Understanding Lua's Garbage Collection

Towards a Formalized Static Analyzer

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

Beta Ziliani
FAMAF, UNC and CONICET
Argentina
beta@mpi-sws.org

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

ABSTRACT

We provide the semantics of garbage collection (GC) for the Lua programming language. Of interest are the inclusion of *finalizers* (akin to destructors in object-oriented languages) and *weak tables* (a particular implementation of weak references). The model expresses several aspects relevant to GC that are not covered in Lua's documentation but that, nevertheless, affect the observable behavior of programs.

Our model is mechanized and can be tested with real programs. Our long-term goal is to provide a formalized static analyzer of Lua programs to detect potential dangers. As a first step, we provide a prototype tool, **LuaSafe**, that typechecks programs to ensure their behavior is not affected by GC. Our model of GC is validated in practice by the experimentation with its mechanization, and in theory by proving several soundness properties.

CCS CONCEPTS

• **Theory of computation → Operational semantics**; **Program analysis**.

ACM Reference Format:
Mallku Soldevila, Beta Ziliani, and Daniel Fridlender. 2020. Understanding Lua's Garbage Collection: Towards a Formalized Static Analyzer. In *22nd International Symposium on Principles and Practice of Declarative Programming (PPDP '20), September 8–10, 2020, Bologna,Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3414080.3414093

1 INTRODUCTION

Lua is an extensively used imperative scripting language. Its popularity grows to the point that it currently has several interpreters and compilers [5], and static analyzers [3]. Among its advocates, Lua has a long standing support within the game industry [10]. However, while being a very fast scripting language, it is noted in *ibid* that:

> *"Using Lua on performance-constrained platforms can definitely be a challenge if you don't understand the ins and outs of Lua's memory usage."*

In particular, Lua's garbage collector (GC) offers a rich interface to let the developer efficiently deal with memory. For instance, it is possible to create a *weak table*, that is, a Lua *table* (akin to a JavaScript's' associative array) whose keys or values are weak references. Thus, when performing garbage collection (also noted as GC), it might decide to collect keys or values from a weak table, even if the table is still in scope.

```
1  local t = {}                          --create an empty table
2  setmetatable(t, {__mode = 'v'})       --set its values as weak
3  t[1] = {}                             --assign an empty table to key 1
4  local i = 0
5  while true do
6    i = i + 1
7    ...                                 --some code, possibly generating garbage
8    if not t[1] then break end
9  end
10
11 return i                              --this value cannot be predicted
```

Figure 1: A non-deterministic program using a weak table.

If improperly used, weak tables can easily break the program's invariants, as the simple program listed in Figure 1 shows. In this program, a table t is created containing only one value, another table referred by a weak reference, and without any other variable bound directly or indirectly to it. That is, there is no other path to the value using only *strong* (*i.e.*, regular) references. Then, such value can be GC'ed at any time, making true the condition **not** t[1] at an arbitrary number of iterations of the loop (the **if** breaks the loop when the value in t[1] is **nil**). Therefore, the returned number of iterations i cannot be predicted.

Weak tables are used mainly for caching values [8], and a good use of such tables will ensure the references are valid prior to accessing them. However, in a realistic program manually validating every use of weak tables is error-prone and, for this reason, it is proposed in [13] that weak references be only used within the scope of a library, subject to a larger scrutiny and testing. However, testing is due to fail given the non-deterministic nature of GC, a problem exacerbated by specificities of the interpreter and the platform in which the program is executed.

Therefore, we aim at performing static analysis on code to detect misuses of weak tables. In this paper we present the first steps towards that direction: a mathematical model of Lua's GC together with a prototype tool, **LuaSafe**, whose aim is to discover potential sources of non-determinism (at the moment, focusing only on GC). Our model builds on top of that from [23], and as such, it can be applied to the study of real Lua programs, missing only a handful of features from the language unrelated to GC.

The model is mechanized in PLT Redex [11] as an extension of the mechanization presented in [23]. It covers weak tables and *finalizers*,

PPDP '20, September 8–10,2020, Bologna, Italy

ACM ISBN 978-1-4503-8821-4/20/09.
https://doi.org/10.1145/3414080.3414093

E ::= [[ ]] | E (e , ...) | v (E_l) | var , ... = E_l
| **local** var , ... = E_l **in** s
| **if** E **then** s **else** s
| **setmetatable**(E_l) | **error** E | ...
E_l ::=v , ... , E , e , ...
s ::= e (e , ...) | var , ... = e , ...
| **local** var , ... = e , ... **in** s
| **while** e **do** s
| **if** e **then** s **else** s
| **setmetatable**(e , ...) | **error** e | ...
e ::= v | e (e , ...) | { [e] = e , ... }
| **function** (x , ...) s **end** | r | ...
v ::= *number* | *string* | *tid* | *cid* | ...
var ::= x | e [e]

Figure 2: Syntax of evaluation contexts, statements, expressions and values.

the latter being functions executed when an element is about to be disposed. We show that programs' behavior is deterministic, under GC without these interfaces. But as soon as finalizers or weak tables are considered, determinism is lost.

After understanding the intricacies of Lua's GC, we develop **LuaSafe**. This tool combines the knowledge about weak tables together with type inference and data-flow analysis in order to detect misuses of weak tables, that could lead to non-deterministic behavior. For instance, it rightfully rejects the program from Figure 1.

More concretely, our contributions are:

- A mathematical model of Lua's GC, including finalizers and weak tables.
- A theoretical framework under which we can express and prove standard soundness properties of our model.
- A formalization of said model in PLT Redex, where we execute programs from Lua's test suite to ensure compliance with Lua.
- A prototype tool, **LuaSafe**, to help uncover potential misuses of weak tables.

The mechanization of the model can be downloaded from [21], and **LuaSafe** can be downloaded from [20].

2 BASICS OF THE MODEL

In this section we introduce the necessary background to understand the model of GC that we will develop in the coming sections. As mentioned in the introduction, we build our model of GC on top of the semantics of Lua presented in [23], and we refer to the cited work for details.

2.1 A subset of Lua

Figure 2 shows an extract of the syntax of the model of Lua on which we are basing our studies (where, for a given non-terminal nt, the expression nt, ... denotes 0 or more repetitions of elements from nt). A Lua program is a statement (s), for instance a function call; multiple-variable assignment; definition of multiple local variables; the primitive **setmetatable** (discussed below); *error objects*, Lua's representation of errors; and several others. As expressions (e) we have values (v); function calls; *table* constructors; *function* definitions; and references (r) to the value store (to be explained below). Values are numbers, strings, table identifiers (*tid*) and closures identifiers (*cid*) (also introduced below).

To model imperative variables we include a mapping $r \rightarrow v$, the *values storage*, denoted with σ. Tables and closures are also manipulated by reference, although to ease the model we create two different sets of *identifiers* for them (*tid* and *cid*). These identifiers refer to tables and closures, respectively, through a new mapping $tid \cup cid \rightarrow table \cup function$, the *objects storage*, denoted with θ. We must point out a difference from the model in [23]: they do not consider references to closures, which in our model are required to faithfully record the cleaning of weak tables (§3.3).

Together with the given terms we include the corresponding *evaluation contexts* (E): terms with a special position marked by [[ ]], a *hole*. They can be used to formalize many context-dependent concepts, but the ones shown here indicate a call-by-value execution of programs, with a left-to-right ordering in the arguments of subexpressions. We will explain later how they are used to actually impose a particular order of execution.

The semantics given is operational and is formalized as a relation, which we will denote with $\overset{L}{\mapsto}$, over configurations of the form $\sigma : \theta : s$. For instance, the following rule formalizes a function call:

$$\frac{\theta(cid) = \textbf{function}\ (x_1, ..., x_n)\ s\ \textbf{end} \qquad \sigma' = (r_1, v_1), ..., (r_n, v_n), \sigma}{\sigma : \theta : cid\ (v_1, ..., v_n) \overset{L}{\mapsto} \sigma' : \theta : s[x_1 \backslash r_1, ..., x_n \backslash r_n]}$$

A function call essentially involves the allocation of its arguments into the values' store σ, with fresh references $r_1, ..., r_n$, and the substitution of the formal parameters of the function by these references, in the function's body ($s[x_1 \backslash r_1, ..., x_n \backslash r_n]$). Note that the closure is referred by its identifier.

The following rule models the fact that the execution of a statement might happen inside a larger program, modeled with the context E:

$$\frac{\sigma : \theta : s \overset{L}{\mapsto} \sigma' : \theta' : s'}{\sigma : \theta : E[\![s]\!] \overset{L}{\mapsto} \sigma' : \theta' : E[\![s']\!]}$$

The pattern from in the left of $\overset{L}{\mapsto}$ indicates that the program can be decomposed into an evaluation context E and a statement s. If the evaluation contexts and the execution rules are well defined, there should be just one way of decomposing any program into an E and an s, and s must be an execution-ready statement (for instance, the one presented above for function calls). The position of the term is determined by the hole of each evaluation context and, as can be seen in Figure 2, it is unique.

2.2 Metatables

Lua presents a powerful metaprogramming mechanism that allows for the modification of the behavior of some operations under unexpected circumstances, like arithmetic operations applied with non-numeric arguments; function calls over non-function values; indexing a table with a nonexistent key; *etc.* At the heart of this mechanism lies the concept of *metatable*, a regular table that maintains handlers to manage unexpected situations, associated with specific keys defined beforehand. For instance, in order to *explain* how a given table should be represented as a string, through the

service tostring, the developer can associate a conversion function with the key "__tostring" in the table's metatable. Some type of objects (tables and *userdata*) allows for the definition of a single metatable per value, while for the remaining there is just one metatable per type.

