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.

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 [4]. 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.

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

```
local t = {}                            --create an empty table
setmetatable(t, {__mode = 'v' })  --set its values as weak
t[1] = {}                    --assign an empty table to key 1
local i = 0
while true do
  i = i + 1
  ...                   --some code, possibly generating garbage
  if not t[1] then break end
end

return i                          --this value cannot be predicted
```

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

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 ill-uses 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 in GC). Our model builds on top of that from [22], 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 [22]. It covers weak tables and *finalizers*, the latter being functions executed when an element is about to be disposed. Without these interfaces, we show, GC is deterministic. 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 ill-uses 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.
- A prototype tool, **LuaSafe**, to help uncover potential misuses of weak tables.

$$
\begin{aligned}
E &::= [\![\;]\!] \mid E\,(\,e\,,\ldots\,) \mid v\,(\,E_l\,) \mid var\,,\,...\,=\,E_l \\
&\quad \mid \textbf{local}\ var\,,\,...\,=\,E_l\ \textbf{in}\ s\ \textbf{end} \\
&\quad \mid \textbf{setmetatable}(E_l) \mid \textbf{error}\ E \mid \ldots \\
E_l &::= v\,,\ldots\,,\,E\,,\,e\,,\ldots \\
s &::= e\,(\,e\,,\ldots\,) \mid var\,,\,...\,=\,e\,,\,... \\
&\quad \mid \textbf{local}\ var\,,\,...\,=\,e\,,\,...\ \textbf{in}\ s\ \textbf{end} \\
&\quad \mid \textbf{setmetatable}(e\,,\ldots) \mid \textbf{error}\ e \mid \ldots \\
e &::= v \mid e\,(\,e\,,\ldots\,) \mid \{\,[e] = e\,,\,...\,\} \\
&\quad \mid \textbf{function}\ (\,x\,,\ldots\,)\ s\ \textbf{end} \mid r \mid \ldots \\
v &::= number \mid string \mid tid \mid cid \mid \ldots \\
var &::= x \mid e\,[e]
\end{aligned}
$$

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

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 [22], 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. 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, the *objects storage*, denoted with θ. We must point out a difference from the model in [22]: 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 $\overset{\text{L}}{\mapsto}$, over configurations of the form $\sigma : \theta : s$. For instance, the following rule formalizes a function call:

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

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:

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

The pattern from in the left of $\overset{\text{L}}{\mapsto}$ 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 [22], 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:

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

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).

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:

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

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 $\text{reach}(l, \pi_2(\theta(tid)), \sigma, \theta \setminus tid)$, 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).

$\text{gc}(s, \sigma, \theta) = (\sigma_1, \theta_1)$, where:

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

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

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

```
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.

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 concepts 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.

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.

```
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.

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.

$$\frac{\begin{array}{c}\forall\ 1 \leq \text{i}, \mathit{field}_\text{i} = v \vee \mathit{field}_\text{i} = [\ v\ ]\ =\ v' \\ \theta_2 = (\ \mathit{tid},\ (\ \text{addkeys}(\{\mathit{field}_1,\ \ldots\})\ ,\ \textbf{nil}, \perp)),\ \theta_1\end{array}}{\theta_1\ :\ \{\mathit{field}_1,\ \ldots\}\ \overset{\text{F}}{\mapsto}\ \theta_2\ :\ \mathit{tid}}$$

$$\frac{\begin{array}{c}\delta(\text{type}, v) \in \{\text{“table”}, \text{“nil”}\} \\ \text{indexmetatable}(\mathit{tid}, \text{“\_\_metatable”}, \theta_1) = \textbf{nil} \\ \theta_2 = \theta_1[\mathit{tid} := (\pi_1(\theta_1(\mathit{tid})), v, \text{set\_fin}(\mathit{tid}, v, \theta_1))]\end{array}}{\theta_1 : \textbf{setmetatable}(\mathit{tid}, v) \overset{\text{F}}{\mapsto}\ \theta_2 : \mathit{tid}}$$

Figure 5: Selected rules extended with finalization.

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 $\perp$, 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 $\perp$, as shown in the first rule of Figure 5. We present its semantics (and the remaining computation rules for finalization), with a new relation, $\overset{\text{F}}{\mapsto}$.

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 $\mathcal{P} \longrightarrow \mathcal{P}$, which should provide an element of $\mathcal{P}$ larger than its argument. We will also need $\perp \in \mathcal{P}$, 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.

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 $\perp$. 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 ($\perp$).

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

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

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

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

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

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

Figure 6: Function set_fin for computing the pos field.

$$\text{gc}_\text{fin}(s, \sigma, \theta) = (\sigma_1, \theta_1', t),\ \mathit{where}$$

$$\mathit{gc} \begin{cases} \sigma = \sigma_1 \uplus \sigma_2 \\ \theta = \theta_1 \uplus \theta_2 \\ \forall \text{l} \in \text{dom}(\sigma_2) \cup \text{dom}(\theta_2), \neg\text{reach}(\text{l}, s, \sigma, \theta) \end{cases}$$

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

Figure 7: GC cycle with finalization.

$$\text{marked}(\mathit{tid}, \theta) \doteq \pi_3(\theta(\mathit{tid})) \notin \{\perp, \oslash\}$$

$$\text{not\_reach\_fin}(l, \sigma, \theta) \doteq \nexists\ \mathit{tid} \in \text{dom}(\theta), l \neq \mathit{tid}\ \wedge \text{marked}(\mathit{tid}, \theta)\ \wedge\ \text{reach}(l, \mathit{tid}, \sigma, \theta)$$

$$\text{fin}(\mathit{tid}, s, \sigma, \theta) \doteq \neg\text{reach}(\mathit{tid}, s, \sigma, \theta) \wedge \text{marked}(\mathit{tid}, \theta)$$

$$\text{next\_fin}(\mathit{tid}, s, \sigma, \theta) \doteq \forall \mathit{tid}' \in \text{dom}(\theta), \text{fin}(\mathit{tid}', s, \sigma, \theta) \Rightarrow \pi_3(\theta(\mathit{tid}')) \leq^{fin} \pi_3(\theta(\mathit{tid}))$$

Figure 8: Predicates for finalization.

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 marked for finalization, otherwise we will lose a call to a finalizer.

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

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

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

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

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 gc_{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.

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.

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

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

Figure 10: Strong occurrences of a table.

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 3].

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.

