Reference Manual of the Lua Debugger 1.0 (beta)

Toms Guisasola Gorham

tomas@icad.puc-rio.br

TeCGraf — Departamento de Informática — PUC-Rio

Date: 1996/06/20 22:37:19

Abstract

Ldb is a debugging system for the extension language Lua, provided as a library of Lua functions to be linked with the application. The library provides debugging features like breakpoints, step-by-step execution and automatic visulization of variables. This document describes the commands of the debugger and shows some examples on how they can be used to help the programmer in the debugging process.

1 Introduction

Ldb is a debugging system for Lua [IdFF95] programs. The system is implemented as a library with a Lua interface for its functions. The library must be linked with the application during the debugging phase, but it may be discarted in the final version of the application.

2 Architecture

The system was built using the debugger interface of Lua 2.3. This interface is based on a mechanism of *hooks*. These hooks are activated in three points: changing of execution line, function calls and function returns. These three points are the only points where the debugger can stop the execution and provide user interaction.

The *call hook* and *return hook* can be used without any change in the source code, however, the *line hook* need *debug information* to work. This information will be automatically generated by the interpreter provided the compilation pragma `$debug` is written at the beginning of the part of the source code that being debugged.[1]

The Lua debugger offers a Dialog Line which is similar to the `gdb` [Fre94] command line. The Dialog Line is a Lua interpreter with some extra features, like compact form for activation of the most used functions and simulation of the local environment of the function being debugged.

Some important concepts are defined in the following subsections. Locations and actions are uncommon concepts used in many debugging functions. The local environment and the Dialog Line have some particularities not found in other debugging systems.

[1] The compilation pragma `$nodebug` can be used to end a piece of code with debug information.

2.1 Locations

Locations are points of the source code of a program. The debugger uses locations to inform the user which point of the program is being debugged. Locations are also used by many debugger commands that need a reference to a point of the source code, such as `break`, `goto` and `list`.

A location is usually indicated by a line number and a file name, but it can also be identified by a function. In this case, the function is converted by the debugger to a pair ⟨line, file⟩ that indicates where the function was defined. The debugger keeps a *current location*, which is the location of the next instruction to be executed.

All commands that needs a location accept arguments in one of the following three forms:

⟨line⟩ which points to a given line of the current file.

⟨line, file⟩ which points to a given line of a given file.

⟨function⟩ which points to the line and file where the given function was defined.

Note that when a function is returning, the debugger sets the current file to the string `<return>` and the current line to zero.

2.2 Actions

An *action* is a string containing a chunck of Lua code. Whenever the user specifies an action, it is imediately pre-compiled as the body of an anonymous Lua function. The debugger offers three features that use actions. The first one (available through the `doat` command) associates an action to a location. This can be used, for example, to implement conditional break points or to print variable values.

The second feature (`doalways`) creates an action to be executed before every line and every function call executed by the interpreter of Lua. This action is called *global action*. This feature can be used, for instance, to follow the behavior of a global variable, looking for a change of its value.

The last use of actions is offered by the `shortcut` command. This command sets an action to be executed as a macro by the user at the Dialog Line, creating an easy way to call the most common commands.

2.3 Local Environments

An important feature of any debugger is to provide access to local variables. **Ldb** provides this access by simulating the *local environment* of the function being debugged at the global level. To set a local environment, all local variables are copied to global ones (with the same names, of course), therfore allowing all Lua statements free access to them. All commands invoked by the user at the Dialog Line will work at this *simulated local environment*, so they can access the local variables as well.

The local environment is simulated only when is necessary to have access to local variables. After, the global environment is restored and the new values of the local variables are copied back to the stack updating the real local environment.

Section 3 explains each command and indicates the working environment of each debugging resource.

2.4 The Dialog Line

The **Ldb** *Dialog Line* is a Lua line interpreter, so it provides all Lua functions, including all the debugging commands listed in the next section. Moreover, it provides a facility for *shortcuts*. A shortcut is called when the Dialog Line starts with a `'`. The letter following the `'` can be associated with an action defined by the user. This action works in the same environment provided to the user, like a macro. Moreover, this action receives a parameter called `line`, which is a string with the rest of the line typed by the user. Therefore, shortcuts may receive parameters with any syntax.

The user can define a new shortcut (or redefine an old one) with the `setshortcut` function. The default configuration file define some shortcuts as seen at section 4.1.

