Understanding Lua's Garbage Collection
Towards a Formalized Static Analyzer

22nd International Symposium on Principles and Practice of Declarative Programming

Mallku Soldevila[1], Beta Ziliani[1] and Daniel Fridlender[2]

[1]FAMAF/UNC and CONICET, [2]FAMAF/UNC

Summary

- About Lua
- Garbage collection in Lua:
  - A first look into garbage collection
  - Approach and contributions of the present work
  - Formalization challenges of GC in Lua
  - Formal semantics of GC
- Mechanization
- Application: LuaSafe
- Future work

About Lua

About Lua

- Extension and data-entry language.
- Features:
  - Procedural programming with data-description facilities (only one structured data-type: *tables*)
  - Fast development: dynamic typing, automatic memory management.
  - Small implementation (∼216KB; for reduced embedding cost).
  - *meta-tables*: meta-programming mechanism to extend the semantics of programming constructions.

About Lua

- Projects using Lua:
  - Heavily used in the video game industry: mobile games, “AAA” games and game engines.
  - Other scriptable software: **Adobe Photoshop Lightroom**, **LuaTex**, **VLC media player**, **Wireshark**,...
  - `www.lua.org/uses.html`.

A first look into garbage collection

A first look into garbage collection

- Lua 5.2 implements 2 garbage collectors based on reachability:
  - *mark-and-sweep*
  - *generational*

A first look into garbage collection

- Lua 5.2 implements 2 garbage collectors based on reachability:
  - *mark-and-sweep*
  - *generational*
- Includes 2 interfaces with the garbage collector:
  - *finalizers*
    - Custom routines for the release of external resources used by the program.

A first look into garbage collection

- Lua 5.2 implements 2 garbage collectors based on reachability:
  - *mark-and-sweep*
  - *generational*
- Includes 2 interfaces with the garbage collector:
  - *finalizers*
    - Custom routines for the release of external resources used by the program.
  - *weak tables*
    - A table whose keys and/or values are referred by weak references.

A first look into garbage collection

Weak tables

```
    creates a cache of closures
local cache1 = {[1] = function() return 1 end,
                [2] = function() return 2 end,
                [3] = function() return 3 end}

    references to closures in cache1
local obj = {method = cache1[1], attr = {}}
local cache2 = {[1] = cache1[2]}

    values are now ref. by weak references (weak tables)
setmetatable(cache1, { __mode = "v"})
setmetatable(cache2, { __mode = "v"})

    weak refs. are not taken into account by the garbage collector
cache1[1]()   ← which field is guaranteed to be accessible?
cache1[2]()
cache1[3]()
```

A first look into garbage collection

Finalizers

```
local a, b, c = {}, {}, {}      ← 3 empty tables
c.__gc = function (t)           ← field in c with key "__gc"
             d = t              and a procedure as value: the finalizer
             print ('bye', t)
         end
setmetatable(a, c)      a and b are marked for finalization
setmetatable(b, c)      finalizer: c.__gc

a = nil       a and b not reachable → they can be finalized
b = nil

collectgarbage ()      garbage collector invokes finalizers

print (d)      a or b permanently resurrected, preventing GC
               (depends on finalization order)
```

Approach and contributions of the present work

Approach and contributions of the present work

- A mathematical model of Lua's GC.

Approach and contributions of the present work

- A mathematical model of Lua's GC.
- A theoretical framework, to express and prove properties of our model.

Approach and contributions of the present work

- A mathematical model of Lua's GC.
- A theoretical framework, to express and prove properties of our model.
- A mechanization in PLT Redex.

Approach and contributions of the present work

- A mathematical model of Lua's GC.
- A theoretical framework, to express and prove properties of our model.
- A mechanization in PLT Redex.
- **LuaSafe**, to help uncover non-deterministic behavior introduced by *weak tables*.

Formalization challenges of GC in Lua

Formalization challenges of GC in Lua

Particularities/challenges in Lua's finalizers semantics:

- Inverse chronological order of finalization.

Formalization challenges of GC in Lua

Particularities/challenges in Lua's finalizers semantics:

- Inverse chronological order of finalization.
- Avoids indestructible objects.

Formalization challenges of GC in Lua

Particularities/challenges in Lua's finalizers semantics:

- Inverse chronological order of finalization.
- Avoids indestructible objects.
- Better support for common data-structures implemented with weak tables (*e.g.*, *property tables*).

Formalization challenges of GC in Lua

Particularities/challenges in Lua's weak tables semantics:

- *Ephemerons* (similar to key/values weak references present in the GHC).
  - See *Eliminating Cycles in Weak Tables*, Alexandra Barros and Roberto Ierusalimschy. 2008.

