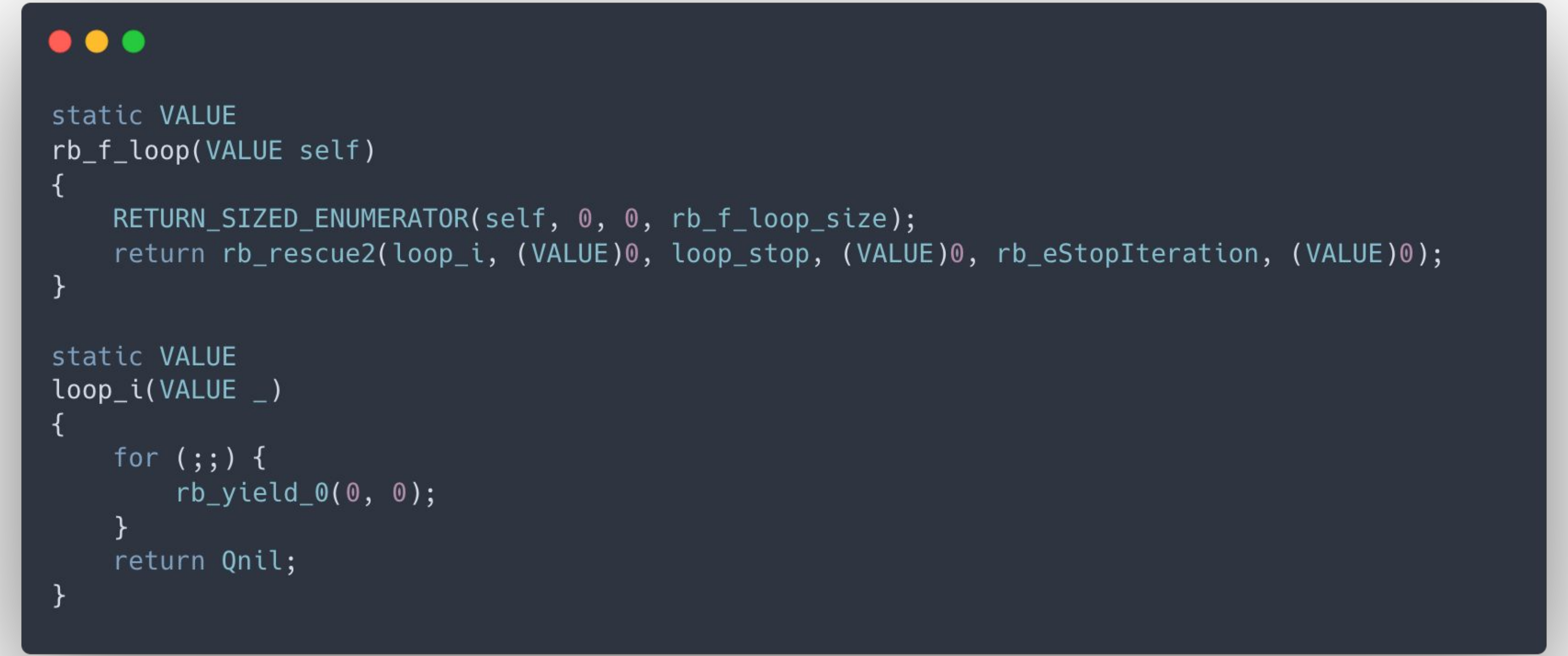
Ruby’s Core Gem

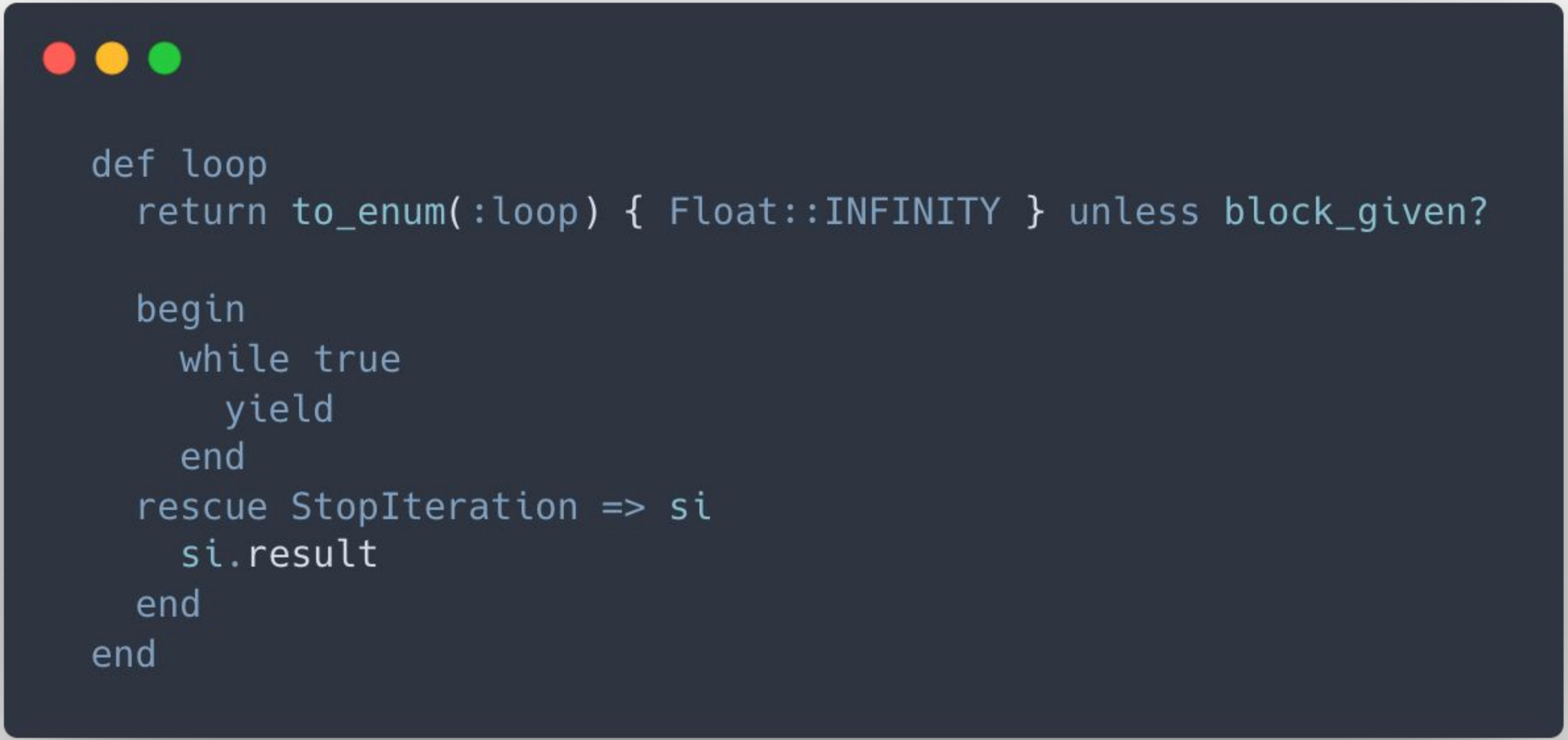
Chris Seaton

RubyConf 2022

The big idea up front



```c
static VALUE
rb_f_loop(VALUE self)
{
    RETURN_SIZED_ENUMERATOR(self, 0, 0, rb_f_loop_size);
    return rb_rescue2(loop_i, (VALUE)0, loop_stop, (VALUE)0, rb_eStopIteration, (VALUE)0);
}

static VALUE
loop_i(VALUE _)
{
    for (;;) {
        rb_yield_0(0, 0);
    }
    return Qnil;
}
```



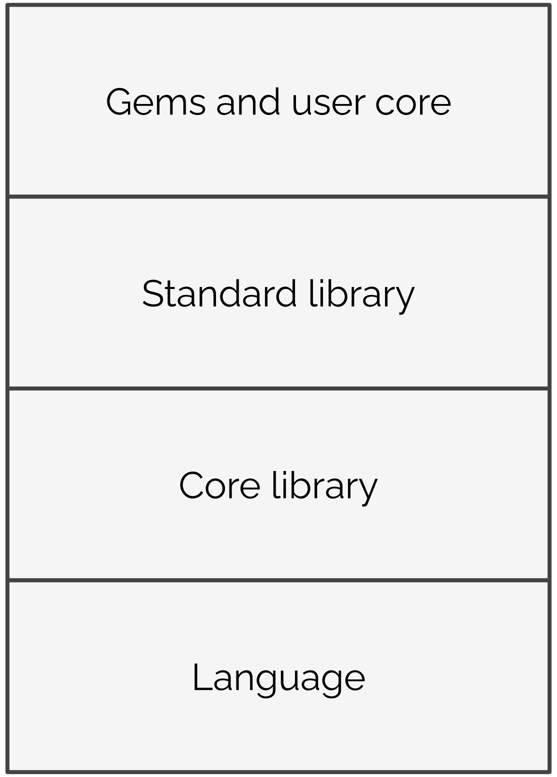
```ruby
def loop
  return to_enum(:loop) { Float::INFINITY } unless block_given?

  begin
    while true
      yield
    end
  rescue StopIteration => si
    si.result
  end
end
```

