Optimizing Lua VM Bytecode using Global Dataflow Analysis

Satoru Kawahara
s1311350@coins.tsukuba.ac.jp

Contents

Chapter 1

Introduction

Lua is a lightweight yet powerful and flexible language to describe. In recent years it plays in active part such as embedded scripting. There are some researches for Lua VM, just-in-time(JIT) compilation [2], run-time type specialization [8], and others. But they lose not only compatible with the VM bytecode, but the portability the VM marks.

Then I tried to optimize the bytecode itself. As a matter of course, it should get the compatibility and portability. I implemented the optimizer (called Opeth) written in MoonScript[1].

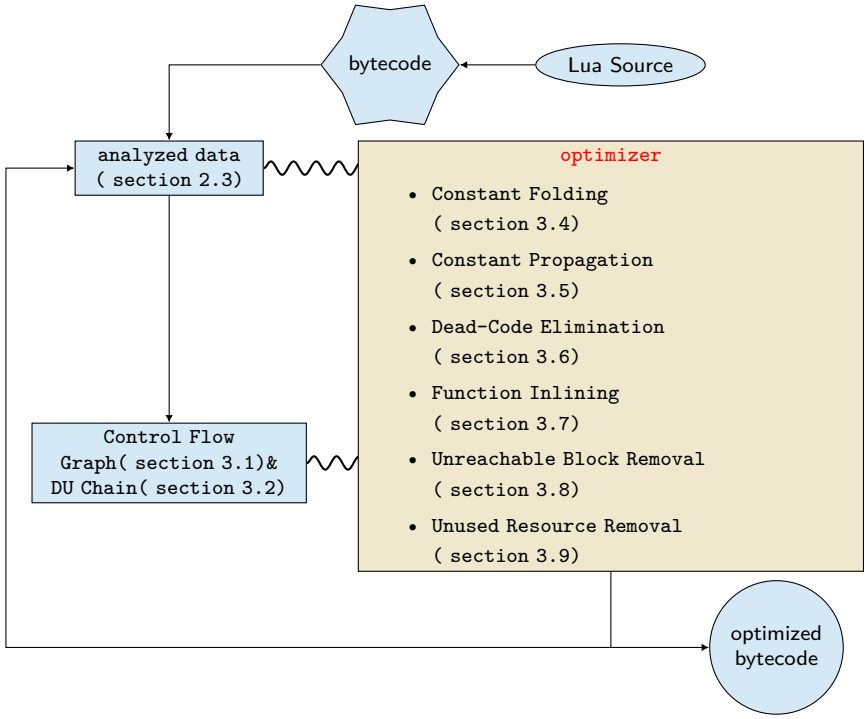

Figure 1.1: optimization image

[1] https://moonscript.org

Chapter 2

The structure of the bytecode

Firstly, the optimizer reads the bytecode and gets the information.
The bytecode structure is following:

2.1 Header block

2.1.1 4 bytes

Header Signature: 0x1B4C7561, the ascii codes of “Esc”, ‘L’, ‘u’ and ‘a’

2.1.2 1 byte

Version Number: The version of the format; in this case is 0x53 for Lua 5.3. High hex digit is the major version number and low hex digit is the minor version number.

2.1.3 1 byte

Format version: 0x00 is the official version.

2.1.4 6 bytes

LUAC_DATA: 0x19930D0A1A0A, data to catch conversion errors. 0x0D0A is “CR LF”, represents the return code on DOS system and 0x0A is “LF”, represents the return code on UNIX systems. We can detect the error if the return codes are changed.

2.1.5 1 byte

Size of `int`

2.1.6 1 byte

Size of `size_t`

2.1.7 1 byte

Size of Lua VM Instruction

2.1.8 1 byte

Size of Lua’s `integer`

2.1.9 1 byte

Size of Lua's `number`

2.1.10 8 bytes

Endianness flag: How represented 0x5678; If it's equal to 0x0000000000005678 then the endianness is big endian, else if it's equal to 0x7856000000000000 then the endianness is little endian.

2.1.11 9 bytes

LUAC_NUM: Checking IEEE754 float format whether it can be decoded to be 370.5.

2.2 Function block

2.2.1 n bytes

'\0' if the debug information is stripped, otherwise 1 byte of (1 + length of filename (≤ 255)) + prefix + the name. If file is generated on standard input, prefix is '=', otherwise it is '@'.

Whether the bytecode is stripped or not is decided with it.

2.2.2 (size of `int`) bytes

The line this function definition: If the function is top level, the number is 0.

2.2.3 (size of `int`) bytes

The last line this function is defined: If the function is top level, the number is 0, too.

2.2.4 1 byte

Parameter: The number of function's arguments.

2.2.5 1 byte

Vararg: Variable arguments flag. If it is '\1' then it uses Variable arguments.

2.2.6 1 byte

Register numbers: The number of registers to use.

2.2.7 (List)

List of Instructions. See 2.2.17.

2.2.8 (List)

List of Constants. See 2.2.18.

2.2.9 (List)

List of Upvalues. See 2.2.19.

2.2.10 (List)

List of Prototypes. See 2.2.20.

2.2.11 (size of `int`) bytes

The number of the instructions.

2.2.12 (List)

The list of the line numbers where each instruction is generated. When the debug information is stripped, its length is zero.

2.2.13 (size of `int`) bytes

The number of local variales.

2.2.14 (List)

The list of local variables' information.

Variable name (2.2.14)	lifespan begin (2.2.14)	lifespan end (2.2.14)

Figure 2.1: The format of Variable information

n bytes

Variable name: 1 byte of (1 + length of variable name (≤ 255)) + the name.

(size of `int`) bytes

Lifespan begin: The beginning of the variable's lifespan.

(size of `int`) bytes

Lifespan end: The end of the variable's lifespan.

When the debug information is stripped, its length is zero.

2.2.15 (size of `int`) bytes

The number of upvalues.

2.2.16 (List)

The list of upvalues' information.

n bytes: Upvalue name

1 byte of (1 + length of upvalue name (≤ 255)) + the name.

When the debug information is stripped, its length is zero.

Next, I write the lists which are not described: Instruction list, Constant list, Upvalue list, and Prototype list.

2.2.17 List of Instructions

The first list is the instruction list.

length of instruction list (integer)	instructions (Figure 2.3)

Figure 2.2: instruction list

In Lua 5.3, instructions have 4 modes:

Table 2.1: 4 modes for the instruction

iABC	opcode R(A) R(B) R(C)
iAsBx	opcode R(A) (signed integer)Bx
iABx	opcode R(A) (unsigned integer)Bx
iAx	opcode R(Ax)

Lua instructions are fixed size, 32 bit. And the structure of instruction is following:

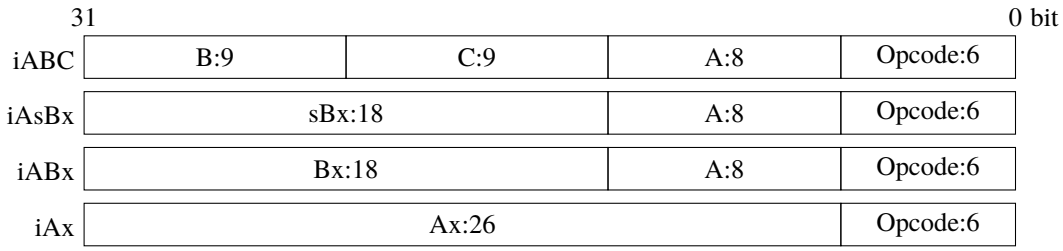

Figure 2.3: Instruction Formats

Lua 5.3 has 47 instructions. `RETURN` instruction is always generated, so the length of the list is at least 1.

2.2.18 List of Constants

The next is constant list. Lua VM has constant pool to fetch the constant value instead of immediate values, which is referenced in the each function.

length of constat list (integer)	constants (Figure 2.5)

Figure 2.4: constant list

For each constant, the type is represented by 1 byte. And the value is endianness-sensitive.

Table 2.2: types of constants

type	value
0x00(`nil`)	0 byte
0x01(`bool`)	1 byte (`0` or `1`)
0x03(`number`)	size of Lua's `number` byte of number (IEEE754 format)
0x13(`integer`)	size of Lua's `integer` byte of signed integer
0x04(`string`)	(length of the string (< 256) + 1) byte of string + `'\0'`
0x14(`long string`)	(length of the string (≥ 256) + 1) byte of string + `'\0'`

So, the representation is below:

type (1 byte)	value (n bytes) (Table 2.2)

Figure 2.5: upvalue list

2.2.19 List of Upvalues

The third is the upvalue list. Upvalue, as known as free variable, is the variable defined in the upper closure.

length of upvalue list (integer)	upvalues (Figure 2.7)

Figure 2.6: upvalue list

The Upvalue format has 2 bytes, the half of which is called "register" and the rest is "instack". "regisnter" is the index to be referred in the instructions. And "instack" is boolean that whether the upvalue is in the upper closure. The format is endianness-sensitive.

Big Endian	register (1 byte)	instack (1 byte)
Little Endian	instack (1 byte)	register (1 byte)

Figure 2.7: upvalue format

2.2.20 List of Prototypes

Finally, the list consists of prototypes.

length of prototype list (integer)	prototypes (function block) (2.2)

Figure 2.8: Prototype list

We can regard prototype as the headless bytecode. Yes, the prototypes are represented bytecode same to the top level.

Because it is not documented, we need to read the source code [6] or unofficial documentations. I referred to the research of Kein-Hong's [5].

2.3 Convert to MoonScript's treatable format

The reader simultaneously executes reading the bytecode and converting the information to MoonScript's treatable format. The format consists of mainly `table` type and some data type, `string` and `number`.

2.3.1 Header block

I represented the header itself as `table`.

Listing 2.1: MoonScript's representation of Header block

```
1 {
2   hsig: string ( subsection 2.1.1)
3   version: string ( subsection 2.1.2)
```

```
  format: string ( subsection 2.1.3)
  luac_data: string ( subsection 2.1.4)

  size: {
    int: number ( subsection 2.1.5)
    size_t: number ( subsection 2.1.6)
    instruction: number ( subsection 2.1.7)
    lua_integer: number ( subsection 2.1.8)
    lua_number: number ( subsection 2.1.9)
  }
}
```

2.3.2 Function block

Function block itself is represented as `table`.

Listing 2.2: MoonScript's representation of Function block

```
{
  chunkname: string  subsection 2.2.1

  line vars: {
    defined: number ( subsection 2.2.2)
    lastdefined: numbers ( subsection 2.2.3)
  }

  params: string ( subsection 2.2.4)
  vararg: string ( subsection 2.2.5)
  regnum: string ( subsection 2.2.6)

  instruction: {
    {
      op: string
      operand......: number
    } ...... ( subsection 2.2.7)
  }

  constant: {
    {
      type: number
      val: some types
    } ...... ( subsection 2.2.8)
  }

  upvalue: {
    {
      instack: number
      reg: number
    } ...... ( subsection 2.2.9)
  }

  prototype: {number, table ......} ( subsection 2.2.10)

  debug: {
    linenum: number ( subsection 2.2.11)
```

```
38      opline: {number ......} ( subsection 2.2.12)
39      varnum: number ( subsection 2.2.13)
40      varinfo: {
41        {
42          varname: string ( section 2.2.14)
43          lifebegin: number ( section 2.2.14)
44          lifeend: number ( section 2.2.14)
45        } ...... ( subsection 2.2.14)
46      }
47      upvnum: number ( subsection 2.2.15)
48      upvinfo: {string ...... ( section 2.2.16) } ( subsection 2.2.16)
49    }
50  }
```

And lastly the optimizer writes tye optimized bytecode based on this tables to a file.
The source code of the bytecode reader/writer is in appendix (section A.5).

Chapter 3

Optimizing

3.1 Control Flow Graph

For the optimizations, firstly, I try to use the control flow analysis. As one of the techniques of the analysis, Control Flow Graph(CFG) is well known.

The nodes of the graph is called “basic blocks”. Accoding to this document [3],

“

> *In the following cases, the directed edge is drawn from the block B_1 to the block B_2:*
>
> 1. *In the last statement of B_1 there is a conditional or unconditional jump to the first statement of B_2*
> 2. *B_1 ends with statements other than unconditional jump, and B_2 comes immediately after B_1 on the letter of the program.*

”

3.1.1 Configuration Method

Listing 3.1: the structure of a basic block

```
{
  start: number -- the starting position
  end: number -- the ending position
  succ: table -- the successor basic block list
  pred: table -- the predecessor basic block list
}
```

1. Let each instruction be the basic blocks. Tag the index of the elements of the instruction list and the index of the next instruction to be executed (the starting position of the successor basic block to point to). Almost all are tagged with the line number+1, but some are different or tagged with multiple destinations.

JMP, FORPREP	the index + RB + 1
LOADBOOL	the index + 2 if RC == 1
TEST, TESTSET, LT, LE, EQ	the index + 1, the index + 2
FORLOOP, TFORLOOP	the index + 1, the index + RB + 1
RETURN, TAILCALL	*none*

RETURN and TAILCALL are set to the last of the block, and the block has no successor basic blocks.

2. Connect each basic block.

   - If the block B_1 points to the starting position of the block B_2, add B_2 to the predecessor basic block list of B_1 and add B_1 to the successor basic block list of B_2
   - else
     (a) Divide B_2 into B_{2a} and B_{2b}:
       – B_{2a}: the starting position is the position of B_2 -1, the ending position is where B_1 poits to, the successor block list are none, and the predecessor block list are which B_2 has.
       – B_{2b}: the starting position is the position of B_2, the ending position is the position of B_2, the successor block list are are which B_2 has, and the predecessor block list are none.
     (b) Add B_{2b} to the predecessor block list of B_{2a}, and add B_{2a} to the successor block list of B_{2b}.
     (c) Add B_{2b} to the predecessor block list of B_1, and add B_1 to the successor block list of B_{2b}.

Apply the method to each closure.
Suppose think about the lua code and the following bytecode.