A table can be set a metatable through the **setmetatable** library service. In [23], tables are modeled as a pair containing the table's data and a table identity for the metatable, which can be **nil**. As an example, the following rule specifies the creation of a table:

$$\frac{tid \notin \mathrm{dom}(\theta_1) \qquad \theta_2 = (tid,\ (\{[\ v_1\ ]\ =\ v_2,\ ...\}\ , \mathbf{nil}),\ \theta_1}{\sigma : \theta_1 : \{[\ v_1\ ]\ =\ v_2,\ ...\} \overset{\mathrm{L}}{\mapsto} \sigma : \theta_2 : tid}$$

After creation, a table does not contain a metatable set. Only trough **setmetatable** one can associate a metatable to the given table.

As we will see in the coming section, metatables play an important role in the semantics of GC.

3 GARBAGE COLLECTION

This section represents our main contribution: an abstract model of Lua's GC, modularly divided in three parts. We start by modeling GC without interfaces (§3.1), laying the basic concepts upon which the interfaces are added: finalizers (§3.2), and weak tables (§3.3). We conclude with our mechanization of the model (§3.4).

3.1 Reachability-based garbage collection.

Lua implements two reachability-based GC strategies: a *mark-and-sweep* collector (the default) and a *generational collector*. The user is entitled to change the algorithm by calling the collectgarbage standard library's service. In this section we will provide a specification for the behavior of a typical reachability-based GC. It should encompass the essential details of the behavior of the two algorithms included in Lua and any other based on reachability. We start with a small set of definitions that we will enrich in coming sections.

Reachability. 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.*, 12]. 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.

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.*, 15].

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 [12, 16], 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 3.1 (Reachability for Simple GC). We say that a location $l \in r \cup tid \cup cid$ is *reachable* in term t, given stores σ and θ, iff:

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

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 ($\pi_1(\theta(tid))$) but also its metatable ($\pi_2(\theta(tid))$). 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 $\mathrm{reach}(l, \pi_2(\theta(tid)), \sigma, \theta \setminus tid)$, at first it will determine if l is exactly $\pi_2(\theta(tid))$ (because it asks for $l \in \pi_2(\theta(tid))$, for $\pi_2(\theta(tid))$ 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 $\pi_2(\theta(tid))$ from θ in the mentioned recursive call.

The last disjunct checks for reachability following a closure identifier *cid* present in the root set of references. We need to expand the reachability tree following the environment of the closure (*i.e.*, the mapping between the external variable's identifiers, present in the body of the closure, and their corresponding references).

We conclude this part on reachability with a minor observation: naturally, the reference manual leaves unspecified details of GC. For instance, it does not mention how metatables affect GC even though it does have an observable effect on programs. One of our major challenges and aim in this work is to unveil such interactions.

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 3.2 (Simple GC cycle).
$\mathrm{gc}(s, \sigma, \theta) = (\sigma_1, \theta_1)$, where:

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

We use $\gamma_1 \uplus \gamma_2$ to denote the union of stores with disjoint domains. This specification states that $\mathrm{gc}(s, \sigma, \theta)$ 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, through a relation $\overset{\mathrm{GC}}{\mapsto}$:

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

We require it to actually perform some change 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. We will define formally this concept in §4. This property will not longer be true when extending GC with finalizers and weak tables.

3.2 Finalizers.

Lua implements finalizers, a mechanism commonly present in programming languages with GC, useful for helping in the proper disposal of external resources used by the program. They are defined by the programmer as a function, which is called by the garbage collector after a value amenable for finalization (table or userdata) becomes garbage. It should be noted that, because finalizers are called by the garbage collector, there is no possibility of determining the precise moment in which finalization will occur. This in contrast with *destructors*, a concept present in languages with deterministic memory management algorithms (*e.g.*, as in C++).

There are several problems that arise from the misuse of this mechanism, associated with the fact that finalizers are called in a non-deterministic fashion, introducing that non-determinism into the execution of the program. Nonetheless, the implementation of finalizers in Lua provides some guarantees about the execution order of finalizers and the treatment given to resurrected objects which makes the algorithm an interesting case study.

```
1 local a, b = {}, {}
2 setmetatable(a, b)
3 b.__gc = function () print("bye") end
4 a = nil
5 collectgarbage()              --nothing is printed
6 local c = {}
7 setmetatable(c, b)
8 b.__gc = function () print("goodbye") end
9 c = nil
10 collectgarbage()       --now it outputs 'goodbye'
11 b.__gc = "not a function"
12 local d = {}
13 setmetatable(d, b)
14 d = nil
15 collectgarbage()                --nothing happens
```

Figure 3: Setting up a finalizer.

```
1 local a, b = {}, {}
2 local c = {__gc = function (o) print("bye", o) end}
3 print(a, b)            --table: 0x56..00  table: 0x56..40
4 setmetatable(a, c)
5 setmetatable(b, c)
6 a, b = nil, nil
7 collectgarbage()
8       --bye table: 0x56..40  (b)  bye  table: 0x56..00  (a)
```

Figure 4: Chronological order of execution of finalizers.

3.2.1 Overview of finalizers in Lua. We will begin with an informal presentation of the semantics of finalizers in Lua. After this, we will show how to extend the previous model of GC to include this interface with the garbage collector.

Setting up a finalizer. The finalizer of an object (table or userdata) is a function stored in the object's metatable, associated with the key "__gc". For *finalization* to occur (*i.e.*, the execution of the finalizer) the key must be defined the first time the corresponding metatable is set. In that case, it is said that the given object is *marked* for finalization. Later definitions of the __gc field will not be considered. The code shown in Figure 3 shows this behavior: when a is set an empty metatable (b in Line 2), even if later on __gc is defined (Line 3), when a is garbage collected (Line 5), no output is produced. But now that b has the __gc field defined, when it is set as a metatable of a new object (Line 7), this object is correctly marked for finalization (Line 10). Also, if the value set in the field __gc is not a function, GC will simply silently ignore the error (lines 11 to 15). As a last remark, the last finalizer set, assuming it is a function, is the one called when the object is disposed.

Execution order of finalizers. The execution order of finalizers is chronologically inverse to the time of the definition of the finalizers. This behavior is explained in Figure 4. This code performs the following steps: 1) creates two tables, a and b; 2) sets a metatable c to these objects containing a finalizer that prints the object being finalized, first for a and then for b; 3) eliminates any reference to a

and b; and 4) invokes the garbage collector. As you can see from the output (Line 8), the order in which the metatable is set affects the order in which the finalizers are called. While not shown in the code, if we swap lines 4 and 5, the result will also be swapped.

Resurrection. During finalization of a given object, its location is passed to the finalizer, turning the object reachable again. This phenomenon is commonly known as *resurrection*, and is normally transient. Then, there exist the possibility that the user code of the finalizer makes permanent the resurrection, by creating an external reference to the object, turning it reachable again even after finalization, preventing it from being collected.

This possibility introduces problems [7] into the implementation of garbage collectors, reduces their effectiveness to reclaim memory unused by the program and could reintroduce into the program objects that do not satisfy representation invariants.

To mitigate this issue, Lua treats finalized objects specially: it does not allow for a finalized object to be marked again for finalization. In that way, the finalizer of an object will never be called twice, avoiding indestructible objects. The object will be destroyed once it becomes unreachable again. This is the only difference of a finalized object: it is still possible to set a new metatable and to configure the resurrected objects' behavior using every metamethod *but* "__gc".

Error handling. During execution of a program, any error in a finalizer is propagated to the main thread of execution. Because finalizers are interleaved with user code, any error thrown from a finalizer appears in a position in the program that cannot be determined in advance. If that position happens to be inside a function that was called in *protected mode* —like a try in other languages— then the error is caught.

When a program ends normally, Lua executes each finalizer of the remaining objects in protected mode. In that circumstance, any error occurred during the execution of a given finalizer, interrupts only that finalizer, allowing for the call of the remaining finalizers. Also, a finalizer ended by an erroneous situation does not prevent the corresponding object from being disposed.

3.2.2 Modeling finalizers. We extend the model to include finalizers in two steps: first we update the internal representation for tables presented in §2 to add information about finalizers; then, we modify the GC model introduced in 3.1, to be aware of the finalization mechanism.

Representation of tables. We extend the tuple for representing a table with a third field, obtaining (table, metatable, pos). The new field pos has three different possible values: if it is $\bot$, it means that there is no finalizer set for the table; if it is $\oslash$, it means that the table cannot be set for finalization (to avoid multiple resurrections); and if it is a value p, of a set of values $\mathcal{P}$ ordered by a given order $<^{fin}$, it means the finalizer is set, with priority p, according to $<^{fin}$. Initially, pos will be $\bot$, as shown in the first rule of Figure 5. We present its semantics (and the remaining computation rules for finalization), with a new relation, $\overset{F}{\mapsto}$.

