Ravi – a Lua 5.3 Dialect

Dibyendu Majumdar

Introduction

- Ravi is a dialect of Lua 5.3
- Features language enhancements to allow *limited* optional static typing of local declarations and function parameters
- Mixes static typing and dynamic typing to maintain (as far as possible) compatibility with Lua
- Lua and Ravi functions can be JIT compiled, automatically or upon user request
- Two JIT compiler implementations - LLVM and libgccjit
- Unit of compilation is a Lua closure
- Not 100% Lua compatible hence new name for the language
- Uses extended bytecodes specialized for types
- For selected benchmarks, Ravi matches LuaJIT performance

History

- Discovered Lua in 2014 while looking for an embedded scripting language
- Got interested in LuaJIT for performance
- However LuaJIT did not work well on all platforms, and did not play well with use cases where it would be embedded in a Java application
- Decided to try to understand LuaJIT with a view to enhancing it
- This was just too hard
- So Ravi was born as an attempt to create an alternative to LuaJIT for specific use case (numeric computing)
- Static typing is used to help the JIT compiler; strong type guarantees are necessary to ensure correctness of JIT compiled code

Comparison with LuaJIT

Ravi	**LuaJIT**
LLVM and libgccjit JIT compilers	Custom tracing JIT compiler
JIT compiler is slow	JIT compiler is fast
Large runtime image due to LLVM	Small runtime image
Not suited for small devices	Suited for small devices
Simpler implementation; easy to understand and support	Complex implementation; significantly harder to understand and support
No FFI, but LLVM binding available	FFI integrated into the system
Like Lua, safe for programmers coding in Lua	Unsafe due to FFI – you need to know what you are doing
Safety and maintainability are top priorities	Performance and small runtime image size are the top priorities

Ravi extension – typed local variables

- Local variables can be annotated with types
- Only 4 static types implemented:
  - Integer (64-bit)
  - Number (double)
  - Integer array (table subtype)
  - Number array (table subtype)
- Local variables initialized automatically
- The static types above are most relevant for numeric computing

```
> function tryme()
>>   local i: integer
>>   local f: number
>> end
> ravi.dumplua(tryme)

function <stdin:1,4> (5 instructions at 000000674CFA6080)
0 params, 2 slots, 0 upvalues, 2 locals, 0 constants, 0 functions
        1       [2]     LOADNIL         0 0
        2       [2]     LOADIZ          0
        3       [3]     LOADNIL         1 0
        4       [3]     LOADFZ          1
        5       [4]     RETURN          0 1
constants (0) for 000000674CFA6080:
locals (2) for 000000674CFA6080:
        0       i       3       6
        1       f       5       6
upvalues (0) for 000000674CFA6080:
```

Ravi extension – typed function arguments

- Function arguments can be annotated with types
- If annotated, type checks performed upon entry to function (i.e. at runtime)
- The type checks ensure that JIT compilation can proceed with certainty regarding the types of the function arguments

```
> function tryme(i: integer, f: number[])
>> end
> ravi.dumplua(tryme)

function <stdin:1,2> (3 instructions at 000000674CFA5A40)
2 params, 2 slots, 0 upvalues, 2 locals, 0 constants, 0 functions
        1       [1]     TOINT           0
        2       [1]     TOARRAYF        1
        3       [2]     RETURN          0 1
constants (0) for 000000674CFA5A40:
locals (2) for 000000674CFA5A40:
        0       i       1       4
        1       f       1       4
upvalues (0) for 000000674CFA5A40:
```

