

20th years of YARV

Koichi Sasada
STORES, Inc.



2004/01/06

Ruby-dev mailing list

Subject: [ruby-dev:22494]
[ANN] YARV: Yet another
RubyVM 0.0.0-

- New year's holidays he has a time to hack
- He had experienced JVM impl. in C++ and in Ruby (as his first Ruby app)
- He had studied CRuby internal, and he had thought that he can make **better** VM

From: "K. Sasada" <ko1@...>
Date: 2004-01-06T04:52:33+09:00
Subject: [ruby-dev:22494] [ANN] YARV: Yet another RubyVM 0.0.0-

あけましておめでとうございます。

お正月に次の拡張ライブラリを作ったので投稿させていただきます。

YARV: Yet another RubyVM 0.0.0-
<http://www.atdot.net/yarv/yarv000minus.lzh>

(で、全然一般向けじゃないような気がするので dev に投げました)

最近の cvs ruby でしか動作確認していません。

Ruby の VM というのもおこがましいほど、まだ何もできないんですが、とりあえず、メソッドを定義して動いた一っつてことで。


しかも、遅いです。たくさんサボってるのに遅いです。

```
-----  
def fib(n)  
  if n < 2 then  
    1  
  else  
    fib(n-2) + fib(n-1)  
  end  
end  
fib(32)  
  
#=>  
25.500000  0.160000  25.660000 ( 25.947715) # orig ruby  
19.650000  0.000000  19.650000 ( 21.261890) # yarv  
-----
```

ご笑覧下されば幸いです。

```
--  
// SASADA Koichi @ namikilab.tuat.ac.jp  
//  
// YARV日記 :  
// http://www.namikilab.tuat.ac.jp/~sasada/diary/200401.html
```

After YARV



- 2004/Oct RubyConf 2004 1st YARV talk in US
- 2007/Dec Ruby 1.9.0 (dev version) released with YARV
- 2008/Dec Ruby 1.9.1 (prod. version) released with YARV
- 2018/Dec Ruby 2.6.0 released with MJIT
- 2021/Dec Ruby 3.1.0 released with YJIT and MJIT
- 2023/Dec Ruby 3.3.0 released with YJIT and RJIT
- 2024/Dec Ruby 3.4.0 will be released

YARV INSIDE
in your “`ruby`” command

Koichi Sasada

- Ruby interpreter developer employed by **STORES, Inc.** (2023~) with @mametter
 - YARV (Ruby 1.9~)
 - Generational/Incremental GC (Ruby 2.1~)
 - Ractor (Ruby 3.0~)
 - debug.gem (Ruby 3.1~)
 - M:N Thread scheduler (Ruby 3.3~)
 - ...
- Ruby Association Director (2012~)



\$1 a month

Select



<https://github.com/sponsors/ko1>

Shortest description ever.



- 2004/Oct RubyConf 2004 1st YARV talk in US
 - **2006/Apr Becomes a faculty member of a University**
 - 2007/Dec Ruby 1.9.0 (dev version) released with YARV
 - 2007/Dec Completed my Ph.D. with YARV
 - 2008/Mar **EuRuKo 2008 @ Prague**
 - 2008/Dec Ruby 1.9.1 (prod. version) released with YARV
 - **2012/Apr Hired by Heroku/Salesforce**
 - 2012/Dec Ruby 2.1 with **Generational GC**
 - 2013/Jun **EuRuKo 2013 @ Athene**
 - 2015/Oct **EuRuKo 2015 @ Salzburg**
 - **2017/Jan Hired by Cookpad**
 - 2018/Dec Ruby 2.6.0 released with MJIT
 - 2020/Dec Ruby 3.0.0 released with **Ractor**
 - 2021/Dec Ruby 3.1.0 released with YJIT and MJIT
 - **2023/Sep Hired by STORES**
 - 2023/Dec Ruby 3.3.0 released with M:N threads, YJIT and RJIT
 - 2024/Sep **EuRuKo 2024 @ Sarajevo**
 - 2024/Dec Ruby 3.4.0 will be released
- I changed my job, but the work is always Ruby implementations.
 - Many opportunity to talk at EuRuKo (4th time). I could visit many wonderful places