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 *cte*s) 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.

$$\begin{aligned}
&\mathrm{eph}(id, (k, v), tid, rt, \sigma, \theta) = \mathrm{reachCte}(id, v, \sigma, \theta, rt) \wedge \\
&\qquad\qquad [k \notin cte \vee \mathrm{reachCte}(k, rt, \sigma, \theta|_{tid|_k}, rt)] \\
&\mathrm{reachTable}(id, tid, \sigma, \theta, rt) = \\
&\qquad [\exists\, (k, v) \in \mathrm{SO}(tid, \theta), \mathrm{eph}(id, (k, v), tid, rt, \sigma, \theta)] \vee \\
&\qquad [\exists\, v \in \mathrm{SO}(tid, \theta), \mathrm{reachCte}(id, v, \sigma, \theta, rt)] \vee \\
&\qquad \mathrm{reachCte}(id, \pi_2(\theta(tid)), \sigma, \theta, rt) \\
&\mathrm{reachCte}(id, t, \sigma, \theta, rt) = id \in t \vee \\
&\qquad\qquad \exists\, r \in t,\ \mathrm{reachCte}(id, \sigma(r), \sigma, \theta, rt) \vee \\
&\qquad\qquad \exists\, tid \in t, \mathrm{reachTable}(id, tid, \sigma, \theta, rt) \vee \\
&\qquad\qquad \exists\, cid \in t, \mathrm{reachCte}(id, \theta(cid), \sigma, \theta, rt)
\end{aligned}$$

Figure 11: Reachability of a collectible element.

Before presenting the predicate reachCte, we introduce the predicate reachTable (shown in Figure 11), which expands the reachability tree for a table *tid*, when determining reachability for an identifier *id* with respect to a term *rt*, and stores σ and θ. We first check for reachability following references from a table *tid*, when it happens to be an ephemeron, 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 cannot be GC, *i.e.*, is not a member of *cte* or it is reachable from the root set of references from *rt*: *i.e.*, the reachability of the value is affected by the reachability of its key. In doing so, we must not take into account v to allow the collection of a field where the only reference to the key comes from the value. We use the notation $\theta|_{tid|_k}$ to denote the resulting store from removing the field with key k from the table *tid*.

If the table is not an ephemeron, 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:

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

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

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

Figure 12: GC cycle extended with weak tables.

(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 value from a weak table by adding the following predicate:

$$\mathrm{not\_fin\_val}(tid, \theta) \doteq \nexists tid' \in \mathrm{dom}(\theta), k \in v/ \\ (\mathrm{wk?}(tid', \theta) \vee \mathrm{wv?}(tid', \theta)) \wedge (k, tid) \in \pi_1(\theta(tid'))$$

(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', 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 weak keys susceptible for finalization, they are removed only after they are 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 $\mathrm{gc}_{\mathrm{fin\_weak}}$

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

Figure 13: Result of a program and associated functions.

metafunction, and its interference with the execution of the program does not differ from what regular GC does.

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 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 13), 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 $\overset{\mathsf{L}}{\mapsto}$ (that is, Lua without GC):

Corollary 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 $\sigma : \theta : s \Downarrow \sigma' : \theta' :$* ***error*** *$v$.*
- *The execution ends normally, with stores σ' and θ', and some values $v, \ldots$ are returned:*
  $\sigma : \theta : s \ \Downarrow \sigma' : \theta' : E\,[\![\textbf{return}\ v, \ldots]\!]$*, 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 $E\,[\![\textbf{return}\ v, \ldots]\!]$, 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 $\overset{\mathsf{L}}{\mapsto}$ is deterministic, which basically boils down to:

Assumption 4.1 (Restrictions to the model).

(1) The memory manager is deterministic, and new references are always created fresh, i.e.*, there is no re-use of memory location.*
(2) There are no services that exposes external variables, like the time, the file system, a random number generator, etc.

The first assumption can be lifted off if services that expose the details of memory management are prohibited (iterators for tables with non-numeric keys, the tostring service, *etc.*).

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 $\overset{\mathsf{L}}{\mapsto}$ (*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 4.2 (Observations). For a given well-formed configuration C, and execution rules $\rightarrow$:

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

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 $\rightarrow = \overset{\mathsf{L}}{\mapsto}$, 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 $\overset{\text{GC}}{\mapsto}$ to $\overset{\text{L}}{\mapsto}$ we are not introducing stuck states.

Observations are useful to describe program equivalence:

Definition 4.3 (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 4.4 (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 $\overset{\text{GC}}{\mapsto}$.

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 $\overset{\text{GC}}{\mapsto}$ the observations over a given program are not altered. The desired statement is captured in the following statement (where $\overset{\text{L+GC}}{\mapsto} = \overset{\text{L}}{\mapsto} \cup \overset{\text{GC}}{\mapsto}$):

Theorem 4.5 (GC correctness). *For a given well-formed configuration $\sigma : \theta : s$,*

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

The proof is included in appendix A.

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

Corollary 4.6 (Determinism of GC). *For a well-formed configuration $\sigma : \theta : s$,* $|\text{obs}(\sigma : \theta : s, \overset{\text{L+GC}}{\mapsto})| = 1$

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 4.7 (Non-deterministic behavior). *Form some well-formed configuration $\sigma : \theta : s$,*

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

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 later.

More concretely, in this section we introduce **LuaSafe**, a prototype static analyzer that aims at the detection of ill-uses 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 $\overset{\text{L}}{\mapsto}$ 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 14 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 in 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.

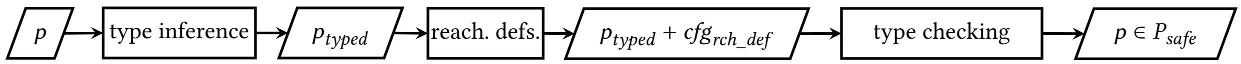

Figure 14: Design of LuaSafe.

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 ::= $\langle v_{\mathrm{st}}$: *prmt*$\rangle$
v_{st} ::= **nil** | *string* | *boolean* | *number*
wkness ::= **strong** | **wk** | **wv** | **wkv**

Figure 15: Grammar for typed terms.

5.1 Type system

Common to all the steps of **LuaSafe** lies the type system. Figure 15 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 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 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 $\vdash_{te} \subseteq \Gamma \times cfg_{rch\_def} \times C \times e \times \Gamma \times t$ and $\vdash_{ts} \subseteq \Gamma \times cfg_{rch\_def} \times C \times s \times \Gamma$, (partially) described in Figure 16 and Figure 17, 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 16 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

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

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

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

Figure 16: Type checking for table indexing.

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 16 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 16 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 17 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 ⟨"__mode" : **str**⟩ and value of singleton type ⟨s : **str**⟩, 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.

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 18 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 19 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

Future work could include one of the several venues of improvement of **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 for this investigations is the poor performance of the resulting programs. The implementation is useful mainly for the testing ideas about static analysis, rather than tackling the analysis of real-world Lua programs. Future work could include the re-implementation of **LuaSafe** 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.