Ravi extension – return type coercion

- If the value of a function call is assigned to a typed variable then a type check / coercion is performed at run time
- Static type checking alone would not provide strong guarantee needed by JIT compiler

```
> function foo()
>>   local i: integer = bar()
>>   local f: number[] = bar()
>>   f = bar()
>> end
> ravi.dumplua(foo)

function <stdin:1,5> (10 instructions at 000000674CFA66C0)
0 params, 3 slots, 1 upvalue, 2 locals, 1 constant, 0 functions
        1       [2]     GETTABUP        0 0 -1  ; _ENV "bar"
        2       [2]     CALL            0 1 2
        3       [2]     TOINT           0
        4       [3]     GETTABUP        1 0 -1  ; _ENV "bar"
        5       [3]     CALL            1 1 2
        6       [3]     TOARRAYF        1
        7       [4]     GETTABUP        2 0 -1  ; _ENV "bar"
        8       [4]     CALL            2 1 2
        9       [4]     MOVEAF          1 2
        10      [5]     RETURN          0 1
constants (1) for 000000674CFA66C0:
        1       "bar"
locals (2) for 000000674CFA66C0:
        0       i       4       11
        1       f       7       11
upvalues (1) for 000000674CFA66C0:
        0       _ENV    0       0
```

Ravi extension - arrays

- Ravi arrays are subtypes of Lua tables
- When types are known static checking is done where possible to ensure correct behaviour
- Table initializers are checked at runtime rather than compile time as each value could result from an expression

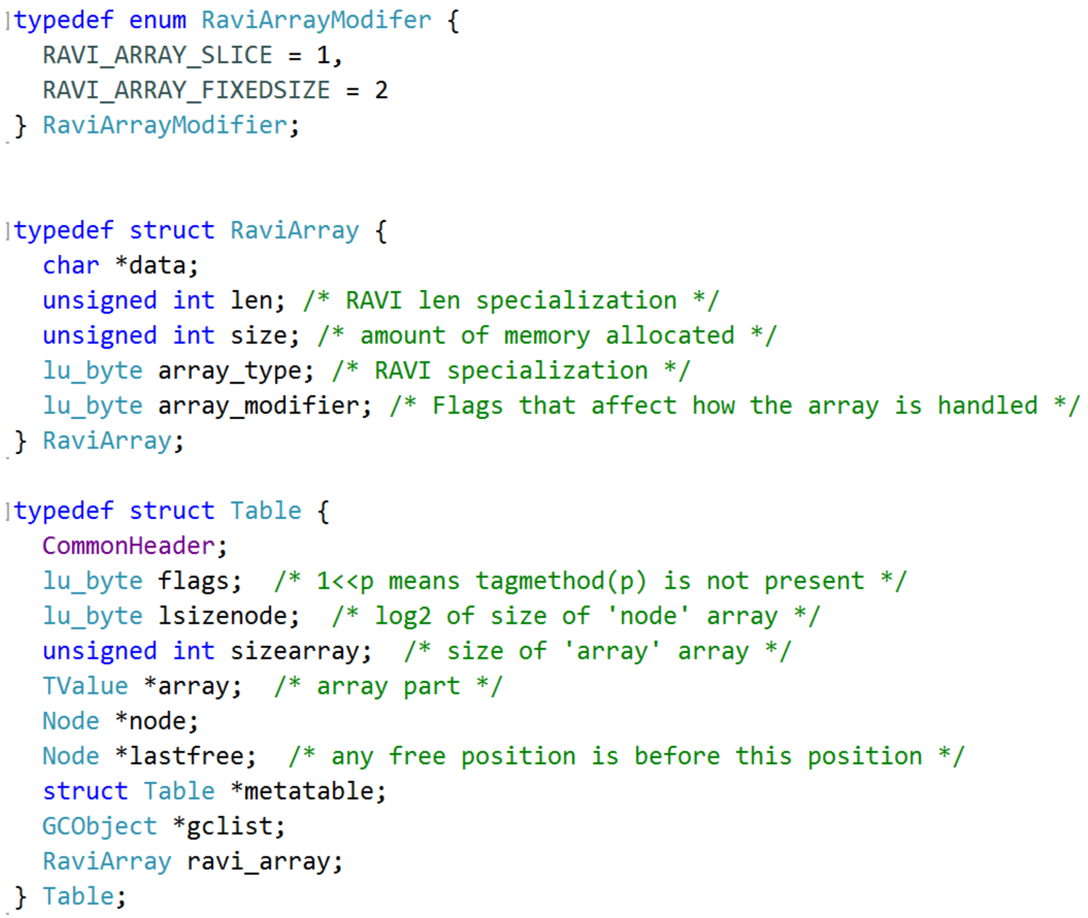
```
>
> function foo()
>>   local f: number[]
>>   f = 'hello'
>> end
stdin:4: Invalid assignment of type: var type 4, expression
type 7 near 'end'
>
> function foo()
>>   local f: number[] = { 'hello', 'world' }
>> end
> foo()
stdin:2: value cannot be converted to number
stack traceback:
        stdin:2: in function 'foo'
        (...tail calls...)
        [C]: in ?
>
```

