Ravi Programming Language Documentation

Release 0.1

Dibyendu Majumdar

Jul 30, 2021

Contents

Contents:

CHAPTER 1

Ravi Programming Language

Ravi is a dialect of Lua with limited optional static typing and features a JIT compiler powered by MIR as well as experimental support for AOT compilation to native code. The name Ravi comes from the Sanskrit word for the Sun. Interestingly a precursor to Lua was Sol which had support for static types; Sol means the Sun in Portugese.

Lua is perfect as a small embeddable dynamic language so why a derivative? Ravi extends Lua with static typing for improved performance when JIT compilation is enabled. However, the static typing is optional and therefore Lua programs are also valid Ravi programs.

There are other attempts to add static typing to Lua - e.g. Typed Lua but these efforts are mostly about adding static type checks in the language while leaving the VM unmodified. The Typed Lua effort is very similar to the approach taken by Typescript in the JavaScript world. The static typing is to aid programming in the large - the code is eventually translated to standard Lua and executed in the unmodified Lua VM.

My motivation is somewhat different - I want to enhance the VM to support more efficient operations when types are known. Type information can be exploited by JIT compilation technology to improve performance. At the same time, I want to keep the language safe and therefore usable by non-expert programmers.

Of course there is the fantastic LuaJIT implementation. Ravi has a different goal compared to LuaJIT. Ravi prioritizes ease of maintenance and support, language safety, and compatibility with Lua 5.3, over maximum performance. For more detailed comparison please refer to the documentation links below.

1.1 Features

- Optional static typing - for details see the reference manual.
- Type specific bytecodes to improve performance
- Compatibility with Lua 5.3 (see Compatibility section below)
- Generational GC from Lua 5.4
- `defer` statement for releasing resources

- Compact JIT backend MIR.
- A distribution with batteries.
- A Visual Studio Code debugger extension - interpreted mode debugger.
- A new compiler framework for JIT and AOT compilation.
- AOT Compilation to shared library.

1.2 Documentation

- For the Lua extensions in Ravi see the Reference Manual.
- MIR JIT Build instructions.
- Also see Ravi Documentation.
- and the slides I presented at the Lua 2015 Workshop.

1.3 Lua Goodies

- An Introduction to Lua attempts to provide a quick overview of Lua for folks coming from other languages.
- Lua 5.3 Bytecode Reference is my attempt to bring up to date the Lua 5.1 Bytecode Reference.
- A patch for Lua 5.3 implements the 'defer' statement.
- A patch for Lua 5.4.[0-2] implements the 'defer' statement.
- Updated patch for Lua 5.4.3 implements the 'defer' statement.

1.4 Lua 5.4 Position Statement

Lua 5.4 relationship to Ravi is as follows:

- Generational GC - back-ported to Ravi.
- New random number generator - back-ported to Ravi.
- Multiple user values can be associated with userdata - under consideration.
- `<const>` variables - not planned.
- `<close>` variables - Ravi has `'defer'` statement which is the better option in my opinion, hence no plans to support `<close>` variables.
- Interpreter performance improvements - these are beneficial to Lua interpreter but not to the JIT backends, hence not much point in back-porting.
- Table implementation changes - under consideration.
- String to number coercion is now part of string library metamethods - back-ported to Ravi.
- utf8 library accepts codepoints up to 2^31 - back-ported to Ravi.
- Removal of compatibility layers for 5.1, and 5.2 - not implemented as Ravi continues to provide these layers as per Lua 5.3.

1.5 Compatibility with Lua 5.3

Ravi should be able to run all Lua 5.3 programs in interpreted mode, but following should be noted:

- Ravi supports optional typing and enhanced types such as arrays (see the documentation). Programs using these features cannot be run by standard Lua. However all types in Ravi can be passed to Lua functions; operations on Ravi arrays within Lua code will be subject to restrictions as described in the section above on arrays.
- Values crossing from Lua to Ravi will be subjected to typechecks should these values be assigned to typed variables.
- Upvalues cannot subvert the static typing of local variables (issue #26) when types are annotated.
- Certain Lua limits are reduced due to changed byte code structure. These are described below.
- Ravi uses an extended bytecode which means it is not compatible with Lua 5.x bytecode.
- Ravi incorporates the new Generational GC from Lua 5.4, hence the GC interface has changed.

Limit name	Lua value	Ravi value
MAXUPVAL	255	125
LUAI_MAXCCALLS	200	125
MAXREGS	255	125
MAXVARS	200	125
MAXARGLINE	250	120

When JIT compilation is enabled there are following additional constraints:

- Ravi will only execute JITed code from the main Lua thread; any secondary threads (coroutines) execute in interpreter mode.
- In JITed code tailcalls are implemented as regular calls so unlike the interpreter VM which supports infinite tail recursion JIT compiled code only supports tail recursion to a depth of about 110 (issue #17)
- Debug api and hooks are not supported in JIT mode

1.6 History

- **2015**
  - Implemented JIT compilation using LLVM
  - Implemented libgccjit based alternative JIT (now discontinued)
- **2016**
  - Implemented debugger for Ravi and Lua 5.3 for Visual Studio Code
- **2017**
  - Embedded C compiler using dmrC project (C JIT compiler) (now discontinued)
  - Additional type-annotations
- **2018**
  - Implemented Eclipse OMR JIT backend (now discontinued)
  - Created Ravi with batteries.
- **2019**

  - New language feature - *defer* statement
  - New JIT backend MIR.
- **2020**
  - New parser / type checker / compiler
  - Generational GC back-ported from Lua 5.4
  - Support for LLVM backend archived
- **2021 (Plan)**
  - Integrated AOT and JIT compilation support
  - Ravi 1.0 release

1.7 License

MIT License

CHAPTER 2

Ravi Extensions to Lua 5.3

Table of Contents

2.1 Introduction

Ravi is based on Lua 5.3. Additionally some features of Lua 5.4 have been back-ported to Ravi. This document describes the enhancements available in Ravi compared to the Lua 5.3 baseline.

- Optional static typing
- `defer` statement
- Generational Garbage Collector
- New random number generator

2.2 Optional Static Typing

Ravi allows you optionally to annotate `local` variables and function parameters with types.

Function return types cannot be annotated because in Lua, functions are un-named values and there is no reliable way for a static analysis of a function call's return value.

The supported type-annotations are as follows:

`integer` denotes an integral value of 64-bits.

`number` denotes a double (floating point) value of 8 bytes.

`integer[]` denotes an array of integers

`number[]` denotes an array of numbers

`table` denotes a Lua table

`string` denotes a string

`closure` denotes a function

`Name [. Name]*` Denotes a string that has a metatable registered in the Lua registry. This allows userdata types to be asserted by their registered names.

2.2.1 General Notes

- Assignments to type-annotated variables are checked at compile time if possible; when the assignments occur due to a function call, runtime type-checking is performed
- If function parameters are decorated with types, Ravi performs implicit type assertion checks on those parameters upon function entry. If the assertions fail then runtime errors are raised.
- Ravi performs type-checking of up-values that reference variables that are annotated with types
- To keep with Lua's dynamic nature Ravi uses a mix of compile type-checking and runtime type checks. However, in Lua, compilation happens at runtime anyway so effectively all checks are at runtime.

2.2.2 Caveats

The Lua C api allows C programmers to manipulate values on the Lua stack. This is incompatible with Ravi's type-checking because the compiler doesn't know about these operations; hence if you need to do such operations from C code, please ensure that values retain their types, or else just write plain Lua code.

Ravi does its best to validate operations performed via the Lua debug api; however, in general, the same caveats apply.

2.2.3 `integer` and `number` types

- `integer` and `number` types are automatically initialized to zero rather than `nil`
- Arithmetic operations on numeric types make use of type-specialized bytecodes that lead to better code-generation

2.2.4 `integer[]` and `number[]` array types

The array types (`number[]` and `integer[]`) are specializations of Lua table with some additional behaviour:

- Arrays must always be initialized:

  ```
  local t: number[] = {} -- okay
  local t2: number[]     -- error!
  ```

  This restriction is placed as otherwise the JIT code would need to insert tests to validate that the variable is not `nil`.

- Specialised operators to get/set from array types are implemented; these makes array-element access more efficient in JIT mode as the access can be inlined
- Operations on array types can be optimised to specialized bytecode only when the array type is known at compile time. Otherwise regular table access will be used, subject to runtime checks.
- The standard table operations on arrays are checked to ensure that the array type is not subverted
- Array types are not compatible with declared table variables, i.e. the following is not allowed:

  ```
  local t: table = {}
  local t2: number[] = t  -- error!

  local t3: number[] = {}
  local t4: table = t3    -- error!
  ```

  But the following is okay:

  ```
  local t5: number[] = {}
  local t6 = t5             -- t6 treated as table
  ```

  These restrictions are applied because declared table and array types generate optimized code that makes assumptions about keys and values. The generated code would be incorrect if the types were not as expected.

- Indices >= 1 should be used when accessing array-elements. Ravi arrays (and slices) have a hidden slot at index 0 for performance reasons, but this is not visible in `pairs()` or `ipairs()`, or when initializing an array using a literal initializer; only direct access via the `[]` operator can see this slot.
- An array will grow automatically (unless the array was created as fixed length using `table.intarray()` or `table.numarray()`) if the user sets the element just past the array length:

  ```
  local t: number[] = {} -- dynamic array
  t[1] = 4.2             -- okay, array grows by 1
  t[5] = 2.4             -- error! as attempt to set value
  ```

- It is an error to attempt to set an element that is beyond `len+1` on dynamic arrays; for fixed length arrays attempting to set elements at positions greater than `len` will cause an error.
- The current used length of the array is recorded and returned by the `len` operation
- The array only permits the right type of value to be assigned (this is also checked at runtime to allow compatibility with Lua)
- Accessing out of bounds elements will cause an error, except for setting the `len+1` element on dynamic arrays. There is a compiler option to omit bounds checking on reads.
- It is possible to pass arrays to functions and return arrays from functions. Arrays passed to functions appear as Lua tables inside those functions if the parameters are untyped - however the tables will still be subject to restrictions as above. If the parameters are typed then the arrays will be recognized at compile time:

```
local function f(a, b: integer[], c)
  -- Here a is dynamic type
  -- b is declared as integer[]
  -- c is also a dynamic type
  b[1] = a[1] -- Okay only if a is actually also integer[]
  b[1] = c[1] -- Will fail if c[1] cannot be converted to an integer
end

local a : integer[] = {1}
local b : integer[] = {}
local c = {1}

f(a,b,c)         -- ok as c[1] is integer
f(a,b, {'hi'})   -- error!
```

- Arrays returned from functions can be stored into appropriately typed local variables - there is validation that the types match:

```
local t: number[] = f() -- type will be checked at runtime
```

- Array types ignore `__index`, `__newindex` and `__len` metamethods.
- Array types cannot be set as metatables for other values.
- `pairs()` and `ipairs()` work on arrays as normal
- There is no way to delete an array element.
- The array data is stored in contiguous memory just like native C arrays; morever the garbage collector does not scan the array data

The following library functions allow creation of array types of defined length.

`table.intarray(num_elements, initial_value)` creates an integer array of specified size, and initializes with initial value. The return type is integer[]. The size of the array cannot be changed dynamically, i.e. it is fixed to the initial specified size. This allows slices to be created on such arrays.

`table.numarray(num_elements, initial_value)` creates an number array of specified size, and initializes with initial value. The return type is number[]. The size of the array cannot be changed dynamically, i.e. it is fixed to the initial specified size. This allows slices to be created on such arrays.

2.2.5 `table` type

A declared table (as shown below) has the following nuances.

- Like array types, a variable of `table` type must be initialized:

```
local t: table = {}
```

- Declared tables allow specialized opcodes for table gets involving integer and short literal string keys; these opcodes result in more efficient JIT code
- Array types are not compatible with declared table variables, i.e. the following is not allowed:

```
local t: table = {}
local t2: number[] = t -- error!
```

- When short string literals are used to access a table element, specialized bytecodes are generated that may be more efficiently JIT compiled:

```
local t: table = { name='dibyendu'}
print(t.name) -- The GETTABLE opcode is specialized in this case
```

- As with array types, specialized bytecodes are generated when integer keys are used

2.2.6 `string`, `closure` and user-defined types

These type-annotations have experimental support. They are not always statically enforced. Furthermore using these types does not affect the JIT code-generation, i.e. variables annotated using these types are still treated as dynamic types.

The scenarios where these type-annotations have an impact are:

- Function parameters containing these annotations lead to type assertions at runtime.
- The type assertion operator @ can be applied to these types - leading to runtime assertions.
- Annotating `local` declarations results in type assertions.
- All three types above allow `nil` assignment.

The main use case for these annotations is to help with type-checking of larger Ravi programs. These type checks, particularly the one for user defined types, are executed directly by the VM and hence are more efficient than performing the checks in other ways.

Examples:

```
-- Create a metatable
local mt = { __name='MyType'}

-- Register the metatable in Lua registry
debug.getregistry().MyType = mt

-- Create an object and assign the metatable as its type
local t = {}
setmetatable(t, mt)

-- Use the metatable name as the object's type
function x(s: MyType)
  local assert = assert
  assert(@MyType(s) == @MyType(t))
  assert(@MyType(t) == t)
end

-- Here we use the string type
function x(s1: string, s2: string)
  return @string( s1 .. s2 )
end

-- The following demonstrates an error caused by the type-checking
-- Note that this error is raised at runtime
function x()
  local s: string
  -- call a function that returns integer value
  -- and try to assign to s
  s = (function() return 1 end)()
end
x() -- will fail at runtime
```

2.2.7 Type Assertions

Ravi does not support defining new types, or structured types based on tables. This creates some practical issues when dynamic types are mixed with static types. For example:

```
local t = { 1,2,3 }
local i: integer = t[1] -- generates an error
```

The above code generates an error as the compiler does not know that the value in `t[1]` is an integer. However often we as programmers know the type that is expected, it would be nice to be able to tell the compiler what the expected type of `t[1]` is above. To enable this Ravi supports type assertion operators. A type assertion is introduced by the '`@`' symbol, which must be followed by the type name. So we can rewrite the above example as:

```
local t = { 1,2,3 }
local i: integer = @integer( t[1] )
```

The type assertion operator is a unary operator and binds to the expression following the operator. We use the parenthesis above to ensure that the type assertion is applied to `t[1]` rather than `t`. More examples are shown below:

```
local a: number[] = @number[] { 1,2,3 }
local t = { @number[] { 4,5,6 }, @integer[] { 6,7,8 } }
local a1: number[] = @number[]( t[1] )
local a2: integer[] = @integer[]( t[2] )
```

For a real example of how type assertions can be used, please have a look at the test program gaussian2.lua

2.2.8 Array Slices

Since release 0.6 Ravi supports array slices. An array slice allows a portion of a Ravi array to be treated as if it is an array - this allows efficient access to the underlying array-elements. The following new functions are available:

`table.slice(array, start_index, num_elements)` creates a slice from an existing *fixed size* array - allowing efficient access to the underlying array-elements.

Slices access the memory of the underlying array; hence a slice can only be created on fixed size arrays (constructed by `table.numarray()` or `table.intarray()`). This ensures that the array memory cannot be reallocated while a slice is referring to it. Ravi does not track the slices that refer to arrays - slices get garbage collected as normal.

Slices cannot extend the array size for the same reasons above.

The type of a slice is the same as that of the underlying array - hence slices get the same optimized JIT operations for array access.

Each slice holds an internal reference to the underlying array to ensure that the garbage collector does not reclaim the array while there are slices pointing to it.

For an example use of slices please see the matmul1_ravi.lua benchmark program in the repository. Note that this feature is highly experimental and not very well tested.

2.2.9 Type Annotation Examples

Example of code that works - you can copy this to the command line input:

```
function tryme()
  local i,j = 5,6
  return i,j
```

(continues on next page)

(continued from previous page)

```
end
local i:integer, j:integer = tryme(); print(i+j)
```

When values from a function call are assigned to a typed variable, an implicit type coercion takes place. In the above example an error would occur if the function returned values that could not converted to integers.

In the following example, the parameter `j` is defined as a `number`, hence it is an error to pass a value that cannot be converted to a `number`:

```
function tryme(j: number)
  for i=1,1000000000 do
    j = j+1
  end
  return j
end
print(tryme(0.0))
```

An example with arrays:

```
function tryme()
  local a : number[], j:number = {}
  for i=1,10 do
    a[i] = i
    j = j + a[i]
  end
  return j
end
print(tryme())
```

Another example using arrays. Here the function receives a parameter `arr` of type `number[]` - it would be an error to pass any other type to the function because only `number[]` types can be converted to `number[]` types:

```
function sum(arr: number[])
  local n: number = 0.0
  for i = 1,#arr do
    n = n + arr[i]
  end
  return n
end

print(sum(table.numarray(10, 2.0)))
```

The `table.numarray(n, initial_value)` creates a `number[]` of specified size and initializes the array with the given initial value.

2.3 The `defer` statement

A new addition to Ravi is the `defer` statement. The statement has the form:

```
defer
  block
end
```

Where `block` is a set of Lua statements.

The `defer` statement creates an anonymous `closure` that will be invoked when the enclosing scope is exited, whether normally or because of an error.

Example:

```
y = 0
function x()
  defer y = y + 1 end
  defer y = y + 1 end
end
x()
assert(y == 2)
```

`defer` statements are meant to be used for releasing resources in a deterministic manner. The syntax and functionality is inspired by the similar statement in the Go language. The implementation is based upon Lua 5.4.

Note that the `defer` statement should be considered a beta feature not yet ready for production use as it is undergoing testing.

2.4 API Functions in Lua

2.4.1 JIT API

auto mode in this mode the compiler decides when to compile a Lua function. The current implementation is very simple - any Lua function call is checked to see if the bytecodes contained in it can be compiled. If this is true then the function is compiled provided either a) function has a fornum loop, or b) it is largish (greater than 150 bytecodes) or c) it is being executed many times (> 50). Because of the simplistic behaviour performance the benefit of JIT compilation is only available if the JIT compiled functions will be executed many times so that the cost of JIT compilation can be amortized.

manual mode in this mode user must explicitly request compilation. This is the default mode. This mode is suitable for library developers who can pre compile the functions in library module table.

A JIT api is available with following functions:

`ravi.jit([b])` returns enabled setting of JIT compiler; also enables/disables the JIT compiler; defaults to true

`ravi.jitname()` returns an identifier for the JIT

`ravi.options()` returns a string with compiled options

`ravi.auto([b [, min_size [, min_executions]]])` returns setting of auto compilation and compilation thresholds; also sets the new settings if values are supplied; defaults are false, 150, 50.

`ravi.compile(func_or_table[, options])` compiles a Lua function (or functions if a table is supplied) if possible, returns `true` if compilation was successful for at least one function. `options` is an optional table with compilation options - in particular `omitArrayGetRangeCheck` - which disables range checks in array get operations to improve performance in some cases. Note that at present if the first argument is a table of functions and has more than 100 functions then only the first 100 will be compiled. You can invoke compile() repeatedly on the table until it returns false. Each invocation leads to a new module being created; any functions already compiled are skipped.

`ravi.iscompiled(func)` returns the JIT status of a function

`ravi.dumplua(func)` dumps the Lua bytecode of the function

`ravi.dumpir(func)` dumps the intermediate code of the compiled function; interpretation up to the JIT backend.

`ravi.optlevel([n])` sets optimization level (0, 1, 2, 3); the interpretation of this is up to the JIT backend.

`ravi.sizelevel([n])` sets LLVM size level (0, 1, 2); the interpretation of this is up to the JIT backend

`ravi.tracehook([b])` Enables support for line hooks via the debug api. Note that enabling this option will result in inefficient JIT as a call to a C function will be inserted at beginning of every Lua bytecode boundary; use this option only when you want to use the debug api to step through code line by line. Currently only supported by LLVM backend.

`ravi.verbosity([b])` Controls the amount of verbose messages generated during compilation.

2.5 Generational Garbage Collection

Ravi incorporates the generational garbage collector from Lua 5.4. Please refer to the Lua 5.4 manual regarding the api changes to support generational collection.

Note that by default, Ravi uses the incremental garbage collector. The generational collector is new and may have bugs in its implementation (a few bugs have been reported on Lua mailing lists, fixes are being applied to Ravi when applicable).

To switch to generational GC:

```
collectgarbage("generational")
```

To switch to incremental GC:

```
collectgarbage("incremental")
```

2.6 Random Number Generator

Ravi incorporates the new random number generator from Lua 5.4. Please refer to the Lua 5.4 manual for api changes in this area.

2.7 C API Extensions

Ravi provides following C API extensions:

```
LUA_API void  (ravi_pushcfastcall)(lua_State *L, void *ptr, int tag);

/* Allowed tags - subject to change. Max value is 128. Note that
   each tag requires special handling in ldo.c */
enum {
  RAVI_TFCF_EXP = 1,
  RAVI_TFCF_LOG = 2,
  RAVI_TFCF_D_D = 3,
};

/* Create an integer array (specialization of Lua table)
 * of given size and initialize array with supplied initial value
 */
LUA_API void ravi_create_integer_array(lua_State *L, int narray,
                                       lua_Integer initial_value);

/* Create an number array (specialization of Lua table)
```

(continues on next page)

(continued from previous page)

```
 * of given size and initialize array with supplied initial value
 */
LUA_API void ravi_create_number_array(lua_State *L, int narray,
                                      lua_Number initial_value);

/* Create a slice of an existing array
 * The original table containing the array is inserted into the
 * the slice as a value against special key so that
 * the parent table is not garbage collected while this array contains a
 * reference to it
 * The array slice starts at start but start-1 is also accessible because of the
 * implementation having array values starting at 0.
 * A slice must not attempt to release the data array as this is not owned by
 * it,
 * and in fact may point to garbage from a memory allocater's point of view.
 */
LUA_API void ravi_create_slice(lua_State *L, int idx, unsigned int start,
                               unsigned int len);

/* Tests if the argument is a number array
 */
LUA_API int ravi_is_number_array(lua_State *L, int idx);

/* Tests if the argument is a integer array
*/
LUA_API int ravi_is_integer_array(lua_State *L, int idx);

/* Get the raw data associated with the number array at idx.
 * Note that Ravi arrays have an extra element at offset 0 - this
 * function returns a pointer to &data[0]. The number of
 * array elements is returned in length.
 */
typedef struct {
  lua_Number *data;
  unsigned int length;
} Ravi_NumberArray;
LUA_API void ravi_get_number_array_rawdata(lua_State *L, int idx, Ravi_NumberArray␣
↪*array_data);

/* Get the raw data associated with the integer array at idx.
 * Note that Ravi arrays have an extra element at offset 0 - this
 * function returns a pointer to &data[0]. The number of
 * array elements is returned in length.
 */
typedef struct {
  lua_Integer *data;
  unsigned int length;
} Ravi_IntegerArray;
LUA_API void ravi_get_integer_array_rawdata(lua_State *L, int idx, Ravi_IntegerArray␣
↪*array_data);

/* API to set the output functions used by Lua / Ravi
 * This allows the default implementations to be overridden
 */
LUA_API void ravi_set_writefuncs(lua_State *L, ravi_Writestring writestr, ravi_
↪Writeline writeln, ravi_Writestringerror writestringerr);
```

(continues on next page)

(continued from previous page)

```
/* Following are the default implementations */
LUA_API void ravi_writestring(lua_State *L, const char *s, size_t len);
LUA_API void ravi_writeline(lua_State *L);
LUA_API void ravi_writestringerror(lua_State *L, const char *fmt, const char *p);

/* The debugger can set some data - but only once */
LUA_API void ravi_set_debugger_data(lua_State *L, void *data);
LUA_API void *ravi_get_debugger_data(lua_State *L);

/* Takes a function parameter and outputs the bytecodes to stdout */
LUA_API void ravi_dump_function(lua_State *L);
/* Takes a function parameter and returns a table of lines containing bytecodes for␣
↪the function */
LUA_API int ravi_list_code(lua_State *L);
/* Returns a table with various system limits */
LUA_API int ravi_get_limits(lua_State *L);
```

CHAPTER 3

Instructions for Building With MIR JIT support

Building with MIR support is straightforward as MIR is included in Ravi. On Linux or Mac OSX:

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=$HOME/Software/ravi -DCMAKE_BUILD_TYPE=Release ..
make install
```

That's it.

For Windows, assuming you have Visual Studio 2019 installed, you can build as follows:

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/Software/ravi -DCMAKE_BUILD_TYPE=Release ..
cmake --build . --config Release
```

CHAPTER 4

New Compiler Framework in Ravi

A new compiler framework in Ravi is now available as a preview feature.

The new framework allows both JIT and AOT compilation of Lua/Ravi code. A large subset of the language is supported.

The new compiler differs from the standard Lua/Ravi compiler in following ways:

- Unlike the standard compiler, the new compiler generates Abstract Syntax Trees (ASTs) in the parsing phase.
- A new linear Intermediate Representation (IR) is produced from the AST.
- Finally the IR is translated to C code, which can be JITed using the MIR backend or compiled ahead-of-time using a standard C compiler such as gcc, clang or MSVC.

Additional details regarding the new implementation can be found at the compiler project site.

The new compiler can be invoked in the following ways.

4.1 JIT Compilation

`compiler.load(code: string)` JIT compiles a chunk of code and returns a closure on the stack representing the compiled output.

`compiler.loadfile(filename: string)` Opens the named file and JIT compiles its content as a Lua chunk. Returns a closure on the stack representing the compiled output.

4.2 AOT Compilation

`compiler.compile(filename: string, mainfunc: string)` Compiles the contents of the given filename and generates C code. The function returns the generated code as a string. The supplied `mainfunc` will be the name of the main function in the generated code. The generated C code can be compiled using a C compiler to produce a shared library. Note that on Windows, you will need to provide the Ravi Import library as an argument to the linker when generating the shared library. See examples below.

package.load_ravi_lib(filename: string, mainfunc: string) Loads the shared library with the given filename, and returns the function identified as `mainfunc`. The returned function can be executed to effectively run the contents of the original input file.

4.3 Limitations

There are some limitations in this approach that you need to be aware of.

- The generated code does not have Lua bytecode. Hence Lua interpreter / debugger cannot do anything with these functions.
- The generated code is dependent upon the VM specifics - in particular it relies upon the VM stack setup, the call stack setup, etc. There are additional support functions needed by the compiled code, including data types supported by Ravi. For these reasons, only Ravi can execute the AOT compiled code.
- There is no support for Lua hooks in the generated code, as there are no Lua bytecodes.
- Coroutines are not supported; the generated code can only be executed on the main thread, and moreover cannot yield either.
- Var args are not yet supported.
- The 'defer' statement is not yet implemented.
- The Lua debug interface cannot be used to manipulate objects inside the compiled code.
- The compiler is still in early stages of development, hence bugs can be expected.

4.4 JIT Examples

Here is an example of a simple JIT compile session:

```
f = compiler.load("print 'hello'")
assert(f and type(f) == 'function')
f()
```

We can inspect the function `f`:

```
> ravi.dumplua(f)

main <?:0,0> (0 instructions at 000001300E6C9180)
0 params, 2 slots, 1 upvalue, 0 locals, 2 constants, 0 functions
constants (2) for 000001300E6C9180:
      1       "print"
      2       "hello"
locals (0) for 000001300E6C9180:
upvalues (1) for 000001300E6C9180:
      0       -       1       0
```

For more JIT examples, please have a look at compiler test cases.

4.5 AOT Examples

For AOT compilation, you will need a C compiler, such as `clang`, `gcc` or `MSVC`.

Here is an example session on Windows:

```
-- The session must be started inside a command prompt with 64-bit MSVC enabled
-- We assume Ravi is compiled and installed at /Software/ravi on the local drive.

-- Helper functions
function readall(file)
  local f = assert(io.open(file, "rb"))
  local content = f:read("*all")
  f:close()
  return content
end

function writestring(file,str)
  local f = io.open(file,'w')
  f:write(str)
  f:close()
end

function comptoC(inputfile, outputfile)
  local chunk = readall(inputfile)
  local compiled = compiler.compile(chunk, 'mymain')
  if not compiled then
      error ('Failed to compile')
  end
  writestring(outputfile, compiled)
end

comptoC('gaussian2_lib.lua', 'gaussian2_lib.c')
-- Note compiler options may need to change for Release vs Debug builds
assert(os.execute("cl /c /Os /D WIN32 /DMD gaussian2_lib.c"))
assert(os.execute("link /LIBPATH:/Software/ravi/lib libravi.lib /DLL /MACHINE:X64 /
↪OUT:gaussian2_lib.dll gaussian2_lib.obj"))
local f = package.load_ravi_lib('gaussian2_lib.dll', 'mymain') -- load shared library␣
↪and obtain reference to mymain
assert(f and type(f) == 'function')
local glib = f() -- execute mymain
assert(glib and type(glib) == 'table')
```

For the same example in a Linux environment, have a look at the AOT Examples.

CHAPTER 5

Introduction to Lua

5.1 Introduction

Lua is a small but powerful interpreted language that is implemented as a C library. This guide is meant to help you quickly become familiar with the main features of Lua. This guide assumes you know C, C++, or Java, and perhaps a scripting language like Python - it is not a beginner's guide. Nor is it a tutorial for Lua.

