Noname manuscript No.
(will be inserted by the editor)

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 a 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. Finally, we briefly discuss this avenue of development of our semantics, together with the implementation of a prototype tool to perform static analysis of Lua programs.

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

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

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. We also thank wholeheartedly the anonymous reviewers, whose attention to detail helped us deliver a significantly improved paper.

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. Briefly, some Lua functions can receive a list of arbitrary length of actual parameters. Such functions are known as *vararg functions*. The **print** service is an example of a vararg function. Internally, the list of zero or more actual parameters are manipulated through what we will call a *tuple*. Also, a function can return a list of zero or more values, which will be also modeled as tuples. Over a tuple there are a set of rules that apply, in order to extract their values and use them in the context where the tuple appears: if it appears into a list of values (for example, the list of actual parameters in a function call, as in lines 5-6 in the code shown), we need to append the tuple's values to the list; if it appears where a single value is required, we need to *truncate* the tuple and extract just its first value. As in the example shown, said rules may shield unexpected results, mostly when we deal with *empty* tuples.

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, like its particular take on tuples, 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). As a side note, Lua 5.2 includes interfaces with the garbage collector, namely *weak tables* and *finalizers*, which have an observable impact on the behavior of user programs. In order to account for this, we include on top of our semantics a simple syntactic, non-deterministic, garbage collector to which it is possible to communicate through said interfaces, and we show how the impact on user programs' semantics manifests in that setting.

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 corresponds to that of the selected subset of the language's features, including:

- Every type of Lua value, except *coroutines* and *userdata* (see below);
- Tuples;
- 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)).

For reasons of simplicity, 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.

We develop further the main reasons behind the exclusion of some of these features in §9.

Contributions: As a summary, we present:

1. A formalization of a large portion of Lua, in its version 5.2, including several semantics relevant details of the official interpreter not covered in the reference manual.
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 formalization and a randomly-tested mechanization of the soundness property of the semantics.
5. A brief discussion of applications of the semantics model presented.

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*.

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)
```

§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 introduces two related applications of this work, modeling garbage collection and developing static analysis tools.
§8 discusses related work.
§9 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 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. Note that the memoization only works for values fn(x) different than **nil**. 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 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

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
```

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*" (Ierusalimschy 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 example[2] is listed in Example 4, and it models classes and objects by combining tables, first-class functions—which we already saw in 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

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

Example 5 (Taking the maximum element of an array, together with is 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
```

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 (Ierusalimschy, 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

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
```

