DSL compilation with DynASM

Michael Filonenko
Tarantool

IPONWEB

TAMASHI

Introduction into compiling DSL with DynASM

Tarantool

- High performance database
  - in-memory storage
  - disk storage when RAM amount is not enough

Tarantool API

- Storing data
- Indexing
- Iteration over index

Data query

- Iteration over index
- Filtration

Examples: storing data

```
create_table('People', {'id', 'age',
                                'profession'})
Peoples.index:add('age')
Peoples:insert({1, 23, 'driver'})
Peoples:insert({2, 31, 'musician'})
```

Examples: query

```
iterator = People.indexes.age(30)
for _, tuple in pairs(iterator) do
    if tuple.age > 40 then
        break
    end
    if tuple.profession == 'musician' then
        send(tuple)
    end
end
```

Query from table of people

- For each person with age between 30 and 40
- Filter out only musicians

How to make filtration really fast

Tarantool API

- C
- Lua

Tarantool C API

- Pro: fast filter function
  - Compiled
- Cons: fiddly query modification
  - Recompilation
  - Restarting application

Tarantool Lua API

- Pro: easy query modification
- Con: slower (despite LuaJIT!)
  - Dynamic types
  - Automanaged memory

Domain-specific language

- Pros:
  - Easy query construction
  - Fast
    - Just in time compilation
    - Static types
    - Hard-coded memory management

Roadmap

- Parser of user input
- Abstract syntax tree generation
- Compilation of abstract syntax tree

User input: s-expressions (lisp)

- Atoms
- Lists: can contain atoms and others lists

Examples: s-expressions atoms

- 12
- "Hello world"
- +

Examples: s-expressions lists

- (do-nothing)
- (+ 1 2)
- (- 4 (+ 1 3))
- (= Hello aloha)

Parsing

- Iterate over string capturing atoms and lists (tokenisation)
- Convert to Lua tables
- Validate syntax
- Save metainformation about atoms and lists positions to Lua metatables

Examples: parsing

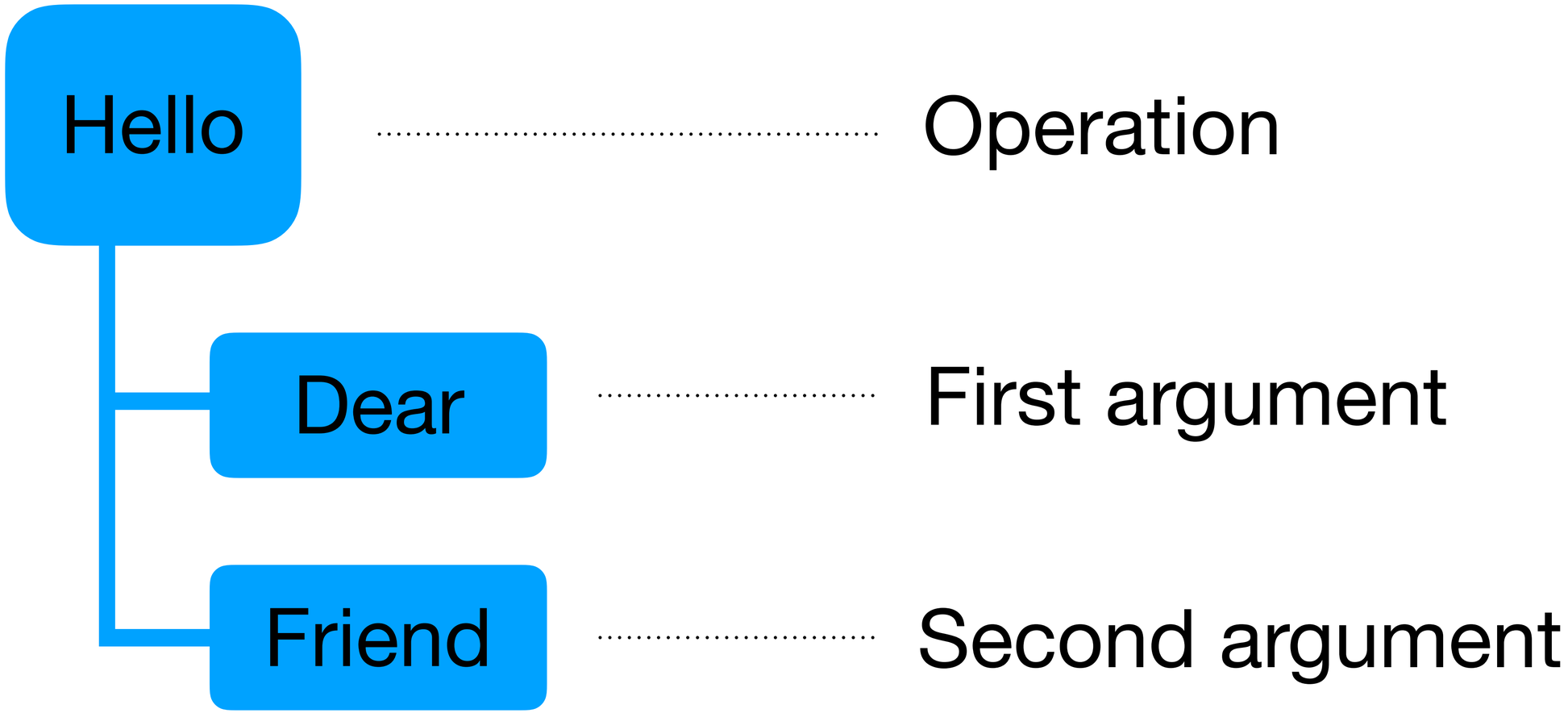
"(= Hello aloha)"