Ravi extension - arrays

- A Ravi array crossing into Lua looks like a table but has restrictions on types of values and indexing operations
- Meta methods not supported on arrays
- Array type uses additional fields in the Lua Table structure
- The array data is held in contiguous memory compatible with native arrays
- Arrays are initialized to 0 not nil
- For performance reasons the arrays have a slot at index 0 but this is not visible in initializers or iterators; however direct indexing will reveal
- The extra slot at index 0 can be used to hold any 8-byte value; for instance the Ravi Matrix library uses this to hold two 32-bit integers
- Accessing out of bounds array elements results in error
- Slices can be created from arrays using a library function; a slice maintains a reference to the original array.
- Arrays can never shrink – they can only grow; no way to delete an array element
- Array growth is automatic when value assigned to last+1 slot
- Arrays maintain their length so computing array length is fast
- The normal Lua hash and array parts cannot directly be accessed in array types; however the slice implementation uses the hash part to hold a reference to parent array
- Array indexing can exploit static typing to generate more efficient code
- C API allows direct access to array data

Ravi extension - arrays

```c
/* Following are the types we will use
** use in parsing. The rationale for types is
** performance - as of now these are the only types that
** we care about from a performance point of view - if any
** other types appear then they are all treated as ANY
**/
typedef enum {
  RAVI_TANY = -1,      /* Lua dynamic type */
  RAVI_TNUMINT = 1,        /* integer number */
  RAVI_TNUMFLT,        /* floating point number */
  RAVI_TARRAYINT,      /* array of ints */
  RAVI_TARRAYFLT,      /* array of doubles */
  RAVI_TFUNCTION,      /* Lua or C Function */
  RAVI_TTABLE,         /* Lua table */
  RAVI_TSTRING,        /* string */
  RAVI_TNIL,           /* NIL */
  RAVI_TBOOLEAN,       /* boolean */
  RAVI_TUSERDATA       /* userdata or lightuserdata */
} ravitype_t;
```