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
```

indexes, beginning at 1. Accordingly, the **for-in** construct knows how to use the 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 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 Lua_0, 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 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.

Some comments on notation: in the following, we will use *nt* to refer to, either, the non-terminal nt or the set of terms generated from the production of said non-terminal (resorting to context for disambiguation); an `nt`, possibly with a numeric sub-index, to refer to a term that belongs to the set of terms generated by *nt*.

Grammar

$s ::= \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{;}$ $\qquad$ $binop ::= \textbf{and} \mid \textbf{or}$

$e ::= v \mid e\ binop\ e \mid unop\ e$ $\qquad$ $unop ::= \textbf{not}$

$v ::= \textbf{nil} \mid bool\_literal$

Semantics

$$\frac{\mathsf{v} \notin \{\textbf{nil}, \textbf{false}\}}{\textbf{if}\ \mathsf{v}\ \textbf{then}\ \mathsf{s}_1\ \textbf{else}\ \mathsf{s}_2\ \rightarrow^{s/e}\ \mathsf{s}_1}\ (\textsc{If-T}) \qquad \frac{\mathsf{v} \in \{\textbf{nil}, \textbf{false}\}}{\textbf{if}\ \mathsf{v}\ \textbf{then}\ \mathsf{s}_1\ \textbf{else}\ \mathsf{s}_2\ \rightarrow^{s/e}\ \mathsf{s}_2}\ (\textsc{If-F})$$

$$\textbf{not}\ \mathsf{v} \rightarrow^{s/e}\ \delta(\textbf{not}, \mathsf{v})\ (\textsc{Not}) \qquad \mathsf{v}\ \mathsf{binop}\ \mathsf{e}\ \rightarrow^{s/e}\ \delta(\mathsf{binop}, \mathsf{v}, \mathsf{e})\ (\textsc{Binop})$$

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

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

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

Fig. 1: Pure fragment of Lua_0.

3.1 The pure fragment of Lua_0

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 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-

Grammar

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

Run-time terms

$e ::= \ldots \mid r$

Semantics

$$\frac{\sigma' = (\mathsf{r}, \mathsf{v}), \sigma}{\sigma : \textbf{local}\ \mathsf{x} = \mathsf{v}\ \textbf{in}\ \mathsf{s}\ \ \rightarrow^{\sigma}\ \ \sigma' : \mathsf{s}[\mathsf{x}\backslash\mathsf{r}]}\ (\textsc{Local-Decl})$$

$$\frac{\sigma' = \sigma[\mathsf{r} := \mathsf{v}]}{\sigma : \mathsf{r} \textbf{=} \mathsf{v}\ \rightarrow^{\sigma}\ \sigma' : \textbf{;}}\ (\textsc{Local-Assgn}) \qquad \sigma : \mathsf{r} \rightarrow^{\sigma}\ \sigma : \sigma(\mathsf{r})\ (\textsc{Local-Deref})$$

Fig. 2: Stateful fragment of Lua_0.

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 $\mathsf{dom}(\sigma)$, we do not force any specific representation and just require it to be a finite set, with elements that must be

Evaluation contexts

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

Standard relation

$$\frac{\mathsf{t} \rightarrow^{\mathsf{s/e}} \mathsf{t}'}{\sigma : \mathsf{E}[\![\mathsf{t}]\!] \mapsto \sigma : \mathsf{E}[\![\mathsf{t}']\!]} \text{ (FWD-Pure)} \qquad \frac{\sigma : \mathsf{t} \rightarrow^{\sigma} \sigma' : \mathsf{t}'}{\sigma : \mathsf{E}[\![\mathsf{t}]\!] \mapsto \sigma' : \mathsf{E}[\![\mathsf{t}']\!]} \text{ (FWD-}\sigma\text{)}$$

Fig. 3: Semantics of Lua_0 programs.

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 $(\mathsf{r}, \mathsf{v}), \sigma$ 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 $\rightarrow^{\sigma}$, 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 $\mathsf{s}[\mathsf{x} \backslash \mathsf{r}]$.

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 $\sigma[\mathsf{r} := \mathsf{v}]$ to mean a new store in which the value of r is replaced with v. This is only defined if $\mathsf{r} \in \mathsf{dom}(\sigma)$, which is an implicit side-condition of Local-Assgn, that must hold in order to be able to apply it. Formally, $\sigma[\mathsf{r} := \mathsf{v}]$ denotes a new store σ', such that $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma)$, where $\sigma'(\mathsf{r}) = \mathsf{v}$ and $\forall \mathsf{r}' \in \mathsf{dom}(\sigma'), \mathsf{r}' \neq \mathsf{r} \Rightarrow \sigma'(\mathsf{r}') = \sigma(\mathsf{r}')$. 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 $\rightarrow^{\sigma}$ 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

σ		t	E
$\emptyset$	:	**local** b = **false** **in** **if** **not** b **then** b = **true** **else** b = **false**	⟦ ⟧
↦ (r1, **false**)	:	**if** **not** r1 **then** r1 = **true** **else** r1 = **false**	**if** **not** ⟦ ⟧ **then** ... **else** ...
↦ (r1, **false**)	:	**if** **not** **false** **then** r1 = **true** **else** r1 = **false**	**if** ⟦ ⟧ **then** ... **else** ...
↦ (r1, **false**)	:	**if** **true** **then** r1 = **true** **else** r1 = **false**	⟦ ⟧
↦ (r1, **false**)	:	r1 = **true**	⟦ ⟧
↦ (r1, **true**)	:	;	

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

the corresponding relation. We are aiming at a deterministic semantics, so there should be only one redex in every 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 ↦ 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.

[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 (`luajit.org`).

Grammar

s ::= ... | s s | **while** e **do** s | **break**

Run-time terms

s ::= ... | **$iter** e **do** s | ⦇ s ⦈$_{label}$

$label$ ::= Break

Evaluation contexts

E_{lf} ::= [ ] | **if** E_{lf} **then** s **else** s
| **local** x **=** E_{lf} **in** s | x **=** E_{lf}
| E_{lf} *binop* e | *unop* E_{lf} | E_{lf} s

E ::= extend E_{lf} with ⦇ E ⦈$_{\text{Break}}$

Semantics

; s $\rightarrow^{s/e}$ s (Seq)

while e **do** s $\rightarrow^{s/e}$ ⦇ **$iter** e **do** s ⦈$_{\text{Break}}$ (While-Start)

$iter e **do** s $\rightarrow^{s/e}$ **if** e **then** s **;** **$iter** e **do** s **else ;** (While-Iter)

⦇ E_{lf} ⟦ **break** ⟧ ⦈$_{\text{Break}}$ $\rightarrow^{s/e}$ **;** (While-Break)

⦇ **;** ⦈$_{\text{Break}}$ $\rightarrow^{s/e}$ **;** (While-End)

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

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 **$iter**. The labeling and **$iter**

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 **$iter** is necessary to avoid repeatedly unfolding a while, stacking up labels.

A loop marked with **$iter** is then unfolded as usual, using an **if-then-else** to check the guard and perform a new iteration (While-Iter). We took from Guha et al. (2010) the idea of labeling the while loop to mark the exit point of a **break**; however, our label is inserted at run-time, while in the aforementioned work it is inserted when *desugaring* the code. Our take reduces the complexity of the desugared code, but at the expense of requiring the extra run-time construct **$iter** 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_{lf} (*label-free* contexts), 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_{lf} 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_{lf}. Extending an evaluation context like E_{lf} to create a new evaluation context E has the effect of renaming each occurrence of E_{lf} with E.

When the execution of the loop ends normally (While-End), we just discard the label.

The presented features cover the main aspects of the semantics and control-flow of the **while** statement. Note that Lua 5.2 does not have a **continue** statement, yet some of the proposed ways of emulating it[4] involve concepts already present into our model: using conditionals, Lua's error mechanism (to be introduced in §4.5), or using a **break**.

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. Note how we extend the evaluation context E_{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 $\delta(\mathsf{op})$. 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

[4] See `lua-users.org/wiki/ContinueProposal`

Grammar

$v ::= ... \mid \mathit{number\_literal} \mid \mathit{string\_literal}$

$e ::= ... \mid e\ \mathit{strictbinop}\ e$

$\mathit{strictbinop} ::= \mathsf{+} \mid \mathsf{-} \mid \mathsf{*} \mid \mathsf{/} \mid \mathsf{\hat{}} \mid \mathsf{\%} \mid \mathsf{..} \mid < \mid \leq \mid > \mid \geq \mid \mathsf{==}$

$\mathit{unop} ::= ... \mid \mathsf{-} \mid \mathsf{\#}$

Run-time terms

$e ::= ... \mid (\!|\ e\ |\!)_{\mathit{label}}$

$\mathit{label} ::= ... \mid \textsc{BinopWO} \mid \textsc{EqFail}$

Evaluation contexts

$E_{\mathit{lf}} ::= \ldots \mid E_{\mathit{lf}}\ \mathit{strictbinop}\ e \mid v\ \mathit{strictbinop}\ E_{\mathit{lf}}$

Semantics of binary operators

$$\frac{\mathsf{op} \in \{\mathsf{+},\mathsf{-},\mathsf{*},\mathsf{/},\mathsf{\hat{}},\mathsf{\%},\mathsf{..},<,<=\} \qquad \delta(\mathsf{op},\mathsf{v}_1,\mathsf{v}_2) \in v}{\mathsf{v}_1\ \mathsf{op}\ \mathsf{v}_2 \rightarrow^{s/e} \delta(\mathsf{op},\mathsf{v}_1,\mathsf{v}_2)}\ (\textsc{Binop})$$

$$\frac{\mathsf{op} \in \{\mathsf{+},\mathsf{-},\mathsf{*},\mathsf{/},\mathsf{\hat{}},\mathsf{\%},\mathsf{..},<,<=\} \qquad \delta(\mathsf{op},\mathsf{v}_1,\mathsf{v}_2) \notin v}{\mathsf{v}_1\ \mathsf{op}\ \mathsf{v}_2 \rightarrow^{s/e} (\!|\ \mathsf{v}_1\ \mathsf{op}\ \mathsf{v}_2\ |\!)_{\textsc{BinopWO}}}\ (\textsc{Binop-WO})$$

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

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 ($\delta(\mathsf{op},\mathsf{v}_1,\mathsf{v}_2) \notin v$). 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).

In case of equality, the behavior is a bit more intricate (Figure 4c). As with previous operators, we abstract into $\delta(\mathsf{==})$ the corresponding details of equality comparison, but we perform some extra processing of the result. However, note that the equality relation is not trivial: *e.g.*, , in order to obtain a semantics that can be tested against the official test suites, it must define proper comparison of IEEE 754 floating point numbers (as in a standard binary of the official interpreter).

$$\frac{\delta(\texttt{==}, \mathsf{v}_1, \mathsf{v}_2) = \textbf{true}}{\mathsf{v}_1 \texttt{ == } \mathsf{v}_2 \;\rightarrow^{s/e}\; \textbf{true}} \;(\textsc{Eq-True})$$

$$\frac{\delta(\texttt{==}, \mathsf{v}_1, \mathsf{v}_2) = \textbf{false} \qquad \delta(\mathsf{type}, \mathsf{v}_1) \neq \texttt{"table"} \vee \delta(\mathsf{type}, \mathsf{v}_2) \neq \texttt{"table"}}{\mathsf{v}_1 \texttt{ == } \mathsf{v}_2 \;\rightarrow^{s/e}\; \textbf{false}} \;(\textsc{Eq-False})$$

$$\frac{\delta(\texttt{==}, \mathsf{v}_1, \mathsf{v}_2) = \textbf{false} \qquad \delta(\mathsf{type}, \mathsf{v}_1) = \texttt{"table"} \qquad \delta(\mathsf{type}, \mathsf{v}_2) = \texttt{"table"}}{\mathsf{v}_1 \texttt{ == } \mathsf{v}_2 \;\rightarrow^{s/e}\; (\!|\, \mathsf{v}_1 \texttt{ == } \mathsf{v}_2 \,|\!)_{\textsc{EqFail}}} \;(\textsc{Eq-Fail})$$

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

In essence, $\delta(\texttt{==})$ 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

The syntax of table constructors, table field indexing and assignment, is presented in Figure 5a, together with their associated semantic rules. 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 (and associated meta-data), 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 the former should be syntactically distinguishable from the latter. The image of θ only contains tables and any table-related meta-data. For the present model, this meta-data reduces to information about the associated metatable, introduced later. Formally,

Syntax

$e ::= \ldots \mid \{\, \mathit{field}, \ldots \}$
$\quad \mid x \; \mathbf{[}\; e \;\mathbf{]}$
$\mathit{field} ::= e \mid \mathbf{[}\; e \;\mathbf{]} = e$
$\mathit{efield} ::= v \mid \mathbf{[}\; v \;\mathbf{]} = v$

Evaluation context

$E_{lf} ::= \ldots \mid E_{lf} \; \mathbf{[}\; e \;\mathbf{]} \mid v \; \mathbf{[}\; E_{lf} \;\mathbf{]}$
$\quad \mid \mathbf{\{}\; \mathit{efield} \;\mathbf{,}\; \ldots \;\mathbf{,}\; \mathbf{[}\; E_{lf} \;\mathbf{]} = e \;\mathbf{,}\; \mathit{field} \;\mathbf{,}\; \ldots \;\mathbf{\}}$
$\quad \mid \mathbf{\{}\; \mathit{efield} \;\mathbf{,}\; \ldots \;\mathbf{,}\; \mathbf{[}\; v \;\mathbf{]} = E_{lf} \;\mathbf{,}\; \mathit{field} \;\mathbf{,}\; \ldots \;\mathbf{\}}$
$\quad \mid \mathbf{\{}\; \mathit{efield} \;\mathbf{,}\; \ldots \;\mathbf{,}\; E_{lf} \;\mathbf{,}\; \mathit{field} \;\mathbf{,}\; \ldots \;\mathbf{\}}$