3 Commands

All commands provided by **Ldb** are Lua functions. Some of them only work when called from within the Dialog Line, like `up` and `down`.

The following subsections explain all the debugger commands. Following the title, there is a line indicating the signature of the function. The parameters are in between the parenthesis; the optional ones are between brackets (e.g. `[number]`). The results of the function are shown at the end of the line before a colon (e.g. `:number`). The parameter *location* indicates the possibilities of indicating a location, i.e., a number, a number and a string or a function.

3.1 `break (location)`

The `break` command creates a breakpoint at a given location. It is equivalent to

```
doat("stop()", location)
```

See also: `delete`, `doat`, `finish` and `goto`.

3.2 `cont ()`

The `cont` command continues the normal execution of the program. It does not receive any parameter.

See also: `finish`, `goto`, `Next`, `step` and `stop`.

3.3 `delete (location)`

The `delete` function eliminates the break point or the action at a given location.

See also: `break`, `doat` and `show`.

3.4 `depth ([number]):number`

The `depth` command selects a specific activation register on the stack to be the local environment of the Dialog Line. It can receive a number indicating the activation register behind the higher one on the stack. If this number is omitted, the level is not changed. It always return the number of the activation register being simulated (relatively to the top register). The highest activation register is refered to as the zero one.

See also: `down`, `up` and `where`.

3.5 `display (string)`

`display` sets expressions to be displayed automatically every time de debugger stops the execution and enters the Dialog Line. It receives a string with the expression to be displayed. The expressions will be evaluated on the local environment, so it can contain references to local variables. `display` can also be invoked by the program being debugged, but the output will be shown only at the Dialog Line. Each expression will be displayed at a separate line, preceded by a number and followed by its value. The number is the identification of the expression and is used by the function `undisplay` to eliminate expressions. The user can set up to 30 expressions.

See also: `undisplay`.

3.6 `doalways ([action])`

The `doalways` command defines the global action. The global action will be executed before every line and every call the interpreter executes. `doalways` can receive a string containing the code of the action. The action is executed on the global environment, so it doesn't have access to local variables of functions. If no argument is passed, the global action is deactivated. The `doalways` command is very powerful; however it makes the program three times slower. There can be only one global action in the system.

See also: `doat` and `show`.

3.7 `doat (action,location)`

The `doat` function defines an action to be executed before the given location. The action works on the simulated local environment of the function indicated by the given location, so it can access local variables.

The user can set up to 30 "doat" actions. There cannot be more than one action at each location.

See also: `break`, `delete`, `doalways` and `goto`.

3.8 `down ([number])`

The `down` command changes the local environment used by the Dialog Line to another activation register down on the stack. It can receive a number indicating what register behind the current one will be the new one. This function is useful to access local variables from other activation registers. As it only works on the Dialog Line, most debugging functions aren't affected by this changing of context. When one of the execution commands (step, cont ...) is called the changed environment is lost.

See also: `depth`, `up` and `where`.

3.9 `file ():string`

The `file` function returns a string with the name of the current file. When the debugger stops at the return of a function the string returned is `<return>`.

See also: `line`.

3.10 `finish ()`

The `finish` command continues the execution until the end of the function in execution.

See also: `goto`.

3.11 `goto (location)`

The `goto` command continues the execution until a given location. This command is built by creating an action at the given location and executing a `cont` command after it. When reached, the action forces an interruption and eliminates itself. It is important to mention that the debugger can stop in another point different than that created by the command. If it occurs, the action will remain associated to the same location.

See also: `cont`, `finish`, `Next`, `step` and `stop`.

3.12 `line ():number`

The `line` function returns the number of the current line. When the debugger stops at the return of a function or at the beginning of a file, the number returned is zero.

See also: `file`.

3.13 `list ([number],[location])`

The `list` command lists some lines of the source code. It receives the number of lines to be listed and an optional location that indicates where to start.

3.14 `Next ([number])`

The `Next` command executes the next lines of the program being debugged. It optionally receives a number indicating how many lines must be executed (the default value is 1). If some of these lines are function calls, the `Next` command step over it as if they were simple instructions.

See also: `cont`, `finish`, `goto`, `step` and `stop`.

3.15 `setshortcut (string,action)`

The `setshortcut` function defines a shortcut to an action. Section 2.2 defines the concept of actions.

3.16 `show ()`

The `show` command displays all breakpoints and actions defined and the global action (if defined): The global action is indicated by `>>` and `<<`; the breakpoints and actions by its locations at the form `[line:file]`.