Listing 3.2: example for constructing CFG

```lua
local x = 3

if x < 5 then
  print"hello"
else
  print"world"
end
```

```
        1       [1]     LOADK           0 -1    ; 3
        2       [3]     LT              0 0 -2  ; - 5
        3       [3]     JMP             0 4     ; to 8
        4       [4]     GETTABUP        1 0 -3  ; _ENV "print"
        5       [4]     LOADK           2 -4    ; "hello"
        6       [4]     CALL            1 2 1
        7       [4]     JMP             0 3     ; to 11
        8       [6]     GETTABUP        1 0 -3  ; _ENV "print"
        9       [6]     LOADK           2 -5    ; "world"
        10      [6]     CALL            1 2 1
        11      [7]     RETURN          0 1
```

First, apply the process 1.

```
{
  {op = "LOADK", line = 1, succ_pos = {2}},
  {op = "LT", line = 2, succ_pos = {3, 4}},
  {op = "JMP", line = 3, succ_pos = 8},
  {op = "GETTABUP", line = 4, succ_pos = {5}},
  {op = "LOADK", line = 5, succ_pos = {6}},
  {op = "CALL", line = 6, succ_pos = {7}},
  {op = "JMP", line = 7, succ_pos = {11}},
  {op = "GETTABUP", line = 8, succ_pos = {9}},
  {op = "LOADK", line = 9, succ_pos = {10}},
  {op = "CALL", line = 10, succ_pos = {11}},
  {op = "RETURN", line = 11, succ_pos = {}}
}
```

And compete by connecting with the process 2.

```
{
  {start = 1, end = 2, succ = {2, 3}, prev = {}}, -- block 1
  {start = 3, end = 4, succ = {4}, prev = {1}}, -- block 2
  {start = 4, end = 7, succ = {5}, prev = {1}}, -- block 3
  {start = 8, end = 10, succ = {2}, prev = {2}},
  {start = 11, end = 11, succ = {}, prev = {3, 4}}
}
```

Passed to the visualiser, it is displayed as following:

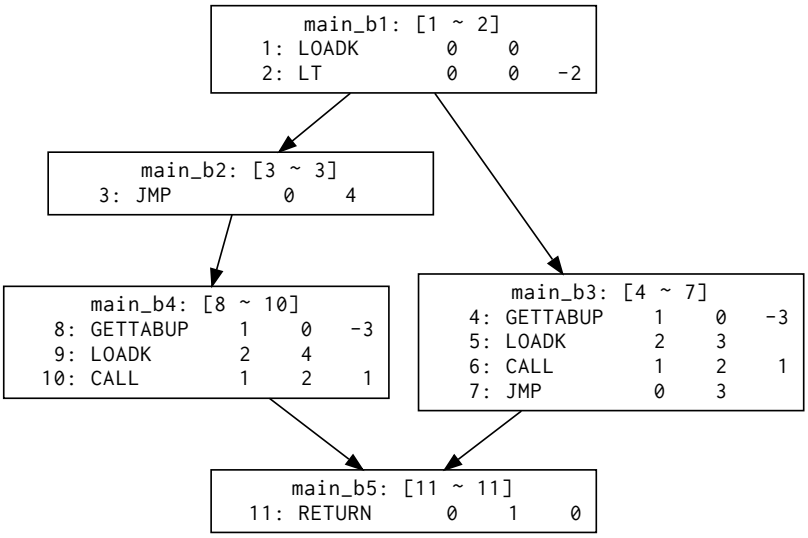

Figure 3.1: the CFG with the visualiser

The source code of the CFG constructor is in appendix (Listing A.3).

3.2 Define-Use Chain

Define-Use Chain (DU Chain), in this context, is a data structure which can refer to the instruction which define (or assign to) the register value, from the instruction which uses the register. In contrast, Use-Define Chain (UD Chain) is a structure which can refer to the use of a register from the definition of the register.

For instance, dead-code elimination, which is described later, uses this data structure. If the number of the use of a register is 0, the instruction which defines a value of the register can be regarded as unnecessary and removed.

In this implementation, I use the mix in DU Chain and UD Chain, which can refer to the use from the definition and to the definition from the use. This is the key to global dataflow analysis.

3.2.1 Configuration Method

1. For each blocks of the CFG, construct sets, `gen` and `use`. The elements of `gen` have a information about which registers is defined and where it is defined. The elements of `use` have a information about which register is used and where it is used.

   These may be clear by the instruction executed.

2. In the basic block, add sets, `in`, `kil` and `out`. `in` replesents which the definition are propagated to the block from the predecessor blocks. `kill` is the intersection of `in` and `gen`. `out` replesents which the definitions are propagated to the succsesor blocks.

   Firstly, these sets have no item.

3. For each blocks, Update `in`, `out` and `kill`. `in` can be the union of all the `out` the predecessor blocks have. `kill` can be the intersectoin of `in` and `gen`. `out` can be the union of *latest* `gen` and the difference between `in` and `kill`. *latest* means that, for each variables in `gen`, pay attention to the last assignment and the ignore the other before.

4. Add a set `def` to the block. `def` can be traced where a register is defined in the block. It is simplely defined, the union of `gen` and `in`.

The source code of the DU/UD Chain constructor is in appendix (Listing A.13).

3.3 Type Inference and Getting Value

For some optimization techniques, it needs a value of a register and a type of the value. Global dataflow analysis may be possible to detect the value and the type.

Types of some instructions which assign a immediate value are detected at once. Aritmetic instructions is detected from the operands and the result. If the operands contains `table`value, as we can say the source code of the bytecode uses a metamethod, it is impossible to detect as positive fail.

If the value or the type is not inferable in that block, query is made to each the predecessor block. While the answer is returned or the query reaches the enter block, it is made to the predecessor blocks. In this case there may be several the candidates of the value or the type, in that case inference is not possible.

The source code of the type inference and getting value is in appendix (Listing A.12).

In this research, I implemented following optimization techniques:

- Constant Folding (section 3.4)
- Constant Propagation (section 3.5)
- Dead-code Elimination (section 3.6)
- Function Inlining (section 3.7)
- Unreachable block Removal (section 3.8)
- Unused Resource Removal (section 3.9)

3.4 Constant Folding

This optimization “executes” some of operation instruction and replace it with immediate value instructions if possible.

Soppose think about the optimization to a instruction `ADD 0 1 2`. If the register `0` and `1` are `number` value, get constants of the register `0` (as `cst(0)`) and `1` (as `cst(1)`). Let the result of `cst(0) + cst(1)` be `rst`. If `rst` is in the constant list, get the index of the value, else add the value to the list and get the length. Let the index or the length be `idx`, and swap the instruction with `LOADK 0 idx`.

3.5 Constant Propagation

This optimization works like this: go to see the instruction defining the register pointed to by the second operand of the MOVE instruction. If the instruction is MOVE, set the second operand of the later MOVE to the second operand of the first MOVE. If the instruction is LOADK, swap MOVE with LOADK.

In this implementation, this optimization itself is few effective for bytecode. It aims to, for each instruction, reduce the dependencies from MOVE and advance the optimizatoins such as **dead-code elimination**.

3.6 Dead-Code Elimination

This optimization removes a instruction from a bytecode one by one. If a register which is defined by LOADK, CLOSURE, LOADNIL or MOVE, check the use of the register by define-use chain (section 3.2). If the use is 0, the instruction defines the register is regarded as be not needed and removed.

In this implementation, the optimization module also removes contitional expressions. For EQ, LT, GT, TEST and TESTSET,like constant fonding (section 3.4), infer the types and the values the registers of the operands points, compare and may remove the instruction.

3.7 Function Inlining

Expands a closure called from CALL instruction. Lua VM adopts register window [7], and the optimization reduces the cost.

If it succeed in fetching the closure called from CALL instruction, replace the instruction with the instructions which the closure contains. To replace, add offsets to the operands to avoid to collide the registers already defined. The offsets are decided by the operands of CALL and the number of arguments of the closure. And a part of these is replaced with other instructions. RETURN needs to replace with MOVE and, if it is not the last of the instructions, add JMP.

But this implementation is imcomplete and may occur segmentation fault, thus the more research to the VM and the instruction is needed.

3.8 Unreachable Block Removal

It is nothing but remove a basic block if it is not enter block and has no predecessor blocks. This optimization is less effective for speed up but can reduce the bytecode size. The reduction make the optimizations itself fast.

3.9 Unused Resource Removal

Delete constants and closures which are no longer used by the optimizations from constant list and prototype list. If a constant or closure is removed from a list, it is necessary to adjust the operand of the instruction pointing to that item. It is also not effective for speed up but effective for the optimization itself.

The source code of the optimizers is in appendix (section A.2).

Chapter 4

Benchmark

4.1 Environment

- OS
  ArchLinux 64bit kernel 4.9.8-1
- CPU
  Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz, 2 Core 4 Thread
- RAm
  DDR3 8GB
- Lua VM
  Lua 5.3.4

4.2 Target Code

Here is a code and generate target bytecode.

Listing 4.1: target code

```
local n

local function f()
  local a = 3
  local b = 4
  local c = 10
  local d = 3
  local e = 10
  return a + b - c * d / e
end

for _ = 0, 100000000 do
  n = f()
end

return n
```

Listing 4.2: target bytecode

```
main <benchmark/calc.lua:0,0> (12 instructions at 0x1b569e0)
0+ params, 7 slots, 1 upvalue, 6 locals, 3 constants, 1 function
       1      [1]    LOADNIL       0 0
       2      [10]   CLOSURE       1 0    ; 0x1b56b40
       3      [12]   LOADK         2 -1   ; 0
       4      [12]   LOADK         3 -2   ; 100000000
       5      [12]   LOADK         4 -3   ; 1
       6      [12]   FORPREP       2 3    ; to 10
       7      [13]   MOVE          6 1
       8      [13]   CALL          6 1 2
       9      [13]   MOVE          0 6
       10     [12]   FORLOOP       2 -4   ; to 7
       11     [16]   RETURN        0 2
       12     [16]   RETURN        0 1
constants (3) for 0x1b569e0:
       1      0
       2      100000000
       3      1
locals (6) for 0x1b569e0:
       0      n      2      13
       1      f      3      13
       2      (for index)   6      11
       3      (for limit)   6      11
       4      (for step)    6      11
       5      _      7      10
upvalues (1) for 0x1b569e0:
       0      _ENV   1      0

function <benchmark/calc.lua:3,10> (11 instructions at 0x1b56b40)
0 params, 7 slots, 0 upvalues, 5 locals, 3 constants, 0 functions
       1      [4]    LOADK         0 -1   ; 3
       2      [5]    LOADK         1 -2   ; 4
       3      [6]    LOADK         2 -3   ; 10
       4      [7]    LOADK         3 -1   ; 3
       5      [8]    LOADK         4 -3   ; 10
       6      [9]    ADD           5 0 1
       7      [9]    MUL           6 2 3
       8      [9]    DIV           6 6 4
       9      [9]    SUB           5 5 6
       10     [9]    RETURN        5 2
       11     [10]   RETURN        0 1
constants (3) for 0x1b56b40:
       1      3
       2      4
       3      10
locals (5) for 0x1b56b40:
       0      a      2      12
       1      b      3      12
       2      c      4      12
       3      d      5      12
       4      e      6      12
upvalues (0) for 0x1b56b40:
```

4.3 Results

The optimizer always use Unused Resource Removal.

Table 4.1: Benchmark timings and number of instructions for different optimizations

Sort		Time (s)	The number of instructions	bytecode size(byte)
No-optimized	(Listing 4.2)	7.953	23	262
Constant Folding	(Listing 4.3)	6.440	23	289
Without Constant Folding	(Listing 4.4)	5.679	19	206
Constant Propagation	(Listing 4.5)	8.033	23	262
Without Constant Propagation	(Listing 4.6)	1.338	11	147
Funciton Inlining	(Listing 4.7)	6.064	32	325
Without Funciton Inlining	(Listing 4.8)	4.278	15	212
Dead-Code Elimination	(Listing 4.9)	8.633	23	262
Without Dead-Code Elimination	(Listing 4.10)	4.319	32	370
Unreachable Block Removal	(Listing 4.11)	8.945	23	262
Without Unreachable Block Removal	(Listing 4.12)	0.810	9	139
Full-optimized	(Listing 4.13)	0.825	9	139

Listing 4.3: Constant Folding

```
main <?:0,0> (12 instructions at 0x252e9e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 3 constants, 1 function
  1 [-] LOADNIL  0 0
  2 [-] CLOSURE  1 0 ; 0x252eb00
  3 [-] LOADK    2 -1 ; 0
  4 [-] LOADK    3 -2 ; 100000000
  5 [-] LOADK    4 -3 ; 1
  6 [-] FORPREP  2 3 ; to 10
  7 [-] MOVE     6 1
  8 [-] CALL     6 1 2
  9 [-] MOVE     0 6
  10  [-] FORLOOP  2 -4 ; to 7
  11  [-] RETURN  0 2
  12  [-] RETURN  0 1
constants (3) for 0x252e9e0:
  1 0
  2 100000000
  3 1
locals (0) for 0x252e9e0:
upvalues (1) for 0x252e9e0:
  0 - 1 0

function <?:3,10> (11 instructions at 0x252eb00)
0 params, 7 slots, 0 upvalues, 0 locals, 6 constants, 0 functions
  1 [-] LOADK    0 -1 ; 3
  2 [-] LOADK    1 -2 ; 4
  3 [-] LOADK    2 -3 ; 10
  4 [-] LOADK    3 -1 ; 3
  5 [-] LOADK    4 -3 ; 10
  6 [-] LOADK    5 -4 ; 7
  7 [-] LOADK    6 -5 ; 30
  8 [-] LOADK    6 -1 ; 3
  9 [-] LOADK    5 -6 ; 0
```

```
  10  [-] RETURN  5 2
  11  [-] RETURN  0 1
constants (6) for 0x252eb00:
  1 3
  2 4
  3 10
  4 7
  5 30
  6 0
locals (0) for 0x252eb00:
upvalues (0) for 0x252eb00:
```

Listing 4.4: Without Constant Folding

```
main <?:0,0> (19 instructions at 0x245e9e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 6 constants, 0 functions
  1 [-] LOADNIL  0 0
  2 [-] LOADK   2 -1 ; 0
  3 [-] LOADK   3 -2 ; 100000000
  4 [-] LOADK   4 -3 ; 1
  5 [-] FORPREP  2 11 ; to 17
  6 [-] LOADK   7 -4 ; 3
  7 [-] LOADK   8 -5 ; 4
  8 [-] LOADK   9 -6 ; 10
  9 [-] LOADK   10 -4 ; 3
  10  [-] LOADK   11 -6 ; 10
  11  [-] ADD     12 7 8
  12  [-] MUL     13 9 10
  13  [-] DIV     13 13 11
  14  [-] SUB     12 12 13
  15  [-] MOVE    6 12
  16  [-] MOVE    0 6
  17  [-] FORLOOP  2 -12 ; to 6
  18  [-] RETURN  0 2
  19  [-] RETURN  0 1
constants (6) for 0x245e9e0:
  1 0
  2 100000000
  3 1
  4 3
  5 4
  6 10
locals (0) for 0x245e9e0:
upvalues (1) for 0x245e9e0:
  0 - 1 0
```

Listing 4.5: Constant Propagation

```
main <?:0,0> (12 instructions at 0xc8e9e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 3 constants, 1 function
  1 [-] LOADNIL  0 0
  2 [-] CLOSURE  1 0 ; 0xc8eb00
  3 [-] LOADK   2 -1 ; 0
  4 [-] LOADK   3 -2 ; 100000000
  5 [-] LOADK   4 -3 ; 1
```

```
  6 [-] FORPREP  2 3 ; to 10
  7 [-] MOVE    6 1
  8 [-] CALL    6 1 2
  9 [-] MOVE    0 6
  10  [-] FORLOOP  2 -4 ; to 7
  11  [-] RETURN  0 2
  12  [-] RETURN  0 1
constants (3) for 0xc8e9e0:
  1 0
  2 100000000
  3 1
locals (0) for 0xc8e9e0:
upvalues (1) for 0xc8e9e0:
  0 - 1 0

function <?:3,10> (11 instructions at 0xc8eb00)
0 params, 7 slots, 0 upvalues, 0 locals, 3 constants, 0 functions
  1 [-] LOADK   0 -1 ; 3
  2 [-] LOADK   1 -2 ; 4
  3 [-] LOADK   2 -3 ; 10
  4 [-] LOADK   3 -1 ; 3
  5 [-] LOADK   4 -3 ; 10
  6 [-] ADD     5 0 1
  7 [-] MUL     6 2 3
  8 [-] DIV     6 6 4
  9 [-] SUB     5 5 6
  10  [-] RETURN  5 2
  11  [-] RETURN  0 1
constants (3) for 0xc8eb00:
  1 3
  2 4
  3 10
locals (0) for 0xc8eb00:
upvalues (0) for 0xc8eb00:
```