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

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

Figure 17: Type checking: setmetatable.

```
local cache1 = {[1]  = function() return 1 end,
                [2]  = function() return 2 end,
                [3]  = function() return 3 end}
local obj  = {method = cache1[1],  attr  =  {}}
local cache2 = {[1]  = cache1[2]}
setmetatable(cache1, { __mode = "v"})
setmetatable(cache2, { __mode = "v"})
cache1 [1]()
cache1 [2]()
cache1 [3]()
```

Figure 18: Example: Implementation of a simple cache.

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

Figure 19: Example: Tracking the addition of table fields.

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] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *"Compilers Principles, Techniques and Tools"*. Pearson Education Inc., second edition, 2007.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 428–452, Berlin, Heidelberg, 2005. Springer-Verlag.

[3] Anonymous. Lua 5.2 reference manual. https://www.lua.org/manual/5.2/manual.html. Accessed: 2020-05-04.

[4] Anonymous. Lua analyzers. http://lua-users.org/wiki/ProgramAnalysis. Accessed: 2020-05-04.

[5] Anonymous. Lua implementations. http://lua-users.org/wiki/LuaImplementations. Accessed: 2020-05-04.

[6] P. Avgustinov, E. Bodden, E. Hajiyev, L. Hendren, O. LhotÃąk, O. Moor, N. Ongkingco, D. Sereni, G. Sittampalam, J. Tibble, and M. Verbaere. Aspects for trace monitoring. volume 4262, pages 20–39, 01 2006.

[7] H. B. Finalization in the collector interface. Springer, Berlin, Heidelberg, 1992.

[8] A. Barros and R. Ierusalimschy. Eliminating cycles in weak tables. 14:3481–3497, 01 2008.

[9] F. M. de Queiroz. *Optimized Compilation of a Dynamic Language to a Managed Runtime Environment.* PhD thesis, PontifÃŋcia Universidade CatÃşlica do Rio de Janeiro, 2009.

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

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

[12] Y. Gabay and A. J. Kfoury. A calculus for Java's reference objects. *SIGPLAN Not.*, 42(8):9–17, Aug. 2007.

[13] S. Gunasekaran and M. Bynens. Weak references and finalizers. https://v8.dev/features/weak-references. Accessed: 2020-05-04.

[14] M. Higuera-Toledano, S. Yovine, and D. Garbervetsky. *Region-Based Memory Management: An Evaluation of Its Support in RTSJ*, pages 101–127. 11 2012.

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

[16] M. A. Leal and R. Ierusalimschy. A formal semantics for finalizers. *J. UCS*, 11(7):1198–1214, 2005.

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

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

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

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

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

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

Appendices

A Properties of GC

To reach to a proof of the correctness of $\overset{GC}{\mapsto}$ we will require, first, to check for several lemmas about simple properties that hold for both, $\overset{L}{\mapsto}$ and $\overset{GC}{\mapsto}$.

Properties preserved by $\overset{L}{\mapsto}$. The first lemma states that once a binding becomes amenable for collection, it will remain in that state after any computation step from $\overset{L}{\mapsto}$.[1] For its proof we will assume that the reader is familiar with the model presented in [22]. A complete proof would require case analysis on every computation step from said model. For reasons of brevity, we will consider just a few cases.

Lemma A.1. *For configurations* $(\sigma_1 : \theta_1 : s_1)$*,* $(\sigma_2 : \theta_2 : s_2)$*, if* $(\sigma_1 : \theta_1 : s_1) \overset{L}{\mapsto} (\sigma_2 : \theta_2 : s_2)$*, for* $(\sigma_1 : \theta_1 : s_1)$ *well-formed, then* $\forall l \in dom(\sigma_1) \cup dom(\theta_1)$*,* $\neg\text{reach}(l, s_1, \sigma_1, \theta_1) \Rightarrow \neg\text{reach}(l, s_2, \sigma_2, \theta_2)$.

Proof. We will follow the modular structure of $\overset{L}{\mapsto}$ to reason over the step that transforms $(\sigma_1 : \theta_1 : s_1)$ into $(\sigma_2 : \theta_2 : s_2)$. We have the following cases for the step taken from $\overset{L}{\mapsto}$:

- *The computation does not depend on the content of the stores (*i.e., *it does not change bindings from a store or dereferences locations):* then, it can be seen, by case analysis on each computation rule, that such computation step does not introduce any reference into the instruction term. What could happen is that the root set is reduced, by deleting references present into s_1. In any case, for a given $l \in dom(\sigma_1) \cup dom(\theta_1)$, if $\neg\text{reach}(l, s_1, \sigma_1, \theta_1)$ it must be the case that also $\neg\text{reach}(l, s_2, \sigma_2, \theta_2)$.
- *The computation changes or dereferences locations from* σ_1 : for a given $l \in dom(\sigma_1) \cup dom(\theta_1)$, such that $\neg\text{reach}(l, s_1, \sigma_1, \theta_1)$ let us assume that $\text{reach}(l, s_2, \sigma_2, \theta_2)$. To reason about the statement, we would need to do case analysis on every possible computation step that interacts with the values store. As an example, let us consider the implicit dereferencing of references to σ_1.[2] Then it must be the case that s_1 matches against the pattern $E[\![\ r\ ]\!]$, for an evaluation context E and a reference r, and the computation is:

$$\sigma_1 : \theta_1 : E[\![\ \overset{\overset{s_1}{\equiv}}{r}\ ]\!] \overset{L}{\mapsto} \ \sigma_1 : \theta_1 : E[\![\overset{\overset{s_2}{\equiv}}{\sigma_1(r)}]\!]$$

  where both stores remain unmodified after the computation. Then, the root set just changed by replacing r by the references in $\sigma_1(r)$. If $\neg\text{reach}(l, s_1, \sigma_1, \theta_1)$ but $\text{reach}(l, s_2, \sigma_2, \theta_2)$, this would mean that l is reachable from the references in $\sigma_1(r)$. But in s_1, the references from $\sigma_1(r)$ were also reachable, making l reachable in s_1, contradicting our hypothesis. Then, it must be the case that if $\neg\text{reach}(l, s_1, \sigma_1, \theta_1)$, l remains unreachable in $(\sigma_2 : \theta_2 : s_2)$.
- *The computation changes or dereferences locations from* θ_1: let us assume that for a given $l \in dom(\sigma_1) \cup dom(\theta_1)$, $\neg\text{reach}(l, s_1, \sigma_1, \theta_1) \ \wedge \ \text{reach}(l, s_2, \sigma_2, \theta_2)$. Again we will just analyze one case, among every computation that interacts with the store θ_1. We will consider the rule that describes how tables are allocated in θ_1. Then, it must be the case that s_1 matches against the pattern $E[\![\ t\ ]\!]$, for an evaluation context E and a table constructor t, where every field haven been evaluated, making the table ready for allocation. Then, the (simplified) computation is:

$$\frac{tid \notin \text{dom}(\theta_1) \qquad \theta_2 = (tid,\ (t,\ \textbf{nil},\ \bot)),\ \theta_1}{(\sigma_1\ :\ \theta_1\ :\ E[\![\ t\ ]\!]) \overset{L}{\mapsto} (\sigma_1\ :\ \theta_2\ :\ E[\![\ tid\ ]\!])}$$

  where the values store remains unchanged, *i.e.*, $\sigma_2 = \sigma_1$. Then, the root set just changed by replacing the references in t by the fresh table identifier tid. If $\neg\text{reach}(l, s_1, \sigma_1, \theta_1)$ but $\text{reach}(l, s_2, \sigma_2, \theta_2)$, this would mean that $l = tid$, which cannot be the case as $l \in dom(\sigma_1) \cup dom(\theta_1)$ and $tid \notin dom(\sigma_1) \cup dom(\theta_1)$. Then it must be the case that if $\neg\text{reach}(l, s_1, \sigma_1, \theta_1)$, l remains unreachable in $(\sigma_2 : \theta_2 : s_2)$.

[1] Note that such simple property does not hold anymore if we introduce weak tables or finalization.

[2] In [22], for purposes of simplification of the desugared Lua code from test suites, we included implicit dereferencing of references to values, as done in [19].

□

The following definition and lemma capture a standard concept in operational semantics for imperative languages: for a given instruction term, the outcome of its execution under given stores will depend on the content of the reachable portion of said stores.

Definition A.2. For well-formed configurations $(\sigma_1 : \theta_1 : s)$ and $(\sigma_2 : \theta_2 : s)$, we will say that both configurations *coincide in the reachable portion of their stores*, denoted

$$(\sigma_1 : \theta_1 : s) \overset{\mathsf{rch}}{\sim} (\sigma_2 : \theta_2 : s)$$

if and only if $\forall l \in dom(\sigma_1) \cup dom(\theta_1) / \mathsf{reach}(l, s, \sigma_1, \theta_1)$, then:

- $\mathsf{reach}(l, s, \sigma_2, \theta_2)$
- $l \in dom(\sigma_1) \Rightarrow \sigma_1(l) = \sigma_2(l)$
- $l \in dom(\theta_1) \Rightarrow \theta_1(l) = \theta_2(l)$

and the same holds $\forall l \in dom(\sigma_2) \cup dom(\theta_2)$.

In the previous definition, we are assuming that, if needed, it is always possible to provide a renaming of locations from both configurations to make them equivalent in the sense expressed by $\overset{\mathsf{rch}}{\sim}$. Finally, it is easy to show that $\overset{\mathsf{rch}}{\sim}$ is an equivalence relation.

The important property, satisfied by configurations that coincide in the reachable portion of their stores, is stated in the following lemmas:

Lemma A.3. *For well-formed configurations* $(\sigma_1 : \theta_1 : s_1)$ *and* $(\sigma_2 : \theta_2 : s_1)$*, such that:*

$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{rch}}{\sim} (\sigma_2 : \theta_2 : s_1)$$

if $\exists (\sigma_3 : \theta_3 : s_2) / (\sigma_1 : \theta_1 : s_1) \overset{\mathsf{L}}{\mapsto} (\sigma_3 : \theta_3 : s_2)$*, then* $\exists (\sigma_4 : \theta_4 : s_2) / (\sigma_2 : \theta_2 : s_1) \overset{\mathsf{L}}{\mapsto} (\sigma_4 : \theta_4 : s_2)$ *and:*

$$(\sigma_3 : \theta_3 : s_2) \overset{\mathsf{rch}}{\sim} (\sigma_4 : \theta_4 : s_2)$$

Proof. We will follow the modular structure of $\overset{\mathsf{L}}{\mapsto}$ to reason over the step that transforms $(\sigma_2 : \theta_2 : s_1)$ into $(\sigma_4 : \theta_4 : s_2)$:

- *The computation does not change bindings from a store or dereferences locations* : then it must be the case that every information from the stores is already put into the instruction term s_1 so as to make the computation from $\overset{\mathsf{L}}{\mapsto}$ viable, without regard to the content of the stores. Also, after the computation the stores are not modified. It implies that:

$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{L}}{\mapsto} (\overset{\overset{\sigma_1}{=}}{\sigma_3} : \overset{\overset{\theta_1}{=}}{\theta_3} : s_2) \ \wedge \ (\sigma_2 : \theta_2 : s_1) \overset{\mathsf{L}}{\mapsto} (\overset{\overset{\sigma_2}{=}}{\sigma_4} : \overset{\overset{\theta_2}{=}}{\theta_4} : s_2)$$

  The root set of references in both configurations, $(\sigma_3 : \theta_3 : s_2)$ and $(\sigma_4 : \theta_4 : s_2)$, is the same. And, since $(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{rch}}{\sim} (\sigma_2 : \theta_2 : s_1)$ and the stores are not modified after the step from $\overset{\mathsf{L}}{\mapsto}$, it follows that the reachable portion of the stores, from the root set defined by s_2, must coincide, according to $\overset{\mathsf{rch}}{\sim}$, in the configurations obtained after $\overset{\mathsf{L}}{\mapsto}$. Hence:

$$(\sigma_3 : \theta_3 : s_2) \overset{\mathsf{rch}}{\sim} (\sigma_4 : \theta_4 : s_2)$$

- *The computation changes or dereferences locations from* σ_1: we would need to do case analysis on each computation that interacts with the value store. As an example, let us consider the implicit dereferencing of references to the values store. The hypothesis can be rewritten as:

$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{L}}{\mapsto} (\overset{\overset{\sigma_1}{=}}{\sigma_3} : \overset{\overset{\theta_1}{=}}{\theta_3} : s_2)$$

  where s_2 contains the value associated with the reference dereferenced by $\overset{\mathsf{L}}{\mapsto}$. Because:

$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{rch}}{\sim} (\sigma_2 : \theta_2 : s_1)$$

  the dereferencing operation will return the same result, if executed over σ_2. Then:

$$(\sigma_2 : \theta_2 : s_1) \overset{\mathsf{L}}{\mapsto} (\overset{\overset{\sigma_2}{=}}{\sigma_4} : \overset{\overset{\theta_2}{=}}{\theta_4} : s_2)$$

  Finally, because the stores are unmodified, after the step from $\overset{\mathsf{L}}{\mapsto}$, and since the reachable portions of the stores in the original configurations coincide, according to $\overset{\mathsf{rch}}{\sim}$, then, it must be the case that the reachable portions of the stores obtained after $\overset{\mathsf{L}}{\mapsto}$ must also coincide, if we consider the same root of references. Hence:

$$(\sigma_3 : \theta_3 : s_2) \overset{\mathsf{rch}}{\sim} (\sigma_4 : \theta_4 : s_2)$$