20th years of YARV: Yet Another RubyVM

Basic ideas

Good and bad points

Future work

An ordinal view of Ruby programmers

**Ruby (Rails)
app**

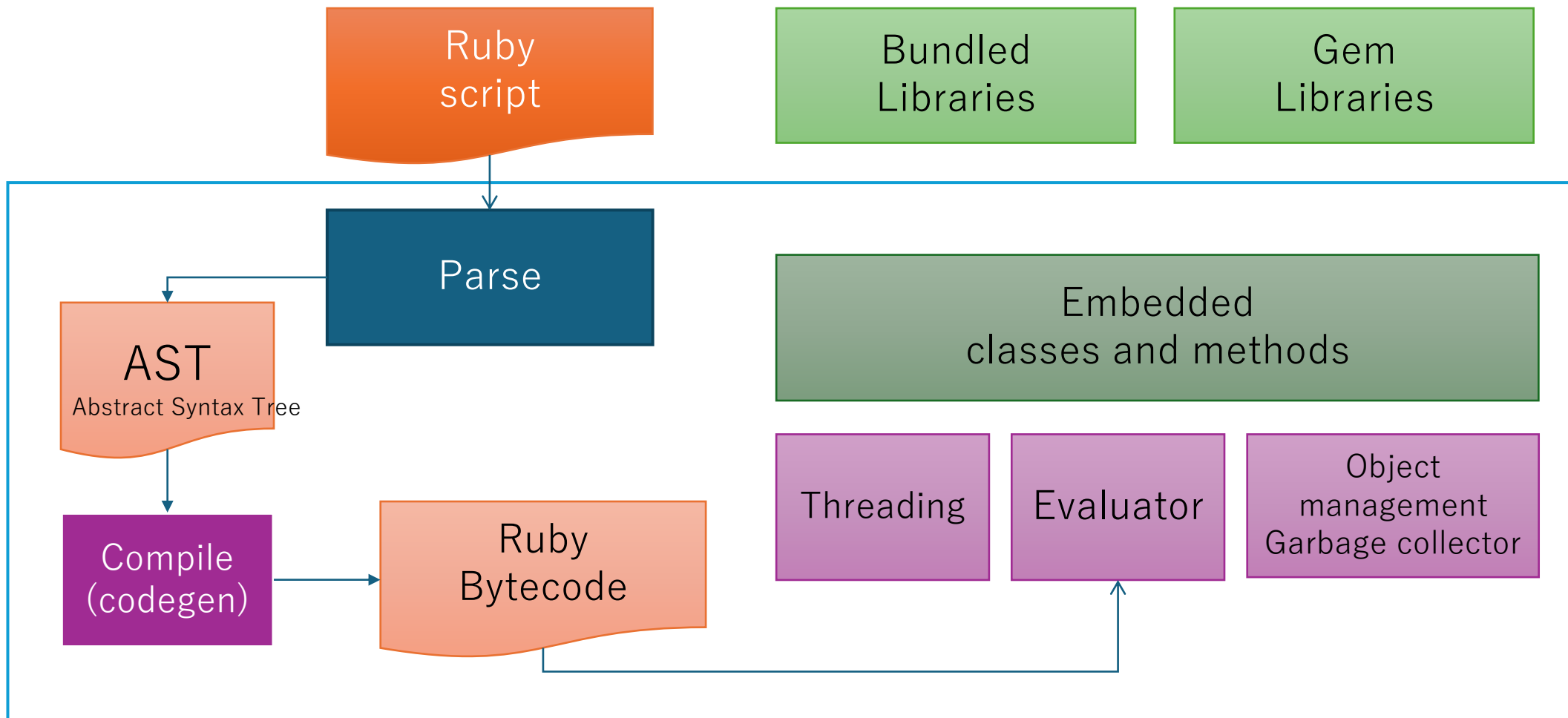
i gigantum umeris insidentes
Standing on the shoulders of giants

So many gems
such as Rails, puma, ... and so on.

