luacheck Documentation

Release 0.10.0

Peter Melnichenko

July 17, 2015

Contents

1 List of warnings **3**
- 1.1 Global variables . . . 4
- 1.2 Unused variables and values . . . 4
- 1.3 Shadowing declarations . . . 5
- 1.4 Control flow and data flow issues . . . 5

2 Command line interface **7**
- 2.1 Command line options . . . 8
- 2.2 Patterns . . . 9
- 2.3 Formatters . . . 10
- 2.4 Caching . . . 10

3 Configuration file **11**
- 3.1 Config options . . . 11
- 3.2 Per-prefix overrides . . . 12

4 Inline options **13**

5 Luacheck module **15**
- 5.1 Report format . . . 15

Contents:

CHAPTER 1

List of warnings

Warnings produced by Luacheck are categorized using three-digit warning codes. Warning codes can be displayed in CLI output using `--codes` CLI option or `code` config option.

Code	Description
111	Setting an undefined global variable.
112	Mutating an undefined global variable.
113	Accessing an undefined global variable.
121	Setting a read-only global variable.
122	Mutating a read-only global variable.
131	Unused implicitly defined global variable.
211	Unused local variable.
212	Unused argument.
213	Unused loop variable.
221	Local variable is accessed but never set.
231	Local variable is set but never accessed.
232	An argument is set but never accessed.
233	Loop variable is set but never accessed.
311	Value assigned to a local variable is unused.
312	Value of an argument is unused.
313	Value of a loop variable is unused.
321	Accessing uninitialized local variable.
411	Redefining a local variable.
412	Redefining an argument.
413	Redefining a loop variable.
421	Shadowing a local variable.
422	Shadowing an argument.
423	Shadowing a loop variable.
431	Shadowing an upvalue.
432	Shadowing an upvalue argument.
433	Shadowing an upvalue loop variable.
511	Unreachable code.
512	Loop can be executed at most once.
521	Unused label.
531	Left-hand side of an assignment is too short.
532	Left-hand side of an assignment is too long.
541	An empty `do end` block.
542	An empty `if` branch.

1.1 Global variables

For each file, Luacheck builds list of defined globals which can be used there. By default only globals from Lua standard library are defined; custom globals can be added using `--globals` CLI option or `globals` config option, and version of standard library can be selected using `--std` CLI option or `std` config option. When an undefined global is set, mutated or accessed, Luacheck produces a warning.

1.1.1 Read-only globals

By default, all standard globals except `_G` and `package` are marked as read-only, so that setting or mutating them produces a warning. Custom read-only globals can be added using `--read-globals` CLI option or `read_globals` config option.

1.1.2 Implicitly defined globals

Luacheck can be configured to consider globals assigned under some conditions to be defined implicitly. When `-d`/`--allow_defined` CLI option or `allow_defined` config option is used, all assignments to globals define them; when `-t`/`--allow_defined_top` CLI option or `allow_defined_top` config option is used, assignments to globals in the top level function scope (also known as main chunk) define them. A warning is produced when an implicitly defined global is not accessed anywhere.

1.1.3 Modules

Files can be marked as modules using `-m`/`--module` CLI option or `module` config option to simulate semantics of the deprecated module function. Globals implicitly defined inside a module are considered part of its interface, are not visible outside and are not reported as unused. Assignments to other globals are not allowed, even to defined ones.

1.2 Unused variables and values

Luacheck generates warnings for all unused local variables except one named `_`. It also detects variables which are set but never accessed or accessed but never set.

1.2.1 Unused values and uninitialized variables

For each value assigned to a local variable, Luacheck computes set of expressions where it could be used. Warnings are produced for unused values (when a value can't be used anywhere) and for accessing uninitialized variables (when no values can reach an expression). E.g. in the following snippet value assigned to `foo` on line 1 is unused, and variable `bar` is uninitialized on line 9:

```
1  local foo = expr1()
2  local bar
3
4  if condition() then
5     foo = expr2()
6     bar = expr3()
7  else
8     foo = expr4()
9     print(bar)
10 end
```

```
11
12 return foo, bar
```

1.2.2 Secondary values and variables

Unused value assigned to a local variable is secondary if its origin is the last item on the RHS of assignment, and another value from that item is used. Secondary values typically appear when result of a function call is put into locals, and only some of them are later used. For example, here value assigned to `b` is secondary, value assigned to `c` is used, and value assigned to `a` is simply unused:

```
1 local a, b, c = f(), g()
2
3 return c
```

A variable is secondary if all values assigned to it are secondary. In the snippet above, `b` is a secondary variable.

Warnings related to unused secondary values and variables can be removed using `-s`/`--no-unused-secondaries` CLI option or `unused_secondaries` config option.