Formalization challenges of GC in Lua

Particularities/challenges in Lua's weak tables semantics:

- *Ephemerons* (similar to key/values weak references present in the GHC).
  - See *Eliminating Cycles in Weak Tables*, Alexandra Barros and Roberto Ierusalimschy. 2008.
- Better support for common data-structures implemented with weak tables (*e.g.*, *property tables*).

Formalization challenges of GC in Lua

Interaction between interfaces

- Finalization checks for reachability taking into account weak tables semantics.

Formalization challenges of GC in Lua

Interaction between interfaces

- Finalization checks for reachability taking into account weak tables semantics.
- Weak tables are cleaned taking into account finalization order.

Formal semantics of GC

Formal semantics of GC

- Extends a previous formalized dynamic semantics for Lua 5.2:
  - Small-steps operational semantics, with concepts from reduction semantics with evaluation contexts.

Formal semantics of GC

- Extends a previous formalized dynamic semantics for Lua 5.2:
  - Small-steps operational semantics, with concepts from reduction semantics with evaluation contexts.
- Specification of the behavior of a correct garbage collector for Lua (abstracted into a meta-function gc_{fin_weak}):
  - Suitable notion of reachability for Lua.

Formal semantics of GC

- Extends a previous formalized dynamic semantics for Lua 5.2:
  - Small-steps operational semantics, with concepts from reduction semantics with evaluation contexts.
- Specification of the behavior of a correct garbage collector for Lua (abstracted into a meta-function gc_{fin_weak}):
  - Suitable notion of reachability for Lua.
  - Specifies fields of weak tables than can be removed.

Formal semantics of GC

- Extends a previous formalized dynamic semantics for Lua 5.2:
  - Small-steps operational semantics, with concepts from reduction semantics with evaluation contexts.
- Specification of the behavior of a correct garbage collector for Lua (abstracted into a meta-function gc_{fin_weak}):
  - Suitable notion of reachability for Lua.
  - Specifies fields of weak tables than can be removed.
  - Identifies the next table to be finalized.

Formal semantics of GC

- Extends a previous formalized dynamic semantics for Lua 5.2:
  - Small-steps operational semantics, with concepts from reduction semantics with evaluation contexts.
- Specification of the behavior of a correct garbage collector for Lua (abstracted into a meta-function gc_{fin_weak}):
  - Suitable notion of reachability for Lua.
  - Specifies fields of weak tables than can be removed.
  - Identifies the next table to be finalized.
  - Specifies interaction between both interfaces.

Formal semantics of GC

Non-deterministic execution steps:

$$\frac{(\sigma', f, t) = \mathsf{gc}_{\mathsf{fin\_weak}}(\sigma, \mathsf{E}[\![\mathsf{s}]\!])}{\sigma : \mathsf{E}[\![\mathsf{s}]\!] \overset{\mathsf{GC+W+F}}{\longmapsto} \sigma' : \mathsf{E}[\![f(t); \mathsf{s}]\!]}$$

Formal semantics of GC

Framework for formal reasoning about GC and sanity-check

- Defines observations over programs (non-termination or returned values), *garbage*, etc.

Formal semantics of GC

Framework for formal reasoning about GC and sanity-check

- Defines observations over programs (non-termination or returned values), *garbage*, etc.

- *GC-correctness*: for a given program p, the observations are preserved by GC-steps without interfaces to the garbage collector:

$$obs(p, \mapsto) = obs(p, \mapsto \cup \stackrel{GC}{\mapsto})$$

Formal semantics of GC

Framework for formal reasoning about GC and sanity-check

- Defines observations over programs (non-termination or returned values), *garbage*, etc.

- *GC-correctness*: for a given program p, the observations are preserved by GC-steps without interfaces to the garbage collector:

$$obs(p, \mapsto) = obs(p, \mapsto \cup \overset{\mathsf{GC}}{\mapsto})$$

- We are setting a framework for future discussion on static analysis of Lua programs and GC.

Mechanization

Mechanization.

- Implemented using Redex.

Mechanization.

- Implemented using Redex.
- Tested against Lua 5.2's test suite: 1444 LOCS (from 6902 LOCS).
  - gc.lua: 125 LOCs from 445 LOCS.

Mechanization.

- Implemented using Redex.
- Tested against Lua 5.2's test suite: 1444 LOCS (from 6902 LOCS).
  - gc.lua: 125 LOCs from 445 LOCS.
- Why?
  - Features not covered by our formalization (mostly, library services) and implementation details (generation of bytecode, performance, etc).

Mechanization.

- Implemented using Redex.
- Tested against Lua 5.2's test suite: 1444 LOCS (from 6902 LOCS).
  - gc.lua: 125 LOCs from 445 LOCS.
- Why?
  - Features not covered by our formalization (mostly, library services) and implementation details (generation of bytecode, performance, etc).
  - gc.lua: features not covered by our model (mainly parameters of the collectgarbage service) and manipulation of large data-structures (performance).