- *The computation changes or dereferences locations from* θ_1: we would need to do case analysis on each computation that interacts with θ_1. As an example, let us consider table allocation. The hypothesis can be rewritten as:

$$(\sigma_1 : \theta_1 : E[\![t]\!]) \overset{\mathsf{L}}{\mapsto} (\overset{\overset{\sigma_3}{=}}{\sigma_1} : \overset{\overset{\theta_3}{=}}{\theta_1 \uplus \{(tid, t, \mathbf{nil}, \perp)\}} : E[\![tid]\!])$$

  Then we can assume that:

$$(\sigma_2 : \theta_2 : E[\![t]\!]) \overset{\mathsf{L}}{\mapsto} (\overset{\overset{\sigma_4}{=}}{\sigma_2} : \overset{\overset{\theta_4}{=}}{\theta_2 \uplus \{(tid, t, \mathbf{nil}, \perp)\}} : E[\![tid]\!])$$

  where, if needed, we could apply a consistent renaming of tables' id in $(\sigma_2 : \theta_2 : E[\![t]\!])$, such that it preserves its equivalence with $(\sigma_1 : \theta_1 : E[\![t]\!])$ and *tid* is available as a fresh table identifier. Then, it follows immediately that:

$$(\sigma_3 : \theta_3 : E[\![tid]\!]) \overset{\mathsf{rch}}{\sim} (\sigma_4 : \theta_4 : E[\![tid]\!])$$

□

Finally, the following lemma express an intuitive property that holds among final configurations that happen to be equivalent, according to $\overset{\mathsf{rch}}{\sim}$:

Lemma A.4. *For final configurations* $(\sigma_1 : \theta_1 : s)$ *and* $(\sigma_2 : \theta_2 : s)$*, such that* $(\sigma_1 : \theta_1 : s) \overset{\mathsf{rch}}{\sim} (\sigma_2 : \theta_2 : s)$*, then:*

$$\mathsf{result}(\sigma_1 : \theta_1 : s) = \mathsf{result}(\sigma_2 : \theta_2 : s)$$

Proof. The result will follow directly from the definition of result, in Figure 13, and $\overset{\mathsf{rch}}{\sim}$. We will do a case analysis on the structure of s, for the configuration $(\sigma_1 : \theta_1 : s)$, considering that it is the final state of a convergent computation:

- $s = \mathbf{return}\, v_1, ..., v_n$: for simplicity we consider the case $n = 1$, and we omit a possible context E where the **return** statement could occur, since it is not taken into account by the notion of result of a program, as defined by result. For larger values of n the reasoning remains the same:
  - $v_1 \in number \cup string$: then, neither $\mathsf{result}(\sigma_1 : \theta_1 : s)$ nor $\mathsf{result}(\sigma_2 : \theta_2 : s)$ depend on the content of the stores. Hence, $\mathsf{result}(\sigma_1 : \theta_1 : s) = \mathsf{result}(\sigma_2 : \theta_2 : s)$.

– $v_1 \in tid \cup cid$: let us consider that $v_1 = tid$ for some $tid \in \mathrm{dom}(\theta_1)$ (the reasoning for the case $v_1 \in cid$ is similar). Then, by definition of result:

$$\mathrm{result}(\sigma_1 : \theta_1 : \mathbf{return}\ tid) = \sigma_1|_S : \theta_1|_T : \mathbf{return}\ tid$$

$$\text{where} \begin{cases} S = \bigcup_{r \in \mathrm{dom}(\sigma_1),\ \mathrm{reach}(r, \mathbf{return}\ tid, \sigma_1, \theta_1)} r \\ T = \bigcup_{id \in \mathrm{dom}(\theta_1),\ \mathrm{reach}(id, \mathbf{return}\ tid, \sigma_1, \theta_1)} id \end{cases}$$

Then, clearly $\mathrm{result}(\sigma_1 : \theta_1 : \mathbf{return}\ tid)$ is depending on the reachable portions of σ_1 and θ_1, beginning with the root set defined by tid. Because

$$(\sigma_1 : \theta_1 : tid) \overset{\mathrm{rch}}{\sim} (\sigma_2 : \theta_2 : tid)$$

the reachable portions of both configurations coincide. Hence, $\mathrm{result}(\sigma_1 : \theta_1 : s) = \mathrm{result}(\sigma_2 : \theta_2 : s)$.

– $s = \mathbf{error}\ v$: this case is identical to the previous one.

– $s = ;$: then, $\mathrm{result}(\sigma_1 : \theta_1 : s)$ is not depending on the content of the stores, and so is the case for $\mathrm{result}(\sigma_2 : \theta_2 : s)$. Hence,

$$\mathrm{result}(\sigma_1 : \theta_1 : s) = \mathrm{result}(\sigma_2 : \theta_2 : s)$$

□

Properties preserved by $\overset{GC}{\mapsto}$. A simple property to ask for is that reachable bindings are preserved, in the sense that they are still reachable and the value to which a given location is mapped is not changed after $\overset{GC}{\mapsto}$. We express this property with the following two lemmas:

Lemma A.5. *For a well-formed configuration* $(\sigma_1 : \theta_1 : s)$*, if* $(\sigma_1 : \theta_1 : s) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s)$*, for some configuration* $(\sigma_2 : \theta_2 : s)$*, then* $\forall r \in dom(\sigma_1), \mathrm{reach}(r, s, \sigma_1, \theta_1) \Rightarrow \sigma_1(r) = \sigma_2(r)$*. The analogous holds for any* $id \in dom(\theta_1)$*.*

Proof. Let $r \in dom(\sigma_1), \mathrm{reach}(r, s, \sigma_1, \theta_1)$. Then, by Definition 3.2 and $\overset{GC}{\mapsto}$, it must be the case that $\mathrm{gc}(s, \sigma_1, \theta_1) = (\sigma_2, \theta_2)$ and $\sigma_1(r) = \sigma_2(r)$.

For elements from $\mathrm{dom}(\theta_1)$ the reasoning is analogous to the previous case. □

Lemma A.6. *For a well-formed configuration* $(\sigma_1 : \theta_1 : s)$*, if* $(\sigma_1 : \theta_1 : s) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s)$*, for some configuration* $(\sigma_2 : \theta_2 : s)$*, then* $\forall l \in dom(\sigma_1) \cup dom(\theta_1), \mathrm{reach}(l, s, \sigma_1, \theta_1) \Rightarrow \mathrm{reach}(l, s, \sigma_2, \theta_2)$.