Listing 4.6: Without Constant Propagation

```
main <?:0,0> (11 instructions at 0xd299e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 3 constants, 0 functions
  1 [-] LOADNIL  0 0
  2 [-] LOADK   2 -1 ; 0
  3 [-] LOADK   3 -2 ; 100000000
  4 [-] LOADK   4 -3 ; 1
  5 [-] FORPREP  2 3 ; to 9
  6 [-] LOADK   12 -1 ; 0
  7 [-] MOVE    6 12
  8 [-] MOVE    0 6
  9 [-] FORLOOP  2 -4 ; to 6
  10  [-] RETURN  0 2
  11  [-] RETURN  0 1
constants (3) for 0xd299e0:
  1 0
  2 100000000
  3 1
locals (0) for 0xd299e0:
```

```
upvalues (1) for 0xd299e0:
  0 - 1 0
```

Listing 4.7: Function Inlining

```
main <?:0,0> (21 instructions at 0x11a69e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 6 constants, 1 function
  1 [-] LOADNIL  0 0
  2 [-] CLOSURE  1 0 ; 0x11a6b50
  3 [-] LOADK    2 -1 ; 0
  4 [-] LOADK    3 -2 ; 100000000
  5 [-] LOADK    4 -3 ; 1
  6 [-] FORPREP  2 12 ; to 19
  7 [-] MOVE     6 1
  8 [-] LOADK    7 -4 ; 3
  9 [-] LOADK    8 -5 ; 4
  10 [-] LOADK   9 -6 ; 10
  11 [-] LOADK   10 -4 ; 3
  12 [-] LOADK   11 -6 ; 10
  13 [-] ADD     12 7 8
  14 [-] MUL     13 9 10
  15 [-] DIV     13 13 11
  16 [-] SUB     12 12 13
  17 [-] MOVE    6 12
  18 [-] MOVE    0 6
  19 [-] FORLOOP  2 -13 ; to 7
  20 [-] RETURN  0 2
  21 [-] RETURN  0 1
constants (6) for 0x11a69e0:
  1 0
  2 100000000
  3 1
  4 3
  5 4
  6 10
locals (0) for 0x11a69e0:
upvalues (1) for 0x11a69e0:
  0 - 1 0

function <?:3,10> (11 instructions at 0x11a6b50)
0 params, 7 slots, 0 upvalues, 0 locals, 3 constants, 0 functions
  1 [-] LOADK    0 -1 ; 3
  2 [-] LOADK    1 -2 ; 4
  3 [-] LOADK    2 -3 ; 10
  4 [-] LOADK    3 -1 ; 3
  5 [-] LOADK    4 -3 ; 10
  6 [-] ADD      5 0 1
  7 [-] MUL      6 2 3
  8 [-] DIV      6 6 4
  9 [-] SUB      5 5 6
  10 [-] RETURN  5 2
  11 [-] RETURN  0 1
constants (3) for 0x11a6b50:
  1 3
  2 4
```

```
  3 10
locals (0) for 0x11a6b50:
upvalues (0) for 0x11a6b50:
```

Listing 4.8: Without Funciton Inlining

```
main <?:0,0> (12 instructions at 0x25b19e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 3 constants, 1 function
  1 [-] LOADNIL  0 0
  2 [-] CLOSURE  1 0 ; 0x25b1b00
  3 [-] LOADK    2 -1 ; 0
  4 [-] LOADK    3 -2 ; 100000000
  5 [-] LOADK    4 -3 ; 1
  6 [-] FORPREP  2 3 ; to 10
  7 [-] MOVE     6 1
  8 [-] CALL     6 1 2
  9 [-] MOVE     0 6
  10  [-] FORLOOP  2 -4 ; to 7
  11  [-] RETURN  0 2
  12  [-] RETURN  0 1
constants (3) for 0x25b19e0:
  1 0
  2 100000000
  3 1
locals (0) for 0x25b19e0:
upvalues (1) for 0x25b19e0:
  0 - 1 0

function <?:3,10> (3 instructions at 0x25b1b00)
0 params, 7 slots, 0 upvalues, 0 locals, 1 constant, 0 functions
  1 [-] LOADK    5 -1 ; 0
  2 [-] RETURN  5 2
  3 [-] RETURN  0 1
constants (1) for 0x25b1b00:
  1 0
locals (0) for 0x25b1b00:
upvalues (0) for 0x25b1b00:
```

Listing 4.9: Dead-Code Elimination

```
main <?:0,0> (12 instructions at 0x15819e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 3 constants, 1 function
  1 [-] LOADNIL  0 0
  2 [-] CLOSURE  1 0 ; 0x1581b00
  3 [-] LOADK    2 -1 ; 0
  4 [-] LOADK    3 -2 ; 100000000
  5 [-] LOADK    4 -3 ; 1
  6 [-] FORPREP  2 3 ; to 10
  7 [-] MOVE     6 1
  8 [-] CALL     6 1 2
  9 [-] MOVE     0 6
  10  [-] FORLOOP  2 -4 ; to 7
  11  [-] RETURN  0 2
  12  [-] RETURN  0 1
constants (3) for 0x15819e0:
```

```
  1 0
  2 100000000
  3 1
locals (0) for 0x15819e0:
upvalues (1) for 0x15819e0:
  0 - 1 0

function <?:3,10> (11 instructions at 0x1581b00)
0 params, 7 slots, 0 upvalues, 0 locals, 3 constants, 0 functions
  1 [-] LOADK   0 -1 ; 3
  2 [-] LOADK   1 -2 ; 4
  3 [-] LOADK   2 -3 ; 10
  4 [-] LOADK   3 -1 ; 3
  5 [-] LOADK   4 -3 ; 10
  6 [-] ADD     5 0 1
  7 [-] MUL     6 2 3
  8 [-] DIV     6 6 4
  9 [-] SUB     5 5 6
  10 [-] RETURN  5 2
  11 [-] RETURN  0 1
constants (3) for 0x1581b00:
  1 3
  2 4
  3 10
locals (0) for 0x1581b00:
upvalues (0) for 0x1581b00:
```

Listing 4.10: Without Dead-Code Elimination

```
main <?:0,0> (21 instructions at 0x201c9e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 8 constants, 1 function
  1 [-] LOADNIL  0 0
  2 [-] CLOSURE  1 0 ; 0x201cb70
  3 [-] LOADK   2 -1 ; 0
  4 [-] LOADK   3 -2 ; 100000000
  5 [-] LOADK   4 -3 ; 1
  6 [-] FORPREP  2 12 ; to 19
  7 [-] MOVE    6 1
  8 [-] LOADK   7 -4 ; 3
  9 [-] LOADK   8 -5 ; 4
  10 [-] LOADK   9 -6 ; 10
  11 [-] LOADK   10 -4 ; 3
  12 [-] LOADK   11 -6 ; 10
  13 [-] LOADK   12 -7 ; 7
  14 [-] LOADK   13 -8 ; 30
  15 [-] LOADK   13 -4 ; 3
  16 [-] LOADK   12 -1 ; 0
  17 [-] LOADK   6 -1 ; 0
  18 [-] LOADK   0 -1 ; 0
  19 [-] FORLOOP  2 -13 ; to 7
  20 [-] RETURN  0 2
  21 [-] RETURN  0 1
constants (8) for 0x201c9e0:
  1 0
  2 100000000
```

```
  3 1
  4 3
  5 4
  6 10
  7 7
  8 30
locals (0) for 0x201c9e0:
upvalues (1) for 0x201c9e0:
  0 - 1 0

function <?:3,10> (11 instructions at 0x201cb70)
0 params, 7 slots, 0 upvalues, 0 locals, 6 constants, 0 functions
  1 [-] LOADK   0 -1 ; 3
  2 [-] LOADK   1 -2 ; 4
  3 [-] LOADK   2 -3 ; 10
  4 [-] LOADK   3 -1 ; 3
  5 [-] LOADK   4 -3 ; 10
  6 [-] LOADK   5 -4 ; 7
  7 [-] LOADK   6 -5 ; 30
  8 [-] LOADK   6 -1 ; 3
  9 [-] LOADK   5 -6 ; 0
  10  [-] RETURN  5 2
  11  [-] RETURN  0 1
constants (6) for 0x201cb70:
  1 3
  2 4
  3 10
  4 7
  5 30
  6 0
locals (0) for 0x201cb70:
upvalues (0) for 0x201cb70:
```

Listing 4.11: Unreachable Block Removal

```
main <?:0,0> (12 instructions at 0x22359e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 3 constants, 1 function
  1 [-] LOADNIL  0 0
  2 [-] CLOSURE  1 0 ; 0x2235b00
  3 [-] LOADK   2 -1 ; 0
  4 [-] LOADK   3 -2 ; 100000000
  5 [-] LOADK   4 -3 ; 1
  6 [-] FORPREP  2 3 ; to 10
  7 [-] MOVE    6 1
  8 [-] CALL    6 1 2
  9 [-] MOVE    0 6
  10  [-] FORLOOP  2 -4 ; to 7
  11  [-] RETURN  0 2
  12  [-] RETURN  0 1
constants (3) for 0x22359e0:
  1 0
  2 100000000
  3 1
locals (0) for 0x22359e0:
upvalues (1) for 0x22359e0:
```

```
  0 - 1 0

function <?:3,10> (11 instructions at 0x2235b00)
0 params, 7 slots, 0 upvalues, 0 locals, 3 constants, 0 functions
  1 [-] LOADK   0 -1 ; 3
  2 [-] LOADK   1 -2 ; 4
  3 [-] LOADK   2 -3 ; 10
  4 [-] LOADK   3 -1 ; 3
  5 [-] LOADK   4 -3 ; 10
  6 [-] ADD     5 0 1
  7 [-] MUL     6 2 3
  8 [-] DIV     6 6 4
  9 [-] SUB     5 5 6
  10  [-] RETURN  5 2
  11  [-] RETURN  0 1
constants (3) for 0x2235b00:
  1 3
  2 4
  3 10
locals (0) for 0x2235b00:
upvalues (0) for 0x2235b00:
```

Listing 4.12: Without Unreachable Block Removal

```
main <?:0,0> (9 instructions at 0x14379e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 3 constants, 0 functions
  1 [-] LOADNIL  0 0
  2 [-] LOADK   2 -1 ; 0
  3 [-] LOADK   3 -2 ; 100000000
  4 [-] LOADK   4 -3 ; 1
  5 [-] FORPREP  2 1 ; to 7
  6 [-] LOADK   0 -1 ; 0
  7 [-] FORLOOP  2 -2 ; to 6
  8 [-] RETURN  0 2
  9 [-] RETURN  0 1
constants (3) for 0x14379e0:
  1 0
  2 100000000
  3 1
locals (0) for 0x14379e0:
upvalues (1) for 0x14379e0:
  0 - 1 0
```

Listing 4.13: Full Optimization

```
main <?:0,0> (9 instructions at 0xb1e9e0)
0+ params, 7 slots, 1 upvalue, 0 locals, 3 constants, 0 functions
  1 [-] LOADNIL  0 0
  2 [-] LOADK   2 -1 ; 0
  3 [-] LOADK   3 -2 ; 100000000
  4 [-] LOADK   4 -3 ; 1
  5 [-] FORPREP  2 1 ; to 7
  6 [-] LOADK   0 -1 ; 0
  7 [-] FORLOOP  2 -2 ; to 6
  8 [-] RETURN  0 2
```

```
  9 [-] RETURN  0 1
constants (3) for 0xb1e9e0:
  1 0
  2 100000000
  3 1
locals (0) for 0xb1e9e0:
upvalues (1) for 0xb1e9e0:
  0 - 1 0
```

Table 4.1 shows the time, the number of instructions and generated bytecode size taken to run various benchmarks for different optimizations. And the above instruction lists are generated from the optimizer with several parameters.

The optimizer can disable specific features (Listing A.21).

4.4 Analysis

Full optimization make the speed performance of the bytecode more than 10x higher.

Among them, the influence by constant folding is the largest (Listing 4.4). In addition to replacing the operation instruction with the immediate instruction, and the dead-code elimination removes the instruction which becomes unnecessary. For the size of bytecode, it is simple that more the number of instructions incleases the size is bigger. The case of constant folding is but different (Listing 4.3). This is that it makes additional constants and the constant list grows.

Next to it, dead-code elimination and function inlining make affects to the perfocmance (Listing 4.8, Listing 4.10). However, function inlining is double-edget thing, so it does not enought effect in it simple substance. This may be because the number of instructoins and the size of bytecode itself incleased by function inlining and they are not removed or “fold”ed (Listing 4.7).

While constant propagation is few effective when it works on its own, full optimization without it is is slower 2x (Listing 4.6).

They improve performance by working complementarily.

Chapter 5

Conclusions

I have implemented a optimizer for Lua VM bytecode, which resulted in considerable performance improvements as shown by benchmarks. Global dataflow analysis, Control Flow Graph and Define Use / Use Define Chain, make highly effect to optimizations. The optimization affects not only to the speedup, but the byecode size.

For the phase of optimization to read or write a bytecode, I have analysed the structure of the VM bytecode. It is not documented so that it was difficult to implement the reader and writer.

5.1 Future Work

I have implemented the optimizer, including bytecode reader and writer. However, there is a lot of room to improve.

5.1.1 The Implementation of Function Inlining

In some cases, function inlining fails to appropriately optimize a bytecode and the bytecode occurs segmentation fault. The reason is not clear, so it is necessary to research to the VM and debugging more and more.

5.1.2 Other Optimization Techniques

I implemented some optimization techniques, but there are so many other technuques [1] and they are also effective to the bytecode. For instance, loop unrolling is widely effective. The loop in the benchmark code (Listing 4.2) is essentially no effect and it may be removed.

5.1.3 Optimization for The Optimizer

Some algorithms in the implementation is a little too raugh, and the execution speed of the optimizer itself is slow. Opeth provides a function to optimize at runtime.

```
optimizer = require'opeth.opeth'

f = -> ..... -- target function
g = optimizer f -- optimized function
g! -- run faster than `g`
```

The optimization for the optimizer itself make benefit to the runtime optimization.

Bibliography

[1] Nullstone Corporation. Compiler optimizations. http://www.compileroptimizations.com/.

[2] Jason D. Davies. Optimizing lua, 2005. https://www.jasondavies.com/optimising-lua/JasonDaviesDissertation.pdf.

[3] Ikuo Tanaka, Masataka Sasa, Munahiro Takimoto, and Tan Watanabe. コンパイラの基盤技術と実践 – コンパイラ・インフラストラクチャ COINS を用いて. 2008.

[4] Dibyendu Majumdar. Lua 5.3 bytecode reference. http://the-ravi-programming-language.readthedocs.io/en/latest/lua_bytecode_reference.html.

[5] Kein-Hong Man. A no-frills introduction to lua 5.1 vm instructions, 2006. http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf.

[6] PUC Rio. source code for lua 5.3. https://www.lua.org/source/5.3/.

[7] Roberto Ierusalimuschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0, 2003. https://www.lua.org/doc/jucs05.pdf.

[8] Michael Schroder. Optimizing lua using run-time type specialization, 2012. https://www.complang.tuwien.ac.at/anton/praktika-fertig/schroeder/thesis.pdf.