1.3 Shadowing declarations

Luacheck detects declarations of local variables shadowing previous declarations, unless the variable is named `_`. If the previous declaration is in the same scope as the new one, it is called redefining.

Note that it is **not** necessary to define a new local variable when overwriting an argument:

```
1 local function f(x)
2    local x = x or "default" -- bad
3 end
4
5 local function f(x)
6    x = x or "default" -- good
7 end
```

1.4 Control flow and data flow issues

The following control flow and data flow issues are detected:

- Unreachable code and loops that can be executed at most once (e.g. due to an unconditional break);
- Unused labels;
- Unbalanced assignments;
- Empty blocks.

CHAPTER 2

Command line interface

`luacheck` program accepts files, directories and rockspecs as arguments.

- Given a file, `luacheck` will check it.
- Given `-`, `luacheck` will check stdin.
- Given a directory, `luacheck` will check all files with `.lua` extension within it. This feature requires LuaFileSystem (installed automatically if LuaRocks was used to install Luacheck).
- Given a rockspec (a file with `.rockspec` extension), `luacheck` will check all files with `.lua` extension mentioned in the rockspec in `build.install.lua`, `build.install.bin` and `build.modules` tables.

The output of `luacheck` consists of separate reports for each checked file and ends with a summary:

```
$ luacheck src
Checking src/bad_code.lua                         Failure

    src/bad_code.lua:3:16: unused variable helper
    src/bad_code.lua:3:23: unused variable length argument
    src/bad_code.lua:7:10: setting non-standard global variable embrace
    src/bad_code.lua:8:10: variable opt was previously defined as an argument on line 7
    src/bad_code.lua:9:11: accessing undefined variable hepler

Checking src/good_code.lua                        OK
Checking src/python_code.lua                      Syntax error

    spec/samples/python_code.lua:1:6: expected '=' near '__future__'

Checking src/unused_code.lua                      Failure

    src/unused_code.lua:3:18: unused argument baz
    src/unused_code.lua:4:8: unused loop variable i
    src/unused_code.lua:5:13: unused variable q
    src/unused_code.lua:7:11: unused loop variable a
    src/unused_code.lua:7:14: unused loop variable b
    src/unused_code.lua:7:17: unused loop variable c
    src/unused_code.lua:13:7: value assigned to variable x is unused
    src/unused_code.lua:14:1: value assigned to variable x is unused
    src/unused_code.lua:22:1: value assigned to variable z is unused

Total: 14 warnings / 1 error in 4 files
```

`luacheck` exits with 0 if no warnings or errors occured and with a positive number otherwise.

2.1 Command line options

Short options that do not take an argument can be combined into one, so that `-qqu` is equivalent to `-q -q -u`. For long options, both `--option value` or `--option=value` can be used.

Options taking several arguments can be used several times; `--ignore foo --ignore bar` is equivalent to `--ignore foo bar`.

Note that options that may take several arguments, such as `--globals`, should not be used immediately before positional arguments; given `--globals foo bar file.lua`, `luacheck` will consider all `foo`, `bar` and `file.lua` global and then panic as there are no file names left.

Option	Meaning
`-g|--no-global`	Filter out warnings related to global variables.
`-u|--no-unused`	Filter out warnings related to unused variables and values.
`-r|--no-redefined`	Filter out warnings related to redefined variables.
`-a|--no-unused-args`	Filter out warnings related to unused arguments and loop variables.
`-s|--no-unused-secondaries`	Filter out warnings related to unused variables set together with used ones. See *Secondary values and variables*
`--std <std>`	Set standard globals. `<std>` must be one of: • `_G` - globals of the Lua interpreter `luacheck` runs on (default); • `lua51` - globals of Lua 5.1; • `lua52` - globals of Lua 5.2; • `lua52c` - globals of Lua 5.2 compiled with LUA_COMPAT_ALL; • `lua53` - globals of Lua 5.3; • `lua53c` - globals of Lua 5.3 compiled with LUA_COMPAT_5_2; • `luajit` - globals of LuaJIT 2.0; • `min` - intersection of globals of Lua 5.1, Lua 5.2 and LuaJIT 2.0; • `max` - union of globals of Lua 5.1, Lua 5.2 and LuaJIT 2.0; • `none` - no standard globals.
`--globals [<global>] ...`	Add custom globals on top of standard ones.
`--read-globals [<global>] ...`	Add read-only globals.
`--new-globals [<global>] ...`	Set custom globals. Removes custom globals added previously.
`--new-read-globals [<global>] ...`	Set read-only globals. Removes read-only globals added previously.
`-c|--compat`	Equivalent to `--std max`.
`-d|--allow-defined`	Allow defining globals implicitly by setting them. See *Implicitly defined globals*
`-t|--allow-defined-top`	Allow defining globals implicitly by setting them in the top level scope. See *Implicitly defined globals*