Run-time terms

$v ::= \ldots \mid \mathit{tid}$
$\mathit{label} ::= \ldots \mid \textsc{WrongKey} \mid \textsc{NonTable}$

Function for adding numeric keys (excerpt)

$\mathsf{addkeys}(\{\mathsf{v_1}, \ldots, \mathsf{v_n}, [\mathsf{k_1}] = \mathsf{v_{n+1}}, \ldots\}) = \{[1] = \mathsf{v_1}, \ldots, [\mathsf{n}] = \mathsf{v_n}, [\mathsf{k_{1'}}] = \mathsf{v_{n+1'}}, \ldots\}$
where $\mathsf{k_{1'}}, \ldots \in \{\mathsf{k_1}, \ldots\} \setminus [1, \mathsf{n}]$

Semantics

$$\frac{\theta_2 = (\mathsf{tid},\ (\mathsf{addkeys}(\{\mathsf{efield_1},\ ...\})\ ,\ \mathbf{nil}\ )),\ \theta_1}{\theta_1 : \{\mathsf{efield_1},\ ...\}\ \rightarrow^{\theta}\ \theta_2 : \mathsf{tid}}\ (\textsc{Table-Constr})$$

$$\frac{\delta(\mathsf{rawget}, \mathsf{tid}, \mathsf{v_1}, \theta_1) \neq \mathbf{nil} \qquad (\theta_2, \mathsf{tid}) = \delta(\mathsf{rawset}, \mathsf{tid}, \mathsf{v_1}, \mathsf{v_2}, \theta_1)}{\theta_1 : \mathsf{tid}\ \mathbf{[}\mathsf{v_1}\mathbf{]} = \mathsf{v_2}\ \rightarrow^{\theta}\ \theta_2 : ;}\ (\textsc{Table-Update})$$

$$\frac{\delta(\mathsf{rawget}, \mathsf{tid}, \mathsf{v_1}, \theta) = \mathbf{nil}}{\theta : \mathsf{tid}\ \mathbf{[}\mathsf{v_1}\mathbf{]} = \mathsf{v_2}\ \rightarrow^{\theta}\ \theta : (\!|\ \mathsf{tid}\ \mathbf{[}\mathsf{v_1}\mathbf{]} = \mathsf{v_2}\ |\!)_{\textsc{WrongKey}}}\ (\textsc{Table-Update-WK})$$

$$\frac{\delta(\mathsf{type}, \mathsf{v_1}) \neq \text{"table"}}{\theta : \mathsf{v_1}\ \mathbf{[}\mathsf{v_2}\mathbf{]} = \mathsf{v_3}\ \rightarrow^{\theta}\ \theta : (\!|\ \mathsf{v_1}\ \mathbf{[}\mathsf{v_2}\mathbf{]} = \mathsf{v_3}\ |\!)_{\textsc{NonTable}}}\ (\textsc{Table-Update-NT})$$

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

$\theta : \mathit{tid} \to \mathit{tables} \times \{\mathbf{nil}, \mathit{tid}\}$, where *tables* denotes the set of tables as described by the grammar in Figure 5a, and **nil** or *tid* are the possible metatable-related values associated with the given table, to be introduced later. We capture a new store θ resulting from, either, extending a previous store 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 $\mathbf{[}v\mathbf{]} = v$). The meta-function addkeys adds absent keys—namely, consecutive natural numbers—to fields of the form v. It also discards fields already present in the constructor that contain numeric keys in the interval $[1, \mathsf{n}]$. Here, n is the quantity of fields without keys in the constructor (hence, the fields

$$\delta(\mathsf{rawget}, \mathsf{v}_1, \mathsf{v}_2, \theta) = \delta(\mathsf{error}, ...) \quad \textit{if } \delta(\mathsf{type}, \mathsf{v}_1) \neq \text{"table"}$$

$$\delta(\mathsf{rawget}, \mathsf{tid}, \mathsf{v}_i, \theta) = \begin{cases} \mathsf{v}_j & \textit{if } \theta(\mathsf{tid}) = (\{..., \ [\mathsf{v}'_i]{=}\mathsf{v}_j, \ ...\}, \ ...) \\ & \textit{and } \delta(\mathbf{==}, \mathsf{v}_i, \mathsf{v}'_i) = \mathbf{true} \\ \mathbf{nil} & \textit{otherwise} \end{cases}$$

$$\delta(\mathsf{rawset}, \mathsf{tid}, \mathsf{v}_1, \mathsf{v}_2, \theta_1) = (\theta_2, \mathsf{tid}),$$
$$\textit{where} \begin{cases} \mathsf{v}_1 \notin \{\mathbf{nil}, \mathbf{nan}\} \\ \theta_1(\mathsf{tid}) = (\{..., \mathsf{efield}_{i-1}, [\mathsf{v}'_1]{=}\mathsf{v}_3, \mathsf{efield}_{i+1}, ...\}, ...) \\ \delta(\mathbf{==}, \mathsf{v}_1, \mathsf{v}'_1) = \mathbf{true} \\ \theta_2 = \begin{cases} \theta_1[\mathsf{tid} := (\{..., \mathsf{efield}_{i-1}, \mathsf{efield}_{i+1}, ...\}, ...)] & \textit{if } \mathsf{v}_2 = \mathbf{nil} \\ \theta_1[\mathsf{tid} := (\{..., \mathsf{efield}_{i-1}, [\mathsf{v}'_1]{=}\mathsf{v}_2, \mathsf{efield}_{i+1}, ...\}, ...)] & \textit{if } \mathsf{v}_2 \neq \mathbf{nil} \end{cases} \end{cases}$$

$$\delta(\mathsf{rawset}, \mathsf{tid}, \mathsf{v}, \mathsf{v}', \theta_1) = (\theta_2, \mathsf{tid}),$$
$$\textit{where} \begin{cases} \mathsf{v} \notin \{\mathbf{nil}, \mathbf{nan}\} \\ \theta_1(\mathsf{tid}) = (\{[\mathsf{v}_1]{=}\mathsf{v}'_1, ..., [\mathsf{v}_n]{=}\mathsf{v}'_n\}, ...) \\ \forall \ \mathsf{k} \ \in \ \{\mathsf{v}_1, ..., \mathsf{v}_n\}, \delta(\mathbf{==}, \mathsf{v}, \mathsf{k}) = \mathbf{false} \\ \theta_2 = \begin{cases} \theta_1 & \textit{if } \mathsf{v}' = \mathbf{nil} \\ \theta_1[\mathsf{tid} := (\{[\mathsf{v}]{=}\mathsf{v}', [\mathsf{v}_1]{=}\mathsf{v}'_1, ..., [\mathsf{v}_n]{=}\mathsf{v}'_n\}, ...)] & \textit{if } \mathsf{v}' \neq \mathbf{nil} \end{cases} \end{cases}$$

$$\delta(\mathsf{rawset}, \mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \theta) = (\theta, \delta(\mathsf{error}, ...)) \ , \ \textit{if } \delta(\mathsf{type}, \mathsf{v}_1) \neq \text{"table"} \vee \mathsf{v}_2 \in \{\mathbf{nil}, \mathbf{nan}\}$$

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

for which addkeys adds the corresponding numeric keys). We show an excerpt of its formal definition, in Figure 5a. Note that Lua's reference manual is not precise in this point, and different definitions of addkeys can have an observable impact on the semantics of operations like computing the *length* of a table (through the operator #), or iterating a table, through library services like ipairs. Our mechanization of addkeys successfully passes the official interpreter's tests, and as such, can serve as a frame for future implementations.

Finally, returning to Table-Constr (Figure 5a), 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.

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. $\delta(\mathsf{rawget}, \mathsf{tid}, \mathsf{v}, \theta)$ yields either the value associated with v in $\theta(\mathsf{tid})$ or **nil**, if there is no associated value. Note that we use the equality comparison function ($\delta(\mathbf{==})$) to determine which field is

Grammar

$$
\begin{array}{rcl}
s & ::= & \ldots \mid var\ \texttt{,}\ \ldots \texttt{=}\ e\ \texttt{,}\ \ldots \\
 & & \mid \textbf{local}\ x \texttt{,} \ldots \texttt{=}\ e \texttt{,}\ \ldots\ \textbf{in}\ s \\
var & ::= & x \mid e\ \texttt{[}\ e\ \texttt{]} \\
evar & ::= & r \mid v\ \texttt{[}\ v\ \texttt{]}
\end{array}
$$

Evaluation context

$$
\begin{array}{rcl}
E_{lf} & ::= & \ldots \\
 & & \mid \textbf{local}\ x\ \texttt{,}\ \ldots\ \texttt{=}\ v\ \texttt{,}\ \ldots\ \texttt{,}\ E_{lf}\ \texttt{,}\ e\ \texttt{,}\ \ldots\ \textbf{in}\ s \\
 & & \mid evar\ \texttt{,}\ \ldots\ \texttt{,}\ E_{lf}\ \texttt{[}\ e\ \texttt{]}\ \texttt{,}\ \ldots\ \texttt{=}\ e\ \texttt{,}\ \ldots \\
 & & \mid evar\ \texttt{,}\ \ldots\ \texttt{,}\ v\ \texttt{[}\ E_{lf}\ \texttt{]}\ \texttt{,}\ \ldots\ \texttt{=}\ e\ \texttt{,}\ \ldots \\
 & & \mid evar\ \texttt{,}\ \ldots\ \texttt{=}\ v\ \texttt{,}\ \ldots\ \texttt{,}\ E_{lf}\ \texttt{,}\ \ldots
\end{array}
$$

Semantics

$$\frac{\mathsf{k} \geq 1 \qquad \mathsf{v_{n-k+1}} = \textbf{nil}, ..., \mathsf{v_n} = \textbf{nil}}{\begin{array}{l}\mathsf{evar_1,...,evar_n = v_1,...,v_{n-k}} \ \rightarrow^{s/e} \\ \quad \mathsf{evar_1,...,evar_n = v_1,...,v_{n-k},v_{n-k+1},...,v_n}\end{array}}\ (\textsc{Assgn-Fewer})$$

$$\frac{\mathsf{k} \geq 1}{\mathsf{evar_1,...,evar_n = v_1,...,v_n,...,v_{n+k}} \ \rightarrow^{s/e} \ \mathsf{evar_1,...,evar_n = v_1,...,v_n}}\ (\textsc{Assgn-More})$$

$$\frac{\mathsf{n} \geq 2}{\begin{array}{l}\mathsf{evar_1,...,evar_n = v_1,...,v_n} \ \rightarrow^{s/e} \\ \quad \mathsf{evar_1 = evar_1 \quad evar_2,...,evar_n = v_2,...,v_n}\end{array}}\ (\textsc{Assgn-Split})$$

$$\frac{\sigma' = (\mathsf{r_1}, \mathsf{v_1}), ..., (\mathsf{r_n}, \mathsf{v_n}), \sigma}{\sigma : \textbf{local}\ \mathsf{x_1,...,x_n = v_1,...,v_n}\ \textbf{in}\ \mathsf{s} \ \rightarrow^{\sigma} \ \sigma' : \mathsf{s}[\mathsf{x_1} \backslash \mathsf{r_1}, ..., \mathsf{x_n} \backslash \mathsf{r_n}]}\ (\textsc{Local-Decl})$$

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

being indexed with the given key. Recall from §4.1.2 that $\delta(\texttt{==})$ should define the details of a proper comparison between Lua values, and, also, should not resort to the meta-tables mechanism when comparing different table values (which is not the case with the language == comparison operator).

$\delta(\mathsf{rawset}, \mathsf{tid}, \mathsf{v_1}, \mathsf{v_2}, \theta)$ yields a new θ where the table referenced by tid associates $\mathsf{v_2}$ with value $\mathsf{v_1}$. In the first equation we consider an existing key, in which case we remove or update the field, depending of the value $\mathsf{v_2}$. Note how **nil** and **nan** cannot be the keys of a table field. As with rawget, the keys are compared using $\delta(\texttt{==})$, and the same considerations apply with regard to the semantics consequences when using user-defined equality comparison relations. 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.

Grammar

$v ::= \ldots \mid \textbf{function}\ \ell\ \textbf{(}\ x\ \textbf{,}\ \ldots\ \textbf{)}\ s \mid \textbf{function}\ \ell\ \textbf{(}\ x\ \textbf{,}\ \ldots\ \textbf{,}\ \textbf{...}\ \textbf{)}\ s$

$s ::= \ldots \mid \textbf{\$statFcall}\ e\ \textbf{(}e\ \textbf{,}\ \ldots\ \textbf{)} \mid \textbf{\$statFcall}\ e\ \textbf{:}\ x\ \textbf{(}e\ \textbf{,}\ \ldots\ \textbf{)}$
$\quad \mid \textbf{return}\ e\textbf{,}\ \ldots$

$e ::= \ldots \mid e\textbf{(}\ e\ \textbf{,}\ \ldots\ \textbf{)} \mid e\textbf{:}\ x\ \textbf{(}\ e\ \textbf{,}\ \ldots\ \textbf{)} \mid \textbf{...} \mid \textbf{(}\ e\textbf{)}$

Run-time terms

$e ::= \ldots \mid < e\ \textbf{,}\ \ldots > \mid (\!|\ s\ |\!)_{label}$
$label ::= \ldots \mid \textsc{RetStat} \mid \textsc{RetExp} \mid \textsc{WFunCall}$

Evaluation context

$E_{\mathsf{el}} ::= \quad v\ \textbf{,}\ \ldots\ \textbf{,}\ E_{\mathit{lf}}\ \textbf{,}\ e\ \textbf{,}\ \ldots$

$E_{\mathit{lf}} ::= \ldots \mid \textbf{\$statFcall}\ E_{\mathit{lf}}\ \textbf{(}\ e\ \textbf{,}\ \ldots\ \textbf{)} \mid \textbf{\$statFcall}\ v\ \textbf{(}\ E_{\mathsf{el}}\ \textbf{)}$
$\quad \mid \textbf{\$statFcall}\ E_{\mathit{lf}}\ \textbf{:}\ x\ \textbf{(}\ e\ \textbf{,}\ \ldots\ \textbf{)} \mid \textbf{return}\ E_{\mathsf{el}}$
$\quad \mid E_{\mathit{lf}}\ \textbf{(}\ e\ \textbf{,}\ \ldots\ \textbf{)} \mid v\ \textbf{(}\ E_{\mathsf{el}}\ \textbf{)} \mid E_{\mathit{lf}}\ \textbf{:}\ x\ \textbf{(}\ e\ \textbf{,}\ \ldots\ \textbf{)} \mid \textbf{(}\ E_{\mathit{lf}}\textbf{)}$
$\quad \mid (\!|\ E_{\mathit{lf}}\ |\!)_{\textsc{RetExp}} \mid (\!|\ E_{\mathit{lf}}\ |\!)_{\textsc{RetStat}}$

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

Similarly, we can have arbitrary lists in the assignment, with the addition that we can update variables *and* tables' fields.

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. This is so since rvalues can be local variables and/or table fields, and we need to distinguish each case in order to properly explain their semantics. 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). Finally, by splitting a multiple assignment into several single variable assignments we also model what happens in Lua when there is an lvalue that is repeated: the assignments are effectively repeated over the local variable or table field.

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). Repeated occurrences of variable identifiers (as lvalues) are allowed, and their correspondent rvalues are evaluated. Then, when the introduction of the variable in the corresponding scope is applied (through the substitution function), only the last variable definition is taken into account.

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. 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 = **function** ℓ_1 () **end**.

Functions have two constructors; the difference being the *vararg* arguments **...** (we use this special notation to avoid confusion with ellipses also used in the grammar; when showing Lua code, we will use the conventional syntax ...). 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 < e , ... >, and cannot be constructed by the user. They are not values and, hence, cannot be stored.

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* **(***e***)**, 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(); print('hi'), or in an expression, as in 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.[5]

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
```