As mentioned, $<^{fin}$ is defined chronologically by the moment in which an object has been marked for finalization. For our semantics, it suffices to have a function next with signature $\mathcal{P} \to \mathcal{P}$, which should provide an element of $\mathcal{P}$ larger than its argument. We will also need $\bot \in \mathcal{P}$, and to be minimum with respect to $<^{fin}$. When a metatable is set with the corresponding call to **setmetatable** (second rule of Figure 5), we use a helper function set_fin to compute the corresponding value of pos.

$$\frac{\forall\ 1 \le \mathsf{i}, \mathit{field}_\mathsf{i} = v \vee \mathit{field}_\mathsf{i} = [\ v\ ] \ =\ v' \qquad \theta_2 = (\ \mathit{tid},\ (\ \text{addkeys}(\{\mathit{field}_1,\ \ldots\})\ ,\ \mathbf{nil},\ \bot\ ),\ \theta_1}{\theta_1\ :\ \{\mathit{field}_1,\ \ldots\} \overset{F}{\mapsto}\ \theta_2\ :\ \mathit{tid}}$$

$$\frac{\delta(\text{type}, v) \in \{\text{“table”}, \text{“nil”}\} \qquad \text{indexmetatable}(\mathit{tid}, \text{“__metatable”}, \theta_1) = \mathbf{nil} \qquad \theta_2 = \theta_1[\mathit{tid} := (\pi_1(\theta_1(\mathit{tid})), v, \text{set\_fin}(\mathit{tid}, v, \theta_1))]}{\theta_1 : \mathbf{setmetatable}(\mathit{tid}, v) \overset{F}{\mapsto}\ \theta_2 : \mathit{tid}}$$

Figure 5: Selected rules extended with finalization.

$$\text{set\_fin}(\mathit{tid}, v, \theta) = \oslash, \qquad \mathit{if}\ \pi_3(\theta(\mathit{tid})) = \oslash \tag{1}$$

$$\text{set\_fin}(\mathit{tid}, \mathbf{nil}, \theta) = \bot, \qquad \mathit{if}\ \pi_3(\theta(\mathit{tid})) \neq \oslash \tag{2}$$

$$\text{set\_fin}(\mathit{tid}, \mathit{tid}_\mathsf{m}, \theta) = \pi_3(\theta(\mathit{tid})), \mathit{if} \begin{cases} \pi_2(\theta(\mathit{tid})) = \mathit{tid}_\mathsf{m} \\ \pi_3(\theta(\mathit{tid})) \neq \oslash \end{cases} \tag{3}$$

$$\text{set\_fin}(\mathit{tid}, \mathit{tid}_\mathsf{m}, \theta) = \bot, \qquad \mathit{if} \begin{cases} \text{“__gc”} \notin \pi_1(\theta(\mathit{tid}_\mathsf{m})) \\ \pi_2(\theta(\mathit{tid})) \neq \mathit{tid}_\mathsf{m} \\ \pi_3(\theta(\mathit{tid})) \neq \oslash \end{cases} \tag{4}$$

$$\text{set\_fin}(\mathit{tid}, \mathit{tid}_\mathsf{m}, \theta) = \text{next}(p), \qquad \mathit{if} \begin{cases} \text{“__gc”} \in \pi_1(\theta(\mathit{tid}_\mathsf{m})) \\ \pi_2(\theta(\mathit{tid})) \neq \mathit{tid}_\mathsf{m} \\ \pi_3(\theta(\mathit{tid})) \neq \oslash \end{cases} \tag{5}$$

$$\text{where } p = \max^{<^{fin}}(\text{filter}(\text{map}(\pi_3, \text{img}(\theta)), \lambda\ \text{pos.pos} \neq \oslash))$$

Figure 6: Function set_fin for computing the pos field.

Figure 6 shows the set_fin function, which takes two tables (a table identifier *tid* and the proposed metatable) and a store θ. The metatable is another table identifier tid_m or **nil**, and returns the new pos value. The first equation shows the main use of the $\oslash$: no matter what is the value of the metatable, if the previous pos field of the table contains an $\oslash$, then it returns $\oslash$ to ensure no finalization can happen again on *tid*. The second equation specifies one of the situations when a given table is unmarked for finalization: if the metatable is **nil**, and the previous value of pos is not $\oslash$, then it returns $\bot$. The third equation considers the case when the same metatable is set, in which case the pos field remain unchanged (we use the bracket to mean that every condition must apply). The fourth equation considers the case when the metatable does not contain the "__gc" metamethod: it is unmarked for finalization ($\bot$). In the last equation set_fin returns the next value of the maximum of every pos in θ, if the metatable contains the metamethod "__gc".

Specification of GC with finalization. We enrich the specification of GC from §3.1 to make it aware of finalization (figures 7 and 8). The new predicate, gc_fin, returns two stores σ_1 and θ_1', and a term t, the finalizer to be executed if appropriate. The first part of the predicate (*gc*) replicates the gc predicate from §3.1, and states that we can split the stores into two disjoint parts, the ones to be discarded (σ_2 and θ_2) and the rest (σ_1 and θ_1). But now the partitions have additional conditions (*fin*): first, every discarded table *tid* in θ_2 must not be

$$
\begin{array}{l}
\mathsf{gc}_{\mathsf{fin}}(s,\sigma,\theta) = (\sigma_1, \theta_1', t), \textit{where} \\
gc \left\{ \begin{array}{l}
\sigma = \sigma_1 \uplus \sigma_2 \\
\theta = \theta_1 \uplus \theta_2 \\
\forall \mathsf{l} \in \mathsf{dom}(\sigma_2) \cup \mathsf{dom}(\theta_2), \neg\mathsf{reach}(\mathsf{l}, s, \sigma, \theta)
\end{array} \right. \\
fin \left\{ \begin{array}{l}
\forall tid \in \mathsf{dom}(\theta_2), \\
\quad \neg\mathsf{marked}(tid, \theta_2) \\
\forall l \in \mathsf{dom}(\sigma_2) \cup \mathsf{dom}(\theta_2), \\
\quad \mathsf{not\_reach\_fin}(l, \sigma_1, \theta_1) \\
[\exists tid \in \mathsf{dom}(\theta_1), \\
\quad \mathsf{fin}(tid, s, \sigma, \theta) \\
\quad \mathsf{next\_fin}(tid, s, \sigma, \theta) \\
\quad v = \mathsf{indexmetatable}(tid, \text{“\_\_gc”}, \theta_1) \\
\quad v \in cid \Rightarrow t = v(tid) \\
\quad v \notin cid \Rightarrow t = \textbf{nil} \\
\quad \theta_1' = \theta_1[tid := (\pi_1(\theta_1(tid)), \pi_2(\theta_1(tid)), \oslash)] \\
\mathsf{or}: \\
\quad t = \textbf{nil} \\
\quad \theta_1' = \theta_1]
\end{array} \right.
\end{array}
$$

Figure 7: GC cycle with finalization.

$$
\begin{array}{l}
\mathsf{marked}(tid, \theta) \doteq \pi_3(\theta(tid)) \notin \{\bot, \oslash\} \\
\mathsf{not\_reach\_fin}(l, \sigma, \theta) \doteq \nexists\, tid \in \mathsf{dom}(\theta), l \neq tid \wedge \\
\qquad\qquad \mathsf{marked}(tid, \theta) \wedge \mathsf{reach}(l, tid, \sigma, \theta) \\
\mathsf{fin}(tid, s, \sigma, \theta) \doteq \neg\mathsf{reach}(tid, s, \sigma, \theta) \wedge \mathsf{marked}(tid, \theta) \\
\mathsf{next\_fin}(tid, s, \sigma, \theta) \doteq \forall tid' \in \mathsf{dom}(\theta), \\
\qquad\qquad \mathsf{fin}(tid', s, \sigma, \theta) \Rightarrow \pi_3(\theta(tid')) \leq^{fin} \pi_3(\theta(tid))
\end{array}
$$

Figure 8: Predicates for finalization.

marked for finalization, otherwise we will lose a call to a finalizer. Second, we ask that every location from the removed stores is not reachable from the stores that are kept (σ_1 and θ_1).

The previous conditions ensure that θ_2 only contain tables already finalized or not set for finalization, and avoids potential dangling pointer errors when executing a finalizer. The following conditions characterize the next table to be finalized. If there exists a *tid* in θ_1 such that it is finalizable and the next in the order $\leq^{fin}$ (as expressed by the predicates fin and next_fin), and has a proper finalizer set (a function *v* in its "__gc" field), then the next statement to be executed is *v* applied to the table identifier (transiently resurrecting the table), and the new table store θ_1' is the same as θ_1, except that *tid* is forbidden to be marked again for finalization (by setting its pos field to $\oslash$), therefore avoiding more than one resurrection of the table. Note that *tid* is still in the returned θ_1', otherwise it could not be made accessible to the finalizer. In our model, the table is actually collected in another GC cycle, as we cannot know before hand if it will be resurrected or not by its finalizer.

In case there is no table with a valid finalizer, then *t* is **nil** and θ_1' is just θ_1.

Interleaving finalization with the user program. From the definition of $\mathsf{gc}_{\mathsf{fin}}$ given above, it is clear that a single GC cycle encompasses collection of garbage together with at most one call to a finalizer. The reasons are two-fold: first, the small-step fashion of our semantics, and the interleaved execution of finalizers with the user's program. However, this does not prevent the execution of more than one finalizer before the execution of the next user program's instruction, given the non-determinism of the execution rules for GC.

$$
\frac{(\sigma', \theta', v(tid)) = \mathsf{gc}(\sigma, \theta, \mathsf{E}[\![\mathsf{s}]\!])}{\sigma : \theta : \mathsf{E}[\![\mathsf{s}]\!] \overset{\mathsf{F}}{\mapsto} \sigma' : \theta' : \mathsf{E}[\![\ v(tid); \mathsf{s}]\!]}
$$

$$
\frac{(\sigma', \theta', v(tid)) = \mathsf{gc}(\sigma, \theta, \mathsf{E}[\![\mathsf{e}]\!])}{\sigma : \theta : \mathsf{E}[\![\mathsf{e}]\!] \overset{\mathsf{F}}{\mapsto} \sigma' : \theta' : \mathsf{E}[\![\ \textbf{function}\ \$\ ()\ \textbf{return}\ \mathsf{e}\ \textbf{end}\ (v(tid))]\!]}
$$

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

Figure 9: Interleaving the execution of finalizers with the program.