Mechanization.

- Implemented using Redex.
- Tested against Lua 5.2's test suite: 1444 LOCS (from 6902 LOCS).
  - gc.lua: 125 LOCs from 445 LOCS.
- Why?
  - Features not covered by our formalization (mostly, library services) and implementation details (generation of bytecode, performance, etc).
  - gc.lua: features not covered by our model (mainly parameters of the collectgarbage service) and manipulation of large data-structures (performance).
- Every line of code of the test suite that falls within the scope of this work successfully passes the tests, except performance tests.

Mechanization.

- Implemented using Redex.
- Tested against Lua 5.2's test suite: 1444 LOCS (from 6902 LOCS).
    - gc.lua: 125 LOCs from 445 LOCS.
- Why?
    - Features not covered by our formalization (mostly, library services) and implementation details (generation of bytecode, performance, etc).
    - gc.lua: features not covered by our model (mainly parameters of the collectgarbage service) and manipulation of large data-structures (performance).
- Every line of code of the test suite that falls within the scope of this work successfully passes the tests, except performance tests.
- Mechanization available at `github.com/Mallku2/lua-gc-redex-model`.

Application: LuaSafe

Application: LuaSafe

- Problem: $\exists p, obs(p, \mapsto) \neq obs(p, \mapsto \cup \overset{\text{GC+W+F}}{\mapsto})$

Application: LuaSafe

- Problem: $\exists p, obs(p, \mapsto) \neq obs(p, \mapsto \cup \overset{\text{GC+W+F}}{\mapsto})$
- Let $P_{safe} = \{p \mid obs(p, \mapsto) = obs(p, \mapsto \cup \overset{\text{GC+W+F}}{\mapsto})\}$

Application: LuaSafe

- Problem: $\exists p, obs(p, \mapsto) \neq obs(p, \mapsto \cup \overset{\text{GC+W+F}}{\mapsto})$
- Let $P_{safe} = \{p \mid obs(p, \mapsto) = obs(p, \mapsto \cup \overset{\text{GC+W+F}}{\mapsto})\}$
- For a given program p that uses weak tables, recognizing $p \in P_{safe}$ requires:
  - Type information.

Application: LuaSafe

- Problem: $\exists p, obs(p, \mapsto) \neq obs(p, \mapsto \cup \overset{GC+W+F}{\mapsto})$
- Let $P_{safe} = \{p \mid obs(p, \mapsto) = obs(p, \mapsto \cup \overset{GC+W+F}{\mapsto})\}$
- For a given program p that uses weak tables, recognizing $p \in P_{safe}$ requires:
  - Type information.
  - *Weakness* of each table.

Application: LuaSafe

- Problem: $\exists p, obs(p, \mapsto) \neq obs(p, \mapsto \cup \overset{\text{GC+W+F}}{\mapsto})$
- Let $P_{safe} = \{p \mid obs(p, \mapsto) = obs(p, \mapsto \cup \overset{\text{GC+W+F}}{\mapsto})\}$
- For a given program p that uses weak tables, recognizing $p \in P_{safe}$ requires:
  - Type information.
  - *Weakness* of each table.
  - A syntactic approximation of the *reachability tree*.

Application: LuaSafe

- Problem: $\exists p, obs(p, \mapsto) \neq obs(p, \mapsto \cup \overset{GC+W+F}{\mapsto})$
- Let $P_{safe} = \{p \mid obs(p, \mapsto) = obs(p, \mapsto \cup \overset{GC+W+F}{\mapsto})\}$
- For a given program p that uses weak tables, recognizing $p \in P_{safe}$ requires:
  - Type information.
  - *Weakness* of each table.
  - A syntactic approximation of the *reachability tree*.
- We will try a *best-effort* approach to recognize membership to P_{safe}: no changes to user code needed.

Application: LuaSafe

Type system

With our type system, we need to:

- Recognize field accessed (and types of key and value), from a given table.

Application: LuaSafe

Type system

With our type system, we need to:

- Recognize field accessed (and types of key and value), from a given table.
- Recognize *weakness* of the table being accessed.

Application: LuaSafe

Type system

With our type system, we need to:

- Recognize field accessed (and types of key and value), from a given table.
- Recognize *weakness* of the table being accessed.
- Being able to reason about types of several (possible recursive) data structures commonly implemented using Lua tables, and related idioms: objects, trees, lists, etc.