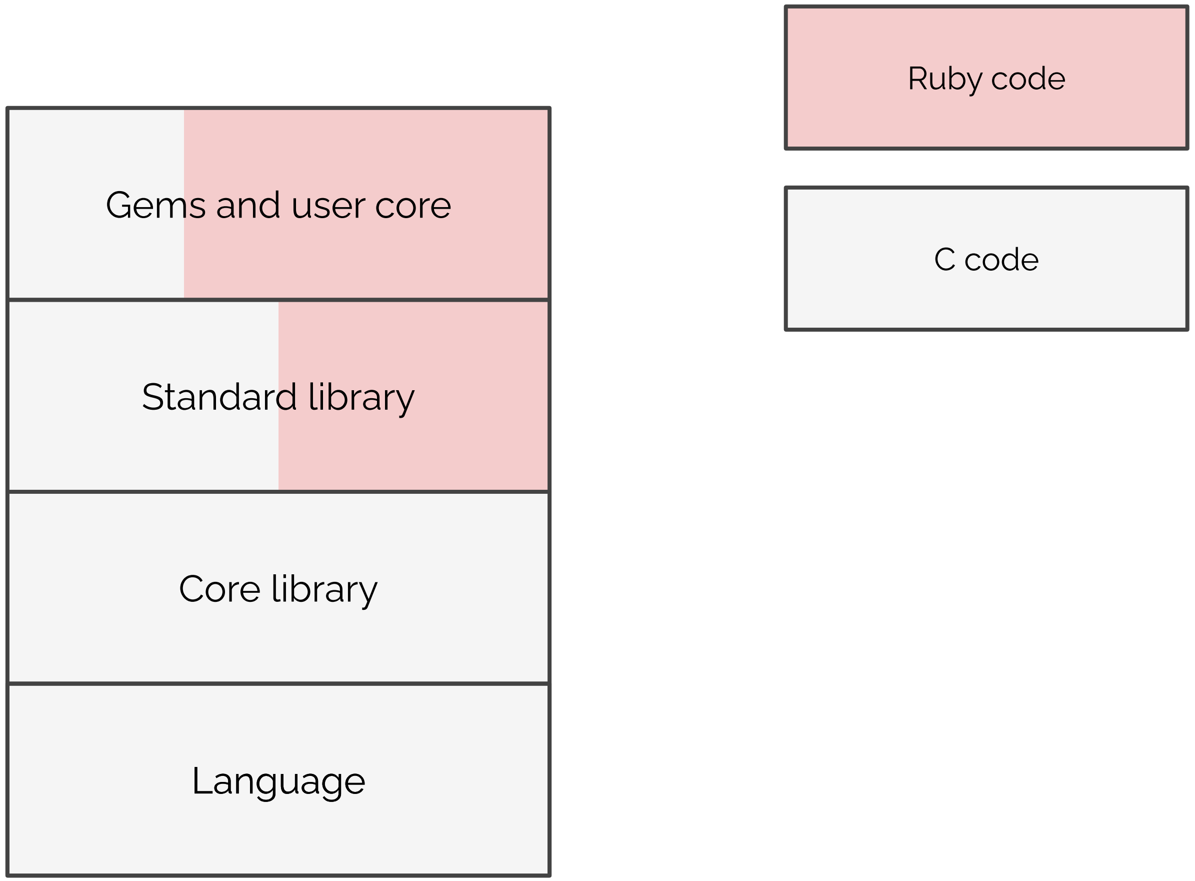
Context on me and my work

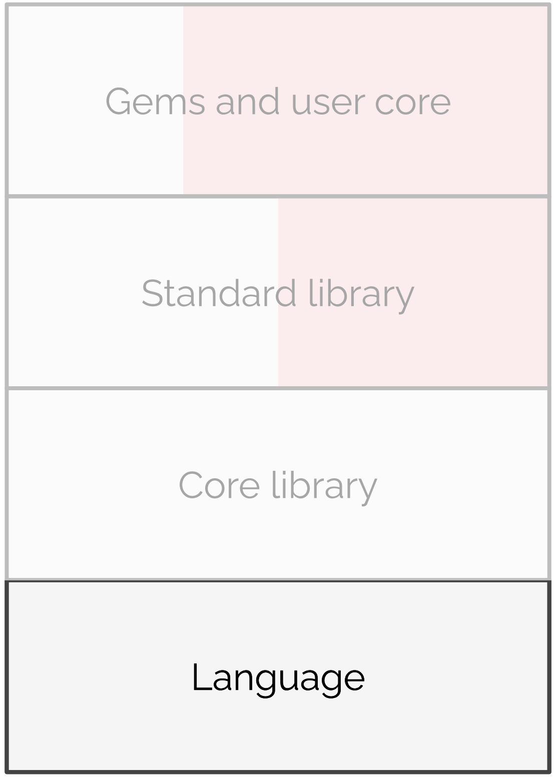
- From Cheshire in the UK (like the cat)
- PhD in compiling Ruby
- Founded TruffleRuby
- Formerly Oracle Labs, now Shopify which is a supportive place with great people
- Interested in optimising idiomatic Ruby code
- Lead a British cavalry squadron in my spare time

  (interested in meeting other Ruby veterans)

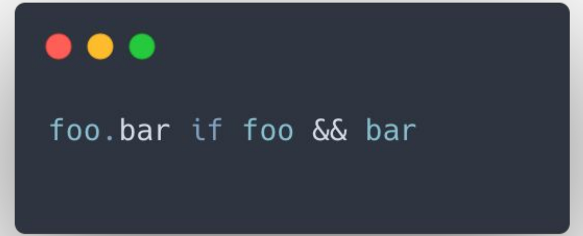
Ruby’s tower of libraries


Gems and user core
Standard library
Core library
Language

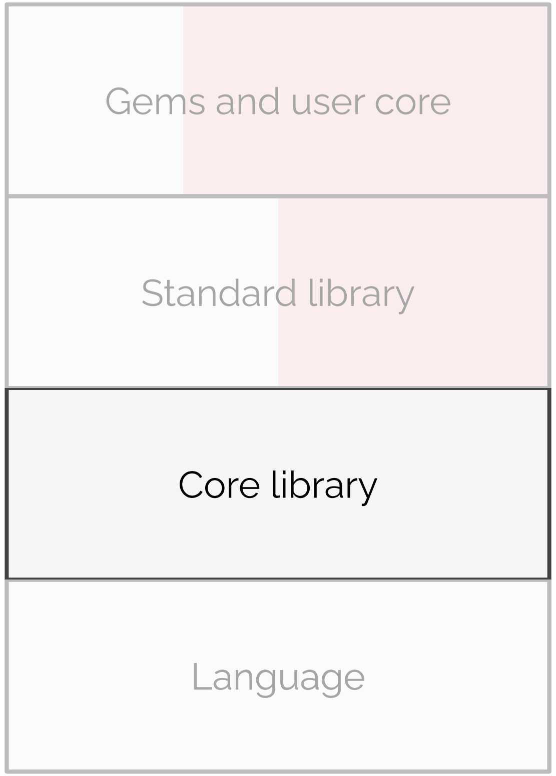

Ruby code
C code
Gems and user core
Standard library
Core library
Language



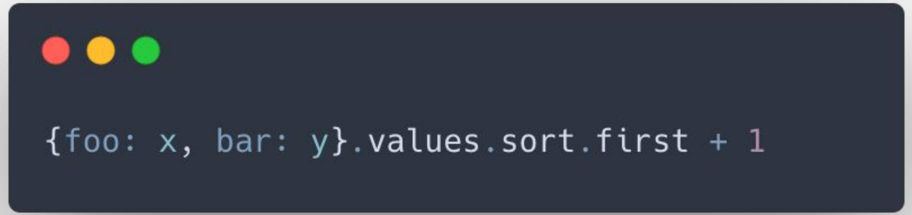
- A smaller number of things provided by the Ruby language itself
- Classes, modules, methods
- Method calls
- if, while, case, and, &&
- Very little else is provided by the language



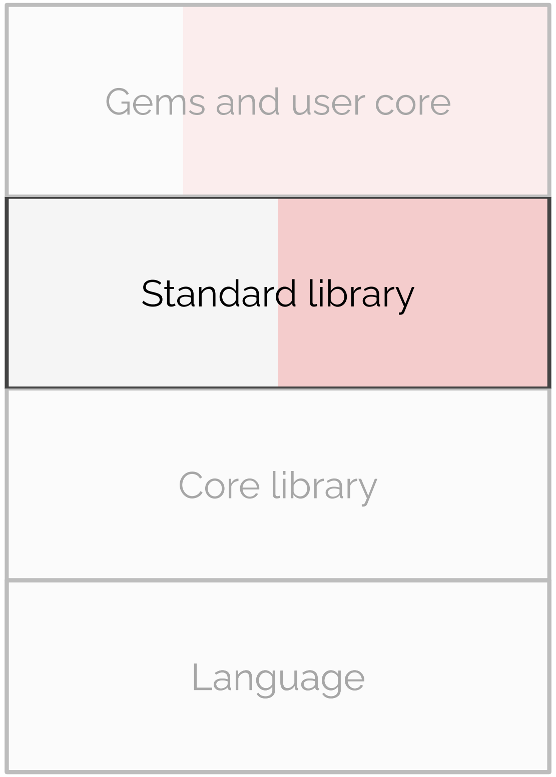
```
foo.bar if foo && bar
```



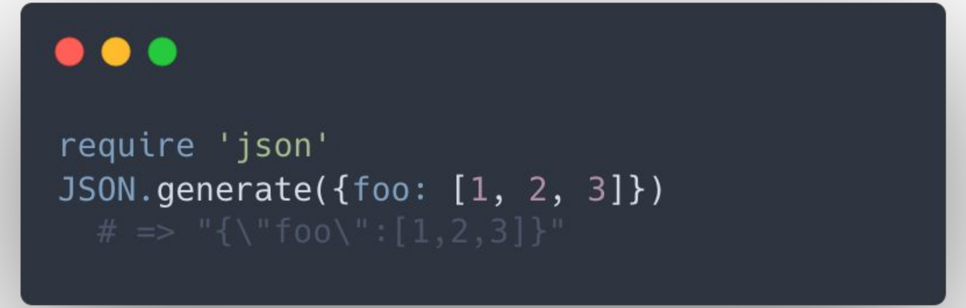
- Array, Hash
- Numbers, strings
- Basic control structures - #loop, #each
- Automatically available - no require
- Implemented as a 'C extension'
- About 2250 methods



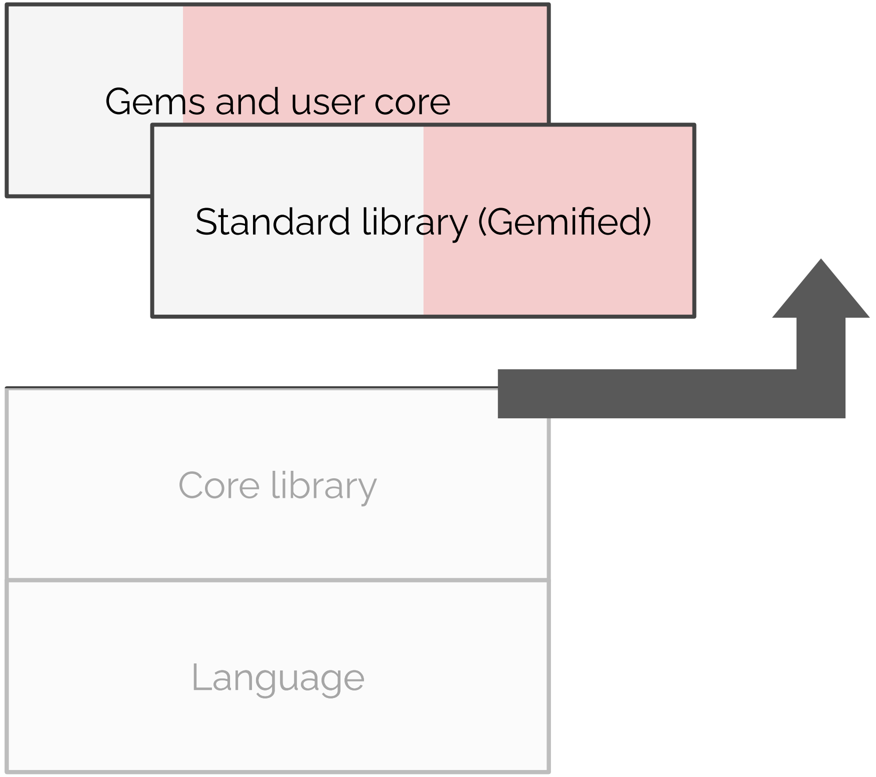
```
{foo: x, bar: y}.values.sort.first + 1
```