5.2 Key Features of Lua

- Lua versions matter
- Lua is dynamically typed like Python
- By default variables in Lua are global unless declared local
- Lua has no line terminators
- There is a single complex / aggregate type called a 'table', which combines hash table/map and array features
- Functions in Lua are values stored in variables; in particular functions do not have names
- Globals in Lua are just values stored in a special Lua table
- Functions in Lua are closures - they can capture variables from outer scope and such variables live on even though the surrounding scope is no longer alive
- Lua functions can return multiple values
- Lua has integer (since 5.3) and floating point types that map to native C types
- A special `nil` value represents non-existent value
- Any value that is not `false` or `nil` is true
- The result of logical `and` and logical `or` is not true or false; these operators select one of the values
- `'~='` is not equals operator and `'..'` is string concatenation operator

- Lua has some nice syntactic sugar for tables and functions
- A Lua script is called a chunk - and is the unit of compilation in Lua
- The Lua stack is a heap allocated structure - and you can think of Lua as a library that manipulates this stack
- Lua functions can be yielded from and resumed later on, i.e., Lua supports coroutines
- Lua is single threaded but its VM is small and encapsulated in a single data structure - hence each OS thread can be given its own Lua VM
- Lua's error handling is based on C setjmp/longjmp, and errors are caught via a special function call mechanism
- Lua has a meta mechanism that enables a DIY class / object system with some syntactic sugar to make it look nice
- Lua supports operator overloading via 'meta' methods
- You can create user defined types in C and make them available in Lua
- Lua compiles code to bytecode before execution
- Lua bytecode is not officially documented and changes from one Lua version to another; moreover the binary dump of the bytecodes is not portable across architectures and also can change between versions
- Lua's compiler is designed to be fast and frugal - it generates code as it parses, there is no intermediate AST construction
- Like C, Lua comes with a very small standard library - in fact Lua's standard library is just a wrapper for C standard library plus some basic utilities for Lua
- Lua's standard library includes pattern matching for strings in which the patterns themselves are strings, rather like regular expressions in Python or Perl, but simpler.
- Lua provides a debug API that can be used to manipulate Lua's internals to a degree - and can be used to implement a debugger
- Lua has an incremental garbage collector
- Lua is Open Source but has a closed development model - external contributions are not possible
- LuaJIT is a JIT compiler for Lua but features an optional high performance C interface mechanism that makes it incompatible with Lua

In the rest of this document I will expand on each of these aspects of Lua.

5.3 Lua versions matter

For all practical purposes only Lua versions 5.1, 5.2 and 5.3 matter. Note however that each of these is considered a major version and therefore is not fully backward compatible (e.g. Lua 5.3 cannot necessarily run Lua 5.1 code) although there is a large common subset.

- Lua 5.2 has a new mechanism for resolving undeclared variables compared to 5.1
- Lua 5.3 has integer number subtype and bitwise operators that did not exist in 5.1 or 5.2
- LuaJIT is 5.1 based but supports a large subset of 5.2 features with some notable exceptions such as the change mentioned above

Mostly what this document covers should be applicable to all these versions, except as otherwise noted.

5.4 Lua is dynamically typed

This means that values have types but variables do not. Example:

```
x = 1 -- x holds an integer
x = 5.0 -- x now holds a floating pont value
x = {} -- x now holds an empty table
x = function() end -- x now holds a function with empty body
```

5.5 Variables are global unless declared local

In the example above, `x` is global. But saying:

```
local x = 1
```

makes `x` local, i.e. its scope and visibility is constrained to the enclosing block of code, and any nested blocks. Note that local variables avoid a lookup in the 'global' table and hence are more efficient. Thus it is common practice to cache values in local variables. For example, `print` is a global function - and following creates a local variable that caches it:

```
local print = print -- caches global print() function
print('hello world!') -- calls the same function as global print()
```

There are some exceptions to the rule:

- the iterator variables declared in a `for` loop are implicitly local.
- function parameters are local to the function

5.6 Lua has no line terminators

Strictly speaking you can terminate Lua statements using `;`. However it is not necessary except in some cases to avoid ambiguity.

This design has some consequences that took me by surprise:

```
local x y = 5
```

Above creates a local variable `x` and sets a global `y` to `5`. Because it actually parses as:

```
local x
y = 5
```

5.7 The 'table' type

Lua's only complex / aggregate data type is a table. Tables are used for many things in Lua, even internally within Lua. Here are some examples:

```
local a = {} -- creates an empty table
local b = {10,20,30} -- creates a table with three array elements at positions 1,2,3
                     -- this is short cut for:
                     -- local b = {}
                     -- b[1] = 10
                     -- b[2] = 20
                     -- b[3] = 30
local c = { name='Ravi' } -- creates a table with one hash map entry
                          -- this is short cut for:
                          -- local c = {}
                          -- c['name'] = 'Ravi'
```

Internally the table is a composite hash table / array structure. Consecutive values starting at integer index 1 are inserted into the array, else the values go into the hash table. Hence, in the example below:

```
local t = {}
t[1] = 20 -- goes into array
t[2] = 10 -- goes into array
t[100] = 1 -- goes into hash table as not consecutive
t.name = 'Ravi' -- goes into hash table
                -- t.name is syntactic sugar for t['name']
```

To iterate over array values you can write:

```
for i = 1,#t do
  print(t[i])
end
```

Note that above will only print 20,10.

To iterate over all values write:

```
for k,v in pairs(t) do
  print(k,v)
end
```

Unfortunately, you need to get a good understanding of when values will go into the array part of a table, because some Lua library functions work only on the array part. Example:

```
table.sort(t)
```

You will see that only values at indices 1 and 2 were sorted. Another frequent problem is that the only way to reliably know the total number of elements in a table is to count the values. The # operator returns the length of the consecutive array elements starting at index 1.

5.8 Functions are values stored in variables

You already saw that we can write:

```
local x = function()
          end
```

This creates a function and stores it in local variable `x`. This is the same as:

```
local function x()
end
```

Omitting the `local` keyword would create `x` in global scope.

Functions can be defined within functions - in fact all Lua functions are defined within a 'chunk' of code, which gets wrapped inside a Lua function.

Internally a function has a 'prototype' that holds the compiled code and other meta data regarding the function. An instance of the function in created when the code executes. You can think of the 'prototype' as the 'class' of the function, and the function instance is akin to an object created from this class.

5.9 Globals are just values in a special table

Globals are handled in an interesting way. Whenever a name is used that is not found in any of the enclosing scopes and is not declared `local`, then Lua will access/create a variable in a table accessed by the name `_ENV` (this applies to Lua 5.2 and above - Lua 5.1 had a different mechanism). Actually `_ENV` is just a captured value that points to a special table in Lua by default. This table access becomes evident when you look at the bytecode generated for some Lua code:

```
function hello()
  print('hello world')
end
```

Generates following (in Lua 5.3):

```
function <stdin:1,3> (4 instructions at 00000151C0AA9530)
0 params, 2 slots, 1 upvalue, 0 locals, 2 constants, 0 functions
      1       [2]     GETTABUP        0 0 -1  ; _ENV "print"
      2       [2]     LOADK           1 -2    ; "hello world"
      3       [2]     CALL            0 2 1
      4       [3]     RETURN          0 1
constants (2) for 00000151C0AA9530:
      1       "print"
      2       "hello world"
locals (0) for 00000151C0AA9530:
upvalues (1) for 00000151C0AA9530:
      0       _ENV    0       0
```

The `GETTABUP` instruction looks up the name 'print' in the captured table variable `_ENV`. Lua uses the term 'upvalue' for captured variables.

5.10 Functions in Lua are closures

Lua functions can reference variables in outer scopes - and such references can be captured by the function so that even if the outer scope does not exist anymore the variable still lives on:

```
-- x() returns two anonymous functions
x = function()
  local a = 1
  return  function(b)
            a = a+b
            return a
```

(continues on next page)

(continued from previous page)

```
          end,
          function(b)
            a = a+b
            return a
          end
end

-- call x
m,n = x()
m(1) -- returns 2
n(1) -- returns 3
```

In the example above, the local variable `a` in function `x()` is captured inside the two anonymous functions that reference it. You can see this if you dump Lua 5.3 bytecode for `m`:

```
function <stdin:1,1> (6 instructions at 00000151C0AD3AB0)
1 param, 2 slots, 1 upvalue, 1 local, 0 constants, 0 functions
      1       [1]     GETUPVAL        1 0     ; a
      2       [1]     ADD             1 1 0
      3       [1]     SETUPVAL        1 0     ; a
      4       [1]     GETUPVAL        1 0     ; a
      5       [1]     RETURN          1 2
      6       [1]     RETURN          0 1
constants (0) for 00000151C0AD3AB0:
locals (1) for 00000151C0AD3AB0:
      0       b       1       7
upvalues (1) for 00000151C0AD3AB0:
      0       a       1       0
```

The `GETUPVAL` and `SETUPVAL` instructions access captured variables or upvalues as they are known in Lua.

5.11 Lua functions can return multiple values

An example of this already appeared above. Here is another:

```
function foo()
  return 1, 'text'
end

x,y = foo()
```

5.12 Lua has integer and floating point numeric types

Since Lua 5.3 Lua's number type has integer and floating point representations. This is automatically managed; however a library function is provided to tell you what Lua thinks the number type is.

```
x = 1   -- integer
y = 4.2 -- double

print(math.type(x)) -- says 'integer'
print(math.type(y)) -- says 'float'
```

On 64-bit architecture by default an integer is represented as C `int64_t` and floating point as `double`. The representation of the numeric type as native C types is one of the secrets of Lua's performance, as the numeric types do not require 'boxing'.

In Lua 5.3, there is a special division operator `//` that does integer division if the operands are both integer. Example:

```
x = 4
y = 3

print(x//y) -- integer division results in 0
print(x/y) -- floating division results in 1.3333333333333
```

Note that officially the `//` operator does floor division, hence if one or both of its operands is floating point then the result is also a floating point representing the floor of the division of its operands.

Having integer types has also made it natural to have support for bitwise operators in Lua 5.3.

5.13 A special `nil` value represents non-existent value

Lua has special value `nil` that represents no value, and evaluates to false in boolean expressions.

5.14 Any value that is not `false` or `nil` is true

As mentioned above `nil` evaluates to false.

5.15 Logical `and` and logical `or` select one of the values

When you perform a logical `and` or `or` the result is not boolean; these operators select one of the values. This is best illustrated via examples:

```
false or 'hello'              -- selects 'hello'
'hello' and 'world'           -- selects 'world'
false and 'hello'             -- selects false
nil or false                  -- selects false
nil and false                 -- selects nil
```

- `and` selects the first value if it evaluates to false else the second value.
- `or` selects the first value if it evaluates to true else the second value.

5.16 `'~='` is not equals operator and `'..'` is string concatenation operator

For example:

```
print(1 ~= 2)                 -- prints 'true'
print('hello ' .. 'world!')   -- prints 'hello world!')
```

5.17 Lua has some nice syntactic sugar for tables and functions

If you are calling a Lua function with a single string or table argument then the parenthesis can be omitted:

```
print 'hello world' -- syntactic sugar for print('hello world')
options { verbose=true, debug=true } -- syntactic sugar for options( { ... } )
```

Above is often used to create a DSL. For instance, see:

- Lua's bug list
- Premake - a tool similar to CMake

You have already seen that:

```
t = { surname = 'majumdar' }        -- t.surname is sugar for t['surname']
t.name = 'dibyendu'                 -- syntactic sugar for t['name'] = 'dibyendu'
```

A useful use case for tables is as modules. Thus a standard library module like `math` is simply a table of functions. Here is an example:

```
module = { print, type }
module.print('hello')
module.print 'hello'
module.type('hello')
```

Finally, you can emulate an object oriented syntax using the **:** operator:

```
x:foo('hello')                     -- syntactic sugar for foo(x, 'hello')
```

As we shall see, this feature enables Lua to support object orientation.

5.18 A Lua script is called a chunk - and is the unit of compilation in Lua

When you present a script to Lua, it is compiled. The script can be a file or a string. Internally the content of the script is wrapped inside a Lua function. So that means that a script can have `local` variables, as these live in the wrapping function.

It is common practice for scripts to return a table of functions - as then the script can be treated as a module. There is a library function 'require' which loads a script as a module.

Suppose you have following script saved in a file `sample.lua`:

```
-- sample script
local function foo() end
local function bar() end

return { foo=foo, bar=bar }       -- i.e. ['foo'] = foo, ['bar'] = bar
```

Above script returns a table containing two functions.

Now another script can load this as follows:

```
local sample = require 'sample' -- Will call sample.lua script and save its table of␣
↪functions
```

The library function `require()` does more than what is described above, of course. For instance it ensures that the module is only loaded once, and it uses various search paths to locate the script. It can even load C modules. Anyway, now the table returned from the sample script is stored in the local variable 'sample' and we can write:

```
sample.foo()
sample.bar()
```

5.19 The Lua stack is a heap allocated structure

Lua's code operates on heap allocated stacks, rather than the native machine stack. Since Lua is also a C library you can think of Lua as a library that manipulates the heap allocated stacks. In particular, Lua's C api exposes the Lua stack, and requires you to push/pop values on the stack; this approach is unique to Lua.

5.20 Lua functions can be yielded from and resumed later

Lua allows functions to be suspended and resumed. The function suspends itself by calling a library function to yield. Sometime later the function may be resumed by the caller or something else - when resumed, the Lua function continues from the point of suspension.

When yielding you can pass values back to the caller. Similarly when resuming the caller can pass values to the function.

This is perhaps the most advanced feature in Lua, and not one that can be easily demonstrated in a simple way. Following is the simplest example I could think of.

```
function test()
  local message = coroutine.yield('hello')
  print(message)
end

-- create a new Lua stack (thread)
thread = coroutine.create(test)

-- start the coroutine
status,message = coroutine.resume(thread) -- initial start

-- coroutine suspended so we have got control back
-- the coroutine yielded message to us - lets print it
print(message) -- says 'hello', the value returned by yield

-- Resume the coroutine / send it the message 'world'
status,message = coroutine.resume(thread, 'world')

-- above will print 'world'
-- status above will be true
-- but now the coroutine has ended so further calls to resume will return status as␣
↪false
```

By the fact that 'hello' is printed before 'world' we can tell that the coroutine was suspended and then resumed.

In the Lua documentation, the return value from `coroutine.create()` is called a `thread`. However don't confuse this with threads as in C++ or Java. You can think of a Lua `thread` as just another Lua stack. Basically whenever Lua executes any code - the code operates on a Lua stack. Initially there is only one stack (main thread). When you create a coroutine, a new stack is allocated, and the all functions called from the coroutine will operate on this new stack. Since the Lua stack is a heap allocated structure - suspending the coroutine is equivalent to returning back to the caller using a `longjmp()`. The stack is preserved, so that the function that yielded can be resumed later from wherever it suspended itself.

There is no automatic scheduling of Lua coroutines, a coroutine has to be explicitly resumed by the program.

Note also that Lua is single threaded - so you cannot execute the different Lua stacks in parallel in multiple OS threads; a particular Lua instance always runs in a single OS thread. At any point in time only one Lua stack can be active.

5.21 Lua's error handling is based on C setjmp/longjmp

You raise an error in Lua by calling library functions `error()` or `assert()`. Lua library functions can also raise errors. When an error is raised Lua does a C `longjmp` to the nearest location in the call stack where the caller used a 'protected call'. A 'protected call' is a function calling mechanism that does a C `setjmp`.

Here is how a protected call is done:

```
function foo(message)
  -- raise error if message is nil
  if not message then
    error('message expected')
  else
    print(message)
    return 4.2
  end
end

-- call foo('hello') in protected mode
-- this is done using the Lua library function pcall()
status,returnvalue = pcall(foo, 'hello')

-- since this call should succeed, status will be true
-- returnvalue should contain 4.2
assert(returnvalue == 4.2)

-- call foo() without arguments in protected mode
status, returnvalue = pcall(foo)
-- above will fail and status will be false
-- But returnvalue will now have the error message

assert(not status)
print(returnvalue)
-- above prints 'message expected'
```

The Lua error handling mechanism has following issues:

- The code that can raise errors must be encapsulated in a function as `pcall()` can only call functions
- The return values from `pcall()` depend upon whether the call terminated normally or due to an error - so caller needs to check the status of the call and only then proceed
- On raising an error the `longjmp` unwinds the stack - there is no mechanism for any intermediate objects to perform cleanup as is possible in C++ using destructors, or in Java, C++, Python using `finally` blocks, or as done by the `defer` statement in Go
- You can setup a finalizer on Lua user types that will eventually execute when the value is garbage collected - this is typically used to free up memory used by the value - but you have no control over when the finalizer will run, hence relying upon finalizers for cleanup is problematic

5.22 Lua is single threaded but each OS thread can be given its own Lua VM

All of Lua's VM is encapsulated in a single data structure - the Lua State. Lua does not have global state. Thus, you can create as many Lua instances in a single process as you want. Since the VM is so small it is quite feasible to allocate a Lua VM per OS thread.

5.23 Lua has a meta mechanism that enables a DIY class / object system

Firstly simple object oriented method calls can be emulated in Lua by relying upon the **:** operator described earlier. Recollect that:

```
object:method(arg)                          -- is syntactic sugar for method(object,␣
↪arg)
```

The next bit of syntactic sugar is shown below:

```
object = {}
function object:method(arg)
  print('method called with ', self, arg) -- self is automatic parameter and is␣
↪really object
end
```

Above is syntactic sugar for following equivalent code:

```
object = {}
object.method = function(self, arg)
  print('method called with ', self, arg)
end
```

As the object is passed as the `self` argument, the method can access other properties and methods contained in the object, which is just a normal table.

```
object:method('hello')                    -- calls method(object, 'hello')
```

This mechanism is fine for Lua code but doesn't work for user defined values created in C. Lua supports another more sophisticated approach that makes use of a facility in Lua called metatables. A `metatable` is simply an ordinary table that you can associate with any table or user defined type created in C code. The advantage of using the `metatable` approach is that it also works for user defined types created in C code. Here we will look at how it can be applied to Lua code.

Keeping to the same example above, this approach requires us to populate a `metatable` with the methods. We can think of the `metatable` as the class of the object.:

```
Class = {}                  -- our metatable
Class.__index = Class       -- This is a meta property (see description below)

-- define method function in Class
function Class:method(arg)
  print('method called with ', self, arg)
end

```

(continues on next page)

(continued from previous page)

```
-- define factory for creating new objects
function Class:new()
  local object = {}
  setmetatable(object, self)
  return object
end
```

- Notice that we set the field `__index` in the `Class` table to point to itself. This is a special field that Lua recognizes and whenever you access a field in an object, if the field is not found in the object and if the object has a `metatable` with `__index` field set, the Lua will lookup the field you want in the `metatable`.
- Secondly we set `Class` to be the `metatable` for the object in the new method.

As a result of above, in the example below:

```
object = Class:new()
object:method('hello')
```

Lua notices that there is no `method` field in object. But object has a `metatable` assigned to it, and this has `__index` set, so Lua looks up `Class.__index['method']` and finds the method.

Essentially this approach enables the concept of a shared class (e.g. Class in this example) that holds common fields. These fields can be methods or other ordinary values - and since the `metatable` is shared by all objects created using the `Class:new()` method, then we have a simple OO system!

This feature can be extended to support inheritance as well, but personally I do not find this useful, and suggest you look up Lua documentation if you want to play with inheritance. My advice is to avoid implementing complex object systems in Lua. However, the `metatable` approach is invaluable for user defined types created in C as these types can be used in more typesafe manner by using OO notation.

CHAPTER 6

Lua 5.3 Bytecode Reference

This is my attempt to bring up to date the Lua bytecode reference. Note that this is work in progress. Following copyrights are acknowledged:

```
A No-Frills Introduction to Lua 5.1 VM Instructions
  by Kein-Hong Man, esq. <khman AT users.sf.net>
  Version 0.1, 2006-03-13
```

6.1 Lua Stack and Registers

Lua employs two stacks. The `Callinfo` stack tracks activation frames. There is the secondary stack `L->stack` that is an array of `TValue` objects. The `Callinfo` objects index into this array. Registers are basically slots in the `L->stack` array.

When a function is called - the stack is setup as follows:

```
stack
|             function reference
|             var arg 1
|             ...
|             var arg n
| base->      fixed arg 1
|             ...
```

(continues on next page)

(continued from previous page)

```
|               fixed arg n
|               local 1
|               ...
|               local n
|               temporaries
|               ...
|   top->
|
V
```

So `top` is just past the registers needed by the function. The number of registers is determined based on parameters, locals and temporaries.

For each Lua function, the `base` of the stack is set to the first fixed parameter or local. All register addressing is done as offset from `base` - so `R(0)` is at `base+0` on the stack.

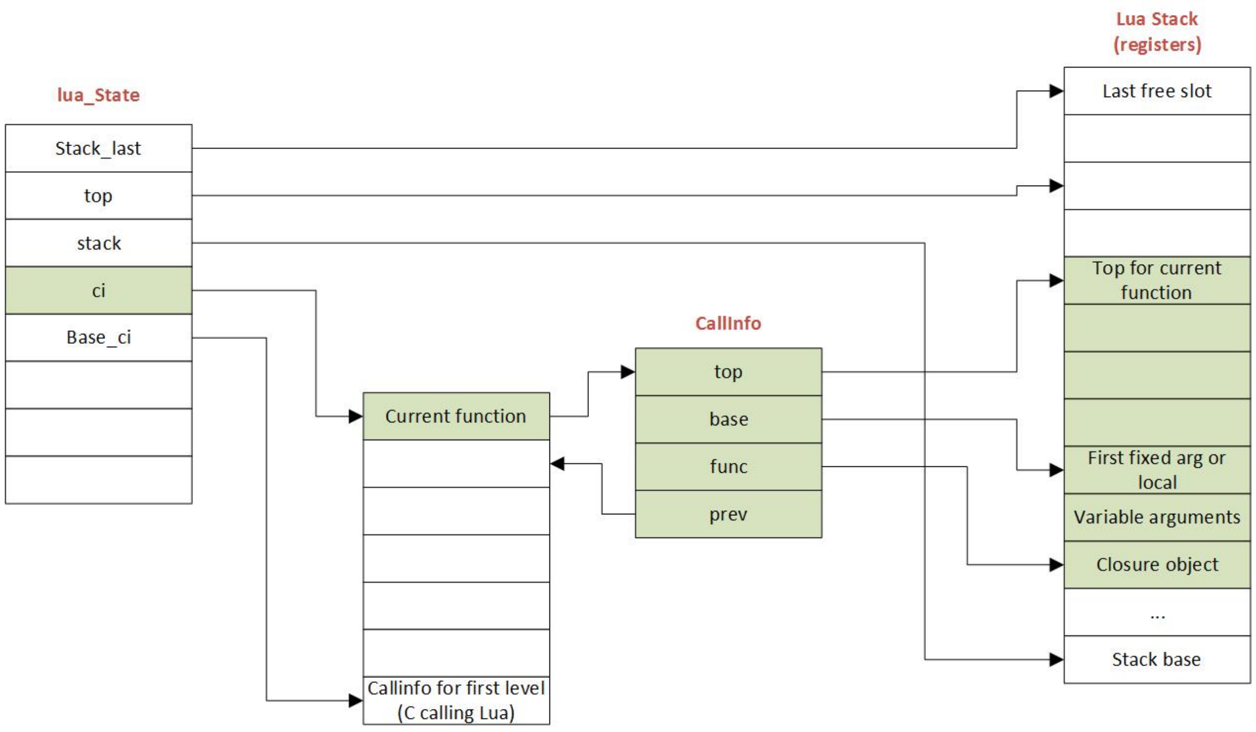

Fig. 1: The figure above shows how the stack is related to other Lua objects.

When the function returns the return values are copied to location starting at the function reference.

6.2 Instruction Notation

R(A) Register A (specified in instruction field A)

R(B) Register B (specified in instruction field B)

R(C) Register C (specified in instruction field C)

PC Program Counter

Kst(n) Element n in the constant list

Upvalue[n] Name of upvalue with index n

Gbl[sym] Global variable indexed by symbol sym

RK(B) Register B or a constant index

RK(C) Register C or a constant index

sBx Signed displacement (in field sBx) for all kinds of jumps

6.3 Instruction Summary

Lua bytecode instructions are 32-bits in size. All instructions have an opcode in the first 6 bits. Instructions can have the following fields:

```
'A' : 8 bits
'B' : 9 bits
'C' : 9 bits
'Ax' : 26 bits ('A', 'B', and 'C' together)
'Bx' : 18 bits ('B' and 'C' together)
'sBx' : signed Bx
```

A signed argument is represented in excess K; that is, the number value is the unsigned value minus K. K is exactly the maximum value for that argument (so that -max is represented by 0, and +max is represented by 2*max), which is half the maximum for the corresponding unsigned argument.

Note that B and C operands need to have an extra bit compared to A. This is because B and C can reference registers or constants, and the extra bit is used to decide which one. But A always references registers so it doesn't need the extra bit.

Opcode	Description
MOVE	Copy a value between registers
LOADK	Load a constant into a register
LOADKX	Load a constant into a register
LOADBOOL	Load a boolean into a register
LOADNIL	Load nil values into a range of registers
GETUPVAL	Read an upvalue into a register
GETTABUP	Read a value from table in up-value into a register
GETTABLE	Read a table element into a register
SETTABUP	Write a register value into table in up-value
SETUPVAL	Write a register value into an upvalue
SETTABLE	Write a register value into a table element
NEWTABLE	Create a new table
SELF	Prepare an object method for calling
ADD	Addition operator
SUB	Subtraction operator
MUL	Multiplication operator
MOD	Modulus (remainder) operator
POW	Exponentation operator
DIV	Division operator
IDIV	Integer division operator
BAND	Bit-wise AND operator

Continued on next page

Table 1 – continued from previous page

Opcode	Description
BOR	Bit-wise OR operator
BXOR	Bit-wise Exclusive OR operator
SHL	Shift bits left
SHR	Shift bits right
UNM	Unary minus
BNOT	Bit-wise NOT operator
NOT	Logical NOT operator
LEN	Length operator
CONCAT	Concatenate a range of registers
JMP	Unconditional jump
EQ	Equality test, with conditional jump
LT	Less than test, with conditional jump
LE	Less than or equal to test, with conditional jump
TEST	Boolean test, with conditional jump
TESTSET	Boolean test, with conditional jump and assignment
CALL	Call a closure
TAILCALL	Perform a tail call
RETURN	Return from function call
FORLOOP	Iterate a numeric for loop
FORPREP	Initialization for a numeric for loop
TFORLOOP	Iterate a generic for loop
TFORCALL	Initialization for a generic for loop
SETLIST	Set a range of array elements for a table
CLOSURE	Create a closure of a function prototype
VARARG	Assign vararg function arguments to registers

6.4 OP_CALL instruction

6.4.1 Syntax

```
CALL A B C    R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1))
```

6.4.2 Description

Performs a function call, with register R(A) holding the reference to the function object to be called. Parameters to the function are placed in the registers following R(A). If B is 1, the function has no parameters. If B is 2 or more, there are (B-1) parameters. If B >= 2, then upon entry to the called function, R(A+1) will become the `base`.

If B is 0, then B = 'top', i.e., the function parameters range from R(A+1) to the top of the stack. This form is used when the number of parameters to pass is set by the previous VM instruction, which has to be one of `OP_CALL` or `OP_VARARG`.

If C is 1, no return results are saved. If C is 2 or more, (C-1) return values are saved. If C == 0, then 'top' is set to last_result+1, so that the next open instruction (`OP_CALL`, `OP_RETURN`, `OP_SETLIST`) can use 'top'.

6.4.3 Examples

Example of `OP_VARARG` followed by `OP_CALL`:

```
function y(...) print(...) end

1 [1] GETTABUP  0 0 -1  ; _ENV "print"
2 [1] VARARG    1 0     ; VARARG will set L->top
3 [1] CALL      0 0 1   ; B=0 so L->top set by previous instruction
4 [1] RETURN    0 1
```

Example of `OP_CALL` followed by `OP_CALL`:

```
function z1() y(x()) end

1 [1] GETTABUP  0 0 -1  ; _ENV "y"
2 [1] GETTABUP  1 0 -2  ; _ENV "x"
3 [1] CALL      1 1 0   ; C=0 so return values indicated by L->top
4 [1] CALL      0 0 1   ; B=0 so L->top set by previous instruction
5 [1] RETURN    0 1
```

Thus upon entry to a function `base` is always the location of the first fixed parameter if any or else `local` if any. The three possibilities are shown below.

```
                                     Two variable args and 1     Two variable args␣
↪and no
Caller   One fixed arg               fixed arg                   fixed args
R(A)     CI->func  [ function    ]   CI->func  [ function    ]   CI->func [ function ␣
↪ ]
R(A+1)   CI->base  [ fixed arg 1 ]             [ var arg 1   ]            [ var arg 1␣
↪ ]
R(A+2)             [ local 1     ]             [ var arg 2   ]            [ var arg 2␣
↪ ]
R(A+3)                               CI->base  [ fixed arg 1 ]   CI->base [ local 1  ␣
↪ ]
R(A+4)                                         [ local 1     ]
```

Results returned by the function call are placed in a range of registers starting from `CI->func`. If `C` is `1`, no return results are saved. If `C` is 2 or more, `(C-1)` return values are saved. If `C` is `0`, then multiple return results are saved. In this case the number of values to save is determined by one of following ways:

- A C function returns an integer value indicating number of results returned so for C function calls this is used (see the value of `n` passed to luaD_poscall() in luaD_precall())
- For Lua functions, the results are saved by the called function's `OP_RETURN` instruction.