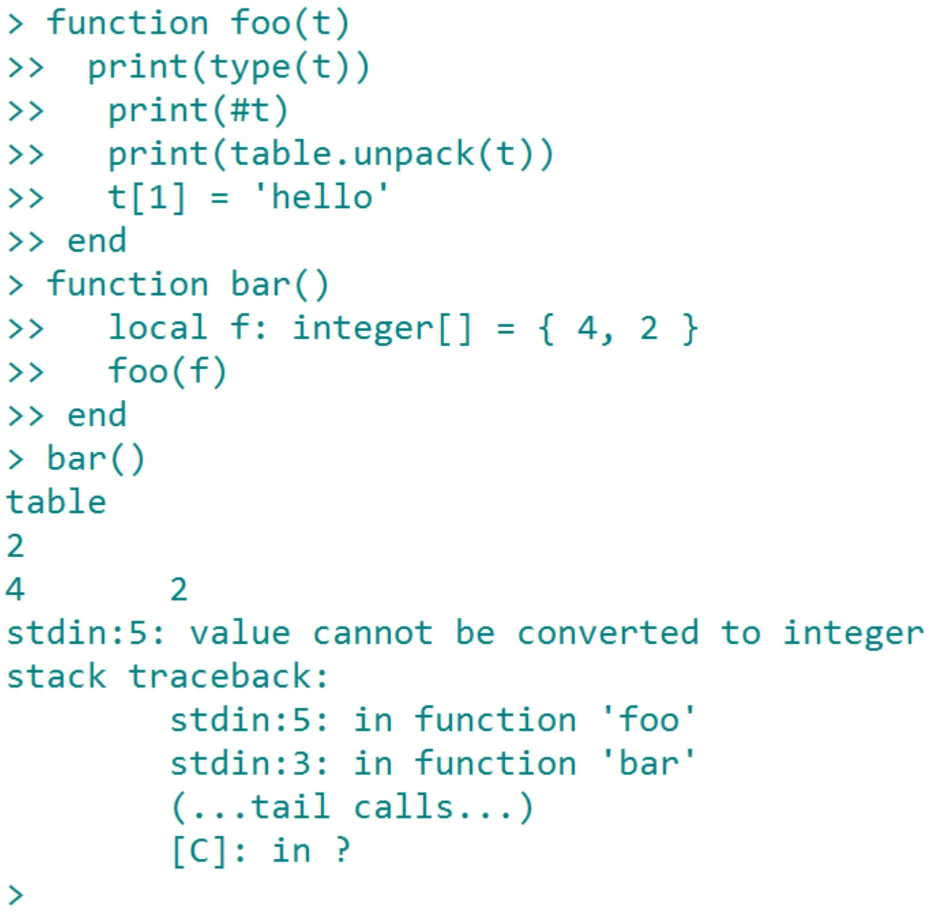
```c
typedef enum RaviArrayModifer {
  RAVI_ARRAY_SLICE = 1,
  RAVI_ARRAY_FIXEDSIZE = 2
} RaviArrayModifier;


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

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

Ravi extension - arrays

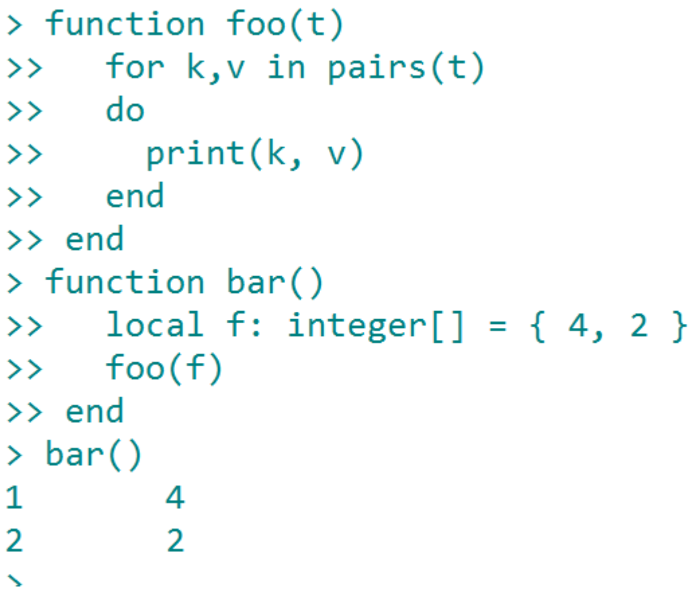
```
> function sum(f: number[])
>>   local n: number = 0.0
>>   for i=1,#f do n = n + f[i] end
>>   return n
>> end
> ravi.dumplua(sum)

function <stdin:1,5> (11 instructions at 000000674CF8E2A0)
1 param, 7 slots, 0 upvalues, 6 locals, 2 constants, 0 functions
        1       [1]     TOARRAYF        0
        2       [2]     LOADK           1 -1    ; 0.0
        3       [3]     LOADK           2 -2    ; 1
        4       [3]     LEN             3 0
        5       [3]     LOADK           4 -2    ; 1
        6       [3]     FORPREP_I1      2 2     ; to 9
        7       [3]     GETTABLE_AF     6 0 5
        8       [3]     ADDFF           1 1 6
        9       [3]     FORLOOP_I1      2 -3    ; to 7
        10      [4]     RETURN          1 2
        11      [5]     RETURN          0 1
constants (2) for 000000674CF8E2A0:
        1       0.0
        2       1
locals (6) for 000000674CF8E2A0:
        0       f       1       12
        1       n       3       12
        2       (for index)     6       10
        3       (for limit)     6       10
        4       (for step)      6       10
        5       i       7       9
upvalues (0) for 000000674CF8E2A0:
```