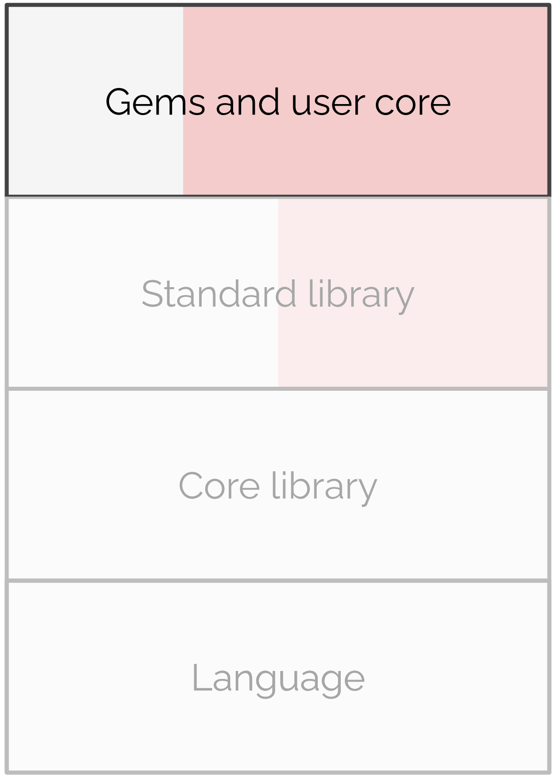


- Needs to be required (some exceptions)
- But available without installing anything
- We won't worry about it much in this talk
- Being 'lifted' in the tower

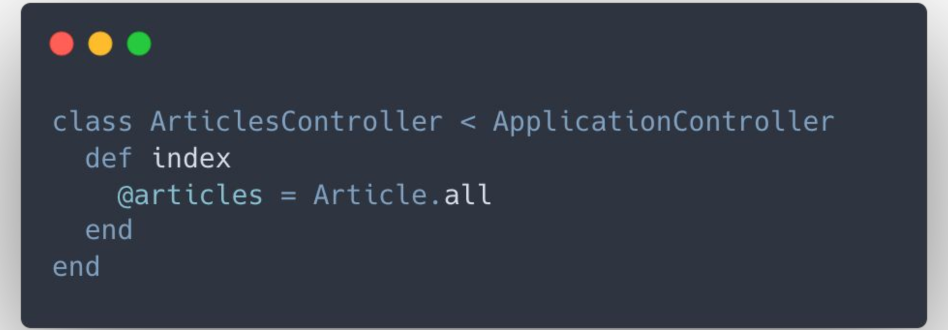


```ruby
require 'json'
JSON.generate({foo: [1, 2, 3]})
  # => "{\"foo\":[1,2,3]}"
```


Gems and user core
Standard library (Gemified)
Core library
Language



- Ruby code loaded at runtime from outside the interpreter
- Can be from a gem or code in your repo
- Makes a difference to us as programmers, but doesn't really make any difference to the Ruby VM


```
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end
end
```

Good things about core as it is

- Always available
  - can’t go wrong
  - can be used to build bigger things like RubyGems
- Available instantly
- Can use VM internals to do things you can’t do in Ruby
- Compilers can be taught about it
  - will explain later!

Bad things about core as it is

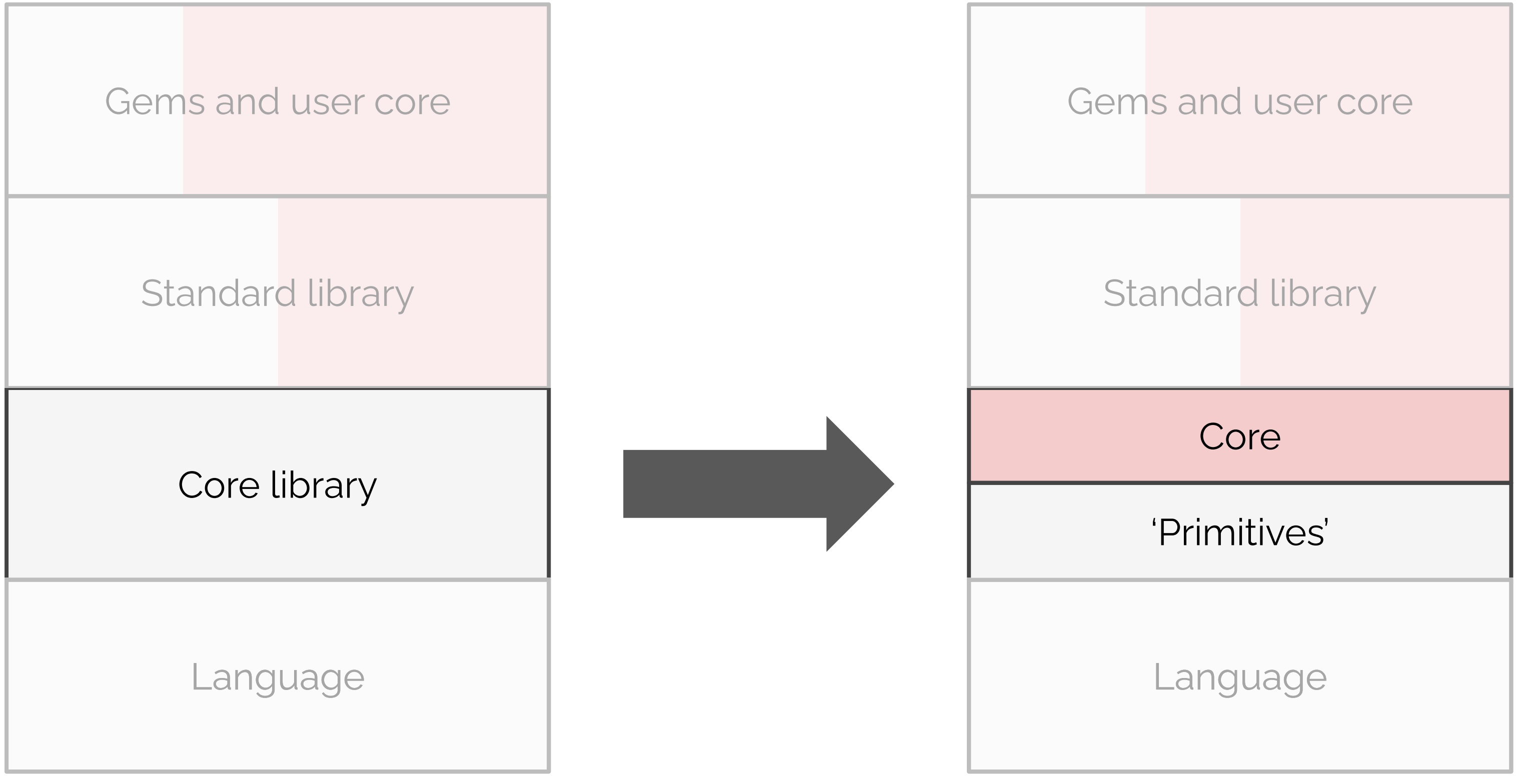
- It’s far too big
- No Ruby code that you can read
- No Ruby code that you can debug
- No Ruby code to profile, coverage, etc
- All ‘C extension’ code…

Bad things about C extension code

- Did you know C code can be worse for performance than Ruby code?

  will explain later

Get the best of both worlds?


Gems and user core
Standard library
Core library
Language
Gems and user core
Standard library
Core
'Primitives'
Language

Best of both worlds

- Bulk of code in Ruby

  can be read, understood, debugged

  can be optimised by the Ruby VM
- Small set of underlying ‘primitives’ as ‘C extensions’

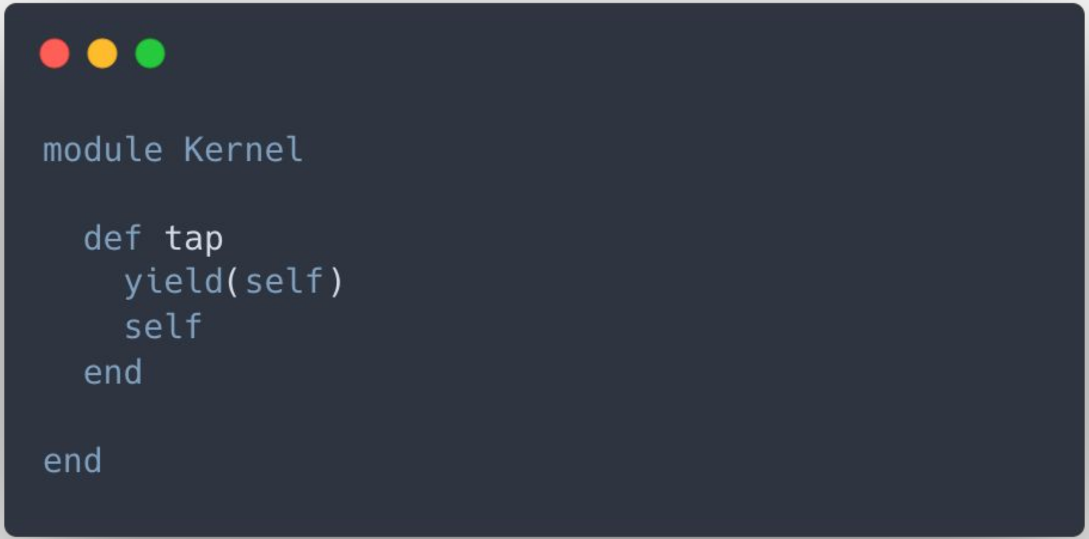
  can teach the compiler about them as there’s not many

  available instantly

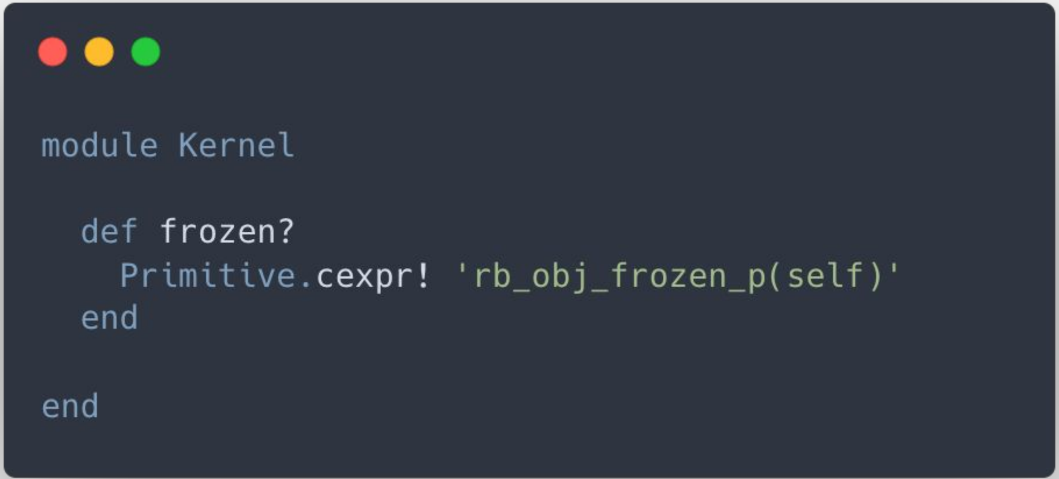
Ruby implementations already do this

- MRI (CRuby) a tiny bit
- JRuby a bit more
- TruffleRuby a bit more still
- (We’ll talk about Rubinius later)