RubyGems/Bundler

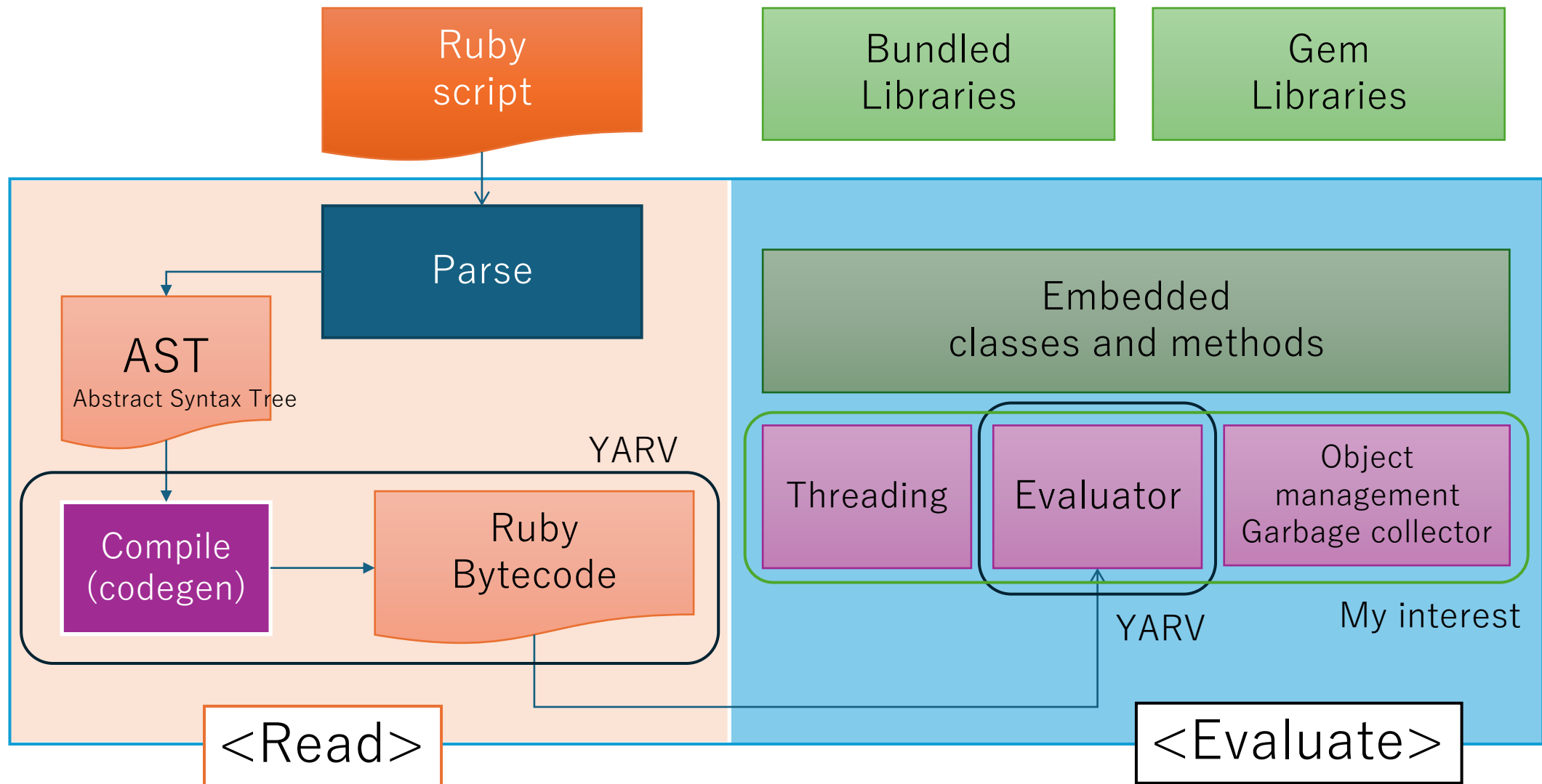
Ruby interpreter

An ordinal view of MRI developers



Operating system under the interpreter

Workflow of Ruby's interpreter



FAQ: Why “Yet Another”?

- There were several Ruby’s virtual machines in 2004
 - Rite, Byte code Ruby, …
 - Only YARV completes the implementation to run Ruby
- CS people like “Yet Another” (e.g. “yacc”)