```
        movabsq $luaV_objlen, %rax
        movq    %r15, %rcx
        callq   *%rax
        movq    32(%rdi), %rdx
        movq    $1, 64(%rdx)
        movl    $19, 72(%rdx)
        movq    32(%rdx), %rax
        movq    48(%rdx), %r8
        cmpq    %r8, %rax
        jg      .LBB0_9
        movq    (%rdx), %rcx
        movl    64(%rcx), %edi
        leaq    (,%rax,8), %rbx
        .align  16, 0x90
.LBB0_6:
        movq    %rax, %rsi
        cmpq    %rdi, %rsi
        jae     .LBB0_12
        movq    56(%rcx), %rax
        movsd   (%rax,%rbx), %xmm0
        movsd   %xmm0, 96(%rdx)
        movl    $3, 104(%rdx)
        addsd   16(%rdx), %xmm0
        movsd   %xmm0, 16(%rdx)
        movl    $3, 24(%rdx)
        leaq    1(%rsi), %rax
        addq    $8, %rbx
        cmpq    %r8, %rsi
        jl      .LBB0_6
        movq    %rsi, 80(%rdx)
        movl    $19, 88(%rdx)
.LBB0_9:
        leaq    16(%rdx), %rdi
```

Ravi extension - arrays



```
> function foo(t)
>>  print(type(t))
>>   print(#t)
>>   print(table.unpack(t))
>>   t[1] = 'hello'
>> end
> function bar()
>>   local f: integer[] = { 4, 2 }
>>   foo(f)
>> end
> bar()
table
2
4       2
stdin:5: value cannot be converted to integer
stack traceback:
        stdin:5: in function 'foo'
        stdin:3: in function 'bar'
        (...tail calls...)
        [C]: in ?
>
```



```
> function foo(t)
>>   for k,v in pairs(t)
>>   do
>>     print(k, v)
>>   end
>> end
> function bar()
>>   local f: integer[] = { 4, 2 }
>>   foo(f)
>> end
> bar()
1       4
2       2
```

Ravi bytecode extensions

- Fornum loops are specialized, especially when index is integer and step is a positive constant (most common use case)
- Bitwise operations are specialized when operands are known to be of integer types
- Numeric operations are specialized when operands are known to be numeric types
- Up-value access is specialized when target is a typed scalar variable
- Array indexing is specialized when types are known at compilation time

```
> function foo(i: integer, j: integer)
>>   local k: integer = i & j
>>   return function(x) k = x end
>> end
> ravi.dumplua(foo)

function <stdin:1,4> (6 instructions at 000000674CFB1CD0)
2 params, 4 slots, 0 upvalues, 3 locals, 0 constants, 1 function
        1       [1]     TOINT           0
        2       [1]     TOINT           1
        3       [2]     BAND_II         2 0 1
        4       [3]     CLOSURE         3 0     ; 000000674CFB1FF0
        5       [3]     RETURN          3 2
        6       [4]     RETURN          0 1
constants (0) for 000000674CFB1CD0:
locals (3) for 000000674CFB1CD0:
        0       i       1       7
        1       j       1       7
        2       k       4       7
upvalues (0) for 000000674CFB1CD0:

function <stdin:3,3> (2 instructions at 000000674CFB1FF0)
1 param, 2 slots, 1 upvalue, 1 local, 0 constants, 0 functions
        1       [3]     SETUPVALI       0 0     ; k
        2       [3]     RETURN          0 1
constants (0) for 000000674CFB1FF0:
locals (1) for 000000674CFB1FF0:
        0       x       1       3
upvalues (1) for 000000674CFB1FF0:
        0       k       1       2
>
```

Ravi Bytecode extensions

MOVE	MOVEI, MOVEF, MOVEAI, MOVEAF
LOADNIL	LOADIZ, LOADFZ
SETUPVAL	SETUPVALI, SETUPVALF, SETUPVALAI, SETUPVALAF
GETTABLE	GETTABLE_AI, GETTABLE_AF
SETTABLE	SETTABLE_AI, SETTABLE_AF, SETTABLE_AII, SETTABLE_AFF
NEWTABLE	NEWARRAYI, NEWARRAYF
ADD	ADDFF, ADDFI, ADDII
SUB	SUBFF, SUBFI, SUBIF, SUBII
MUL	MULFF, MULFI, MULII
DIV	DIVFF, DIVFI, DIVIF, DIVII
BAND	BAND_II
BOR	BOR_II

BXOR	BXOR_II
BNOT	BNOT_I
SHR	SHR_II
SHL	SHL_II
EQ	EQ_II, EQ_FF
LT	LT_II, LT_FF
LE	LE_II, LE_FF
FORPREP	FORPREP_IP, FORPREP_I1
FORLOOP	FORLOOP_IP, FORLOOP_I1
	TOINT, TOFLT, TOARRAYI, TOARRAYF

Performance

Benchmark Program	Lua5.3	Ravi(LLVM)	Luajit 2.1
fornum_test1	9.187	0.31	0.309
fornum_test2	9.57	0.917	0.906
fornum_test3	53.932	4.598	7.778
mandel(4000)	21.247	1.582	1.633
fannkuchen(11)	63.446	4.55	4.751
matmul(1000)	34.604	1.018	0.968

- Above benchmarks were run on Windows 64-bit
- Ravi code made use of static typing
- The LLVM JIT compilation time has been excluded in this comparison

Performance

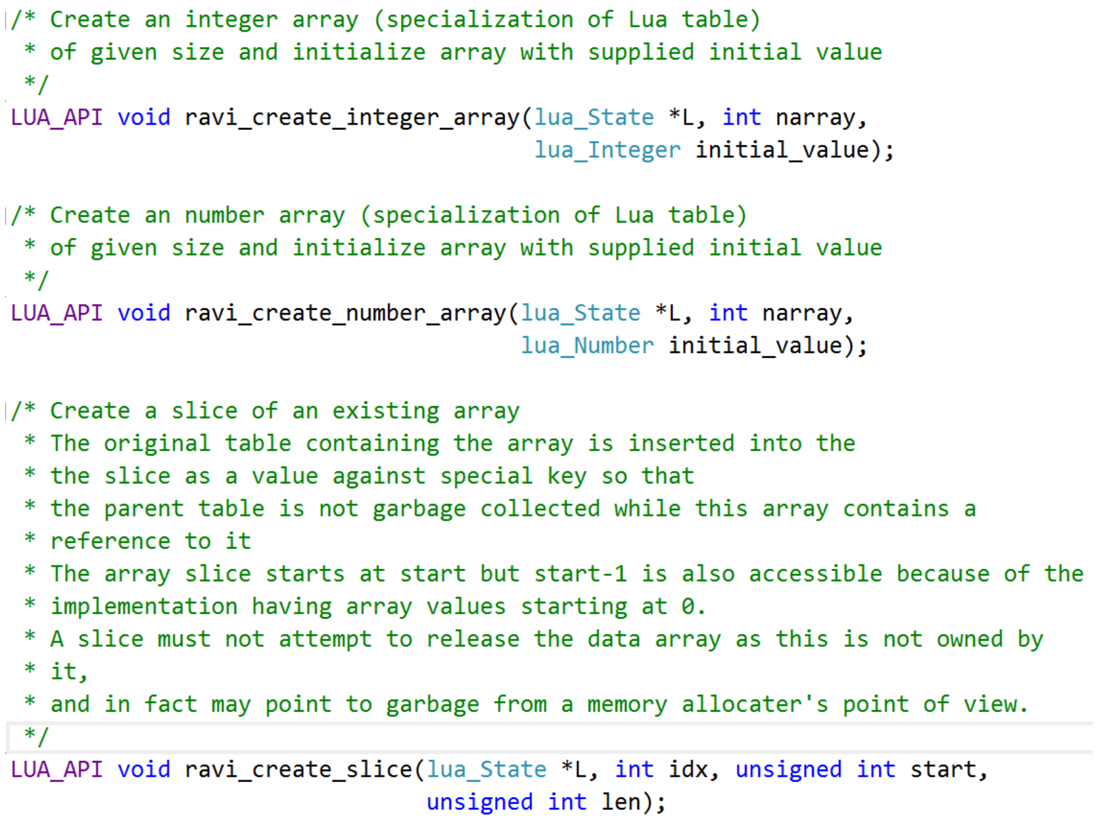
Matmul(1000) implementation	**Timing**	**Remarks**
Lua code interpreted	36.05 seconds	Slightly slower than standard Lua
Lua code JIT compiled	19.06 seconds	Without type information hard to optimise the code
LuaJIT using FFI	0.969 seconds	Equally fast without FFI; includes JIT compilation time
Ravi extensions and JIT compilation	0.986 seconds	Excludes LLVM compilation time and omits array bounds checks on reads
Ravi extensions without JIT	30.7 seconds	Interpreted
Ravi Matrix using OpenBLAS	0.046 seconds	Amazing performance!