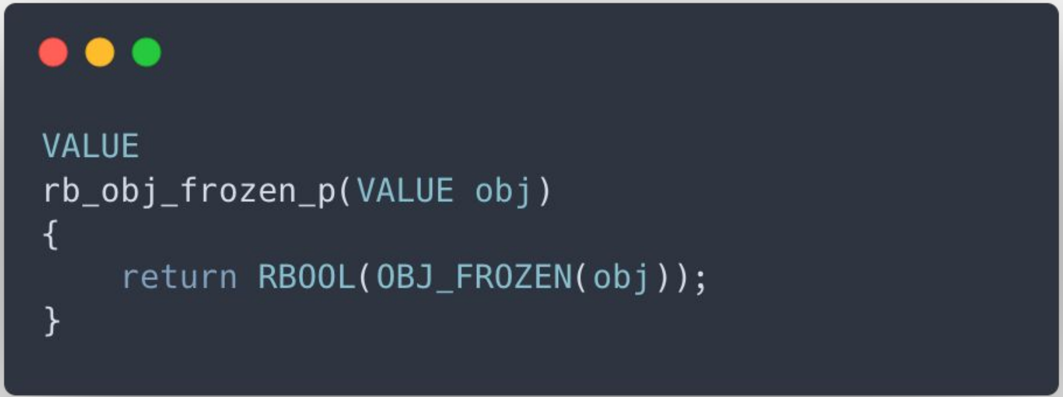
How MRI (CRuby) does it



```ruby
module Kernel

  def tap
    yield(self)
    self
  end

end
```



```
module Kernel

  def frozen?
    Primitive.cexpr! 'rb_obj_frozen_p(self)'
  end

end
```



```
VALUE
rb_obj_frozen_p(VALUE obj)
{
    return RBOOL(OBJ_FROZEN(obj));
}
```

2194 core methods in C

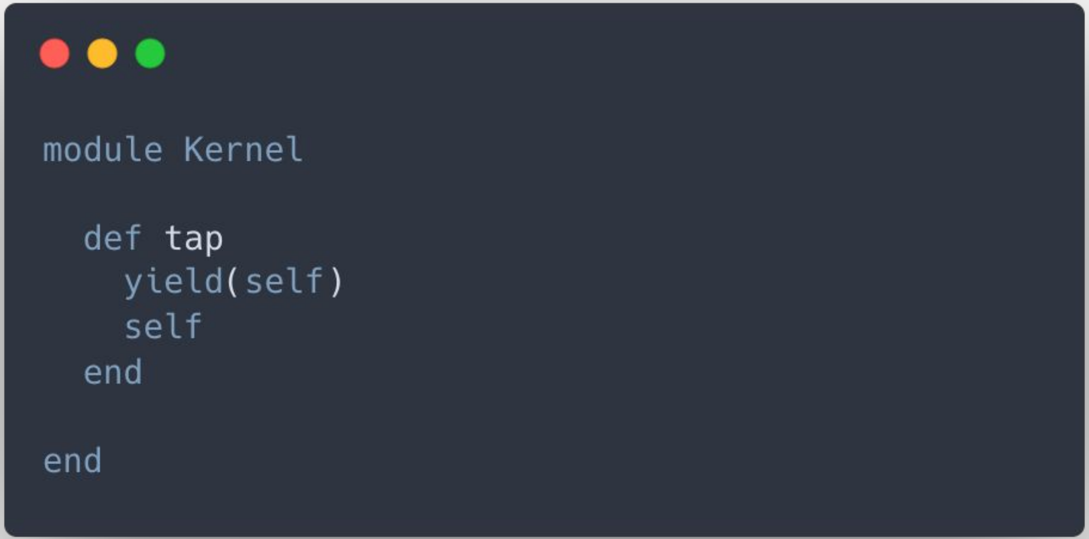
64 core primitives in C

31 instances of inline C

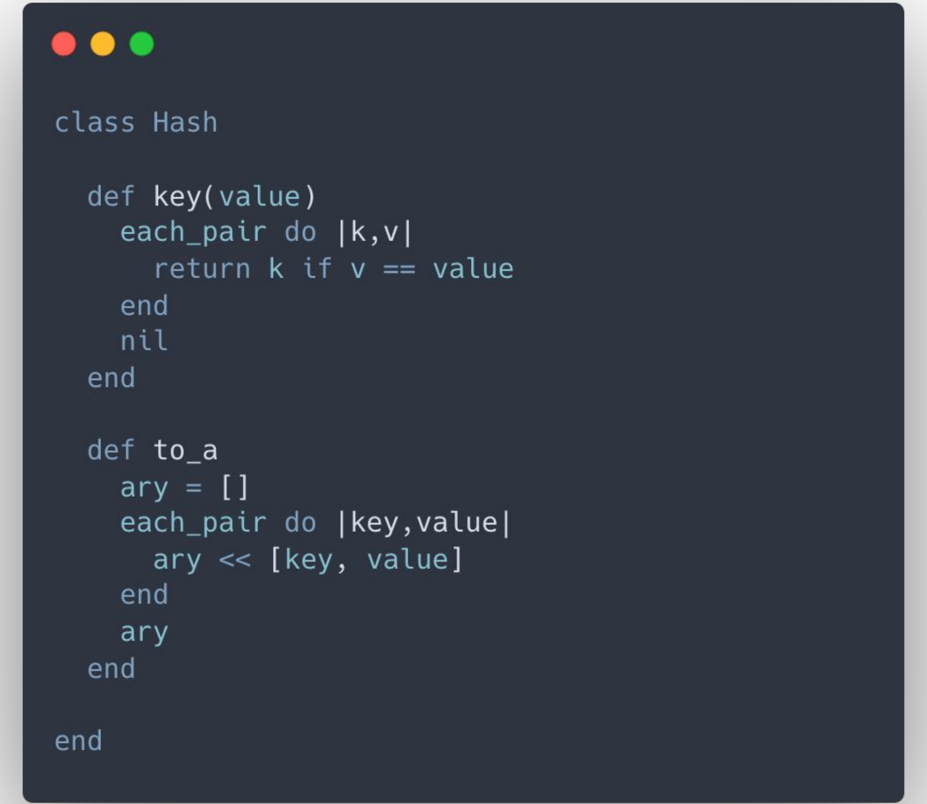
7 special ‘optimised’ core methods

101 core methods in Ruby

How TruffleRuby does it



```ruby
module Kernel

  def tap
    yield(self)
    self
  end

end
```



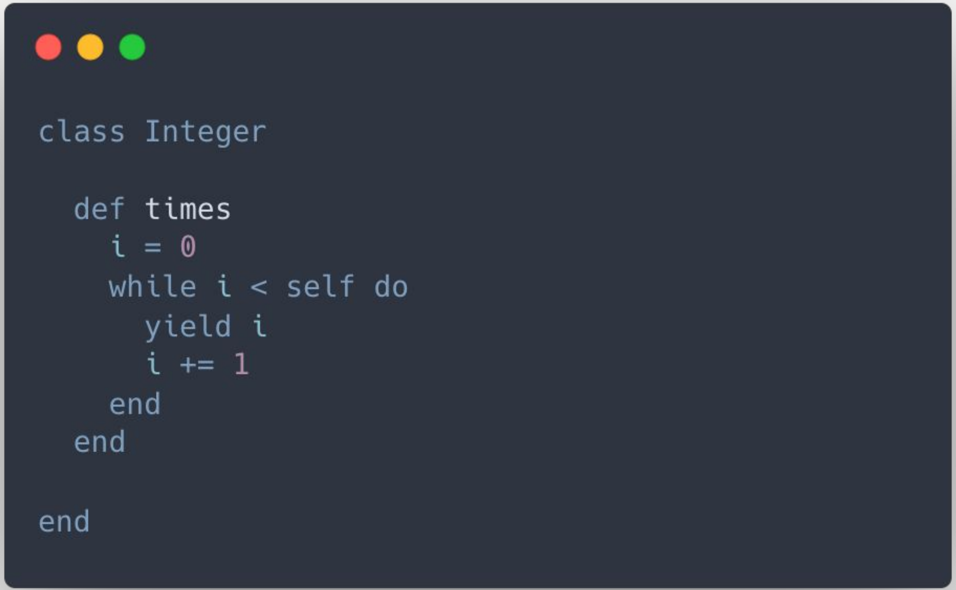
```ruby
class Hash

  def key(value)
    each_pair do |k,v|
      return k if v == value
    end
    nil
  end

  def to_a
    ary = []
    each_pair do |key,value|
      ary << [key, value]
    end
    ary
  end

end
```

611 core methods in Java

353 core primitives in Java

2386 core methods in Ruby

JRuby does it too



```ruby
class Integer

  def times
    i = 0
    while i < self do
      yield i
      i += 1
    end
  end

end
```

Advantages of a Ruby core

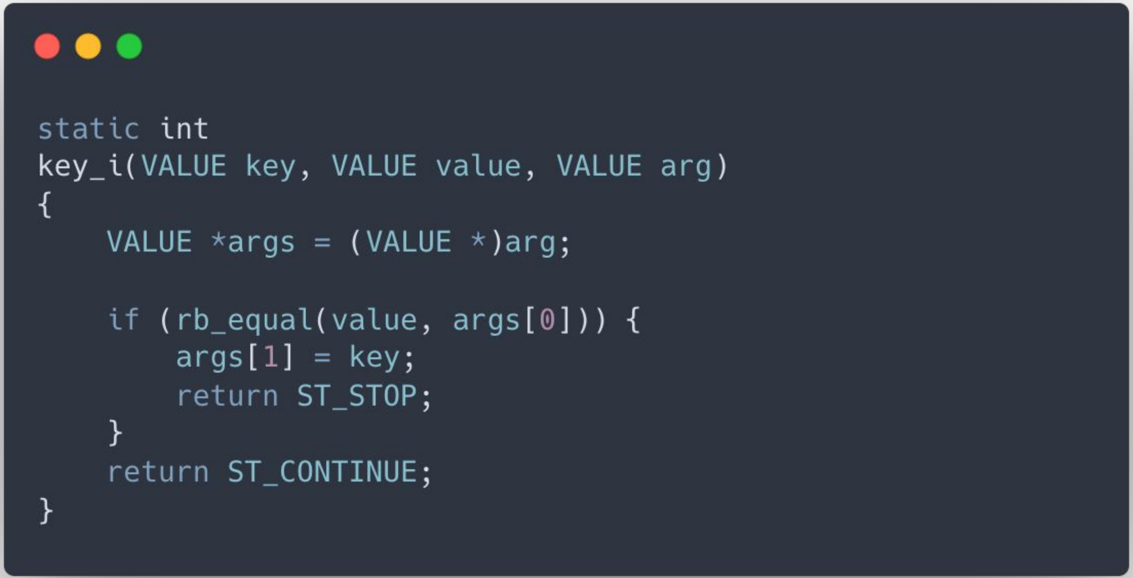
Advantage - understandability

- You can browse the Ruby code to understand it
- Answer your own questions about what core methods really do
- Use your normal debugger, coverage, profiler tools
- No longer a ‘black box’

