Seminario LoRel

Semántica operacional y su aplicación para el estudio de recolección de basura, en Lua 5.2

Mallku Ernesto Soldevila Raffa

Director: Dr. Daniel Fridlender
Co-director: Dr. Beta Ziliani

FAMAF - Universidad Nacional de Córdoba

Summary

- About Lua
- Why do we need a formal semantics for Lua?
- Formal Semantics
- Mechanization
- Practical application
- Future work

About Lua

About Lua

- Extension and data-entry language:
  - Small language, small implementation.
  - Should be extensible.
  - No need for mechanisms for programming-in-the-large.
  - Good data-description facilities.
  - Clear and simple syntax.
- Features:
  - Procedural programming with data-description facilities (only one structured data-type: *tables*)
  - Features for fast development: dynamic typing, automatic memory management.
  - Metaprogramming mechanisms: extension of 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`.

Who could benefit from a formal semantics for Lua?

- Developers of tools for code analysis and language extensions.
- Lua programmers.

Why do we need a formalized semantics of Lua?

Developers of tools for code analysis and language extensions

- Formal proofs of soundness, strengthen the possibilities of static analysis.

- Tools for code analysis:
  - **Luacheck**[1]
  - **Lua Inspect**[2]
  - **LuaSafe**[3]
  - More on `lua-users.org/wiki/ProgramAnalysis`.

- Language extensions
  - **Ravi**[4]
  - **Typed Lua**[5]

---

[1] `https://github.com/mpeterv/luacheck`
[2] `http://lua-users.org/wiki/LuaInspect`
[3] `https://github.com/Mallku2/luasafe-redex`
[4] http://ravilang.github.io/
[5] A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy. A formalization of Typed Lua. In DLS '15, 2015

Why do we need a formalized semantics of Lua?

Lua programmers

- Developers could benefit from it: concise formal description of the semantics of the whole language (no core language approach required for Lua).
- The project can benefit from having people of differente areas testing it.

Formal semantics

Formal semantics

- Design criteria
- Semantics of stateless constructions
- Semantics of state
- Semantics of programs
- Library services
- Metatables
- Garbage collection

Formal semantics

Design criteria

- We are tackling the semantics of a *real* programming language, defined by its interpreters and reference manual (both unsuitable for formal reasoning).

Formal semantics

Design criteria

- We are tackling the semantics of a *real* programming language, defined by its interpreters and reference manual (both unsuitable for formal reasoning).
- We would like for our semantics to serve as a link between the informal understanding of Lua and its study through formal models.

Formal semantics

These lead us to wish for...

- Evidence of the correspondence between formal and informal Lua: testing, evident correspondence with the reference manual.

Formal semantics

These lead us to wish for...

- Evidence of the correspondence between formal and informal Lua: testing, evident correspondence with the reference manual.
- Semantics should make use of concepts already present in the mind of the developer.

Formal semantics

The model

- Concepts from small-steps operational semantics and reduction semantics with evaluation contexts.
  - **Small-step operational semantics**: the execution model of state change (to capture the intuition of the developer).
  - **Reduction semantics with evaluation contexts**: evaluation contexts and their several applications (easiness of description of context-sensitive semantics, modularity), environment using substitution function.

Formal semantics

Semantics of stateless constructions

syntax

$s ::= \textbf{if } e \textbf{ then } s \textbf{ else } s \mid \ldots$

$v ::= \textbf{nil} \mid \textbf{false} \mid \ldots$

$e ::= v \mid e\ op\ e \mid \ldots$

$op ::= + \mid - \mid * \mid / \mid \hat{} \mid \% \mid \ldots$

relations between terms (computations)

$$\frac{v \notin \{\textbf{nil}, \textbf{false}\}}{\textbf{if } v \textbf{ then } s_1 \textbf{ else } s_2 \ \rightarrow\ s_1} \qquad \frac{v \in \{\textbf{nil}, \textbf{false}\}}{\textbf{if } v \textbf{ then } s_1 \textbf{ else } s_2 \ \rightarrow\ s_2}$$

$$\frac{op \in \{+, -, *, /, \hat{}, \%\}}{v\ op\ e \ \rightarrow\ \delta(op, v, e)}$$

interpretation function

$\delta(op,\ n_1,\ n_2) = \hat{n}_1\ op\ \hat{n}_2, op \in \{+, -, *, /, \hat{}, \%\}$

Formal semantics

Semantics of state

syntax

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

computations

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

$$\frac{\sigma' = \sigma[\mathrm{r} := v]}{\sigma : r = v \ \rightarrow^{\sigma}\ \sigma' : ;}$$