Appendix A

Source Code of Opeth

Here is the code of this research implementation. Almost all are written in MoonScript, and a few in Lua.

A.1 Commom Modules

Listing A.1: `opeth/common/opname.lua`

```
MOVE = "MOVE"
LOADK = "LOADK"
LOADKX = "LOADKX"
LOADBOOL = "LOADBOOL"
LOADNIL = "LOADNIL"
GETUPVAL = "GETUPVAL"
GETTABUP = "GETTABUP"

GETTABLE = "GETTABLE"
SETTABUP = "SETTABUP"
SETUPVAL = "SETUPVAL"
SETTABLE = "SETTABLE"
NEWTABLE = "NEWTABLE"

SELF = "SELF"
ADD = "ADD"
SUB = "SUB"
MUL = "MUL"
MOD = "MOD"
POW = "POW"
DIV = "DIV"

IDIV = "IDIV"
BAND = "BAND"
BOR = "BOR"
BXOR = "BXOR"
SHL = "SHL"
SHR = "SHR"
UNM = "UNM"
BNOT = "BNOT"

NOT = "NOT"
LEN = "LEN"
```

```
CONCAT = "CONCAT"
JMP = "JMP"
EQ = "EQ"
LT = "LT"
LE = "LE"
TEST = "TEST"

TESTSET = "TESTSET"
CALL = "CALL"
TAILCALL = "TAILCALL"
RETURN = "RETURN"
FORLOOP = "FORLOOP"
FORPREP = "FORPREP"

TFORCALL = "TFORCALL"
TFORLOOP = "TFORLOOP"
SETLIST = "SETLIST"
CLOSURE = "CLOSURE"
VARARG = "VARARG"
EXTRAARG = "EXTRAARG"
```

Listing A.2: `opeth/common/utils.moon`

```
import concat from table
import char from string

--- utils
----{{{
zsplit = (n = 1) => [c for c in @\gmatch "."\rep n]
string = string -- in THIS chunk, add `zsplit` to `string` module
string.zsplit = zsplit

map = (fn, xs) -> [fn x for x in *xs]
filter = (fn, xs) -> [x for x in *xs when fn x]
foldl = (fn, xr, xs) ->
  for x in *xs
    xr = fn xr, x
  xr

idcomp = (obj1, obj2) -> (tostring obj1) == (tostring obj2)
have = (t, e) -> (filter (=> (idcomp @, e) or @ == e), t)[1]
delete = (t, v) -> table.remove t, i for i = 1, #t when (idcomp t[i], v) or t[i] ==
    v
last = => @[#@]
isk = (rk) -> rk < 0 and (rk % 256) != 0
cstid = (k) -> (math.abs k) % 256 + (k >= 0 and 1 or 0)

undecimal = do
  hexdecode = (cnt = 1) -> ("%02X"\rep cnt)\format

  -- `"ff"` -> `"11111111"`
  hextobin = do
    bintbl = {
      [0]: "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
```

```
      "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    }
    (hex) -> concat map (=> bintbl[(tonumber "0x#{@}")]), hex\zsplit!

  -- `"00011", 4` -> `"0011"`
  adjustdigit = (r, a) ->
    if #r > a
      r\match("#{'.'\rep (#r - a)}(.*)")
    else
      "0"\rep(a - #r) .. r

  -- `"11111111"` -> `256`
  bintoint = (bin) ->
    i = -1
    with ret = 0
      for c in bin\reverse!\gmatch"."
        i += 1
        ret += 2^i * math.tointeger c

  -- `"0xff"` -> `256`
  hextoint = (hex) -> tonumber hex, 16

  -- `"41"` -> `"A"`
  hextochar = (ahex) -> string.char tonumber "0x#{ahex}"

  bintohex = do
    b2htbl = {
      ["0000"]: "0", ["0001"]: "1", ["0010"]: "2", ["0011"]: "3",
      ["0100"]: "4", ["0101"]: "5", ["0110"]: "6", ["0111"]: "7",
      ["1000"]: "8", ["1001"]: "9", ["1010"]: "a", ["1011"]: "b",
      ["1100"]: "c", ["1101"]: "d", ["1110"]: "e", ["1111"]: "f"
    }
    (b) -> b2htbl[b]

  inttobin = => (hextobin "%x"\format @)

  {:hexdecode, :hextobin, :adjustdigit, :bintoint, :hextoint, :hextochar, :
      bintohex, :inttobin}

deepcpy = (t, list = {}) -> with ret = {}
  for k, v in pairs t
    if type(v) == "table"
      kk = tostring v

      unless list[kk]
        list[kk] = v
        ret[k] = deepcpy v, list
      else ret[k] = list[kk]
    else ret[k] = v

prerr = (ne, msg) -> not ne and io.stdout\write(msg , '\n')
----}}}

{:zsplit, :map, :filter, :foldl, :idcomp, :have, :delete, :last, :isk, :cstid, :
```

```
    prerr, :undecimal, :deepcpy}
```

Listing A.3: `opeth/common/blockrealm.moon`

```
import insert, remove, sort from table
import tointeger from math
import map, filter from require'opeth.common.utils'

local get_block

validly_insert = (t, v) ->
  unless v.start and v.end
    error "lack of block elements v.start: #{v.start}, v.end: #{v.end}"

  if v.start > v.end
    error "invalid block"

  unless #(filter (=> @start == v.start and @.end == v.end), t) > 0
    insert t, v
    map tointeger, {v.start, v.end}

    sort t, (a, b) -> a.end < b.start

-- shrink `blk` from `delimp` to `blk.end`,
-- and return new block `blk.start` to `delimp - 1`
split_block = (blk, delimp) ->
  with newblk = {start: blk.start, end: delimp - 1, succ: {blk}, pred: blk.pred}
    blk.start = delimp
    blk.pred = {newblk}

mkcfg = (instruction) ->
  blocks = {}

  for ins_idx = 1, #instruction
    singleblock = {start: ins_idx, end: ins_idx, succ: {}, pred: {}}
    {RA, RB, RC, :op} = instruction[ins_idx]

    singleblock.succ_pos = switch op
      when JMP, FORPREP then {ins_idx + RB + 1}
      when LOADBOOL then {ins_idx + 2} if RC == 1
      when TESTSET, TEST, LT, LE, EQ then {ins_idx + 1, ins_idx + 2}
      when FORLOOP, TFORLOOP then {ins_idx + 1, ins_idx + RB + 1}
      when RETURN, TAILCALL then {}

    validly_insert blocks, singleblock

  blk_idx = 1

  while blocks[blk_idx]
    blk = blocks[blk_idx]

    if blk.succ_pos
      while #blk.succ_pos > 0
        succ_pos = remove blk.succ_pos, 1
```

```
51
52        if blk_ = get_block instruction, succ_pos, blocks
53          if blk_.start < succ_pos
54            newblk = split_block blk_, succ_pos
55            validly_insert blocks, newblk
56            validly_insert blk_.pred, blk
57            validly_insert blk.succ, blk_
58          else
59            validly_insert blk_.pred, blk
60            validly_insert blk.succ, blk_
61        else
62          if #blk.succ_pos > 0
63            insert blk.succ_pos, succ_pos
64          else
65            error "cannot resolve succ_pos #{succ_pos} / ##{#instruction}"
66
67      blk.succ_pos = nil
68    elseif #blk.succ == 0
69      nextblock = blocks[blk_idx + 1]
70
71      if #nextblock.pred > 0
72        validly_insert nextblock.pred, blk
73        validly_insert blk.succ, nextblock
74      else
75        {:start, :pred} = blk
76        remove blocks, blk_idx
77        (for psucci = 1, #p.succ
78          if p.succ[psucci].start == start
79            remove p.succ, psucci
80            validly_insert p.succ, nextblock
81            break
82        ) for p in *pred
83
84        nextblock.start = start
85        nextblock.pred = pred
86        continue
87
88    blk_idx += 1
89
90  blocks
91
92get_block = (instruction, nth, blocks = mkcfg instruction) ->
93  return b for b in *blocks when ((b.start <= nth) and (b.end >= nth))
94
95:get_block, :mkcfg
```

Listing A.4: `opeth/common/oplist.lua`

```
1 if not RETURN then
2   require'opeth.common.opname'
3 end
4
5 return function(abc, abx, asbx, ax)
6   local t = {
```

```
    {MOVE, abc}, {LOADK, abx}, {LOADKX, abx}, {LOADBOOL, abc}, {LOADNIL, abc}, {
        GETUPVAL, abc}, {GETTABUP, abc},
    {GETTABLE, abc}, {SETTABUP, abc}, {SETUPVAL, abc}, {SETTABLE, abc}, {NEWTABLE,
        abc},
    {SELF, abc}, {ADD, abc}, {SUB, abc}, {MUL, abc}, {MOD, abc}, {POW, abc}, {DIV,
        abc},
    {IDIV, abc}, {BAND, abc}, {BOR, abc}, {BXOR, abc}, {SHL, abc}, {SHR, abc}, {UNM,
         abc}, {BNOT, abc},
    {NOT, abc}, {LEN, abc}, {CONCAT, abc}, {JMP, asbx}, {EQ, abc}, {LT, abc}, {LE,
        abc}, {TEST, abc},
    {TESTSET, abc}, {CALL, abc}, {TAILCALL, abc}, {RETURN, abc}, {FORLOOP, asbx}, {
        FORPREP, asbx},
    {TFORCALL, abc}, {TFORLOOP, asbx}, {SETLIST, abc}, {CLOSURE, abx}, {VARARG, abc}
        , {EXTRAARG, ax}
  }

  for i = 1, #t do
    t[i].idx = i
    -- table.insert(t[i], i)
    -- t[i][3] = i
  end

  for k, v in pairs(t) do
    t[v[1] ]= v
  end

  return t
end
```

A.2 Optimizer Modules

Listing A.5: `opeth/opeth/cst_fold.moon`

```
import rtype, rcst from require'opeth.opeth.common.constant'
import cst_lookup, cst_add, swapins from require'opeth.opeth.common.utils'
import du_chain from require'opeth.opeth.common.du_chain'
import insert, concat from table
optbl = require'opeth.opeth.common.optbl'

INF = 1 / 0
NAN = 0 / 0
isnan = => "-nan" == tostring @

(fnblock) ->
  du_cfg = du_chain fnblock

  registercst = (cst, ins_idx, ra) ->
    if cst != INF and (cst != -INF) and not isnan cst
      if cst_idx = cst_lookup fnblock.constant, cst
        swapins fnblock.instruction, ins_idx, {ra, cst_idx - 1, op: LOADK}
      else
        cst_add fnblock.constant, cst
```

```
      swapins fnblock.instruction, ins_idx, {ra, #fnblock.constant - 1, op: LOADK}

    du_cfg = du_chain fnblock
    fnblock.optdebug.modified += 1

  for ins_idx = 1, #fnblock.instruction
    {RA, RB, RC, :op} = fnblock.instruction[ins_idx]

    switch op
      when ADD, SUB, MUL, DIV, MOD, IDIV, BAND, BXOR, BOR, SHL, SHR, POW
        if (rtype fnblock, ins_idx, RB, du_cfg) == "number"
          if (rtype fnblock, ins_idx, RC, du_cfg) == "number"
            has_cst, cst = rcst fnblock, ins_idx, RA, du_cfg
            registercst cst, ins_idx, RA if has_cst
      when NOT
        switch (rtype fnblock, ins_idx, RA, du_cfg)
          when "bool"
            has_cst, cst = rcst fnblock, ins_idx, RB, du_cfg
            registercst cst, ins_idx, RA if has_cst
          when "string", "number"
            registercst false, ins_idx, RA
      when UNM
        switch rtype fnblock, ins_idx, RA, du_cfg
          when "number"
            has_cst, cst = rcst fnblock, ins_idx, RA, du_cfg
            registercst cst, ins_idx, RA if has_cst
      when LEN
        if (rtype fnblock, ins_idx, RB, du_cfg) == "string"
          has_cst, len = rcst fnblock, ins_idx, RA, du_cfg
          registercst len, ins_idx, RA if has_cst
      when CONCAT
        has_cst, cst = rcst fnblock, ins_idx, RA, du_cfg
        registercst cst, ins_idx, RA if has_cst
```

Listing A.6: `opeth/opeth/cst_prop.moon`

```
import filter from require'opeth.common.utils'
import rtype, rcst from require'opeth.opeth.common.constant'
import cst_lookup, cst_add, removeins, swapins from require'opeth.opeth.common.
    utils'
import get_block from require'opeth.common.blockrealm'
import du_chain, this_def from require'opeth.opeth.common.du_chain'

(fnblock) ->
  fnblock.optdebug\start_rec!

  du_cfg = du_chain fnblock
  ins_idx = 1

  hoisting = (to_idx, from_ins, ra) ->
    fnblock.instruction[to_idx] = {ra, from_ins[2], from_ins[3], op: from_ins.op}
    fnblock.optdebug.modified += 1
    du_cfg = du_chain fnblock

```

```
18    while fnblock.instruction[ins_idx]
19      ins = fnblock.instruction[ins_idx]
20      {RA, RB, RC, :op} = ins
21
22      if op == MOVE
23        if RA == RB
24          removeins fnblock.instruction, ins_idx
25          fnblock.optdebug.modified += 1
26          du_cfg = du_chain fnblock
27          continue
28
29        blk = get_block nil, ins_idx, du_cfg
30
31        if d_rb = this_def blk, ins_idx, RB
32          if #d_rb.used == 1 and #d_rb.used[1].defined == 1
33            moved_idx = d_rb.line
34            if pins = fnblock.instruction[moved_idx]
35              {pRA, pRB, pRC, op: pop} = pins
36
37              switch pop
38                -- when ADD, SUB, MUL, DIV, MOD, IDIV, BAND, BXOR, BOR, SHL, SHR, POW
39                  -- typeRB = rtype fnblock, moved_idx, pRB, du_cfg
40                  -- typeRC = rtype fnblock, moved_idx, pRB, du_cfg
41
42                  -- if typeRB == "number" and typeRC == "number"
43                    -- if d_rb.def
44                      -- if #(filter (=> (@reg == pRB or @reg == pRC) and moved_idx <
                            @line and @line < ins_idx), d_rb.def) == 0
45                        -- hoisting ins_idx, pins, RA
46                when MOVE
47                  if d_rb.def and #(filter (=> @reg == pRB and moved_idx < @line and
                        @line < ins_idx), d_rb.def) == 0
48                    hoisting ins_idx, pins, RA
49                when LOADK
50                  hoisting ins_idx, pins, RA
51
52                -- TODO: consider of closed variables
53                -- when CLOSURE
54                  -- hoisting fnblock, ins_idx, moved_idx, pins, RA
55
56                  -- proto = fnblock.prototype[pRB + 1]
57
58                  -- for u in *proto.upvalue
59                    -- if u.instack == 1 and u.reg == pRA
60                      -- u.reg = RA
61
62      ins_idx += 1
```