Advantage - shared code

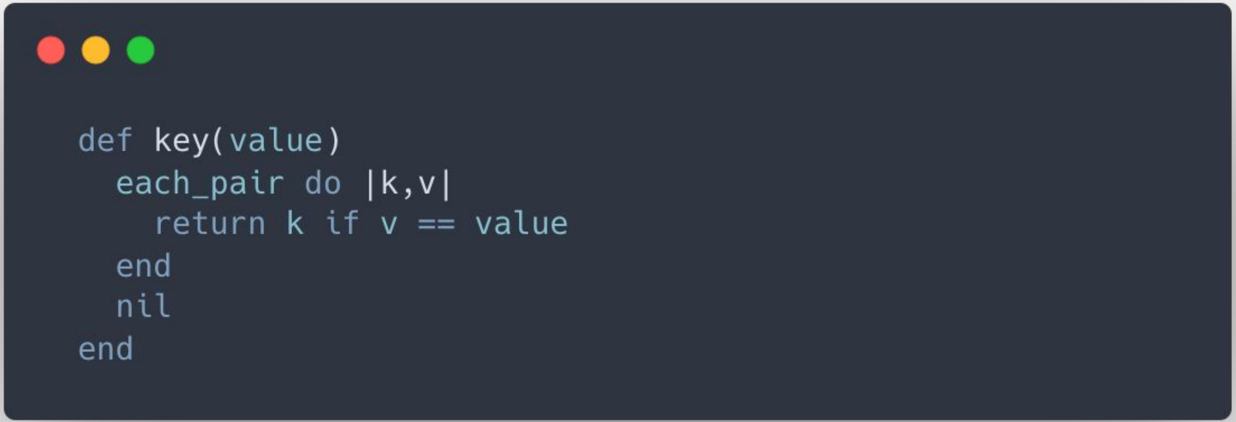
- MRI, TruffleRuby, JRuby, Artichoke, whatever comes next, could all share the same core library
- Each would implement a smaller set of primitives their own way
- VM people can focus on making the primitives work well
- Other people can focus on making the core library work well

Advantage - optimisation



```
static int
key_i(VALUE key, VALUE value, VALUE arg)
{
    VALUE *args = (VALUE *)arg;

    if (rb_equal(value, args[0])) {
        args[1] = key;
        return ST_STOP;
    }
    return ST_CONTINUE;
}
```

Advantage - optimisation



```
def key(value)
  each_pair do |k,v|
    return k if v == value
  end
  nil
end
```

Disadvantages of a Ruby core

Disadvantage - parse time

- Have to parse all this Ruby code at startup
- We said it’s better for optimisation, but that’s only when the optimisations have had time to run!
- People already have to do things like `--disable=gems` for command-line tools to reduce startup time, this would make it much worse

Disadvantage - parse time - mitigate it

- MRI embeds the YARV bytecode, not parse it
- TruffleRuby embeds the objects into the executable, not parse it
- TruffleRuby can start up more quickly than MRI due to this

Disadvantage - memory

- Ruby code is quite a bit bigger than compiled C code
- The profiles, inlining, and splitting, and things that make Ruby code faster also take up more memory
- The optimisation that we say we get takes up even more memory still

Disadvantage - memory - mitigate it

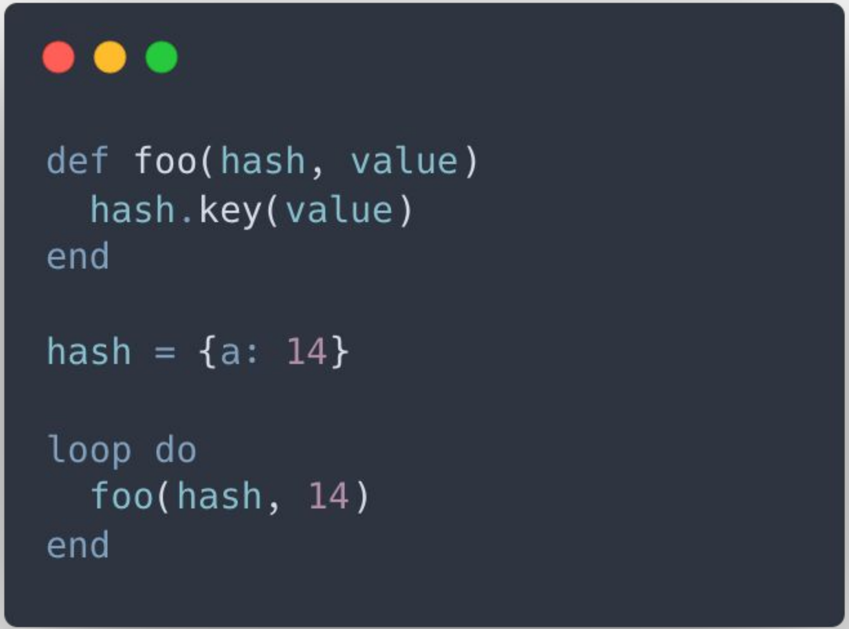
- Don’t actually really have any great ways to mitigate it
- Does anyone else? Open to ideas
- At least it’s per-process, not per-user

One alternative

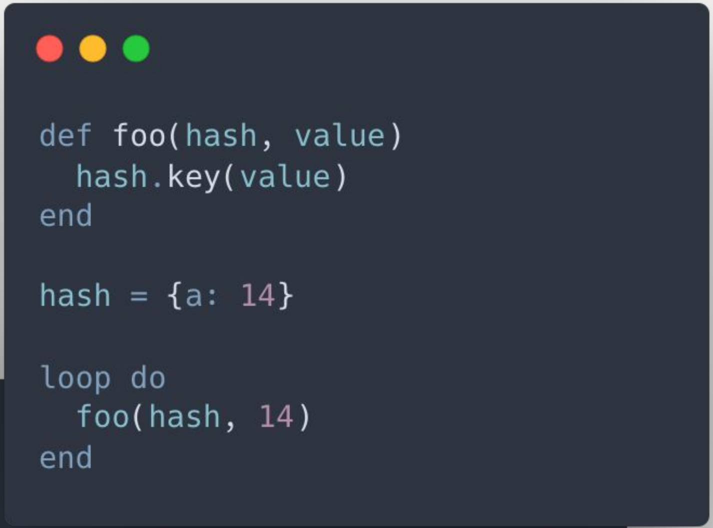
Sulong - interpret C code with profiling

- Sulong is a C interpreter and JIT
- TruffleRuby uses it to run C extensions
- Requires some truly heroic work to restore the performance of native C code - so slow to warm up

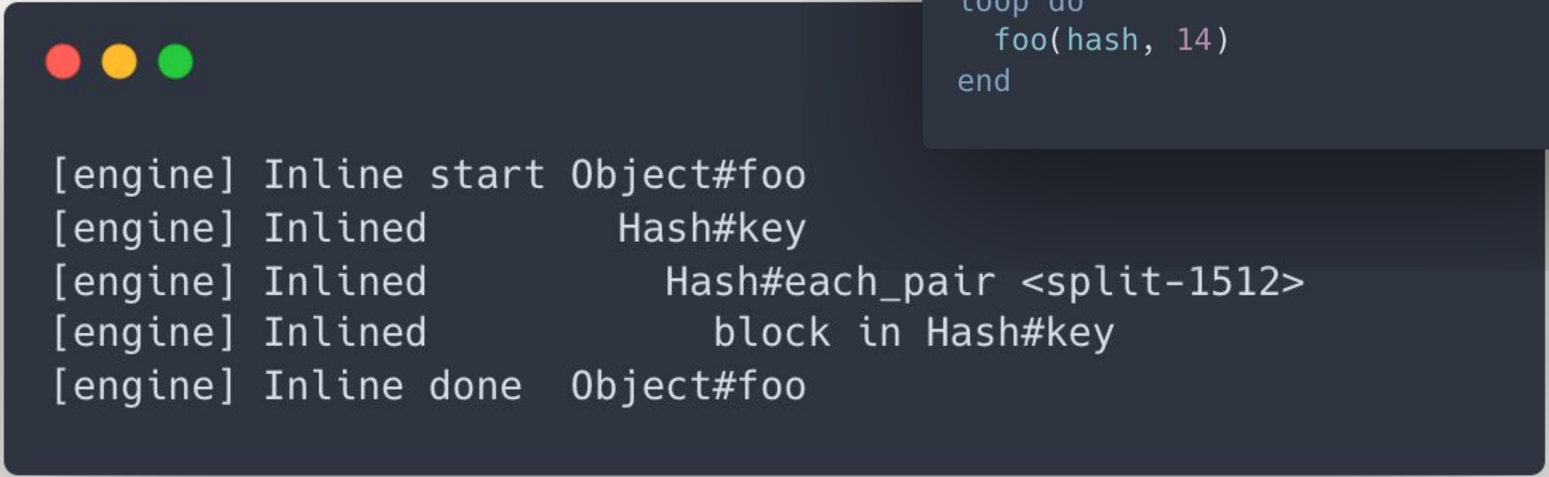
A practical demonstration



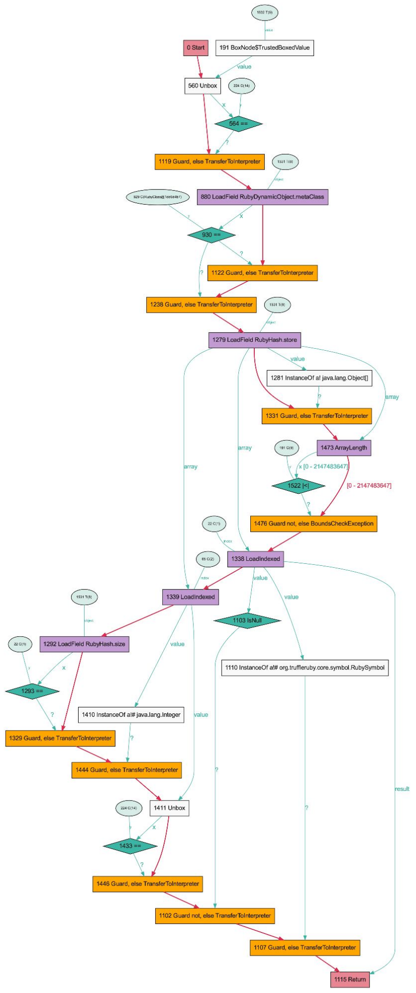
```
def foo(hash, value)
  hash.key(value)
end

hash = {a: 14}

loop do
  foo(hash, 14)
end
```

