luacheck Documentation

Release 0.7.3

Peter Melnichenko

July 17, 2015

Contents

Contents:

CHAPTER 1

Types of warnings

Luacheck generates warnings of three types:

- warnings related to global variables;
- warnings related to unused local variables and values;
- warnings related to redefined local variables.

1.1 Global variables

To determine whether an assignment to a global or accessing a global should produce a warning, Luacheck builds a list of defined globals for each file. Globals can be defined explicitly or implicitly. Accessing or setting an undefined global produces or warning of corresponding subtype. All warnings related to globals can be disabled using `-g`/`--no-global` CLI option or `global` config option.

1.1.1 Explicitly defined globals

Explicitly defined globals consist of standard and custom globals. Standard globals are globals provided by Lua interpreter, and can be set using `--std` CLI option or `std` config option. Custom globals are globals accessible due to other reasons, and can be set using `--globals` CLI option or `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.

If an implicitly defined global is not accessed anywhere, a warning is produced, unless `--no-unused-globals` CLI option or `unused_globals` config option is used.

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 not visible outside and are not reported as unused. Additionally, only assignments to implicitly defined globals are allowed.

1.2 Unused variables

Luacheck generates warnings for all unused local variables except one named _. These warnings can be disabled using `-u`/`--no-unused` CLI option or `unused` config option.

Detection of unused arguments and loop variables can be disabled using `-a`/`--no-unused-args` CLI option or `unused_args` config option.

1.2.1 Unused values

Luacheck also detects unused values: redundant assignments to variables which are then not used before another assignment. As an example, in the following snippet value assigned to `foo` on line 1 is unused, as it is overwritten in both branches of `if` statement before being used:

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

Detection of unused values can be disabled using `-v`/`--no-unused-values` CLI option or `unused_values` config option.

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
```

Secondary variables are unused variables initialized with a secondary value. 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.

1.2.3 Unset variables

Luacheck generates warnings for local variables that are accessed but never set. These warnings can be removed using `--no-unset` CLI option or `unset` config option.

1.3 Redefined variables

Luacheck detects declarations of local variables shadowing previous declarations in the same scope, unless the variable is named _. This diagnostic can be disabled using `-r`/`--no-redefined` CLI option or `redefined` config option.

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

```
local function f(x)
   local x = x or "default" -- bad
end

local function f(x)
   x = x or "default" -- good
end
```

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.
- 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
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 time; `--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.
`-r|--no-redefined`	Filter out warnings related to redefined variables.
`-u|--no-unused`	Filter out warnings related to unused variables.
`-a|--no-unused-args`	Filter out warnings related to unused arguments and loop variables.
`-v|--no-unused-values`	Filter out warnings related to unused values.
`-s|--no-unused-secondaries`	Filter out warnings related to unused variables set together with used ones. See *Secondary values and variables*
`--no-unset`	Filter out warnings related to unset 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; • `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.
`--new-globals [<global>] ...`	Set custom globals. Removes custom 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*
`-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 <var> [<var>] ...`	Filter out warnings related to variables named `<var>`.
`--only <var> [<var>] ...`	Filter out warnings not related to variables named `<var>`.
`-l <limit>|--limit <limit>`	Exit with 0 if there are `<limit>` or less warnings (default: `0`).
`--config <config>`	Path to custom configuration file (default: `.luacheckrc`).
`--no-config`	Do not look up custom configuration file.
`-q|--quiet`	Suppress report output for files without warnings. • `-qq` - Suppress output of warnings. • `-qqq` - Only output summary.
`--no-color`	Do not colorize output.
`-h|--help`	Show help and exit.

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.

3.1 Config format

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

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

```
std = "min"
unused_args = false
```

3.2 Per-file 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>]`. For example, the following config re-enables detection of unused arguments only for `myfile.lua`:

```
std = "min"
unused_args = false

files["myfile.lua"] = {
   unused_args = true
}
```

Note that `files` table supports autovivification, so that

```
files["myfile.lua"].unused_args = true
```

and

```
files["myfile.lua"] = {
   unused_args = true
}
```

are equivalent.

CHAPTER 4

Luacheck module

`luacheck` module is a single function. Use `local luacheck = require "luacheck"` to import it.

The first argument of the function should be an array. Each element should be either a file name (string) or an open file handle, in which case `luacheck` will read it till EOF and close it.

4.1 Options

The second argument, if present, should be a table of options. Options are interpreted similarly to corresponding command line switches; see *Command line options*.

Option	Type	Default value
`options.global`	Boolean	`true`
`options.redefined`	Boolean	`true`
`options.unused`	Boolean	`true`
`options.unused_args`	Boolean	`true`
`options.unused_values`	Boolean	`true`
`options.unused_secondaries`	Boolean	`true`
`options.unset`	Boolean	`true`
`options.std`	String or array of strings	`"_G"`
`options.globals`	Array of strings	`{}`
`options.compat`	Boolean	`false`
`options.allow_defined`	Boolean	`false`
`options.allow_defined_top`	Boolean	`false`
`options.module`	Boolean	`false`
`options.unused_globals`	Boolean	`true`
`options.ignore`	Array of strings	`{}`
`options.only`	Array of strings	(Do not filter)

When checking `n`-th file, `luacheck` will try to combine `options[n]` with general options, similarly to how per file config tables overwrite general config table. See Configuration file.

4.2 Report format

The `luacheck` function returns a report. A 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 only have `error` field containing `"I/O"` or `"syntax"`.

A warning is a table with fields `type`, `subtype` and `vartype` indicating the type of warning (see Types of warnings), and fields `line` and `column` pointing to the source of the warning. For warnings related to redefined variables there also are fields `prev_line` and `prev_column` pointing to the previous declaration of the variable.

Luacheck is a tool for linting and static analysis of Lua code. It is able to spot usage of undefined global variables, unused local variables and a few other typical problems within Lua programs.

Luacheck provides a command line interface as well as a Lua module which can be used by other programs.

This is the documentation for the 0.7.3 version.