YARV

Basic ideas

Stack based virtual machine

Auto generation from instruction definitions

Optimizations

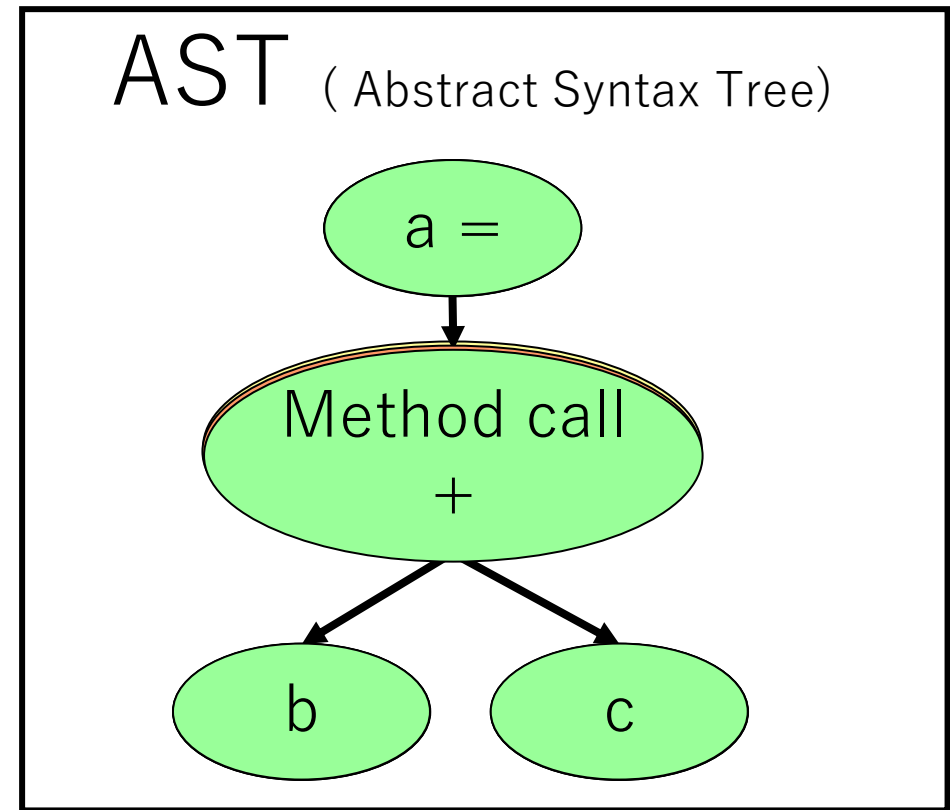
Basic idea

Stack machine

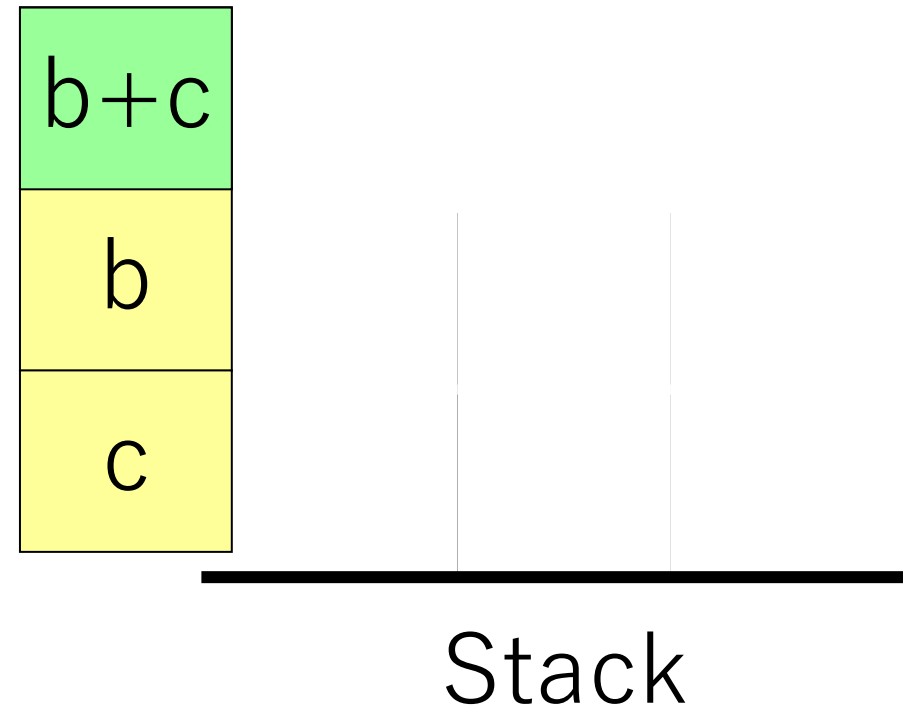
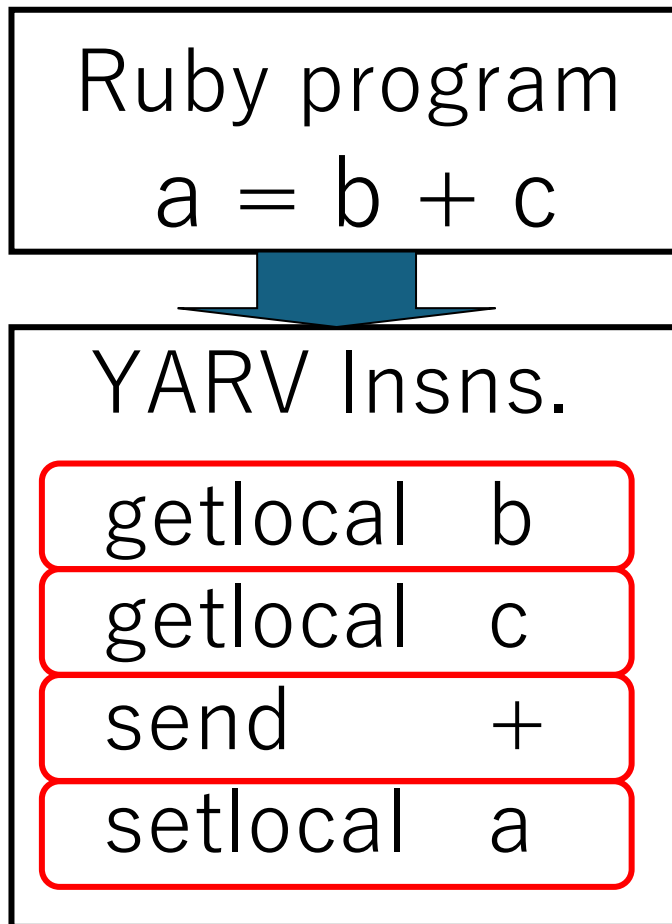
- All operations uses a stack
 - Simple because no register allocation issues
- vs. register machines
 - Computation uses registers
- JVM, .NET, Smalltalk VM, p-code, ...

Ruby before YARV (~Ruby 1.8)

Ruby program

$$a = b + c$$


Stack machine



Basic idea

Automatic generations

- Generate many files from instruction definition (insns.def)
 - VM execution code
 - Complete C code [“vmgen: a generator of efficient virtual machine interpreters” by M. Anton Ertl, 2002]
 - Optimized instructions
 - JIT compiled code by MJIT
 - VM meta data (instructions list and so on)
 - Disassembler
 - Serializer
 - Documents
- Designed for “easy modification”

Automatic generations based on abstract instruction definitions

```
// insns.def
```

```
DEFINE_INSN
```

```
getlocal
```

```
(lindex_t idx, rb_num_t level) (Instruction operands)  
() (Popped values from stack)  
(VALUE val) (Push values to stack)
```

```
{  
    val = *(vm_get_ep(GET_EP(), level) - idx);  
    RB_DEBUG_COUNTER_INC(lvar_get);  
    (void)RB_DEBUG_COUNTER_INC_IF(lvar_get_dynamic, level > 0);  
}
```

```
// simplified automatic generated vm.inc
```

```
INSN_ENTRY(getlocal)
```

```
{  
    lindex_t idx = (lindex_t)GET_OPERAND(1); <fetch operands>  
    rb_num_t level = (rb_num_t)GET_OPERAND(2);  
    const bool MAYBE_UNUSED(leaf) = INSN_ATTR(leaf);  
    VALUE val;  
    ADD_PC(INSN_ATTR(width));  
    <body>  
    val = *(vm_get_ep(GET_EP(), level) - idx);  
    RB_DEBUG_COUNTER_INC(lvar_get);  
    (void)RB_DEBUG_COUNTER_INC_IF(lvar_get_dynamic, level > 0);  
    TOPN(0) = val;  
    END_INSN(getlocal); <push to stack code>  
}
```