What remains to specify is how finalization is actually interleaved with the user program. This is stated by the rules in Figure 9. We allow for the possibility of interleaving the finalization step with any statement or expression to be executed. The first case can be expressed directly, as shown in the first rule. Interleaving it with an expression, shown in the second rule, requires some more work, since we cannot express directly the concatenation of expressions. In that case, we reduce the desired execution order of expressions to the one defined for function call.

Finally, if no finalizer is chosen (third rule), as before, we ask for some of the stores returned to be modified in order disallow infinite sequences of GC steps.

3.3 Weak tables

A *weak table* is a table whose keys and/or values are referred by *weak references*: references which are not taken into account by the garbage collector when determining reachability. In Lua, among the types included into our model, only tables and closures can be garbage collected from weak tables, the general rule being that *"only objects that have an explicit construction are removed from weak tables"* [§2.5.2 of 4].

In order to specify a table's weakness, the user adds in the table's metatable the key "__mode" with a string value containing the characters 'k' (for keys to be referred by weak references) and/or 'v' (for values to be referred by weak references) .

Introducing weak tables into the model. To model weak tables we do not introduce weak references explicitly. Instead, we modify the criterion used to determine the reachability of a given reference to consider its occurrences on weak tables, according to the tables' weakness. Key to the new definition of GC cycle is a new predicate reachCte that allows us to consider the reachability of a *collectible table element* (*cte*), which is an element of the set with the same name formed from the union of table and closures identifiers.

Reachability of a cte. We distinguish two situations with respect to the reachability of a *cte*: either there is a path from the root set of references to the value itself using just *strong* references (non-weak references), or every path to the value from the root set contains at least one weak reference. In the first case the value will not be garbage collected, and we refer to such value as *strongly reachable*. In the second case the value can be GC.

$$\mathrm{SO}(tid,\theta) = \begin{cases} \{k_i | k_i \in (\{k_1,\ldots\} \cap cte)\} & if\ \ \mathrm{wv?}(tid,\theta) \\ & \quad \wedge \neg \mathrm{wk?}(tid,\theta) \\ \{v | v \in \{k_1, v_1, \ldots\} \cap cte\} & if\ \ \neg(\mathrm{wv?}(tid,\theta) \\ & \quad \vee \mathrm{wk?}(tid,\theta)) \\ \{(k_i, v_i) | v_i \in \{v_1,\ldots\} \cap cte\} & if\ \ \neg \mathrm{wv?}(tid,\theta) \\ & \quad \wedge \mathrm{wk?}(tid,\theta) \\ \emptyset & otherwise \end{cases}$$

where $\pi_1(\theta(tid)) = \{[k_1] = v_1, \ldots\}$

Figure 10: Strong occurrences of a table.

$$\mathrm{eph}(id,(k,v),tid,rt,\sigma,\theta) = \mathrm{reachCte}(id,v,\sigma,\theta,rt) \wedge [k \notin cte \vee \mathrm{reachCte}(k,rt,\sigma,\theta|_{tid|_k},rt)]$$

$$\begin{aligned} \mathrm{reachTable}(id,tid,\sigma,\theta,rt) = {} & [\exists\,(k,v) \in \mathrm{SO}(tid,\theta), \mathrm{eph}(id,(k,v),tid,rt,\sigma,\theta)] \vee \\ & [\exists\, v \in \mathrm{SO}(tid,\theta), \mathrm{reachCte}(id,v,\sigma,\theta,rt)] \vee \\ & \mathrm{reachCte}(id,\pi_2(\theta(tid)),\sigma,\theta,rt) \end{aligned}$$

$$\begin{aligned} \mathrm{reachCte}(id,t,\sigma,\theta,rt) = {} & id \in t \vee \\ & \exists\, r \in t,\ \mathrm{reachCte}(id,\sigma(r),\sigma,\theta,rt) \vee \\ & \exists\, tid \in t, \mathrm{reachTable}(id,tid,\sigma,\theta,rt) \vee \\ & \exists\, cid \in t, \mathrm{reachCte}(id,\theta(cid),\sigma,\theta,rt) \end{aligned}$$

Figure 11: Reachability of a collectible element.

In order to distinguish these cases, we define what are a table's *strong occurrences* (Figure 10): the keys and/or values of a table (limited to *ctes*) that are not referred by weak references. If a given table has weak values then just its keys' occurrences are considered strong (predicates wk? and wv?, elided for brevity, allow us to know the weakness of a given table). The second and fourth cases can be explained on the same basis. The third case, weak keys and strong values, has to do with what is known as an *ephemeron* table, which is treated in a special way by the garbage collector, in order to avoid the problems that arise with cycles into a weak table (*e.g.*, values referring to their own keys), which could prevent them from proper GC, or between weak tables with this level of weakness, which could delay GC (see [8] for an analysis of the problem from Lua's perspective). In an ephemeron table, an occurrence of a value from *cte* as the value of a table field is considered strong just if its associated key is still strongly reachable. Because this is not a property that can be determined locally, by just looking at the table being inspected, we return each key-value pair.

Before presenting the predicate reachCte, we introduce the predicate reachTable (shown in Figure 11), which expands the reachability tree from a table *tid*, when determining reachability of an identifier *id*. Given the semantics of ephemerons (explained later), we need to keep track of the original program *rt*, from which the original root set is computed.

The first disjunt of reachTable checks for reachability following references from the table *tid*, when it happens to be an ephemeron (*i.e.*, weak keys and strong values), as specified by the predicate eph. This predicate says that *id* is reachable from the value of a field (k, v) of an ephemeron table *tid* iff it is strongly reachable from v, according to reachCte, and the key k, either, cannot be GC, *i.e.*, it is not a member of *cte*, or it is reachable (from the root set of references in *rt*). That is, the reachability of the value is affected by the reachability of its key. In doing so, we must not take into account v. This is related with the recolection of fields from an ephemeron: it allows for the removal of a field where the only reference to the key comes from the value. We use the notation $\theta|_{tid|_k}$ to denote the store resulting from removing the field with key k from the table *tid*.

Returning to reachTable, if the table is not an ephemeron (second disjunt), we just need to consider each strong occurrence of a *cte* present into the table, as defined by SO. Finally, for any table found during the expansion of the reachability tree, we also need to look into its metatable, as it was the case when defining the predicate reach, in §3.1.

We now turn to the definition of reachCte (also in Figure 11), which will have almost the same signature as reach, except for the addition of the term from which the root set of references is determined for the case of ephemeron tables. As an aside, while it is possible to give a primitive or well-founded recursive definition, it would require cumbersome expressions for the recursive calls over stores of decreasing size. Instead, we followed [16] and defined the desired predicate as the least fixed point that satisfies the previous equation.

The predicate is defined assuming that the mere occurrence of a *cte* into t implies that such value is strongly reachable. Recursive cases are defined such that they maintain this property of t. The second disjunct dereferences references to values found into the term t. Next we expand the reachability tree by following tables, as expressed by reachTable. The last disjunct checks into the environment of the closures found during expansion, as in Definition 3.1.

GC cycle. Note that by enriching the notion of reachability with weak references, it could be possible for the garbage collector to remove the binding of a table or closure identifier which is not strongly reachable but that is still present into a reachable weak table. This, of course, would generate dangling pointers if the program tries to dereference such identifiers through the weak table.

If we forget about finalizers, we avoid such problems by simply replacing the predicate reach in Definition 3.2 of gc with the new predicate reachCte. However, when considering finalizer, special care must be taken. We therefore introduce a new predicate $\mathrm{gc}_{\mathrm{fin\_weak}}$ (Figure 12), which is based on a modified $\mathrm{gc}_{\mathrm{fin}}$ predicate. In concrete, the new new predicate $\mathrm{gc}'_{\mathrm{fin}}$ is a verbatim copy of $\mathrm{gc}_{\mathrm{fin}}$ but with the following changes:

(1) We replace reach with reachCte in the fin predicate.
(2) We prevent for finalization to occur on a table that is also present as a weak value in some weak table, by adding the following predicate:

$$\begin{aligned} \mathrm{not\_fin\_val}(tid,\theta) \doteq {} & \nexists\ tid' \in \mathrm{dom}(\theta)\ / \\ & \mathrm{wv?}(tid',\theta)\ \wedge (k,tid) \in \pi_1(\theta(tid')), \\ & \text{for some key } k \end{aligned}$$

The predicate helps us to specify the behavior of resurrected objects in weak tables, for the particular case of weak values.[1]
(3) We remove the *fin* portion of the predicate to let the new $\mathrm{gc}_{\mathrm{fin\_weak}}$ predicate take care of marking the table with $\oslash$.

Essentially, after obtaining a new θ'_1 from $\mathrm{gc}'_{\mathrm{fin}}$, the returned object store θ'''_1 might have a few discrepancies from that of θ'_1,

[1] See [19], section 2.5.2.