6.4.4 More examples

```
x=function() y() end
```

Produces:

```
function <stdin:1,1> (3 instructions at 000000CECB2BE040)
0 params, 2 slots, 1 upvalue, 0 locals, 1 constant, 0 functions
  1        [1]      GETTABUP         0  0 -1   ; _ENV "y"
  2        [1]      CALL             0  1  1
  3        [1]      RETURN           0  1
constants (1) for 000000CECB2BE040:
  1        "y"
locals (0) for 000000CECB2BE040:
```

(continues on next page)

(continued from previous page)

```
upvalues (1) for 000000CECB2BE040:
  0       _ENV    0       0
```

In line [2], the call has zero parameters (field B is 1), zero results are retained (field C is 1), while register 0 temporarily holds the reference to the function object from global y. Next we see a function call with multiple parameters or arguments:

```
x=function() z(1,2,3) end
```

Generates:

```
function <stdin:1,1> (6 instructions at 000000CECB2D7BC0)
0 params, 4 slots, 1 upvalue, 0 locals, 4 constants, 0 functions
  1       [1]     GETTABUP        0 0 -1  ; _ENV "z"
  2       [1]     LOADK           1 -2    ; 1
  3       [1]     LOADK           2 -3    ; 2
  4       [1]     LOADK           3 -4    ; 3
  5       [1]     CALL            0 4 1
  6       [1]     RETURN          0 1
constants (4) for 000000CECB2D7BC0:
  1       "z"
  2       1
  3       2
  4       3
locals (0) for 000000CECB2D7BC0:
upvalues (1) for 000000CECB2D7BC0:
  0       _ENV    0       0
```

Lines [1] to [4] loads the function reference and the arguments in order, then line [5] makes the call with an operand B value of 4, which means there are 3 parameters. Since the call statement is not assigned to anything, no return results need to be retained, hence field C is 1. Here is an example that uses multiple parameters and multiple return values:

```
x=function() local p,q,r,s = z(y()) end
```

Produces:

```
function <stdin:1,1> (5 instructions at 000000CECB2D6CC0)
0 params, 4 slots, 1 upvalue, 4 locals, 2 constants, 0 functions
  1       [1]     GETTABUP        0 0 -1  ; _ENV "z"
  2       [1]     GETTABUP        1 0 -2  ; _ENV "y"
  3       [1]     CALL            1 1 0
  4       [1]     CALL            0 0 5
  5       [1]     RETURN          0 1
constants (2) for 000000CECB2D6CC0:
  1       "z"
  2       "y"
locals (4) for 000000CECB2D6CC0:
  0       p       5       6
  1       q       5       6
  2       r       5       6
  3       s       5       6
upvalues (1) for 000000CECB2D6CC0:
  0       _ENV    0       0
```

First, the function references are retrieved (lines [1] and [2]), then function y is called first (temporary register 1). The CALL has a field C of 0, meaning multiple return values are accepted. These return values become the parameters to function z, and so in line [4], field B of the CALL instruction is 0, signifying multiple parameters. After the call to function z, 4 results are retained, so field C in line [4] is 5. Finally, here is an example with calls to standard library functions:

```
x=function() print(string.char(64)) end
```

Leads to:

```
function <stdin:1,1> (7 instructions at 000000CECB2D6220)
0 params, 3 slots, 1 upvalue, 0 locals, 4 constants, 0 functions
  1        [1]      GETTABUP        0 0 -1  ; _ENV "print"
  2        [1]      GETTABUP        1 0 -2  ; _ENV "string"
  3        [1]      GETTABLE        1 1 -3  ; "char"
  4        [1]      LOADK           2 -4    ; 64
  5        [1]      CALL            1 2 0
  6        [1]      CALL            0 0 1
  7        [1]      RETURN          0 1
constants (4) for 000000CECB2D6220:
  1        "print"
  2        "string"
  3        "char"
  4        64
locals (0) for 000000CECB2D6220:
upvalues (1) for 000000CECB2D6220:
  0        _ENV    0        0
```

When a function call is the last parameter to another function call, the former can pass multiple return values, while the latter can accept multiple parameters.

6.5 OP_TAILCALL instruction

6.5.1 Syntax

```
TAILCALL  A B C return R(A)(R(A+1), ... ,R(A+B-1))
```

6.5.2 Description

Performs a tail call, which happens when a return statement has a single function call as the expression, e.g. return foo(bar). A tail call results in the function being interpreted within the same call frame as the caller - the stack is replaced and then a 'goto' executed to start at the entry point in the VM. Only Lua functions can be tailcalled. Tailcalls allow infinite recursion without growing the stack.

Like `OP_CALL`, register R(A) holds the reference to the function object to be called. B encodes the number of parameters in the same manner as a `OP_CALL` instruction.

C isn't used by TAILCALL, since all return results are significant. In any case, Lua always generates a 0 for C, to denote multiple return results.

6.5.3 Examples

An `OP_TAILCALL` is used only for one specific return style, described above. Multiple return results are always produced by a tail call. Here is an example:

```
function y() return x('foo', 'bar') end
```

Generates:

```
function <stdin:1,1> (6 instructions at 000000C3C24DE4A0)
0 params, 3 slots, 1 upvalue, 0 locals, 3 constants, 0 functions
  1        [1]      GETTABUP         0 0 -1  ; _ENV "x"
  2        [1]      LOADK            1 -2    ; "foo"
  3        [1]      LOADK            2 -3    ; "bar"
  4        [1]      TAILCALL         0 3 0
  5        [1]      RETURN           0 0
  6        [1]      RETURN           0 1
constants (3) for 000000C3C24DE4A0:
  1        "x"
  2        "foo"
  3        "bar"
locals (0) for 000000C3C24DE4A0:
upvalues (1) for 000000C3C24DE4A0:
  0        _ENV     0        0
```

Arguments for a tail call are handled in exactly the same way as arguments for a normal call, so in line [4], the tail call has a field B value of 3, signifying 2 parameters. Field C is 0, for multiple returns; this due to the constant LUA_MULTRET in lua.h. In practice, field C is not used by the virtual machine (except as an assert) since the syntax guarantees multiple return results. Line [5] is a `OP_RETURN` instruction specifying multiple return results. This is required when the function called by `OP_TAILCALL` is a C function. In the case of a C function, execution continues to line [5] upon return, thus the RETURN is necessary. Line [6] is redundant. When Lua functions are tailcalled, the virtual machine does not return to line [5] at all.

6.6 OP_RETURN instruction

6.6.1 Syntax

```
RETURN  A B return R(A), ... ,R(A+B-2)
```

6.6.2 Description

Returns to the calling function, with optional return values.

First `OP_RETURN` closes any open upvalues by calling luaF_close().

If B is 1, there are no return values. If B is 2 or more, there are (B-1) return values, located in consecutive registers from R(A) onwards. If B is 0, the set of values range from R(A) to the top of the stack.

It is assumed that if the VM is returning to a Lua function then it is within the same invocation of the `luaV_execute()`. Else it is assumed that `luaV_execute()` is being invoked from a C function.

If B is 0 then the previous instruction (which must be either `OP_CALL` or `OP_VARARG`) would have set `L->top` to indicate how many values to return. The number of values to be returned in this case is R(A) to L->top.

If B > 0 then the number of values to be returned is simply B-1.

`OP_RETURN` calls luaD_poscall() which is responsible for copying return values to the caller - the first result is placed at the current `closure`'s address. `luaD_poscall()` leaves `L->top` just past the last result that was copied.

If `OP_RETURN` is returning to a Lua function and if the number of return values expected was indeterminate - i.e. `OP_CALL` had operand C = 0, then `L->top` is left where `luaD_poscall()` placed it - just beyond the top of the result list. This allows the `OP_CALL` instruction to figure out how many results were returned. If however `OP_CALL` had invoked with a value of C > 0 then the expected number of results is known, and in that case, `L->top` is reset to the calling function's `C->top`.

If `luaV_execute()` was called externally then `OP_RETURN` leaves `L->top` unchanged - so it will continue to be just past the top of the results list. This is because luaV_execute() does not have a way of informing callers how many values were returned; so the caller can determine the number of results by inspecting `L->top`.

6.6.3 Examples

Example of `OP_VARARG` followed by `OP_RETURN`:

```
function x(...) return ... end

1 [1]  VARARG           0 0
2 [1]  RETURN           0 0
```

Suppose we call `x(1,2,3)`; then, observe the setting of `L->top` when `OP_RETURN` executes:

```
(LOADK A=1 Bx=-2)       L->top = 4, ci->top = 4
(LOADK A=2 Bx=-3)       L->top = 4, ci->top = 4
(LOADK A=3 Bx=-4)       L->top = 4, ci->top = 4
(TAILCALL A=0 B=4 C=0)  L->top = 4, ci->top = 4
(VARARG A=0 B=0)        L->top = 2, ci->top = 2  ; we are in x()
(RETURN A=0 B=0)        L->top = 3, ci->top = 2
```

Observe that `OP_VARARG` set `L->top` to `base+3`.

But if we call `x(1)` instead:

```
(LOADK A=1 Bx=-2)       L->top = 4, ci->top = 4
(LOADK A=2 Bx=-3)       L->top = 4, ci->top = 4
(LOADK A=3 Bx=-4)       L->top = 4, ci->top = 4
(TAILCALL A=0 B=4 C=0)  L->top = 4, ci->top = 4
(VARARG A=0 B=0)        L->top = 2, ci->top = 2 ; we are in x()
(RETURN A=0 B=0)        L->top = 1, ci->top = 2
```

Notice that this time `OP_VARARG` set `L->top` to `base+1`.

6.7 OP_JMP instruction

6.7.1 Syntax

```
JMP A sBx   pc+=sBx; if (A) close all upvalues >= R(A - 1)
```

6.7.2 Description

Performs an unconditional jump, with sBx as a signed displacement. sBx is added to the program counter (PC), which points to the next instruction to be executed. If sBx is 0, the VM will proceed to the next instruction.

If R(A) is not 0 then all upvalues >= R(A-1) will be closed by calling luaF_close().

`OP_JMP` is used in loops, conditional statements, and in expressions when a boolean true/false need to be generated.

6.7.3 Examples

For example, since a relational test instruction makes conditional jumps rather than generate a boolean result, a JMP is used in the code sequence for loading either a true or a false:

```
function x() local m, n; return m >= n end
```

Generates:

```
function <stdin:1,1> (7 instructions at 00000034D2ABE340)
0 params, 3 slots, 0 upvalues, 2 locals, 0 constants, 0 functions
  1       [1]     LOADNIL         0 1
  2       [1]     LE              1 1 0   ; to 4 if false    (n <= m)
  3       [1]     JMP             0 1     ; to 5
  4       [1]     LOADBOOL        2 0 1
  5       [1]     LOADBOOL        2 1 0
  6       [1]     RETURN          2 2
  7       [1]     RETURN          0 1
constants (0) for 00000034D2ABE340:
locals (2) for 00000034D2ABE340:
  0       m       2       8
  1       n       2       8
upvalues (0) for 00000034D2ABE340:
```

Line[2] performs the relational test. In line [3], the JMP skips over the false path (line [4]) to the true path (line [5]). The result is placed into temporary local 2, and returned to the caller by RETURN in line [6].

6.8 OP_VARARG instruction

6.8.1 Syntax

```
VARARG  A B R(A), R(A+1), ..., R(A+B-1) = vararg
```

6.8.2 Description

`VARARG` implements the vararg operator `...` in expressions. `VARARG` copies B-1 parameters into a number of registers starting from R(A), padding with nils if there aren't enough values. If B is 0, `VARARG` copies as many values as it can based on the number of parameters passed. If a fixed number of values is required, B is a value greater than 1. If any number of values is required, B is 0.

6.8.3 Examples

The use of VARARG will become clear with the help of a few examples:

```
local a,b,c = ...
```

Generates:

```
main <(string):0,0> (2 instructions at 00000029D9FA8310)
0+ params, 3 slots, 1 upvalue, 3 locals, 0 constants, 0 functions
      1       [1]     VARARG          0 4
      2       [1]     RETURN          0 1
constants (0) for 00000029D9FA8310:
locals (3) for 00000029D9FA8310:
      0       a       2       3
      1       b       2       3
      2       c       2       3
upvalues (1) for 00000029D9FA8310:
      0       _ENV    1       0
```

Note that the main or top-level chunk is a vararg function. In this example, the left hand side of the assignment statement needs three values (or objects.) So in instruction [1], the operand B of the VARARG instruction is (3+1), or 4. VARARG will copy three values into a, b and c. If there are less than three values available, nils will be used to fill up the empty places.

```
local a = function(...) local a,b,c = ... end
```

This gives:

```
main <(string):0,0> (2 instructions at 00000029D9FA72D0)
0+ params, 2 slots, 1 upvalue, 1 local, 0 constants, 1 function
      1       [1]     CLOSURE         0 0     ; 00000029D9FA86D0
      2       [1]     RETURN          0 1
constants (0) for 00000029D9FA72D0:
locals (1) for 00000029D9FA72D0:
      0       a       2       3
upvalues (1) for 00000029D9FA72D0:
      0       _ENV    1       0

function <(string):1,1> (2 instructions at 00000029D9FA86D0)
0+ params, 3 slots, 0 upvalues, 3 locals, 0 constants, 0 functions
      1       [1]     VARARG          0 4
      2       [1]     RETURN          0 1
constants (0) for 00000029D9FA86D0:
locals (3) for 00000029D9FA86D0:
      0       a       2       3
      1       b       2       3
      2       c       2       3
upvalues (0) for 00000029D9FA86D0:
```

Here is an alternate version where a function is instantiated and assigned to local a. The old-style arg is retained for compatibility purposes, but is unused in the above example.

```
local a; a(...)
```

Leads to:

```
main <(string):0,0> (5 instructions at 00000029D9FA6D30)
0+ params, 3 slots, 1 upvalue, 1 local, 0 constants, 0 functions
      1       [1]     LOADNIL         0 0
      2       [1]     MOVE            1 0
      3       [1]     VARARG          2 0
      4       [1]     CALL            1 0 1
      5       [1]     RETURN          0 1
constants (0) for 00000029D9FA6D30:
```

(continues on next page)

(continued from previous page)

```
locals (1) for 00000029D9FA6D30:
       0        a        2        6
upvalues (1) for 00000029D9FA6D30:
       0        _ENV     1        0
```

When a function is called with `...` as the argument, the function will accept a variable number of parameters or arguments. On instruction [3], a `VARARG` with a B field of 0 is used. The `VARARG` will copy all the parameters passed on to the main chunk to register 2 onwards, so that the `CALL` in the next line can utilize them as parameters of function `a`. The function call is set to accept a multiple number of parameters and returns zero results.

```
local a = {...}
```

Produces:

```
main <(string):0,0> (4 instructions at 00000029D9FA8130)
0+ params, 2 slots, 1 upvalue, 1 local, 0 constants, 0 functions
       1        [1]      NEWTABLE         0 0 0
       2        [1]      VARARG           1 0
       3        [1]      SETLIST          0 0 1    ; 1
       4        [1]      RETURN           0 1
constants (0) for 00000029D9FA8130:
locals (1) for 00000029D9FA8130:
       0        a        4        5
upvalues (1) for 00000029D9FA8130:
       0        _ENV     1        0
```

And:

```
return ...
```

Produces:

```
main <(string):0,0> (3 instructions at 00000029D9FA8270)
0+ params, 2 slots, 1 upvalue, 0 locals, 0 constants, 0 functions
       1        [1]      VARARG           0 0
       2        [1]      RETURN           0 0
       3        [1]      RETURN           0 1
constants (0) for 00000029D9FA8270:
locals (0) for 00000029D9FA8270:
upvalues (1) for 00000029D9FA8270:
       0        _ENV     1        0
```

Above are two other cases where `VARARG` needs to copy all passed parameters over to a set of registers in order for the next operation to proceed. Both the above forms of table creation and return accepts a variable number of values or objects.

6.9 OP_LOADBOOL instruction

6.9.1 Syntax

```
LOADBOOL A B C    R(A) := (Bool)B; if (C) pc++
```

6.9.2 Description

Loads a boolean value (true or false) into register R(A). true is usually encoded as an integer 1, false is always 0. If C is non-zero, then the next instruction is skipped (this is used when you have an assignment statement where the expression uses relational operators, e.g. M = K>5.) You can use any non-zero value for the boolean true in field B, but since you cannot use booleans as numbers in Lua, it's best to stick to 1 for true.

`LOADBOOL` is used for loading a boolean value into a register. It's also used where a boolean result is supposed to be generated, because relational test instructions, for example, do not generate boolean results – they perform conditional jumps instead. The operand C is used to optionally skip the next instruction (by incrementing PC by 1) in order to support such code. For simple assignments of boolean values, C is always 0.

6.9.3 Examples

The following line of code:

```
f=load('local a,b = true,false')
```

generates:

```
main <(string):0,0> (3 instructions at 0000020F274C2610)
0+ params, 2 slots, 1 upvalue, 2 locals, 0 constants, 0 functions
        1       [1]     LOADBOOL        0 1 0
        2       [1]     LOADBOOL        1 0 0
        3       [1]     RETURN          0 1
constants (0) for 0000020F274C2610:
locals (2) for 0000020F274C2610:
        0       a       3       4
        1       b       3       4
upvalues (1) for 0000020F274C2610:
        0       _ENV    1       0
```

This example is straightforward: Line [1] assigns true to local a (register 0) while line [2] assigns false to local b (register 1). In both cases, field C is 0, so PC is not incremented and the next instruction is not skipped.

Next, look at this line:

```
f=load('local a = 5 > 2')
```

This leads to following bytecode:

```
main <(string):0,0> (5 instructions at 0000020F274BAE00)
0+ params, 2 slots, 1 upvalue, 1 local, 2 constants, 0 functions
        1       [1]     LT              1 -2 -1 ; 2 5
        2       [1]     JMP             0 1     ; to 4
        3       [1]     LOADBOOL        0 0 1
        4       [1]     LOADBOOL        0 1 0
        5       [1]     RETURN          0 1
constants (2) for 0000020F274BAE00:
        1       5
        2       2
locals (1) for 0000020F274BAE00:
        0       a       5       6
upvalues (1) for 0000020F274BAE00:
        0       _ENV    1       0
```

This is an example of an expression that gives a boolean result and is assigned to a variable. Notice that Lua does not optimize the expression into a true value; Lua does not perform compile-time constant evaluation for relational operations, but it can perform simple constant evaluation for arithmetic operations.

Since the relational operator `LT` does not give a boolean result but performs a conditional jump, `LOADBOOL` uses its C operand to perform an unconditional jump in line [3] – this saves one instruction and makes things a little tidier. The reason for all this is that the instruction set is simply optimized for if…then blocks. Essentially, `local a = 5 > 2` is executed in the following way:

```
local a
if 2 < 5 then
  a = true
else
  a = false
end
```

In the disassembly listing, when `LT` tests 2 < 5, it evaluates to true and doesn't perform a conditional jump. Line [2] jumps over the false result path, and in line [4], the local a (register 0) is assigned the boolean true by the instruction `LOADBOOL`. If 2 and 5 were reversed, line [3] will be followed instead, setting a false, and then the true result path (line [4]) will be skipped, since `LOADBOOL` has its field C set to non-zero.

So the true result path goes like this (additional comments in parentheses):

```
1        [1]      LT               1 -2 -1 ; 2 5        (if 2 < 5)
2        [1]      JMP              0 1     ; to 4
4        [1]      LOADBOOL         0 1 0   ;            (a = true)
5        [1]      RETURN           0 1
```

and the false result path (which never executes in this example) goes like this:

```
1        [1]      LT               1 -2 -1 ; 2 5        (if 2 < 5)
3        [1]      LOADBOOL         0 0 1                (a = false)
5        [1]      RETURN           0 1
```

The true result path looks longer, but it isn't, due to the way the virtual machine is implemented. This will be discussed further in the section on relational and logic instructions.

6.10 OP_EQ, OP_LT and OP_LE Instructions

Relational and logic instructions are used in conjunction with other instructions to implement control structures or expressions. Instead of generating boolean results, these instructions conditionally perform a jump over the next instruction; the emphasis is on implementing control blocks. Instructions are arranged so that there are two paths to follow based on the relational test.

```
EQ  A B C if ((RK(B) == RK(C)) ~= A) then PC++
LT  A B C if ((RK(B) <  RK(C)) ~= A) then PC++
LE  A B C if ((RK(B) <= RK(C)) ~= A) then PC++
```

6.10.1 Description

Compares RK(B) and RK(C), which may be registers or constants. If the boolean result is not A, then skip the next instruction. Conversely, if the boolean result equals A, continue with the next instruction.

EQ is for equality. LT is for "less than" comparison. LE is for "less than or equal to" comparison. The boolean A field allows the full set of relational comparison operations to be synthesized from these three instructions. The Lua code generator produces either 0 or 1 for the boolean A.

For the fall-through case, a *OP_JMP instruction* is always expected, in order to optimize execution in the virtual machine. In effect, EQ, LT and LE must always be paired with a following JMP instruction.

6.10.2 Examples

By comparing the result of the relational operation with A, the sense of the comparison can be reversed. Obviously the alternative is to reverse the paths taken by the instruction, but that will probably complicate code generation some more. The conditional jump is performed if the comparison result is not A, whereas execution continues normally if the comparison result matches A. Due to the way code is generated and the way the virtual machine works, a JMP instruction is always expected to follow an EQ, LT or LE. The following JMP is optimized by executing it in conjunction with EQ, LT or LE.

```
local x,y; return x ~= y
```

Generates:

```
main <(string):0,0> (7 instructions at 0000001BC48FD390)
0+ params, 3 slots, 1 upvalue, 2 locals, 0 constants, 0 functions
        1       [1]     LOADNIL         0 1
        2       [1]     EQ              0 0 1
        3       [1]     JMP             0 1     ; to 5
        4       [1]     LOADBOOL        2 0 1
        5       [1]     LOADBOOL        2 1 0
        6       [1]     RETURN          2 2
        7       [1]     RETURN          0 1
constants (0) for 0000001BC48FD390:
locals (2) for 0000001BC48FD390:
        0       x       2       8
        1       y       2       8
upvalues (1) for 0000001BC48FD390:
        0       _ENV    1       0
```

In the above example, the equality test is performed in instruction [2]. However, since the comparison need to be returned as a result, LOADBOOL instructions are used to set a register with the correct boolean value. This is the usual code pattern generated if the expression requires a boolean value to be generated and stored in a register as an intermediate value or a final result.

It is easier to visualize the disassembled code as:

```
if x ~= y then
  return true
else
  return false
end
```

The true result path (when the comparison result matches A) goes like this:

```
1   [1] LOADNIL     0   1
2   [1] EQ          0   0   1    ; to 4 if true     (x ~= y)
3   [1] JMP         1            ; to 5
5   [1] LOADBOOL    2   1   0    ; true             (true path)
6   [1] RETURN      2   2
```

While the false result path (when the comparison result does not match A) goes like this:

```
1   [1] LOADNIL     0   1
2   [1] EQ          0   0   1    ; to 4 if true    (x ~= y)
4   [1] LOADBOOL    2   0   1    ; false, to 6     (false path)
6   [1] RETURN      2   2
```

Comments following the EQ in line [2] lets the user know when the conditional jump is taken. The jump is taken when "the value in register 0 equals to the value in register 1" (the comparison) is not false (the value of operand A). If the comparison is x == y, everything will be the same except that the A operand in the EQ instruction will be 1, thus reversing the sense of the comparison. Anyway, these are just the Lua code generator's conventions; there are other ways to code x ~= y in terms of Lua virtual machine instructions.

For conditional statements, there is no need to set boolean results. Lua is optimized for coding the more common conditional statements rather than conditional expressions.

```
local x,y; if x ~= y then return "foo" else return "bar" end
```

Results in:

```
main <(string):0,0> (9 instructions at 0000001BC4914D50)
0+ params, 3 slots, 1 upvalue, 2 locals, 2 constants, 0 functions
      1        [1]     LOADNIL         0 1
      2        [1]     EQ              1 0 1    ; to 4 if false    (x ~= y)
      3        [1]     JMP             0 3      ; to 7
      4        [1]     LOADK           2 -1     ; "foo"            (true block)
      5        [1]     RETURN          2 2
      6        [1]     JMP             0 2      ; to 9
      7        [1]     LOADK           2 -2     ; "bar"            (false block)
      8        [1]     RETURN          2 2
      9        [1]     RETURN          0 1
constants (2) for 0000001BC4914D50:
      1        "foo"
      2        "bar"
locals (2) for 0000001BC4914D50:
      0        x        2        10
      1        y        2        10
upvalues (1) for 0000001BC4914D50:
      0        _ENV     1        0
```

In the above conditional statement, the same inequality operator is used in the source, but the sense of the EQ instruction in line [2] is now reversed. Since the EQ conditional jump can only skip the next instruction, additional JMP instructions are needed to allow large blocks of code to be placed in both true and false paths. In contrast, in the previous example, only a single instruction is needed to set a boolean value. For if statements, the true block comes first followed by the false block in code generated by the code generator. To reverse the positions of the true and false paths, the value of operand A is changed.

The true path (when x ~= y is true) goes from [2] to [4]–[6] and on to [9]. Since there is a RETURN in line [5], the JMP in line [6] and the RETURN in [9] are never executed at all; they are redundant but does not adversely affect performance in any way. The false path is from [2] to [3] to [7]–[9] onwards. So in a disassembly listing, you should see the true and false code blocks in the same order as in the Lua source.

The following is another example, this time with an elseif:

```
if 8 > 9 then return 8 elseif 5 >= 4 then return 5 else return 9 end
```

Generates:

```
main <(string):0,0> (13 instructions at 0000001BC4913770)
0+ params, 2 slots, 1 upvalue, 0 locals, 4 constants, 0 functions
      1       [1]     LT              0 -2 -1 ; 9 8
      2       [1]     JMP             0 3     ; to 6
      3       [1]     LOADK           0 -1    ; 8
      4       [1]     RETURN          0 2
      5       [1]     JMP             0 7     ; to 13
      6       [1]     LE              0 -4 -3 ; 4 5
      7       [1]     JMP             0 3     ; to 11
      8       [1]     LOADK           0 -3    ; 5
      9       [1]     RETURN          0 2
      10      [1]     JMP             0 2     ; to 13
      11      [1]     LOADK           0 -2    ; 9
      12      [1]     RETURN          0 2
      13      [1]     RETURN          0 1
constants (4) for 0000001BC4913770:
      1       8
      2       9
      3       5
      4       4
locals (0) for 0000001BC4913770:
upvalues (1) for 0000001BC4913770:
      0       _ENV    1       0
```

This example is a little more complex, but the blocks are structured in the same order as the Lua source, so interpreting the disassembled code should not be too hard.

6.11 OP_TEST and OP_TESTSET instructions

6.11.1 Syntax

```
TEST        A C     if (boolean(R(A)) != C) then PC++
TESTSET     A B C   if (boolean(R(B)) != C) then PC++ else R(A) := R(B)

where boolean(x) => ((x == nil || x == false) ? 0 : 1)
```

6.11.2 Description

These two instructions used for performing boolean tests and implementing Lua's logical operators.

Used to implement `and` and `or` logical operators, or for testing a single register in a conditional statement.

For `TESTSET`, register `R(B)` is coerced into a boolean (i.e. `false` and `nil` evaluate to `0` and any other value to `1`) and compared to the boolean field `C` (`0` or `1`). If `boolean(R(B))` does not match `C`, the next instruction is skipped, otherwise `R(B)` is assigned to `R(A)` and the VM continues with the next instruction. The `and` operator uses a `C` of `0` (false) while `or` uses a C value of `1` (true).

`TEST` is a more primitive version of `TESTSET`. `TEST` is used when the assignment operation is not needed, otherwise it is the same as `TESTSET` except that the operand slots are different.

For the fall-through case, a `JMP` is always expected, in order to optimize execution in the virtual machine. In effect, `TEST` and `TESTSET` must always be paired with a following `JMP` instruction.

6.11.3 Examples

TEST and TESTSET are used in conjunction with a following JMP instruction, while TESTSET has an addditional conditional assignment. Like EQ, LT and LE, the following JMP instruction is compulsory, as the virtual machine will execute the JMP together with TEST or TESTSET. The two instructions are used to implement short-circuit LISP-style logical operators that retains and propagates operand values instead of booleans. First, we'll look at how and and or behaves:

```
f=load('local a,b,c; c = a and b')
```

Generates:

```
main <(string):0,0> (5 instructions at 0000020F274CF1A0)
0+ params, 3 slots, 1 upvalue, 3 locals, 0 constants, 0 functions
        1       [1]     LOADNIL         0 2
        2       [1]     TESTSET         2 0 0   ; to 4 if true
        3       [1]     JMP             0 1     ; to 5
        4       [1]     MOVE            2 1
        5       [1]     RETURN          0 1
constants (0) for 0000020F274CF1A0:
locals (3) for 0000020F274CF1A0:
        0       a       2       6
        1       b       2       6
        2       c       2       6
upvalues (1) for 0000020F274CF1A0:
        0       _ENV    1       0
```

An and sequence exits on false operands (which can be false or nil) because any false operands in a string of and operations will make the whole boolean expression false. If operands evaluates to true, evaluation continues. When a string of and operations evaluates to true, the result is the last operand value.