Proof. We will prove it by induction on the minimum number of dereferences of locations from σ_1 or θ_1 that needs to be performed to reach to a given location l, for which $\mathrm{reach}(l, s, \sigma_1, \theta_1)$ holds. By looking at Definition 3.1, one of the following cases should hold:

- $l \in s$: then it follows directly that $\mathrm{reach}(l, s, \sigma_2, \theta_2)$.
- $\exists r \in \mathrm{dom}(\sigma_1), l \in \sigma_1(r)$, which is in a reachability path of minimum distance, from the root set to r : then $\mathrm{reach}(r, s, \sigma_1, \theta_1)$, and by inductive hypothesis, $\mathrm{reach}(r, s, \sigma_2, \theta_2)$. Also, by lemma A.5, $\sigma_1(r) = \sigma_2(r)$. Then $l \in \sigma_2(r)$ and $\mathrm{reach}(l, s, \sigma_2, \theta_2)$, by definition.
- $\exists tid \in \mathrm{dom}(\theta_1), l \in \pi_1(\theta_1(tid))$, which is in a reachability path of minimum distance, from the root set to l: then $\mathrm{reach}(tid, s, \sigma_1, \theta_1)$, and by inductive hypothesis, $\mathrm{reach}(tid, s, \sigma_2, \theta_2)$. Also, by Lemma A.5, $\theta_1(tid) = \theta_2(tid)$. Then $l \in \pi_1(\theta_2(tid))$ and $\mathrm{reach}(l, s, \sigma_2, \theta_2)$ by definition.
- $\exists cid \in \mathrm{dom}(\theta_1), l \in \theta_1(cid)$, which is in a reachability path of minimum distance, from the root set to l: the reasoning is analogous to the previous case. It follows directly that $\mathrm{reach}(l, s, \sigma_2, \theta_2)$.
- $\exists tid \in \mathrm{dom}(\theta_1), l \in \pi_2(\theta_2(tid))$, which is in a reachability path of minimum distance, from the root set to l: the situation is analogous to the previous case. It follows directly that $\mathrm{reach}(l, s, \sigma_2, \theta_2)$.

□

Corollary A.7. *For well-formed configurations* $(\sigma_1 : \theta_1 : s)$ *and* $(\sigma_2 : \theta_2 : s)$*, if* $(\sigma_1 : \theta_1 : s) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s)$*, then* $(\sigma_1 : \theta_1 : s) \overset{\mathrm{rch}}{\sim} (\sigma_2 : \theta_2 : s)$*.*

Proof. It is a direct consequence of lemmas A.5, A.6 and the definition of $\overset{\mathrm{rch}}{\sim}$. □

While the following lemma directly refers to the notion of well-formedness of configurations, it is not required to describe in detail such notion in order to gain confidence about the following statement and its proof, since they are intuitive enough (for details about well-formedness, we refer the reader to [22]). Also, the lemma will allow us to extend the mentioned progress property for $\overset{L}{\mapsto}$ to the semantics obtained adding $\overset{GC}{\mapsto}$. In particular, it will guarantee that the introduced notion of observations over programs is well-defined also for $\overset{L}{\mapsto} \cup \overset{GC}{\mapsto}$, allowing us to state the desired correctness for $\overset{GC}{\mapsto}$.

Lemma A.8. *For a well-formed configuration* $(\sigma_1 : \theta_1 : s)$*, if* $(\sigma_1 : \theta_1 : s) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s)$*, for some configuration* $(\sigma_2 : \theta_2 : s)$*, then* $(\sigma_2 : \theta_2 : s)$ *is well-formed.*

Proof. From the definition of $\overset{GC}{\mapsto}$, it follows that the step does not change the instruction term. Also, by the previous lemmas, it follows that $\overset{GC}{\mapsto}$ does not introduce dangling pointers. They also state that $\overset{GC}{\mapsto}$ does not modify the stores in any other way, besides removing garbage. Then, it must be the case that also $(\sigma_2 : \theta_2 : s)$ is well-formed. □

Lemma A.9. *Over a well-formed configuration* $(\sigma : \theta : s)$*, only a finite number of* $\overset{GC}{\mapsto}$ *steps can be applied.*

Proof. By Definition 3.2 and $\overset{GC}{\mapsto}$, if

$$(\sigma : \theta : s) \overset{GC}{\mapsto} (\sigma' : \theta' : s)$$

then it must be the case that either σ' or θ' is a proper subset of σ or θ, respectively. Then, being the stores partial finite functions, it is clear that GC can be performed at most a finite number of steps. □

The following lemma is a useful tool taken from [18]. It codifies a simple intuition of plain GC: it must be possible to postpone any GC step, without changing the observations of the program. In its statement we use the fact that $\overset{GC}{\mapsto}$ does not change the instruction term.

Lemma A.10 (Postponement). *For a given well-formed configuration* $(\sigma_1 : \theta_1 : s_1)$*, if*

$$(\sigma_1 : \theta_1 : s_1) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s_1) \overset{L}{\mapsto} (\sigma_3 : \theta_3 : s_2).$$

then $\exists(\sigma_4 : \theta_4 : s_2)$ *such that:*

$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{L}}{\mapsto} (\sigma_4 : \theta_4 : s_2) \overset{\mathsf{GC}}{\mapsto} (\sigma_3' : \theta_3' : s_2)$$

where $(\sigma_3 : \theta_3 : s_2) \overset{\mathsf{rch}}{\sim} (\sigma_3' : \theta_3' : s_2)$.

Proof. We will follow the modular structure of $\overset{\mathsf{L}}{\mapsto}$ to reason over the step that transforms $(\sigma_2 : \theta_2 : s_1)$ into $(\sigma_3 : \theta_3 : s_2)$:

- *The computation does not change bindings from a store or dereferences locations*: then it must be the case that every information from the stores is already put into the instruction term s_1 so as to make the computation from $\overset{\mathsf{L}}{\mapsto}$ viable, without regard to the content of the stores. Then, the hypothesis can be rewritten as:
$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{GC}}{\mapsto} (\sigma_2 : \theta_2 : s_1) \overset{\mathsf{L}}{\mapsto} (\sigma_2 : \theta_2 : s_2)$$
If we take $(\sigma_4 : \theta_4 : s_2) = (\sigma_1 : \theta_1 : s_2)$, then we can assert that:
$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{L}}{\mapsto} (\sigma_1 : \theta_1 : s_2) = (\sigma_4 : \theta_4 : s_2)$$
where we exploited the fact that, for the previous $\overset{\mathsf{L}}{\mapsto}$ step to be performed, the actual content of the stores does not affect the applicability and the outcome of said computation. Then, by Lemma A.1, if a binding was ready to be collected in $(\sigma_1 : \theta_1 : s_1)$ it will remain in that state in $(\sigma_1 : \theta_1 : s_2)$. So, by the non-deterministic nature of $\overset{\mathsf{GC}}{\mapsto}$, we could ask for it to remove the same bindings that changed the stores from $(\sigma_1 : \theta_1 : s_1)$ into the stores from $(\sigma_2 : \theta_2 : s_1)$. Hence, it must be the case that $(\sigma_1 : \theta_1 : s_2) \overset{\mathsf{GC}}{\mapsto} (\sigma_2 : \theta_2 : s_2)$ holds. We obtained:
$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{L}}{\mapsto} (\sigma_1 : \theta_1 : s_2) \overset{\mathsf{GC}}{\mapsto} (\sigma_2 : \theta_2 : s_2)$$
Finally, $(\sigma_3 : \theta_3 : s_2) \overset{\mathsf{rch}}{\sim} (\sigma_3' : \theta_3' : s_2)$ because
$$(\sigma_3 : \theta_3 : s_2) = (\sigma_2 : \theta_2 : s_2) = (\sigma_3' : \theta_3' : s_2)$$
- *The computation changes or dereferences locations from* σ_1: we would need to do case analysis on each computation that interacts with the value store. As an example, let us consider the implicit dereferencing of a reference to σ_1. That is, the $\overset{\mathsf{L}}{\mapsto}$ step should be:
$$(\sigma_2 : \theta_2 : E[\![\overset{s_1}{\equiv}\!\! r\ ]\!]) \overset{\mathsf{L}}{\mapsto} (\sigma_2 : \theta_2 : E[\![\overset{s_2}{\equiv}\!\! \sigma_2(r)\ ]\!])$$
Then, the hypothesis can be rewritten as:
$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{GC}}{\mapsto} (\sigma_2 : \theta_2 : s_1) \overset{\mathsf{L}}{\mapsto} (\sigma_2 : \theta_2 : s_2)$$
If we take $(\sigma_4 : \theta_4 : s_3) = (\sigma_1 : \theta_1 : s_2)$, we can assert that:
$$(\sigma_1 : \theta_1 : E[\![\overset{s_1}{\equiv}\!\! r\ ]\!]) \overset{\mathsf{L}}{\mapsto} (\sigma_1 : \theta_1 : E[\![\overset{s_2}{\equiv}\!\! \sigma_2(r)\ ]\!])$$
because r is reachable in $(\sigma_1 : \theta_1 : s_1)$, and the $\overset{\mathsf{GC}}{\mapsto}$ step from the hypothesis preserves its binding, in the sense expressed in Lemma A.5: hence, if it was possible to perform the dereferencing in $(\sigma_2 : \theta_2 : s_1)$ (by hypothesis), it must be possible to perform it in $(\sigma_1 : \theta_1 : s_1)$, obtaining the same result. Finally, by preservation of bindings ready for collection after a $\overset{\mathsf{L}}{\mapsto}$ step, Lemma A.1, and the non-deterministic behaviour of $\overset{\mathsf{GC}}{\mapsto}$, we could ask for the GC step to remove exactly the necessary bindings so that $(\sigma_1 : \theta_1 : s_2) \overset{\mathsf{GC}}{\mapsto} (\sigma_2 : \theta_2 : s_2)$ holds. We obtained:
$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{L}}{\mapsto} (\sigma_1 : \theta_1 : s_2) \overset{\mathsf{GC}}{\mapsto} (\sigma_2 : \theta_2 : s_2)$$
Finally, $(\sigma_3 : \theta_3 : s_2) \overset{\mathsf{rch}}{\sim} (\sigma_3' : \theta_3' : s_2)$ because
$$(\sigma_3 : \theta_3 : s_2) = (\sigma_2 : \theta_2 : s_2) = (\sigma_3' : \theta_3' : s_2)$$
- *The computation changes or dereferences locations from* θ_1: we would need to do case analysis on each computation that interacts with θ_1. As an example, let us consider table allocation. Then, the hypothesis can be rewritten as:
$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{GC}}{\mapsto} (\sigma_2 : \theta_2 : s_1) \overset{\mathsf{L}}{\mapsto} (\sigma_2 : \theta_2 \uplus \{(tid, t)\} : E[\![\overset{s_2}{\equiv}\!\! tid\ ]\!])$$
for an adequate internal representation of a table, t, and table identifier tid, that, for our purposes, it will be useful if $tid \notin dom(\theta_1)$. If it is not the case, we can continue with our reasoning over an appropriate α-converted configuration, where the references in $(\sigma_1 : \theta_1 : s_1)$ are consistently changed so as to make $tid \notin dom(\theta_1)$. It is because of cases like this one that we cannot assert a stronger postponement statement, as the one in [18]: we are not talking about convergence towards a single configuration; we need to think in terms of $\overset{\mathsf{rch}}{\sim}$-equivalent configurations.
If we take
$$(\sigma_4 : \theta_4 : s_3) = (\sigma_1 : \theta_1 \uplus \{(tid, t)\} : s_2)$$
we know that:
$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{L}}{\mapsto} (\sigma_1 : \theta_1 \uplus \{(tid, t)\} : s_2)$$
where we can ask for the instruction term to be exactly $s_2 = E[\![\ tid\ ]\!]$. By Lemma A.1 we know that every binding which is ready for collection in $(\sigma_1 : \theta_1 : s_1)$ is in the same state in $(\sigma_1 : \theta_1 \uplus \{(tid, t)\} : s_2)$. Even more, such bindings just belongs to σ_1 or θ_1. Then, by the non-deterministic nature of $\overset{\mathsf{GC}}{\mapsto}$ we could ask for it to remove just the necessary bindings so as to make true
$$(\sigma_1 : \theta_1 \uplus \{(tid, t)\} : s_2) \overset{\mathsf{GC}}{\mapsto} (\sigma_2 : \theta_2 \uplus \{(tid, t)\} : s_2).$$
Then, the following holds:
$$(\sigma_1 : \theta_1 : s_1) \overset{\mathsf{L}}{\mapsto} \ldots \overset{\mathsf{GC}}{\mapsto} (\sigma_2 : \theta_2 \uplus \{(tid, t)\} : s_2)$$
Finally, $(\sigma_3 : \theta_3 : s_2) \overset{\mathsf{rch}}{\sim} (\sigma_3' : \theta_3' : s_2)$ because
$$(\sigma_3 : \theta_3 : s_2) = (\sigma_2 : \theta_2 \uplus \{(tid, t)\} : s_2) = (\sigma_3' : \theta_3' : s_2)$$

□

Correctness of simple GC The expected statement of GC correctness should mention that, for a given configuration, the observations under $\overset{\mathsf{L}}{\mapsto}$ should be the same that those under $\overset{\mathsf{L+GC}}{\mapsto}$ (*i.e.*, $\overset{\mathsf{L}}{\mapsto} \cup \overset{\mathsf{GC}}{\mapsto}$). However, under $\overset{\mathsf{L}}{\mapsto}$ and $\overset{\mathsf{L+GC}}{\mapsto}$ we expect the observations to be just a singleton: the programs either diverge or reach to a end, returning some results or an error object. Giving this observation, we could change the statement of GC correctness to reach to a property that can be proved with less effort: given a configuration, under $\overset{\mathsf{L}}{\mapsto}$ its execution reaches to a end, if and only if its execution reaches to an end under $\overset{\mathsf{L+GC}}{\mapsto}$, and, in both cases, what is returned (either values or error objects) is the same.