Listing A.7: `opeth/opeth/func_inline.moon`

```
1  import cst_lookup, cst_add, insertins, removeins, swapins, adjust_jump_pos_down,
       adjust_jump_pos_up from require'opeth.opeth.common.utils'
2  import du_chain, root_def, this_def from require'opeth.opeth.common.du_chain'
3  import get_block from require'opeth.common.blockrealm'
```

```
import undecimal, deepcpy, cstid from require'opeth.common.utils'
import hextoint from undecimal

trace_MOVE = (instruction, n, du_cfg) ->
  switch instruction[n].op
    when MOVE
      if blk = get_block nil, n, du_cfg
        if moved = this_def blk, n, instruction[n][2]
          trace_MOVE instruction, moved.line, du_cfg
    when CLOSURE then n

max_reg = (instruction, pos) ->
  with maxn = 0 do for i = 1, pos
    {RA} = instruction[i]
    maxn = math.max maxn, RA

is_recursive = (fnblock, clos_ins) ->
  proto = fnblock.prototype[clos_ins[2] + 1]

  with bool = false
    for pu in *proto.upvalue
      bool or= pu.instack == 1 and pu.reg == clos_ins[1]

lookup_upvalue_index = (upvlist, upvalue) ->
  for i = 1, #upvlist
    if upvlist[i].reg == upvalue.reg and upvlist[i].instack == upvalue.instack
      return i

(fnblock) ->
  du_cfg = du_chain fnblock
  ins_idx = 1

  while ins_idx <= #fnblock.instruction
    ins = fnblock.instruction[ins_idx]
    {RA, RB, RC, :op} = ins

    switch op
      when CALL
        blk = get_block nil, ins_idx, du_cfg

        unless blk.start < ins_idx
          ins_idx += 1
          continue

        if d_ra = this_def blk, ins_idx - 1, RA
          if clos_idx = trace_MOVE fnblock.instruction, d_ra.line, du_cfg
            proto_idx = fnblock.instruction[clos_idx][2] + 1

            if proto = deepcpy fnblock.prototype[proto_idx]
              if (hextoint proto.regnum) + (hextoint fnblock.regnum) < 256 and not
                  is_recursive fnblock, fnblock.instruction[clos_idx]
                params = hextoint proto.params

                -- #arg for the closure
```

```
57                argnum = RA + RB - 2
58
59                cst_transfer = (prev_ins, rx) ->
60                  positive = (prev_ins[rx]) >= 0
61                  cst = proto.constant[cstid prev_ins[rx] ].val
62                  prev_ins[rx] = if cidx = cst_lookup fnblock.constant, cst
63                    positive and cidx - 1 or -cidx
64                  else
65                    cidx = cst_add fnblock.constant, cst
66                    positive and cidx - 1 or -cidx
67
68                proto_ins_idx = 1
69                OFFS = (RB == 0 and ((max_reg fnblock.instruction, ins_idx) + 2) or (
                      RA + RB)) - params
70                modifiable = true
71                jmp_store = {}
72
73                while proto_ins_idx <= #proto.instruction
74                  prev_ins = proto.instruction[proto_ins_idx]
75                  {pRA, pRB, pRC, op: prev_op} = prev_ins
76
77                  switch prev_op
78                    when LOADK, GETGLOBAL, SETGLOBAL
79                      prev_ins[1] += OFFS
80                      cst_transfer prev_ins, 2
81                    when MOVE, UNM, NOT, LEN, TESTSET
82                      prev_ins[1] += OFFS
83                      prev_ins[2] += OFFS
84                    when LOADNIL
85                      prev_ins[1] += OFFS
86                      prev_ins[2] += OFFS if pRB > 0
87                    when ADD, SUB, MUL, MOD, POW, DIV, IDIV, BAND, BOR, BXOR, SHL,
                          SHR, SETTABLE
88                      prev_ins[1] += OFFS
89
90                      if pRB < 0 then cst_transfer prev_ins, 2
91                      else prev_ins[2] += OFFS
92
93                      if pRC < 0 then cst_transfer prev_ins, 3
94                      else prev_ins[3] += OFFS
95                    when GETUPVAL
96                      prev_upv = proto.upvalue[pRB + 1]
97
98                      if prev_upv.instack == 0
99                        if fnblock.upvalue[prev_upv.reg + 1]
100                         prev_ins[1] += OFFS
101                         prev_ins[2] = prev_upv.reg
102                       else
103                         modifiable = false
104                         break
105                     else
106                       if def = root_def blk, ins_idx, prev_upv.reg
107                         swapins proto.instruction, proto_ins_idx, {pRA + OFFS, def.
                              reg, 0, op: MOVE}
```

```
108                     else
109                       modifiable = false
110                       break
111                   when GETTABUP
112                     prev_upv = proto.upvalue[pRB + 1]
113
114                     if pRC < 0
115                       cst_transfer prev_ins, 3
116                     else
117                       prev_ins[3] += OFFS
118
119                     if prev_upv.instack == 0
120                       if fnblock.upvalue[prev_upv.reg + 1]
121                         prev_ins[1] += OFFS
122                         prev_ins[2] = prev_upv.reg
123                       else
124                         modifiable = false
125                         break
126                     else swapins proto.instruction, proto_ins_idx, {pRA + OFFS,
                            prev_upv.reg + OFFS, prev_ins[3], op: GETTABLE}
127                   when SETUPVAL
128                     prev_upv = proto.upvalue[pRA + 1]
129
130                     if prev_upv.instack == 0
131                       modifiable = false
132                       break
133
134                     swapins proto.instruction, proto_ins_idx, {prev_upv.reg, pRB +
                          OFFS, op: MOVE}
135                   when EQ, LT, LE
136                     if pRB < 0 then cst_transfer prev_ins, 2
137                     else prev_ins[2] += OFFS
138
139                     if pRC < 0 then cst_transfer prev_ins, 3
140                     else prev_ins[3] += OFFS
141                   when GETTABLE, SELF
142                     prev_ins[1] += OFFS
143                     prev_ins[2] += OFFS
144
145                     if pRC < 0 then cst_transfer prev_ins, 3
146                     else prev_ins[3] += OFFS
147                   when LOADBOOL, CLOSURE, CALL, FORPREP, FORLOOP, TFORLOOP,
                        TFORCALL, TEST, NEWTABLE
148                     prev_ins[1] += OFFS
149                   when JMP
150                     _ = 0 -- skip
151                   when RETURN
152                     nextins = fnblock.instruction[ins_idx + 1]
153
154                     -- the number of return values
155                     if nextins.op == CALL
156                       if pRB == 1 then nextins[2] = 1
157                       elseif pRB > 1 then nextins[2] = pRB
158
```

```
159                   removeins proto.instruction, proto_ins_idx
160                   proto_ins_idx -= 1
161
162                   if RC != 1 and pRB != 1
163                     movelimit = pRB == 0 and (max_reg proto.instruction,
                            proto_ins_idx) or pRA + pRB - 2
164
165                     for moved_reg = movelimit, pRA, -1
166                       moveRA = RA + moved_reg - pRA -- register for caller to put
                              the return value
167                       insertins proto.instruction, proto_ins_idx + 1, {moveRA,
                              moved_reg + OFFS, 0, op: MOVE}
168                       proto_ins_idx += 1
169
170                     proto_ins_idx += 1
171                   elseif proto_ins_idx > 0
172                     insertins proto.instruction, proto_ins_idx, {RA, RA + RC - 1,
                            op: LOADNIL}
173                     proto_ins_idx += 1
174
175                   if proto_ins_idx < #proto.instruction - 1
176                     jmp = {proto_ins_idx, 0, op: JMP}
177                     insertins proto.instruction, proto_ins_idx, jmp
178                     proto_ins_idx += 1
179                     table.insert jmp_store, jmp
180                   else
181                     break
182                 when EXTRAARG
183                   cst_transfer prev_ins, 1
184                 when TAILCALL
185                   modifiable = false
186                   break
187
188               proto_ins_idx += 1
189
190             if modifiable
191               -- remove CALL from main
192               removeins fnblock.instruction, ins_idx
193               fnblock.optdebug.modified += 1
194               proto_ins_idx -= 1
195
196               for jmp in *jmp_store
197                 jmp[2] = proto_ins_idx - jmp[1] - 1
198                 jmp[1] = 0
199
200               for pii = 1, proto_ins_idx
201                 insertins fnblock.instruction, ins_idx + pii - 1, proto.
                        instruction[pii], true
202                 fnblock.optdebug.modified += 1
203
204               adjust_jump_pos_up fnblock.instruction, ins_idx, nil, proto_ins_idx
205               adjust_jump_pos_down fnblock.instruction, ins_idx + proto_ins_idx,
                      nil, proto_ins_idx
206
```

```
                ins_idx += 1
                du_cfg = du_chain fnblock

    ins_idx += 1
```

Listing A.8: `opeth/opeth/dead_elim.moon`

```
import rtype, rcst from require'opeth.opeth.common.constant'
import foldl from require'opeth.common.utils'
import removeins, swapins from require'opeth.opeth.common.utils'
import get_block from require'opeth.common.blockrealm'
import du_chain, root_def, this_def from require'opeth.opeth.common.du_chain'
optbl = require'opeth.opeth.common.optbl'

xor = (p, q) -> (p or q) and not (p and q)

(fnblock) ->
  du_cfg = du_chain fnblock
  ins_idx = 1

  proc_rm = (ins_idx) =>
    removeins fnblock.instruction, ins_idx
    ins_idx -= 1
    fnblock.optdebug\mod_inc!
    du_cfg = du_chain fnblock

  while fnblock.instruction[ins_idx]
    ins = fnblock.instruction[ins_idx]
    {RA, RB, RC, :op} = ins

    switch op
      when LOADK, CLOSURE
        blk = get_block nil, ins_idx, du_cfg

        -- if blk.start != blk.end
        if d_ra = this_def blk, ins_idx, RA
          if d_ra.used == nil or #d_ra.used == 0
            -- print ins_idx + fnblock.optdebug.modified, RA, RB
            swapins fnblock.instruction, ins_idx, {RA, RA, 0, op: MOVE}
            ins_idx -= 1
            fnblock.optdebug\mod_inc!
            du_cfg = du_chain fnblock
            -- proc_rm fnblock, ins_idx
            continue
      when MOVE
        if RA == RB
          proc_rm fnblock, ins_idx
          continue
        else
          blk = get_block nil, ins_idx, du_cfg

          -- if blk.start != blk.end
          if d_ra = this_def blk, ins_idx, RA
            if d_ra.used == nil or #d_ra.used == 0
```

```
48              if d_rb = root_def blk, ins_idx, RB
49                if d_rb.line > 0 and
50                    not foldl ((bool, op) -> bool or op == fnblock.instruction[d_rb.
                          line].op),
51                      false, {GETTABUP, GETTABLE, CALL}
52                  proc_rm fnblock, ins_idx
53                  continue
54        when LOADNIL
55          blk = get_block nil, ins_idx, du_cfg
56
57          if blk.start != blk.end
58            if #[u for u in *blk.def when u.line == ins_idx and #u.used > 0] == 0
59              proc_rm fnblock, ins_idx
60              continue
61        -- when LOADBOOL
62          -- blk = get_block nil, ins_idx, du_cfg
63
64          -- if #blk.pred == 0
65            -- proc_rm fnblock, ins_idx
66            -- continue
67          -- else
68            -- sscope = get_block nil, ins_idx + 1, du_cfg
69            -- if #sscope.pred == 0
70              -- proc_rm fnblock, ins_idx
71              -- ins[3] = 0 if RC == 1
72              -- continue
73        when FORLOOP
74          -- empty forloop
75          if RB == -1 and fnblock.instruction[ins_idx - 1].op == FORPREP
76            proc_rm fnblock, ins_idx - 1
77            proc_rm fnblock, ins_idx - 1
78            continue
79        -- iterator function call must not be removed
80        -- when TFORLOOP
81        -- when JMP
82          -- proc_rm fnblock, ins_idx if RA == 0 and RB == 0
83        when LT, LE, EQ
84          if "number" == rtype fnblock, ins_idx, RB, du_cfg
85            if "number" == rtype fnblock, ins_idx, RC, du_cfg
86              has_cstRB, cstRB = rcst fnblock, ins_idx, RB, du_cfg
87
88              if has_cstRB
89                has_cstRC, cstRC = rcst fnblock, ins_idx, RC, du_cfg
90
91                if has_cstRC
92                  cond = (RA == 1) != optbl[ins.op] cstRB, cstRC
93
94                  proc_rm fnblock, ins_idx
95                  proc_rm fnblock, ins_idx if cond
96                  continue
97        when TEST
98          typeRA = rtype fnblock, ins_idx, RA, du_cfg
99          cond = switch typeRA
100           when nil, "table", "userdata"
```

```
          ins_idx += 1
          continue
        when "bool"
          has_cstRA, cstRA = rcst fnblock, ins_idx, RA, du_cfg

          unless has_cstRA
            ins_idx += 1
            continue

          cstRA
        when "nil" then false
        else true

      proc_rm fnblock, ins_idx
      -- if cond then pc++
      proc_rm fnblock, ins_idx if xor (RC != 0), cond -- RC ~= 0 and (not cond) or
          cond
      continue
    when TESTSET
      typeRB = rtype fnblock, ins_idx, RB, du_cfg
      cond = switch typeRB
        when nil, "table", "userdata"
          ins_idx += 1
          continue
        when "bool"
          has_cstRB, cstRB = rcst fnblock, ins_idx, RB, du_cfg

          unless has_cstRB
            ins_idx += 1
            continue

          cstRB
        when "nil" then false
        else true

      proc_rm fnblock, ins_idx

      unless xor (RC != 0), cond
        swapins fnblock.instruction, ins_idx, {RA, RB, 0, op: MOVE}
        du_cfg = du_chain fnblock
        fnblock.optdebug\mod_inc!
        proc_rm fnblock, ins_idx + 1

      continue

  ins_idx += 1
```

Listing A.9: `opeth/opeth/unreachable_remove.moon`

```
import removeins from require'opeth.opeth.common.utils'
import mkcfg from require'opeth.common.blockrealm'

(fnblock) ->
  for cfg in *(mkcfg fnblock.instruction)
```

```
    -- unreachable? the block, the beggining of which line is greater than 1
    --  and doesn't have the predecessive blocks
    start = cfg.start
    if start > 1 and #cfg.pred == 0

      if #fnblock.instruction < start then break
      if start == cfg.end then continue

      for _ = start, cfg.end
        switch fnblock.instruction[start].op
          when LOADBOOL
            if fnblock.instruction[start - 1].op == LOADBOOL
              fnblock.instruction[start - 1][3] = 0
              break
          when JMP
            break if fnblock.instruction[start][1] > 0

        removeins fnblock.instruction, start

      fnblock.optdebug.modified += cfg.end - start + 1
      break -- :)
```