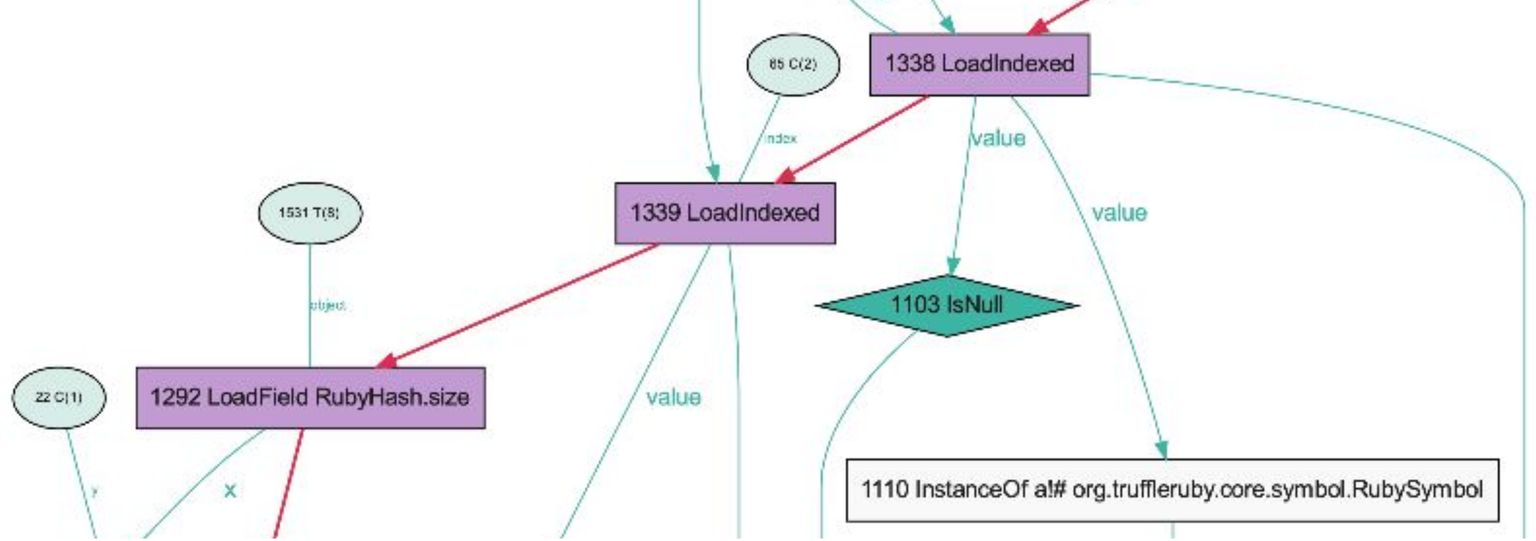


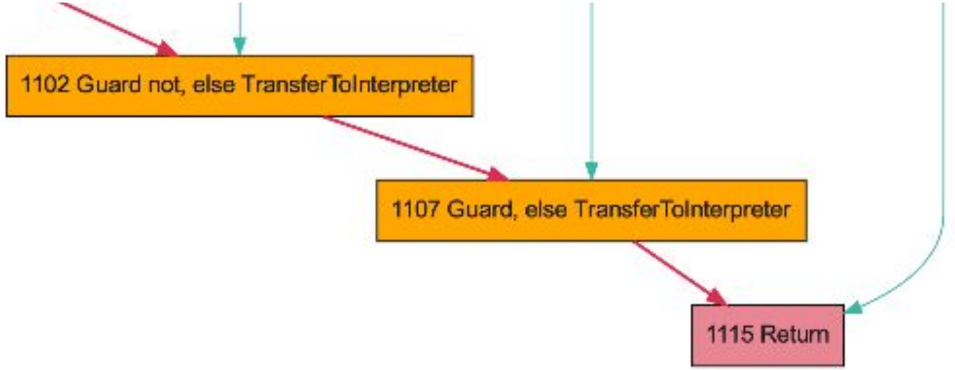
```
def foo(hash, value)
  hash.key(value)
end

hash = {a: 14}

loop do
  foo(hash, 14)
end
```



```
[engine] Inline start Object#foo
[engine] Inlined        Hash#key
[engine] Inlined          Hash#each_pair <split-1512>
[engine] Inlined            block in Hash#key
[engine] Inline done  Object#foo
```


1532 T(9)
value
0 Start
191 BoxNode$TrustedBoxedValue
value
560 Unbox
224 C(14)
x
y
564 ==
?
1119 Guard, else TransferToInterpreter
1531 T(8)
object
929 C(RubyClass@1dd6d4b7)
880 LoadField RubyDynamicObject.metaClass
y
x
930 ==
?
?
1122 Guard, else TransferToInterpreter
1238 Guard, else TransferToInterpreter
1531 T(8)
object
1279 LoadField RubyHash.store
value
1281 InstanceOf a! java.lang.Object[]
?
array
1331 Guard, else TransferToInterpreter
1473 ArrayLength
181 C(0)
y
x [0 - 2147483647]
[0 - 2147483647]
array
array
1522 |<|
?
1476 Guard not, else BoundsCheckException
22 C(1)
index
1338 LoadIndexed
85 C(2)
index
value
value
1339 LoadIndexed
1531 T(8)
object
1103 IsNull
1292 LoadField RubyHash.size
22 C(1)
y
x
value
1110 InstanceOf a!# org.truffleruby.core.symbol.RubySymbol
1293 ==
?
1410 InstanceOf a!# java.lang.Integer
value
1329 Guard, else TransferToInterpreter
?
?
?
result
1444 Guard, else TransferToInterpreter
1411 Unbox
224 C(14)
y
x
1433 ==
?
1446 Guard, else TransferToInterpreter
1102 Guard not, else TransferToInterpreter
1107 Guard, else TransferToInterpreter
1115 Return


1338 LoadIndexed
85 C(2)
index
value
value
1339 LoadIndexed
1531 T(8)
object
1103 IsNull
value
22 C(1)
1292 LoadField RubyHash.size
y
x
1110 InstanceOf a!# org.truffleruby.core.symbol.RubySymbol


1102 Guard not, else TransferToInterpreter
1107 Guard, else TransferToInterpreter
1115 Return

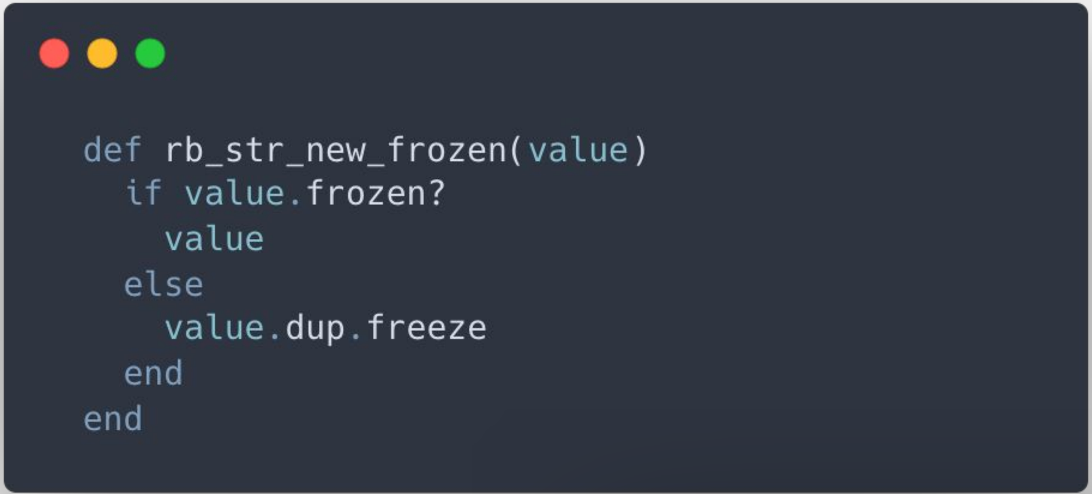
A potential way forward

- Move the majority of core into Ruby
- Leave a smaller, better defined set of primitives
- Use TruffleRuby’s core as a starting point
- Teach our compilers and static analysis tools more about these primitives
- A smaller, more manageable, more analysable, Ruby
- But works exactly the same as now for application developers
- Does it literally need to be a gem?

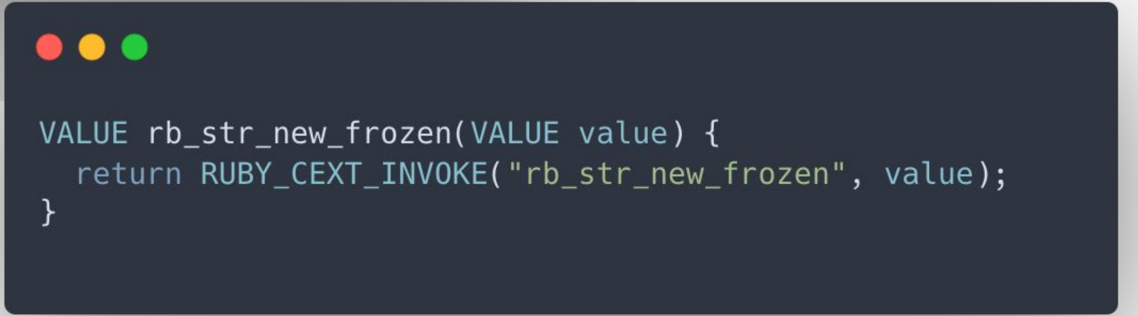
Attribution to Rubinius

- A lot of TruffleRuby’s core library originated from Rubinius, but has been maintained by us for a few years now
- Excellent work by Evan Phoenix, Brian Shirai, and others

