Decoding Lua: Formal Semantics for the Developer and the Semanticist

Mallku Soldevila[1], Beta Ziliani[1], Bruno Silvestre[2], Daniel Fridlender[3] and Fabio Mascarenhas[4]

[1]FAMAF/UNC and CONICET, [2]INF/UFG, [3]FAMAF/UNC, [4]DCC/UFRJ

24 October 2017

Summary

- About Lua.
- Why do we need a formal semantics of Lua?
- The Semantics.
- The mechanization.
- Future work.

About Lua

About Lua

- Extension programming language.
  - Good data-description facilities.
  - Small language, small implementation.
  - Should be extensible.
  - Clear and simple syntax.
  - No need for mechanisms for programming-in-the-large.
- Concretely:
  - Procedural programming with data-description facilities.
  - Features for fast development: dynamic typing, automatic memory management.
  - Metaprogramming mechanisms: modification of values' behaviour under special circumstances.

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**,...
  - Look at `www.lua.org/uses.html`.

Why do we need a formalized semantics of Lua?

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

Developers of tools for code analysis and language extensions

- Tools for code analysis:
  - **Luacheck**[1]
  - **Lua Inspect**[2]
  - More on `lua-users.org/wiki/ProgramAnalysis`.
- Language extensions
  - **Ravi**[3]
  - **Typed Lua**[4]
- Formal proofs of soundness, strengthen the possibilities of static analysis (e.g., *weak tables*).

---

[1] `https://github.com/mpeterv/luacheck`
[2] `http://lua-users.org/wiki/LuaInspect`
[3] http://ravilang.github.io/
[4] 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

- From λ_{JS}, S5, λ_{π}: it's plausible to give a formal semantics for real programming languages, using (mostly) just common mathematical knowledge.
- They even provide a lightweight mechanization.
- 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 be benefited from having people of differente areas testing it (JSCert).

Semantics

- The model.
- Semantics of stateless constructions.
- Semantics of state.
- Semantics of programs.
- Built-in services.
- Metatables.

The model

- Concepts from small-steps operational semantics and reduction semantics with evaluation contexts.

  - **Small-step operational semantics**: the execution model of state (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.

Semantics of stateless constructions

syntax

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

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

$e ::= v \mid e \textbf{ and } e \mid e \textbf{ or } e \mid \ldots$

relations between terms (computations)

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

$$\frac{op \in \{\textbf{and}, \textbf{or}\}}{v\ op\ e \rightarrow^{e} \delta(op, v, e)}$$

interpretation function

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

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

46 %

Semantics of state

syntax

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

computations

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

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

Semantics of programs

evaluation contexts

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

embedding relations using evaluation contexts

$$\frac{e \rightarrow^{\mathrm{e}} e'}{\sigma : E[\![e]\!] \mapsto \sigma : E[\![e']\!]} \qquad \frac{s \rightarrow^{\mathrm{s}} s'}{\sigma : E[\![s]\!] \mapsto \sigma : E[\![s']\!]}$$

$$\frac{\sigma : s \rightarrow^{\mathrm{s\_}\sigma} \sigma' : s'}{\sigma : E[\![s]\!] \mapsto \sigma' : E[\![s']\!]}$$

Built-in services

- Abstracts the details of the semantics of a service into an interpretation function (δ):

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

- Our def. of execution environment: global variables bound with wrapper procedures of a **$builtIn** form:

```
type = function (v)
         return $builtIn type(v)
       end
```

Metatables

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

```
local t = {}
print(t)     >> table: 0x68d7f0
print(next(t))     >> nil
t()     >> attempt to call local 't' (a table value)
setmetatable(t, { __call = function () print("Callable!") end})
print(t)     >> table: 0x68d7f0
print(next(t))     >> nil
t()     >> Callable!
```

- Useful to develop DSLs.