[5] 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.

$$\frac{1 \leq i \leq \mathsf{min}(m,n) \Rightarrow \mathsf{v'}_i = \mathsf{v}_i \qquad i > m \Rightarrow \mathsf{v'}_i = \textbf{nil} \qquad \sigma' = (\mathsf{r}_1, \mathsf{v'}_1), ..., (\mathsf{r_n}, \mathsf{v'}_n), \sigma}{\sigma : (\textbf{function}\ \ell\ (\mathsf{x}_1, ..., \mathsf{x}_n)\ \mathsf{s})\ (\mathsf{v}_1, ..., \mathsf{v}_m)\ \rightarrow^{\sigma} \qquad \sigma' : (\!|\ \mathsf{s}\ [\mathsf{x}_1 \backslash \mathsf{r}_1, ..., \mathsf{x}_n \backslash \mathsf{r}_n]\ |\!)_{\text{RETEXP}}}\ \text{(E-CALL)}$$

$$\frac{\begin{array}{c} 1 \leq i \leq \mathsf{min}(m,n) \Rightarrow \mathsf{v'}_i = \mathsf{v}_i \\ i > m \Rightarrow \mathsf{v'}_i = \textbf{nil} \qquad \mathsf{tuple} = < \mathsf{v}_{n+1}, ..., \mathsf{v}_m > \qquad \sigma' = (\mathsf{r}_1, \mathsf{v'}_1), ..., (\mathsf{r}_n, \mathsf{v'}_n), \sigma \end{array}}{\begin{array}{c} \sigma : (\textbf{function}\ \ell\ (\mathsf{x}_1, ..., \mathsf{x}_n, \textbf{...})\ \mathsf{s})\ (\mathsf{v}_1, ..., \mathsf{v}_m)\ \rightarrow^{\sigma} \\ \sigma' : (\!|\ \mathsf{s}\ [\mathsf{x}_1 \backslash \mathsf{r}_1, ..., \mathsf{x}_n \backslash \mathsf{r}_n, \textbf{...} \backslash \mathsf{tuple}]\ |\!)_{\text{RETEXP}} \end{array}}\ \text{(E-CALLVARG)}$$

$$\frac{\delta(\mathsf{type}, \mathsf{v}) \neq \text{"function"}}{\mathsf{v}\ (\mathsf{v}_1, ..., \mathsf{v_n})\ \rightarrow^{s/e}\ (\!|\ \mathsf{v}\ (\mathsf{v}_1, ..., \mathsf{v_n})\ |\!)_{\text{WFUNCALL}}}\ \text{(E-CALLWRONG)}$$

$$\mathsf{v{:}name}\ \textbf{(}\mathsf{e}_1\textbf{,...,}\mathsf{e}_n\textbf{)}\ \rightarrow^{s/e}\ \mathsf{v}\textbf{[}\text{"name"}\textbf{]}\ \textbf{(}\mathsf{v}\textbf{,}\mathsf{e}_1\textbf{,...,}\mathsf{e}_n\textbf{)}\ \text{(E-MCALL)}$$

$$(\!|\ E_{\mathit{lf}}[\![\ \textbf{return}\ (\textbf{function}\ \ell\ (\mathsf{x}_1, ...)\ \mathsf{s})\ (\mathsf{v}_1, ...)\ ]\!]\ |\!)_{\text{RETEXP}} \qquad \rightarrow^{s/e} \qquad \begin{array}{c}(\textbf{function}\ \ell\ (\mathsf{x}_1, ...)\ \mathsf{s})\ (\mathsf{v}_1, ...)\end{array}\ \text{(E-POPSF)}$$

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

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.

A last aspect of function calls, that can be captured into the model presented so far, is related with *tail calls*: nested function calls can reuse the stack frame of the actual called function, if the function call is the last instruction.[6] Specifically, in Lua 5.2 tail calls are calls of the form **return** (**function** ℓ (x1, ...) s) (v1, ...). In that case, the called function will reuse the stack frame of the actual function from which it is being called. In our model, we include just one piece of information from a regular stack frame: the position to which the function must return, denoted with the labels RetStat and RetExp. And by means of the manipulation of this representation we can model tail calls. We explain it below.

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. Recognize function calls in tail position and act accordingly.

[6] We thank the reviewers for inviting us to enrich our model with this and other features.

3. Adjust the list of actual parameters and perform the function call itself.
4. According to the place of the call (statement or expression), discard the returned values or put them in a tuple.
5. In the case of an expression, distinguish if the tuple must be appended to other values or trimmed.

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 partially described by in Figure 7b, last rule, only for the case of a nested function call made from another function call embedded into an expression. In order to better understand it, we first explain the next step.

The third 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. Note that they are not stored, as it does occur with the remaining actual parameters. 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): the binding occurrence binds any vararg mark that occurs in the body of the function, except if it appears in the body of a nested function definition, regardless of it being or not a vararg function. The following interaction with the official interpreter (taken from Manura (2007, Issue #1)) illustrates this scoping rule:

```
> function f( ... )
    return function() return ... end
  end

stdin :1: cannot use ' ... ' outside a vararg function near ' ... '
```

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.

Now, previous to the function call itself, we try to recognize if it is in tail position. Remember that, in Lua 5.2, tail calls reduce to calls of the form:

$$\textbf{return}\ (\textbf{function}\ \ell\ \textbf{(}\mathsf{x}_1, ...\textbf{)}\ \mathsf{s})\ (\mathsf{v}_1, ...)$$

This step will be responsibility of the top level relation $\mapsto$: it will be in charge of guaranteeing that a tail call is correctly processed before actually executing the function call itself. Once that a tail call is recognized, we manipulate the stack of calls: in our model, we just look for the innermost RetStat or RetExp label around the function call, and remove it together with the whole rest of computation between the label and the tail call itself. That is, we remove the computations that would be discarded when performing the return of the function being called in tail position. More formally, we look for a function call that looks like this, for the case of a tail call nested into another function call done in the position of an expression:

$$\langle\!|\ E_{lf}[\![\ \textbf{return}\ (\textbf{function}\ \ell\ \textbf{(}\mathsf{x}_1, ...\textbf{)}\ \mathsf{s})\ (\mathsf{v}_1, ...)\ ]\!]\ |\!\rangle_{\textsc{RetExp}}$$

In that case, and in terms closer to an actual implementation, we could say that we *pop* the actual stack frame from the call stack, obtaining:

$$(\textbf{function}\ \ell\ \textbf{(}\mathsf{x}_1, ...\textbf{)}\ \mathsf{s})\ (\mathsf{v}_1, ...)$$

This is performed by rule E-PopSF, in Figure 7b. Then, the actual execution of the function call, as described by the previous semantics rules, will install a new stack frame for the current tail call. Together, all these steps model the phenomenon of a tail call.

A last remark: in the context of a programming language with interfaces with the garbage collector, like finalizers, these implementation details have an observable effect on the semantics of programs. As an example, consider the following program:

```
function f(a)
  local b = {}
  return f(b)
end

local a = {}
f(a)
```

In the previous program, given that the recursive call to `f` is in tail position, at each recursive call the whole content of the stack frame of the previous call is discarded. This implies that, at each recursive call, the reference to the parameter `a` from the previous call is lost (except for the first call, made in the last line). On the other hand, if the recursive call to `f` would be just `f ()`, instead of **return** `f ()`, then the call would not be in tail position, and every stack frame of every call would be preserved, including the reference to the parameter `a`. As we explain in (Soldevila et al., 2020) and §7, Lua 5.2 has a reachability-based garbage collector that, in each recursive tail call of `f`, would consider unreachable the parameter of the previous call. Given that we have interfaces with the garbage collector, this difference in the internal treatment of the stack frames could be observed at the semantics level.

$$\langle\!|\ ;\ |\!\rangle_{\text{RETSTAT}} \to^{s/e}\ ;\ (\text{S-RETSKIP}) \qquad \langle\!|\ E_{lf}[\![\ \textbf{return}\ v, \ldots\ ]\!]\ |\!\rangle_{\text{RETSTAT}} \to^{s/e}\ ;\ (\text{S-RETURN})$$

$$\langle\!|\ ;\ |\!\rangle_{\text{RETEXP}} \to^{s/e} <> (\text{E-RETSKIP})$$

$$\langle\!|\ E_{lf}[\![\ \textbf{return}\ v, \ldots\ ]\!]\ |\!\rangle_{\text{RETEXP}} \to^{s/e} < v, \ldots > (\text{E-RETURN})$$

$$\langle\!|\ E_{lf}[\![\ \textbf{return}\ v, \ldots\ ]\!]\ |\!\rangle_{\text{BREAK}} \to^{s/e} \textbf{return}\ v, \ldots\ (\text{S-RETURNBREAK})$$

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

The fourth 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.