Automatic generations based on abstract instruction definitions

- Easy to modify and introduce instructions
 - No need to write boring code like stack pushing and popping
 - No need to modify disassembler and so on
- Easy to introduce optimization
 - Operands unification (specialization)
 - `getlocal 1 -> getlocal1`
 - Instructions unification
 - `putnil ; setlocal → putnil_setlocal`
 - Stack caching [“Stack caching for interpreters” by M. Anton Ertl. S, 1995]
 - Caching nth top of stack on CPU registers
 - Easy to select instruction dispatch method
 - switch/case in standard C
 - Dynamic threading with gcc extension

Basic idea Optimizations

- Ruby specific characteristics we need to know
 - The nature of dynamic
 - Many method invocations
 - Many method invocations with a block

Optimization

The nature of dynamic – The limitations

- Add and modify the definitions while executing the program
 - All class/method definitions are dynamic changes in Ruby
 - Meta-programming features like `Class.new`, `define_method`, ...
 - Many dynamic code execution like `eval`, `load`, ...
- Limitation on optimization
 - “1 + 2” can returns anything by redefinitions

Optimization

Many method invocations

- Most of operations are abstracted by method dispatch
 - Accessing instance variables (obj.foo)
 - Accessing Array (Array#[[]])
 - ...
 - So we need to improve the performance of it
- Approaches
 - Method caching ([My EURUKO2015 talk])
 - Caching the method body
 - Caching the method invocation function
 - Specialized instructions
 - Introduce special instructions for some methods

Optimization

Many method invocations with a block

- With block, local variables should be saved correctly (escaping) in heap
 - `def f(&b)=b; def g(i)=f{p i}; ...; g(0).call`
- However, most of case the escaping is not needed
 - `path = ...; open(path) {...}`
- YARV delayed the escaping if it is needed
 - Remain all local variables on a stack
 - Escape all accessible local variables **when Proc is created**
 - 👍 This technique improves the performance
 - 👎 This technique introduces huge complexity

Other optimizations on YARV

- Dynamic dispatch for VM instruction dispatch
- Peephole optimizations
- No penalty “rescue/ensure” execution
- Inline constant cache
- Deduplicate frozen literal objects
- Lazy Proc creation for a block parameter
- On the fly trace instruction insertion
- Support dumping and loading instructions
- ...

Good and Bad points

My achievements and regrets

Good achievement

Define how to run Ruby in instructions

- Defining “Ruby” with stack machine instructions
 - Ruby has **many specifications**, such as complex method and block parameters, complex flow control mechanism, special cases for the “ease of use” and so on
 - Before YARV, **nobody had known** which instructions are needed to represent Ruby’s flexible (cursed as VM implementor perspective) functionalities completely



Ruby

A PROGRAMMER'S BEST FRIEND

But no friend to interpreter developers

Good achievement

Define how to run Ruby in instructions

- Many optimizations are allowed such as I listed before
- Some tools relies on it
 - YJIT/MJIT
 - Typeprof v1
 - debug.gem
 - ...

FYI: Method parameters

- `def f(m1, o1=..., *r, p1, k1:..., rk1:, **rkw, &b)`
 - `m1`: mandatory parameter(s)
 - `o1`: optional parameter(s)
 - `r`: rest parameter
 - `p1`: post parameter(s)
 - `k1`: keyword parameter(s)
 - `rk1`: required keyword parameter(s)
 - `rkw`: rest keyword parameter
 - `b`: block parameter
- Other specs on method parameters
 - Anonymous parameters (`..., *, **, &`)
 - `def f(_, _, _) = nil`
 - `def f((a, b)) = nil`

FYI: Block parameters

- `iter{|m1, o1=..., *r, p1, k1:..., rk1:, **rkw, &b; l1|`
 - Can you explain what happens?
 - It depends on how to yield the block
 - `def iter = yield(expr)`
- The following code has different meanings
 - `iter{|m1|}`
 - `iter {|m1,|`

FYI: Readable instruction names of VM

- “getlocal” instead of magical naming like “gl”
 - I frustrated to read magical assembly language
 - There is no storage limitation now a day

Bad point

Not well-defined instructions

- JVM and other VMs defines instructions
 - Well-considered instructions
 - Many tools assume this specification
 - Compatible \Leftrightarrow Difficult to change
- YARV is designed as “easy-to-change”
 - “We can change if it is not good”
 - Not well-considered, not compatible
- Not enough “foundation” for tools
 - In fact, YJIT relies on current specifications, so it is hard to modify now a day

BTW why no JIT compiler from Koichi?

- I don't like to write assembly directly (hard for me :p)
- I had thought JIT was not valuable to pay an effort more
 - Planned and there are some JIT compiler proposals with C compiler but only in academic papers
 - JIT compiler only speed up $n\%$ ($n < 10$? low score) on real application
 - GC was huge overhead (before generational GC)
 - Need an effort about and it is hard with **limited development resource**
 - Support multiple CPU architectures
 - Continuous maintenance
 - Catch up VM and other changes
- Surprised YJIT team did it, and I respect the achievements

FAQ: Why VM? Why not JIT?