The stated property will imply the preservation of observations, as defined in Definition 4.3, but it will allow us to focus just on convergent computations; preservation of divergent computations will be a consequence of the double implication structure of the statement:

Theorem A.11 (GC correctness). *For a given well-formed configuration* $\sigma : \theta : s$,

$$(\sigma : \theta : s) \Downarrow_{\overset{\mathsf{L}}{\mapsto}} (\sigma' : \theta' : s') \Leftrightarrow (\sigma : \theta : s) \Downarrow_{\overset{\mathsf{L+GC}}{\mapsto}} (\sigma'' : \theta'' : s'')$$

and $\mathsf{result}(\sigma' : \theta' : s') = \mathsf{result}(\sigma'' : \theta'' : s'')$.

Proof. Let us assume that $(\sigma : \theta : s) \Downarrow_{\overset{L}{\mapsto}} (\sigma' : \theta' : s')$. Then $(\sigma' : \theta' : s')$ is a final configuration where result is defined. Because $\overset{L}{\mapsto} \subseteq \overset{L+GC}{\mapsto}$, it is always possible to emulate the previous trace by not using $\overset{GC}{\mapsto}$ steps. Then, $(\sigma : \theta : s) \Downarrow_{\overset{L+GC}{\mapsto}} (\sigma' : \theta' : s')$, where it follows that, in both cases, the computations returns the same, under $\overset{L+GC}{\mapsto}$ and $\overset{L}{\mapsto}$.

On the other hand, let us assume that

$$(\sigma : \theta : s) \Downarrow_{\overset{L+GC}{\mapsto}} (\sigma' : \theta' : s')$$

Then, it must be the case that there exist a finite trace of computation steps, as follows:

$$(\sigma : \theta : s) \overset{L+GC}{\mapsto} (\sigma_1 : \theta_1 : s_1) \overset{L+GC}{\mapsto} \dots \overset{L+GC}{\mapsto} (\sigma_n : \theta_n : s_n)$$

where $(\sigma_n : \theta_n : s_n) = (\sigma' : \theta' : s')$ is a final configuration over which result is defined.

By applying inductive reasoning over the number of computation steps and the Postponement Lemma A.10, it can be shown that we can rewrite the previous trace as follows:

$$(\sigma : \theta : s) \overset{L}{\mapsto} \dots \overset{L}{\mapsto} (\sigma_{i'} : \theta_{i'} : s_{i'}) \overset{GC}{\mapsto} \dots \overset{GC}{\mapsto} (\sigma_{n'} : \theta_{n'} : s_{i'})$$

where every computation that does not involve GC is performed at the beginning. We obtained a convergent trace consisting only in $\overset{L}{\mapsto}$ steps. That is:

$$(\sigma : \theta : s) \Downarrow_{\overset{L}{\mapsto}} (\sigma_{i'} : \theta_{i'} : s_{i'})$$

What remains is to see if the result is also preserved. To that end, note that the postponement lemma used also tells us that

$$(\sigma_{n'} : \theta_{n'} : s_{i'}) \overset{\mathsf{rch}}{\sim} (\sigma_n : \theta_n : s_n)$$

Then, because final configurations which are $\overset{\mathsf{rch}}{\sim}$ represent the same result, according to Lemma A.4, it follows that

$$\mathsf{result}(\sigma_{n'} : \theta_{n'} : s_{i'}) = \mathsf{result}(\sigma_n : \theta_n : s_n)$$

Finally, because $\overset{\mathsf{rch}}{\sim}$ is closed under $\overset{GC}{\mapsto}$ steps, Lemma A.7, it must be the case that:

$$(\sigma_{i'} : \theta_{i'} : s_{i'}) \overset{\mathsf{rch}}{\sim} (\sigma_{n'} : \theta_{n'} : s_{i'})$$

Hence,

$$\mathsf{result}(\sigma_{i'} : \theta_{i'} : s_{i'}) = \mathsf{result}(\sigma_{n'} : \theta_{n'} : s_{i'}) = \mathsf{result}(\sigma_n : \theta_n : s_n)$$

□

An immediate corollary of the previous theorem is that, under $\overset{L+GC}{\mapsto}$, the set of observations over programs is a singleton, even under the non-determinism nature of $\overset{GC}{\mapsto}$:

Corollary A.12. *For a well-formed configuration* $\sigma : \theta : s$, $|\mathsf{obs}(\sigma : \theta : s, \overset{L+GC}{\mapsto})| = 1$

Proof. It follows immediately from the previous theorem and the determinism of programs under $\overset{L}{\mapsto}$. □

Now, based on the observations of the beginning of this section, we can state an equivalent version of correctness for simple GC, but in terms of the notion of observations previously defined:

Corollary A.13 (GC correctness). *For a given well-formed configuration* $\sigma : \theta : s$,

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

Proof. It follows directly from the previous corollary, together with Theorem A.11. Then, if $\mathsf{result}(\sigma', \theta', s') \in \mathsf{obs}(\sigma : \theta : s, \overset{L}{\mapsto})$, for $(\sigma : \theta : s) \Downarrow_{\overset{L}{\mapsto}} (\sigma' : \theta' : s')$, by theorem A.11, the previous occurs if and only if $(\sigma : \theta : s) \Downarrow_{\overset{L+GC}{\mapsto}} (\sigma'' : \theta'' : s'')$, where

$$\mathsf{result}(\sigma', \theta', s') = \mathsf{result}(\sigma'', \theta'', s'')$$

Hence $\mathsf{result}(\sigma', \theta', s') \in \mathsf{obs}(\sigma : \theta : s, \overset{L+GC}{\mapsto})$, and we can conclude that $\mathsf{obs}(\sigma : \theta : s, \overset{L}{\mapsto}) = \mathsf{obs}(\sigma : \theta : s, \overset{L+GC}{\mapsto})$. The converse is analogous.

If $\bot \in \mathsf{obs}(\sigma : \theta : s, \overset{L}{\mapsto})$, by correctness of GC, it must happen if and only if $\bot \in \mathsf{obs}(\sigma : \theta : s, \overset{L+GC}{\mapsto})$, and because of the determinism of both, $\overset{L+GC}{\mapsto}$ and $\overset{L}{\mapsto}$, we can conclude that:

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

The converse is analogous. □