{'=', 'Hello', 'Aloha'}

Abstract syntax tree (AST)

- the node of tree is operation
- and node children are arguments for operation

Examples: AST

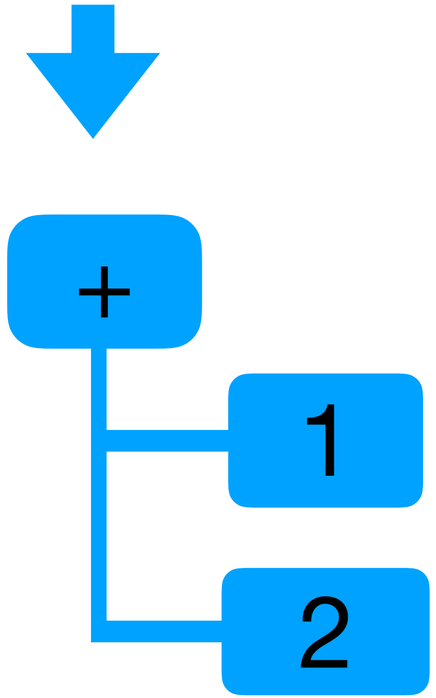


S-expression to AST conversion rules

- first list element is operation
- other elements are arguments

Examples: s-expression to AST

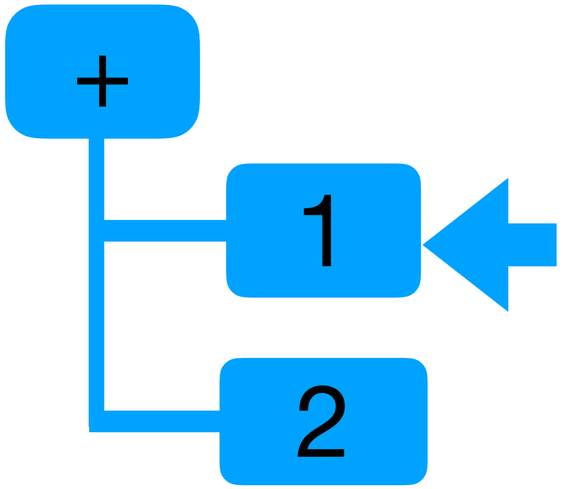
(+ 1 2)



Stack-based AST compiler

- traverses the tree and evaluates it:
  - argument nodes are pushed onto stack as they appear in the tree
  - operation nodes pop operands, produce the result and push it back