Large Applications, Frequent Changes



> 100 million
lines of Hack/PHP

A new version every
75 minutes¹



shopify

> 3 million
lines of Ruby

30-40 times a day
(every 36-48min)²



> million
lines of Python⁴

30-50 times a day³

1. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. Guilherme Ottoni, Bin Liu. CGO'21

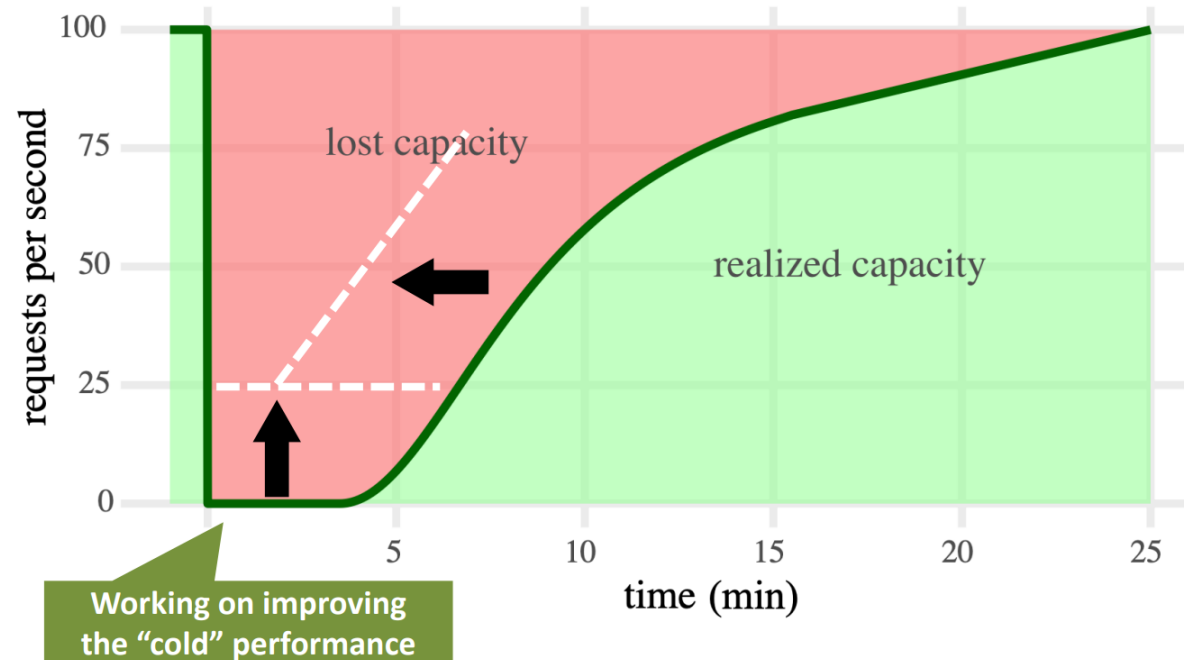
2. <https://shopify.engineering/automatic-deployment-at-shopify>

3. <https://instagram-engineering.com/continuous-deployment-at-instagram-1e18548f01d1>

4. <https://instagram-engineering.com/let-your-code-type-hint-itself-introducing-open-source-monkeytype-a855c7284881>

FAQ: Why VM? Why not JIT?

Building Faster Interpreters



Good achievements

Optimize method dispatch

- Inline method caching
 - Huge optimizing effort
 - [K.Sasada: Lightweight Method Dispatch on MRI, EURUKO 2015]
 - Rewriting several times
 - Invalidation algorithm
 - Ractor-safe (multi-thread safe) algorithm

Bad: No method inlining

- To eliminate the method dispatch overhead, introducing method inlining is straightforward approach, but not yet
 - `def foo = nil; foo(); foo(); foo #=> no-op code`
 - Because it is hard to handle exceptional cases
 - Getting backtrace information
 - Exception handling
 - ...
- YJIT does it for some cases
- In future, even on VM we can do more

Good achievement

Optimization: Specialized instructions

- Introduce special instructions for some methods
 - Lightweight and frequent built-in methods
 - We can omit general method invocation overhead
 - About 26 instructions
 - Integer#+, Float#+, String#+, ...
 - Array#[], Hash#[], ...
 - String#empty?, String#size, Array#size, Object#nil?, ...
 - Check redefinitions each time
- Good: Easy to implement and huge performance impact
 - 2004/01/11 [ruby-dev:22565] 3x faster on fib

Specialized instructions

An example of #+

```
vm_opt_plus(VALUE recv, VALUE obj)
{
    if (FIXNUM_2_P(recv, obj) &&
        BASIC_OP_UNREDEFINED_P(BOP_PLUS, INTEGER_...)) {
        return rb_fix_plus_fix(recv, obj);
    }
    else if (FLONUM_2_P(recv, obj) &&
             BASIC_OP_UNREDEFINED_P(BOP_PLUS, FLOAT_...)) {
        return DBL2NUM(RFLOAT_VALUE(recv) + RFLOAT_VALUE(obj));
    }
    else if ...
    else // normal method dispatch
}
```

Specialized instructions

Bad: Poor extensibility

- “Lightweight and frequent built-in methods”
 - How lightweight? How frequent? It depends on the app
 - How many instructions can we add?
 - Why not user defined methods?
- Slows down other methods
 - `UserDefinedClass#+`
- Future possibilities
 - Introduce more general framework to by-pass general method invocation
 - Inlining lightweight methods with JIT