An even more radical idea

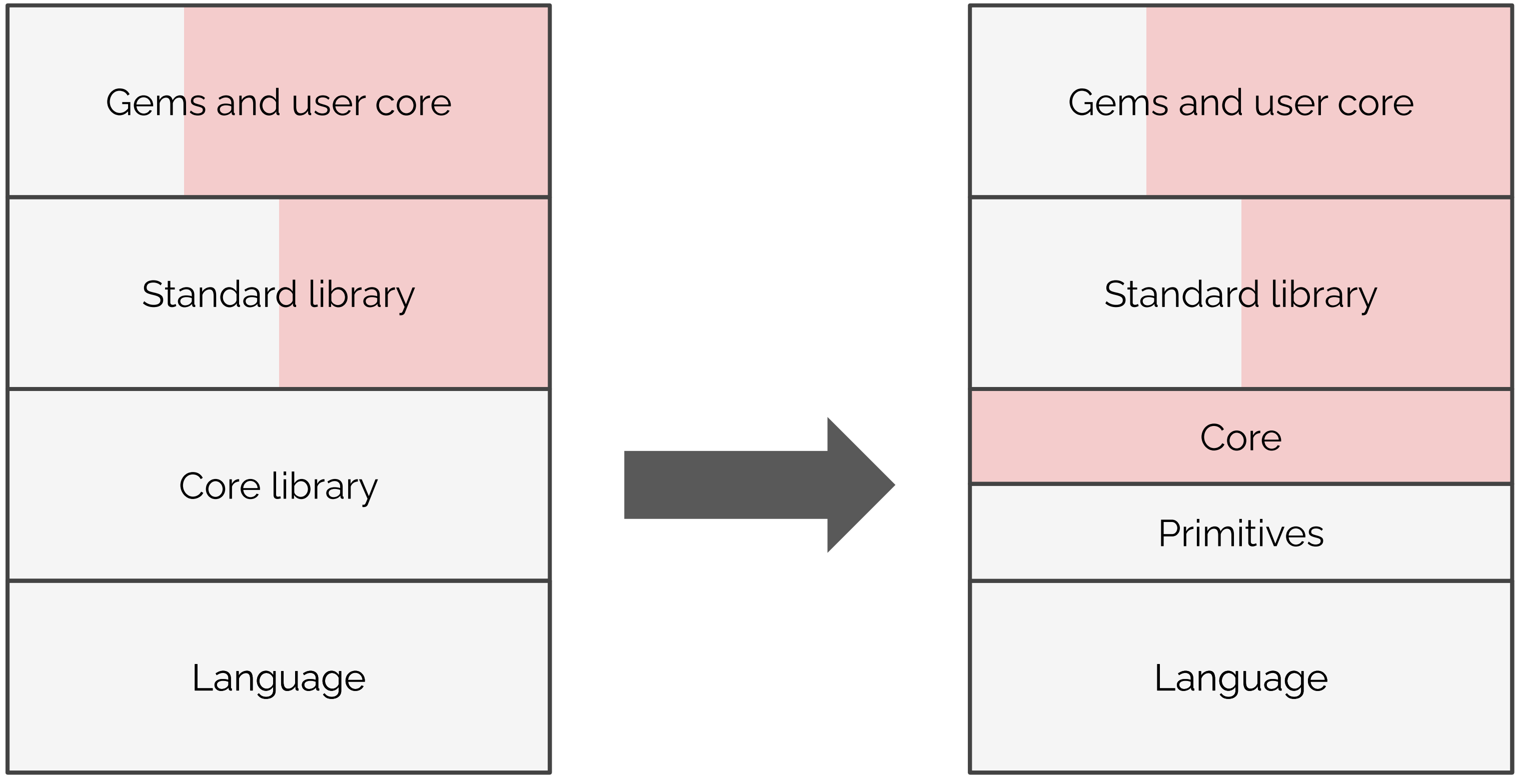


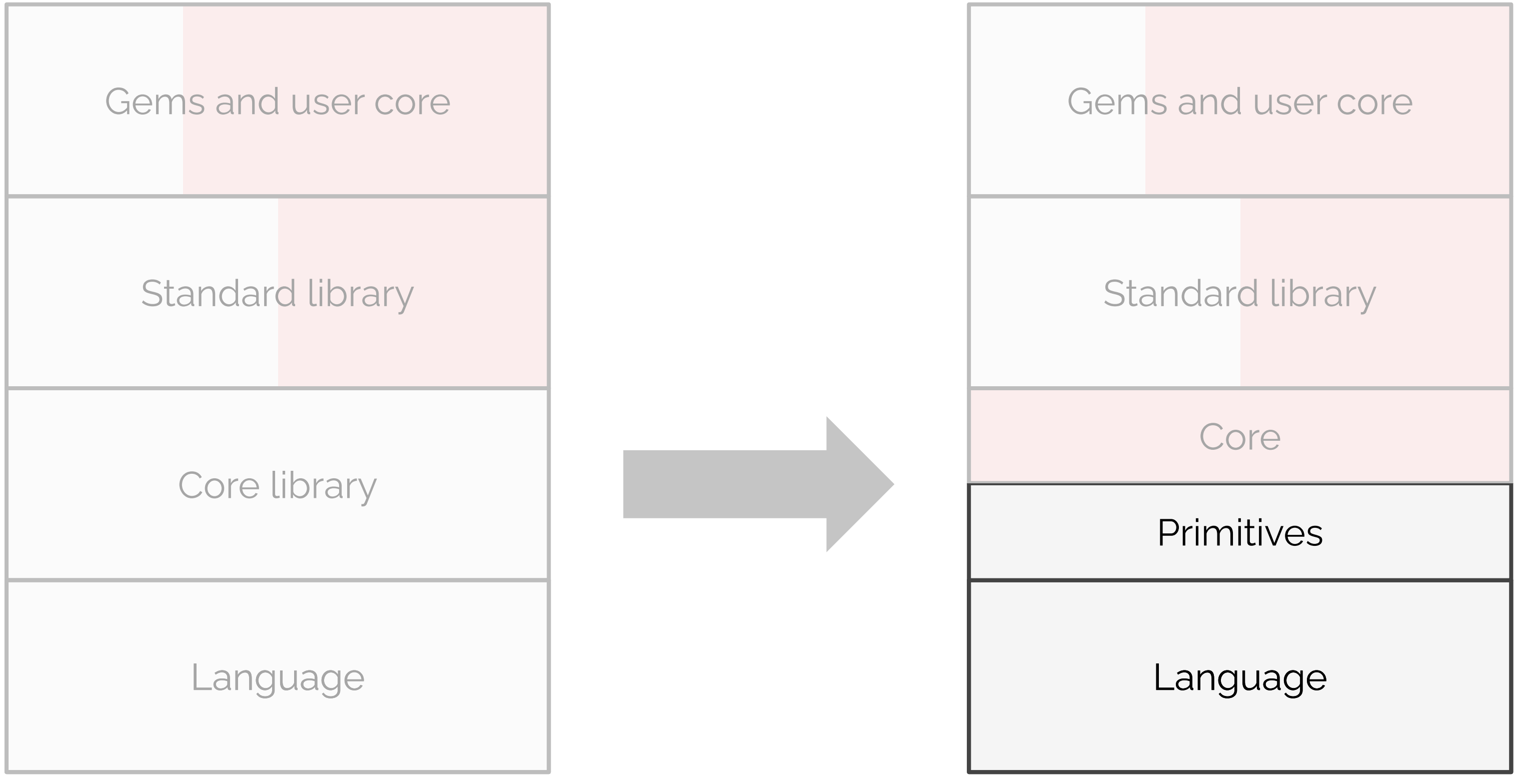
```ruby
def rb_str_new_frozen(value)
  if value.frozen?
    value
  else
    value.dup.freeze
  end
end
```



```c
VALUE rb_str_new_frozen(VALUE value) {
  return RUBY_CEXT_INVOKE("rb_str_new_frozen", value);
}
```

Conclusions


Gems and user core
Standard library
Core library
Language
Gems and user core
Standard library
Core
Primitives
Language


Gems and user core
Standard library
Core library
Language
Gems and user core
Standard library
Core
Primitives
Language

Is it a good idea?

👍

- Understandability
- Shareability
- Debugability
- Optimisability
- Analysability

👎

- Impact on startup time
  (we have a solution)
- Impact on memory usage
  (not sure we have a solution

Surely worth trying

- We have a core in TruffleRuby we could start trying with
- Going to become more relevant as MRI gets more sophisticated
- The future of Ruby