Listing A.10: `opeth/opeth/unused_remove.moon`

```
import map, filter, isk from require'opeth.common.utils'
import remove from table

(fnblock) ->
  -- clean unused closures
  closdef = filter (=> @op == CLOSURE), fnblock.instruction
  closidx = 0

  while fnblock.prototype[closidx + 1]
    unless (filter (=> @[2] == closidx), closdef)[1]
      remove fnblock.prototype, closidx + 1
      fnblock.optdebug\mod_inc!
      map (=> @[2] -= 1), filter (=> @[2] >= closidx), closdef
      continue

    closidx += 1

  -- clean unused constants
  cstidx = 0

  while fnblock.constant[cstidx + 1]
    unless (filter (=> switch @op
        when EXTRAARG then @[1] == cstidx
        when LOADK, GETGLOBAL, SETGLOBAL then @[2] == cstidx
        when GETTABLE, SELF, GETTABUP then (isk @[3]) and (@[3] == -(cstidx + 1))
        when ADD, SUB, MUL, DIV, MOD, IDIV, BAND, BXOR, BOR, SHL, SHR, POW, EQ, LT,
            LE, SETTABLE, SETTABUP
          ((isk @[2]) and (@[2] == -(cstidx + 1))) or
            ((isk @[3]) and (@[3] == -(cstidx + 1)))
        ), fnblock.instruction)[1]
```

```
      remove fnblock.constant, cstidx + 1
      fnblock.optdebug\mod_inc!
      map (=> switch @op
        when EXTRAARG then @[1] -= 1 if @[1] >= cstidx
        when LOADK then @[2] -= 1 if @[2] >= cstidx
        when GETTABLE, SELF, GETTABUP then @[3] += 1 if (isk @[3]) and @[3] <
            -cstidx
        when ADD, SUB, MUL, DIV, MOD, IDIV, BAND, BXOR, BOR, SHL, SHR, POW, EQ, LT,
            LE, SETTABLE, SETTABUP
          if (isk @[2]) and @[2] < -cstidx
            @[2] += 1

          if (isk @[3]) and @[3] < -cstidx
            @[3] += 1
        ), fnblock.instruction
      continue

    cstidx += 1
```

A.3 Common Modules for Optimizers

Listing A.11: `opeth/opeth/common/utils.moon`

```
oplist = require'opeth.common.oplist'!

cst_lookup = (constant, cst) ->
  for i = 1, #constant do if constant[i].val == cst then return i

v2typ = (cst) ->
  switch type cst
    when "number"
      math.type(cst) == "integer" and 0x13 or 0x3
    when "string"
      #cst > 255 and 0x14 or 0x4

cst_add = (constant, cst) ->
  with idx = #constant + 1
    constant[idx] = {type: v2typ(cst), val: cst}

-- adjust_jump_pos = (instruction, ins_idx, is_remove) ->
  -- for j = 1, #instruction
adjust_jump_pos_core = (j, instruction, ins_idx, is_remove, plus = 1) ->
  jins = instruction[j]
  error "#{j} / #{#instruction}", 4 unless jins
  jRB = jins[2]

  switch jins.op
    when JMP, FORPREP
      if is_remove
        if (j < ins_idx and j + jRB + 1 > ins_idx)
          jins[2] -= plus
        elseif (j > ins_idx and j + jRB + 1 < ins_idx)
```

```
30          jins[2] += plus
31        else
32          if (j < ins_idx + 1 and j + jRB >= ins_idx)
33            jins[2] += plus
34          elseif (j > ins_idx + 1 and j + jRB + 1 <= ins_idx)
35            jins[2] -= plus
36      when FORLOOP, TFORLOOP
37        if j >= ins_idx and j + jRB + 1 <= ins_idx
38          jins[2] -= is_remove and -plus or plus
39
40  adjust_jump_pos_down = (instruction, ins_idx, is_remove, plus) ->
41    for j = ins_idx, #instruction
42      adjust_jump_pos_core j, instruction, ins_idx, is_remove, plus
43
44  adjust_jump_pos_up = (instruction, ins_idx, is_remove, plus) ->
45    for j = ins_idx, 1, -1
46      adjust_jump_pos_core j, instruction, ins_idx, is_remove, plus
47
48  adjust_jump_pos = (instruction, ins_idx, is_remove, plus) ->
49    for i = 1, #instruction
50      adjust_jump_pos_core i, instruction, ins_idx, is_remove, plus
51
52  insertins = (instruction, ins_idx, ins, is_unchanged_pos) ->
53    assert ((type ins[1]) == (type ins[2])) and
54      ((type ins[1]) == "number") and
55      (ins[3] and ((type ins[3]) == "number") or true),
56      "insertins #3: invalid instruction `#{ins.op} #{ins[1]} #{ins[2]} #{ins[3] and
          ins[3] or ""}`"
57
58    assert oplist[ins.op], "insertins #3: invalid op `#{ins.op}'"
59
60    assert instruction[ins_idx],
61      "insertins #2: attempt to insert out of range of the instructions (#{ins_idx} /
          #{#instruction})"
62
63    table.insert instruction, ins_idx, ins
64    adjust_jump_pos instruction, ins_idx unless is_unchanged_pos
65
66  removeins = (instruction, ins_idx, is_unchanged_pos) ->
67    assert instruction[ins_idx],
68      "removeins #2: attempt to remove out of range of the instructions (#{ins_idx} /
          #{#instruction})"
69
70    table.remove instruction, ins_idx
71    adjust_jump_pos instruction, ins_idx, true unless is_unchanged_pos
72
73  swapins = (instruction, ins_idx, ins) ->
74    removeins instruction, ins_idx, true
75    insertins instruction, ins_idx, ins, true
76
77  :insertins, :removeins, :swapins, :adjust_jump_pos, :adjust_jump_pos_up, :
      adjust_jump_pos_down, :cst_lookup, :v2typ, :cst_add
```

Listing A.12: `opeth/opeth/common/constant.moon`

```
import mkcfg, get_block from require'opeth.common.blockrealm'
import du_chain, this_use, this_def, root_def from require'opeth.opeth.common.
    du_chain'
import map, foldl, filter, have, isk, cstid from require'opeth.common.utils'
import insert, concat from table
optbl = require'opeth.opeth.common.optbl'

FUNVAR = "userdata"

local rtype

typewidth = (fnblock, blk, ins_idx, reg, du_cfg, visited) ->
  with typs = {}
    for use in *blk.use do with use
      if .line == ins_idx and .reg == reg and #.defined == 1
        typ = rtype fnblock, .defined[1].line, reg, du_cfg, visited
        typ or= FUNVAR
        insert typs, typ unless have typs, typ

rtype = (fnblock, ins_idx, reg, du_cfg = (du_chain fnblock), visited = {}) ->
  for v in *visited
    if v.reg == reg and v.idx == ins_idx
      return v.typ

  v_ = {idx: ins_idx, :reg}

  insert visited, v_

  if ins_idx == 0
    v_.typ = FUNVAR
    return FUNVAR

  if isk reg
    cst = fnblock.constant[(math.abs reg) % 256 + (reg >= 0 and 1 or 0)]
    return cst and type cst.val

  fallback = (reg_ = reg) ->
    if ins_idx == 1 then return FUNVAR

    blk = get_block nil, ins_idx, du_cfg
    if ins_idx > blk.start then rtype fnblock, ins_idx - 1, reg_, du_cfg, visited
    else
      typs = with t = {}
        typs_ = typewidth fnblock, blk, ins_idx, reg_, du_cfg, visited
        insert t, e for e in *typs_ when not have t, e

      typs[1] if #typs == 1

  ins = fnblock.instruction[ins_idx]
  {RA, RB, RC, :op} = ins

  (=>
    v_.typ = @
```

```
    @
  ) switch op
    when LOADK
      if reg == RA
        cst = fnblock.constant[(math.abs RB) % 256 + (RB >= 0 and 1 or 0)]
        cst and type cst.val
      else fallback!
    when NEWTABLE, SETTABLE, SETLIST
      if reg == RA then "table"
      else fallback!
    when MOVE
      if reg == RA then rtype fnblock, ins_idx, RB, du_cfg, visited
      else fallback!
    when GETTABUP, GETTABLE
      if reg == RA then FUNVAR
      else fallback!
    when LOADNIL
      if reg == RA or reg == RB then "nil"
      else fallback!
    when LOADBOOL
      if reg == RA then "bool"
      else fallback!
    when CLOSURE
      if reg == RA then "function"
      else fallback!
    when CONCAT
      for ci = RB, RC
        t = rtype fnblock, ins_idx, ci, du_cfg, visited
        if t != "string" and t != "number"
          return nil
      "string"
    when CALL
      if RA <= reg
        blk = get_block nil, ins_idx, du_cfg
        maxdef = foldl ((s, d) -> d.line == ins_idx and (d.reg > s and d.reg or s)
            or s), -1, blk.def

        if reg <= maxdef then nil
        else fallback!
      else fallback!
    when LEN
      if reg == RB
        typs = typewidth fnblock, (get_block nil, ins_idx, du_cfg), ins_idx, reg,
            du_cfg, visited
        typs[1] if #typs == 1
      elseif typRB == "string" then "number"
      else fallback!
    when NOT
      if reg == RA
        switch fallback RB
          when "table", "userdata", nil then nil
          else "bool"
      else fallback!
    when UNM
```

```
105       if reg == RA
106         if (fallback RB) == "number" then "number"
107       else fallback!
108     when ADD, SUB, MUL, DIV, MOD, IDIV, BAND, BXOR, BOR, SHL, SHR, POW
109       blk = get_block nil, ins_idx, du_cfg
110
111       typRB = if RA == RB
112         typs = typewidth fnblock, blk, ins_idx, RB, du_cfg, visited
113         typs[1] if #typs == 1
114       elseif isk RB
115         type fnblock.constant[-RB].val
116       else fallback RB
117
118       typRC = if RA == RC
119         typs = typewidth fnblock, blk, ins_idx, RC, du_cfg, visited
120
121         typs[1] if #typs == 1
122       elseif RB == RC then typRB
123       elseif isk RC then type fnblock.constant[-RC].val
124       else fallback RC
125
126       if reg == RA and typRB == typRC and typRB == "number" then "number"
127       elseif reg == RB then typRB
128       elseif reg == RC then typRC
129       else fallback!
130     when VARARG
131       if RA <= reg and reg <= (RA + RB - 2) then nil
132       else fallback!
133     when GETUPVAL
134       if reg == RA then nil
135       else fallback!
136     when TESTSET
137       if reg == RB or reg == RA then fallback RB
138       else fallback!
139     when SELF
140       if reg == RA + 1 then fallback RB
141       elseif reg == RA then nil
142       else fallback!
143     else -- SETTABUP, JMP, TEST, EQ, LT, LE, TFORLOOP, TFORCALL, FORLOOP, FORPREP
144       fallback!
145
146 -- return `true, value` or `false`, `true, ...` means "value is decidable"
147 rcst = (fnblock, ins_idx, reg, du_cfg = (du_chain fnblock), visited) ->
148   if ins_idx == 0 then return false -- may be functoin argument
149
150   ins = fnblock.instruction[ins_idx]
151
152   {RA, RB, RC, :op} = ins
153
154   fallback = (reg_ = reg) ->
155     if ins_idx == 1 then return nil
156
157     blk = get_block nil, ins_idx, du_cfg
158
```

```
159     if ins_idx > blk.start
160       has_cst, cst = rcst fnblock, ins_idx - 1, reg_, du_cfg
161       has_cst and cst or nil
162     else
163       if d_rx = root_def blk, ins_idx, reg_
164         -- watch defined position if `reg_` is not `RA`
165         has_cst, cst = rcst fnblock, d_rx.line, reg_, du_cfg if d_rx.line != ins_idx
                and d_rx.reg_ != reg_
166         has_cst and cst or nil
167       else
168         csts = {}
169
170         for pred in *blk.pred
171           has_cst, cst = rcst fnblock, pred.end, reg_, du_cfg, visited
172           insert csts, cst if has_cst and not have csts, cst
173
174         csts[1] if #csts == 1
175
176         -- for pred in *blk.pred
177           -- cst_t = {rtype fnblock, pred.end, reg_, du_cfg, visited}
178           -- is_uniq = true
179
180           -- for c in *csts
181             -- if c[1] and c[2] == cst_t[2]
182               -- is_uniq = false
183               -- break
184
185           -- insert csts, {rtype fnblock, pred.end, reg_, du_cfg, visited} if
                is_uniq
186
187         -- csts[1][2] if #csts == 1
188
189   if op != LOADK and reg != RA
190     if reg < 0 and isk reg
191       cst = fnblock.constant[cstid reg]
192       if cst then return true, cst.val
193       else return false
194
195     cst = fallback!
196     return cst != nil, cst
197
198   (=> @ != nil, @) switch op
199     when LOADK then fnblock.constant[cstid RB].val
200     when LOADBOOL then RB != 0
201     when CALL
202       if RA <= reg
203         blk = get_block nil, ins_idx, du_cfg
204         maxdef = foldl ((s, d) -> d.line == ins_idx and (d.reg > s and d.reg or s)
              or s), -1, blk.def
205
206         if reg <= maxdef then nil
207         else fallback!
208       else fallback!
209     when MOVE
```

```
      blk = get_block nil, ins_idx, du_cfg
      use = this_use blk, ins_idx, RB

      if #use.defined == 1
        has_cst, cst = rcst fnblock, use.defined[1].line, use.defined[1].reg, du_cfg

        if has_cst then cst
      else fallbak RB
    when LEN
      blk = get_block nil, ins_idx, du_cfg
      has_cst, str = do
        d_rb = this_def blk, ins_idx, RB
        rcst fnblock, d_rb.line, RB, du_cfg

      #str if has_cst -- `LEN X X` can't determine which to return, R(A) or R(B)
    when UNM
      if cst = fallback RB
        -cst
    when NOT
      if cst = fallback RB
        not cst
    when ADD, SUB, MUL, DIV, BAND, BXOR, BOR, SHL, SHR, POW
      blk = get_block nil, ins_idx, du_cfg

      has_cstB, cstRB = if isk RB
        if cst = fnblock.constant[cstid RB] then true, cst.val
        else false
      else
        if RA == RB
          cst = fallback RB
          cst != nil, cst
        elseif u_rb = this_use blk, ins_idx, RB
          if #u_rb.defined == 1
            rcst fnblock, u_rb.defined[1].line, RB, du_cfg

      has_cstC, cstRC = if isk RC
        if cst = fnblock.constant[cstid RC] then true, cst.val
        else false
      elseif RB == RC then has_cstB, cstRB
      else
        if RA == RC
          cst = fallback RC
          cst != nil, cst
        elseif u_rc = this_use blk, ins_idx, RC
          if #u_rc.defined == 1
            rcst fnblock, u_rc.defined[1].line, RC, du_cfg

      if has_cstB and has_cstC
        optbl[op] cstRB, cstRC
    when IDIV, MOD
      blk = get_block nil, ins_idx, du_cfg

      has_cstC, cstRC = if isk RC
        if cst = fnblock.constant[cstid RC] then true, cst.val
```

```
      else false
    else
      if RA == RC
        cst = fallback RC
        cst != nil, cst
      elseif u_rc = this_use blk, ins_idx, RC
        if #u_rc.defined == 1
          rcst fnblock, u_rc.defined[1].line, RC, du_cfg

    if has_cstC and cstRC == 0 then return nil

    has_cstB, cstRB = if isk RB
      if cst = fnblock.constant[cstid RB] then true, cst.val
      else false
    elseif RB == RC then has_cstC, cstRC
    else
      if RA == RB
        cst = fallback RB
        cst != nil, cst
      elseif u_rb = this_use blk, ins_idx, RB
        if #u_rb.defined == 1
          rcst fnblock, u_rb.defined[1].line, RB, du_cfg

    if has_cstB and has_cstC
      optbl[op] cstRB, cstRC
  -- `CONCAT` only checks all the types of `R(range RB, RC)`
  when CONCAT
    typ_cst = rtype fnblock, ins_idx, reg, du_cfg

    return unless typ_cst == "string"

    csts = {}

    for cat_reg = RB, RC
      if cst = fallback cat_reg
        insert csts, cst
      else return

    concat csts if #csts == (RC - RB + 1)
  else fallback!

:rtype, :rcst
```