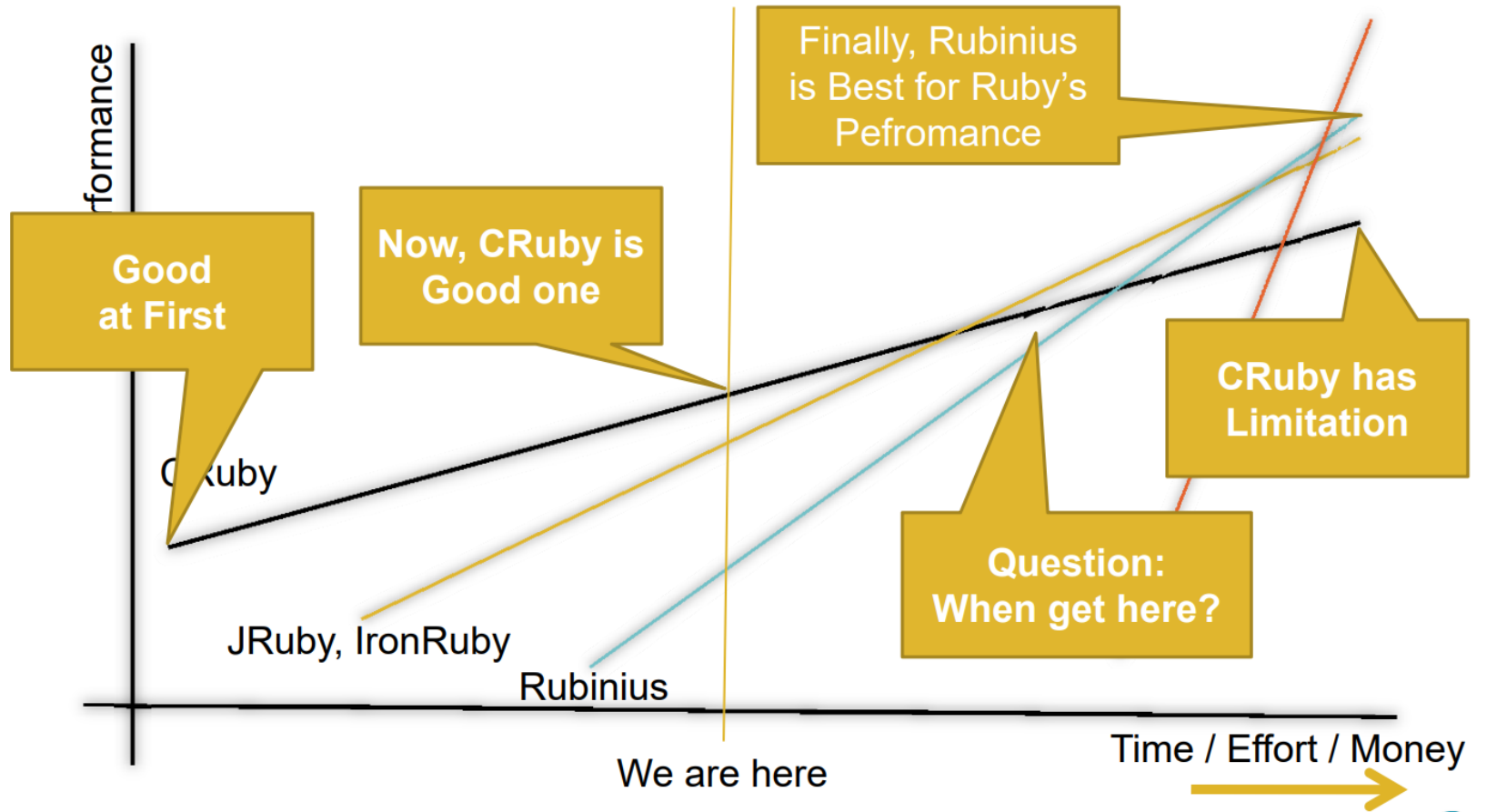
Good and Bad Block management

- Good achievement: Lazy Proc creation on method call with block. It is one of YARV's invention
 - Calling a method with a block is faster as a normal method dispatch
 - Bad point: Not enough optimizations for block dispatch (yielding a block)
 - Method dispatch is well optimized, but...
 - “Block dispatch” (yielding a block) code is complicated
 - Do you know the specification of block's parameters of Ruby? It is nightmare.
 - And not well optimized
- Rooms for easy performance improvement

Good and Bad Handle instructions

- Good achievement: Providing ways to handle instructions
 - `RubyVM::InstructionSequence#disasm` `#=> String`
 - Return human readable instruction sequence for debugging
 - `RubyVM::InstructionSequence#to_a` `#=> Array`
 - Easy to handle by Ruby
 - No loader in default because there is no verifier yet
 - `RubyVM::InstructionSequence#to_binary` `#=> String`
 - Fast loading representation
 - bootsnap uses it
- Good achievement: `prelude.rb`
 - Support writing built-in methods in Ruby (not in C)
 - Embedding instructions in bytes acquired by “`to_binary`”