$\mathrm{gc}_{\mathsf{fin\_weak}}(s,\sigma,\theta) = (\sigma_1', \theta_1''', s')$, where $(\sigma_1', \theta_1', s') = \mathrm{gc}'_{\mathsf{fin}}(s,\sigma,\theta)$, and:

$$
wt \begin{cases}
\exists \theta_1'', \mathrm{dom}(\theta_1'') = \mathrm{dom}(\theta_1') \\
\forall tid \in \mathrm{dom}(\theta_1''), \theta_1''(tid) = \theta_1'(tid) \vee \\
\quad [\pi_1(\theta_1''(tid)) \subset \pi_1(\theta_1'(tid)) \wedge \\
\quad \exists (\mathsf{k}, \mathsf{v}) \in \pi_1(\theta_1'(\mathsf{tid})), (\mathsf{k}, \mathsf{v}) \notin \pi_1(\theta_1''(tid))/ \\
\quad reach \begin{cases} \mathsf{wk?}(tid,\theta) \wedge k \in cte \wedge \neg \mathrm{reachCte}(\mathsf{k}, s, \sigma, \theta, s) \\ \vee \\ \mathsf{wv?}(tid,\theta) \wedge v \in cte \wedge \neg \mathrm{reachCte}(\mathsf{v}, s, \sigma, \theta, s) \end{cases} \\
\quad fin\ key \begin{cases} \mathsf{wk?}(tid,\theta) \Rightarrow \\ \nexists\ tid', \mathrm{marked}(tid', \theta) \wedge \mathrm{reachCte}(\mathsf{k}, tid', \sigma, \theta, s) \end{cases} \\
\quad rem \begin{cases} \pi_2(\theta_1''(tid)) = \pi_2(\theta_1'(tid)) \\ \pi_3(\theta_1''(tid)) = \pi_3(\theta_1'(tid)) \quad] \end{cases}
\end{cases}
$$

$$
fin \begin{cases}
\theta_1''' = \theta_1''[tid := (\pi_1(\theta_1''(tid)), \pi_2(\theta_1''(tid)), \oslash)], \textit{if } s' = v(tid)) \\
\quad \vee \\
\theta_1''' = \theta_1'', \textit{ if } s' = \mathbf{nil}
\end{cases}
$$

Figure 12: GC cycle extended with weak tables.

since GC may remove fields of tables when their keys or values are not strongly reachable.

More concretely, θ_1''' is the store obtained from updating θ_1'' after marking with $\oslash$ the table being finalized, if applicable (*fin*). And θ_1'' is obtained from θ_1' after noting that they have the same domain (table ids), and for every table *tid*, they either have the same definition or the table in θ_1' has a field (k, v) that is not present in θ_1''' and:

reach: The field has a not strongly reachable key or value, depending on the table weakness. Note that we pass s as the last argument of reachCte, to preserve it as the root set of references from which any new expansion of the reachability tree must begin.

fin key: In the case of resurrected weak keys (*i.e.*, weak keys being finalized or accessible only through objects being finalized), they are removed only after they are finalized, or the resurrected object which can reach this key is finalized. This restriction allows for a finalizer of a weak key to access any information associated with that key.

rem: The remaining components of the internal representation of tables are not altered.

Finally, there is no need for the redefinition of the GC step: the details of GC of weak tables are all abstracted into the $\mathrm{gc}_{\mathsf{fin\_weak}}$ metafunction, and its interference with the execution of the program does not differ from what regular GC does.

3.4 Mechanization

The formalization presented in this section is mechanized in PLT Redex, and can be downloaded from [21]. Since it includes a large portion of Lua already covered in [23], it is possible to execute Lua programs and observe their effect on the memory. Our development offers an interface to execute snippets of code to their completion, optionally obtaining a visualization of the trace of the program. The execution may include non-deterministic steps of GC and/or direct calls to the collectgarbage service.

```
1 (execute "
2   print('weak tables')
3   lim = 15
4   a = {}; setmetatable(a, {__mode = 'k'});
5   -- fill a with some collectable ' indices
6   for i=1,lim do a[{}] = i end
7   -- and some non-collectable ones
8   for i=1,lim do a[i] = i end
9   for i=1,lim do
10    local s=string.rep('@', i); a[s] = s..'#' end
11  collectgarbage()
12  local i = 0
13  for k,v in pairs(a) do
14    assert(k==v or k..'#'==v); i=i+1 end
15  assert(i == 2*lim)"
16 ( list "print" "setmetatable" "string" " string.rep "
17 " collectgarbage " "pairs" "assert "))
```

Figure 13: Executing a program from the test suite of Lua.

For instance, Figure 13 shows how to execute a snippet of code taken from Lua's test suite using the deterministic semantics relation in which collecgarbage must be called explicitly. The code is taken from gc.lua as is, except that the global variable lim, declared in Line 3, which in gc.lua is defined earlier. The execution ends successfully, *i.e.*, passing the assertions in lines 14 and 15. The final configuration includes the garbage generated in the last loop, which can be removed adding a last call to collectgarbage at the end.

A point worth to mention is that in order to execute the GC in our mechanization, we must naturally implement an actual algorithm. We decided for a simple *stop-the-world* algorithm. Since real programming languages (like Lua) tend to approximate the root set of references by using just the variables which are in scope (the environment) —regardless of their presence in what remains to be computed— we artificially keep literal occurrences of the variables in scope. We do this by embedding the environment into the program. This implies that in the example from Figure 13, a last call to collectgarbage will return a configuration where the σ store contains the bindings from such variables.

At the moment we can successfully run 125 LOCs out of the 445 included in gc.lua. The remaining ones include features not present in our model (mainly parameters of the collectgarbage service), or that simply takes too long to execute. For the coverage of the rest of the test suite we refer the reader to [23], as the inclusion of GC did not improve the coverage within those files.

4 PROPERTIES OF GC

In this section we present the formal framework used to study properties of our specification of GC. We conclude this section with an important theorem about Correctness of GC, and in the way we provide the necessary tools required to discuss about non-deterministic computations; which form the foundation stone of **LuaSafe** (§5).

4.1 Result of a program

We start by defining the notion of *result* of a Lua program. Essentially, it consists of the term from the last configuration of its

$$\text{result}(\sigma : \theta : E\ [\![\textbf{return } v_1, ..., v_n]\!]) = \sigma|_S : \theta|_T : \textbf{return } v_1, ..., v_n$$
$$\textit{where} \begin{cases} E \text{ does not contain a return point} \\ S = \bigcup_{r \in \text{dom}(\sigma), R(r)} r \\ T = \bigcup_{id \in \text{dom}(\theta), R(id)} id \\ R(i) = \text{reach}(i, \textbf{return } v_1, ..., v_n, \sigma, \theta) \end{cases}$$
$$\text{result}(\sigma : \theta : \textbf{\$err } v) = \sigma|_S : \theta|_T : \textbf{\$err } v$$
where S *and* T *are defined as before, but with* $R(i) = \text{reach}(i, \textbf{\$err } v, \sigma, \theta)$
$$\text{result}(\sigma : \theta : ;) = \emptyset : \emptyset : ;$$

Figure 14: Result of a program and associated functions.

convergent computation, together with the information from the stores needed to give meaning to the term's free variables. That is, we strip off from the stores any information irrelevant to the final computation of the program.

To capture the previous idea we use a function, result (Figure 14), that given a final configuration of a program it extracts the required information from the stores to explain the result represented by said configuration. In order to understand the different cases considered by the function, we must state a standard corollary of the progress property for our semantics, which explains the expected final configurations for $\overset{L}{\mapsto}$ (that is, Lua without GC):

Assumption 4.1 (Corollary of progress). *For every well formed configuration $\sigma : \theta : s$, just one of the following situations hold:*

- *The execution diverges, denoted $\sigma : \theta : s \Uparrow$.*
- *The execution ends with an error **error** v, and stores σ' and θ', denoted $\sigma : \theta : s \Downarrow \sigma' : \theta' : \textbf{error } v$.*
- *The execution ends normally, with stores σ' and θ', and some values v, ... are returned:*
  $\sigma : \theta : s \Downarrow \sigma' : \theta' : E\ [\![\textbf{return } v, ...]\!]$*, where E does not contain the point to which the **return** statement must jump.*
- *The execution ends normally, with stores σ' and θ', and no value is returned:*
  $\sigma : \theta : s \Downarrow \sigma' : \theta' : ;$

Where it corresponds, the resulting configuration is also well formed.

The condition expressed for the evaluation context E, in the case of a computation that ends in $E\ [\![\textbf{return } v, ...]\!]$, implies that the **return** statements occurs outside of a function: it is the result returned by the program, which will be received, for example, in the host application where the Lua program is embedded.

We omit the notion of well-formedness, as it is standard: it rules out not just ill-formed programs, but also ill-formed terms that represent intermediate computations. It express, mainly, restrictions that cannot be captured by our context-free grammar.

Coming back to the function result, it considers each possible final configuration, keeping only the bindings from the stores that are needed to completely describe the result. It uses the function reach from §3.1. In the case of a **return**, it strips out the context E. Though simple, in the context of syntactic GCs such notion of result is not be sensible to different syntactic GC strategies, or even to the complete absence of GC.

Computing the result of a program allows us to compare different runs from the same program. We assume that there exists an α-conversion between locations from σ and θ, even when real programming languages often provide several library services that may break α-conversion. For example, in Lua it is possible to convert a table id to a string using the library service tostring. Naturally, if we include this service, we would be able to write programs whose returned values will depend upon obscure details of memory management, and that will be beyond formal treatment for the purpose of comparison of results. Thus, we assume that the semantics of $\overset{L}{\mapsto}$ is deterministic, which basically boils down to:

Assumption 4.2 (Restrictions to the model). *We assume none of the following is permitted:*

1. *Services that expose the details of memory management (*e.g.*, the* tostring *service).*
2. *Services that expose external variables (*e.g.*, the time, the file system, random numbers,* etc.*).*
3. Userdata *values, part of Lua's interface with a host C program. Naturally, since we have no control over the host, we cannot make claims about its determinism.*

The first assumption can be lifted off if we assume that the memory manager is deterministic and new references are always created fresh, *i.e.*, there is no re-use of memory locations.

4.2 Observations

The standard sanity check of our specification of GC (without interfaces to the garbage collector), consists in showing that the addition of a step of GC does not change the semantics of the running program. In the context of our dynamic semantics we capture this idea with a notion of *observations* over programs.