Examples: AST evaluation

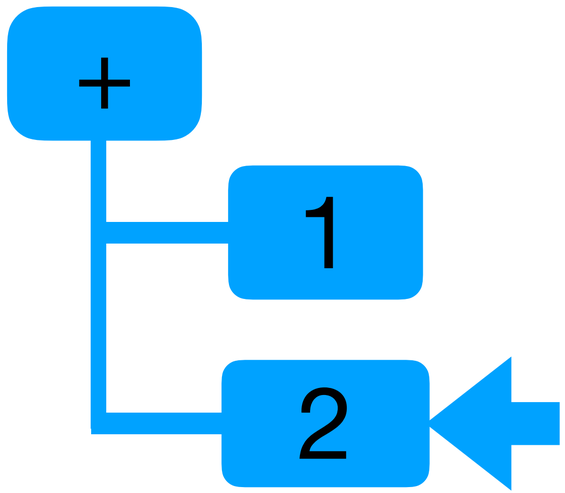



```
push 1 to stack

push 2 to stack

add stack[1] and stack[2] to temp

pop all arguments

push temp
```

AST

pseudocode

Examples: AST evaluation

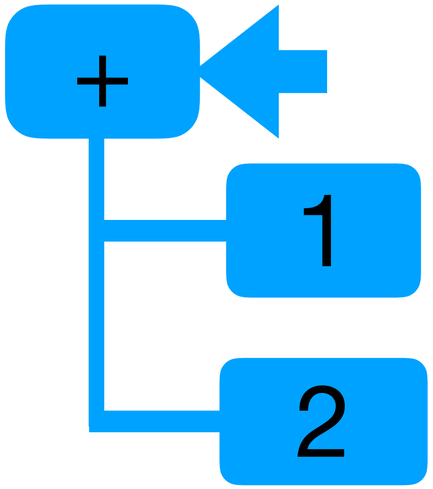



```
push 1 to stack

push 2 to stack

add stack[1] and stack[2] to temp

pop all arguments

push temp
```

AST

pseudocode

Examples: AST evaluation




```
push 1 to stack

push 2 to stack

add stack[1] and stack[2] to temp

pop all arguments

push temp
```

AST

pseudocode

DynASM toolset

- Core - libdasm
  - Compiling and linking code
- Binding - dasm.lua
  - Foreign function interface to core
- Translator of mixed code - dynasm.lua
  - Preprocess Lua/assembly code

Generating code with DynASM

- Code generator (not inline assembly)
- Mixed code: Lua and assembly

```
function nop(destination, ...)
  |nop  -- write to destination
  |ret
end
```

Preprocessing and loading mixed code

```
local dynasm = require('dynasm')
local gen = dynasm.loadfile('gen.dasl')()
```

Compiling code with DynASM

```
-- allocate memory
local state, globals = dasm.new(nop....)

-- generate machine code
gen.nop(state)

-- link references
local buf, size = state:build()
```

Running generated code

```
local ffi = require('ffi')

local callable = ffi.cast('void __cdecl (*) ()', buf)
callable()
```

full Lua/assembly file - x64.dasl

```
local ffi = require('ffi') -- required
local dasm = require('dasm') -- required

-- must be the first instruction
-- setup assembler architecture (x86_64)
|.arch x64
-- container for instruction binary representation
|.actionlist actions
-- container for global labels
|.globalnames globalnames

local gen = {}

function gen.nop(Dst)
        |nop
        |ret
end

function gen.int3(Dst)
        |int3
end

return {gen = gen, actions = actions, globalnames = globalnames}
```