In line [2], C is 0. Since B is 0, therefore R(B) refers to the local a. Since R(B) is nil then boolean(R(B)) evaluates to 0. Thus C matches boolean(R(B)). Therefore the value of a is assigned to c and the next instruction which is a JMP is executed. This is equivalent to:

```
if a then
  c = b       -- executed by MOVE on line [4]
else
  c = a       -- executed by TESTSET on line [2]
end
```

The c = a portion is done by TESTSET itself, while MOVE performs c = b. Now, if the result is already set with one of the possible values, a TEST instruction is used instead:

```
f=load('local a,b; a = a and b')
```

Generates:

```
main <(string):0,0> (5 instructions at 0000020F274D0A70)
0+ params, 2 slots, 1 upvalue, 2 locals, 0 constants, 0 functions
        1       [1]     LOADNIL         0 1
        2       [1]     TEST            0 0     ; to 4 if true
        3       [1]     JMP             0 1     ; to 5
        4       [1]     MOVE            0 1
        5       [1]     RETURN          0 1
constants (0) for 0000020F274D0A70:
locals (2) for 0000020F274D0A70:
```

(continues on next page)

(continued from previous page)

```
      0       a       2       6
      1       b       2       6
upvalues (1) for 0000020F274D0A70:
      0       _ENV    1       0
```

Here C is 0, and boolean(R(A)) is 0 too, so that the TEST instruction on line [2] does not skip the next instruction which is a JMP. The TEST instruction does not perform an assignment operation, since a = a is redundant. This makes TEST a little faster. This is equivalent to:

```
if a then
  a = b
end
```

Next, we will look at the or operator:

```
f=load('local a,b,c; c = a or b')
```

Generates:

```
main <(string):0,0> (5 instructions at 0000020F274D1AB0)
0+ params, 3 slots, 1 upvalue, 3 locals, 0 constants, 0 functions
      1       [1]     LOADNIL         0 2
      2       [1]     TESTSET         2 0 1   ; to 4 if false
      3       [1]     JMP             0 1     ; to 5
      4       [1]     MOVE            2 1
      5       [1]     RETURN          0 1
constants (0) for 0000020F274D1AB0:
locals (3) for 0000020F274D1AB0:
      0       a       2       6
      1       b       2       6
      2       c       2       6
upvalues (1) for 0000020F274D1AB0:
      0       _ENV    1       0
```

An or sequence exits on true operands, because any operands evaluating to true in a string of or operations will make the whole boolean expression true. If operands evaluates to false, evaluation continues. When a string of or operations evaluates to false, all operands must have evaluated to false.

In line [2], C is 1. Since B is 0, therefore R(B) refers to the local a. Since R(B) is nil then boolean(R(B)) evaluates to 0. Thus C does not match boolean(R(B)). Therefore, the next instruction which is a JMP is skipped and execution continues on line [4]. This is equivalent to:

```
if a then
  c = a      -- executed by TESTSET on line [2]
else
  c = b      -- executed by MOVE on line [4]
end
```

Like the case of and, TEST is used when the result already has one of the possible values, saving an assignment operation:

```
f=load('local a,b; a = a or b')
```

Generates:

```
main <(string):0,0> (5 instructions at 0000020F274D1010)
0+ params, 2 slots, 1 upvalue, 2 locals, 0 constants, 0 functions
      1       [1]     LOADNIL         0 1
      2       [1]     TEST            0 1     ; to 4 if false
      3       [1]     JMP             0 1     ; to 5
      4       [1]     MOVE            0 1
      5       [1]     RETURN          0 1
constants (0) for 0000020F274D1010:
locals (2) for 0000020F274D1010:
      0       a       2       6
      1       b       2       6
upvalues (1) for 0000020F274D1010:
      0       _ENV    1       0
```

Short-circuit logical operators also means that the following Lua code does not require the use of a boolean operation:

```
f=load('local a,b,c; if a > b and a > c then return a end')
```

Leads to:

```
main <(string):0,0> (7 instructions at 0000020F274D1150)
0+ params, 3 slots, 1 upvalue, 3 locals, 0 constants, 0 functions
      1       [1]     LOADNIL         0 2
      2       [1]     LT              0 1 0   ; to 4 if true
      3       [1]     JMP             0 3     ; to 7
      4       [1]     LT              0 2 0   ; to 6 if true
      5       [1]     JMP             0 1     ; to 7
      6       [1]     RETURN          0 2
      7       [1]     RETURN          0 1
constants (0) for 0000020F274D1150:
locals (3) for 0000020F274D1150:
      0       a       2       8
      1       b       2       8
      2       c       2       8
upvalues (1) for 0000020F274D1150:
      0       _ENV    1       0
```

With short-circuit evaluation, `a > c` is never executed if `a > b` is false, so the logic of the Lua statement can be readily implemented using the normal conditional structure. If both `a > b` and `a > c` are true, the path followed is [2] (the `a > b` test) to [4] (the `a > c` test) and finally to [6], returning the value of `a`. A `TEST` instruction is not required. This is equivalent to:

```
if a > b then
  if a > c then
    return a
  end
end
```

For a single variable used in the expression part of a conditional statement, `TEST` is used to boolean-test the variable:

```
f=load('if Done then return end')
```

Generates:

```
main <(string):0,0> (5 instructions at 0000020F274D13D0)
0+ params, 2 slots, 1 upvalue, 0 locals, 1 constant, 0 functions
      1       [1]     GETTABUP        0 0 -1  ; _ENV "Done"
```

(continues on next page)

(continued from previous page)

```
        2       [1]     TEST            0 0     ; to 4 if true
        3       [1]     JMP             0 1     ; to 5
        4       [1]     RETURN          0 1
        5       [1]     RETURN          0 1
constants (1) for 0000020F274D13D0:
        1       "Done"
locals (0) for 0000020F274D13D0:
upvalues (1) for 0000020F274D13D0:
        0       _ENV    1       0
```

In line [2], the `TEST` instruction jumps to the `true` block if the value in temporary register 0 (from the global `Done`) is `true`. The `JMP` at line [3] jumps over the `true` block, which is the code inside the if block (line [4]).

If the test expression of a conditional statement consist of purely boolean operators, then a number of TEST instructions will be used in the usual short-circuit evaluation style:

```
f=load('if Found and Match then return end')
```

Generates:

```
main <(string):0,0> (8 instructions at 0000020F274D1C90)
0+ params, 2 slots, 1 upvalue, 0 locals, 2 constants, 0 functions
        1       [1]     GETTABUP        0 0 -1  ; _ENV "Found"
        2       [1]     TEST            0 0     ; to 4 if true
        3       [1]     JMP             0 4     ; to 8
        4       [1]     GETTABUP        0 0 -2  ; _ENV "Match"
        5       [1]     TEST            0 0     ; to 7 if true
        6       [1]     JMP             0 1     ; to 8
        7       [1]     RETURN          0 1
        8       [1]     RETURN          0 1
constants (2) for 0000020F274D1C90:
        1       "Found"
        2       "Match"
locals (0) for 0000020F274D1C90:
upvalues (1) for 0000020F274D1C90:
        0       _ENV    1       0
```

In the last example, the true block of the conditional statement is executed only if both `Found` and `Match` evaluate to `true`. The path is from [2] (test for `Found`) to [4] to [5] (test for `Match`) to [7] (the true block, which is an explicit `return` statement.)

If the statement has an `else` section, then the `JMP` on line [6] will jump to the false block (the `else` block) while an additional `JMP` will be added to the true block to jump over this new block of code. If `or` is used instead of `and`, the appropriate C operand will be adjusted accordingly.

Finally, here is how Lua's ternary operator (:? in C) equivalent works:

```
f=load('local a,b,c; a = a and b or c')
```

Generates:

```
main <(string):0,0> (7 instructions at 0000020F274D1A10)
0+ params, 3 slots, 1 upvalue, 3 locals, 0 constants, 0 functions
        1       [1]     LOADNIL         0 2
        2       [1]     TEST            0 0     ; to 4 if true
        3       [1]     JMP             0 2     ; to 6
        4       [1]     TESTSET         0 1 1   ; to 6 if false
```

(continues on next page)

(continued from previous page)

```
      5       [1]     JMP             0 1     ; to 7
      6       [1]     MOVE            0 2
      7       [1]     RETURN          0 1
constants (0) for 0000020F274D1A10:
locals (3) for 0000020F274D1A10:
      0       a       2       8
      1       b       2       8
      2       c       2       8
upvalues (1) for 0000020F274D1A10:
      0       _ENV    1       0
```

The `TEST` in line [2] is for the `and` operator. First, local `a` is tested in line [2]. If it is false, then execution continues in [3], jumping to line [6]. Line [6] assigns local `c` to the end result because since if `a` is false, then `a and b` is `false`, and `false or c` is `c`.

If local `a` is `true` in line [2], the `TEST` instruction makes a jump to line [4], where there is a `TESTSET`, for the `or` operator. If `b` evaluates to `true`, then the end result is assigned the value of `b`, because `b or c` is `b` if `b` is `not false`. If `b` is also `false`, the end result will be `c`.

For the instructions in line [2], [4] and [6], the target (in field A) is register 0, or the local `a`, which is the location where the result of the boolean expression is assigned. The equivalent Lua code is:

```
if a then
  if b then
    a = b
  else
    a = c
  end
else
  a = c
end
```

The two `a = c` assignments are actually the same piece of code, but are repeated here to avoid using a `goto` and a label. Normally, if we assume `b` is `not false` and `not nil`, we end up with the more recognizable form:

```
if a then
  a = b     -- assuming b ~= false
else
  a = c
end
```

6.12 OP_FORPREP and OP_FORLOOP instructions

6.12.1 Syntax

```
FORPREP     A sBx   R(A)-=R(A+2); pc+=sBx
FORLOOP     A sBx   R(A)+=R(A+2);
                    if R(A) <?= R(A+1) then { pc+=sBx; R(A+3)=R(A) }
```

6.12.2 Description

Lua has dedicated instructions to implement the two types of `for` loops, while the other two types of loops uses traditional test-and-jump.

`FORPREP` initializes a numeric for loop, while `FORLOOP` performs an iteration of a numeric for loop.

A numeric for loop requires 4 registers on the stack, and each register must be a number. R(A) holds the initial value and doubles as the internal loop variable (the internal index); R(A+1) is the limit; R(A+2) is the stepping value; R(A+3) is the actual loop variable (the external index) that is local to the for block.

`FORPREP` sets up a for loop. Since `FORLOOP` is used for initial testing of the loop condition as well as conditional testing during the loop itself, `FORPREP` performs a negative step and jumps unconditionally to `FORLOOP` so that `FORLOOP` is able to correctly make the initial loop test. After this initial test, `FORLOOP` performs a loop step as usual, restoring the initial value of the loop index so that the first iteration can start.

In `FORLOOP`, a jump is made back to the start of the loop body if the limit has not been reached or exceeded. The sense of the comparison depends on whether the stepping is negative or positive, hence the "<?=" operator. Jumps for both instructions are encoded as signed displacements in the `sBx` field. An empty loop has a `FORLOOP sBx` value of -1.

`FORLOOP` also sets R(A+3), the external loop index that is local to the loop block. This is significant if the loop index is used as an upvalue (see below.) R(A), R(A+1) and R(A+2) are not visible to the programmer.

The loop variable ends with the last value before the limit is reached (unlike C) because it is not updated unless the jump is made. However, since loop variables are local to the loop itself, you should not be able to use it unless you cook up an implementation-specific hack.

6.12.3 Examples

For the sake of efficiency, `FORLOOP` contains a lot of functionality, so when a loop iterates, only one instruction, `FORLOOP`, is needed. Here is a simple example:

```
f=load('local a = 0; for i = 1,100,5 do a = a + i end')
```

Generates:

```
main <(string):0,0> (8 instructions at 000001E9F0DF52F0)
0+ params, 5 slots, 1 upvalue, 5 locals, 4 constants, 0 functions
        1       [1]     LOADK           0 -1    ; 0
        2       [1]     LOADK           1 -2    ; 1
        3       [1]     LOADK           2 -3    ; 100
        4       [1]     LOADK           3 -4    ; 5
        5       [1]     FORPREP         1 1     ; to 7
        6       [1]     ADD             0 0 4
        7       [1]     FORLOOP         1 -2    ; to 6
        8       [1]     RETURN          0 1
constants (4) for 000001E9F0DF52F0:
        1       0
        2       1
        3       100
        4       5
locals (5) for 000001E9F0DF52F0:
        0       a       2       9
        1       (for index)     5       8
        2       (for limit)     5       8
        3       (for step)      5       8
```

(continues on next page)

(continued from previous page)

```
      4       i       6       7
upvalues (1) for 000001E9F0DF52F0:
      0       _ENV    1       0
```

In the above example, notice that the `for` loop causes three additional local pseudo-variables (or internal variables) to be defined, apart from the external loop index, `i`. The three pseudovariables, named `(for index)`, `(for limit)` and `(for step)` are required to completely specify the state of the loop, and are not visible to Lua source code. They are arranged in consecutive registers, with the external loop index given by R(A+3) or register 4 in the example.

The loop body is in line [6] while line [7] is the `FORLOOP` instruction that steps through the loop state. The `sBx` field of `FORLOOP` is negative, as it always jumps back to the beginning of the loop body.

Lines [2]–[4] initialize the three register locations where the loop state will be stored. If the loop step is not specified in the Lua source, a constant 1 is added to the constant pool and a `LOADK` instruction is used to initialize the pseudo-variable `(for step)` with the loop step.

`FORPREP` in lines [5] makes a negative loop step and jumps to line [7] for the initial test. In the example, at line [5], the internal loop index (at register 1) will be (1-5) or -4. When the virtual machine arrives at the `FORLOOP` in line [7] for the first time, one loop step is made prior to the first test, so the initial value that is actually tested against the limit is (-4+5) or 1. Since 1 < 100, an iteration will be performed. The external loop index `i` is then set to 1 and a jump is made to line [6], thus starting the first iteration of the loop.

The loop at line [6]–[7] repeats until the internal loop index exceeds the loop limit of 100. The conditional jump is not taken when that occurs and the loop ends. Beyond the scope of the loop body, the loop state (`(for index)`, `(for limit)`, `(for step)` and `i`) is not valid. This is determined by the parser and code generator. The range of PC values for which the loop state variables are valid is located in the locals list.

Here is another example:

```
f=load('for i = 10,1,-1 do if i == 5 then break end end')
```

This leads to:

```
main <(string):0,0> (8 instructions at 000001E9F0DEC110)
0+ params, 4 slots, 1 upvalue, 4 locals, 4 constants, 0 functions
      1       [1]     LOADK           0 -1    ; 10
      2       [1]     LOADK           1 -2    ; 1
      3       [1]     LOADK           2 -3    ; -1
      4       [1]     FORPREP         0 2     ; to 7
      5       [1]     EQ              1 3 -4  ; - 5
      6       [1]     JMP             0 1     ; to 8
      7       [1]     FORLOOP         0 -3    ; to 5
      8       [1]     RETURN          0 1
constants (4) for 000001E9F0DEC110:
      1       10
      2       1
      3       -1
      4       5
locals (4) for 000001E9F0DEC110:
      0       (for index)     4       8
      1       (for limit)     4       8
      2       (for step)      4       8
      3       i       5       7
upvalues (1) for 000001E9F0DEC110:
      0       _ENV    1       0
```

In the second loop example above, except for a negative loop step size, the structure of the loop is identical. The body of the loop is from line [5] to line [7]. Since no additional stacks or states are used, a break translates simply to a `JMP` instruction (line [6]). There is nothing to clean up after a FORLOOP ends or after a JMP to exit a loop.

6.13 OP_TFORCALL and OP_TFORLOOP instructions

6.13.1 Syntax

```
TFORCALL     A C         R(A+3), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2))
TFORLOOP     A sBx       if R(A+1) ~= nil then { R(A)=R(A+1); pc += sBx }
```

6.13.2 Description

Apart from a numeric for loop (implemented by FORPREP and FORLOOP), Lua has a generic for loop, implemented by TFORCALL and TFORLOOP.

The generic for loop keeps 3 items in consecutive register locations to keep track of things. R(A) is the iterator function, which is called once per loop. R(A+1) is the state, and R(A+2) is the control variable. At the start, R(A+2) has an initial value. R(A), R(A+1) and R(A+2) are internal to the loop and cannot be accessed by the programmer.

In addition to these internal loop variables, the programmer specifies one or more loop variables that are external and visible to the programmer. These loop variables reside at locations R(A+3) onwards, and their count is specified in operand C. Operand C must be at least 1. They are also local to the loop body, like the external loop index in a numerical for loop.

Each time TFORCALL executes, the iterator function referenced by R(A) is called with two arguments: the state and the control variable (R(A+1) and R(A+2)). The results are returned in the local loop variables, from R(A+3) onwards, up to R(A+2+C).

Next, the TFORLOOP instruction tests the first return value, R(A+3). If it is nil, the iterator loop is at an end, and the for loop block ends by simply moving to the next instruction.

If R(A+3) is not nil, there is another iteration, and R(A+3) is assigned as the new value of the control variable, R(A+2). Then the TFORLOOP instruction sends execution back to the beginning of the loop (the sBx operand specifies how many instructions to move to get to the start of the loop body).

6.13.3 Examples

This example has a loop with one additional result (v) in addition to the loop enumerator (i):

```
f=load('for i,v in pairs(t) do print(i,v) end')
```

This produces:

```
main <(string):0,0> (11 instructions at 0000014DB7FD2610)
0+ params, 8 slots, 1 upvalue, 5 locals, 3 constants, 0 functions
        1       [1]     GETTABUP        0 0 -1  ; _ENV "pairs"
        2       [1]     GETTABUP        1 0 -2  ; _ENV "t"
        3       [1]     CALL            0 2 4
        4       [1]     JMP             0 4     ; to 9
        5       [1]     GETTABUP        5 0 -3  ; _ENV "print"
        6       [1]     MOVE            6 3
        7       [1]     MOVE            7 4
        8       [1]     CALL            5 3 1
        9       [1]     TFORCALL        0 2
```

(continues on next page)

(continued from previous page)

```
      10       [1]      TFORLOOP        2 -6    ; to 5
      11       [1]      RETURN          0 1
constants (3) for 0000014DB7FD2610:
      1        "pairs"
      2        "t"
      3        "print"
locals (5) for 0000014DB7FD2610:
      0        (for generator) 4        11
      1        (for state)     4        11
      2        (for control)   4        11
      3        i        5        9
      4        v        5        9
upvalues (1) for 0000014DB7FD2610:
      0        _ENV     1        0
```

The iterator function is located in register 0, and is named `(for generator)` for debugging purposes. The state is in register 1, and has the name `(for state)`. The control variable, `(for control)`, is contained in register 2. These correspond to locals R(A), R(A+1) and R(A+2) in the `TFORCALL` description. Results from the iterator function call is placed into register 3 and 4, which are locals `i` and `v`, respectively. On line [9], the operand C of `TFORCALL` is 2, corresponding to two iterator variables (`i` and `v`).

Lines [1]–[3] prepares the iterator state. Note that the call to the `pairs()` standard library function has 1 parameter and 3 results. After the call in line [3], register 0 is the iterator function (which by default is the Lua function `next()` unless `__pairs` meta method has been overriden), register 1 is the loop state, register 2 is the initial value of the control variable (which is `nil` in the default case). The iterator variables `i` and `v` are both invalid at the moment, because we have not entered the loop yet.

Line [4] is a `JMP` to `TFORCALL` on line [9]. The `TFORCALL` instruction calls the iterator function, generating the first set of enumeration results in locals `i` and `v`.

The `TFORLOOP` insruction executes and checks whether `i` is `nil`. If it is not `nil`, then the internal control variable (register 2) is set to the value in `i` and control goes back to to the start of the loop body (lines [5]–[8]).

The body of the generic `for` loop executes (`print(i,v)`) and then `TFORCALL` is encountered again, calling the iterator function to get the next iteration state. Finally, when the `TFORLOOP` finds that the first result from the iterator is `nil`, the loop ends, and execution continues on line [11].

6.14 OP_CLOSURE instruction

6.14.1 Syntax

```
CLOSURE A Bx    R(A) := closure(KPROTO[Bx])
```

6.14.2 Description

Creates an instance (or closure) of a function prototype. The `Bx` parameter identifies the entry in the parent function's table of closure prototypes (the field `p` in the struct `Proto`). The indices start from 0, i.e., a parameter of Bx = 0 references the first closure prototype in the table.

The `OP_CLOSURE` instruction also sets up the `upvalues` for the closure being defined. This is an involved process that is worthy of detailed discussion, and will be described through examples.

6.14.3 Examples

Let's start with a simple example of a Lua function:

```
f=load('function x() end; function y() end')
```

Here we are creating two Lua functions/closures within the main chunk. The bytecodes for the chunk look this:

```
main <(string):0,0> (5 instructions at 0000020E8A352930)
0+ params, 2 slots, 1 upvalue, 0 locals, 2 constants, 2 functions
        1       [1]     CLOSURE         0 0     ; 0000020E8A352A70
        2       [1]     SETTABUP        0 -1 0  ; _ENV "x"
        3       [1]     CLOSURE         0 1     ; 0000020E8A3536A0
        4       [1]     SETTABUP        0 -2 0  ; _ENV "y"
        5       [1]     RETURN          0 1
constants (2) for 0000020E8A352930:
        1       "x"
        2       "y"
locals (0) for 0000020E8A352930:
upvalues (1) for 0000020E8A352930:
        0       _ENV    1       0

function <(string):1,1> (1 instruction at 0000020E8A352A70)
0 params, 2 slots, 0 upvalues, 0 locals, 0 constants, 0 functions
        1       [1]     RETURN          0 1
constants (0) for 0000020E8A352A70:
locals (0) for 0000020E8A352A70:
upvalues (0) for 0000020E8A352A70:

function <(string):1,1> (1 instruction at 0000020E8A3536A0)
0 params, 2 slots, 0 upvalues, 0 locals, 0 constants, 0 functions
        1       [1]     RETURN          0 1
constants (0) for 0000020E8A3536A0:
locals (0) for 0000020E8A3536A0:
upvalues (0) for 0000020E8A3536A0:
```

What we observe is that the first `CLOSURE` instruction has parameter `Bx` set to 0, and this is the reference to the closure 0000020E8A352A70 which appears at position 0 in the table of closures within the main chunk's `Proto` structure.

Similarly the second `CLOSURE` instruction has parameter `Bx` set to 1, and this references the closure at position 1 in the table, which is 0000020E8A3536A0.

Other things to notice is that the main chunk got an automatic upvalue named `_ENV`:

```
upvalues (1) for 0000020E8A352930:
        0       _ENV    1       0
```

The first `0` is the index of the upvalue in the main chunk. The `1` following the name is a boolean indicating that the upvalue is located on the stack, and the last `0` is identifies the register location on the stack. So the Lua Parser has setup the `upvalue` reference for `_ENV`. However note that there is no actual local in this case; the `_ENV` upvalue is special and is setup by the Lua lua_load() API function.

Now let's look at an example that creates a local up-value:

```
f=load('local u,v; function p() return v end')
```

We get following bytecodes:

```
main <(string):0,0> (4 instructions at 0000022149BBA3B0)
0+ params, 3 slots, 1 upvalue, 2 locals, 1 constant, 1 function
        1       [1]     LOADNIL         0 1
        2       [1]     CLOSURE         2 0     ; 0000022149BBB7B0
        3       [1]     SETTABUP        0 -1 2  ; _ENV "p"
        4       [1]     RETURN          0 1
constants (1) for 0000022149BBA3B0:
        1       "p"
locals (2) for 0000022149BBA3B0:
        0       u       2       5
        1       v       2       5
upvalues (1) for 0000022149BBA3B0:
        0       _ENV    1       0

function <(string):1,1> (3 instructions at 0000022149BBB7B0)
0 params, 2 slots, 1 upvalue, 0 locals, 0 constants, 0 functions
        1       [1]     GETUPVAL        0 0     ; v
        2       [1]     RETURN          0 2
        3       [1]     RETURN          0 1
constants (0) for 0000022149BBB7B0:
locals (0) for 0000022149BBB7B0:
upvalues (1) for 0000022149BBB7B0:
        0       v       1       1
```

In the function ‘p’ the upvalue list contains:

```
upvalues (1) for 0000022149BBB7B0:
        0       v       1       1
```

This says that the up-value is in the stack (first ‘1’) and is located at register ‘1’ of the parent function. Access to this upvalue is indirectly obtained via the `GETUPVAL` instruction on line 1.

Now, lets look at what happens when the upvalue is not directly within the parent function:

```
f=load('local u,v; function p() u=1; local function q() return v end end')
```

In this example, we have 1 upvalue reference in function ‘p’, which is ‘u’. Function ‘q’ has one upvalue reference ‘v’ but this is not a variable in ‘p’, but is in the grand-parent. Here are the resulting bytecodes:

```
main <(string):0,0> (4 instructions at 0000022149BBFE40)
0+ params, 3 slots, 1 upvalue, 2 locals, 1 constant, 1 function
        1       [1]     LOADNIL         0 1
        2       [1]     CLOSURE         2 0     ; 0000022149BBFC60
        3       [1]     SETTABUP        0 -1 2  ; _ENV "p"
        4       [1]     RETURN          0 1
constants (1) for 0000022149BBFE40:
        1       "p"
locals (2) for 0000022149BBFE40:
        0       u       2       5
        1       v       2       5
upvalues (1) for 0000022149BBFE40:
        0       _ENV    1       0

function <(string):1,1> (4 instructions at 0000022149BBFC60)
0 params, 2 slots, 2 upvalues, 1 local, 1 constant, 1 function
        1       [1]     LOADK           0 -1    ; 1
        2       [1]     SETUPVAL        0 0     ; u
```

(continues on next page)

(continued from previous page)

```
      3       [1]     CLOSURE         0 0     ; 0000022149BC06B0
      4       [1]     RETURN          0 1
constants (1) for 0000022149BBFC60:
      1       1
locals (1) for 0000022149BBFC60:
      0       q       4       5
upvalues (2) for 0000022149BBFC60:
      0       u       1       0
      1       v       1       1

function <(string):1,1> (3 instructions at 0000022149BC06B0)
0 params, 2 slots, 1 upvalue, 0 locals, 0 constants, 0 functions
      1       [1]     GETUPVAL        0 0     ; v
      2       [1]     RETURN          0 2
      3       [1]     RETURN          0 1
constants (0) for 0000022149BC06B0:
locals (0) for 0000022149BC06B0:
upvalues (1) for 0000022149BC06B0:
      0       v       0       1
```

We see that ‘p’ got the upvalue ‘u’ as expected, but it also got the upvalue ‘v’, and both are marked as ‘instack’ of the parent function:

```
upvalues (2) for 0000022149BBFC60:
      0       u       1       0
      1       v       1       1
```

The reason for this is that any upvalue references in the inmost nested function will also appear in the parent functions up the chain until the function whose stack contains the variable being referenced. So although the function ‘p’ does not directly reference ‘v’, but because its child function ‘q’ references ‘v’, ‘p’ gets the upvalue reference to ‘v’ as well.

Observe the upvalue list of ‘q’ now:

```
upvalues (1) for 0000022149BC06B0:
      0       v       0       1
```

‘q’ has one upvalue reference as expected, but this time the upvalue is not marked ‘instack’, which means that the reference is to an upvalue and not a local in the parent function (in this case ‘p’) and the upvalue index is ‘1’ (i.e. the second upvalue in ‘p’).

6.14.4 Upvalue setup by OP_CLOSURE

When the `CLOSURE` instruction is executed, the up-values referenced by the prototype are resolved. So that means the actual resolution if upvalues occurs at runtime. This is done in the function pushclosure().

6.14.5 Caching of closures

The Lua VM maintains a cache of closures within each function prototype at runtime. If a closure is required that has the same set of upvalues as referenced by an existing closure then the VM reuses the existing closure rather than creating a new one. This is illustrated in this contrived example:

```
f=load('local v; local function q() return function() return v end end; return q(),␣
↪q()')
```

When the statement `return q(), q()` is executed it will end up returning two closures that are really the same instance, as shown by the result of executing this code:

```
> f()
function: 000001E1E2F007E0      function: 000001E1E2F007E0
```

6.15 OP_GETUPVAL and OP_SETUPVAL instructions

6.15.1 Syntax

```
GETUPVAL  A B     R(A) := UpValue[B]
SETUPVAL  A B     UpValue[B] := R(A)
```

6.15.2 Description

`GETUPVAL` copies the value in upvalue number `B` into register `R(A)`. Each Lua function may have its own upvalue list. This upvalue list is internal to the virtual machine; the list of upvalue name strings in a prototype is not mandatory.