We parameterize the definition over a relation $\rightarrow$ that formalizes execution steps. For our studies, $\rightarrow$ will be $\overset{L}{\mapsto}$ (*i.e.*, our original model of Lua's operational semantics) with or without GC steps. We will reuse the notation introduced in Corollary 4.1 to speak about the convergence of computations, but now we will subscript with $\rightarrow$, to indicate that we are computing using only the execution rules from $\rightarrow$. For brevity, we will use C for a variable that ranges over the set of configurations.

Definition 4.1 (Observations). For a given well-formed configuration C, and execution rules $\rightarrow$:

$$\text{obs}(C, \rightarrow) = \{\bot \mid C \Uparrow_{\rightarrow}\} \cup \{\text{result}(C') \mid C \Downarrow_{\rightarrow} C'\}$$

The previous definition hinges on the fact that a progress property holds for $\rightarrow$: if result is defined over the last configuration of a convergent computation, this configuration must be a valid final configuration. While this is true for $\rightarrow = \overset{L}{\mapsto}$, we have not provided evidence that this is also the case after the addition of GC and its interfaces. Later, in §4.4, we will argue that by including $\overset{GC}{\mapsto}$ to $\overset{L}{\mapsto}$ we are not introducing stuck states.

Observations are useful to describe program equivalence:

Definition 4.2 (Program equivalence).

$$(C, \rightarrow) \equiv (C', \rightarrow') \Leftrightarrow \text{obs}(C, \rightarrow) = \text{obs}(C', \rightarrow')$$

4.3 Garbage

With the definitions developed so far we can now formalize a notion of garbage as a binding (a pair reference-value) that can be removed without changing the meaning of a program:

Definition 4.3 (Garbage). For a given well-formed configuration $\sigma \uplus \{(r, v)\} : \theta : s$, operational semantics $\rightarrow$, the binding (r, v) is *garbage* with respect to $\rightarrow$, iff:

$$(\sigma \uplus \{(r, v)\} : \theta : s, \rightarrow) \equiv (\sigma : \theta : s, \rightarrow)$$

A binding from θ is defined as garbage in an analogous manner.

The concepts introduced so far will allow us to define and study a notion of correctness of GC in the absence of its interfaces, but also could be of use for future studies of properties and applications of the model for weak tables and finalization.

4.4 Correctness of $\overset{GC}{\mapsto}$.

With the previously defined notions we can tackle the study of several desirable properties of GC. For GC without interfaces we can perform its standard sanity check, *i.e.*, to prove its soundness property: the addition of a GC step does not change the semantics of a program. Informally, it consists in showing that by adding $\overset{GC}{\mapsto}$ the observations over a given program are not altered. The desired statement is captured in the following statement (where $\overset{L+GC}{\mapsto} = \overset{L}{\mapsto} \cup \overset{GC}{\mapsto}$):

Theorem 4.4 (GC correctness). *For a given well-formed configuration $\sigma : \theta : s$,*

$$(\sigma : \theta : s, \overset{L}{\mapsto}) \equiv (\sigma : \theta : s, \overset{L+GC}{\mapsto})$$

The proof is included in the appendix [22].

We obtain as corollary that $\overset{L+GC}{\mapsto}$ is deterministic:

Corollary 4.5 (Determinism of GC). *For a well-formed configuration $\sigma : \theta : s$, $|\text{obs}(\sigma : \theta : s, \overset{L+GC}{\mapsto})| = 1$*

Naturally, after the introduction of weak tables or finalizers, programs may no longer exhibit deterministic behavior, hence the requirement of a set of observations in order to be able to express the possible outcomes of a Lua program under $\mapsto$, the complete dynamic semantics:

Theorem 4.6 (Non-deterministic behavior). *For some well-formed configuration $\sigma : \theta : s$,*

$$(\sigma : \theta : s, \overset{L+GC}{\mapsto}) \not\equiv (\sigma : \theta : s, \mapsto)$$

Proof. Consider the program presented in §1 (Figure 1). □

As a first attempt in recovering the deterministic behavior of programs that make use of weak tables, in the next section we introduce **LuaSafe**.

5 LUASAFE: ENSURING GC-SAFENESS

As the code in Figure 1 shows, a program using weak tables could exhibit non-deterministic behavior. Nonetheless, given the usefulness of weak tables to easily implement several data-structures (*e.g.*, caches, weak sets, property tables) [8], it is important to understand their semantics, and to have tools to prevent common pitfalls in their use. In the first part of the present paper we aimed at the former, and now we turn our attention into what constitutes the first steps into the latter.

More concretely, in this section we introduce **LuaSafe**, a prototype static analyzer that aims at the detection of misuses of weak tables, that could lead to non-deterministic behavior. We are mostly concerned with access to fields of weak tables that are not strongly reachable. While the general problem is known to be undecidable [15], 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, being $\mapsto$ the dynamic semantics that includes $\overset{L}{\mapsto}$ and GC with interfaces, if $\text{obs}(\emptyset : \emptyset : p, \mapsto)$ is a singleton we say that p is *gc-safe*, and 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 [15].

As this is the first step taken in implementing **LuaSafe**, we assume some restrictions—that we mention where relevant—on the Lua programs under consideration. We expect in the future to increase the analysis power of the tool.

Overview. Figure 15 shows the design of **LuaSafe**. As a first step, we take a user program p, an we infer the type of its local variables and function definitions. In particular, at this point we recognize if the evaluation of a given expression involves the access to a field of a weak table, and to determine the kind of information such field contains. This is important to understand if the result of such evaluation could be unpredictable. The result of type inference is an annotated program p_{typed}.

The next step consists in the extraction of information useful to determine if the references in a given expression are reachable from the root set of references. To compute the root set at some point of the program, we use a syntactic approximation consisting of the set of definitions of variables which are valid at that point. That is, we solve the problem of *reachable definitions* [1] for p_{typed} by constructing its *control flow graph* (cfg), annotating each expression and statement of the program with the set of definitions that are valid at that particular point, obtaining cfg_{rch_def}.

The last step consists in taking p_{typed} and cfg_{rch_def}, and performing type checking over p_{typed}. In that way, we are able to reconstruct the type of complex expressions, and to recognize whether the evaluation of a given expression involves the access to a field of a weak table. If it is the case, we will query cfg_{rch_def} for the set of valid variable definitions at the corresponding point of the program and determine the reachability of the corresponding table field, following the semantics of weak tables from §3.3.

In the reminder of this section we explain the different steps of **LuaSafe** and present examples showing its potential.

5.1 Type system

Common to all the steps of **LuaSafe** lies the type system. Figure 16 shows the language extended with type annotations for local variables and function definitions. As for the types, we have primitive types, *prmt*, where we include the **nil** type, **num**bers, **bool**eans and **str**ings. Then, we have *singleton types st*, which *lift* to the level

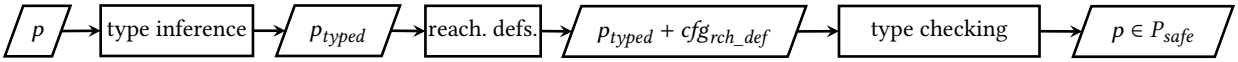

Figure 15: Design of LuaSafe.

Typed language

s ::= ... | **local** x : t , ... = e , ... **in** s **end**
e ::= ... | t **function** (x : t , ...) s **end**

Types

t ::= *prmt* | *st* | **dyn** | $t \rightarrow t$ | { [*st*] : t ... } *wkness* | $\mu\ y\ .\ t$ | $t \times t$ | **()**
prmt ::= **nil** | **num** | **bool** | **str**
st ::= $\langle v_{\text{st}} : prmt \rangle$
v_{st} ::= **nil** | *string* | *boolean* | *number*
wkness ::= **strong** | **wk** | **wv** | **wkv**

Figure 16: Grammar for typed terms.

of types a literal value v_{st} (**nil**, a number, a string, or a boolean). Singleton types serve two purposes in our work: to allow us to statically know which field of a table is being indexed, and therefore to know if the access is valid or not; and to track the changes in the weakness of tables at each call to **setmetatable**.

Besides primitive and singleton types we have the **dyn** supertype for variables whose type cannot be properly inferred statically; function types, $t \rightarrow t$; table types, { [*st*] : t ... } *wkness*, which include a tag (*wkness*) indicating the weakness of the table, and are restricted to be indexed by singleton types; recursive types, $\mu y.\ t$, to better support common programming idioms using tables; and product types, $t \times t$ and **()** (the empty tuple), which we use to express the domains of functions, though they have many other roles in typing Lua programs (see, for example, [17]).

Types are ordered by a typical subtyping relation <:, except for minor simplifications: **dyn** is the supertype of every type; every primitive type p is the supertype of any $\langle v : p \rangle$; subtyping for function and recursive types will be reduced to reflexivity, for purposes of simplification of type inference; table types are related by width, depth and permutation subtyping; and product types are covariant. As an important remark, we do not take a tables' weakness into account for subtyping in order to let the weakness of a table to change through a given program.

5.2 Type inference

Our type inference algorithm is based mostly on ideas introduced in [2], where it is presented a type inference algorithm for a language that includes some features of JavaScript. For reasons of brevity we will not cover its details. We refer the reader to the cited work and our mechanization with PLT Redex.

Informally, the essence of the process consists in traversing the AST of a given program, generating constraints over the type that we should assign to each expression. These constraints are generated observing the way in which expressions are used in the program. For our purposes constraints relates types of terms, according to our subtyping relation, and restricts the fields that a given table type should have.