The fifth 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' 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()) --=> 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 $\langle\!|\ \textbf{return nil}\ |\!\rangle_{\text{RETEXP}}$. With an empty E_{lf} context, we perform one E-RETURN step and get a tuple with

Contexts for tuple processing (excerpt)

$E_{\mathsf{tel}} ::= v \ , \ \ldots \ , \ [\![\,]\!] \ , \ e_1 \ , \ e_2 \ \ldots$
$E_{\mathsf{t}} \ ::= \textbf{if} \ [\![\,]\!] \ \textbf{then} \ s \ \textbf{else} \ s \mid \textbf{local} \ x \ , \ \ldots \ = E_{\mathsf{tel}} \ \textbf{in} \ s \mid \mathit{evar} \ , \ \ldots \ = E_{\mathsf{tel}}$
$\quad \mid (\![\,]\!) \mid \{ \ \mathit{efield} \ , \ \ldots \ , \ [\![\,]\!] \ , \ \mathit{field}_1 \ , \ \mathit{field}_2 \ , \ \ldots \ \} \mid \ldots$

$E_{\mathsf{ael}} ::= v \ , \ \ldots \ , \ [\![\,]\!]$
$E_{\mathsf{a}} \ ::= \textbf{local} \ x \ , \ \ldots \ = E_{\mathsf{ael}} \ \textbf{in} \ s \mid \mathit{evar} \ , \ \ldots \ = E_{\mathsf{ael}} \mid \{ \ \mathit{efield} \ , \ \ldots \ , \ [\![\,]\!] \ \} \mid \ldots$

Appending function (excerpt)

$\mathsf{append}(\mathsf{evar}, \ldots = \mathsf{v}_1, \ldots, [\![\,]\!], \ < \mathsf{v}_2, \mathsf{v}_3, \ldots >) = \mathsf{evar}, \ldots = \mathsf{v}_1, \ldots, \mathsf{v}_2, \mathsf{v}_3, \ldots$
$\mathsf{append}(\mathsf{evar}, \ldots = \mathsf{v}_1, \ldots, [\![\,]\!], \ <>) = \mathsf{evar}, \ldots = \mathsf{v}_1, \ldots$

Semantics

$$E_{\mathsf{t}}\,[\![< \mathsf{v}_1, \mathsf{v}_2, \ldots >]\!] \rightarrow^{\mathsf{s/e}} E_{\mathsf{t}}\,[\![\mathsf{v}_1]\!] \ (\textsc{Tuple-One})$$

$$E_{\mathsf{t}}\,[\![< >]\!] \rightarrow^{\mathsf{s/e}} E_{\mathsf{t}}\,[\![\textbf{nil}]\!] \ (\textsc{Tuple-Zero})$$

$$E_{\mathsf{a}}\,[\![< \mathsf{v}_1, \ldots >]\!] \rightarrow^{\mathsf{s/e}} \mathsf{append}(E_{\mathsf{a}}, < \mathsf{v}_1, \ldots >) \ (\textsc{Tuple-Append})$$

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

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 **nil**s.

Different is the case when we do not return anything:

```
function g() end
print(g(), g(), g()) -->> 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* (Ierusalimschy 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:[7]

Example 10 (Caching of functions.)

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

```
1 f = function() end
2 g = function() end
3 print(f == g) −−>> false
4 h = function() return function() end end
5 print(h() == h()) −−>> 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:

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). The environment of a closure, including every local variable whose scope reaches to the given closure, is part of what the official interpreter recognizes as "observable difference" (besides the function definition itself), and, accordingly, differences in a scope is what triggers the creation of new closures, from the same function definition. In our formalization we consider, as a representation of a closure, the whole function definition with its embedded environment, as it is when the focus of reduction reaches to the function definition (see Corollary 2, §5). Therefore, returning to the previous example, we would include in the corresponding representation of the closures created, each new instance from each iteration of the variable i. At each iteration we would end up with a different closure and, hence, no closure would be reused.

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

Grammar

svc ::= **assert** | **error** | ...

e ::= ... | **$builtIn** svc (e,...))

Semantics

$$\frac{\mathsf{svc} \in \{\mathsf{type}, \mathsf{assert}, \mathsf{error}, \mathsf{pcall}, \mathsf{select}, ...\}}{\textbf{\$builtIn}\ \mathsf{svc}\ (\mathsf{v}_1, ..., \mathsf{v}_n) \rightarrow^{s/e} \delta(\mathsf{svc},\ \mathsf{v}_1, ..., \mathsf{v}_n)}\ (\textsc{Svc-NoState})$$

$$\frac{\mathsf{svc} \in \{\mathsf{ipairs}, \mathsf{next}, \mathsf{pairs}, \mathsf{getmetatable}, ...\}}{\theta : \textbf{\$builtIn}\ \mathsf{svc}\ (\mathsf{v}_1, ..., \mathsf{v}_n) \rightarrow^{\theta} \theta : \delta(\mathsf{svc}, \mathsf{v}_1, ..., \mathsf{v}_n, \theta)}\ (\textsc{Svc-Const})$$

$$\frac{\mathsf{svc} \in \{\mathsf{rawset}, \mathsf{setmetatable}\} \qquad (\theta_2, \mathsf{v}) = \delta(\mathsf{svc}, \mathsf{v}_1, ..., \mathsf{v}_n, \theta_1)}{\theta_1 : \textbf{\$builtIn}\ \mathsf{svc}\ (\mathsf{v}_1, ..., \mathsf{v}_n) \rightarrow^{\theta} \theta_2 : \mathsf{v}}\ (\textsc{Svc-Update})$$

Fig. 8: Interface with the δ function.

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).[8] 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 if 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 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

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

$$\mathsf{indexmeta}(\mathsf{tid}, \mathsf{v}, \theta) \doteq \delta(\mathsf{rawget},\ \mathsf{tid}, \mathsf{v}, \theta)$$
$$\mathsf{protmeta}(\mathsf{tid}, \theta)\ \doteq \mathsf{indexmeta}(\mathsf{tid}, \text{"\_\_metatable"}, \theta)$$
$$\mathsf{prot?}(\mathsf{tid}, \theta)\ \doteq \mathsf{protmeta}(\mathsf{tid}, \theta) \neq \mathbf{nil}$$

$$\delta(\mathsf{setmetatable}, \mathsf{tid}, \mathsf{v}, \theta_1) = (\mathsf{tid}, \theta_2) \quad \textit{if } \delta(\mathsf{type}, \mathsf{v}) \in \{\text{"table"}, \text{"nil"}\} \wedge \neg\mathsf{prot?}(\mathsf{tid}, \theta_1) \wedge \theta_2 = \theta_1[\mathsf{tid} := (\theta_1(\mathsf{tid})(1), \mathsf{v})]$$

$$\delta(\mathsf{setmetatable}, \mathsf{v}_1, \mathsf{v}_2, \theta) = (\delta(\mathsf{error}, \text{"}\ldots\text{"}), \theta) \quad \textit{if } \delta(\mathsf{type}, \mathsf{v}_2) \notin \{\text{"table"}, \text{"nil"}\} \vee \mathsf{prot?}(\mathsf{tid}, \theta_1) \vee \delta(\mathsf{type}, \mathsf{v}_1) \neq \text{"table"}$$

$$\delta(\mathsf{getmetatable}, \mathsf{tid}, \theta) = \begin{cases} \mathsf{protmeta}(\mathsf{tid}, \theta) & \textit{if } \mathsf{prot?}(\mathsf{tid}, \theta) \\ \theta(\mathsf{tid})(2) & \textit{otherwise} \end{cases}$$

$$\delta(\mathsf{getmetatable}, \mathsf{v}, \theta) = \begin{cases} \mathbf{nil} & \textit{if } \mathsf{tid}_{\delta(\mathsf{type},\mathsf{v})} \notin \mathsf{dom}(\theta) \\ \mathsf{tid}_{\delta(\mathsf{type},\mathsf{v})} & \textit{if } \mathsf{tid}_{\delta(\mathsf{type},\mathsf{v})} \in \mathsf{dom}(\theta) \wedge \neg\mathsf{prot?}(\mathsf{tid}_{\delta(\mathsf{type},\mathsf{v})}, \theta) \\ \mathsf{protmeta}(\mathsf{tid}_{\delta(\mathsf{type},\mathsf{v})}, \theta) & \textit{otherwise} \end{cases}$$

Fig. 9: Primitives for setting and getting metatables.

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 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 $\mathsf{tid}_{\delta(\mathsf{type},\mathsf{v})}$. Then, determining if such metatable is set reduces to ask if $\mathsf{tid}_{\delta(\mathsf{type},\mathsf{v})} \in \theta$. Note that such metatables might be protected too.

4.4.2 Resolving labeled terms (excerpt).

In this section we introduce a new relation $\rightarrow^{\mathsf{meta}}$ 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 $\rightarrow^{\mathsf{meta}}$ relation for field updates. The inner workings of the metatables mechanism is handled by the meta-function new_index, while $\rightarrow^{\mathsf{meta}}$ 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 (NonTable), 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

$$\mathsf{new\_index}(\theta : (\!|\, \mathsf{v}_1\ [\mathsf{v}_2] = \mathsf{v}_3 \,|\!)_{\textsc{label}}) = \theta : \textbf{\$statFcall}\ \mathsf{v}_4\ (\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3) \quad \textit{if} \begin{cases} \textsc{label} \in \{\textsc{WrongKey}, \textsc{NonTable}\} \\ \mathsf{v}_4 = \mathsf{indexmeta}(\mathsf{v}_1, \text{"__newindex"}, \theta) \\ \delta(\mathsf{type}, \mathsf{v}_4) = \text{"function"} \end{cases}$$

$$\mathsf{new\_index}(\theta : (\!|\, \mathsf{v}_1\ [\mathsf{v}_2] = \mathsf{v}_3 \,|\!)_{\textsc{label}}) = \theta : \mathsf{v}_4\ [\mathsf{v}_2] = \mathsf{v}_3 \quad \textit{if} \begin{cases} \textsc{label} \in \{\textsc{WrongKey}, \textsc{NonTable}\} \\ \mathsf{v}_4 = \mathsf{indexmeta}(\mathsf{v}_1, \text{"__newindex"}, \theta) \\ \mathsf{v}_4 \neq \textbf{nil} \\ \delta(\mathsf{type}, \mathsf{v}_4) \neq \text{"function"} \end{cases}$$

