YJIT's Three Languages

Multiple Representations, All At Once

Noah Gibbs, 2022
RubyConfTH
ruby.social/@codefolio

First: Whoah

Who's This Guy?

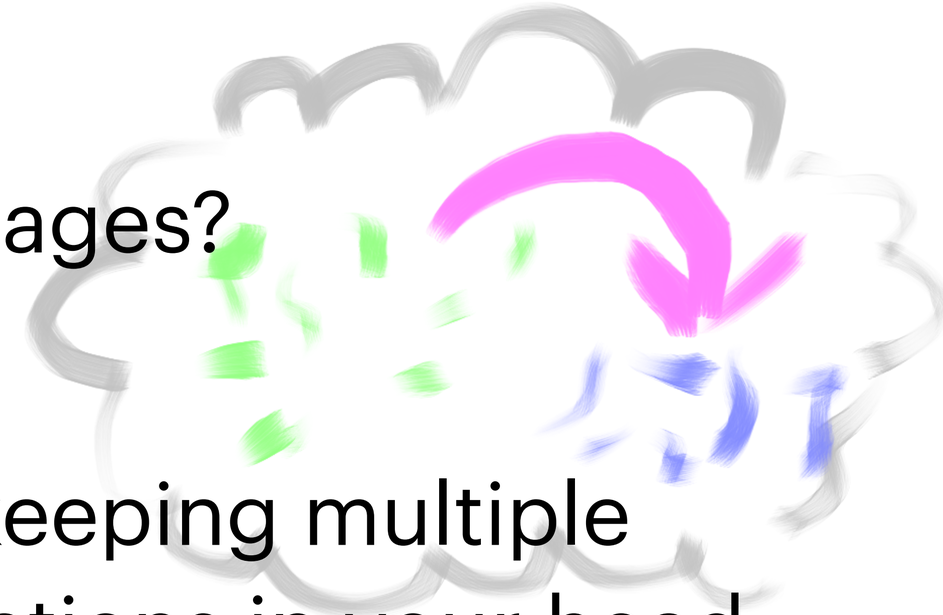
- I work on YJIT at Shopify, especially speed.yjit.org
- I wrote the book Rebuilding Rails
- I'm writing Rebuilding HTTP
- Ask me for bear stickers

I Love Questions

- This topic is complicated
- If you have a question, someone else does too
- Please ask, I'd love to answer it
- The worst that will happen is I'll say "I'm not answering that today" or "wait for a later slide"
- There's no Q&A at the end - ask before the end!

Why Do We Care?

What's interesting about three languages?



A lot of software dev work is about keeping multiple different-and-overlapping representations in your head. YJIT is a ***great*** example.

What's YJIT?

YJIT, Briefly

- A Just-In-Time compiler, built into recent CRuby
- Makes frequently-used code fast by compiling to native machine code
- Waits until runtime to see what code is used often, and how it gets used