The solution proposed in [2], which we follow, works in steps. First, for each expression, a new type variable is constructed; then, these variables are constrained. Once a set of constraints *Cs* is

$$\frac{\Gamma_1, cfg_{rch\_def}, C[\![\ [\![\ ]\!][e_2]\ ]\!] \vdash_{te} e_1 : \Gamma_2 : \{[st_1] : t_1, \ldots\}\ \textbf{strong} \quad \Gamma_2, cfg_{rch\_def}, C[\![\ e_1[\![\ ]\!]\ ]\!] \vdash_{te} e_2 : \Gamma_3 : st_2 \quad \vdash_{mtch} \{[st_1] : t_1, \ldots\} \approx \{\ldots, [st_2] : t_2, \ldots\}}{\Gamma_1, cfg_{rch\_def}, C \vdash_{te} e_1[e_2] : \Gamma_3 : t_2}$$

$$\frac{\Gamma_1, cfg_{rch\_def}, C[\![\ [\![\ ]\!][e_2]\ ]\!] \vdash_{te} e_1 : \Gamma_2 : \{[st_1] : t, \ldots\}\ \textbf{wv} \quad \Gamma_2, cfg_{rch\_def}, C[\![\ e_1\ [\![\ ]\!]\ ]\!] \vdash_{te} e_1 : \Gamma_3 : st_2 \quad \vdash_{mtch} \{[st_1] : t, \ldots\} \approx \{\ldots, [st_2] : cte, \ldots\} \quad \text{reachCte}(cfg_{rch\_def}[C], e_1[e_2], \Gamma_3)}{\Gamma_1, cfg_{rch\_def}, C \vdash_{te} e_1[e_2] : \Gamma_3 : cte}$$

$$\frac{\Gamma_1, cfg_{rch\_def}, C[\![\ [\![\ ]\!][e_2]\ ]\!] \vdash_{te} e_1 : \Gamma_2 : \{[st_1] : t_1, \ldots\}\ \textbf{wv} \quad \Gamma_2, cfg_{rch\_def}, C[\![\ e_1\ [\![\ ]\!]\ ]\!] \vdash_{te} e_2 : \Gamma_3 : st_2 \quad \vdash_{mtch} \{[st_1] : t_1, \ldots\} \approx \{\ldots, [st_2] : t_2, \ldots\} \quad t_2 \notin cte}{\Gamma_1, cfg_{rch\_def}, C \vdash_{te} e_1[e_2] : \Gamma_3 : t_2}$$

Figure 17: Type checking for table indexing.

generated for a given program, the algorithm proceed by inferring new constraints from *Cs*, for which it is guaranteed that, if a solution exist for *Cs*, then the same solution solves the newly inferred constraints. This step intends to make evident the existence of a solution or expose any inconsistency present among the constraints, showing the absence of a solution. The last step generates solutions for each constraint.

Our type inference algorithm follows the previously described process, with minor additions to tackle the problem of type inference given our subtyping relation, which is slightly more complex: we have the supertype **dyn**, tuple types and a slightly more complex subtyping relation for primitive types, since we also have singleton types. The main additions involve enriching the expressiveness of the language to express constraints over types, and an added step that refines the possible types that could be assigned to an expression, for the case of primitive types.

5.3 Computing the Control Flow Graph

In order to compute the cfg for the program we follow traditional ideas from [1] adapted to Lua code. The resulting cfg_{rch_def} contains a family of sets of definitions of variables that are valid at every statement and expression of the program being typed. We identify each of such points with a context *C*, that we need to update accordingly through the whole type checking process. Such contexts also serve to identify the exact point in the program where the tool identified a potentially non-deterministic behavior. cfg_{rch_def} is indexed by these contexts. For brevity we do not show its definition, but it can be seen in the mechanization accompanying this paper.

5.4 Type checking

For brevity, we focus on the peculiarities of determining gc-safeness. Type checking is described by the typing relations $\vdash_{te} \subseteq \Gamma \times cfg_{rch\_def} \times$

$$\frac{\Gamma_1(x) = \{[st] : t, \ldots\}\ wkness_1 \qquad \Gamma_1, cfg_{rch\_def}, C[\![\ \textbf{setmetatable}(x, [\![\ ]\!])\ ]\!] \vdash_{te} e : \Gamma_2 : \{\ldots, [\langle \text{“\_\_mode”} : \textbf{str}\rangle] : \langle s : \textbf{str}\rangle, \ldots\}\ wkness_2 \quad \text{`v'} \in s \qquad \Gamma_3 = \Gamma_2[x : \{[st] : t, \ldots\}\ \textbf{wv}]}{\Gamma_1, cfg_{rch\_def}, C \vdash_{ts} \textbf{setmetatable}(x, e) : \Gamma_3}$$

$$\frac{\Gamma_1(x) = \{[st_1] : t_1, \ldots\}\ wkness_1 \qquad \Gamma_1, cfg_{rch\_def}, C[\![\ \textbf{setmetatable}(x, [\![\ ]\!])\ ]\!] \vdash_{te} e : \Gamma_2 : \{[st_2] : t_2, \ldots\}\ wkness_2 \quad \nvdash_{mtch} \{[st_2] : t_2, \ldots\} \approx \{\ldots, [\langle \text{“\_\_mode”} : \textbf{str}\rangle] : \langle s : \textbf{str}\rangle, \ldots\} \vee \text{`k'}, \text{`v'} \notin s \qquad \Gamma_3 = \Gamma_2[x : \{[st_1] : t_1, \ldots\}\ \textbf{strong}]}{\Gamma_1, cfg_{rch\_def}, C \vdash_{ts} \textbf{setmetatable}(x, e) : \Gamma_3}$$

Figure 18: Type checking: setmetatable.

$C \times e \times \Gamma \times t$ and $\vdash_{ts} \subseteq \Gamma \times cfg_{rch\_def} \times C \times s \times \Gamma$, (partially) described in Figure 17 and Figure 18, respectively.

We denote with Γ the environments mapping variable identifiers with their types. Since we are typing a dynamic language, the statements and expressions could change this mapping because of assignments of the same variable to values having different type. Therefore, the typing relation includes a second environment to reflect the changes. In the typing rules we ensure that the the values assigned to a certain variable have types related by subtyping.

The first rule in Figure 17 shows the typing of a (non-weak) table indexing, $e_1[e_2]$. As mentioned, in this prototype we simplify type checking by assuming that each key is a literal value. Nonetheless, this is enough to type check common idioms involving tables, in Lua. Assuming that e_2 can be successfully typed as st_2, we look into the type of e_1 for a field with the same type, $\vdash_{mtch} \{[st_1] : t_1, \ldots\} \approx \{\ldots, [st_2] : t_2, \ldots\}$. In that case, we successfully type the whole indexing expression as t_2, carrying in Γ_3 the (possible) modifications to the original environment.

The second rule in Figure 17 shows the typing of the indexing of a table that has weak values. If we can determine that the value being indexed belongs to the set of values that can be garbage-collected (*cte*), we need to check for the reachability of the value, to ensure a deterministic behavior of the indexing. We query cfg_{rch_def} for the set of definitions of variables that reaches the point C of the program (*i.e.*, the table indexing), and then traverse that set of definitions to check for the reachability of the exact expression $e_1[e_2]$ as expressed by the predicate reachCte from §3.3 (properly adapted to work with the information from our static analysis).

The last rule in Figure 17 shows the case of indexing a table that has weak values, but when the value being accessed does not belong to the set *cte*. In this case there is no risk of non-determinism.

For the present prototype the case of tables with weak keys (ephemerons) is trivially solved, since for any table, all of its keys will not belong to the set *cte* of values that could be garbage collected. If we allow also *cte*s as keys, checking for determinism would proceed analogous to the case of weak values, though with the expected modifications dictated by the semantics of ephemerons.

Another requirement for our typing relations is for them to recognize and keep track of changes in the weakness of a given table, as a result of calls to the service **setmetatable**. Figure 18 shows the typing rules for calls to this service. In the first rule we show the case when the given metatable contains the corresponding field to inform about a change in the weakness of the table. We therefore require the metatable to have a field with key of singleton type $\langle$“__mode” : **str**$\rangle$ and value of singleton type $\langle s : \textbf{str}\rangle$, with s containing the character 'v'. The environment Γ_3 will contain the updated weakness of table x. The last rule shows the case of a call to **setmetatable** with a metatable which does not contain proper information about changes in weakness of the table: it will result in the table's weakness being set to **strong**, regardless of the original weakness of the table.

```
1  local cache1 = {[1] = function() return 1 end,
2                  [2] = function() return 2 end,
3                  [3] = function() return 3 end}
4  local obj = {method = cache1[1], attr = {}}
5  local cache2 = {[1] = cache1[2]}
6  setmetatable(cache1, { __mode = "v"})
7  setmetatable(cache2, { __mode = "v"})
8  cache1[1]()
9  cache1[2]()
10 cache1[3]()
```

Figure 19: Example: Implementation of a simple cache.

```
1 local t1 = {}
2 t1["attr1"] = 1
3 t1["method"] = function(x) return x + t1["attr1"] end
4 t1["attr2"] = (t1["method"](t1["attr1"]))
5 setmetatable(t1, {__mode = "v"})
6 t1["method"](t1["attr2"])
```

Figure 20: Example: Tracking the addition of table fields.

5.5 Examples

In this section we show the capabilities of code analysis of the present version of **LuaSafe** with examples that, though artificial in concept, are meant to pinpoint the possibilities of the proposed approach. As mentioned in the introduction, the program from Figure 1 is correctly flagged as non-deterministic.