Listing A.13: `opeth/opeth/common/du_chain.moon`

```
import have, filter, map, foldl, last from require'opeth.common.utils'
import get_block, mkcfg from require'opeth.common.blockrealm'
import insert, sort, remove from table
import max, tointeger from math
STACKTOP = 254

have_pos = (s, e) -> (filter ((b) -> b.line == e.line and b.reg == e.reg), s)[1]

-- T ∩ S
```

```
intersec = (t = {}, s = {}) -> [e for e in *t when have_pos s, e]

-- T - S
diff = (t = {}, s = {}) -> [e for e in *t when not have_pos s, e]

-- T ∪ S
union = (t = {}, s = {}) -> with ret = [e for e in *t]
  insert ret, e for e in *(diff s, t)

-- latest registers' status
latest = (t) ->
  with ret = {}do for e in *t
    -- If no instruction overwrites `reg` ?
    if #(filter (=> @reg == e.reg), ret) == 0
      insert ret, e
    else
      if #(filter (=> @reg == e.reg and @line < e.line), ret) > 0
        for ri = 1, #ret
          if ret[ri].reg == e.reg
            remove ret, ri
            insert ret, e
            break

pos_tgen = (ins_idx) -> (rx) -> {line: ins_idx, reg: rx}

du_chain = (fnblock, cfg = mkcfg fnblock.instruction) ->
  instruction = fnblock.instruction
  upvs = {}

  for block in *cfg
    gen = with d = {}
      block.gen = d
      if block.start == 1
        -- 0: R(vx) <- ARG(vx) for vx = 0, function_arguments
        insert d, (pos_tgen 0) r for r = 0, tointeger (tonumber fnblock.params, 16)
            - 1

    use = with u = {}
      block.use = u

    for ins_idx = block.start, block.end
      ins = instruction[ins_idx]
      {RA, RB, RC} = map tointeger, ins

      pos_t = pos_tgen ins_idx

      switch ins.op
        -- R(A) = R(B) (`op` R(C))
        when ADD, SUB, MUL, MOD, POW, DIV, IDIV, BAND, BOR, BXOR, SHL, SHR, BNOT,
            NOT, UNM, NEWTABLE
          insert gen, pos_t RA
          insert use, pos_t RB if RB >= 0
          insert use, pos_t RC if RC and (RC >= 0 and RC != RB)
        when MOVE, LEN, TESTSET
```

```
62          insert gen, pos_t RA
63          insert use, pos_t RB
64        when LOADK, LOADKX, GETUPVAL, LOADBOOL
65          insert gen, pos_t RA
66          -- insert use, RB if RB >= 0
67        when GETTABUP
68          insert gen, pos_t RA
69          insert use, pos_t RC if RC >= 0
70        when GETTABLE
71          insert gen, pos_t RA
72          insert use, pos_t RB
73          insert use, pos_t RC if RC >= 0
74        when SETTABLE
75          insert gen, pos_t RA
76          insert use, pos_t RB if RB >= 0
77          insert use, pos_t RC if RC >= 0
78        when SETUPVAL, TEST
79          insert use, pos_t RA
80        when SETTABUP
81          insert use, pos_t RB if RB >= 0
82          insert use, pos_t RC if RC >= 0
83        when CLOSURE
84          insert gen, pos_t RA
85
86          -- consider `GETUPVAL` in closure[ins[2] + 1]
87          proto = fnblock.prototype[RB + 1]
88
89          for u in *proto.upvalue
90            if u.instack == 1
91              insert use, pos_t u.reg
92              insert upvs, u.reg
93        when LOADNIL
94          insert gen, pos_t r for r = RA, RA + RB
95        -- `t:f()` to R(A + 1) = `f`; R(A) = `t`
96        when SELF
97          insert gen, pos_t RA
98          insert gen, pos_t RA + 1
99          insert use, pos_t RB
100       when CALL
101         insert use, pos_t a for a = RA, RA + RB - 1
102
103         uselimit = RB == 0 and (#gen > 0 and (max unpack [u.reg for u in *gen]) or
                STACKTOP) or (RA + RB - 1)
104         insert use, pos_t a for a = RA, uselimit
105
106         def_relat = RC == 0 and ((with dp = filter (=> @ > uselimit), [i.line for
                i in *gen] do sort dp)[1] or STACKTOP) or RA + RC - 2
107         insert gen, pos_t r for r = RA, def_relat
108
109         -- I've given up to check whether `SETUPVAL` is used in the closure of R(A
                ),
110         --  so assume that ALL the value the previous CLOSURE instruction closed
                is defined/used.
111         for u in *upvs
```

```
112           -- insert gen, pos_t u
113           insert use, pos_t u
114         when TAILCALL
115           arglimit = RB == 1 and 0 or (RB == 0 and (#use > 0 and (max unpack [u.reg
                for u in *use]) or STACKTOP) or RA + RB - 1)
116           insert use, pos_t a for a = RA, arglimit
117         when EQ, LT, LE
118           insert use, pos_t RB if RB >= 0
119           insert use, pos_t RC if RC >= 0
120         when FORLOOP
121           insert gen, pos_t RA
122           insert gen, pos_t RA + 3
123           insert use, pos_t RA
124           insert use, pos_t RA + 1
125         when FORPREP
126           insert gen, pos_t RA
127           insert use, pos_t RA
128           insert use, pos_t RA + 2
129         when TFORCALL
130           insert gen, pos_t r for r = RA + 3, RA + 2 + RC
131           insert use, pos_t u for u = RA, RA + 2
132         when TFORLOOP
133           insert use, pos_t RA + 1
134           insert gen, pos_t RA
135
136           with instruction[ins_idx - 1]
137             assert .op == TFORCALL, "next TO TFORCALL must be TFORLOOP"
138         when SETLIST
139           len = RB != 0 and RB or (#use > 0 and (max unpack [u.reg for u in *use])
                or STACKTOP)
140           insert use, pos_t RA + i for i = 0, len
141         when VARARG
142           genlimit = RB == 0 and STACKTOP or RA + RB - 2
143           insert gen, pos_t r for r = RA, genlimit
144         when CONCAT
145           insert gen, pos_t RA
146           insert use, pos_t a for a = RB, RC
147         when RETURN
148           ret = RB == 1 and -1 or (RB == 0 and (#use > 0 and (max unpack [u.reg for
                u in *use]) or STACKTOP) or RA + RB - 2)
149           insert use, pos_t r for r = RA, ret
150         when JMP
151           insert use, pos_t RA - 1 if RA > 0
152         -- nop
153
154       with block
155         .in, .kill, .out = {}, {}, {modified: true}
156
157     while foldl ((bool, blk) -> bool or blk.out.modified), false, cfg
158       for block in *cfg do with block
159         out = .out
160         .in = foldl ((in_, pblk) -> union in_, pblk.out), {}, .pred
161         .kill = intersec .in, .gen
162         .out = union (latest .gen), diff .in, .kill
```

```
163       .out.modified = #(diff .out, out) > 0
164
165
166   -- referring `use.defined` <--> `def.used`
167   for block in *cfg do with block
168     .def = union .gen, .in
169
170     for use in *.use
171       use.defined = {}
172
173       if defined = last latest filter ((g) -> g.line < use.line and g.reg == use.reg
              ), .gen
174         insert use.defined, defined unless have use.defined, defined
175         unless defined.used
176           defined.used = {use}
177         else
178           insert defined.used, use unless have use.defined, use
179       else
180         for defined in *(filter ((i) -> i.reg == use.reg), .in)
181           insert use.defined, defined unless have use.defined, defined
182           unless defined.used
183             defined.used = {use}
184           else
185             insert defined.used, use unless have defined.used, use
186
187     for d in *.def
188       d.used or= {}
189
190   for blk in *cfg do with blk
191     .out.modified, .kill, .gen, .out, .in = nil
192
193   cfg
194
195 -- utils
196 this_use = (blk, ins_idx, reg) ->
197   for u in *blk.use
198     if u.line == ins_idx and u.reg == reg
199       return u
200
201 this_def = (blk, ins_idx, reg) ->
202   last latest filter (=> @line <= ins_idx and @reg == reg), blk.def
203
204 root_def = do
205   pred_def = (blk, reg, visited = {}) ->
206     if have visited, blk
207       return
208     insert visited, blk
209
210     if d = last latest filter (=> @reg == reg), blk.def
211       return d
212
213     preds = [pred_def pred, reg, visited for pred in *blk.pred]
214
215     if #preds == 1
```

```
216       preds[1]
217
218   (blk, ins_idx, reg) ->
219     if d = last latest filter (=> @line <= ins_idx and @reg == reg), blk.def
220       return d
221
222     pred_def blk, reg
223
224 :du_chain, :this_use, :this_def, :root_def
```

Listing A.14: opeth/opeth/common/optbl.moon

```
1  {
2      ADD:   (a, b) -> a + b
3      SUB:   (a, b) -> a - b
4      MUL:   (a, b) -> a * b
5      DIV:   (a, b) -> a / b
6      MOD:   (a, b) -> a % b
7      POW:   (a, b) -> a ^ b
8
9      IDIV:  (a, b) -> a // b
10     BAND:  (a, b) -> a & b
11     BOR:   (a, b) -> a | b
12     BXOR:  (a, b) -> a ~ b
13     SHL:   (a, b) -> a << b
14     SHR:   (a, b) -> a >> b
15
16     LT:    (a, b) -> a < b
17     LE:    (a, b) -> a <= b
18     EQ:    (a, b) -> a == b
19 }
```

A.4 Modules for The Opeth Command

Listing A.15: opeth/opeth/cmd/optimizer.moon

```
1  import map, deepcpy from require'opeth.common.utils'
2  Debuginfo = require'opeth.opeth.cmd.debuginfo'
3
4  print_moddiffgen = (optfn, optname) -> (fnblock) ->
5    fnblock.optdebug\start_rec!
6    optfn fnblock
7    fnblock.optdebug\print_modified optname
8
9  opt_names = {
10   {
11     name:"unreachable blocks removal"
12     description: "remove all the blocks which are unreachable for the top"
13   }
14   {
15     name: "constant fold"
16     description: "evaluate some operations beforehand"
```

```
  }
  {
    name: "constant propagation"
  }
  {
    name:"dead-code elimination"
  }
  {
    name:"function inlining"
  }
}

unreachable_remove = print_moddiffgen require'opeth.opeth.unreachable_remove',
    opt_names[1].name
cst_fold = print_moddiffgen require'opeth.opeth.cst_fold', opt_names[2].name
cst_prop = print_moddiffgen require'opeth.opeth.cst_prop', opt_names[3].name
dead_elim = print_moddiffgen require'opeth.opeth.dead_elim', opt_names[4].name
func_inline = print_moddiffgen require'opeth.opeth.func_inline', opt_names[5].name
unused_remove = print_moddiffgen require'opeth.opeth.unused_remove', "unused
    resources removal"

opt_tbl = {
  unreachable_remove
  (=> func_inline @ if #@prototype > 0)
  cst_fold
  cst_prop
  dead_elim
  mask: (mask) =>
    newtbl = deepcpy @
    newtbl[i] = (=>) for i in *mask
    newtbl
}

optimizer = (fnblock, mask, verbose) ->
  unless fnblock.optdebug
    fnblock.optdebug = Debuginfo 0, 0, nil, verbose
  else fnblock.optdebug\reset_modified!

  map (=> @ fnblock), opt_tbl\mask mask

  for pi = 1, #fnblock.prototype
    debuginfo = Debuginfo fnblock.optdebug.level + 1, pi, fnblock.optdebug\fmt!,
        verbose
    fnblock.prototype[pi].optdebug = debuginfo
    optimizer fnblock.prototype[pi], mask, verbose

  optimizer fnblock, mask if fnblock.optdebug.modified > 0

recursive_clean = (fnblock, verbose) ->
  unused_remove fnblock

  for pi = 1, #fnblock.prototype
    debuginfo = Debuginfo fnblock.optdebug.level + 1, pi, fnblock.optdebug\fmt!,
        verbose
```

```
    fnblock.prototype[pi].optdebug = debuginfo
    recursive_clean fnblock.prototype[pi], verbose

setmetatable {:opt_names},
  __call: (fnblock, mask = {}, verbose) =>
    optimizer fnblock, mask, verbose
    recursive_clean fnblock, verbose
    fnblock
```

Listing A.16: `opeth/opeth/cmd/metainfo.moon`

```
-- name        = $(awk '$1 !~ /^--/ {print $1}' HERE)
-- version     = $(awk '$1 !~ /^--/ {print $1}' opeth/opeth/cmd/version.lua)
-- description = $(awk '$1 !~ /^--/ {print $5}' HERE)
name: "opeth" , version: (require'opeth.opeth.cmd.version') , description: "Lua VM
    Bytecode Optimizer"
```

Listing A.17: `opeth/opeth/cmd/debuginfo.moon`

```
class
  new: (@level, @no, @parent, @verbose = false, @modified = 0) =>
  fmt: => @parent and "#{@parent}->#{@level}.#{@no}" or "main"
  start_rec: => @rec = @modified
  stop_rec: => with @rec do @rec = nil
  mod_inc: => @modified += 1
  mod_dec: => @modified -= 1
  mod_add: (add) => @modified += add
  reset_modified: => @modified = 0
  print_modified: (module_name) =>
    if @verbose and @rec
      print "#{module_name}##{@fmt!}: #{@modified - @stop_rec!} modified"
```

Listing A.18: `opeth/opeth/cmd/version.lua`

```
return 0.0
```

A.5 Bytecode Reader/Writer

Listing A.19: `opeth/bytecode/reader.moon`

```
import concat from table
import char from string

import zsplit, map, prerr, undecimal from require'opeth.common.utils'
import hexdecode, hextobin, adjustdigit, bintoint, hextoint, hextochar, bintohex
    from undecimal

string = string
string.zsplit = zsplit