See also: `delete` and `doalways`.

3.17 `stack ()`

This command is a synonym of the `where` command.

See also: `where`.

3.18 `step ([number])`

The `step` command executes the next steps of the program being debugged. It optionally receives a number indicating how many lines must be executed (the default value is 1). If some of these lines are function calls, the `step` command step into it.

See also: `cont`, `finish`, `goto`, `Next` and `stop`.

3.19 `stop ([condition])`

The `stop` command signs an interruption of the execution. It's parameter is a condition that indicates if the stop must occur or not. If omitted, the parameter is considered true.

See also: `cont`, `finish`, `goto`, `Next` and `step`.

3.20 `undisplay (number)`

The `undisplay` command eliminates one of the expressions defined by the `display` command to be automatically displayed. It receives the number of the expression that must be deleted (these numbers are shown besides the expression).

See also: `display`.

3.21 `up ([number])`

The `up` command changes the local environment used by the Dialog Line to another activation register up on the stack. It can receive a number indicating what register above the current one will be the new one. As it only works on the Dialog Line, most debugging functions aren't affected by this changing of context. When one of the execution commands (step, cont ...) is called, the changed environment is lost.

See also: `depth`, `down` and `where`.

3.22 `where ()`

The `where` command is used to display the execution stack. It tries to show all possible informations about each function stacked, like the line each one was executing when the above was called. If where can't show these informations, then it displays the number of the line where the function was defined.

Some special cases are a function call (which shows the function name without any line number or file name) and a function return (when the string <return> is shown). These are special cases because there are no current line being debugged.

See also: depth, down and up.

4 Using the Debugger

To use the Lua Debugger, the user must do two things: add the debugger to the application[2] and insert two lines into the source code to include the header file and open the **Ldb** library. The following line will include the header file of the **Ldb** library.

```
#include "ldblib.h"
```

To open the library, the user must insert the following line just after the open of the lua libraries being used by the application:

```
ldblib_open ();
```

After that, the debugger is correctly linked with the application. The application will run normally if no errors occur. The error fallback is redefined to print the error message and call the Dialog Line. So, whenever an error occurs, the debugger is automatically invoked.

Even when there is no errors, the user may want to run step-by-step to find out logical errors. In this case, the user would like to stop the execution at the beginnig of the program or at a certain point previously known. This must be done by calling one or more of the debugging commands to force the interruption of the normal execution. For instance, a line with a `stop()` or a `break(5,"file.lua")` call can be inserted into the Lua source code of the application. A better way to determine these breakpoints is to put those lines at the configuration file of the debugger. See next section for how to do that.

4.1 Configuring the debugger

The debugger offers a configuration facility to the user. When it starts (during the execution of function `ldblib_open`), **Ldb** tries to load the configuration file `ldbrc.lua` which can contains Lua commands. These commands can create functions, variables or shortcuts and can be used to solve conflicts of names with other Lua libraries such as EDG [FFG95]. The EDG library defines the names `line` and `list` which are also defined by the debugger, so some names of the library opened first will be overwritten by the other one. Adding some commands to the `ldbrc.lua` file that define another names for the debugger functions can solve this problem.

The name of the configuration file can be determined by the environment variable `LDB_CONFIG`. The default file (`ldbrc.lua`) predefines some shortcuts:

[2]The debugger is distributed as a set of four files: the manual (`ldbman.ps`), the header file (`ldblib.h`), the source file (`ldblib.c`) and the default configuration file (`ldbrc.lua`). To add the debugger to the application, the user should add the `.c` file to the list of files of the application.

```
'-  down
'+  up
'?  list of commands
'c  cont
'd  display
'f  finish
'g  goto
'n  Next
'p  print
'q  exit
's  step
'v  show
'w  where
```

Acknowledgments

The author would like to thanks CAPES and SEPROC/CENPES/Petrobrás for the financial support.

References

[FFG95] W. Celes Filho, L. H. Figueiredo, and M. Gattass. Edg: Uma ferramenta para criação de interfaces gráficas interativas. In *SIBGRAPI 95*, pages 241–248, 1995.

[Fre94] Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA. *Debugging with GDB*, Edition 4.12, for GDB version 4.12 edition, January 1994.

[IdFF95] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes Filho. Reference Manual of the Programming Language Lua 2.3. Technical Report 8/95, PUC-Rio, 1995.

Index

Contents