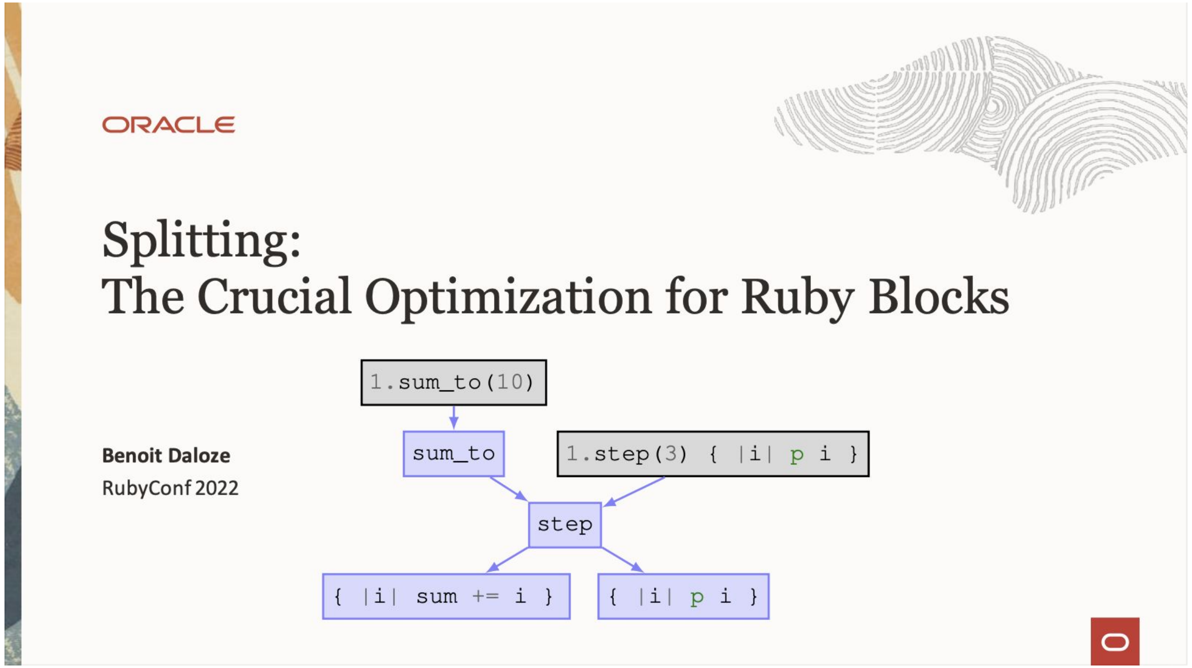
What else to check out

TRUFFLE RUBY

github.com/oracle/truffleruby/tree/master/src/main/ruby/truffleruby/core

graalvm.org/ruby

chrisseaton.com/truffleruby



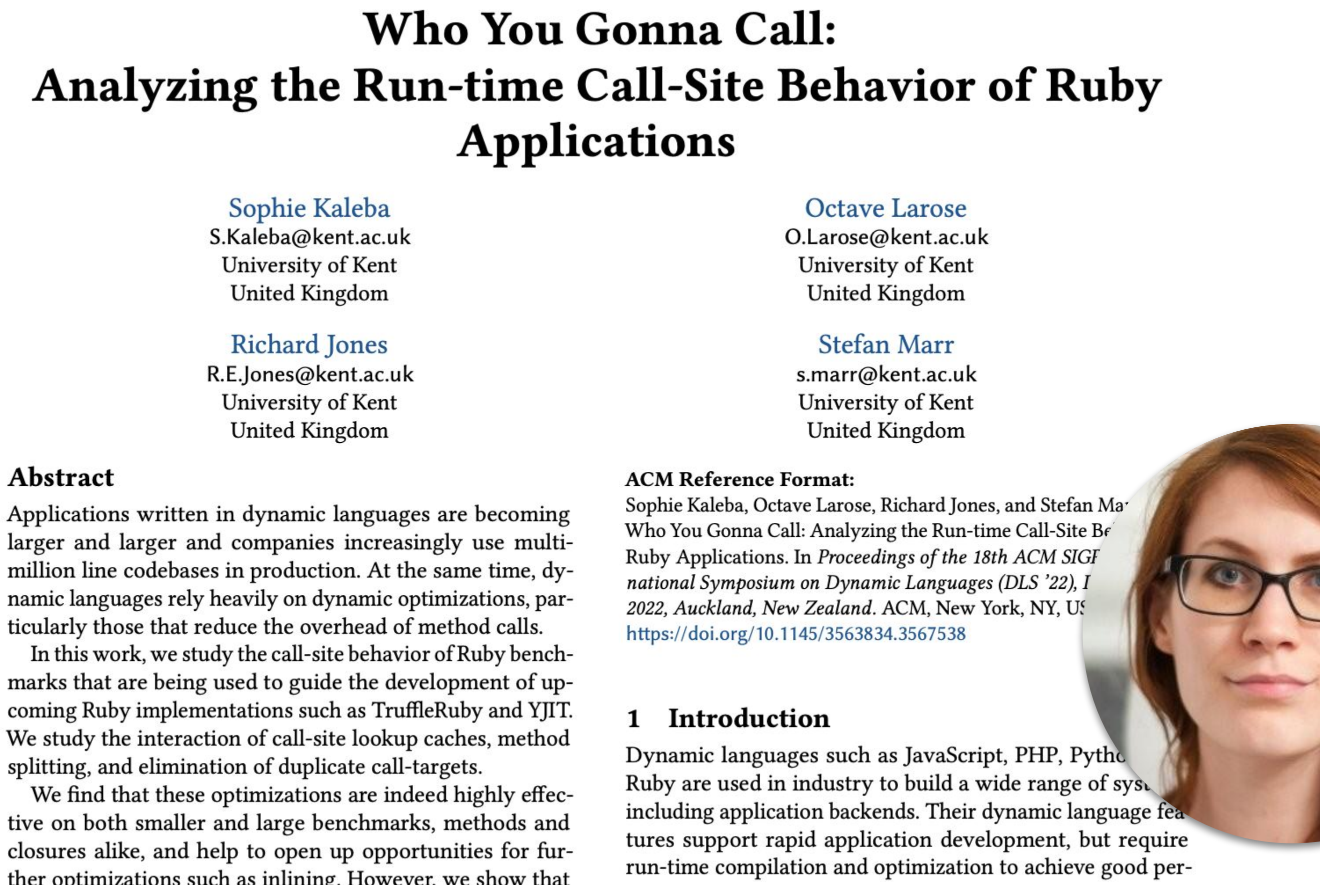
ORACLE

Splitting:
The Crucial Optimization for Ruby Blocks

Benoit Daloze
RubyConf 2022

$$\begin{tikzcd}[column sep=small]
 & \boxed{\texttt{1.sum\_to(10)}} \arrow[d] & & \\
 & \boxed{\texttt{sum\_to}} \arrow[dr] & & \boxed{\texttt{1.step(3) \{ |i| p i \}}} \arrow[dl] \\
 & & \boxed{\texttt{step}} \arrow[dl] \arrow[dr] & \\
 & \boxed{\texttt{\{ |i| sum += i \}}} & & \boxed{\texttt{\{ |i| p i \}}}
\end{tikzcd}$$

Benoit Daloze's talk today at 3pm in A



Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications

Sophie Kaleba
S.Kaleba@kent.ac.uk
University of Kent
United Kingdom

Octave Larose
O.Larose@kent.ac.uk
University of Kent
United Kingdom

Richard Jones
R.E.Jones@kent.ac.uk
University of Kent
United Kingdom

Stefan Marr
s.marr@kent.ac.uk
University of Kent
United Kingdom

Abstract

Applications written in dynamic languages are becoming larger and larger and companies increasingly use multi-million line codebases in production. At the same time, dynamic languages rely heavily on dynamic optimizations, particularly those that reduce the overhead of method calls.

In this work, we study the call-site behavior of Ruby benchmarks that are being used to guide the development of upcoming Ruby implementations such as TruffleRuby and YJIT. We study the interaction of call-site lookup caches, method splitting, and elimination of duplicate call-targets.

We find that these optimizations are indeed highly effective on both smaller and large benchmarks, methods and closures alike, and help to open up opportunities for further optimizations such as inlining. However, we show that

ACM Reference Format:
Sophie Kaleba, Octave Larose, Richard Jones, and Stefan Ma
Who You Gonna Call: Analyzing the Run-time Call-Site Be
Ruby Applications. In *Proceedings of the 18th ACM SIGP*
national Symposium on Dynamic Languages (DLS '22), I
2022, Auckland, New Zealand. ACM, New York, NY, US
https://doi.org/10.1145/3563834.3567538

1 Introduction

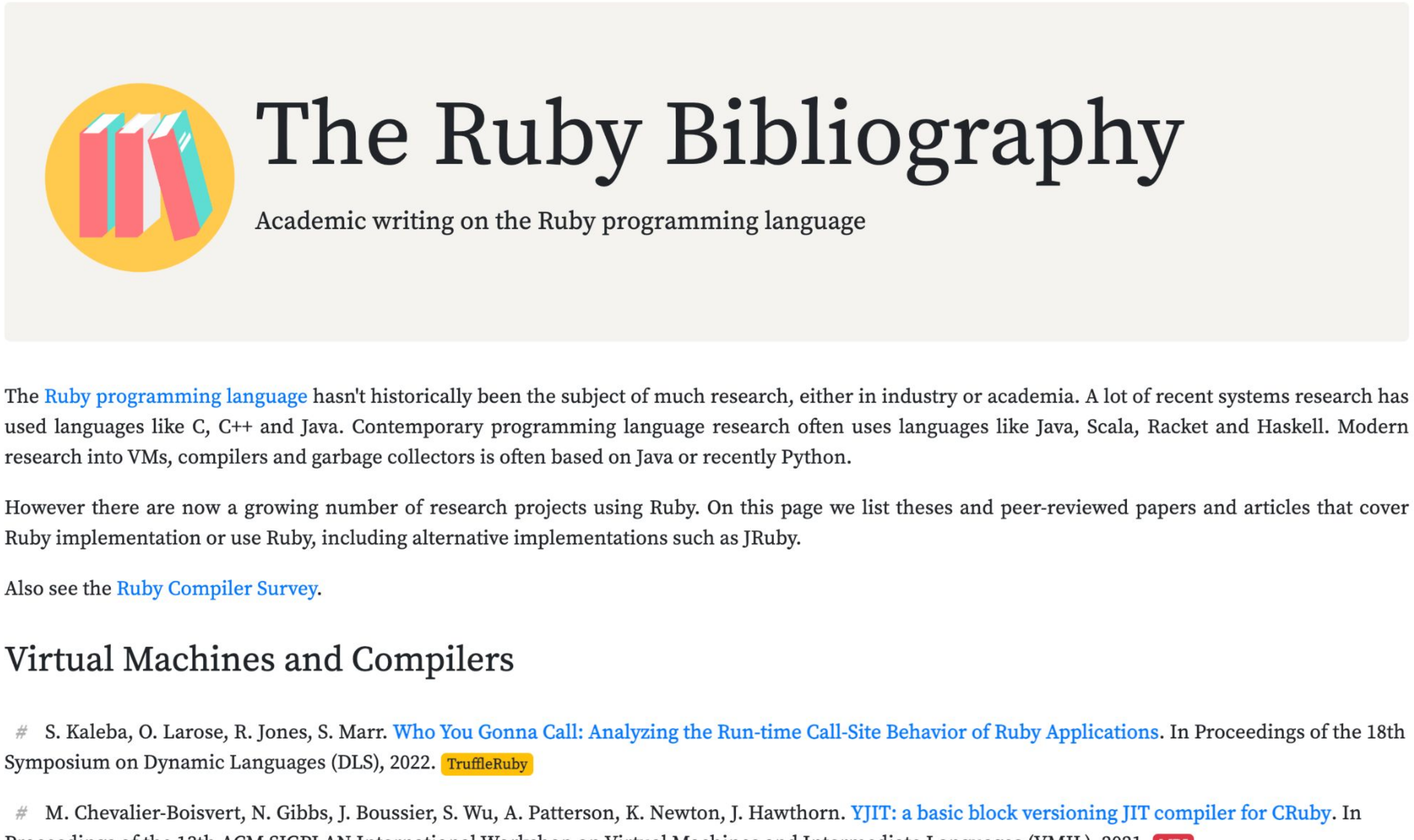
Dynamic languages such as JavaScript, PHP, Pytho
Ruby are used in industry to build a wide range of syst
including application backends. Their dynamic language fea
tures support rapid application development, but require
run-time compilation and optimization to achieve good per-

```ruby
def sweep
  each_object do |obj|
    if object_marked?(obj)
      unmark_object obj
    else
      reclaim_address obj.address
    end
  end
end
```

rbx/lib/mark_sweepgc.rb:79-87

ruby-compilers.com/rubinius/#history

rubybib.org



The Ruby Bibliography

Academic writing on the Ruby programming language

The Ruby programming language hasn't historically been the subject of much research, either in industry or academia. A lot of recent systems research has used languages like C, C++ and Java. Contemporary programming language research often uses languages like Java, Scala, Racket and Haskell. Modern research into VMs, compilers and garbage collectors is often based on Java or recently Python.

However there are now a growing number of research projects using Ruby. On this page we list theses and peer-reviewed papers and articles that cover Ruby implementation or use Ruby, including alternative implementations such as JRuby.

Also see the Ruby Compiler Survey.

Virtual Machines and Compilers

S. Kaleba, O. Larose, R. Jones, S. Marr. Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications. In Proceedings of the 18th Symposium on Dynamic Languages (DLS), 2022. TruffleRuby

M. Chevalier-Boisvert, N. Gibbs, J. Boussier, S. Wu, A. Patterson, K. Newton, J. Hawthorn. YJIT: a basic block versioning JIT compiler for CRuby. In Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL), 2021.