`SETUPVAL` copies the value from register `R(A)` into the upvalue number `B` in the upvalue list for that function.

6.15.3 Examples

`GETUPVAL` and `SETUPVAL` instructions use internally-managed upvalue lists. The list of upvalue name strings that are found in a function prototype is for debugging purposes; it is not used by the Lua virtual machine and can be stripped by `luac`. During execution, upvalues are set up by a `CLOSURE`, and maintained by the Lua virtual machine. In the following example, function `b` is declared inside the main chunk, and is shown in the disassembly as a function prototype within a function prototype. The indentation, which is not in the original output, helps to visually separate the two functions.

```
f=load('local a; function b() a = 1 return a end')
```

Leads to:

```
main <(string):0,0> (4 instructions at 000002853D5177F0)
0+ params, 2 slots, 1 upvalue, 1 local, 1 constant, 1 function
        1       [1]     LOADNIL         0 0
        2       [1]     CLOSURE         1 0     ; 000002853D517920
        3       [1]     SETTABUP        0 -1 1  ; _ENV "b"
        4       [1]     RETURN          0 1
constants (1) for 000002853D5177F0:
        1       "b"
locals (1) for 000002853D5177F0:
        0       a       2       5
upvalues (1) for 000002853D5177F0:
        0       _ENV    1       0

  function <(string):1,1> (5 instructions at 000002853D517920)
  0 params, 2 slots, 1 upvalue, 0 locals, 1 constant, 0 functions
          1       [1]     LOADK           0 -1    ; 1
          2       [1]     SETUPVAL        0 0     ; a
          3       [1]     GETUPVAL        0 0     ; a
```

(continues on next page)

(continued from previous page)

```
        4       [1]     RETURN          0 2
        5       [1]     RETURN          0 1
constants (1) for 000002853D517920:
        1       1
locals (0) for 000002853D517920:
upvalues (1) for 000002853D517920:
        0       a       1       0
```

In the main chunk, the local `a` starts as a `nil`. The `CLOSURE` instruction in line [2] then instantiates a function closure with a single upvalue, `a`. In line [3] the closure is assigned to global `b` via the `SETTABUP` instruction.

In function `b`, there is a single upvalue, *a*. In Pascal, a variable in an outer scope is found by traversing stack frames. However, instantiations of Lua functions are first-class values, and they may be assigned to a variable and referenced elsewhere. Moreover, a single prototype may have multiple instantiations. Managing upvalues thus becomes a little more tricky than traversing stack frames in Pascal. The Lua virtual machine solution is to provide a clean interface to access upvalues via `GETUPVAL` and `SETUPVAL`, while the management of upvalues is handled by the virtual machine itself.

Line [2] in function `b` sets upvalue a (upvalue number 0 in the upvalue table) to a number value of `1` (held in temporary register `0`.) In line [3], the value in upvalue `a` is retrieved and placed into register `0`, where the following `RETURN` instruction will use it as a return value. The `RETURN` in line [5] is unused.

6.16 OP_NEWTABLE instruction

6.16.1 Syntax

```
NEWTABLE A B C   R(A) := {} (size = B,C)
```

6.16.2 Description

Creates a new empty table at register R(A). B and C are the encoded size information for the array part and the hash part of the table, respectively. Appropriate values for B and C are set in order to avoid rehashing when initially populating the table with array values or hash key-value pairs.

Operand B and C are both encoded as a 'floating point byte' (so named in lobject.c) which is `eeeeexxx` in binary, where x is the mantissa and e is the exponent. The actual value is calculated as `1xxx*2^(eeeee-1)` if `eeeee` is greater than `0` (a range of `8` to `15*2^30`). If `eeeee` is `0`, the actual value is `xxx` (a range of `0` to `7`.)

If an empty table is created, both sizes are zero. If a table is created with a number of objects, the code generator counts the number of array elements and the number of hash elements. Then, each size value is rounded up and encoded in B and C using the floating point byte format.

6.16.3 Examples

Creating an empty table forces both array and hash sizes to be zero:

```
f=load('local q = {}')
```

Leads to:

```
main <(string):0,0> (2 instructions at 0000022C1877A220)
0+ params, 2 slots, 1 upvalue, 1 local, 0 constants, 0 functions
      1       [1]     NEWTABLE        0 0 0
      2       [1]     RETURN          0 1
constants (0) for 0000022C1877A220:
locals (1) for 0000022C1877A220:
      0       q       2       3
upvalues (1) for 0000022C1877A220:
      0       _ENV    1       0
```

More examples are provided in the description of `OP_SETLIST` instruction.

6.17 OP_SETLIST instruction

6.17.1 Syntax

```
SETLIST A B C   R(A)[(C-1)*FPF+i] := R(A+i), 1 <= i <= B
```

6.17.2 Description

Sets the values for a range of array elements in a table referenced by R(A). Field B is the number of elements to set. Field C encodes the block number of the table to be initialized. The values used to initialize the table are located in registers R(A+1), R(A+2), and so on.

The block size is denoted by FPF. FPF is 'fields per flush', defined as `LFIELDS_PER_FLUSH` in the source file lopcodes.h, with a value of 50. For example, for array locations 1 to 20, C will be 1 and B will be 20.

If B is 0, the table is set with a variable number of array elements, from register R(A+1) up to the top of the stack. This happens when the last element in the table constructor is a function call or a vararg operator.

If C is 0, the next instruction is cast as an integer, and used as the C value. This happens only when operand C is unable to encode the block number, i.e. when C > 511, equivalent to an array index greater than 25550.

6.17.3 Examples

We'll start with a simple example:

```
f=load('local q = {1,2,3,4,5,}')
```

This generates:

```
main <(string):0,0> (8 instructions at 0000022C18756E50)
0+ params, 6 slots, 1 upvalue, 1 local, 5 constants, 0 functions
      1       [1]     NEWTABLE        0 5 0
      2       [1]     LOADK           1 -1    ; 1
      3       [1]     LOADK           2 -2    ; 2
      4       [1]     LOADK           3 -3    ; 3
      5       [1]     LOADK           4 -4    ; 4
      6       [1]     LOADK           5 -5    ; 5
      7       [1]     SETLIST         0 5 1   ; 1
      8       [1]     RETURN          0 1
constants (5) for 0000022C18756E50:
```

(continues on next page)

(continued from previous page)

```
      1       1
      2       2
      3       3
      4       4
      5       5
locals (1) for 0000022C18756E50:
      0       q       8       9
upvalues (1) for 0000022C18756E50:
      0       _ENV    1       0
```

A table with the reference in register 0 is created in line [1] by NEWTABLE. Since we are creating a table with no hash elements, the array part of the table has a size of 5, while the hash part has a size of 0.

Constants are then loaded into temporary registers 1 to 5 (lines [2] to [6]) before the SETLIST instruction in line [7] assigns each value to consecutive table elements. The start of the block is encoded as 1 in operand C. The starting index is calculated as (1-1)*50+1 or 1. Since B is 5, the range of the array elements to be set becomes 1 to 5, while the objects used to set the array elements will be R(1) through R(5).

Next is a larger table with 55 array elements. This will require two blocks to initialize. Some lines have been removed and ellipsis (. . .) added to save space:

```
> f=load('local q = {1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0, \
>> 1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0, \
>> 1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,}')
```

The generated code is:

```
main <(string):0,0> (59 instructions at 0000022C187833C0)
0+ params, 51 slots, 1 upvalue, 1 local, 10 constants, 0 functions
      1       [1]     NEWTABLE        0 30 0
      2       [1]     LOADK           1 -1    ; 1
      3       [1]     LOADK           2 -2    ; 2
      4       [1]     LOADK           3 -3    ; 3
      ...
      51      [3]     LOADK           50 -10  ; 0
      52      [3]     SETLIST         0 50 1  ; 1
      53      [3]     LOADK           1 -1    ; 1
      54      [3]     LOADK           2 -2    ; 2
      55      [3]     LOADK           3 -3    ; 3
      56      [3]     LOADK           4 -4    ; 4
      57      [3]     LOADK           5 -5    ; 5
      58      [3]     SETLIST         0 5 2   ; 2
      59      [3]     RETURN          0 1
constants (10) for 0000022C187833C0:
      1       1
      2       2
      3       3
      4       4
      5       5
      6       6
      7       7
      8       8
      9       9
      10      0
locals (1) for 0000022C187833C0:
      0       q       59      60
upvalues (1) for 0000022C187833C0:
```

(continues on next page)

(continued from previous page)

```
      0         _ENV     1         0
```

Since FPF is 50, the array will be initialized in two blocks. The first block is for index 1 to 50, while the second block is for index 51 to 55. Each array block to be initialized requires one `SETLIST` instruction. On line [1], `NEWTABLE` has a field B value of 30, or 00011110 in binary. From the description of `NEWTABLE`, `xxx` is `1102`, while `eeeee` is `112`. Thus, the size of the array portion of the table is `(1110)*2^(11-1)` or `(14*2^2)` or `56`.

Lines [2] to [51] sets the values used to initialize the first block. On line [52], `SETLIST` has a B value of 50 and a C value of 1. So the block is from 1 to 50. Source registers are from R(1) to R(50).

Lines [53] to [57] sets the values used to initialize the second block. On line [58], `SETLIST` has a B value of 5 and a C value of 2. So the block is from 51 to 55. The start of the block is calculated as `(2-1)*50+1` or `51`. Source registers are from R(1) to R(5).

Here is a table with hashed elements:

```
> f=load('local q = {a=1,b=2,c=3,d=4,e=5,f=6,g=7,h=8,}')
```

This results in:

```
main <(string):0,0> (10 instructions at 0000022C18783D20)
0+ params, 2 slots, 1 upvalue, 1 local, 16 constants, 0 functions
      1         [1]      NEWTABLE        0 0 8
      2         [1]      SETTABLE        0 -1 -2 ; "a" 1
      3         [1]      SETTABLE        0 -3 -4 ; "b" 2
      4         [1]      SETTABLE        0 -5 -6 ; "c" 3
      5         [1]      SETTABLE        0 -7 -8 ; "d" 4
      6         [1]      SETTABLE        0 -9 -10        ; "e" 5
      7         [1]      SETTABLE        0 -11 -12       ; "f" 6
      8         [1]      SETTABLE        0 -13 -14       ; "g" 7
      9         [1]      SETTABLE        0 -15 -16       ; "h" 8
      10        [1]      RETURN          0 1
constants (16) for 0000022C18783D20:
      1         "a"
      2         1
      3         "b"
      4         2
      5         "c"
      6         3
      7         "d"
      8         4
      9         "e"
      10        5
      11        "f"
      12        6
      13        "g"
      14        7
      15        "h"
      16        8
locals (1) for 0000022C18783D20:
      0         q        10       11
upvalues (1) for 0000022C18783D20:
      0         _ENV     1         0
```

In line [1], `NEWTABLE` is executed with an array part size of 0 and a hash part size of 8.

On lines [2] to line [9], key-value pairs are set using `SETTABLE`. The `SETLIST` instruction is only for initializing array elements. Using `SETTABLE` to initialize the key-value pairs of a table in the above example is quite efficient as it can reference the constant pool directly.

If there are both array elements and hash elements in a table constructor, both SETTABLE and SETLIST will be used to initialize the table after the initial NEWTABLE. In addition, if the last element of the table constructor is a function call or a vararg operator, then the B operand of SETLIST will be 0, to allow objects from R(A+1) up to the top of the stack to be initialized as array elements of the table.

```
> f=load('return {1,2,3,a=1,b=2,c=3,foo()}')
```

Leads to:

```
main <(string):0,0> (12 instructions at 0000022C18788430)
0+ params, 5 slots, 1 upvalue, 0 locals, 7 constants, 0 functions
        1       [1]     NEWTABLE        0 3 3
        2       [1]     LOADK           1 -1    ; 1
        3       [1]     LOADK           2 -2    ; 2
        4       [1]     LOADK           3 -3    ; 3
        5       [1]     SETTABLE        0 -4 -1 ; "a" 1
        6       [1]     SETTABLE        0 -5 -2 ; "b" 2
        7       [1]     SETTABLE        0 -6 -3 ; "c" 3
        8       [1]     GETTABUP        4 0 -7  ; _ENV "foo"
        9       [1]     CALL            4 1 0
        10      [1]     SETLIST         0 0 1   ; 1
        11      [1]     RETURN          0 2
        12      [1]     RETURN          0 1
constants (7) for 0000022C18788430:
        1       1
        2       2
        3       3
        4       "a"
        5       "b"
        6       "c"
        7       "foo"
locals (0) for 0000022C18788430:
upvalues (1) for 0000022C18788430:
        0       _ENV    1       0
```

In the above example, the table is first created in line [1] with its reference in register 0, and it has both array and hash elements to be set. The size of the array part is 3 while the size of the hash part is also 3.

Lines [2]–[4] loads the values for the first 3 array elements. Lines [5]–[7] set the 3 key-value pairs for the hash part of the table. In lines [8] and [9], the call to function foo is made, and then in line [10], the SETLIST instruction sets the first 3 array elements (in registers 1 to 3) plus whatever additional results returned by the foo function call (from register 4 onwards). This is accomplished by setting operand B in SETLIST to 0. For the first block, operand C is 1 as usual. If no results are returned by the function, the top of stack is at register 3 and only the 3 constant array elements in the table are set.

Finally:

```
> f=load('local a; return {a(), a(), a()}')
```

This gives:

```
main <(string):0,0> (11 instructions at 0000022C18787AD0)
0+ params, 5 slots, 1 upvalue, 1 local, 0 constants, 0 functions
        1       [1]     LOADNIL         0 0
        2       [1]     NEWTABLE        1 2 0
        3       [1]     MOVE            2 0
```

(continues on next page)

(continued from previous page)

```
        4       [1]     CALL            2 1 2
        5       [1]     MOVE            3 0
        6       [1]     CALL            3 1 2
        7       [1]     MOVE            4 0
        8       [1]     CALL            4 1 0
        9       [1]     SETLIST         1 0 1   ; 1
        10      [1]     RETURN          1 2
        11      [1]     RETURN          0 1
constants (0) for 0000022C18787AD0:
locals (1) for 0000022C18787AD0:
        0       a       2       12
upvalues (1) for 0000022C18787AD0:
        0       _ENV    1       0
```

Note that only the last function call in a table constructor retains all results. Other function calls in the table constructor keep only one result. This is shown in the above example. For vararg operators in table constructors, please see the discussion for the VARARG instruction for an example.

6.18 OP_GETTABLE and OP_SETTABLE instructions

6.18.1 Syntax

```
GETTABLE A B C   R(A) := R(B)[RK(C)]
SETTABLE A B C   R(A)[RK(B)] := RK(C)
```

6.18.2 Description

OP_GETTABLE copies the value from a table element into register R(A). The table is referenced by register R(B), while the index to the table is given by RK(C), which may be the value of register R(C) or a constant number.

OP_SETTABLE copies the value from register R(C) or a constant into a table element. The table is referenced by register R(A), while the index to the table is given by RK(B), which may be the value of register R(B) or a constant number.

All 3 operand fields are used, and some of the operands can be constants. A constant is specified by setting the MSB of the operand to 1. If RK(C) need to refer to constant 1, the encoded value will be (256 | 1) or 257, where 256 is the value of bit 8 of the operand. Allowing constants to be used directly reduces considerably the need for temporary registers.

6.18.3 Examples

```
f=load('local p = {}; p[1] = "foo"; return p["bar"]')
```

This compiles to:

```
main <(string):0,0> (5 instructions at 000001FA06FCC3F0)
0+ params, 2 slots, 1 upvalue, 1 local, 3 constants, 0 functions
        1       [1]     NEWTABLE        0 0 0
        2       [1]     SETTABLE        0 -1 -2 ; 1 "foo"
        3       [1]     GETTABLE        1 0 -3  ; "bar"
```

(continues on next page)

(continued from previous page)

```
      4       [1]     RETURN          1 2
      5       [1]     RETURN          0 1
constants (3) for 000001FA06FCC3F0:
      1       1
      2       "foo"
      3       "bar"
locals (1) for 000001FA06FCC3F0:
      0       p       2       6
upvalues (1) for 000001FA06FCC3F0:
      0       _ENV    1       0
```

In line [1], a new empty table is created and the reference placed in local p (register 0). Creating and populating new tables is discussed in detail elsewhere. Table index 1 is set to 'foo' in line [2] by the SETTABLE instruction.

The R(A) value of 0 points to the new table that was defined in line [1]. In line [3], the value of the table element indexed by the string 'bar' is copied into temporary register 1, which is then used by RETURN as a return value.

6.19 OP_SELF instruction

6.19.1 Syntax

```
SELF  A B C   R(A+1) := R(B); R(A) := R(B)[RK(C)]
```

6.19.2 Description

For object-oriented programming using tables. Retrieves a function reference from a table element and places it in register R(A), then a reference to the table itself is placed in the next register, R(A+1). This instruction saves some messy manipulation when setting up a method call.

R(B) is the register holding the reference to the table with the method. The method function itself is found using the table index RK(C), which may be the value of register R(C) or a constant number.

6.19.3 Examples

A SELF instruction saves an extra instruction and speeds up the calling of methods in object oriented programming. It is only generated for method calls that use the colon syntax. In the following example:

```
f=load('foo:bar("baz")')
```

We can see SELF being generated:

```
main <(string):0,0> (5 instructions at 000001FA06FA7830)
0+ params, 3 slots, 1 upvalue, 0 locals, 3 constants, 0 functions
      1       [1]     GETTABUP        0 0 -1  ; _ENV "foo"
      2       [1]     SELF            0 0 -2  ; "bar"
      3       [1]     LOADK           2 -3    ; "baz"
      4       [1]     CALL            0 3 1
      5       [1]     RETURN          0 1
constants (3) for 000001FA06FA7830:
      1       "foo"
      2       "bar"
```

(continues on next page)

(continued from previous page)

```
      3       "baz"
locals (0) for 000001FA06FA7830:
upvalues (1) for 000001FA06FA7830:
      0       _ENV    1       0
```

The method call is equivalent to: `foo.bar(foo, "baz")`, except that the global `foo` is only looked up once. This is significant if metamethods have been set. The `SELF` in line [2] is equivalent to a `GETTABLE` lookup (the table is in register 0 and the index is constant 1) and a `MOVE` (copying the table reference from register 0 to register 1.)

Without `SELF`, a `GETTABLE` will write its lookup result to register 0 (which the code generator will normally do) and the table reference will be overwritten before a `MOVE` can be done. Using `SELF` saves roughly one instruction and one temporary register slot.

After setting up the method call using `SELF`, the call is made with the usual `CALL` instruction in line [4], with two parameters. The equivalent code for a method lookup is compiled in the following manner:

```
f=load('foo.bar(foo, "baz")')
```

And generated code:

```
main <(string):0,0> (6 instructions at 000001FA06FA6960)
0+ params, 3 slots, 1 upvalue, 0 locals, 3 constants, 0 functions
      1       [1]     GETTABUP        0 0 -1  ; _ENV "foo"
      2       [1]     GETTABLE        0 0 -2  ; "bar"
      3       [1]     GETTABUP        1 0 -1  ; _ENV "foo"
      4       [1]     LOADK           2 -3    ; "baz"
      5       [1]     CALL            0 3 1
      6       [1]     RETURN          0 1
constants (3) for 000001FA06FA6960:
      1       "foo"
      2       "bar"
      3       "baz"
locals (0) for 000001FA06FA6960:
upvalues (1) for 000001FA06FA6960:
      0       _ENV    1       0
```

The alternative form of a method call is one instruction longer, and the user must take note of any metamethods that may affect the call. The `SELF` in the previous example replaces the `GETTABLE` on line [2] and the `GETTABUP` on line [3]. If `foo` is a local variable, then the equivalent code is a `GETTABLE` and a `MOVE`.

6.20 OP_GETTABUP and OP_SETTABUP instructions

6.20.1 Syntax

```
GETTABUP A B C   R(A) := UpValue[B][RK(C)]
SETTABUP A B C   UpValue[A][RK(B)] := RK(C)
```

6.20.2 Description

`OP_GETTABUP` and `OP_SETTABUP` instructions are similar to the `OP_GETTABLE` and `OP_SETTABLE` instructions except that the table is referenced as an upvalue. These instructions are used to access global variables, which since Lua 5.2 are accessed via the upvalue named `_ENV`.

6.20.3 Examples

```
f=load('a = 40; local b = a')
```

Results in:

```
main <(string):0,0> (3 instructions at 0000028D955FEBF0)
0+ params, 2 slots, 1 upvalue, 1 local, 2 constants, 0 functions
        1       [1]     SETTABUP        0 -1 -2 ; _ENV "a" 40
        2       [1]     GETTABUP        0 0 -1  ; _ENV "a"
        3       [1]     RETURN          0 1
constants (2) for 0000028D955FEBF0:
        1       "a"
        2       40
locals (1) for 0000028D955FEBF0:
        0       b       3       4
upvalues (1) for 0000028D955FEBF0:
        0       _ENV    1       0
```

From the example, we can see that ‘b’ is the name of the local variable while ‘a’ is the name of the global variable.

Line [1] assigns the number 40 to global ‘a’. Line [2] assigns the value in global ‘a’ to the register 0 which is the local ‘b’.

6.21 OP_CONCAT instruction

6.21.1 Syntax

```
CONCAT A B C   R(A) := R(B).. ... ..R(C)
```

6.21.2 Description

Performs concatenation of two or more strings. In a Lua source, this is equivalent to one or more concatenation operators (‘..’) between two or more expressions. The source registers must be consecutive, and C must always be greater than B. The result is placed in R(A).

6.21.3 Examples

`CONCAT` accepts a range of registers. Doing more than one string concatenation at a time is faster and more efficient than doing them separately:

```
f=load('local x,y = "foo","bar"; return x..y..x..y')
```

Generates:

```
main <(string):0,0> (9 instructions at 0000028D9560B290)
0+ params, 6 slots, 1 upvalue, 2 locals, 2 constants, 0 functions
        1       [1]     LOADK           0 -1    ; "foo"
        2       [1]     LOADK           1 -2    ; "bar"
        3       [1]     MOVE            2 0
        4       [1]     MOVE            3 1
```

(continues on next page)

(continued from previous page)

```
      5       [1]     MOVE            4 0
      6       [1]     MOVE            5 1
      7       [1]     CONCAT          2 2 5
      8       [1]     RETURN          2 2
      9       [1]     RETURN          0 1
constants (2) for 0000028D9560B290:
      1       "foo"
      2       "bar"
locals (2) for 0000028D9560B290:
      0       x       3       10
      1       y       3       10
upvalues (1) for 0000028D9560B290:
      0       _ENV    1       0
```

In this example, strings are moved into place first (lines [3] to [6]) in the concatenation order before a single `CONCAT` instruction is executed in line [7]. The result is left in temporary local 2, which is then used as a return value by the `RETURN` instruction on line [8].

```
f=load('local a = "foo".."bar".."baz"')
```

Compiles to:

```
main <(string):0,0> (5 instructions at 0000028D9560EE40)
0+ params, 3 slots, 1 upvalue, 1 local, 3 constants, 0 functions
      1       [1]     LOADK           0 -1    ; "foo"
      2       [1]     LOADK           1 -2    ; "bar"
      3       [1]     LOADK           2 -3    ; "baz"
      4       [1]     CONCAT          0 0 2
      5       [1]     RETURN          0 1
constants (3) for 0000028D9560EE40:
      1       "foo"
      2       "bar"
      3       "baz"
locals (1) for 0000028D9560EE40:
      0       a       5       6
upvalues (1) for 0000028D9560EE40:
      0       _ENV    1       0
```

In the second example, three strings are concatenated together. Note that there is no string constant folding. Lines [1] through [3] loads the three constants in the correct order for concatenation; the `CONCAT` on line [4] performs the concatenation itself and assigns the result to local 'a'.

6.22 OP_LEN instruction

6.22.1 Syntax

```
LEN A B     R(A) := length of R(B)
```

6.22.2 Description

Returns the length of the object in R(B). For strings, the string length is returned, while for tables, the table size (as defined in Lua) is returned. For other objects, the metamethod is called. The result, which is a number, is placed in

R(A).

6.22.3 Examples

The LEN operation implements the # operator. If # operates on a constant, then the constant is loaded in advance using LOADK. The LEN instruction is currently not optimized away using compile time evaluation, even if it is operating on a constant string or table:

```
f=load('local a,b; a = #b; a= #"foo"')
```

Results in:

```
main <(string):0,0> (5 instructions at 000001DC21778C60)
0+ params, 3 slots, 1 upvalue, 2 locals, 1 constant, 0 functions
        1       [1]     LOADNIL         0 1
        2       [1]     LEN             0 1
        3       [1]     LOADK           2 -1    ; "foo"
        4       [1]     LEN             0 2
        5       [1]     RETURN          0 1
constants (1) for 000001DC21778C60:
        1       "foo"
locals (2) for 000001DC21778C60:
        0       a       2       6
        1       b       2       6
upvalues (1) for 000001DC21778C60:
        0       _ENV    1       0
```

In the above example, LEN operates on local b in line [2], leaving the result in local a. Since LEN cannot operate directly on constants, line [3] first loads the constant "foo" into a temporary local, and only then LEN is executed.

6.23 OP_MOVE instruction

6.23.1 Syntax

```
MOVE A B     R(A) := R(B)
```

6.23.2 Description

Copies the value of register R(B) into register R(A). If R(B) holds a table, function or userdata, then the reference to that object is copied. MOVE is often used for moving values into place for the next operation.

6.23.3 Examples

The most straightforward use of MOVE is for assigning a local to another local:

```
f=load('local a,b = 10; b = a')
```

Produces:

```
main <(string):0,0> (4 instructions at 000001DC217566D0)
0+ params, 2 slots, 1 upvalue, 2 locals, 1 constant, 0 functions
       1       [1]     LOADK           0 -1    ; 10
       2       [1]     LOADNIL         1 0
       3       [1]     MOVE            1 0
       4       [1]     RETURN          0 1
constants (1) for 000001DC217566D0:
       1       10
locals (2) for 000001DC217566D0:
       0       a       3       5
       1       b       3       5
upvalues (1) for 000001DC217566D0:
       0       _ENV    1       0
```

You won't see `MOVE` instructions used in arithmetic expressions because they are not needed by arithmetic operators. All arithmetic operators are in 2- or 3-operand style: the entire local stack frame is already visible to operands R(A), R(B) and R(C) so there is no need for any extra `MOVE` instructions.

Other places where you will see `MOVE` are:

- When moving parameters into place for a function call.
- When moving values into place for certain instructions where stack order is important, e.g. `GETTABLE`, `SETTABLE` and `CONCAT`.
- When copying return values into locals after a function call.

6.24 OP_LOADNIL instruction

6.24.1 Syntax

```
LOADNIL A B     R(A), R(A+1), ..., R(A+B) := nil
```

6.24.2 Description

Sets a range of registers from R(A) to R(B) to nil. If a single register is to be assigned to, then R(A) = R(B). When two or more consecutive locals need to be assigned nil values, only a single `LOADNIL` is needed.

6.24.3 Examples

`LOADNIL` uses the operands A and B to mean a range of register locations. The example for `MOVE` earlier shows `LOADNIL` used to set a single register to `nil`.

```
f=load('local a,b,c,d,e = nil,nil,0')
```

Generates:

```
main <(string):0,0> (4 instructions at 000001DC21780390)
0+ params, 5 slots, 1 upvalue, 5 locals, 1 constant, 0 functions
       1       [1]     LOADNIL         0 1
       2       [1]     LOADK           2 -1    ; 0
       3       [1]     LOADNIL         3 1
```

(continues on next page)

(continued from previous page)

```
      4       [1]     RETURN          0 1
constants (1) for 000001DC21780390:
      1       0
locals (5) for 000001DC21780390:
      0       a       4       5
      1       b       4       5
      2       c       4       5
      3       d       4       5
      4       e       4       5
upvalues (1) for 000001DC21780390:
      0       _ENV    1       0
```

Line [1] nils locals a and b. Local c is explicitly initialized with the value 0. Line [3] nils d and e.

6.25 OP_LOADK instruction

6.25.1 Syntax

```
LOADK A Bx    R(A) := Kst(Bx)
```

6.25.2 Description

Loads constant number Bx into register R(A). Constants are usually numbers or strings. Each function prototype has its own constant list, or pool.

6.25.3 Examples

`LOADK` loads a constant from the constant list into a register or local. Constants are indexed starting from 0. Some instructions, such as arithmetic instructions, can use the constant list without needing a `LOADK`. Constants are pooled in the list, duplicates are eliminated. The list can hold nils, booleans, numbers or strings.

```
f=load('local a,b,c,d = 3,"foo",3,"foo"')
```

Leads to:

```
main <(string):0,0> (5 instructions at 000001DC21780B50)
0+ params, 4 slots, 1 upvalue, 4 locals, 2 constants, 0 functions
      1       [1]     LOADK           0 -1    ; 3
      2       [1]     LOADK           1 -2    ; "foo"
      3       [1]     LOADK           2 -1    ; 3
      4       [1]     LOADK           3 -2    ; "foo"
      5       [1]     RETURN          0 1
constants (2) for 000001DC21780B50:
      1       3
      2       "foo"
locals (4) for 000001DC21780B50:
      0       a       5       6
      1       b       5       6
      2       c       5       6
      3       d       5       6
```

(continues on next page)

(continued from previous page)

```
upvalues (1) for 000001DC21780B50:
        0       _ENV    1       0
```

The constant 3 and the constant "foo" are both written twice in the source snippet, but in the constant list, each constant has a single location.