Additionally, Figure 19 shows the implementation of a cache-like structure, cache1 in Line 1, as a table with weak values. This cache stores several closures in fields indexed by different numbers. Beginning from Line 4, we create weak and strong references to the closures stored in cache1. In Line 4 we create an object-like table, obj, where we store a reference to one of the closures from cache1. In Line 5 we define another cache-like table, cache2, and we add another reference to a closure stored in cache1. In lines 6–7 we set cache1 and cache2 to have weak values. What follows are accesses to the closures in cache1, through indexing. **LuaSafe** correctly recognizes that the indexing in Line 8 is safe, since it involves the access of a *cte* (a closure), stored in a table with weak values, but for which there is a strong reference coming from the presence of

the closure as a method from obj. The situation is different for the last two accesses (lines 8 and 9): it recognizes two different kinds of ill accesses: in Line 9 the indexing involves a *cte* value from a weak table, but for which every reachability path contains at least one weak reference (besides cache1, it is only referenced from a value of cache2): *i.e.*, it is not strongly reachable. In Line 10, the value accessed has no other reference besides the one from cache1.

In Figure 20 we illustrate the possibility of keeping track of the addition of new fields to tables, by means of assignments. The example features a table, t1, which is defined field by field, with every new field defined in terms of the previous. The tool recognizes that, in the function call in Line 6, there is a *cte* being accessed which is not strongly reachable. Also, type inference and type checking correctly solve the type of the parameter being passed in the call, which is not a *cte*, hence, there is no risk of non-deterministic behavior. The example also serves to showcase some of the constructions of the language that **LuaSafe** handles, which includes every syntactic form except functions returning multiple values, assignment and definitions of multiple variables and tables with *cte*s keys.

6 FUTURE AND RELATED WORK

In the future we plan to incorporate several improvements to **LuaSafe**: an enriched type system, proofs of soundness of type inference and checking, and the inclusion of language features that were left out of this first prototype. The main known drawback of using PLT Redex, though, is its poor performance. At the moment, the implementation is mainly useful for testing ideas about static analysis, rather than for tackling the analysis of real-world Lua programs. Therefore, once **LuaSafe** is stabilized, we need to re-implement it in a more efficient language.

Another a promising line of work for the future is to adapt the core concepts to the new ECMAScript, which includes weak references and finalizers [13].

As for related work, we group them in three: formalizations of GC, theoretic tools related to the inference of types, and tools for static analysis of GC.

Formalizations of GC:. Leal *et. al.* present in [16] a formal semantics for a λ-calculus extended with references (strong and weak), and finalizers. From the literature surveyed, this is the only work where both interfaces to the GC are considered. The semantics presented for finalization does not impose an order of execution among finalizers, and resurrected objects' semantics does not differ from live ones. Also, there is no interaction between weak references and finalization. As described in §3.2, Lua's implementation of finalization imposes a chronological order of finalization, and resurrected objects' semantics differs from live ones in certain conditions, even with regard to resurrected objects present in weak tables. This adds a certain level of interaction between finalization and weak tables.

Morrisett *et. al.* present in [18] a reduction semantics for GC (named λ_{GC}), but without any interface with the garbage collector. The theory developed for proving correctness for GC served as a major source of inspiration for our own development. The given specification for a GC cycle does not consider reachability, but rather observes for the appearance of free variables when removing a given binding from the heap. In [12] is shown that specifying GC in terms of reachability results in an increased expressiveness of the resulting model, reflected in the possibility of emulating even more trace-based GC strategies. We followed that path.

Donnelly *et. al.* extended λ_{GC} including weak references [15]. They use their model (named λ_{weak}) to tackle the semantics of the key/values weak references present in the GHC implementation of Haskell (a concept similar to ephemerons, also present in Lua). Also, they present a type system for their model and show how to use it in the collection of reachable garbage (*i.e.*, *semantic* garbage). Finally, they tackle the problem of the introduction of non-determinism into the evaluation of a program that makes use of weak reference. They provide a decidable syntactic criterion for recognizing programs well-behaved with regard to GC (*i.e.*, with a deterministic behavior, regardless of their use of weak reference), and characterize semantically a larger class of programs with the same deterministic behavior. Because λ_{weak} is directly derived form λ_{GC} it lacks the expressiveness of a model based on reachability. On the other hand, the theory developed for their model is based on a set of observations over programs that considers the possibility of a non-deterministic behavior. Being non-determinism a phenomenon also present in our model, their theory served as a source of inspiration for the development of ours.

The concept of ephemerons and their implementation in Lua is described in [8]. However, they are not studied into a formal setting.

Type inference for Lua: Type inference for Lua has being already tackled by Mascarenhas *et. al.* in [9] to obtain an optimized compiler for Lua 5.1. In the same vein, Maidl *et. al.* present Typed Lua [17], a type system for Lua 5.2 that tackles several of the complexities of the language, with special care in the typing of common idioms used by the community of Lua.

While not strictly related to Lua, Anderson *et. al.* introduce in [2] a type inference algorithm for a language similar to JavaScript, together with the formulation of several properties that characterize the soundness of the proposed approach.

Static analysis for GC:. In [6] the authors consider a form of local static analysis to detect the type of reference (*collectable*, *weak*, and *strong*) that occurs in a trace of execution. They use this information to avoid memory-leaks. In [14] the static analysis performed is used to determine the correct scope of weak and strong references. Donnelly *et. al.* propose in [15] the recognition of gc-safeness, first, by providing a restricted set of programs, characterized syntactically, for which it can be asserted their deterministic behavior, and later, by a semantic definition of a wider class of programs, though not recognizable through syntactic analysis. In contrast, our approach to gc-safeness recognition aims at receiving the user program as it is, and doing a best-effort attempt in reasoning about the program's behavior.

REFERENCES

[1] Alfred V. Aho, Minica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *"Compilers Principles, Techniques and Tools"* (second ed.). Pearson Education Inc.
[2] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards Type Inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*. Springer-Verlag, Berlin, Heidelberg, 428–452. https://doi.org/10.1007/11531142_19
[3] Anonymous. 2014. Lua Analyzers. http://lua-users.org/wiki/ProgramAnalysis. (2014). Accessed: 2020-05-04.

[4] Anonymous. 2015. Lua 5.2 Reference Manual. https://www.lua.org/manual/5.2/manual.html. (2015). Accessed: 2020-05-04.

[5] Anonymous. 2018. Lua Implementations. http://lua-users.org/wiki/LuaImplementations. (2018). Accessed: 2020-05-04.

[6] Pavel Avgustinov, Eric Bodden, Elnar Hajiyev, Laurie Hendren, Ondrej LhotÃąk, Oege Moor, Neil Ongkingco, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Mathieu Verbaere. 2006. Aspects for Trace Monitoring, Vol. 4262. 20–39. https://doi.org/10.1007/11940197_2

[7] Hayes B. 1992. Finalization in the collector interface. *Bekkers Y., Cohen J. (eds) Memory Management. IWMM 1992. Lecture Notes in Computer Science.*

[8] Alexandra Barros and Roberto Ierusalimschy. 2008. Eliminating Cycles in Weak Tables. 14 (01 2008), 3481–3497.

[9] Fabio Mascarenhas de Queiroz. 2009. *Optimized Compilation of a Dynamic Language to a Managed Runtime Environment.* Ph.D. Dissertation. PontifÃŋcia Universidade CatÃşlica do Rio de Janeiro.

[10] Mark DeLoura. 2009. The Engine Survey. http://www.satori.org/2009/03/the-engine-survey-general-results/. (2009). Accessed: 2020-05-04.

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

[12] Yarom Gabay and Assaf J. Kfoury. 2007. A Calculus for Java's Reference Objects. *SIGPLAN Not.* 42, 8 (Aug. 2007), 9–17. https://doi.org/10.1145/1294297.1294299

[13] Sathya Gunasekaran and Mathias Bynens. 2019. Weak references and finalizers. https://v8.dev/features/weak-references. (2019). Accessed: 2020-05-04.

[14] M. Higuera-Toledano, Sergio Yovine, and Diego Garbervetsky. 2012. *Region-Based Memory Management: An Evaluation of Its Support in RTSJ.* 101–127. https://doi.org/10.1007/978-1-4419-8158-5_5

[15] Assaf Kfoury Kevin Donnelly, J. J. Hallett. 2006. Formal semantics of weak references. In *ISMM '06 Proceedings of the 5th international symposium on Memory management.* 126–137.

[16] Marcus Amorim Leal and Roberto Ierusalimschy. 2005. A Formal Semantics for Finalizers. *J. UCS* 11, 7 (2005), 1198–1214. https://doi.org/10.3217/jucs-011-07-1198

[17] A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy. 2015. A Formalization of Typed Lua. In *DLS '15.* 13. https://doi.org/10.1145/2816707.2816709

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

[19] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. 2011-2013. *Lua 5.2 Reference Manual.* Available at www.lua.org/manual/5.2/manual.html.

[20] Mallku Soldevila. 2020. Code for **LuaSafe**. https://github.com/Mallku2/luasafe-redex. (2020). Accessed: 2020-05-04.

[21] Mallku Soldevila. 2020. Code for Lua's semantics in PLT Redex. https://github.com/Mallku2/lua-gc-redex-model. (2020). Accessed: 2020-05-04.

[22] Mallku Soldevila, Beta Ziliani, and Daniel Fridlender. 2020. Appendix. https://people.mpi-sws.org/~beta/papers/ppdp20-appendix.pdf. (2020). Accessed: 2020-05-04.

[23] Mallku Soldevila, Beta Ziliani, Bruno Silvestre, Daniel Fridlender, and Fabio Mascarenhas. 2017. Decoding Lua: Formal Semantics for the Developer and the Semanticist. In *Proceedings of the 13th ACM SIGPLAN Dynamic Languages Symposium (DLS 2017).*