$$\mathsf{new\_index}(\theta_1 : (\!|\, \mathsf{tid}\ [\mathsf{v}_1] = \mathsf{v}_2 \,|\!)_{\textsc{WrongKey}}) = \theta_2 : \ ; \quad \textit{if} \begin{cases} \mathsf{indexmeta}(\mathsf{tid}, \text{"__newindex"}, \theta_1) = \textbf{nil} \\ (\theta_2, \mathsf{tid}) = \delta(\mathsf{rawset}, \mathsf{tid}, \mathsf{v}_1, \mathsf{v}_2, \theta_1) \end{cases}$$

$$\mathsf{new\_index}(\theta_1 : (\!|\, \mathsf{tid}\ [\mathsf{v}_1] = \mathsf{v}_2 \,|\!)_{\textsc{WrongKey}}) = \theta_2 : \textbf{\$err}... \quad \textit{if} \begin{cases} \mathsf{indexmeta}(\mathsf{tid}, \text{"__newindex"}, \theta_1) = \textbf{nil} \\ (\theta_2, \textbf{\$err}...) = \delta(\mathsf{rawset}, \mathsf{tid}, \mathsf{v}_1, \mathsf{v}_2, \theta_1) \end{cases}$$

$$\mathsf{new\_index}(\theta_1 : (\!|\, \mathsf{tid}\ [\mathsf{v}_1] = \mathsf{v}_2 \,|\!)_{\textsc{NonTable}}) = \theta_2 : \delta(\mathsf{error}, \text{"error updating..."}) \quad \textit{if}\ \mathsf{indexmeta}(\mathsf{tid}, \text{"__newindex"}, \theta_1) = \textbf{nil}$$

$$\frac{\mathsf{new\_index}(\theta_1 : (\!|\, \mathsf{v}_1\ [\mathsf{v}_2] = \mathsf{v}_3 \,|\!)_{\textsc{label}} = \theta_2 : \mathsf{s}}{\theta_1 : (\!|\, \mathsf{v}_1\ [\mathsf{v}_2] = \mathsf{v}_3 \,|\!)_{\textsc{label}} \rightarrow^{\mathsf{meta}} \theta_2 : \mathsf{s}}\ (\textsc{M-Upd})$$

Fig. 10: Metatable mechanism for field update.

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 $\delta(\mathsf{error})$—which is introduced later in §4.5. Finally, if the label is NonTable, 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 $\mathsf{E}[\![\,\textbf{\$err}\ \mathsf{v}\,]\!]$, 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:

$$\mathsf{E'}[\![\,\textbf{\$err}\ \mathsf{v}\,]\!] \ \rightarrow^{\mathsf{s/e}}\ \textbf{\$err}\ \mathsf{v}$$

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

$$\frac{\mathsf{t} \ \rightarrow^{\mathsf{s/e}}\ \mathsf{t'}}{\sigma : \theta : \mathsf{E''}\,[\![\,\mathsf{t}\,]\!] \ \mapsto\ \sigma : \theta : \mathsf{E''}\,[\![\,\mathsf{t'}\,]\!]}$$

we have several possible ways of propagating an error, depending on the decomposition of context E between contexts $\mathsf{E'}$ and $\mathsf{E''}$, 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 **$err** v, which allows for the inclusion of any Lua value as information attached to the error. $(\!|\, e \,|\!)^{v}_{\text{PROTMD}}$ and $(\!|\, E \,|\!)^{v}_{\text{PROTMD}}$ represent an expression or a context that must run in protected mode, where v is the error handler passed to **xpcall**. Treating function calls in protected mode just as expressions will simplify the semantics, as shown below. We also include the same protected mode labels but without the handlers, which has a particular meaning to be shown below. 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).

Grammar

$s ::= \ldots \mid \textbf{\$err}\ v$

$e ::= \ldots \mid \textbf{\$err}\ v \mid (\!|\ e\ |\!)^{v}_{\text{PROTMD}} \mid (\!|\ e\ |\!)_{\text{PROTMD}}$

$E ::= \ldots \mid (\!|\ E\ |\!)^{v}_{\text{PROTMD}} \mid (\!|\ E\ |\!)_{\text{PROTMD}}$

Semantics

$$\frac{E_{np} \neq [\![\ ]\!]}{\sigma : \theta : E_{np}\,[\![\ \textbf{\$err}\ \mathsf{v}\ ]\!] \ \mapsto\ \sigma : \theta\ :\ \textbf{\$err}\ \mathsf{v}} \ \text{(E-TERMINATION)}$$

$$(\!|\ \langle \mathsf{v}, \ldots \rangle\ |\!)^{\mathsf{v}'}_{\text{PROTMD}} \ \rightarrow^{\mathsf{s/e}}\ \langle \textbf{true}, \mathsf{v}, \ldots \rangle \ \text{(E-PROTTRUE)}$$

$$\frac{\delta(\mathsf{type}, \mathsf{v}_2) = \text{"function"}}{(\!|\ E_{np}\,[\![\ \textbf{\$err}\ \mathsf{v}_1\ ]\!]\ |\!)^{\mathsf{v}_2}_{\text{PROTMD}} \ \rightarrow^{\mathsf{s/e}}\ (\!|\ (\mathsf{v}_2(\mathsf{v}_1))\ |\!)_{\text{PROTMD}}} \ \text{(E-PROTHANDLER)}$$

$$\frac{\delta(\mathsf{type}, \mathsf{v}_2) \neq \text{"function"}}{(\!|\ E_{np}\,[\![\ \textbf{\$err}\ \mathsf{v}_1\ ]\!]\ |\!)^{\mathsf{v}_2}_{\text{PROTMD}} \ \rightarrow^{\mathsf{s/e}}\ \langle \textbf{false}, \text{"error"} \rangle} \ \text{(E-PROTHANDLERERR)}$$

$$(\!|\ \mathsf{v}\ |\!)_{\text{PROTMD}} \ \rightarrow^{\mathsf{s/e}}\ \langle \textbf{false}, \mathsf{v} \rangle \ \text{(E-PROTFALSE)}$$

$$(\!|\ E_{np}\,[\![\ \textbf{\$err}\ \mathsf{v}_1\ ]\!]\ |\!)_{\text{PROTMD}} \ \rightarrow^{\mathsf{s/e}}\ \langle \textbf{false}, \text{"error"} \rangle \ \text{(PROTERR)}$$

$$\delta(\textbf{xpcall}, \mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3, \ldots) = (\!|\ \mathsf{v}_2\ (\mathsf{v}_3, \ldots)\ |\!)^{\mathsf{v}_2}_{\text{PROTMD}}$$

Fig. 11: Error object and protected mode.

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. Finally, note that, by considering function calls in protected mode just as expression, we simplify the last step of the mechanism: we do not need to reason if a given tuple must be discarded or not. We are into an expression, so the tuples are always returned.

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 $\sigma : \theta : \mathtt{t}$ must hold to be well-formed, for the given stores σ and θ, and term $\mathtt{t}$ (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 contextual information required to check the well-formedness of $\mathtt{t}$. In essence, C keeps track of local variables, while loops, labeled terms and function definitions. Using just term contexts to keep track of contextual information, allows us to include every information needed to reason about well-formedness in a single object, that is also just a kind of term context. That is, we do not need to define new constructs with the corresponding operations to manipulate them. We just reuse the language constructions, and the operations for matching and manipulation of contexts. For example, we can maintain information about variables in scope, but also occurrences of while loops or labeled terms, with the same kind of construction, and ask about that information by means of plain pattern matching.

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

$\vdash_{wf\mathtt{t}}$ is defined recursively over the structure of the term $\mathtt{t}$, as can be seen, for example, in WF-If. Some rules require checking side conditions, like WF-Ref, in which the reference must exists in the corresponding store, in order to check for

Contexts for well-formedness of terms (excerpt)

$$C ::= [\![\,]\!] \mid \textbf{while } e \textbf{ then } C \mid (\!|\, C \,|\!)_{\textsc{Break}} \mid \textbf{local } x \textbf{ , } \ldots \textbf{ = } e \textbf{ , } \ldots \textbf{ in } C \mid \textbf{function } \ell(x \textbf{ , } \ldots)\ C \textbf{ end} \mid \ldots$$

Well-formedness of terms (selected rules)

$$\frac{\mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{e} \qquad \mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{s}_1 \qquad \mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{s}_2}{\mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \textbf{if } \mathsf{e} \textbf{ then } \mathsf{s}_1 \textbf{ else } \mathsf{s}_2} \ (\text{WF-If})$$

$$\frac{\mathsf{r} \in \mathsf{dom}(\sigma)}{\mathsf{C} \ \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{r}} \ (\text{WF-Ref})$$

$$\mathsf{C}' [\![\, \textbf{local } \mathsf{x}_1, ..., \mathsf{x}, \mathsf{x}_2, ... = \mathsf{e}, ... \textbf{ in } \mathsf{C}'' \,]\!] \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{x} \ (\text{WF-VarLocal})$$

$$\mathsf{C}' [\![\, \textbf{function } \mathsf{x}_1\ (\mathsf{x}_2, ..., \mathsf{x}, \mathsf{x}_3, ...)\ \mathsf{C}'' \,]\!] \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{x} \ (\text{WF-VarFun})$$

$$\frac{\mathsf{C'} \neq \mathsf{C''} \ [\![\, \textbf{function } \ell\ (x, \ldots)\ \mathsf{C''}' \textbf{ end} \,]\!]}{\mathsf{C} [\![\, \textbf{while } \mathsf{e} \textbf{ do } \mathsf{C'} \textbf{ end} \,]\!] \vdash_{wf\mathsf{t}} \sigma : \theta : \textbf{break}} \ (\text{WF-BreakWhile})$$

$$\frac{\mathsf{C'} \neq \mathsf{C''} \ [\![\, \textbf{function } \ell\ (x, \ldots)\ \mathsf{C''}' \textbf{ end} \,]\!]}{\mathsf{C} [\![\, (\!|\, \mathsf{C'} \,|\!)_{\textsc{Break}} \,]\!] \vdash_{wf\mathsf{t}} \sigma : \theta : \textbf{break}} \ (\text{WF-BreakUnfolded})$$

$$\frac{\mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{e} \ldots \qquad \mathsf{C} [\![\, \textbf{local } \mathsf{x}, \ldots = \mathsf{e}, \ldots \textbf{ in } [\![\,]\!] \,]\!] \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{s}}{\mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \textbf{local } \mathsf{x}, \ldots = \mathsf{e}, \ldots \textbf{ in } \mathsf{s}} \ (\text{WF-Local})$$

$$\frac{\mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{e} \qquad \mathsf{C} [\![\, \textbf{while } \mathsf{e} \textbf{ do } [\![\,]\!] \,]\!] \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{s}}{\mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \textbf{while } \mathsf{e} \textbf{ do } \mathsf{s}} \ (\text{WF-While})$$

$$\frac{\mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{v}_2 \qquad \mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{v}_3 \qquad \mathsf{v}_1 \in \mathsf{dom}(\theta) \qquad \mathsf{v}_2 \notin \mathsf{dom}(\theta(\mathsf{v}_1)(1)) \qquad \mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{v}_1}{\mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : (\!|\, \mathsf{v}_1[\mathsf{v}_2] = \mathsf{v}_3 \,|\!)_{\textsc{WrongKey}}} \ (\text{WF-WrongKey})$$