Continued on next page

Table 2.1 – continued from previous page

Option	Meaning
`-m|--module`	Limit visibility of implicitly defined globals to their files. See *Modules*
`--no-unused-globals`	Filter out warnings related to set but unused global variables.
`--ignore | -i <patt> [<patt>] ...`	Filter out warnings matching patterns.
`--enable | -o <patt> [<patt>] ...`	Do not filter out warnings matching patterns.
`--only | -o <patt> [<patt>] ...`	Filter out warnings not matching patterns.
`--no-inline`	Disable inline options.
`--config <config>`	Path to custom configuration file (default: `.luacheckrc`).
`--no-config`	Do not look up custom configuration file.
`--cache [<cache>]`	Path to cache file. (default: `.luacheckcache`). See *Caching*
`--no-cache`	Do not use cache.
`-j|--jobs`	Check `<jobs>` files in parallel. Requires LuaLanes.
`--formatter <formatter>`	Use custom formatter. `<formatter>` must be a module name or one of: • `TAP` - Test Anything Protocol formatter; • `JUnit` - JUnit XML formatter; • `plain` - simple warning-per-line formatter; • `default` - standard formatter.
`-q|--quiet`	Suppress report output for files without warnings. • `-qq` - Suppress output of warnings. • `-qqq` - Only output summary.
`--codes`	Show warning codes.
`--no-color`	Do not colorize output.
`-v|--version`	Show version of luacheck and its dependencies and exit.
`-h|--help`	Show help and exit.

2.2 Patterns

CLI options `--ignore`, `--enable` and `--only` and corresponding config options allow filtering warnings using pattern matching on warning codes, variable names or both. If a pattern contains a slash, the part before slash matches warning code and the part after matches variable name. Otherwise, if a pattern contains a letter or underscore, it matches variable name. Otherwise, it matches warning code. E.g.:

Pattern	Matching warnings
4.2	Shadowing declarations of arguments or redefining them.
.*_	Warnings related to variables with _ suffix.
4.2/.*_	Shadowing declarations of arguments with _ suffix or redefining them.

Unless already anchored, patterns matching variable names are anchored at both sides and patterns matching warning codes are anchored at their beginnings. This allows to filter warnings by category (e.g. `--only 1` focuses `luacheck` on global-related warnings).

2.3 Formatters

CLI option `--formatter` allows selecting a custom formatter for `luacheck` output. A custom formatter is a Lua module returning a function with three arguments: report as returned by `luacheck` module (see *Report format*), array of file names and table of options. Options contain values assigned to `quiet`, `color`, `limit`, `codes` and `formatter` options in CLI or config. Formatter function must return a string.

2.4 Caching

If LuaFileSystem is available, Luacheck can cache results of checking files. On subsequent checks, only files which have changed since the last check will be rechecked, improving run time significantly. Changing options (e.g. defining additional globals) does not invalidate cache. Caching can be enabled by using `--cache <cache>` option or `cache` config option. Using `--cache` without an argument or setting `cache` config option to `true` sets `.luacheckcache` as the cache file. Note that `--cache` must be used every time `luacheck` is run, not on the first run only.

CHAPTER 3

Configuration file

By default, `luacheck` tries to load configuration from `.luacheckrc` file in the current directory. Path to config can be set using `--config` option. Config loading can be disabled using `--no-config` flag.

Config is simply a Lua script executed by `luacheck`. It may set various options by assigning to globals.

3.1 Config options

Option	Type	Default value
`color`	Boolean	`true`
`codes`	Boolean	`false`
`formatter`	String	`"default"`
`cache`	Boolean or string	`false`
`jobs`	Positive integer	`1`
`global`	Boolean	`true`
`unused`	Boolean	`true`
`redefined`	Boolean	`true`
`unused_args`	Boolean	`true`
`unused_secondaries`	Boolean	`true`
`std`	String or array of strings	`"_G"`
`globals`	Array of strings	`{}`
`new_globals`	Array of strings	(Do not overwrite)
`read_globals`	Array of strings	`{}`
`new_read_globals`	Array of strings	(Do not overwrite)
`compat`	Boolean	`false`
`allow_defined`	Boolean	`false`
`allow_defined_top`	Boolean	`false`
`module`	Boolean	`false`
`unused_globals`	Boolean	`true`
`ignore`	Array of patterns (see *Patterns*)	`{}`
`enable`	Array of patterns	`{}`
`only`	Array of patterns	(Do not filter)
`inline`	Boolean	`true`