Formal semantics

Semantics of programs

evaluation contexts

$$E ::= [\,] \mid \textbf{if } E \textbf{ then } s \textbf{ else } s \mid \textbf{local } x = \mathrm{E} \textbf{ in } s \mid x = E \mid E \textit{ binop } e \mid v \textit{ binop } E$$

embedding relations using evaluation contexts

$$\frac{t \rightarrow t'}{\sigma : E[\![t]\!] \mapsto \sigma : E[\![t']\!]}$$

$$\frac{\sigma : t \rightarrow^{\sigma} \sigma' : t'}{\sigma : E[\![t]\!] \mapsto \sigma' : E[\![t']\!]}$$

Formal semantics

Library services

- Specification of library services captured with the interpretation function (δ):

$$\frac{\mathsf{l} \in \{\mathsf{assert}, \mathsf{error}, \mathsf{pcall}, \mathsf{select}, ...\}}{\mathbf{\$builtIn}\ \mathsf{l}\ \mathbf{(}v_1, ..., v_\mathsf{n}\mathbf{)} \rightarrow \delta(\mathsf{l}, v_1, ..., v_\mathsf{n})}$$

$$\delta(\mathsf{select}, v_1, v_2, ..., v_n) = \begin{cases} \langle v_{\hat{v_1}+1}, ..., v_n \rangle & \text{if} \quad 1 \leq \hat{v_1} \leq n-1 \\ \langle\, \rangle & \text{if} \quad n \leq \hat{v_1} \\ \langle v_{n-1+|\hat{v_1}|}, ..., v_n \rangle & \text{if} \quad -(n-1) \leq \hat{v_1} \leq -1 \\ \langle n-1 \rangle & \text{if} \quad v_1 = "\#" \end{cases}$$

Formal semantics

Metatables

- An ordinary Lua table that defines the behaviour of a given value under certain special operations:

```
local t = {}
t()      >> attempt to call local 't' (a table value)
setmetatable(t, { __call = function () print("Callable!") end})
t()      >> Callable!
```

- Useful to develop DSLs.

Formal semantics

Metatables: formalization

- Being Lua an extensible language, it's not a surprise that it has being growing on metamethods: the semantics should be versatile in that aspect.

- The special operation is tagged:

$$\frac{\delta(\text{type}, v_1) \neq \text{``function''}}{\sigma : v_1\ (v_2, \ldots) \ \rightarrow^{\sigma}\ \sigma : (\!|\ v_1\ (v_2, \ldots)\ |\!)_{\textbf{WrongFunCall}}}$$

- The metatable mechanism solves the situation:

$$\frac{\begin{array}{c} v_3 = \text{indexmetatable}(v_1,\ \text{``\_\_call''},\ \sigma) \\ v_3 \notin \{\textbf{nil},\ \textbf{false}\} \end{array}}{\sigma : (\!|\ v_1\ (v_2, \ldots)\ |\!)_{\textbf{WrongFunCall}} \rightarrow^{\text{meta}}\ \sigma : v_3(v_1, v_2, \ldots)}$$

Formal semantics

Properties of $\mapsto$

- Formalization (on paper): 11 pages long, without garbage collection (not suitable for proofs on paper).

- No features to export to proof assistants: opportunity to work on Redex $\rightarrow$ Coq.

Properties of $\mapsto$ - Redex lightweight approach: random-testing with redex-check

Evidence for progress of $\mapsto$: for a $\vdash$ that should capture well-formedness of configurations, test its soundness property:

- For a given configuration $\sigma : s$, if $\vdash \sigma : s$, then it is a final configuration or it is an intermediate computation state.
- $5 * 10^6$ terms generated by redex-check.

Properties of $\mapsto$ - Redex lightweight approach: random-testing with redex-check

Even though the mechanization successfully passed the tests taken from Lua's test suite, redex-check showed that the mechanization still had flaws:

- Mostly, grammar ambiguities.
- Really useful to polish definition of well-formedness of configurations, for the semantics of a real programming language (tricky!).

Garbage collection

Formal semantics

Garbage collection (GC)

Lua 5.2 implements 2 garbage collectors based on reachability:

- *mark-and-sweep*
- *generational*

Garbage collection (GC)

Includes 2 interfaces with the garbage collector:

- *finalizers*:
  - Useful for helping in the proper disposal of external resources used by the program.
  - Chronological order of finalization, avoids indestructible objects.

Formal semantics

Garbage collection (GC)

Includes 2 interfaces with the garbage collector:

- *finalizers*:
  - Useful for helping in the proper disposal of external resources used by the program.
  - Chronological order of finalization, avoids indestructible objects.
- *weak tables*:
  - A table whose keys and/or values are referred by weak references.
  - Mitigate common GC problems (*ephemerons*), provide support for common data-structures implemented with weak tables (*property tables*).