How YJIT Works

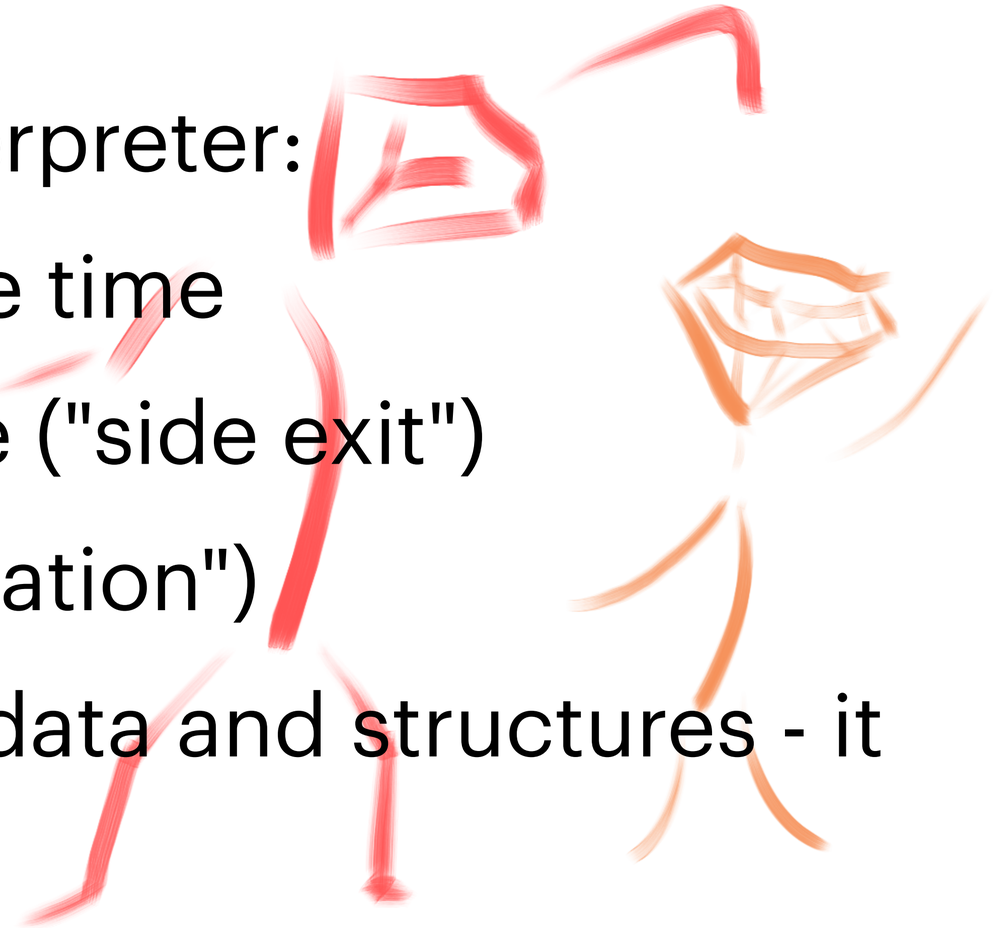
- At startup your code isn't compiled yet
- After 30 calls, YJIT compiles a specific method to machine code and starts running it
- May switch between run and compile several times for the same method
- Not concurrent - the process does one then the other, one at a time

YJIT and Ruby, Dancing

When YJIT swaps back to the interpreter:

- a case it can't handle at compile time
- a case it can't handle at runtime ("side exit")
- assumptions break ("de-optimization")

YJIT uses CRuby's same runtime data and structures - it has to, to swap constantly



A Little YJIT Theory

YJIT is Based on Lazy BBV

- Based on Maxime Chevalier-Boisvert's ECOOP 2015 whitepaper on Lazy Basic Block Versioning
- Divides methods into "basic blocks"; keeps context about what type various data has
- Not Ruby-specific; initial implementation was Javascript

Methods

- YJIT compiles an ISEQ (usually a method) made of Basic Blocks
- Each Basic Block is made of CRuby bytecodes.
- Curious how YJIT divides up a method into blocks? You can ask a dev build of YJIT to disassemble.

```
puts RubyVM::YJIT.disasm(method :whatever_method)
```

Multiplicity: Why Do We Care?

Software Dev: SO MANY Languages

- Your programming language is a language
- Your domain is a language
- APIs are tiny little languages too
- SQL, Javascript, CSS, Sass, TypeScript...
- A good developer has to juggle all this

Why Use YJIT as an Example?

- Obvious languages
- Concrete > Abstract
- Interesting mix

One Language: Ruby Bytecode

How Ruby Bytecodes Work

- Ruby turns your code into bytecode, in multiple steps
- Ruby bytecode is a stack machine
- A Ruby bytecode instruction will pop its arguments from the stack and then push the return value
- This is true whether or not you're using YJIT

A Bytecode Example

You can get a Ruby bytecode disassembly like this.

```
puts RubyVM::InstructionSequence.disassemble(method :whatever_method)
```

(This snippet is also at the resource url.)