6.26 Binary operators

Lua 5.3 implements a bunch of binary operators for arithmetic and bitwise manipulation of variables. These insructions have a common form.

6.26.1 Syntax

```
ADD   A B C   R(A) := RK(B) + RK(C)
SUB   A B C   R(A) := RK(B) - RK(C)
MUL   A B C   R(A) := RK(B) * RK(C)
MOD   A B C   R(A) := RK(B) % RK(C)
POW   A B C   R(A) := RK(B) ^ RK(C)
DIV   A B C   R(A) := RK(B) / RK(C)
IDIV  A B C   R(A) := RK(B) // RK(C)
BAND  A B C   R(A) := RK(B) & RK(C)
BOR   A B C   R(A) := RK(B) | RK(C)
BXOR  A B C   R(A) := RK(B) ~ RK(C)
SHL   A B C   R(A) := RK(B) << RK(C)
SHR   A B C   R(A) := RK(B) >> RK(C)
```

6.26.2 Description

Binary operators (arithmetic operators and bitwise operators with two inputs.) The result of the operation between RK(B) and RK(C) is placed into R(A). These instructions are in the classic 3-register style.

RK(B) and RK(C) may be either registers or constants in the constant pool.

Opcode	Description
ADD	Addition operator
SUB	Subtraction operator
MUL	Multiplication operator
MOD	Modulus (remainder) operator
POW	Exponentation operator
DIV	Division operator
IDIV	Integer division operator
BAND	Bit-wise AND operator
BOR	Bit-wise OR operator
BXOR	Bit-wise Exclusive OR operator
SHL	Shift bits left
SHR	Shift bits right

The source operands, RK(B) and RK(C), may be constants. If a constant is out of range of field B or field C, then the constant will be loaded into a temporary register in advance.

6.26.3 Examples

```
f=load('local a,b = 2,4; a = a + 4 * b - a / 2 ^ b % 3')
```

Generates:

```
main <(string):0,0> (9 instructions at 000001DC21781DD0)
0+ params, 4 slots, 1 upvalue, 2 locals, 3 constants, 0 functions
        1       [1]     LOADK           0 -1    ; 2
        2       [1]     LOADK           1 -2    ; 4
        3       [1]     MUL             2 -2 1  ; 4 -       (loc2 = 4 * b)
        4       [1]     ADD             2 0 2               (loc2 = A + loc2)
        5       [1]     POW             3 -1 1  ; 2 -       (loc3 = 2 ^ b)
        6       [1]     DIV             3 0 3               (loc3 = a / loc3)
        7       [1]     MOD             3 3 -3              (loc3 = loc3 % 3)
        8       [1]     SUB             0 2 3               (a = loc2 - loc3)
        9       [1]     RETURN          0 1
constants (3) for 000001DC21781DD0:
        1       2
        2       4
        3       3
locals (2) for 000001DC21781DD0:
        0       a       3       10
        1       b       3       10
upvalues (1) for 000001DC21781DD0:
        0       _ENV    1       0
```

In the disassembly shown above, parts of the expression is shown as additional comments in parentheses. Each arithmetic operator translates into a single instruction. This also means that while the statement `count = count + 1` is verbose, it translates into a single instruction if count is a local. If count is a global, then two extra instructions are required to read and write to the global (`GETTABUP` and `SETTABUP`), since arithmetic operations can only be done on registers (locals) only.

The Lua parser and code generator can perform limited constant expression folding or evaluation. Constant folding only works for binary arithmetic operators and the unary minus operator (`UNM`, which will be covered next.) There is no equivalent optimization for relational, boolean or string operators.

The optimization rule is simple: If both terms of a subexpression are numbers, the subexpression will be evaluated at compile time. However, there are exceptions. One, the code generator will not attempt to divide a number by 0 for DIV and MOD, and two, if the result is evaluated as a NaN (Not a Number) then the optimization will not be performed.

Also, constant folding is not done if one term is in the form of a string that need to be coerced. In addition, expression terms are not rearranged, so not all optimization opportunities can be recognized by the code generator. This is intentional; the Lua code generator is not meant to perform heavy duty optimizations, as Lua is a lightweight language. Here are a few examples to illustrate how it works (additional comments in parentheses):

```
f=load('local a = 4 + 7 + b; a = b + 4 * 7; a = b + 4 + 7')
```

Generates:

```
main <(string):0,0> (8 instructions at 000001DC21781650)
0+ params, 2 slots, 1 upvalue, 1 local, 5 constants, 0 functions
        1       [1]     GETTABUP        0 0 -1  ; _ENV "b"
        2       [1]     ADD             0 -2 0  ; 11 -              (a = 11 + b)
        3       [1]     GETTABUP        1 0 -1  ; _ENV "b"
        4       [1]     ADD             0 1 -3  ; - 28              (a = b + 28)
```

(continues on next page)

(continued from previous page)

```
        5        [1]      GETTABUP         1 0 -1  ; _ENV "b"
        6        [1]      ADD              1 1 -4  ; - 4              (loc1 = b + 4)
        7        [1]      ADD              0 1 -5  ; - 7              (a = loc1 + 7)
        8        [1]      RETURN           0 1
constants (5) for 000001DC21781650:
        1        "b"
        2        11
        3        28
        4        4
        5        7
locals (1) for 000001DC21781650:
        0        a        3        9
upvalues (1) for 000001DC21781650:
        0        _ENV     1        0
```

For the first assignment statement, `4+7` is evaluated, thus 11 is added to b in line [2]. Next, in line [3] and [4], `b` and `28` are added together and assigned to a because multiplication has a higher precedence and `4*7` is evaluated first. Finally, on lines [5] to [7], there are two addition operations. Since addition is left-associative, code is generated for `b+4` first, and only after that, `7` is added. So in the third example, Lua performs no optimization. This can be fixed using parentheses to explicitly change the precedence of a subexpression:

```
f=load('local a = b + (4 + 7)')
```

And this leads to:

```
main <(string):0,0> (3 instructions at 000001DC21781EC0)
0+ params, 2 slots, 1 upvalue, 1 local, 2 constants, 0 functions
        1        [1]      GETTABUP         0 0 -1  ; _ENV "b"
        2        [1]      ADD              0 0 -2  ; - 11
        3        [1]      RETURN           0 1
constants (2) for 000001DC21781EC0:
        1        "b"
        2        11
locals (1) for 000001DC21781EC0:
        0        a        3        4
upvalues (1) for 000001DC21781EC0:
        0        _ENV     1        0
```

Now, the `4+7` subexpression can be evaluated at compile time. If the statement is written as:

```
local a = 7 + (4 + 7)
```

the code generator will generate a single `LOADK` instruction; Lua first evaluates `4+7`, then `7` is added, giving a total of `18`. The arithmetic expression is completely evaluated in this case, thus no arithmetic instructions are generated.

In order to make full use of constant folding in Lua, the user just need to remember the usual order of evaluation of an expression's elements and apply parentheses where necessary. The following are two expressions which will not be evaluated at compile time:

```
f=load('local a = 1 / 0; local b = 1 + "1"')
```

This produces:

```
main <(string):0,0> (3 instructions at 000001DC21781380)
0+ params, 2 slots, 1 upvalue, 2 locals, 3 constants, 0 functions
        1        [1]      DIV              0 -2 -1 ; 1 0
```

(continues on next page)

(continued from previous page)

```
        2       [1]     ADD             1 -2 -3 ; 1 "1"
        3       [1]     RETURN          0 1
constants (3) for 000001DC21781380:
        1       0
        2       1
        3       "1"
locals (2) for 000001DC21781380:
        0       a       2       4
        1       b       3       4
upvalues (1) for 000001DC21781380:
        0       _ENV    1       0
```

The first is due to a divide-by-0, while the second is due to a string constant that needs to be coerced into a number. In both cases, constant folding is not performed, so the arithmetic instructions needed to perform the operations at run time are generated instead.

TODO - examples of bitwise operators.

6.27 Unary operators

Lua 5.3 implements following unary operators in addition to `OP_LEN`.

6.27.1 Syntax

```
UNM   A B     R(A) := -R(B)
BNOT  A B     R(A) := ~R(B)
NOT   A B     R(A) := not R(B)
```

6.27.2 Description

The unary operators perform an operation on R(B) and store the result in R(A).

Opcode	Description
UNM	Unary minus
BNOT	Bit-wise NOT operator
NOT	Logical NOT operator

6.27.3 Examples

```
f=load('local p,q = 10,false; q,p = -p,not q')
```

Results in:

```
main <(string):0,0> (6 instructions at 000001DC21781290)
0+ params, 3 slots, 1 upvalue, 2 locals, 1 constant, 0 functions
        1       [1]     LOADK           0 -1    ; 10
        2       [1]     LOADBOOL        1 0 0
        3       [1]     UNM             2 0
```

(continues on next page)

(continued from previous page)

```
      4       [1]     NOT             0 1
      5       [1]     MOVE            1 2
      6       [1]     RETURN          0 1
constants (1) for 000001DC21781290:
      1       10
locals (2) for 000001DC21781290:
      0       p       3       7
      1       q       3       7
upvalues (1) for 000001DC21781290:
      0       _ENV    1       0
```

As UNM and NOT do not accept a constant as a source operand, making the LOADK on line [1] and the LOADBOOL on line [2] necessary. When an unary minus is applied to a constant number, the unary minus is optimized away. Similarly, when a not is applied to true or false, the logical operation is optimized away.

In addition to this, constant folding is performed for unary minus, if the term is a number. So, the expression in the following is completely evaluated at compile time:

```
f=load('local a = - (7 / 4)')
```

Results in:

```
main <(string):0,0> (2 instructions at 000001DC217810B0)
0+ params, 2 slots, 1 upvalue, 1 local, 1 constant, 0 functions
      1       [1]     LOADK           0 -1    ; -1.75
      2       [1]     RETURN          0 1
constants (1) for 000001DC217810B0:
      1       -1.75
locals (1) for 000001DC217810B0:
      0       a       2       3
upvalues (1) for 000001DC217810B0:
      0       _ENV    1       0
```

Constant folding is performed on 7/4 first. Then, since the unary minus operator is applied to the constant 1.75, constant folding can be performed again, and the code generated becomes a simple LOADK (on line [1]).

TODO - example of BNOT.

CHAPTER 7

Lua Parsing and Code Generation Internals

7.1 Stack and Registers

Lua employs two stacks. The `Callinfo` stack tracks activation frames. There is the secondary stack `L->stack` that is an array of `TValue` objects. The `Callinfo` objects index into this array. Registers are basically slots in the `L->stack` array.

When a function is called - the stack is setup as follows:

```
stack
|            function reference
|  base->    parameter 1
|            ...
|            parameter n
|            local 1
|            ...
|            local n
|  top->
|
V
```

So top is just past the registers needed by the function. The number of registers is determined based on locals and temporaries.

The base of the stack is set to just past the function reference - i.e. on the first parameter or register. All register addressing is done as offset from base - so `R(0)` is at `base+0` on the stack.

A description of the stack and registers from Mike Pall on Lua mailing list is reproduced below.

7.1.1 Sliding Register Window - by Mike Pall

Note: this is a reformatted version of a post on Lua mailing list (see MP6 link below).

The Lua 5 VM employs a sliding register window on top of a stack. Frames (named CallInfo aka ‘ci’ in the source) occupy different (overlapping) ranges on the stack. Successive frames are positioned exactly over the passed arguments

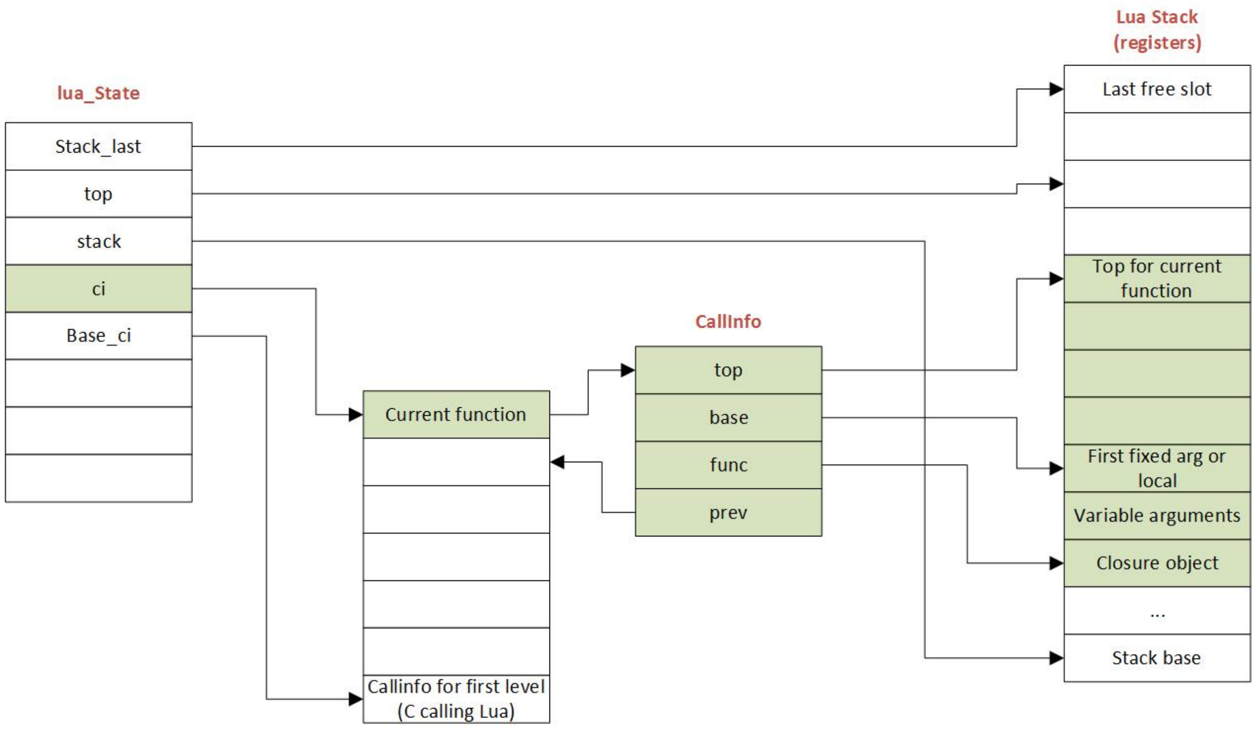

Fig. 1: The figure shows how the stack is related to other Lua objects.

(luaD_precall). The compiler ensures that there are no live variables after the arguments for a call. Return values need to be copied down (with truncate/extend) to the slot holding the function object (luaD_poscall). This is because the compiler has no idea how many values another function may return – only how many need to be stored.

Example:

```
function f2(arg1, arg2, ..., argN)
  local local1, local2, ...
  ...
  return ret1, ret2, ..., retO
end

function f1(arg1, arg2, ..., argM)
  local local1, local2, ...
  ...
  local ret1, ret2, ..., retP = f2(arg1, arg2, ..., argN)
  ...
end
```

Simplified stack diagram:

```
stack
|
| time: >>>> call >>>>>>>>>>>>>>>>>> call ~~~~~~~~~~~~~~~~~~ return >>>>>
|
|   ciX.func-> f1       f1       f1                                 f1
|   ciX.base-> arg1     arg1     arg1                               arg1
|              arg2     arg2     arg2                               arg2
|              ...      ...      ...                                ...
```

(continues on next page)

(continued from previous page)

```
|              argM    argM    argM                               argM
|   ciX.topC->         local1  local1                             local11
|                      local2  local2                             local2
|                      local3  local3                             local3
|                      ...     ...                                ...
|                              f2      ciY.func-> f2      f2      ret1
|                              arg1    ciY.base-> arg1    arg1    ret2
|                              arg2               arg2    arg2    ...
|                              ...                ...     ...     retP
|                              argN               argN    argN
|   ciX.topL-> ------  ------  ------  ciY.topC-> local1  local1
|                                                 local2  local2
|                                                 ...     ...
|                                                         ret1
|                                                         ret2
|                                                         ...
|                                                         retO
|                                      ciY.topL-> ------  ------
V
```

Note that there is only a single 'top' for each frame:

For Lua functions the top (tagged topL in the diagram) is set to the base plus the maximum number of slots used. The compiler knows this and stores it in the function prototype. The top pointer is used only temporarily for handling variable length argument and return value lists.

For C functions the top (tagged topC in the diagram) is initially set to the base plus the number of passed arguments. C functions can access their part of the stack via Lua API calls which in turn change the stack top. C functions return an integer that indicates the number of return values relative to the stack top.

In reality things are a bit more complex due to overlapped locals, block scopes, varargs, coroutines and a few other things. But this should get you the basic idea.

7.2 Parsing and Code Generation

- The parser is in lparser.c.
- The code generator is in both above and lcode.c.

The parser and code generator are arguably the most complex piece in the whole of Lua. The parser is one-pass - and generates code as it parses. That is, there is no AST build phase. This is primarily for efficiency it seems. The parser uses data structures on the stack - there are no heap allocated structures. Where needed the C stack itself is used to build structures - for example, as the assignment statement is parsed, there is recursion, and a stack based structure is built that links to structures in the call stack.

The main object used by the parser is the `struct expdesc`:

```
typedef struct expdesc {
  expkind k;
  union {
    struct {  /* for indexed variables (VINDEXED) */
      short idx;  /* index (R/K) */
      lu_byte t;  /* table (register or upvalue) */
      lu_byte vt;  /* whether 't' is register (VLOCAL) or upvalue (VUPVAL) */
    } ind;
    int info;  /* for generic use */
```

(continues on next page)

(continued from previous page)

```
    lua_Number nval;  /* for VKFLT */
    lua_Integer ival;    /* for VKINT */
  } u;
  int t;  /* patch list of 'exit when true' */
  int f;  /* patch list of 'exit when false' */
  int ravi_type; /* RAVI change: type of the expression if known, else LUA_TNONE */
} expdesc;
```

The code is somewhat hard to follow as the `expdesc` objects go through various states and are also reused when needed.

As the parser generates code while parsing it needs to go back and patch the generated instructions when it has more information. For example when a function call is parsed the parser assumes that only 1 value is expected to be returned - but later this is patched when more information is available. The most common example is when the register where the value will be stored (operand A) is not known - in this case the parser later on updates this operand in the instruction. I believe jump statements have similar mechanics - however I have not yet gone through the details of these instructions.

7.2.1 Handling of Stack during parsing

Functions have a register window on the stack. The stack is represented in `LexState->dyd.actvar` (Dyndata) structure (see llex.h). The register window of the function starts from `LexState->dyd.actvar.arr[firstlocal]`.

The 'active' local variables of the function extend up to `LexState->dyd.actvar.arr[nactvar-1]`. Note that when parsing a `local` declaration statement the `nactvar` is adjusted at the end of the statement so that during parsing of the statement the `nactvar` covers locals up to the start of the statement. This means that local variables come into scope (become 'active') after the local statement ends. However, if the local statement defines a function then the variable becomes 'active' before the function body is parsed.

A tricky thing to note is that while `nactvar` is adjusted at the end of the statement - the 'stack' as represented by `LexState->dyd.actvar.arr` is extended to the required size as the local variables are created by `new_localvar()`.

When a function is the topmost function being parsed, the registers between `LexState->dyd.actvar.arr[nactvar]` and `LexState->dyd.actvar.arr[freereg-1]` are used by the parser for evaluating expressions - i.e. these are part of the local registers available to the function

Note that function parameters are handled as locals.

Example of what all this mean. Let's say we are parsing following chunk of code:

```
function testfunc()
  -- at this stage 'nactvar' is 0 (no active variables)
  -- 'firstlocal' is set to current top of the variables stack
  -- LexState->dyd.actvar.n (i.e. excluding registers used for expression evaluation)
  -- LexState->dyd.actvar.n = 0 at this stage
  local function tryme()
    -- Since we are inside the local statement and 'tryme' is a local variable,
    -- the LexState->dyd.actvar.n goes to 1. As this is a function definition
    -- the local variable declaration is deemed to end here, so 'nactvar' for␣
↪testfunc()
    -- is gets set to 1 (making 'tryme' an active variable).
    -- A new FuncState is created for 'tryme' function.
    -- The new tryme() FunState has 'firstlocal' set to value of LexState->dyd.actvar.
↪n, i.e., 1
```

(continues on next page)

(continued from previous page)

```
    local i,j = 5,6
    -- After 'i' is parsed, LexState->dyd.actvar.n = 2, but 'nactvar' = 0 for tryme()
    -- After 'j' is parsed, LexState->dyd.actvar.n = 3, but 'nactvar' = 0 for tryme()
    -- Only after the full statement above is parsed, 'nactvar' for tryme() is set to
↪'2'
    -- This is done by adjustlocalvar().
    return i,j
  end
  -- Here two things happen
  -- Firstly the FuncState for tryme() is popped so that
  -- FuncState for testfunc() is now at top
  -- As part of this popping, leaveblock() calls removevars()
  -- to adjust the LexState->dyd.actvar.n down to 1 where it was
  -- at before parsing the tryme() function body.
  local i, j = tryme()
  -- After 'i' is parsed, LexState->dyd.actvar.n = 2, but 'nactvar' = 1 still
  -- After 'j' is parsed, LexState->dyd.actvar.n = 3, but 'nactvar' = 1 still
  -- At the end of the statement 'nactvar' is set to 3.
  return i+j
end
-- As before the leaveblock() calls removevars() which resets
-- LexState->dyd.actvar.n to 0 (the value before testfunc() was parsed)
```

A rough debug trace of the above gives:

```
function testfunc()
  -- open_func -> fs->firstlocal set to 0 (ls->dyd->actvar.n), and fs->nactvar reset␣
↪to 0
  local function tryme()
    -- new_localvar -> registering var tryme fs->f->locvars[0] at ls->dyd->actvar.
↪arr[0]
    -- new_localvar -> ls->dyd->actvar.n set to 1
    -- adjustlocalvars -> set fs->nactvar to 1
    -- open_func -> fs->firstlocal set to 1 (ls->dyd->actvar.n), and fs->nactvar␣
↪reset to 0
    -- adjustlocalvars -> set fs->nactvar to 0 (no parameters)
    local i,j = 5,6
    -- new_localvar -> registering var i fs->f->locvars[0] at ls->dyd->actvar.arr[1]
    -- new_localvar -> ls->dyd->actvar.n set to 2
    -- new_localvar -> registering var j fs->f->locvars[1] at ls->dyd->actvar.arr[2]
    -- new_localvar -> ls->dyd->actvar.n set to 3
    -- adjustlocalvars -> set fs->nactvar to 2
    return i,j
    -- removevars -> reset fs->nactvar to 0
  end
  local i, j = tryme()
  -- new_localvar -> registering var i fs->f->locvars[1] at ls->dyd->actvar.arr[1]
  -- new_localvar -> ls->dyd->actvar.n set to 2
  -- new_localvar -> registering var j fs->f->locvars[2] at ls->dyd->actvar.arr[2]
  -- new_localvar -> ls->dyd->actvar.n set to 3
  -- adjustlocalvars -> set fs->nactvar to 3
  return i+j
  -- removevars -> reset fs->nactvar to 0
end
```

7.2.2 Notes on Parser by Sven Olsen

"discharging" expressions

"discharging" takes an expression of arbitrary type, and converts it to one having particular properties.

the lowest-level discharge function is `discharge2vars ()`, which converts an expression into one of the two "result" types; either a `VNONRELOC` or a `VRELOCABLE`.

if the variable in question is a `VLOCAL`, `discharge2vars` will simply change the stored type to `VNONRELOC`.

much of lcode.c assumes that the it will be working with discharged expressions. in particular, it assumes that if it encounters a `VNONRELOC` expression, and `e->info < nactvar`, then the register referenced is a local, and therefore shouldn't be implicitly freed after use.

local variables

however, the relationship between `nactvar` and locals is actually somewhat more complex – as each local variable appearing in the code has a collection of data attached to it, data that's being accumulated and changed as the lexer moves through the source.

`fs->nlocvars` stores the total number of named locals inside the function – recall that different local variables are allowed to overlap the same register, depending on which are in-scope at any particular time.

the list of locals that are active at any given time is stored in `ls->dyd` – a vector of stack references that grows or shrinks as locals enter or leave scope.

managing the lifetime of local variables involves several steps. first, new locals are declared using `new_localvar`. this sets their names and creates new references in `dyd`. soon thereafter, the parser is expected to call `adjustlocalvar(ls,nvars)`, with `nvars` set to the number of new locals. `adjustlocalvar` increments `fs->nactvar` by `nvars`, and marks the startpc's of all the locals.

note that neither `new_localvar` or `adjustlocalvar` ensures that anything is actually inside the registers being labeled as locals. failing to initialize said registers is an easy way to write memory access bugs (peter's original table unpack patch includes one such).

after `adjustlocalvar` is called, `luaK_exp2nextreg()` will no longer place new data inside the local's registers – as they're no longer part of the temporary register stack.

when the time comes to deactivate locals, that's done via `removevars(tolevel)`. `tolevel` is assumed to contain `nactvars` as it existed prior to entering the previous block. thus, the number of locals to remove should simply be `fs->nactvar-tolevel`. `removevars(tolevel)` will decrement `nactvars` down to `tolevel`. it also shrinks the `dyd` vector, and marks the endpc's of all the removed locals.

except in between `new_localvar` and `adjustlocalvar` calls, i believe that:

```
fs->ls->dyd->actvar.n - fs->firstlocal == fs->nactvar
```

temporary registers

`freereg` is used to manage the temporary register stack – registers between [`fs->nactvars,fs->freereg`) are assumed to belong to expressions currently being stored by the parser.

`fs->freereg` is incremented explicitly by calls to `luaK_reserveregs`, or implicitly, inside `luaK_exp2nextreg`. it's decremented whenever a `freereg(r)` is called on a register in the temporary stack (i.e., a register for which `r >= fs->nactvar`).

the temporary register stack is cleared when `leaveblock()` is called, by setting `fs->freereg=fs->nactvar`. it's also partially cleared in other places – for example, inside the evaluation of table constructors.

note that `freereg` just pops the top of the stack if r does not appear to be a local – thus it doesn't necessarily, free r. one of the important sanity checks that you'll get by enabling `lua_assert()` checks that the register being freed is also the top of the stack.

when writing parser patches, it's your job to ensure that the registers that you've reserved are freed in an appropriate order.

when a `VINDEXED` expression is discharged, `freereg()` will be called on both the table and the index register. otherwise, `freereg` is only called from `freeexp()` – which gets triggered anytime an expression has been "used up"; typically, anytime it's been transformed into another expression.

7.2.3 State Transitions

The state transitions for `expdesc` structure are as follows:

exp-kind	Description	State Transitions
VVOID	This is used to indicate the lack of value - e.g. function call with no arguments, the rhs of local variable declaration, and empty table constructor	None
VRELOCABLE	This is used to indicate that the result from expression needs to be set to a register. The operation that created the expression is referenced by the `u.info` parameter which contains an offset into the `code` of the function that is being compiled So you can access this instruction by calling `getcode(FuncState *, expdesc *)` The operations that result in a `VRELOCABLE` object include `OP_CLOSURE` `OP_NEWTABLE` `OP_GETUPVAL` `OP_GETTABUP` `OP_GETTABLE` `OP_NOT` and code for binary and unary expressions that produce values (arithmetic operations, bitwise operations, concat, length). The associated code instruction has operand `A` unset (defaulted to 0) - this the `VRELOCABLE` expression must be later transitioned to `VNONRELOC` state when the register is set.	In terms of transitions the following expression kinds convert to `VRELOCABLE`: `VVARARG` `VUPVAL` (`OP_GETUPVAL` `VINDEXED` (`OP_GETTABUP` or `OP_GETTABLE` And following expression states can result from a `VRELOCABLE` expression: `VNONRELOC` which means that the result register in the instruction operand A has been set.
VNONRELOC	This state indicates that the output or result register has been set. The register is referenced in `u.info` parameter. Once set the register cannot be changed for this expression; subsequent operations involving this expression can refer to the register to obtain the result value.	As for transitions, the `VNONELOC` state results from `VRELOCABLE` after a register is assigned to the operation referenced by `VRELOCABLE`. Also a `VCALL` expression transitions to `VNONRELOC` expression - `u.info` is set to the operand `A` in the call instruction. `VLOCAL` `VNIL` `VTRUE` `VFALSE` `VK` `VKINT` `VKFLT` and `VJMP` expressions transition to `VNONRELOC`.
VLOCAL	This is used when referencing local variables. `u.info` is set to the local variable's register.	The `VLOCAL` expression may transition to `VNONRELOC` although this doesn't change the `u.info` parameter.
VCALL	This results from a function call. The `OP_CALL` instruction is referenced by `u.info` parameter and may be retrieved by calling `getcode(FuncState *, expdesc *)`. The `OP_CALL` instruction gets changed to `OP_TAILCALL` if the function call expression is the value of a `RETURN` statement. The instructions operand `C` gets updated when it is known the number of expected results from the function call.	In terms of transitions, the `VCALL` expression transitions to `VNONRELOC` When this happens the result register in `VNONRELOC` (`u.info` is set to the operand `A` in the `OP_CALL` instruction.
VINDEXED	This expression represents a table access. The `u.ind.t` parameter is set to the register or upvalue? that holds the table, the `u.ind.idx` is set to the register or constant that is the key, and `u.ind.vt` is either `VLOCAL` or `VUPVAL`	The `VINDEXED` expression transitions to `VRELOCABLE` When this happens the `u.info` is set to the offset of the code that contains the opcode `OP_GETTABUP` if `u.ind.vt` was `VUPVAL` or `OP_GETTABLE` if `u.ind.vt` was `VLOCAL`

7.2.4 Examples of Parsing

example 1

We investigate the simple code chunk below:

```
local i,j; j = i*j+i
```

The compiler allocates following local registers, constants and upvalues:

```
constants (0) for 0000007428FED950:
locals (2) for 0000007428FED950:
        0       i       2       5
        1       j       2       5
upvalues (1) for 0000007428FED950:
        0       _ENV    1       0
```

Some of the parse steps are highlighted below.