Ravi Matrix using userdata metamethod indexing without type checking	93.58 seconds	Slower than interpreted Lua!
Ravi Matrix using userdata with type checking	211 seconds	Type checking uses the optimisation described in Lua mailing list

- Userdata indexing performance is very poor; even interpreted Lua is faster
- **Indexing performance main reason for introducing arrays in Ravi**

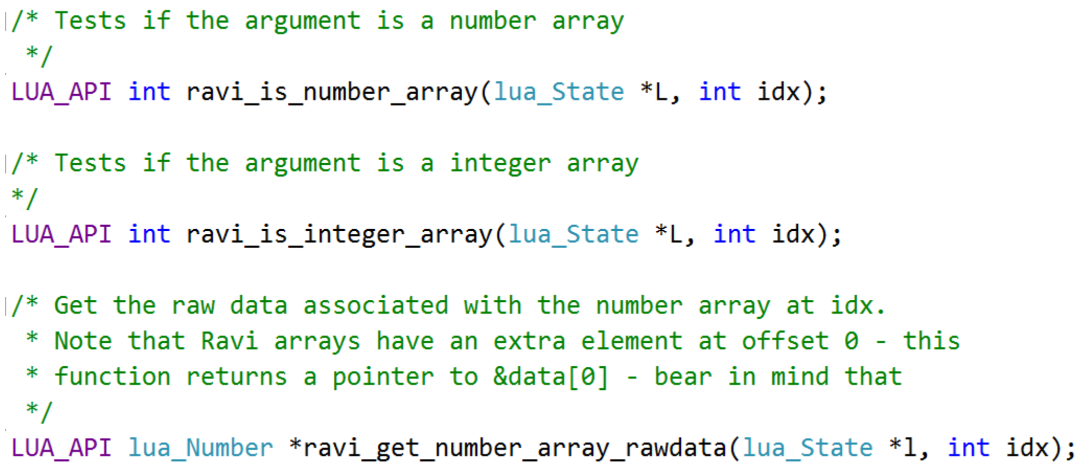
Lua API extensions

- Lua code can call following API functions:
  - `ravi.jit(mode)` – sets JIT on/off; defaults to true
  - `ravi.dumplua(function)` – dumps Lua bytecode
  - `ravi.compile(function)` – JIT compiles a Lua function
  - `ravi.auto(mode[ ,min_size [, min_exeutions]])` – sets auto compilation; defaults are true, 150, 50. Additionally if function has a fornum loop then also JIT compilation is triggered when auto compilation is switched on.
  - `ravi.dumpir(function)` – dumps the LLVM IR
  - `ravi.dumpasm(function)` – dumps the generated assembly code
  - `ravi.optlevel(level)` – sets optimizer level (0-3); default is 2
  - `ravi.sizelevel(level)` – sets code size level (0-3)
  - `table.intarray(num_elements, init_value)` – returns integer[]
  - `table.numarray(num_elements, init_value)` – returns number[]
  - `table.slice(array, start_index, num_elements)` – returns slice, original array memory is frozen (i.e. array cannot be resized anymore due to memory reference)

C API extensions



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

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

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



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

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

/* Get the raw data associated with the number array at idx.
 * Note that Ravi arrays have an extra element at offset 0 - this
 * function returns a pointer to &data[0] - bear in mind that
 */
LUA_API lua_Number *ravi_get_number_array_rawdata(lua_State *l, int idx);
```