A Bytecode Example

So let's do it:

```
puts RubyVM::InstructionSequence.disassemble(method :whatever_method)
```

```
def test(input)
  puts "Yes"
  if input
    puts "Well, maybe..."
  end
end
```

A Bytecode Example

Bytecode:

```
== disasm: #<ISeq:test@disassemble_test.rb:2 (2,0)-(7,3)> (catch: FALSE)
local table (size: 1, argc: 1 [opts: 0, rest: -1, post: 0, block: -1, kw: -1@-1, kwrest: -1])
[ 1] input@0<Arg>
0000 putself                                                (   3)[LiCa]
0001 putstring                            "Yes"
0003 opt_send_without_block               <calldata!mid:puts, argc:1, FCALL|ARGS_SIMPLE>
0005 pop
0006 getlocal_WC_0                        input@0           (   4)[Li]
0008 branchunless                         16
0010 putself                                                (   5)[Li]
0011 putstring                            "Well, maybe..."
0013 opt_send_without_block               <calldata!mid:puts, argc:1, FCALL|ARGS_SIMPLE>
0015 leave                                                  (   7)[Re]
0016 putnil                                                 (   5)
0017 leave                                                  (   7)[Re]
```

```
def test(input)
  puts "Yes"
  if input
    puts "Well, maybe..."
  end
end
```

YJIT Compiles Bytecodes

- Each bytecode turns into a chunk of machine code
- The chunks are mostly consecutive for a method

You're in luck right now. If you want to know more about bytecodes, Kevin Newton is writing a whole series on them:

https://kddnewton.com/2022/11/30/advent-of-yarv-part-0.html

ISEQ (method):

```
def my_method(both)
  call_one()
  call_two if both
end
```

Basic Blocks

```
putself
```

```
opt_send_without_block
```

```
pop
getlocal_WC_0
branchunless
```

```
putself
```

```
opt_send_without_block
```

```
leave
```

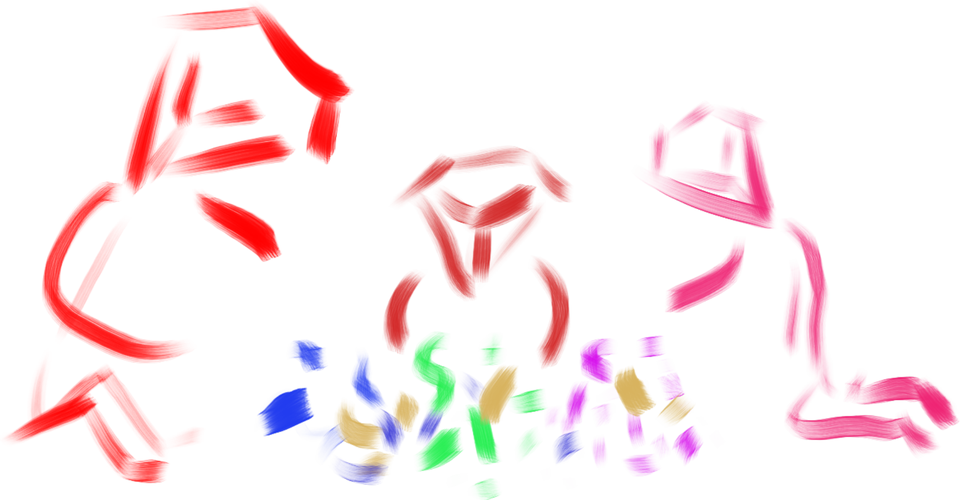
```
putnil
leave
```

x86 Assembly

```
# putself
...c119: mov rax, qword ptr [r13 + 0x
...c11d: mov qword ptr [rbx], rax
```

```
# opt_send_without_block
...c120: movabs rax, 0x7f5dc219c320
...c12a: cmp qword ptr [rbx], rax
...c12d: jne 0x56339311e0d3
# RUBY_VM_CHECK_INTS(ec)
...c133: mov eax, dword ptr [r12 + 0x2
...c138: test eax, eax
...c13a: jne 0x56339311e0b2
# stack overflow check
...c140: lea rax, [rbx + 0x98]
...c147: cmp r13, rax
...c14a: jbe 0x56339311e0b2
# store caller sp
...c150: lea rax, [rbx]
...c153: mov qword ptr [r13 + 8], rax
# save PC to CFP
...c157: movabs rax, 0x563392c0c2a8
...c161: mov qword ptr [r13], rax
```