Reference to variable `i` which is located in register `0`. The `p` here is the pointer address of `expdesc` object so you can see how the same object evolves:

```
{p=0000007428E1F170, k=VLOCAL, register=0}
```

Reference to variable `j` located in register `1`:

```
{p=0000007428E1F078, k=VLOCAL, register=1}
```

Now the MUL operator is applied so we get following. Note that the previously `VLOCAL` expression for `i` is now `VNONRELOC`:

```
{p=0000007428E1F170, k=VNONRELOC, register=0} MUL {p=0000007428E1F078, k=VLOCAL,␣
↪register=1}
```

Next code gets generated for the `MUL` operator and we can see that first expression is replaced by a `VRELOCABLE` expression. Note also that the `MUL` operator is encoded in the `VRELOCABLE` expression as instruction `1` which is decoded below:

```
{p=0000007428E1F170, k=VRELOCABLE, pc=1, instruction=(MUL A=0 B=0 C=1)}
```

Now a reference to `i` is again required:

```
{p=0000007428E1F078, k=VLOCAL, register=0}
```

And the `ADD` operator must be applied to the result of the `MUL` operator and above. Notice that a temporary register `2` has been allocated to hold the result of the `MUL` operator, and also notice that as a result the `VRELOCABLE` has now changed to `VNONRELOC`:

```
{p=0000007428E1F170, k=VNONRELOC, register=2} ADD {p=0000007428E1F078, k=VLOCAL,␣
↪register=0}
```

Next the result of the `ADD` expression gets encoded similarly to `MUL` earlier. As this is a `VRELOCABLE` expression it will be later on assigned a result register:

```
{p=0000007428E1F170, k=VRELOCABLE, pc=2, instruction=(ADD A=0 B=2 C=0)}
```

Eventually above gets assigned a result register and becomes `VNONRELOC` (not shown here) - and so the final generated code looks like below:

```
main <(string):0,0> (4 instructions at 0000007428FED950)
0+ params, 3 slots, 1 upvalue, 2 locals, 0 constants, 0 functions
        1       [1]     LOADNIL         0 1
        2       [1]     MUL             2 0 1
        3       [1]     ADD             1 2 0
        4       [1]     RETURN          0 1
```

7.3 Links

- (MP1) Lua Code Reading Order
- (RL1) Registers allocation and GC
- (MP2) LuaJIT interpreter optmisations
- (MP3) Performance of Switch Based Dispatch
- (MP4) Challenges for static compilation of dynamic langauges
- (MP5) VM Internals (bytecode format)
- (RL2) Upvalues in closures
- (LHF) Lua bytecode dump format
- (MP6) Register VM and sliding stack window
- (SO1) Sven Olsen's notes on registers from Sven Olsen's Lua Users Wiki page
- (KHM) No Frills Introduction to Lua 5.1 VM Instructions
- (MP7) LuaJIT Roadmap 2008
- (MP8) LuaJIT Roadmap 2011

CHAPTER 8

Ravi Parsing and ByteCode Implementation Details

This document covers the enhancements to the Lua parser and byte-code generator. The Ravi JIT implementation is described elsewhere.

8.1 Introduction

Since the reason for introducing optional static typing is to enhance performance primarily - not all types benefit from this capability. In fact it is quite hard to extend this to generic recursive structures such as tables without encurring significant overhead. For instance - even to represent a recursive type in the parser will require dynamic memory allocation and add great overhead to the parser.

From a performance point of view the only types that seem worth specializing are:

- integer (64-bit int)
- number (double)
- array of integers
- array of numbers
- table

8.2 Implementation Strategy

I want to build on existing Lua types rather than introducing completely new types to the Lua system. I quite like the minimalist nature of Lua. However, to make the execution efficient I am adding new type specific opcodes and enhancing the Lua parser/code generator to encode these opcodes only when types are known. The new opcodes will execute more efficiently as they will not need to perform type checks. Morever, type specific instructions will lend themselves to more efficient JIT compilation.

I am adding new opcodes that cover arithmetic operations, array operations, variable assignments, etc..

8.3 Modifications to Lua Bytecode structure

An immediate issue is that the Lua bytecode structure has a 6-bit opcode which is insufficient to hold the various opcodes that I will need. Simply extending the size of this is problematic as then it reduces the space available to the operands A B and C. Furthermore the way Lua bytecodes work means that B and C operands must be 1-bit larger than A - as the extra bit is used to flag whether the operand refers to a constant or a register. (Thanks to Dirk Laurie for pointing this out).

I am amending the bit mapping in the 32-bit instruction to allow 9-bits for the byte-code, 7-bits for operand A, and 8-bits for operands B and C. This means that some of the Lua limits (maximum number of variables in a function, etc.) have to be revised to be lower than the default.

8.4 New OpCodes

The new instructions are specialised for types, and also for register/versus constant. So for example `OP_RAVI_ADDFI` means add `number` and `integer`. And `OP_RAVI_ADDFF` means add `number` and `number`. The existing Lua opcodes that these are based on define which operands are used.

Example:

```
local i=0; i=i+1
```

Above standard Lua code compiles to:

```
[0] LOADK A=0 Bx=-1
[1] ADD A=0 B=0 C=-2
[2] RETURN A=0 B=1
```

We add type info using Ravi extensions:

```
local i:integer=0; i=i+1
```

Now the code compiles to:

```
[0] LOADK A=0 Bx=-1
[1] ADDII A=0 B=0 C=-2
[2] RETURN A=0 B=1
```

Above uses type specialised opcode `OP_RAVI_ADDII`.

8.5 Type Information

The basic first step is to add type information to Lua.

As the parser progresses it creates a vector of `LocVar` for each function containing a list of local variables. I have enhanced `LocVar` structure in `lobject.h` to hold type information.

```
/* Following are the types we will use
** use in parsing. The rationale for types is
** performance - as of now these are the only types that
** we care about from a performance point of view - if any
** other types appear then they are all treated as ANY
**/
```

(continues on next page)

(continued from previous page)

```
typedef enum {
  RAVI_TANY = -1,       /* Lua dynamic type */
  RAVI_TNUMINT,    /* integer number */
  RAVI_TNUMFLT,    /* floating point number */
  RAVI_TARRAYINT, /* array of ints */
  RAVI_TARRAYFLT,  /* array of doubles */
  RAVI_TFUNCTION,
  RAVI_TTABLE,
  RAVI_TSTRING,
  RAVI_TNIL,
  RAVI_TBOOLEAN
} ravitype_t;

/*
** Description of a local variable for function prototypes
** (used for debug information)
*/
typedef struct LocVar {
  TString *varname;
  int startpc;  /* first point where variable is active */
  int endpc;    /* first point where variable is dead */
  ravitype_t ravi_type; /* RAVI type of the variable - RAVI_TANY if unknown */
} LocVar;
```

The `expdesc` structure is used by the parser to hold nodes in the expression tree. I have enhanced the `expdesc` structure to hold the type of an expression.

```
typedef struct expdesc {
  expkind k;
  union {
    struct {  /* for indexed variables (VINDEXED) */
      short idx;  /* index (R/K) */
      lu_byte t;  /* table (register or upvalue) */
      lu_byte vt;  /* whether 't' is register (VLOCAL) or upvalue (VUPVAL) */
      ravitype_t key_type; /* key type */
    } ind;
    int info;  /* for generic use */
    lua_Number nval;  /* for VKFLT */
    lua_Integer ival;    /* for VKINT */
  } u;
  int t;  /* patch list of 'exit when true' */
  int f;  /* patch list of 'exit when false' */
  ravitype_t ravi_type; /* RAVI change: type of the expression if known, else RAVI_
↪TANY */
} expdesc;
```

Note the addition of type information in two places. Firstly at the `expdesc` level which identifies the type of the `expdesc`. Secondly in the *ind* structure - the `key_type` is used to track the type of the key that will be used to index into a table.

The table structure has been enhanced to hold additional information for array usage.

```
typedef enum RaviArrayModifer {
  RAVI_ARRAY_SLICE = 1,
  RAVI_ARRAY_FIXEDSIZE = 2
} RaviArrayModifier;

```

(continues on next page)

(continued from previous page)

```
typedef struct RaviArray {
  char *data;
  unsigned int len; /* RAVI len specialization */
  unsigned int size; /* amount of memory allocated */
  lu_byte array_type; /* RAVI specialization */
  lu_byte array_modifier; /* Flags that affect how the array is handled */
} RaviArray;

typedef struct Table {
  CommonHeader;
  lu_byte flags;  /* 1<<p means tagmethod(p) is not present */
  lu_byte lsizenode;  /* log2 of size of 'node' array */
  unsigned int sizearray;  /* size of 'array' array */
  TValue *array;  /* array part */
  Node *node;
  Node *lastfree;  /* any free position is before this position */
  struct Table *metatable;
  GCObject *gclist;
  RaviArray ravi_array;
} Table;
```

8.6 Parser Enhancements

The parser needs to be enhanced to generate type specific instructions at various points.

8.6.1 Local Variable Declarations

First enhancement needed is when local variable declarations are parsed. We need to allow the type to be defined for each variable and ensure that any assignments are type-checked. This is somewhat complex process, due to the fact that assignments can be expressions involving function calls. The last function call is treated as a variable assignment - i.e. all trailing variables are assumed to be assigned values from the function call - if not the variables are set to nil by default.

The entry point for parsing a local statement is `localstat()` in `lparser.c`. This function has been enhanced to parse the type annotations supported by Ravi. The modified function is shown below.

```
/* Parse
 *   name : type
 *   where type is 'integer', 'integer[]',
 *                 'number', 'number[]'
 */
static ravitype_t declare_localvar(LexState *ls) {
  /* RAVI change - add type */
  TString *name = str_checkname(ls);
  /* assume a dynamic type */
  ravitype_t tt = RAVI_TANY;
  /* if the variable name is followed by a colon then we have a type
   * specifier
   */
  if (testnext(ls, ':')) {
    TString *typename = str_checkname(ls); /* we expect a type name */
    const char *str = getaddrstr(typename);
    /* following is not very nice but easy as
```

(continues on next page)

(continued from previous page)

```
     * the lexer doesn't need to be changed
     */
    if (strcmp(str, "integer") == 0)
      tt = RAVI_TNUMINT;
    else if (strcmp(str, "number") == 0)
      tt = RAVI_TNUMFLT;
    if (tt == RAVI_TNUMFLT || tt == RAVI_TNUMINT) {
      /* if we see [] then it is an array type */
      if (testnext(ls, '[')) {
        checknext(ls, ']');
        tt = (tt == RAVI_TNUMFLT) ? RAVI_TARRAYFLT : RAVI_TARRAYINT;
      }
    }
  }
  new_localvar(ls, name, tt);
  return tt;
}

/* parse a local variable declaration statement - called from statement() */
static void localstat (LexState *ls) {
  /* stat -> LOCAL NAME {',' NAME} ['=' explist] */
  int nvars = 0;
  int nexps;
  expdesc e;
  e.ravi_type = RAVI_TANY;
  /* RAVI while declaring locals we need to gather the types
   * so that we can check any assignments later on.
   * TODO we may be able to use register_typeinfo() here
   * instead.
   */
  int vars[MAXVARS] = { 0 };
  do {
    /* RAVI changes start */
    /* local name : type = value */
    vars[nvars] = declare_localvar(ls);
    /* RAVI changes end */
    nvars++;
  } while (testnext(ls, ','));
  if (testnext(ls, '='))
    nexps = localvar_explist(ls, &e, vars, nvars);
  else {
    e.k = VVOID;
    nexps = 0;
  }
  localvar_adjust_assign(ls, nvars, nexps, &e);
  adjustlocalvars(ls, nvars);
}
```

The do-while loop is responsible for parsing the variable names and the type annotations. As each variable name is parsed we detect if there is a type annotation, if and if present the type is recorded in the array `vars`.

Parameter lists may have static type annotations as well, so when parsing parameters we again need to invoke `declare_localvar()`.

```
static void parlist (LexState *ls) {
  /* parlist -> [ param { ',' param } ] */
  FuncState *fs = ls->fs;
```

(continues on next page)

(continued from previous page)

```
  Proto *f = fs->f;
  int nparams = 0;
  f->is_vararg = 0;
  if (ls->t.token != ')') {  /* is 'parlist' not empty? */
    do {
      switch (ls->t.token) {
        case TK_NAME: {  /* param -> NAME */
          /* RAVI change - add type */
          declare_localvar(ls);
          nparams++;
          break;
        }
        case TK_DOTS: {  /* param -> '...' */
          luaX_next(ls);
          f->is_vararg = 1;
          break;
        }
        default: luaX_syntaxerror(ls, "<name> or '...' expected");
      }
    } while (!f->is_vararg && testnext(ls, ','));
  }
  adjustlocalvars(ls, nparams);
  f->numparams = cast_byte(fs->nactvar);
  luaK_reserveregs(fs, fs->nactvar);  /* reserve register for parameters */
  for (int i = 0; i < f->numparams; i++) {
    ravitype_t tt = raviY_get_register_typeinfo(fs, i);
    DEBUG_VARS(raviY_printf(fs, "Parameter [%d] = %v\n", i + 1, getlocvar(fs, i)));
    /* do we need to convert ? */
    if (tt == RAVI_TNUMFLT || tt == RAVI_TNUMINT) {
      /* code an instruction to convert in place */
      luaK_codeABC(ls->fs, tt == RAVI_TNUMFLT ? OP_RAVI_TOFLT : OP_RAVI_TOINT, i, 0,␣
↪0);
    }
    else if (tt == RAVI_TARRAYFLT || tt == RAVI_TARRAYINT) {
      /* code an instruction to convert in place */
      luaK_codeABC(ls->fs, tt == RAVI_TARRAYFLT ? OP_RAVI_TOFARRAY : OP_RAVI_TOIARRAY,
↪ i, 0, 0);
    }
  }
}
```

Additionally for parameters that are decorated with static types we need to introduce new instructions to coerce the types at run time. That is what is happening in the for loop at the end.

The `declare_localvar()` function passes the type of the variable to `new_localvar()` which records this in the `LocVar` structure associated with the variable.

```
static int registerlocalvar (LexState *ls, TString *varname, int ravi_type) {
  FuncState *fs = ls->fs;
  Proto *f = fs->f;
  int oldsize = f->sizelocvars;
  luaM_growvector(ls->L, f->locvars, fs->nlocvars, f->sizelocvars,
                  LocVar, SHRT_MAX, "local variables");
  while (oldsize < f->sizelocvars) {
    /* RAVI change initialize */
    f->locvars[oldsize].startpc = -1;
    f->locvars[oldsize].endpc = -1;
```

(continues on next page)

(continued from previous page)

```
    f->locvars[oldsize].ravi_type = RAVI_TANY;
    f->locvars[oldsize++].varname = NULL;
  }
  f->locvars[fs->nlocvars].varname = varname;
  f->locvars[fs->nlocvars].ravi_type = ravi_type;
  luaC_objbarrier(ls->L, f, varname);
  return fs->nlocvars++;
}

/* create a new local variable in function scope, and set the
 * variable type (RAVI - added type tt) */
static void new_localvar (LexState *ls, TString *name, ravitype_t tt) {
  FuncState *fs = ls->fs;
  Dyndata *dyd = ls->dyd;
  /* register variable and get its index */
  /* RAVI change - record type info for local variable */
  int i = registerlocalvar(ls, name, tt);
  checklimit(fs, dyd->actvar.n + 1 - fs->firstlocal,
                  MAXVARS, "local variables");
  luaM_growvector(ls->L, dyd->actvar.arr, dyd->actvar.n + 1,
                  dyd->actvar.size, Vardesc, MAX_INT, "local variables");
  /* variable will be placed at stack position dyd->actvar.n */
  dyd->actvar.arr[dyd->actvar.n].idx = cast(short, i);
  DEBUG_VARS(raviY_printf(fs, "new_localvar -> registering %v fs->f->locvars[%d] at␣
↪ls->dyd->actvar.arr[%d]\n", &fs->f->locvars[i], i, dyd->actvar.n));
  dyd->actvar.n++;
  DEBUG_VARS(raviY_printf(fs, "new_localvar -> ls->dyd->actvar.n set to %d\n", dyd->
↪actvar.n));
}
```

The next bit of change is how the expressions are handled following the = symbol. The previously built `vars` array is passed to a modified version of `explist()` called `localvar_explist()`. This handles the parsing of expressions and then ensuring that each expression matches the type of the variable where known. The `localvar_explist()` function is shown next.

```
static int localvar_explist(LexState *ls, expdesc *v, int *vars, int nvars) {
  /* explist -> expr { ',' expr } */
  int n = 1;  /* at least one expression */
  expr(ls, v);
#if RAVI_ENABLED
  ravi_typecheck(ls, v, vars, nvars, 0);
#endif
  while (testnext(ls, ',')) {
    luaK_exp2nextreg(ls->fs, v);
    expr(ls, v);
#if RAVI_ENABLED
    ravi_typecheck(ls, v, vars, nvars, n);
#endif
    n++;
  }
  return n;
}
```

The main changes compared to `explist()` are the calls to `ravi_typecheck()`. Note that the array `vars` is passed to the `ravi_typecheck()` function along with the current variable index in `n`. The `ravi_typecheck()` function is reproduced below.

```
static void ravi_typecheck(LexState *ls, expdesc *v, int *vars, int nvars, int n)
{
  if (n < nvars && vars[n] != RAVI_TANY && v->ravi_type != vars[n]) {
    if (v->ravi_type != vars[n] &&
        (vars[n] == RAVI_TARRAYFLT || vars[n] == RAVI_TARRAYINT) &&
        v->k == VNONRELOC) {
      /* as the bytecode for generating a table is already
       * emitted by this stage we have to amend the generated byte code
       * - not sure if there is a better approach.
       * We look for the last bytecode that is OP_NEWTABLE
       * and that has the same destination
       * register as v->u.info which is our variable
       * local a:integer[] = { 1 }
       *                     ^ We are just past this and
       *                       about to assign to a
       */
      int i = ls->fs->pc - 1;
      for (; i >= 0; i--) {
        Instruction *pc = &ls->fs->f->code[i];
        OpCode op = GET_OPCODE(*pc);
        int reg;
        if (op != OP_NEWTABLE)
          continue;
        reg = GETARG_A(*pc);
        if (reg != v->u.info)
          continue;
        op = (vars[n] == RAVI_TARRAYINT) ? OP_RAVI_NEWARRAYI : OP_RAVI_NEWARRAYF;
        SET_OPCODE(*pc, op); /* modify opcode */
        DEBUG_CODEGEN(raviY_printf(ls->fs, "[%d]* %o ; modify opcode\n", i, *pc));
        break;
      }
      if (i < 0)
        luaX_syntaxerror(ls, "expecting array initializer");
    }
    /* if we are calling a function then convert return types */
    else if (v->ravi_type != vars[n] &&
            (vars[n] == RAVI_TNUMFLT || vars[n] == RAVI_TNUMINT) &&
            v->k == VCALL) {
      /* For local variable declarations that call functions e.g.
       * local i = func()
       * Lua ensures that the function returns values
       * to register assigned to variable i and above so that no
       * separate OP_MOVE instruction is necessary. So that means that
       * we need to coerce the return values in situ.
       */
      /* Obtain the instruction for OP_CALL */
      Instruction *pc = &getcode(ls->fs, v);
      lua_assert(GET_OPCODE(*pc) == OP_CALL);
      int a = GETARG_A(*pc); /* function return values
                                will be placed from register pointed
                                by A and upwards */
      int nrets = GETARG_C(*pc) - 1; /* operand C contains
                                        number of return values expected  */
      /* Note that at this stage nrets is always 1
       * - as Lua patches in the this value for the last
       * function call in a variable declaration statement
       * in adjust_assign and localvar_adjust_assign */
```

(continues on next page)

(continued from previous page)

```
      /* all return values that are going to be assigned
         to typed local vars must be converted to the correct type */
      int i;
      for (i = n; i < (n+nrets); i++)
        /* do we need to convert ? */
        if ((vars[i] == RAVI_TNUMFLT || vars[i] == RAVI_TNUMINT))
          /* code an instruction to convert in place */
          luaK_codeABC(ls->fs,
                       vars[i] == RAVI_TNUMFLT ?
                                  OP_RAVI_TOFLT : OP_RAVI_TOINT,
                       a+(i-n), 0, 0);
        else if ((vars[i] == RAVI_TARRAYFLT || vars[i] == RAVI_TARRAYINT))
          /* code an instruction to convert in place */
          luaK_codeABC(ls->fs,
                       vars[i] == RAVI_TARRAYFLT ?
                                  OP_RAVI_TOFARRAY : OP_RAVI_TOIARRAY,
                       a + (i - n), 0, 0);
    }
    else if ((vars[n] == RAVI_TNUMFLT || vars[n] == RAVI_TNUMINT) &&
             v->k == VINDEXED) {
      if (vars[n] == RAVI_TNUMFLT && v->ravi_type != RAVI_TARRAYFLT ||
        vars[n] == RAVI_TNUMINT && v->ravi_type != RAVI_TARRAYINT)
        luaX_syntaxerror(ls, "Invalid local assignment");
    }
    else
      luaX_syntaxerror(ls, "Invalid local assignment");
  }
}
```

There are several parts to this function.

The simple case is when the type of the expression matches the variable.

Secondly if the expression is a table initializer then we need to generate specialized opcodes if the target variable is supposed to be `integer[]` or `number[]`. The specialized opcode sets up some information in the `Table` structure. The problem is that this requires us to modify `OP_NEWTABLE` instruction which has already been emitted. So we scan the generated instructions to find the last `OP_NEWTABLE` instruction that assigns to the register associated with the target variable.

Next bit of special handling is for function calls. If the assignment makes a function call then we perform type coercion on return values where these values are being assigned to variables with defined types. This means that if the target variable is `integer` or `number` we issue opcodes `TOINT` and `TOFLT` respectively. If the target variable is `integer[]` or `number[]` then we issue `TOIARRAY` and `TOFARRAY` respectively. These opcodes ensure that the values are of required type or can be cast to the required type.

Note that any left over variables that are not assigned values, are set to 0 if they are of integer or number type, else they are set to nil as per Lua's default behavior. This is handled in `localvar_adjust_assign()` which is described later on.

Finally the last case is when the target variable is `integer` or `number` and the expression is a table / array access. In this case we check that the table is of required type.

The `localvar_adjust_assign()` function referred to above is shown below.

```
static void localvar_adjust_assign(LexState *ls, int nvars, int nexps, expdesc *e) {
  FuncState *fs = ls->fs;
  int extra = nvars - nexps;
  if (hasmultret(e->k)) {
```

(continues on next page)

(continued from previous page)

```
    extra++;  /* includes call itself */
    if (extra < 0) extra = 0;
    /* following adjusts the C operand in the OP_CALL instruction */
    luaK_setreturns(fs, e, extra);  /* last exp. provides the difference */
#if RAVI_ENABLED
    /* Since we did not know how many return values to process in localvar_explist()␣
↪we
    * need to add instructions for type coercions at this stage for any remaining
    * variables
    */
    ravi_coercetype(ls, e, extra);
#endif
    if (extra > 1) luaK_reserveregs(fs, extra - 1);
  }
  else {
    if (e->k != VVOID) luaK_exp2nextreg(fs, e);  /* close last expression */
    if (extra > 0) {
      int reg = fs->freereg;
      luaK_reserveregs(fs, extra);
      /* RAVI TODO for typed variables we should not set to nil? */
      luaK_nil(fs, reg, extra);
#if RAVI_ENABLED
      /* typed variables that are primitives cannot be set to nil so
       * we need to emit instructions to initialise them to default values
       */
      ravi_setzero(fs, reg, extra);
#endif
    }
  }
}
```

As mentioned before any variables left over in a local declaration that have not been assigned values must be set to default values appropriate for the type. In the case of trailing values returned by a function call we need to coerce the values to the required types. All this is done in the `localvar_adjust_assign()` function above.

Note that local declarations have a complication that until the declaration is complete the variable does not come in scope. So we have to be careful when we wish to map from a register to the local variable declaration as this mapping is only available after the variable is activated. Couple of helper routines are shown below.

```
/* translate from local register to local variable index
 */
static int register_to_locvar_index(FuncState *fs, int reg) {
  int idx;
  lua_assert(reg >= 0 && (fs->firstlocal + reg) < fs->ls->dyd->actvar.n);
  /* Get the LocVar associated with the register */
  idx = fs->ls->dyd->actvar.arr[fs->firstlocal + reg].idx;
  lua_assert(idx < fs->nlocvars);
  return idx;
}

/* get type of a register - if the register is not allocated
 * to an active local variable, then return RAVI_TANY else
 * return the type associated with the variable.
 * This is a RAVI function
 */
ravitype_t raviY_get_register_typeinfo(FuncState *fs, int reg) {
  int idx;
```

(continues on next page)

(continued from previous page)

```
  LocVar *v;
  if (reg < 0 || reg >= fs->nactvar || (fs->firstlocal + reg) >= fs->ls->dyd->actvar.
↪n)
    return RAVI_TANY;
  /* Get the LocVar associated with the register */
  idx = fs->ls->dyd->actvar.arr[fs->firstlocal + reg].idx;
  lua_assert(idx < fs->nlocvars);
  v = &fs->f->locvars[idx];
  /* Variable in scope so return the type if we know it */
  return v->ravi_type;
}
```

Note the use of `register_to_localvar_index()` in functions below.

```
/* Generate instructions for converting types
 * This is needed post a function call to handle
 * variable number of return values
 * n = number of return values to adjust
 */
static void ravi_coercetype(LexState *ls, expdesc *v, int n)
{
  if (v->k != VCALL || n <= 0) return;
  /* For local variable declarations that call functions e.g.
  * local i = func()
  * Lua ensures that the function returns values to register
  * assigned to variable and above so that no separate
  * OP_MOVE instruction is necessary. So that means that
  * we need to coerce the return values in situ.
  */
  /* Obtain the instruction for OP_CALL */
  Instruction *pc = &getcode(ls->fs, v);
  lua_assert(GET_OPCODE(*pc) == OP_CALL);
  int a = GETARG_A(*pc); /* function return values will be placed
                            from register pointed by A and upwards */
  /* all return values that are going to be assigned
   to typed local vars must be converted to the correct type */
  int i;
  for (i = a + 1; i < a + n; i++) {
    /* Since this is called when parsing local statements the
     * variable may not yet have a register assigned to it
     * so we can't use raviY_get_register_typeinfo()
     * here. Instead we need to check the variable definition - so we
     * first convert from local register to variable index.
     */
    int idx = register_to_locvar_index(ls->fs, i);
    /* get variable's type */
    ravitype_t ravi_type = ls->fs->f->locvars[idx].ravi_type;
    /* do we need to convert ? */
    if (ravi_type == RAVI_TNUMFLT || ravi_type == RAVI_TNUMINT)
      /* code an instruction to convert in place */
      luaK_codeABC(ls->fs, ravi_type == RAVI_TNUMFLT ?
                   OP_RAVI_TOFLT : OP_RAVI_TOINT, i, 0, 0);
    else if (ravi_type == RAVI_TARRAYINT || ravi_type == RAVI_TARRAYFLT)
      luaK_codeABC(ls->fs, ravi_type == RAVI_TARRAYINT ?
                   OP_RAVI_TOIARRAY : OP_RAVI_TOFARRAY, i, 0, 0);
  }
}
```

(continues on next page)

(continued from previous page)

```
static void ravi_setzero(FuncState *fs, int from, int n) {
  int last = from + n - 1;  /* last register to set nil */
  int i;
  for (i = from; i <= last; i++) {
    /* Since this is called when parsing local statements
     * the variable may not yet have a register assigned to
     * it so we can't use raviY_get_register_typeinfo()
     * here. Instead we need to check the variable definition - so we
     * first convert from local register to variable index.
     */
    int idx = register_to_locvar_index(fs, i);
    /* get variable's type */
    ravitype_t ravi_type = fs->f->locvars[idx].ravi_type;
    /* do we need to convert ? */
    if (ravi_type == RAVI_TNUMFLT || ravi_type == RAVI_TNUMINT)
      /* code an instruction to convert in place */
      luaK_codeABC(fs, ravi_type == RAVI_TNUMFLT ?
         OP_RAVI_LOADFZ : OP_RAVI_LOADIZ, i, 0, 0);
  }
}
```

8.6.2 Assignments

Assignment statements have to be enhanced to perform similar type checks as for local declarations. Fortunately he assignment goes through the function `luaK_storevar()` in `lcode.c`. A modified version of this is shown below.

```
void luaK_storevar (FuncState *fs, expdesc *var, expdesc *ex) {
  switch (var->k) {
    case VLOCAL: {
      check_valid_store(fs, var, ex);
      freeexp(fs, ex);
      exp2reg(fs, ex, var->u.info);
      return;
    }
    case VUPVAL: {
      int e = luaK_exp2anyreg(fs, ex);
      luaK_codeABC(fs, OP_SETUPVAL, e, var->u.info, 0);
      break;
    }
    case VINDEXED: {
      OpCode op = (var->u.ind.vt == VLOCAL) ?
                      OP_SETTABLE : OP_SETTABUP;
      if (op == OP_SETTABLE) {
        /* table value set - if array access then use specialized versions */
        if (var->ravi_type == RAVI_TARRAYFLT &&
            var->u.ind.key_type == RAVI_TNUMINT)
          op = OP_RAVI_FARRAY_SET;
        else if (var->ravi_type == RAVI_TARRAYINT &&
                 var->u.ind.key_type == RAVI_TNUMINT)
          op = OP_RAVI_IARRAY_SET;
      }
      int e = luaK_exp2RK(fs, ex);
      luaK_codeABC(fs, op, var->u.ind.t, var->u.ind.idx, e);
      break;
```

(continues on next page)

(continued from previous page)

```
    }
    default: {
      lua_assert(0);  /* invalid var kind to store */
      break;
    }
  }
  freeexp(fs, ex);
}
```

Firstly note the call to `check_valid_store()` for a local variable assignment. The `check_valid_store()` function validates that the assignment is compatible.

