) Expression as statements. <http://lua-users.org/wiki/ExpressionsAsStatements>, accessed: 2020-05-04
- Lua Developers (2014) Lua analyzers. <http://lua-users.org/wiki/ProgramAnalysis>, accessed: 2020-05-04
- Lua Developers (2017) Uses. <https://www.lua.org/uses.html>, accessed: 2020-05-04
- Lua Developers (2018) Lua implementations. <http://lua-users.org/wiki/LuaImplementations>, accessed: 2020-05-04
- Lua Developers (2020) Lua directory. <http://lua-users.org/wiki/LuaDirectory>, accessed: 2020-05-04
- LuaTeX (2018) Lualatex. <http://www.lualatex.org/languages.html>, accessed: 2020-05-04
- Maffeis S, Mitchell JC, Taly A (2008) An operational semantics for JavaScript. In: APLAS '08
- Manura D (2007) Vararg the second class citizen. <http://lua-users.org/wiki/VarargTheSecondClassCitizen>, accessed: 2020-05-04
- Moura AL, Ierusalimsky R (2009) Revisiting coroutines. *TOPLAS* 31(2):6:1–6:31, DOI 10.1145/1462166.1462167, URL <http://doi.acm.org/10.1145/1462166.1462167>
- Pierce BC (2002) Types and Programming Languages, 1st edn. The MIT Press
- Politz JG, Carroll MJ, Lerner BS, Pombrio J, Krishnamurthi S (2012) A tested semantics for getters, setters, and eval in JavaScript. In: DLS '12
- Politz JG, Martinez A, Milano M, Warren S, Patterson D, Li J, Chitipothu A, Krishnamurthi S (2013) Python: The full monty: A tested semantics for the Python programming language. In: OOPSLA '13
- Mascarenhas de Queiroz F (2009) Optimized compilation of a dynamic language to a managed runtime environment. PhD thesis, Pontifícia Universidade Católica

do Rio de Janeiro

Rossberg A editor (2021) Webassembly specification. <https://webassembly.github.io/spec/core/>, accessed: 2021-02-20

Soldevila M, Ziliani B, Silvestre B, Fridlender D, Mascarenhas F (2017) Decoding Lua: Formal semantics for the developer and the semanticist. In: Proceedings of the 13th ACM SIGPLAN Dynamic Languages Symposium, DLS 2017

Soldevila M, Ziliani B, Fridlender D (2020) Understanding Lua’s garbage collection: Towards a formalized static analyzer. In: Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming, PPDP 2020