One Language: YJIT's Context and IR



BBV and Context

- YJIT remembers types in a Context, such as whether a variable is nil, true, false, String, Array, etc.
- Different Basic Blocks have different Contexts
- In some source code, a variable can have different types on different successive calls and so each different type can have its own Context ("chaining")

Context Example

A Context remembers types. What does that look like?

```
# What type is this at compile-time? Might be unknown...
let val_type = ctx.get_opnd_type(insn_opnd);

# What type is the top object on Ruby's stack?
let arg_type = ctx.get_opnd_type(StackOpnd(0))

# We have proven this is a string with a runtime guard (or else we exited)
ctx.upgrade_opnd_type(insn_opnd, Type::CString);

# Take a data entry off Ruby's internal stack, and keep track of its type and location
let recv = ctx.stack_pop(1);
```

IR -> Assembly

YJIT generates an Intermediate Representation (IR) and then translates to x86_64 or AARCH64/ARM64 code.

```
// Conditionally move the length of the heap array
let flags_opnd = Opnd::mem((8 * SIZEOF_VALUE) as u8, array_reg, RUBY_OFFSET_RBASIC_FLAGS);
asm.test(flags_opnd, (RARRAY_EMBED_FLAG as u64).into());
let array_len_opnd = Opnd::mem(
    (8 * size_of::<std::os::raw::c_long>()) as u8,
    asm.load(array_opnd),
    RUBY_OFFSET_RARRAY_AS_HEAP_LEN,
);
let array_len_opnd = asm.csel_nz(emb_len_opnd, array_len_opnd);
```

Putting It Together

String Concatenation

```
fn jit_rb_str_concat(
    jit: &mut JITState,
    ctx: &mut Context,
    asm: &mut Assembler,
    ocb: &mut OutlinedCb,
    _ci: *const rb_callinfo,
    _cme: *const rb_callable_method_entry_t,
    _block: Option<IseqPtr>,
    _argc: i32,
    _known_recv_class: *const VALUE,
) -> bool {
```

String Concatenation

```
// The << operator can accept integer codepoints for characters  as the
// argument. We only specially optimise string arguments.
// If the peeked-at compile time argument is something other than a string,
// assume it won't be a string later either.
let comptime_arg = jit_peek_at_stack(jit, ctx, 0);
if ! unsafe { RB_TYPE_P(comptime_arg, RUBY_T_STRING) } {
    return false;
}
```

String Concatenation

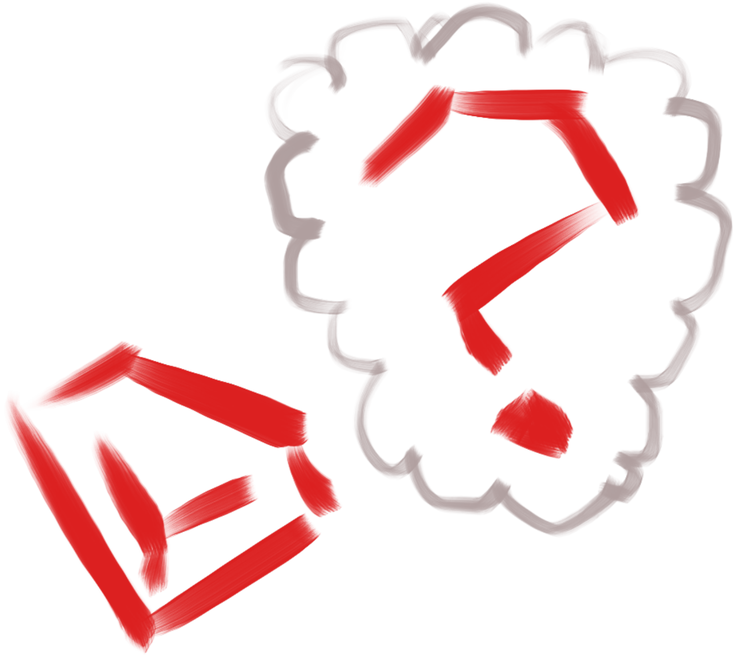
```
// Generate a side exit
let side_exit = get_side_exit(jit, ocb, ctx);


let arg_type = ctx.get_opnd_type(StackOpnd(0));


// Pop arguments off Ruby's internal stack
let concat_arg = ctx.stack_pop(1);
let recv = ctx.stack_pop(1);
```