```

```
insgen = (ins) ->
  abc = (a, b, c) ->
    unpack map (=> with r = bintoint @ do if r > 255 then return 255 - r), {a, b, c}
  abx = (a, b, _b) ->
    unpack map bintoint, {a, b .. _b}
  asbx = (a, b, _b) ->
    mpjs = map bintoint, {a, b .. _b}
    mpjs[2] -= 2^17 - 1
    unpack mpjs
  ax = (a, _, _) -> bintoint a

  oplist = require'opeth.common.oplist' abc, abx, asbx, ax
  setmetatable oplist,
    __index: (v) =>
      if e = rawget @, v then e
      else error "invalid op: #{math.tointeger v}"

  b, c, a, i = (hextobin ins)\match "(#{"."\rep 9})(#{"."\rep 9})(#{"."\rep 8})(#{"."\rep 6})"
  {op, fn} = oplist[(bintoint i) + 1]

  {:op, fn(a, b, c)}

-- XXX: supported little endian 64bit float only
ieee2f = (rd) ->
  mantissa = (rd\byte 7) % 16
  for i = 6, 1, -1 do mantissa = mantissa * 256 + rd\byte i
  exponent = ((rd\byte 8) % 128) * 16 + ((rd\byte 7) // 16)
  exponent == 0 and 0 or ((mantissa * 2 ^ -52 + 1) * ((rd\byte 8) > 127 and -1 or 1)) * (2 ^ (exponent - 1023))

-- Reader class
-- add common operations to string and file object
-- {{{
class Reader
  read = (n) =>
    if n == "*a" then n = #@
    @cur += n
    local ret

    ret, @val = @val\match("^(#{(".")\rep n})(.*)$")
    ret
  new: (file, val) =>
    typ = type file
    file = switch typ
      when "userdata" then file
      when "string" then assert io.open(file, "r"), "Reader.new #1: failed to open file `#{file}'"
      when "nil" then nil
      else error "Reader.new receives only the type of string or file (got `#{typ}')"

    @val = val or file\read "*a"
    @priv = {:file, val: @val}
```

```
    @cur = 1
  __shr: (n) => read @, n
  __len: => #@priv.val - @cur + 1
  close: =>
    @priv.file\close!
    @priv = nil
  seek: (s, ofs) =>
    if s == "seek"
      @cur = 0
      @val = @priv.val
    else
      unless ofs then @cur
      else
        if type(ofs) != "number"
          error "Reader\\seek #2 require number, got #{type ofs}"
        else
          @cur += ofs
          @val = @priv.val\match ".*$", @cur
-- }}}

-- decodeer
----{{{
read_header = (rd) ->
  {
    hsig: rd >> 4
    version: (hexdecode! (rd >> 1)\byte!)\gsub("(%d)(%d)", "%1.%2")
    format: (rd >> 1)\byte!
    luac_data: rd >> 6
    size: {
      int: (rd >> 1)\byte!
      size_t: (rd >> 1)\byte!
      instruction: (rd >> 1)\byte!
      lua_integer: (rd >> 1)\byte!
      lua_number: (rd >> 1)\byte!
    }

    -- luac_int, 0x5678
    endian: (rd >> 8) == ((char(0x00))\rep(6) .. char(0x56, 0x78)) and 0 or 1

    -- luac_num, checking IEEE754 float format
    luac_num: rd >> 9
  }

assert_header = (header) ->
  with header
    assert .hsig == char(0x1b, 0x4c, 0x75, 0x61), "HEADER SIGNATURE ERROR" -- header
        signature
    assert .luac_data == char(0x19, 0x93, 0x0d, 0x0a, 0x1a, 0x0a), "PLATFORM
       CONVERSION ERROR"
    assert 370.5 == (ieee2f .luac_num), "IEEE754 FLOAT ERROR"

providetools = (rd, header) ->
  import endian, size from header or read_header rd

```

```
  adjust_endianness = if endian < 1 then (=> @) else (xs) -> [xs[i] for i = #xs, 1, -1]
  undumpchar = -> hexdecode! (rd >> 1)\byte!
  undump_n = (n) -> hexdecode(n) unpack adjust_endianness {(rd >> n)\byte 1, n}
  undumpint = -> undump_n tonumber size.int

  :adjust_endianness, :undump_n, :undumpchar, :undumpint

read_fnblock = (rd, header = (read_header rd), has_debug) ->
  import adjust_endianness, undump_n, undumpchar, undumpint from providetools rd, header

  local instnum

  {
    chunkname:
      with ret = table.concat [char hextoint undumpchar! for _ = 2, hextoint undumpchar!]
        has_debug = has_debug or #ret > 0

    line: {
      defined: undumpint!
      lastdefined: undumpint!
    }

    params: undumpchar!
    vararg: undumpchar!
    regnum: undumpchar! -- number of register to use

    -- instructions: [num (size of int)] [instructions..]
    -- instruction: [inst(4)]
    instruction: do
      -- with num: hextoint undumpint!
      (=> [insgen undumpint! for _ = 1, @]) with num = hextoint undumpint!
        instnum = num

    -- constants: [num (size of int)] [constants..]
    -- constant: [type(1)] [...]
    constant: for _ = 1, hextoint undumpint!
      with type: (rd >> 1)\byte!
        .val = switch .type
          when 0x1
            -- bool
            undumpchar!
          when 0x3
            -- number
            ieee2f rd >> header.size.lua_number
          when 0x13
            -- signed integer
            n = undump_n header.size.lua_integer
            if n\match"^[^0-7]" then 0x10000000000000000 + hextoint n
            else hextoint n
          when 0x4, 0x14
            -- string
```

```
          if s = (=> concat adjust_endianness map hextochar, (undump_n @)\zsplit 2
              if @ > 0) with len = hextoint undumpchar!
              if len == 0xff -- #str > 255
                len = hextoint undump_n header.size.lua_integer
              return len - 1 -- remove '\0' in internal expression
            s
          else ""
        else nil

    upvalue: for _ = 1, hextoint undumpint!
      u = adjust_endianness {(hextoint undumpchar!), (hextoint undumpchar!)}
      {reg: u[1], instack: u[2]} -- {reg, instack}, instack is whether it is in
          stack

    prototype: [read_fnblock rd, header, has_debug for i = 1, hextoint undumpint!]

    debug: with ret = {}
      .linenum = hextoint undumpint!

      if has_debug then .opline = [hextoint undumpint! for _ = 1, instnum]

      .varnum = hextoint undumpint!

      if has_debug then .varinfo = for _ = 1, .varnum
        {
          varname: concat adjust_endianness map hextochar, (undump_n (hextoint
              undumpchar!) - 1)\zsplit 2
          life: {
            begin: hextoint undumpint! -- lifespan begin
            end: hextoint undumpint! -- lifespan end
          }
        }

      .upvnum = hextoint undumpint!

      if has_debug then .upvinfo = for _ = 1, .upvnum
        concat adjust_endianness map hextochar, (undump_n (hextoint undumpchar!) -
            1)\zsplit 2
  }
-- }}}

read = (reader, top = true) ->
  header = assert_header read_header reader
  fnblock = read_fnblock reader, header

  :header, :fnblock

Reader.__base.read = read
:Reader, :read
```

Listing A.20: `opeth/bytecode/writer.moon`

```
import concat from table
import char from string
```

```
import floor, tointeger from math

import zsplit, map, insgen, prerr, undecimal from require'opeth.common.utils'
import hexdecode, hextobin, adjustdigit, bintoint, hextoint, hextochar, bintohex,
    inttobin from undecimal
op_list = require'opeth.common.oplist' "abc", "abx", "asbx", "ab"

string = string
string.zsplit = zsplit

-- TODO: now only supported signed 64bit float
f2ieee = (flt) ->
  if flt == 0 then return "0"\rep 64

  bias = 1023
  abs_flt = math.abs flt
  e, m = math.modf abs_flt

  while e == 0
    abs_flt *= 2
    bias -= 1
    e, m = math.modf abs_flt

  while e > 9.223372e18
    e /= 2
    bias += 1

  mb = ""
  pa = (inttobin e)\match"0*1(.*)" or ""
  e = #pa + bias

  for b in pa\gmatch"."
    if #mb == 52 then break
    mb ..= b

  eb = adjustdigit (hextobin "%x"\format e)\match"0*(.*)", 11

  for i = -1, -(52 - #mb), -1
    p = 2^i

    if m - p >= 0
      m -= p
      mb ..= "1"
      if m == 0
        while #mb < 52 do mb ..= "0"
        break
    else mb ..= "0"

  (flt < 0 and "1" or "0") .. eb .. mb

-- Writer class
-- interface to write to file
-- {{{
class Writer
```

```
  new: (cont) =>
    typ = type cont
    @cont = switch typ
      when "userdata" then cont
      when "string" then assert io.open(cont, "w+b"), "Writer.new #1: failed to open file `#{cont}'"
      when "nil" then {block: "", write: ((a) => @block ..= a), flush: (=>), close: (=>), seek: (=>), read: (=>)}
      else error "Writer.new receives only the type of string or file (got `#{typ}')"
    @size = 0
  __shl: (v) =>
    @size += #v
    with @ do @cont\write v
  __len: => @size
  close: =>
    @cont\flush!
    @cont\close!
  show: =>
    pos = @cont\seek "cur"
    @cont\seek "set"
    with @cont\read "*a"
      @cont\seek "set", pos
-- }}}

-- write (re) encoded data to file
-- {{{
local adjust_endianness

regx = (i) -> hextobin "%x"\format i
writeint = (wt, int, dig = 8) -> map (=> wt << hextochar @), adjust_endianness (adjustdigit ("%x"\format int), dig)\zsplit 2

write_fnblock = (wt, fnblock, has_debug) ->
  import chunkname, line, params, vararg, regnum, instruction, constant, upvalue, prototype, debug from fnblock

  -- chunkname
  -- {{{
  if has_debug or #chunkname > 0
    has_debug = true
    wt << char #chunkname + 1
    map (=> wt << @), chunkname\zsplit!
  else wt << "\0"
  -- }}}

  -- parameters
  -- {{{
  map (=> writeint wt, (hextoint @)), {line.defined, line.lastdefined}
  map (=> wt << hextochar @), {
     params
     vararg
     regnum
    }
```

```
105   -- }}}
106
107   -- instruction
108   -- {{{
109   writeint wt, #instruction
110
111   for i = 1, #instruction
112     {RA, RB, RC, :op} = instruction[i]
113     a = adjustdigit (regx RA), 8
114     rbc = if RC
115       concat map (=> adjustdigit (regx if @ < 0 then 2^8 - 1 - @ else @), 9), {RB,
              RC}
116     else adjustdigit (regx if op_list[op][2] == "asbx" then RB +2^17-1 else RB), 18
117
118     bins = rbc ..a..(adjustdigit (regx (op_list[op].idx - 1)), 6)
119     assert #bins == 32
120     map (=> wt << hextochar @), adjust_endianness (concat map (=> bintohex @), bins\
          zsplit 4)\zsplit 2
121   -- }}}
122
123   -- constant
124   -- {{{
125   writeint wt, #constant
126
127   for i = 1, #constant do with constant[i]
128     wt << char .type
129
130     switch .type
131       when 0x1 then wt << char .val
132       when 0x3
133         wt << c for c in *(adjust_endianness [("0x"..(bintohex cxa) .. (bintohex cxb
                ))\char! for cxa, cxb in (f2ieee .val)\gmatch "(....)(....)"])
134       when 0x13 then writeint wt, .val, 16
135       when 0x4, 0x14
136         if #.val > 0xff
137           wt << char 0xff
138           writeint wt, #.val + 1, 16
139         else writeint wt, #.val + 1, 2
140
141         wt << .val
142   -- }}}
143
144   -- upvalue
145   -- {{{
146   writeint wt, #upvalue
147   map (=> wt << char @), adjust_endianness {upvalue[i].reg, upvalue[i].instack} for
          i = 1, #upvalue
148   -- }}}
149
150   -- prototype
151   -- {{{
152   writeint wt, #prototype
153   write_fnblock wt, prototype[i], has_debug for i = 1, #prototype
154   -- }}}
```

```
155
156   -- debug
157   -- {{{
158   -- {:linenum, :opline, :varnum, :varinfo, :upvnum, :upvinfo} = debug
159   import linenum, opline, varnum, varinfo, upvnum, upvinfo from debug
160
161   writeint wt, (has_debug and linenum or 0)
162
163   if has_debug then for i = 1, #(opline or "")
164     writeint wt, opline[i]
165
166   writeint wt, (has_debug and varnum or 0)
167
168   if has_debug then for i = 1, #(varinfo or "")
169     with varinfo[i]
170       writeint wt, #.varname+1, 2
171       wt << .varname
172       writeint wt, .life.begin
173       writeint wt, .life.end
174
175   writeint wt, (has_debug and upvnum or 0)
176
177   if has_debug then for i = 1, #(upvinfo or "")
178     writeint wt, #upvinfo[i]+1, 2
179     wt << upvinfo[i]
180   -- }}}
181
182 write = (wt, vmformat) ->
183   import header, fnblock from vmformat
184   adjust_endianness = header.endian < 1 and (=> @) or (xs) -> [xs[i] for i = #xs, 1
          , -1]
185
186   with header
187     map (=> wt << @), {
188         .hsig
189         (hextochar tointeger .version * 10)
190         (char .format)
191         .luac_data
192       }
193
194     with .size
195       map (=> wt << (char @)), {
196           .int
197           .size_t
198           .instruction
199           .lua_integer
200           .lua_number
201         }
202
203     map (=> wt << @), {
204         (concat adjust_endianness (((char 0x00)\rep 6) .. char 0x56, 0x78)\zsplit!)
205         .luac_num
206         .has_debug
207       }
```

```
208
209   write_fnblock wt, fnblock
210
211   wt
212 -- }}}
213
214 Writer.__base.write = write
215 :Writer, :write
```

A.6 Opeth

Listing A.21: opeth/bin/opeth.moon

```
1  #!/usr/bin/env moon
2
3  import read, Reader from require'opeth.bytecode.reader'
4  import write, Writer from require'opeth.bytecode.writer'
5  import gettime from require'socket'
6  import name, description, version from require'opeth.opeth.cmd.metainfo'
7  argparse = require'argparse'
8  optimizer = require'opeth.opeth.cmd.optimizer'
9
10 fn_time = (fn) ->
11   t1 = gettime!
12   fn!
13   t2 = gettime!
14   t2 - t1
15
16 inscounter = (fnblock) ->
17   with cnt = #fnblock.instruction
18     for proto in *fnblock.prototype
19       cnt += inscounter proto
20
21 args = (=> @parse!) with argparse name, description
22   \argument "input",                                  "luac file"
23   \option(  "-o --output",        "output file",      "optimized.out"
24     )\overwrite false
25   \option( "-x --disable-optimize", "disable a part of optimizer"
26     )\argname("index"
27     )\args("1+"
28     )\convert(=> tonumber @
29     )\target"mask"
30   \flag    "-V --verbose",       "verbose optimization process"
31   \flag    "-T --time",          "measure the time"
32   \flag(   "-v --version",       "version information"
33     )\action(->
34       print "#{name} v#{version}\n#{description}"
35       os.exit 0
36     )
37   \flag(   "--show-optimizations", "show a sort of otimization"
38     )\action(->
39       for o in *optimizer.opt_names
```

```
40        print "%-26s : %s"\format o.name, o.description
41      os.exit 0
42    )
43
44  ((ok, cont) ->
45    unless ok
46      msg = "\noutput file is none\n"
47
48      if cont\match "interrupted!"
49        msg = "interrupted!#{msg}"
50      else
51        msg = "#{cont}#{msg}"
52
53      io.stderr\write "\n[Error]: #{msg}"
54  ) pcall ->
55    rd = Reader args.input
56    wt = Writer args.output
57
58    io.write "read from #{args.input} (size: #{#rd} byte" if args.verbose
59    local vmfmt
60    rtime = (fn_time -> vmfmt = rd\read!) * 1000
61
62    io.write if args.time then args.verbose and ", time: #{rtime} msec)\n" or "read
          time: #{rtime} msec\n"
63    elseif args.verbose then ")\n"
64    else ""
65
66    rd\close!
67
68    insnum = if args.verbose then inscounter vmfmt.fnblock
69    otime = fn_time -> (optimizer vmfmt.fnblock, args.mask, args.verbose).chunkname =
          ""
70    print "#{args.verbose and "(" or ""}optimize time: #{otime * 1000} msec#{args.
          verbose and ")" or ""}" if args.time
71
72    (=> @\close!) with wt
73      wtime = (fn_time -> wt\write vmfmt) * 1000
74      print "change of the number of instructions: #{insnum} -> #{inscounter vmfmt.
            fnblock}" if args.verbose
75      io.write "\nwrite to #{args.output} (size: #{#wt} byte" if args.verbose
76      io.write if args.time then args.verbose and ", time: #{wtime} msec)\n" or "write
            time: #{wtime} msec\n"
77      elseif args.verbose then ")\n"
78      else ""
```