Well-formedness of configurations

$$\begin{aligned} \mathsf{C} \vdash_{wf\mathsf{c}} \sigma : \theta : \mathsf{t} \doteq\ & \forall \mathsf{r} \in \mathsf{dom}(\sigma), \mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \sigma(\mathsf{r}) && (1)\\ & \wedge\ \forall \mathsf{tid} \in \mathsf{dom}(\theta), \mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \theta(\mathsf{tid})(1) && (2)\\ & \wedge\ \mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \theta(\mathsf{tid})(2) && (3)\\ & \wedge\ \mathsf{C} \vdash_{wf\mathsf{t}} \sigma : \theta : \mathsf{t} && (4) \end{aligned}$$

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

the absence of *free occurrences* of references, that would render the corresponding term *stuck*. Additionally, some rules require specific contextual information: for a variable x, we need to check that the context contains a **local** or **function** signature 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). In both cases, we also need to check that the **break** statement is actually *immediately* inside the loop or labeled term: this is, that the statement is not inside another construction surrounding it, like a function definition. The contextual 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 representing situations that must be handled by meta-tables, we need to make sure that 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) **$err** v
(b) **return** v, ...
(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 $\vdash_{wfc} \sigma' : \theta' : s'$.

Note that in order to prove this and the following statements, we require a clear formalization of the operations of matching and decomposition.Then, we could prove the soundness of $\vdash_{wfc}$ by induction on the structure of the proof of $\vdash_{wfc} \sigma : \theta : s$.

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 pointed by an evaluation context, 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) $\theta : t$
  (4) $\sigma : \theta : t$

In order to prove it, given that we are looking for the next redex into a given term, we could rely on the information that we have in the proof of $\vdash_{wf\mathsf{c}} \sigma : \theta : \mathsf{s}$. It checks the side-conditions of every semantics rule, in order to guarantee that a given term is not stuck: that is, if it is not a result, then it is a redex. Then, we could prove this statement by means of induction on the structure of the proof of $\vdash_{wf\mathsf{c}} \sigma : \theta : \mathsf{s}$.

Additionally, the proof will depend on a correct correspondence between evaluation contexts and semantics relations. That is, that each evaluation context should point to just one term that is the next redex, according to the semantics relations.

From these properties we infer two relevant corollaries. The first one is a direct consequence of the previous lemmas: there are no stuck terms, and the semantics is deterministic. Hence:

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 $\vdash_{wf\mathsf{c}} \sigma : \theta : s$, just one of the following situations holds:

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

Finally, the following corollary of soundness of $\vdash_{wf\mathsf{c}}$ 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' identifiers), and well-formed configuration $\sigma : \theta : \mathsf{s}$, if there exists σ' and θ' such that $\sigma : \theta : \mathsf{s} \mapsto \sigma' : \theta' : \mathsf{s}'$, then s' is a closed term.*

Note that, otherwise, the resulting configuration $\sigma' : \theta' : \mathsf{s}'$ could not be considered well-formed, as the soundness of $\vdash_{wf\mathsf{c}}$ guarantees. This means that, once the focus of execution has reached to a closure, every external local 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 in 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 support 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:

$$\frac{\sigma' = \sigma[\mathsf{r} := \mathsf{v}]}{\sigma : \mathsf{r} = \mathsf{v} \;\rightarrow^{\sigma}\; \sigma' : ;} \;(\textsc{Local-Assgn})$$

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.[9] 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.

[9] 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.

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) (briefly introduced in §7), we had to introduce two changes in order to better emulate the behavior of a garbage collector. First, we change the introduction 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). We do it in a way that the variables' references that are still in scope are maintained (in some structure) through the whole execution of the code in their scope.

In a common implementation of a compiler of a programming language with reachability-based garbage collection, a variable that is in scope is always considered reachable, regardless of its actual reachability from what's left to be computed. This is so because of the way in which the root set of references is calculated (for example, just by looking at the stack of a function call).[10] On the other hand, in the model presented so far, once that a variable is not reachable from what's left to be computed in its scope, we would consider it unreachable. It doesn't matter if the code that belongs to its scope is still under execution. This difference with real implementations naturally introduces some observable effects in the context of a programming language with interfaces to its garbage collector, as it happens with Lua 5.2.

In order to better approximate the behavior of a real implementation, we keep the references that correspond to introduced variables 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 references 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 closer as to how the official implementation of Lua works, and allows us to have a model that better represents the common phenomenons associated with garbage collection.