LLVM

Pros

- Well documented intermediate representation called LLVM IR
- The LLVM IRBuilder implements type checks so that basic type errors are caught by the builder
- Verifier to check that the generated IR is valid
- CLANG can generate LLVM IR; very useful for checking what the IR should look like

Cons

- LLVM IR is low level – lots of tedious coding required
- LLVM is huge in size. Lua on its own is tiny - but when linked to LLVM the resulting binary is a monster
- Compilation is costly so only beneficial when Lua function will be used again and again
- LLVM must be statically linked

JIT Compilation architecture

- The unit of compilation is a Lua function
- Each Lua function is compiled to a Module/Function in LLVM parlance (Module=Compilation Unit)
- The compiled code is attached to the Lua function prototype (Proto)
- The compiled code is garbage collected as normal by Lua
- The decision to call a JIT compiled version is made in the Lua Infrastructure (specifically in `luaD_precall()` function in `ldo.c`)
- The JIT compiler translates Lua/Ravi bytecode to LLVM IR - i.e. it does not translate Lua source code
- There is no in-lining of Lua functions
- Generally the JIT compiler implements the same instructions as in `lvm.c` - however for some bytecodes the code calls a C function rather than generating inline IR. These opcodes are OP_LOADNIL, OP_NEWTABLE, OP_RAVI_NEWARRAYI, OP_RAVI_NEWARRAYF, OP_SETLIST, OP_CONCAT, OP_CLOSURE, OP_VARARG
- Ravi represents Lua values as done by Lua 5.3 - i.e. in a 16 byte structure

Problem areas

- The Lua program counter (savedpc) is not maintained in JIT code therefore debug API doesn’t work with JITed functions
- Maintaining the program counter would inhibit optimisation; perhaps a debug mode can be implemented
- Co-routines not supported in JIT mode; therefore only main thread executes JITed code; co-routines (secondary threads) always work in interpreted mode. Resuming a JITed function is a hard problem
- Tail calls are implemented as normal calls in JITed code hence tail recursion is limited to a finite depth
- Currently only 64-bit integer implemented

Batteries

- Aim to provide a bunch of standard libraries with Ravi; however these are additional packages rather than part of Ravi
- Work ongoing in following areas:
  - LLVM bindings – users can generate machine code from Lua
  - Ravi-Matrix – wrapper for BLAS and LAPACK libraries; OpenBLAS supported
  - Ravi-GSL – wrapper for GNU Scientific Library
  - Ravi-Symbolic – will wrap SymPy’s SymEngine

Closing thoughts about Ravi

- In Lua, byte-code is generated while parsing – hence it is harder to implement static type checks; so far have managed to workaround issues but the implementation is ugly – not yet confident that all corner cases are handled correctly
- Introducing AST will degrade code generation performance and increase memory usage but on plus side may allow future enhancements such as incorporating a macro facility similar to Metalua
- Lua’s parsing and code generation implementation is one of the most complex parts of Lua; documentation is sparse in this area
- Maintaining compatibility with Lua could be difficult if significant changes occur to the Lua language or implementation; hence need to ensure merging of upstream changes is relatively easy (complete new codebase would cause the issues LuaJIT is having with incorporating upstream changes)
- Ravi as it stands is a specialized dialect for a particular use case (Desktop or Server, numeric computing); this makes it difficult to get others interested in contributing to Ravi (so far no contributions)
- Making a more generic language would entail providing better support for aggregate types; but this is hard to do in Lua due to existing semantics of tables (Wren illustrates how one might approach this)
- LuaJIT, Pure, Julia – all offer easy and efficient FFI; but there is no safe way to offer this in Ravi
- Function calls are expensive in Lua and Ravi – I would love to have a solution for in-lining functions; macros seem the most promising approach
- It would be nice to be able to share generated code across Lua states as JIT compilation is expensive

Closing thoughts about Lua

- Small yet powerful
- Carefully designed implementation
- Somewhat geeky although appears simple at first glance (fornum loops, logical operators, metatables, DIY class systems, co-routines)
- Core VM encapsulated in well defined API – even standard Lua libraries need to go through the API
- Hugely appreciate the availability of the Lua test suite
- Sadly not well known in some programming communities

Links

- http://ravilang.org
- https://github.com/dibyendumajumdar/ravi