Application: LuaSafe

Type system

- Primitive types (*prmt*): **nil**, **num**, **bool**, **str**

Application: LuaSafe

Type system

- Primitive types (*prmt*): **nil**, **num**, **bool**, **str**
- *Singleton* types (st): $\langle v_{\text{st}} : prmt \rangle$
  - $v_{\text{st}} \in \{\textbf{nil}\} \cup string \cup boolean \cup number$

Application: LuaSafe

Type system

- Primitive types (*prmt*): **nil**, **num**, **bool**, **str**
- *Singleton* types (st): $\langle v_{\text{st}} : prmt \rangle$
  - $v_{\text{st}} \in \{\textbf{nil}\} \cup string \cup boolean \cup number$
- Table type: $\{\ [\ st\ ] : t\ \ldots\ \}\ wkness$
  - $wkness \in \{\textbf{strong}, \textbf{wk}, \textbf{wv}, \textbf{wkv}\}$

Application: LuaSafe

Type system

- Primitive types (*prmt*): **nil**, **num**, **bool**, **str**
- *Singleton* types (st): $\langle v_{\text{st}} : \mathit{prmt} \rangle$
  - $v_{\text{st}} \in \{\textbf{nil}\} \cup \mathit{string} \cup \mathit{boolean} \cup \mathit{number}$
- Table type: $\{\ [\ \mathit{st}\ ] : t\ ...\ \}\ \mathit{wkness}$
  - $\mathit{wkness} \in \{\textbf{strong}, \textbf{wk}, \textbf{wv}, \textbf{wkv}\}$
  - With singleton types, determining weakness of a table reduces to a type checking problem.

Application: LuaSafe

Type system

- Primitive types (*prmt*): **nil**, **num**, **bool**, **str**
- *Singleton* types (st): $\langle v_{\text{st}} : prmt \rangle$
  - $v_{\text{st}} \in \{\textbf{nil}\} \cup string \cup boolean \cup number$
- Table type: $\{\ [\ st\ ] : t\ \ldots\ \}\ wkness$
  - $wkness \in \{\textbf{strong}, \textbf{wk}, \textbf{wv}, \textbf{wkv}\}$
  - With singleton types, determining weakness of a table reduces to a type checking problem.
- Recursive types: $\boldsymbol{\mu}\ y \boldsymbol{.}\ t$
  - Just for table types.

Application: LuaSafe

Type system

Type inference

- Based on previous work on type inference for JavaScript[1].

[1] Anderson, Christopher and Giannini, Paola and Drossopoulou, Sophia. *Towards Type Inference for JavaScript*. In Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05.

Application: LuaSafe

Type system

Type inference

- Based on previous work on type inference for JavaScript[1].
- Type inferred: solution of a set of constraints about sub-typing relation between possible types for the expressions of the program.

[1] Anderson, Christopher and Giannini, Paola and Drossopoulou, Sophia. *Towards Type Inference for JavaScript*. In Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05.

Application: LuaSafe

Syntactic approximation of the reachability tree.

For a given point into a program, we use the set of *reaching definitions* to determine reachability.

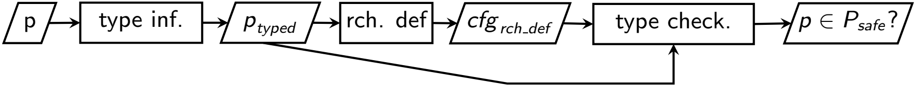
Application: LuaSafe

Syntactic approximation of the reachability tree.

For a given point into a program, we use the set of *reaching definitions* to determine reachability.

```
local cache1 = {[1] = function() return 1 end,
                [2] = function() return 2 end,
                [3] = function() return 3 end}
   reach. def: { cache1 = { [1] = ... } }
local obj = {method = cache1[1], attr = {}}
  {cache1 = { [1] = ... },
   obj = method = cache1[1], ...} }
local cache2 = {[1] = cache1[2]}
  {cache1 = { [1] = ... },
   obj = {method = cache1[1], ...},
   cache2 = {[1] = cache1[2]} }
setmetatable(cache1, { __mode = "v"})
setmetatable(cache2, { __mode = "v"})
cache1[1]()      ← same set of reach. defs is preserved up to
cache1[2]()      this point
cache1[3]()
```

Application: LuaSafe



Application: LuaSafe

```
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]()
```

```
"Access to: "
'cache1 [ 2 ]
"may exhibit nondeterministic behavior"
"Access to: "
'cache1 [ 3 ]
"may exhibit nondeterministic behavior"
```

Figure: Implementation of a simple cache.

Future work

Future work

- Include missing Lua features.
- Redex $\rightarrow$ Coq:
  - Machine-checked proofs.
  - Extraction of a verified interpreter.
- Improve LuaSafe:
  - Soundness of static analysis.
  - Improve type inference.
  - Better syntactic approx. of reach. tree.
  - Improve performance.
- Recognition of semantic garbage based on type checking.

Thanks!