Formal semantics

Garbage collection (GC)

Interaction between interfaces:

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

Formal semantics

GC: formalization

- Non-deterministic execution steps:

$$\frac{(\sigma', f, t) = \mathsf{gc}_{\mathsf{fin\_weak}}(\sigma, \mathsf{E}[\![\mathsf{s}]\!])}{\sigma : \mathsf{E}[\![\mathsf{s}]\!] \ \mapsto^{\mathsf{gc}_{\mathsf{fin\_weak}}} \sigma' : \mathsf{E}[\![f(t); \mathsf{s}]\!]}$$

Formal semantics

GC: formalization

- Non-deterministic execution steps:

$$\frac{(\sigma', f, t) = \mathrm{gc}_{\mathrm{fin\_weak}}(\sigma, \mathrm{E}[\![\mathrm{s}]\!])}{\sigma : \mathrm{E}[\![\mathrm{s}]\!] \mapsto^{\mathrm{gc}_{\mathrm{fin\_weak}}} \sigma' : \mathrm{E}[\![f(t); \mathrm{s}]\!]}$$

- gc_{fin_weak} specifies the behavior of a correct garbage collector for Lua:
  - Suitable notion of reachability for Lua.
  - Specifies the portion of the store that can be reclaimed.
  - Specifies fields of weak tables than can be removed.
  - Identifies the next table to be finalized.
  - Specifies interaction between both interfaces.

Formal semantics

GC: properties

- Framework for GC and sanity-check:
  - Define appropriate notions of *result* of programs, observations over programs (non-termination or returned values), *garbage*.

GC: properties

- Framework for GC and sanity-check:
  - Define appropriate notions of *result* of programs, observations over programs (non-termination or returned values), *garbage*.
  - Sanity check: $\mapsto^{gc}$ (GC-steps without interfaces to the garbage collector) preserves reachability, result depends on reachability, *postponement* lemma.

GC: properties

- Framework for GC and sanity-check:
  - Define appropriate notions of *result* of programs, observations over programs (non-termination or returned values), *garbage*.
  - Sanity check: $\mapsto^{gc}$ (GC-steps without interfaces to the garbage collector) preserves reachability, result depends on reachability, *postponement* lemma.
  - *GC-correctness*: for a given program P, the observations are preserved by GC-steps without interfaces to the garbage collector (*i.e.,* GC-steps only remove garbage):

$$obs(P, \mapsto) = obs(P, \mapsto \cup \mapsto^{gc})$$

Features left

Formal semantics

- Features left:
  - Types: coroutines (an independent thread of execution; only suspends its execution by explicitly calling a yield function) and userdata (to allow arbitrary C data to be stored in Lua variables).
  - **goto**.
  - Remaining standard library's services: coroutine, string, table.

Mechanization

The mechanization.

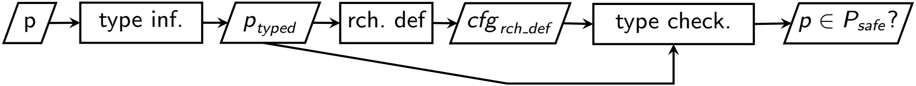
- Implemented using Redex.
- Tested against Lua 5.2's test suite: 1382 LOCS (from 6902 LOCS).
- Why?
  - Language features not covered by our formalization (mostly, library services).
  - Tests of implementation details of the interpreter and not the language's semantics: generation of bytecode, performance, etc.
- Every line of code of the test suite that falls within the scope of this work (except for GC: poor performance) successfully passes the tests.
- Mechanization available at `github.com/Mallku2/lua-gc-redex-model`.

Practical application

Practical application

- Problem: $\exists p, obs(p, \mapsto) \neq obs(p, \mapsto \cup \mapsto^{gc\_fin\_weak})$
- New possibilities for static analysis over Lua programs: LuaSafe
- Let $P_{safe} = \{p \mid obs(p, \mapsto) = obs(p, \mapsto \cup \mapsto^{gc\_fin\_weak})\}$
- 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*.

Practical application



Practical application

```
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"
```

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.

Publications

- *Decoding Lua: Formal semantics for the developer and the semanticist.*
  M. Soldevila, B. Ziliani, B. Silvestre, D. Fridlender, and F. Mascarenhas. In Proceedings of the 13th ACM SIGPLAN Dynamic Languages Symposium, DLS 2017, 2017.
- *Understanding Lua's Garbage Collection - Towards a Formalized Static Analyzer.*
  M. Soldevila, B. Ziliani, and D. Fridlender. In Proceedings of the 22nd Symposium on Principles and Practice of Declarative Programming, PPDP 2020, Bologna, Italy, September 8–10, 2020.

¡GRACIAS!