Metatables

- Formalization of the mechanism:
  - The special operation is tagged:

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

  - The metatable mechanism solves the situation:

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

- Some of the features formalized:
  - Every type of Lua value, except coroutines and userdata.
  - Metatables.
  - Identity of closures.
  - Dynamic execution of source code.
  - Error handling.
  - Services of the standard library: basic functions and services from the libraries **math**, **tables** and **string**.

- Features left:
  - Coroutines and userdata.
  - GC and *weak tables*.
  - **goto** and **repeat** statement.
  - Remaining standard library's services.

Mechanization

The mechanization.

- Implemented using PLT Redex.
- Tested against Lua 5.2's test suite:

File	Features tested	Coverage
calls.lua	**functions and calls**	77.83%
closure.lua	**closures**	48.5%
constructs.lua	**syntax and short-circuit opts.**	63.18%
events.lua	**metatables**	90.4%
locals.lua	**local variables and environments**	62.3%
math.lua	**numbers and math lib**	82.2%
nextvar.lua	**tables, next, and for**	53.24%
sort.lua	**(parts of) table library**	24.1%
vararg.lua	**vararg**	100%

- Next step: testing against libraries written in pure Lua.

- What's left from the test suite:
  - Language features not covered by our formalization (mentioned later).
  - Tests of implementation details of the interpreter and not the language's semantics.
- Every line of code of the test suite that falls within the scope of this work successfully passes the tests.
- Mechanization available at `github.com/Mallku2/lua-redex-model`.

Future work

- Strengthen the possibilities of static analysis
  - Weak references (wr): don't prevent the data they point to from being garbage collected.
  - Lua introduces wr by means of *weak tables*: Lua's tables whose elements are wr.

```
local t = {}    >> New table...
setmetatable(t, {__mode = "v"})    >> ...whose values are
                                   >> referred by wr.
```

  - wr are a way of interfacing with the GC: it opens the possibility of writing programs with GC dependent behaviour.

Future work

- Strengthen the possibilities of static analysis

```
local t = {}
setmetatable(t, {__mode = "v"})
t[“foo”] = {}     >> Just one ref. to this value: a wr.
```

Future work

- Strengthen the possibilities of static analysis

```
local t = {}
setmetatable(t, {__mode = "v"})
t[“foo”] = {}     >> Just one ref. to this value: a wr.
local i = 0
while t[“foo”] do     >> GC will delete the value t[“foo”]
    print(i)
    i = i + 1
end
```

Future work

- Strengthen the possibilities of static analysis

```
local t = {}
setmetatable(t, {__mode = "v"})
t["foo"] = {}     >> Just one ref. to this value: a wr.
local i = 0
while t["foo"] do     >> GC will delete the value t["foo"]
    print(i)
    i = i + 1
end
```

```
~$ lua wt.lua
0
.
.
.
391

~$ lua wt.lua
0
.
.
.
337
```

96 %

Future work

- Strengthen the possibilities of static analysis

```
local t = {}
setmetatable(t, {__mode = "v"})
t["foo"] = {}      >> Just one ref. to this value: a wr.
local i = 0
while t["foo"] do      >> GC will delete the value t["foo"]
    print(i)
    i = i + 1
end
```

```
~$ lua wt.lua
0
.
.
.
391

~$ lua wt.lua
0
.
.
.
337
```

} Donelly et. al., "*Formal Semantics of Weak References*"

96 %

Future work

- Add missing features:
  - goto (replace evaluation contexts by *program contexts*[5]).
  - Coroutines[6].
  - GC and weak tables (in progress).
  - Remaining services of the standard library.
- Check desired properties on a proof assistant: PLT Redex $\rightarrow$ Coq.
- Enjoy it:
  - Recognise GC-safe Lua programs.

[5] R. Krebbers and F. Wiedijk. *Separation logic for non-local control flow and block scope variables*. In FOSSACS'13, 2013

[6] A. L. Moura and R. Ierusalimschy. *Revisiting coroutines*. TOPLAS, 31(2):6:1–6:31, February 2009