String Concatenation

```
// If we're not compile-time certain that this will always be a string,
// guard at runtime
if arg_type != Type::CString && arg_type != Type::TString {
    let arg_opnd = asm.load(concat_arg);
    if !arg_type.is_heap() {
        asm.comment("guard arg not immediate");
        asm.test(arg_opnd, (RUBY_IMMEDIATE_MASK as u64).into());
        asm.jnz(side_exit.as_side_exit());
        asm.cmp(arg_opnd, Qfalse.into());
        asm.je(side_exit.as_side_exit());
    }
    guard_object_is_string(asm, arg_opnd, side_exit);
}
```

String Concatenation

```
asm.comment("<< on strings");
let stack_ret = ctx.stack_push(Type::CString);
let ret_opnd = asm.ccall(rb_str_buf_append as *const u8,
                         vec![recv, concat_arg]);
asm.mov(stack_ret, ret_opnd);
```



Once Again - Why?

The Power of Multiple

This kind of code gets its power from carefully thinking through the different languages - runtime vs compile-time, Ruby vs IR vs native.

Summary

How Far Along is YJIT?

- Pretty fast, pretty robust
- Latest speed results for x86 and ARM at speed.yjit.org
- Best for long-running code, not (e.g.) gem or bundle CLI
- "Limited production-ready"
- Extra-stability release with ARM support in Ruby 3.2.0
- Ruby 3.2.0 will be released 25th Dec, 2022

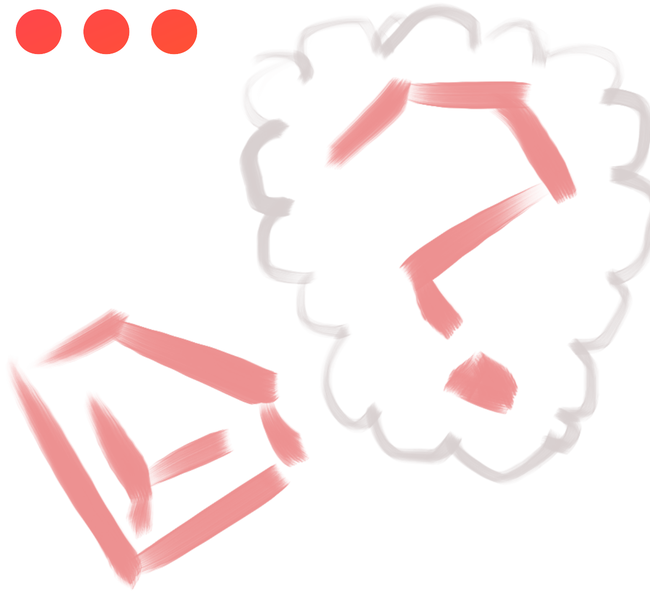
References

- YJIT is part of the CRuby source code - see https://github.com/ruby/ruby
- https://arxiv.org/abs/1411.0352 - Lazy BBV whitepaper

You can find various YJIT resources at https://codefol.io/speaking/rubyconfth2022

Resource URL: https://codefol.io/speaking/rubyconfth2022

No Q&A, But...



Mastodon: ruby.social/codefolio
https://codefol.io