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
```

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

[10] The first set of references from which reachability is computed.

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, and Redex itself.

As in (Politz et al., 2012), we also split desugaring into 2 pieces: a compiler from Lua source code to our language that produces (possible) open terms; and a definition of an evaluation context into which the previously compiled term is plugged. This context contains the standard implementation of the execution environment: a table, containing the set of bindings to library services, which, in our model, are just invocations to the services through the **$builtIn** construction. This is our definition of the original execution environment _ENV (see §2.2). Under normal circumstances, load will assume that the global variables in the interpreted code refer to said environment: they are just fields of the table bound to _ENV. However, the user can pass, as an argument to load, a table that will be used as a replacement to _ENV. In that case, global variables will be interpreted as fields of the given table.

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.*).

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 where 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.

– 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.[11]

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.

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, 26 minutes to completion without testing garbage collection, and 1:22 hours including

[11] While we do model tail calls, the actual testing of the mechanism in the official test suite consists in performing several recursive calls, beyond the stack limit, to actually test if the mechanism is being used. Given that performing such amount of recursive calls is not feasible within our mechanization (because of the time required), we do not include such tests.

it.[12] 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.[13] Given the computational model of reduction semantics—in which at each step the whole configuration is matched against the (possible) several patterns of 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 2 minutes on the same 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

[12] On a system running Arch Linux updated to May, 2022, Racket v8.3, and 12 parallel processes on an Intel Core i7-10870H CPU @ 4GHz × 8, with 16GB of RAM.

[13] Remember that, at each iteration, a new local variable i is created, to contain the corresponding value in the interval $[1, 100]$.

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.[14]

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 preparing 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 are interested in random-testing are predicates

[14] 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.

quantified over a well-formed configuration $\sigma : \theta : \mathsf{s}$, *i.e.*, not just terms that matches against the left-hand side of the semantic rules. Therefore, to increase the confidence on the testing, we generate just a fraction of the total cases following the semantics rules, and ask redex-check to generate the remaining terms by just following the pattern $\sigma : \theta : \mathsf{s}$.

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 50.000 attempts we obtain, in an average of ten runs, close to 43.000 well-formed configurations that successfully pass the tests, with complete coverage of the rules.

To better understand the high percentage of well-formed configurations, note that our preparation function takes ill-formed terms and tries to turn them into well-formed ones, also attempting to directly replace conflicting sub-terms by (simpler) well-formed ones.

During random-testing, the terms produced by redex-check may grow in size and complexity after each failed attempt at generating a term. It is possible to specify a rate at which the terms grow in size, by providing a bound to it. According to the documentation of redex-check, said bound *"bounds the height of the generated term (measured as the height of its parse tree)"*. We used the default bound growth rate which grows as the base 5 logarithm of the number of previous attempts at generating terms. To get a general idea of the complexity of the terms generated, on a run with 50.000 attempts, the maximum bound used is 6, which may generate configurations with programs consisting of, roughly, 250 statements (each with its expressions), and stores with 30 elements (without counting those added by the preparation function).[15]

7 A use case: modeling garbage collection

In this section we present a brief tour to an existing application of the dynamic semantics presented so far. The content is a distilled version of Soldevila et al. (2020).

7.1 Garbage collection

As we already mentioned, we used our dynamic semantics to model garbage collection (GC) in Lua. In essence, the task reduces to include a non-deterministic semantics rule that performs a GC cycle. Now, in order to reason about its behavior, we need to study it in the context of a complete dynamic semantics and see how it interacts with the rest of the semantics model. For plain GC, *i.e.*, without considering the interfaces with the garbage collector, we expect each collection step to preserve the semantics of the program. This amounts to *observing* the same behavior of the program before and after the GC step.

[15] To obtain these measures we added custom Racket code to our module Tests/RandomTesting/soundness/soundness_rand_test.rkt. We do not include it in the provided source code, tough the code reduces to counting well-formed configurations and recognizing the biggest configurations generated.

A more interesting situation arises when we include the interfaces with the garbage collector: *weak tables* and *finalizers*. Weak tables are tables whose keys and/or values are referred by *weak references*. These references are not taken into account by the garbage collector when determining reachability, and can therefore be collected at any time. Finalizers are similar, but not equal, to *destructors* in languages with explicitly-managed memory. Such interfaces might introduce observably diverging results. Our work enables the development and certification of static analyzers for ruling out potentially-dangerous non-determinism from a program.

In this section we first present **LuaSafe** (§7.2), a tool that uses the modeled semantics of weak tables to ensure deterministic execution of programs, and then we take a sneak peak at how we model (simple) garbage collection (§7.3).

7.2 **LuaSafe** in a nutshell

LuaSafe is a prototype static analyzer that aims at the detection of misuses of weak tables, that could lead to non-deterministic behavior. While the general problem is known to be undecidable (Kevin Donnelly, 2006), we propose an approximation to the solution by combining techniques from statics semantics (type inference, type checking and data-flow analysis) together with weak tables' semantics.

For a given Lua program `p`, we say that `p` is *gc-safe* if it exhibits a deterministic behavior under $\mapsto$ extended with GC and its interfaces. Then, we denote with P_{safe} the set of gc-safe programs. In our approach we aim at taking a user program and trying our best to guess if it belongs to P_{safe}, without asking the user for modifications of the program or to use weak tables according to some particular idioms, as proposed in Kevin Donnelly (2006).

In order to achieve this, **LuaSafe** proceeds in a series of steps, including a type system which makes heavy use of the formal tools required to understand Lua's GC. In the coming section we make a dip at the underlying formalism.

7.3 Modeling garbage collection

The purpose of GC is to remove from memory (the store) information that will not be used by the remaining computations of the program. One of the simplest and commonly used approaches to find such information is based on the notion of *reachability* (*e.g.*, Gabay and Kfoury, 2007). The idea is simple: given the set of references that literally occur in the program (the *root set*), it must be the case that any *information* (*e.g.*, value in a store) that may be used by the program must be *reachable* from that set. Conversely, any *binding* (a reference with its value) in the store that cannot be reached from the root set, will not be accessible from the program and, therefore, can be safely removed as it will not be needed in the remaining computations of the program.

Lua implements two reachability-based GC strategies: a *mark-and-sweep* collector (the default) and a *generational collector*. In this section we will provide a specification for the behavior of a typical reachability-based GC that encompasses the essential details of the behavior of the two algorithms included in Lua and any other based on reachability.

In the context of this work, those values which are not reachable will be called *garbage*. This notion, sufficient to model Lua's GC, is purely syntactic: it will take into account just the literal occurrence of references in the program, or their reachability from this set of references that occur literally, to determine if a given value is garbage or not. In contrast, there are approaches, to identify garbage, where also the semantics of the program may be taken into account (*e.g.*, Kevin Donnelly, 2006).

To formally capture the notion of garbage, it will be easier to begin with the definition of reachable references. The only difference worth to mention, in comparison with common definitions found in the literature (Leal and Ierusalimschy, 2005; Gabay and Kfoury, 2007), is the inclusion of metatables: a metatable of a reachable table is considered reachable, so a reachability *path*, that is, a path between a reference and the root set, might also go through a metatable.

Informally, a location (value reference or an identifier) will be reachable with respect to a given term t, and corresponding stores, if one of the following conditions hold:

- The location occurs literally in t.
- The location is reachable from the information associated with a reachable location. This includes:
  - The location is reachable from the closure associated with a reachable location.
  - The location is reachable from the table associated with a reachable location.
  - The location is reachable from a metatable of a reachable table identifier.

This is formalized in the following definition:

Definition 2 (Reachability for Simple GC) We say that a location $\mathsf{l} \in \mathsf{r} \cup \mathsf{tid}$ is *reachable* in term t, given stores σ and θ, iff:

$$\begin{aligned}\mathsf{reach}(\mathsf{l},\mathsf{t},\sigma,\theta) = {} & \mathsf{l} \in \mathsf{t} \vee \\ & (\exists \mathsf{r} \in \mathsf{t},\ \mathsf{reach}(\mathsf{l},\sigma(\mathsf{r}),\sigma \setminus \mathsf{r},\theta)) \vee \\ & \exists\, \mathsf{tid} \in \mathsf{t},\ (\mathsf{reach}(\mathsf{l},\pi_1(\theta(\mathsf{tid})),\sigma,\theta \setminus \mathsf{tid}) \vee \\ & \qquad \mathsf{reach}(\mathsf{l},\pi_2(\theta(\mathsf{tid})),\sigma,\theta \setminus \mathsf{tid}))\end{aligned}$$

The model proposed in Soldevila et al. (2020) manages closures through references, in a similar fashion as with tables. Here, we avoid that aspect in order to keep it simpler and closer to the model presented in this work, keeping in mind that it is only and approximation of the actual notion of reachability we use in the cited work.

We write $l \in t$ to indicate that l occurs literally in term t, and write $\gamma \setminus l$ as the store obtained by removing the binding of l in γ. Informally, this predicate states that either l occurs in t, or there is a reference in t such that l is reachable from it.

To avoid cycles generated from mutually recursive definitions, in the stores, that would render undefined the preceding predicate, we remove from the stores the bindings already considered. We assume the predicate is false if a given location occurs in t but does not belong to the domain of any of the stores.

Note that for a table *tid* we not only check its content ($\pi_1(\theta(\mathsf{tid}))$) but also its metatable ($\pi_2(\theta(\mathsf{tid}))$). That is, a table's metatable is considered reachable when the table itself is reachable. Observe that, being metatables ordinary tables,

they can contain other tables' ids or even closures, which in turn may have other locations embedded into them. Naturally, if metatables were not taken into account for reachability, we could run straight into the problem of dangling references any time a metamethod is recovered from the metatable. Also, note that during the recursive call $\mathsf{reach}(\mathsf{l}, \pi_2(\theta(\mathsf{tid})), \sigma, \theta \setminus \mathsf{tid})$, at first it will determine if l is exactly $\pi_2(\theta(\mathsf{tid}))$ (because it asks for $\mathsf{l} \in \pi_2(\theta(\mathsf{tid}))$, for $\pi_2(\theta(\mathsf{tid}))$ being either **nil** or a table identifier) and, if not, it will continue with the inspection of the content of the metatable, by dereferencing its id, given that it is not **nil**. Hence, we do not remove $\pi_2(\theta(\mathsf{tid}))$ from θ in the mentioned recursive call.

Specification of a garbage collection cycle. We keep abstract the specification of a cycle of GC in order to accommodate to any implementation of GC:

Definition 3 (Simple GC cycle)
$\mathsf{gc}(\mathsf{s}, \sigma, \theta) = (\sigma_1, \theta_1)$, where:

- $\sigma = \sigma_1 \uplus \sigma_2$
- $\theta = \theta_1 \uplus \theta_2$
- $\forall \mathsf{l} \in \mathsf{dom}(\sigma_2) \cup \mathsf{dom}(\theta_2), \neg\mathsf{reach}(\mathsf{l}, \mathsf{s}, \sigma, \theta)$

We use $\gamma_1 \uplus \gamma_2$ to denote the union of stores with disjoint domains. This specification states that $\mathsf{gc}(\mathsf{s}, \sigma, \theta)$ returns two stores, σ_1 and θ_1, which are a subsets of the stores provided as arguments, σ and θ. We do not specify how these subsets are determined. We just require that the remaining part of the stores (σ_2 and θ_2) do not contain references that are reachable from the program s. Satisfied this condition, it is safe to run code s in the new stores σ_1 and θ_1, as no dereferencing of a dangling pointer may occur.

Observe that the previous specification does not impose σ_1 and θ_1 to be *maximal*, meaning they might have non-reachable references with respect to s.

Using the previous specification of GC, we can extend our model of Lua with a non-deterministic step of GC:

$$\frac{(\sigma', \theta') = \mathsf{gc}(\mathsf{s}, \sigma, \theta) \qquad \sigma' \neq \sigma \vee \theta' \neq \theta}{\sigma : \theta : \mathsf{s} \mapsto \sigma' : \theta' : \mathsf{s}}$$

We require it to actually perform some changes to the stores to ensure progress. This obviously introduces non-determinism: at any time, as long as there is some garbage left, we can choose to collect the garbage or to continue with the execution of the program. But, for the definition provided so far, this non-determinism should not change the behavior of the program: every execution path will eventually lead to the same result. This property will not longer be true when extending GC with the interfaces with the garbage collector present in Lua 5.2. We further develop the model in the cited bibliography.

8 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 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 is formally specified in its core documentation (Rossberg A, 2021), and introduced in (Haas et al., 2017).

Another 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 other 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.

9 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 with coroutines and goto being the most salient missing features.

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, that follows 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. We are currently interested in that venue of research, and leave for future research extending and updating the model to current versions of Lua.

From the missing aspects of the semantics, the most interesting ones from a semantic point of view are **goto** and coroutines. About the former, it is relevant to mention that it is not a widely used keyword. A code search[16] shows only 37k occurrences of "goto" in about two millions of files, including a significant fraction of spurious occurrences in comments. As such, it seems the effort required to extend the semantics seems unjustified: program configurations has to be enriched to include information about possible jumps, and always keep the already executed code. This is studied in detail in Krebbers and Wiedijk (2013) for a C-like language.

Coroutines, on the other hand, are significantly more used (a similar search returns about 164k hits). Luckily, they are already understood from a formal point of view: Moura et al. (2004) presents a reduction semantics specifically tailored for expressing Lua's coroutines, and it seems it could be easily added on top of the concepts already present in our model.

[16] `https://sourcegraph.com/search?q=context:global+lang:Lua+%22+goto+%22+count:1000000&patternType=regexp&case=yes`

References

Adobe (2019) Adobe Lightroom®. https://www.adobe.io/apis/creativecloud/lightroom.html, accessed: 2020-05-04

Bodin M, Charguéraud A, Filaretti D, Gardner P, Maffeis 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, Finlder RB, Flatt M (2009) Semantics Engineering with PLT Redex. The MIT Press

Gabay Y, Kfoury AJ (2007) A calculus for java's reference objects. SIGPLAN Not 42(8):9–17, DOI 10.1145/1294297.1294299, URL http://doi.acm.org/10.1145/1294297.1294299

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

Haas A, Rossberg A, Schuff DL, Titzer BL, Holman M, Gohman D, Wagner L, Zakai A, Bastien J (2017) Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA, PLDI 2017, p 185–200, DOI 10.1145/3062341.3062363, URL https://doi.org/10.1145/3062341.3062363

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

Ierusalimschy R, de Figueiredo LH, Celes W (2001a) The evolution of an extension language: a history of lua. In: Brazilian Symposium on Programming Languages

Ierusalimschy R, de Figueiredo LH, Celes W (2001b) The evolution of an extension language: a history of Lua. In: Brazilian Symposium on Programming Languages

Ierusalimschy 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

Kevin Donnelly AK J J Hallett (2006) Formal semantics of weak references. In: ISMM '06 Proceedings of the 5th international symposium on Memory management, pp 126–137

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, Rafkind 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
Krebbers R, Wiedijk F (2013) Separation logic for non-local control flow and block scope variables. In: FOSSACS'13, DOI 10.1007/978-3-642-37075-5_17
Leal MA, Ierusalimschy R (2005) A formal semantics for finalizers. J UCS 11(7):1198–1214, DOI 10.3217/jucs-011-07-1198, URL https://doi.org/10.3217/jucs-011-07-1198
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) Luatex. http://www.luatex.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 A, Rodriguez N, Ierusalimschy R (2004) Coroutines in lua. Journal of Universal Computer Science 10
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