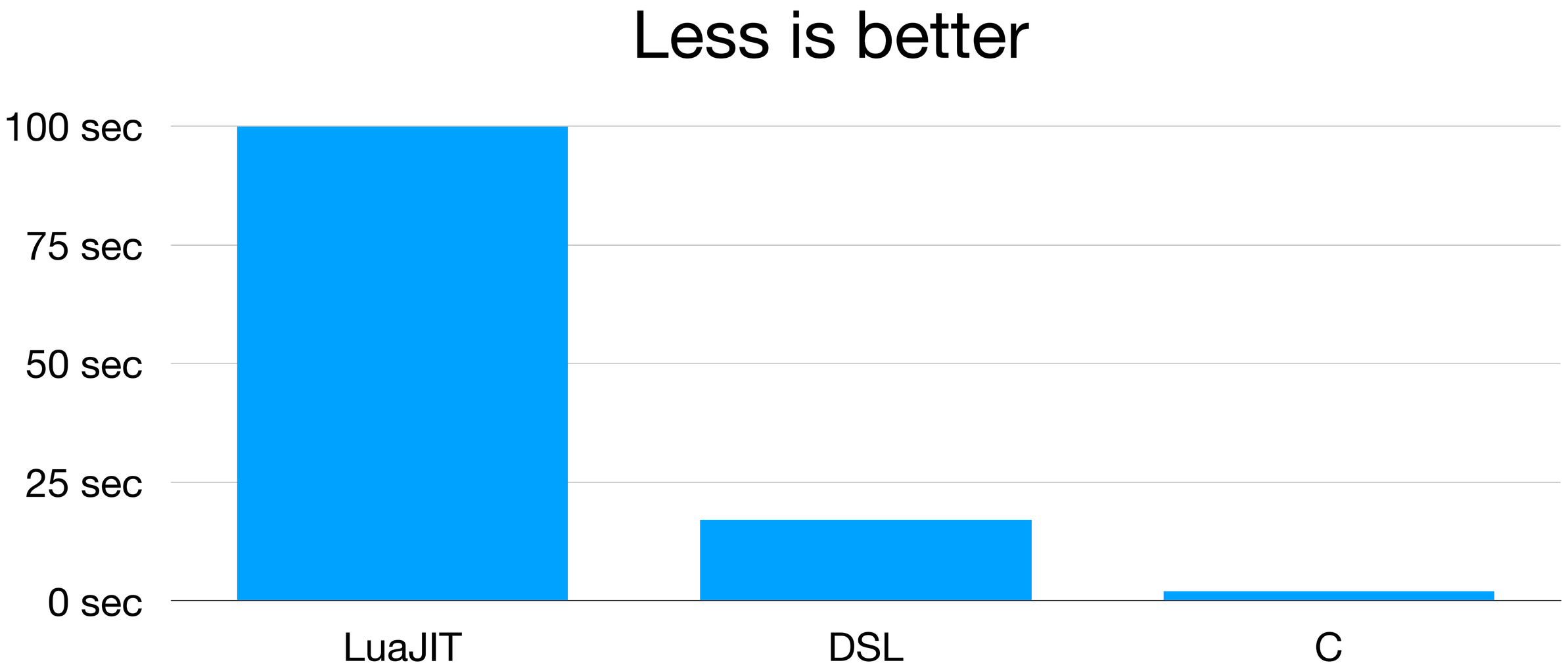
full compiler file - compile.lua

```
local ffi = require('ffi')
local dasm = require('dasm')
local dynasm = require('dynasm')

-- load generators
local x64 = dynasm.loadfile('x64.dasl')()

-- make compiler state from generators
local state, globals = dasm.new(x64.actions)

-- generate code
x64.gen.nop(state)

--check, link and encode the code
local buf, size = state:build()

local callable = ffi.cast('void __cdecl (*) ()', buf)
callable()
```

stack-based s-expression compiler skeleton

```
local free_reg = 0 -- rax

function push(Dst, imm) -- push atom
         |push imm
end

function add(Dst, context)
         |pop Rq(0)
         |add qword [rsp], Rq(0)
end

function compile(Dst, ast)
  if type(ast) ~= 'table' then
    push(Dst, tonumber(ast))
  end

  if ast[1] == '+' then
    compile(ast[2])
    compile(ast[3])
    add(Dst)
  end
end
```

Benchmarks - scanning 16m rows



Conclusions

- 3 steps to make compiler from s-expressions to assembler
- embedded into LuaJIT

github.com/filonenko-mikhail/dynasm

Questions?

DynASM

- luajit.org/dynasm.html
- corsix.github.io/dynasm-doc/
- http://bit.ly/lua-dynasm

x86_64 assembly

- felixcloutier.com/x86
- http://bit.ly/call-contract
- http://bit.ly/x64-example