Evolution of VM Performance My Prediction



2008/12/11

Future of Ruby VM - RubyConf2008



Quoted from "Future of Ruby VM" by Koichi Sasada

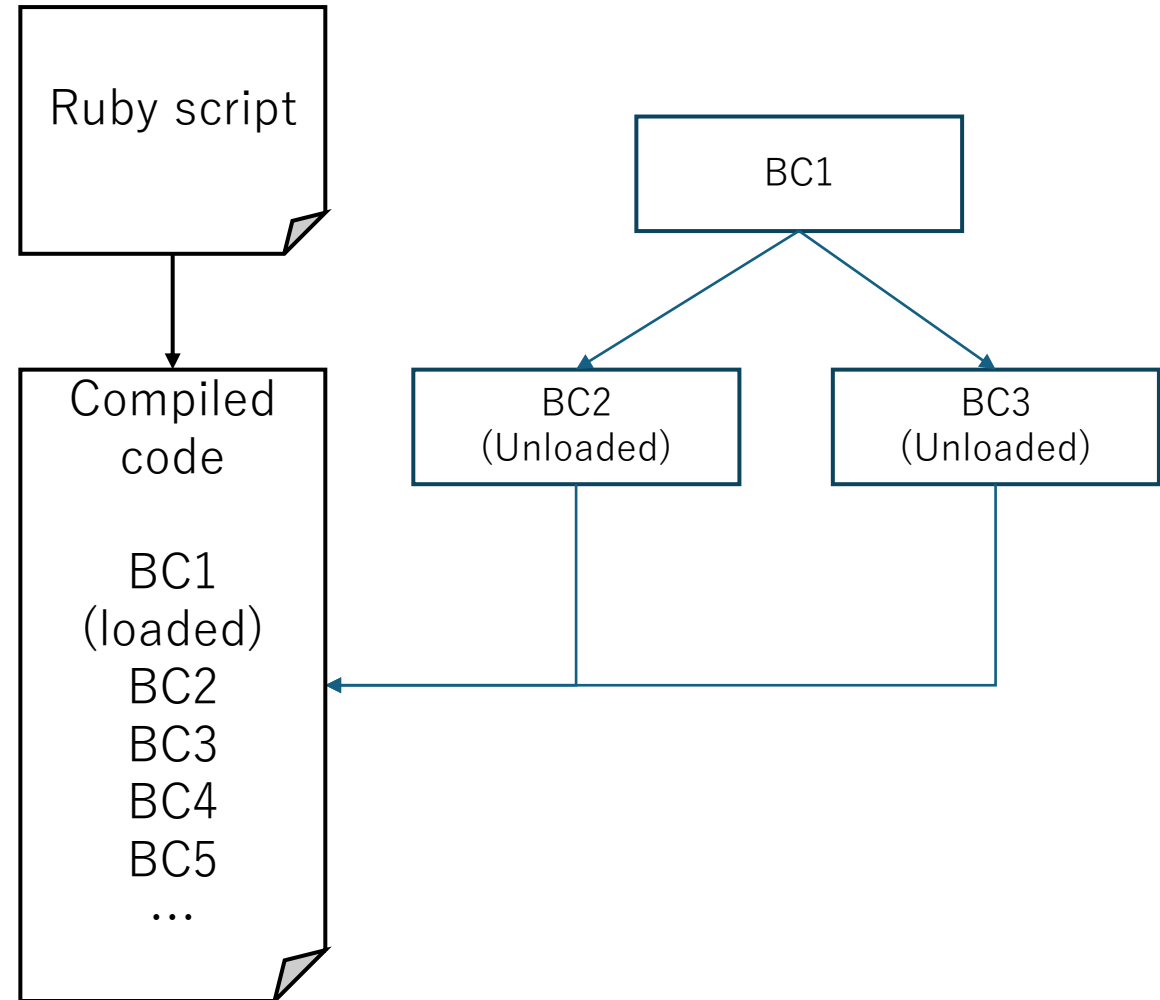
Good and Bad Handle instructions

- Bad: Need to “load” at boot time
 - Loading binary at boot time
 - Now it is not an issue because only a few methods are written in Ruby
 - Run class/method definitions in Ruby
- Future possibilities
 - We can make lazy loading
 - Remain built-in methods in bytes and load “when it is called”
 - All gems for an application can be packed into one binary and utilize lazy loading technique to improve boot time

Implementation technique

Lazy loading

- Load bytecodes on demand
- Make “unloaded” empty BC
 - Points compiled code
- Load bytecode when it is needed
- To execute BC1, empty BC2 and BC3 are created, BC4 and BC5 is not created completely



Future work

- Further optimizations
 - Method inlining
 - Optimize “yield”
 - Faster bootstrap by pre-compiled code with lazy loading
- Flexible JIT compiler
 - Easy development
 - Faster bootstrap

Research project

Flexible JIT compiler

- Inherit YARV's "easy modification" design
- Challenges
 - Support all Ruby's specifications
 - No duplicate definitions
 - Fast runtime performance
 - Fast warmup time
 - Organize a research team

Look back 20 years

- I got a happy chance to participate Ruby development
 - Many interesting hacking topics: VM, GC, Thread
 - Thanks to Ruby's architecture knowledge I can try many hacks
- Great experience and life events
 - Many conferences
 - Got job(s. 4 jobs!)
 - Got married and got children
- I can only have gratitude for happy 20 years

Conclusion

- YARV development in 20 years
 - There are many achievements and regrets
 - Rooms to improve more
- Thank you for using Ruby/YARV