An example of a config which makes `luacheck` ensure that only globals from the portable intersection of Lua 5.1, Lua 5.2, Lua 5.3 and LuaJIT 2.0 are used, as well as disables detection of unused arguments:

```
std = "min"
ignore = {"211"}
```

3.2 Per-prefix overrides

The environment in which `luacheck` loads the config contains a special global `files`. When checking a file `<path>`, `luacheck` will override options from the main config with entries from `files[<path_prefix>]`, applying entries for shorter prefixes first. This allows to override options for a specific file by setting `files[<path>]`, and for all files in a directory by setting `files[<dir>/]`. For example, the following config re-enables detection of unused arguments only for files in `src/dir`, but not for `src/dir/myfile.lua`:

```
std = "min"
ignore = {"211"}

files["src/dir/"] = {
   enable = {"211"}
}

files["src/dir/myfile.lua"] = {
   ignore = {"211"}
}
```

Note that `files` table supports autovivification, so that

```
files["myfile.lua"].ignore = {"211"}
```

and

```
files["myfile.lua"] = {
   ignore = {"211"}
}
```

are equivalent.

CHAPTER 4

Inline options

Luacheck supports setting some options directly in the checked files using inline configuration comments. An inline configuration comment starts with `luacheck:` label, possibly after some whitespace. The body of the comment should contain comma separated options, where option invocation consists of its name plus space separated arguments. The following options are supported:

Option	Number of arguments
compat	0
module	0
allow_defined	0
allow_defined_top	0
std	1
globals	0+
new_globals	0+
read_globals	0+
new_read_globals	0+
ignore	0+ (without arguments everything is ignored)
enable	1+
only	1+

Part of the file affected by inline option dependes on where it is placed. If there is any code on the line with the option, only that line is affected; otherwise, everthing till the end of the current closure is. In particular, inline options at the top of the file affect all of it:

```
1  -- luacheck: globals g1 g2, ignore foo
2  local foo = g1(g2) -- No warnings emitted.
3
4  -- The following unused function is not reported.
5  local function f() -- luacheck: ignore
6     -- luacheck: globals g3
7     g3() -- No warning.
8  end
9
10 g3() -- Warning is emitted as the inline option defining g3 only affected function f.
```

For fine-grained control over inline option visibility use `luacheck: push` and `luacheck: pop` directives:

```
1  -- luacheck: push
2  -- luacheck: ignore foo
3  foo() -- No warning.
4  -- luacheck: pop
5  foo() -- Warning is emitted.
```

Inline options can be completely disabled using `--no-inline` CLI option or `inline` config option.

CHAPTER 5

Luacheck module

Use `local luacheck = require "luacheck"` to import `luacheck` module. It contains the following functions:

- `luacheck.get_report(source)`: Given source string, returns analysis data (an array) or nil and syntax error table (table with fields `line`, `column`, `offset`, `msg`).
- `luacheck.process_reports(reports, options)`: Processes array of analysis reports and applies options. `reports[i]` uses `options`, `options[i]`, `options[i][1]`, `options[i][2]`, ... as options, overriding each other in that order. Options table is a table with fields similar to config options; see *Config options*. Analysis reports with field `error` are ignored. `process_reports` returns final report, see *Report format*.
- `luacheck.check_strings(sources, options)`: Checks array of sources using options, returns final report. Tables in `sources` array are ignored.
- `luacheck.check_files(files, options)`: Checks array of files using options, returns final report. Open file handles can passed instead of filenames, in which case they will be read till EOF and closed.

`luacheck._VERSION` contains Luacheck version as a string in `MAJOR.MINOR.PATCH` format.

Using `luacheck` as a function is equivalent to calling `luacheck.check_files`.

5.1 Report format

A final report is an array of file reports plus fields `warnings` and `errors` containing total number of warnings and errors, correspondingly.

A file report is an array of warnings. If an error occured while checking a file, its report will have `error` field containing `"I/O"` or `"syntax"`. In case of syntax error, `line` (number), `colunmn` (number), `offset` (number) and `msg` (string) fields are also present.

A warning is a table with field `code` indicating the type of warning (see List of warnings), and fields `line` and `column` pointing to the source of the warning. Absence of `code` field indicates that the warning is related to a broken inline configuration comment; then, `invalid` field marks comments with invalid syntax, and `unpaired` field marks unpaired push/pop comments.

Warnings of some types can also have additional fields:

Codes	Additional fields
111	`module` field indicates that assignment is to a non-module global variable.
211	`func` field indicates that unused variable is a function.
212	`vararg` field indicated that variable length argument is unused.
4..	`prev_line` and `prev_column` fields contain location of the overwritten defintion.

Other fields may be present for internal reasons.

This is documentation for Luacheck 0.10.0, a static analyzer and a linter for Lua.