Secondly if the assignment is to an indexed variable, i.e., table, then we need to generate special opcodes for arrays.

8.6.3 MOVE opcodes

Any `MOVE` instructions must be modified so that if the target is register that hosts a variable of known type then we need to generate special instructions that do a type conversion during the move. This is handled in `discharge2reg()` function which is reproduced below.

```
static void discharge2reg (FuncState *fs, expdesc *e, int reg) {
  luaK_dischargevars(fs, e);
  switch (e->k) {
    case VNIL: {
      luaK_nil(fs, reg, 1);
      break;
    }
    case VFALSE: case VTRUE: {
      luaK_codeABC(fs, OP_LOADBOOL, reg, e->k == VTRUE, 0);
      break;
    }
    case VK: {
      luaK_codek(fs, reg, e->u.info);
      break;
    }
    case VKFLT: {
      luaK_codek(fs, reg, luaK_numberK(fs, e->u.nval));
      break;
    }
    case VKINT: {
      luaK_codek(fs, reg, luaK_intK(fs, e->u.ival));
      break;
    }
    case VRELOCABLE: {
      Instruction *pc = &getcode(fs, e);
      SETARG_A(*pc, reg);
      DEBUG_EXPR(raviY_printf(fs, "discharge2reg (VRELOCABLE set arg A) %e\n", e));
      DEBUG_CODEGEN(raviY_printf(fs, "[%d]* %o ; set A to %d\n", e->u.info, *pc,␣
↪reg));
      break;
    }
    case VNONRELOC: {
      if (reg != e->u.info) {
        /* code a MOVEI or MOVEF if the target register is a local typed variable */
        int ravi_type = raviY_get_register_typeinfo(fs, reg);
```

(continues on next page)

(continued from previous page)

```
        switch (ravi_type) {
        case RAVI_TNUMINT:
          luaK_codeABC(fs, OP_RAVI_MOVEI, reg, e->u.info, 0);
          break;
        case RAVI_TNUMFLT:
          luaK_codeABC(fs, OP_RAVI_MOVEF, reg, e->u.info, 0);
          break;
        case RAVI_TARRAYINT:
          luaK_codeABC(fs, OP_RAVI_MOVEIARRAY, reg, e->u.info, 0);
          break;
        case RAVI_TARRAYFLT:
          luaK_codeABC(fs, OP_RAVI_MOVEFARRAY, reg, e->u.info, 0);
          break;
        default:
          luaK_codeABC(fs, OP_MOVE, reg, e->u.info, 0);
          break;
        }
      }
      break;
    }
    default: {
      lua_assert(e->k == VVOID || e->k == VJMP);
      return;  /* nothing to do... */
    }
  }
  e->u.info = reg;
  e->k = VNONRELOC;
}
```

Note the handling of `VNONRELOC` case.

8.6.4 Expression Parsing

The expression evaluation process must be modified so that type information is retained and flows through as the parser evaluates the expression. This involves ensuring that the type information is passed through as the parser modifies, reuses, creates new `expdesc` objects. Essentially this means keeping the `ravi_type` correct.

Additionally when arithmetic operations take place two things need to happen: a) specialized opcodes need to be emitted and b) the type of the resulting expression needs to be set.

```
static void codeexpval (FuncState *fs, OpCode op,
                        expdesc *e1, expdesc *e2, int line) {
  lua_assert(op >= OP_ADD);
  if (op <= OP_BNOT && constfolding(fs, getarithop(op), e1, e2))
    return;  /* result has been folded */
  else {
    int o1, o2;
    int isbinary = 1;
    /* move operands to registers (if needed) */
    if (op == OP_UNM || op == OP_BNOT || op == OP_LEN) {  /* unary op? */
      o2 = 0;  /* no second expression */
      o1 = luaK_exp2anyreg(fs, e1);  /* cannot operate on constants */
      isbinary = 0;
    }
    else {  /* regular case (binary operators) */
```

(continues on next page)

(continued from previous page)

```
      o2 = luaK_exp2RK(fs, e2);  /* both operands are "RK" */
      o1 = luaK_exp2RK(fs, e1);
    }
    if (o1 > o2) {  /* free registers in proper order */
      freeexp(fs, e1);
      freeexp(fs, e2);
    }
    else {
      freeexp(fs, e2);
      freeexp(fs, e1);
    }
#if RAVI_ENABLED
    if (op == OP_ADD &&
      (e1->ravi_type == RAVI_TNUMFLT || e1->ravi_type == RAVI_TNUMINT) &&
      (e2->ravi_type == RAVI_TNUMFLT || e2->ravi_type == RAVI_TNUMINT))
      generate_binarithop(fs, e1, e2, o1, o2, 0);
    else if (op == OP_MUL &&
      (e1->ravi_type == RAVI_TNUMFLT || e1->ravi_type == RAVI_TNUMINT) &&
      (e2->ravi_type == RAVI_TNUMFLT || e2->ravi_type == RAVI_TNUMINT))
      generate_binarithop(fs, e1, e2, o1, o2, OP_RAVI_MULFF - OP_RAVI_ADDFF);

    /* todo optimize the SUB opcodes when constant is small */
    else if (op == OP_SUB &&
             e1->ravi_type == RAVI_TNUMFLT &&
             e2->ravi_type == RAVI_TNUMFLT) {
      e1->u.info = luaK_codeABC(fs, OP_RAVI_SUBFF, 0, o1, o2);
    }
    else if (op == OP_SUB &&
             e1->ravi_type == RAVI_TNUMFLT &&
             e2->ravi_type == RAVI_TNUMINT) {
      e1->u.info = luaK_codeABC(fs, OP_RAVI_SUBFI, 0, o1, o2);
    }
    /* code omitted here  .... */
    else {
#endif
      e1->u.info = luaK_codeABC(fs, op, 0, o1, o2);  /* generate opcode */
#if RAVI_ENABLED
    }
#endif
    e1->k = VRELOCABLE;  /* all those operations are relocable */
    if (isbinary) {
      if ((op == OP_ADD || op == OP_SUB || op == OP_MUL || op == OP_DIV)
        && e1->ravi_type == RAVI_TNUMFLT && e2->ravi_type == RAVI_TNUMFLT)
        e1->ravi_type = RAVI_TNUMFLT;
      else if ((op == OP_ADD || op == OP_SUB || op == OP_MUL || op == OP_DIV)
        && e1->ravi_type == RAVI_TNUMFLT && e2->ravi_type == RAVI_TNUMINT)
        e1->ravi_type = RAVI_TNUMFLT;
      else if ((op == OP_ADD || op == OP_SUB || op == OP_MUL || op == OP_DIV)
        && e1->ravi_type == RAVI_TNUMINT && e2->ravi_type == RAVI_TNUMFLT)
        e1->ravi_type = RAVI_TNUMFLT;
      else if ((op == OP_ADD || op == OP_SUB || op == OP_MUL)
        && e1->ravi_type == RAVI_TNUMINT && e2->ravi_type == RAVI_TNUMINT)
        e1->ravi_type = RAVI_TNUMINT;
      else if ((op == OP_DIV)
        && e1->ravi_type == RAVI_TNUMINT && e2->ravi_type == RAVI_TNUMINT)
        e1->ravi_type = RAVI_TNUMFLT;
      else
```

(continues on next page)

(continued from previous page)

```
        e1->ravi_type = RAVI_TANY;
    }
    else {
      if (op == OP_LEN || op == OP_BNOT)
        e1->ravi_type = RAVI_TNUMINT;
    }
    luaK_fixline(fs, line);
  }
}
```

When expression reference indexed variables, i.e., tables, we need to emit specialized opcodes if the table is an array. This is done in `luaK_dischargevars()`.

```
void luaK_dischargevars (FuncState *fs, expdesc *e) {
  switch (e->k) {
    case VLOCAL: {
      e->k = VNONRELOC;
      DEBUG_EXPR(raviY_printf(fs, "luaK_dischargevars (VLOCAL->VNONRELOC) %e\n", e));
      break;
    }
    case VUPVAL: {
      e->u.info = luaK_codeABC(fs, OP_GETUPVAL, 0, e->u.info, 0);
      e->k = VRELOCABLE;
      DEBUG_EXPR(raviY_printf(fs, "luaK_dischargevars (VUPVAL->VRELOCABLE) %e\n", e));
      break;
    }
    case VINDEXED: {
      OpCode op = OP_GETTABUP;  /* assume 't' is in an upvalue */
      freereg(fs, e->u.ind.idx);
      if (e->u.ind.vt == VLOCAL) {  /* 't' is in a register? */
        freereg(fs, e->u.ind.t);
        /* table access - set specialized op codes if array types are detected */
        if (e->ravi_type == RAVI_TARRAYFLT &&
            e->u.ind.key_type == RAVI_TNUMINT)
          op = OP_RAVI_FARRAY_GET;
        else if (e->ravi_type == RAVI_TARRAYINT &&
                 e->u.ind.key_type == RAVI_TNUMINT)
          op = OP_RAVI_IARRAY_GET;
        else
          op = OP_GETTABLE;
        if (e->ravi_type == RAVI_TARRAYFLT || e->ravi_type == RAVI_TARRAYINT)
          /* set the type of resulting expression */
          e->ravi_type = e->ravi_type == RAVI_TARRAYFLT ?
                           RAVI_TNUMFLT : RAVI_TNUMINT;
      }
      e->u.info = luaK_codeABC(fs, op, 0, e->u.ind.t, e->u.ind.idx);
      e->k = VRELOCABLE;
      DEBUG_EXPR(raviY_printf(fs, "luaK_dischargevars (VINDEXED->VRELOCABLE) %e\n",␣
↪e));
      break;
    }
    case VVARARG:
    case VCALL: {
      luaK_setoneret(fs, e);
      break;
    }
```

(continues on next page)

(continued from previous page)

```
    default: break;  /* there is one value available (somewhere) */
  }
}
```

8.6.5 fornum statements

The Lua fornum statements create special variables. In order to allows the loop variable to be used in expressions within the loop body we need to set the types of these variables. This is handled in `fornum()` as shown below. Additional complexity is due to the fact that Ravi tries to detect when fornum loops use positive integer step and if this step is `1`; specialized bytecodes are generated for these scenarios.

```
typedef struct Fornuminfo {
  ravitype_t type;
  int is_constant;
  int int_value;
} Fornuminfo;

/* parse the single expressions needed in numerical for loops
 * called by fornum()
 */
static int exp1 (LexState *ls, Fornuminfo *info) {
  /* Since the local variable in a fornum loop is local to the loop and does
   * not use any variable in outer scope we don't need to check its
   * type - also the loop is already optimised so no point trying to
   * optimise the iteration variable
   */
  expdesc e;
  int reg;
  e.ravi_type = RAVI_TANY;
  expr(ls, &e);
  DEBUG_EXPR(raviY_printf(ls->fs, "fornum exp -> %e\n", &e));
  info->is_constant = (e.k == VKINT);
  info->int_value = info->is_constant ? e.u.ival : 0;
  luaK_exp2nextreg(ls->fs, &e);
  lua_assert(e.k == VNONRELOC);
  reg = e.u.info;
  info->type = e.ravi_type;
  return reg;
}

/* parse a for loop body for both versions of the for loop
 * called by fornum(), forlist()
 */
static void forbody (LexState *ls, int base, int line, int nvars, int isnum,␣
↪Fornuminfo *info) {
  /* forbody -> DO block */
  BlockCnt bl;
  OpCode forprep_inst = OP_FORPREP, forloop_inst = OP_FORLOOP;
  FuncState *fs = ls->fs;
  int prep, endfor;
  adjustlocalvars(ls, 3);  /* control variables */
  checknext(ls, TK_DO);
  if (isnum) {
    ls->fs->f->ravi_jit.jit_flags = 1;
    if (info && info->is_constant && info->int_value > 1) {
```

(continues on next page)

(continued from previous page)

```
      forprep_inst = OP_RAVI_FORPREP_IP;
      forloop_inst = OP_RAVI_FORLOOP_IP;
    }
    else if (info && info->is_constant && info->int_value == 1) {
      forprep_inst = OP_RAVI_FORPREP_I1;
      forloop_inst = OP_RAVI_FORLOOP_I1;
    }
  }
  prep = isnum ? luaK_codeAsBx(fs, forprep_inst, base, NO_JUMP) : luaK_jump(fs);
  enterblock(fs, &bl, 0);  /* scope for declared variables */
  adjustlocalvars(ls, nvars);
  luaK_reserveregs(fs, nvars);
  block(ls);
  leaveblock(fs);  /* end of scope for declared variables */
  luaK_patchtohere(fs, prep);
  if (isnum)  /* numeric for? */
    endfor = luaK_codeAsBx(fs, forloop_inst, base, NO_JUMP);
  else {  /* generic for */
    luaK_codeABC(fs, OP_TFORCALL, base, 0, nvars);
    luaK_fixline(fs, line);
    endfor = luaK_codeAsBx(fs, OP_TFORLOOP, base + 2, NO_JUMP);
  }
  luaK_patchlist(fs, endfor, prep + 1);
  luaK_fixline(fs, line);
}

/* parse a numerical for loop, calls forbody()
 * called from forstat()
 */
static void fornum (LexState *ls, TString *varname, int line) {
  /* fornum -> NAME = exp1,exp1[,exp1] forbody */
  FuncState *fs = ls->fs;
  int base = fs->freereg;
  LocVar *vidx, *vlimit, *vstep, *vvar;
  new_localvarliteral(ls, "(for index)");
  new_localvarliteral(ls, "(for limit)");
  new_localvarliteral(ls, "(for step)");
  new_localvar(ls, varname, RAVI_TANY);
  /* The fornum sets up its own variables as above.
     These are expected to hold numeric values - but from Ravi's
     point of view we need to know if the variable is an integer or
     double. So we need to check if this can be determined from the
     fornum expressions. If we can then we will set the
     fornum variables to the type we discover.
  */
  vidx = &fs->f->locvars[fs->nlocvars - 4]; /* index variable - not yet active so get␣
↪it from locvars*/
  vlimit = &fs->f->locvars[fs->nlocvars - 3]; /* index variable - not yet active so␣
↪get it from locvars*/
  vstep = &fs->f->locvars[fs->nlocvars - 2]; /* index variable - not yet active so␣
↪get it from locvars*/
  vvar = &fs->f->locvars[fs->nlocvars - 1]; /* index variable - not yet active so get␣
↪it from locvars*/
  checknext(ls, '=');
  /* get the type of each expression */
  Fornuminfo tidx = { RAVI_TANY,0,0 }, tlimit = { RAVI_TANY,0,0 }, tstep = { RAVI_
↪TNUMINT,0,0 };
```

(continues on next page)

(continued from previous page)

```
  Fornuminfo *info = NULL;
  expl(ls, &tidx);  /* initial value */
  checknext(ls, ',');
  expl(ls, &tlimit);  /* limit */
  if (testnext(ls, ','))
    expl(ls, &tstep);  /* optional step */
  else {  /* default step = 1 */
    tstep.is_constant = 1;
    tstep.int_value = 1;
    luaK_codek(fs, fs->freereg, luaK_intK(fs, 1));
    luaK_reserveregs(fs, 1);
  }
  if (tidx.type == tlimit.type && tlimit.type == tstep.type && (tidx.type == RAVI_
↪TNUMFLT || tidx.type == RAVI_TNUMINT)) {
    if (tidx.type == RAVI_TNUMINT && tstep.is_constant)
      info = &tstep;
    /* Ok so we have an integer or double */
    vidx->ravi_type = vlimit->ravi_type = vstep->ravi_type = vvar->ravi_type = tidx.
↪type;
    DEBUG_VARS(raviY_printf(fs, "fornum -> setting type for index %v\n", vidx));
    DEBUG_VARS(raviY_printf(fs, "fornum -> setting type for limit %v\n", vlimit));
    DEBUG_VARS(raviY_printf(fs, "fornum -> setting type for step %v\n", vstep));
    DEBUG_VARS(raviY_printf(fs, "fornum -> setting type for variable %v\n", vvar));
  }
  forbody(ls, base, line, 1, 1, info);
}
```

8.7 Handling of Upvalues

Upvalues can be used to update local variables that have static typing specified. So this means that upvalues need to be annotated with types as well and any operation that updates an upvalue must be type checked. To support this the Lua parser has been enhanced to record the type of an upvalue in `Upvaldesc`:

```
/*
** Description of an upvalue for function prototypes
*/
typedef struct Upvaldesc {
  TString *name;  /* upvalue name (for debug information) */
  ravitype_t type; /* RAVI type of upvalue */
  lu_byte instack;  /* whether it is in stack */
  lu_byte idx;  /* index of upvalue (in stack or in outer function's list) */
} Upvaldesc;
```

Whenever a new upvalue is referenced, we assign the type of the the upvalue to the expression in function `singlevaraux()` - relevant code is shown below:

```
static int singlevaraux (FuncState *fs, TString *n, expdesc *var, int base) {
  /* ... omitted code ... */
    int idx = searchupvalue(fs, n);  /* try existing upvalues */
    if (idx < 0) {  /* not found? */
      if (singlevaraux(fs->prev, n, var, 0) == VVOID) /* try upper levels */
        return VVOID;  /* not found; is a global */
      /* else was LOCAL or UPVAL */
      idx  = newupvalue(fs, n, var);  /* will be a new upvalue */
```

(continues on next page)

(continued from previous page)

```
    }
    init_exp(var, VUPVAL, idx, fs->f->upvalues[idx].type); /* RAVI : set upvalue type␣
↪*/
    return VUPVAL;
    /* ... omitted code ... */
}
```

The function `newupvalue()` sets the type of a new upvalue:

```
/* create a new upvalue */
static int newupvalue (FuncState *fs, TString *name, expdesc *v) {
  Proto *f = fs->f;
  int oldsize = f->sizeupvalues;
  checklimit(fs, fs->nups + 1, MAXUPVAL, "upvalues");
  luaM_growvector(fs->ls->L, f->upvalues, fs->nups, f->sizeupvalues,
                  Upvaldesc, MAXUPVAL, "upvalues");
  while (oldsize < f->sizeupvalues) f->upvalues[oldsize++].name = NULL;

  f->upvalues[fs->nups].instack = (v->k == VLOCAL);
  f->upvalues[fs->nups].idx = cast_byte(v->u.info);
  f->upvalues[fs->nups].name = name;
  f->upvalues[fs->nups].type = v->ravi_type;
  luaC_objbarrier(fs->ls->L, f, name);
  return fs->nups++;
}
```

When we need to generate assignments to an upvalue (OP_SETUPVAL) we need to use more specialized opcodes that do the necessary conversion at runtime. This is handled in `luaK_storevar()` in `lcode.c`:

```
/* Emit store for LHS expression. */
void luaK_storevar (FuncState *fs, expdesc *var, expdesc *ex) {
  switch (var->k) {
    /* ... omitted code .. */
    case VUPVAL: {
      OpCode op = check_valid_setupval(fs, var, ex);
      int e = luaK_exp2anyreg(fs, ex);
      luaK_codeABC(fs, op, e, var->u.info, 0);
      break;
    }
    /* ... omitted code ... */
  }
}

static OpCode check_valid_setupval(FuncState *fs, expdesc *var, expdesc *ex) {
  OpCode op = OP_SETUPVAL;
  if (var->ravi_type != RAVI_TANY && var->ravi_type != ex->ravi_type) {
    if (var->ravi_type == RAVI_TNUMINT)
      op = OP_RAVI_SETUPVALI;
    else if (var->ravi_type == RAVI_TNUMFLT)
      op = OP_RAVI_SETUPVALF;
    else if (var->ravi_type == RAVI_TARRAYINT)
      op = OP_RAVI_SETUPVAL_IARRAY;
    else if (var->ravi_type == RAVI_TARRAYFLT)
      op = OP_RAVI_SETUPVAL_FARRAY;
    else
      luaX_syntaxerror(fs->ls,
```

(continues on next page)

(continued from previous page)

```
                      luaO_pushfstring(fs->ls->L, "Invalid assignment of "
                                                   "upvalue: upvalue type "
                                                   "%d, expression type %d",
                                       var->ravi_type, ex->ravi_type));
  }
  return op;
}
```

8.8 VM Enhancements

A number of new opcodes are introduced to allow type specific operations.

Currently there are specialized versions of `ADD`, `SUB`, `MUL` and `DIV` operations. This will be extended to cover additional operators such as `IDIV`. The `ADD` and `MUL` operations are implemented in a similar way. Both allow a second operand to be encoded directly in the `C` operand - when the value is a constant in the range [0,127].

One thing to note is that apart from division if an operation involves constants it is folded by Lua. Divisions are treated specially - an expression involving the `0` constant is not folded, even when the `0` is a numerator. Also worth noting is that DIV operator results in a float even when two integers are divided; you have to use `IDIV` to get an integer result - this opcode triggered in Lua 5.3 when the `//` operator is used.

A divide by zero when using integers causes a run time error, whereas for floating point operation the result is NaN.

CHAPTER 9

LLVM Compilation hooks in Ravi

The current approach is Ravi is that a Lua function can be compiled at the function level. (Note that this is the plan - I am working on the implementation).

In terms of changes to support this - we essentially have following. First we have a bunch of C functions - think of these are the compiler API:

```
#ifdef __cplusplus
extern "C" {
#endif

struct lua_State;
struct Proto;

/* Initialise the JIT engine */
int raviV_initjit(struct lua_State *L);

/* Shutdown the JIT engine */
void raviV_close(struct lua_State *L);

/* Compile the given function if possible */
int raviV_compile(struct lua_State *L, struct Proto *p);

/* Free the JIT structures associated with the prototype */
void raviV_freeproto(struct lua_State *L, struct Proto *p);

#ifdef __cplusplus
}
#endif
```

Next the `Proto` struct definition has some extra fields:

```
typedef struct RaviJITProto {
  lu_byte jit_status; // 0=not compiled, 1=can't compile, 2=compiled, 3=freed
  void *jit_data;
```

(continues on next page)

(continued from previous page)

```
  lua_CFunction jit_function;
} RaviJITProto;

/*
** Function Prototypes
*/
typedef struct Proto {
  CommonHeader;
  lu_byte numparams;  /* number of fixed parameters */
  lu_byte is_vararg;
  lu_byte maxstacksize;  /* maximum stack used by this function */
  int sizeupvalues;  /* size of 'upvalues' */
  int sizek;  /* size of 'k' */
  int sizecode;
  int sizelineinfo;
  int sizep;  /* size of 'p' */
  int sizelocvars;
  int linedefined;
  int lastlinedefined;
  TValue *k;  /* constants used by the function */
  Instruction *code;
  struct Proto **p;  /* functions defined inside the function */
  int *lineinfo;  /* map from opcodes to source lines (debug information) */
  LocVar *locvars;  /* information about local variables (debug information) */
  Upvaldesc *upvalues;  /* upvalue information */
  struct LClosure *cache;  /* last created closure with this prototype */
  TString  *source;  /* used for debug information */
  GCObject *gclist;
  /* RAVI */
  RaviJITProto ravi_jit;
} Proto;
```

The `ravi_jit` member is initialized in `lfunc.c`:

```
Proto *luaF_newproto (lua_State *L) {
  GCObject *o = luaC_newobj(L, LUA_TPROTO, sizeof(Proto));
  Proto *f = gco2p(o);
  f->k = NULL;
  /* code ommitted */
  f->ravi_jit.jit_data = NULL;
  f->ravi_jit.jit_function = NULL;
  f->ravi_jit.jit_status = 0; /* not compiled */
  return f;
}
```

The corresponding function to free is:

```
void luaF_freeproto (lua_State *L, Proto *f) {
  raviV_freeproto(L, f);
  luaM_freearray(L, f->code, f->sizecode);
  luaM_freearray(L, f->p, f->sizep);
  luaM_freearray(L, f->k, f->sizek);
  luaM_freearray(L, f->lineinfo, f->sizelineinfo);
  luaM_freearray(L, f->locvars, f->sizelocvars);
  luaM_freearray(L, f->upvalues, f->sizeupvalues);
  luaM_free(L, f);
}
```

When a Lua Function is called it goes through `luaD_precall()` in `ldo.c`. This has been modified to invoke the compiler / use compiled version:

```
/*
** returns true if function has been executed (C function)
*/
int luaD_precall (lua_State *L, StkId func, int nresults) {
  lua_CFunction f;
  CallInfo *ci;
  int n;  /* number of arguments (Lua) or returns (C) */
  ptrdiff_t funcr = savestack(L, func);
  switch (ttype(func)) {

      /* omitted */

  case LUA_TLCL: {  /* Lua function: prepare its call */
    CallInfo *prevci = L->ci; /* RAVI - for validation */
    StkId base;
    Proto *p = clLvalue(func)->p;
    n = cast_int(L->top - func) - 1;  /* number of real arguments */
    luaD_checkstack(L, p->maxstacksize);
    for (; n < p->numparams; n++)
      setnilvalue(L->top++);  /* complete missing arguments */
    if (!p->is_vararg) {
      func = restorestack(L, funcr);
      base = func + 1;
    }
    else {
      base = adjust_varargs(L, p, n);
      func = restorestack(L, funcr);  /* previous call can change stack */
    }
    ci = next_ci(L);  /* now 'enter' new function */
    ci->nresults = nresults;
    ci->func = func;
    ci->u.l.base = base;
    ci->top = base + p->maxstacksize;
    lua_assert(ci->top <= L->stack_last);
    ci->u.l.savedpc = p->code;  /* starting point */
    ci->callstatus = CIST_LUA;
    ci->jitstatus = 0;
    L->top = ci->top;
    luaC_checkGC(L);  /* stack grow uses memory */
    if (L->hookmask & LUA_MASKCALL)
      callhook(L, ci);
    if (compile) {
      if (p->ravi_jit.jit_status == 0) {
        /* not compiled */
        raviV_compile(L, p, 0);
      }
      if (p->ravi_jit.jit_status == 2) {
        /* compiled */
        lua_assert(p->ravi_jit.jit_function != NULL);
        ci->jitstatus = 1;
        /* As JITed function is like a C function
         * employ the same restrictions on recursive
         * calls as for C functions
         */
        if (++L->nCcalls >= LUAI_MAXCCALLS) {
```

(continues on next page)

(continued from previous page)

```
          if (L->nCcalls == LUAI_MAXCCALLS)
            luaG_runerror(L, "C stack overflow");
          else if (L->nCcalls >= (LUAI_MAXCCALLS + (LUAI_MAXCCALLS >> 3)))
            luaD_throw(L, LUA_ERRERR);  /* error while handing stack error */
        }
        /* Disable YIELDs - so JITed functions cannot
         * yield
         */
        L->nny++;
        (*p->ravi_jit.jit_function)(L);
        L->nny--;
        L->nCcalls--;
        lua_assert(L->ci == prevci);
        /* Return a different value from 1 to
         * allow luaV_execute() to distinguish between
         * JITed function and true C function
         */
        return 2;
      }
    }
    return 0;
  }
  default: {  /* not a function */

        /* omitted */
  }
  }
}
```

Note that the above returns 2 if compiled Lua function is called. The behaviour in `lvm.c` is similar to that when a C function is called.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search