How to write Ruby extensions with Crystal

@gaar4ica
Anna Shcherbinina

Artec 3D

I like Ruby

5

5 YO, a lot of stuff

So, I like Ruby

But, sometimes,
I need my code to work
faster

Case of possible bottle neck

```ruby
def fibonacci(n)
  n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2)
end
```

C

Ruby has C bindings

C is hard to learn and understand

Static types

Allocating memory

And tons of other complications for Ruby developer

I’m ruby developer,

I don’t want to write my code in C

NET

C# OWL PDF Redis

BASIC

Scala Node.js

C++ XML

I’m ruby developer,

I want to write my code in Ruby.

But to be honest…

I’m ruby developer,
I want to write my code in Ruby

Also, I like Crystal

Crystal has ruby-like

Spoiler about crystal

Guess the language

```
# define class Dog
class Dog
  def initialize(breed, name)
    @breed = breed
    @name = name
  end

  def bark
    puts 'Ruff! Ruff!'
  end

  def display
    puts "I am of #{@breed} breed and my name is #{@name}"
  end
end

# make an object
d = Dog.new('Labrador', 'Benzy')
```

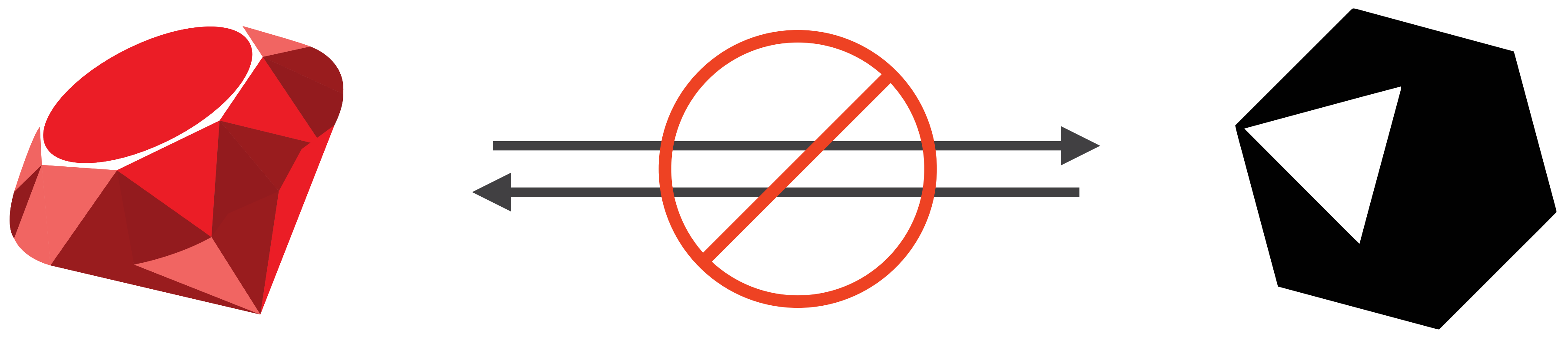
Crystal compiles to native code

Spoiler about crystal

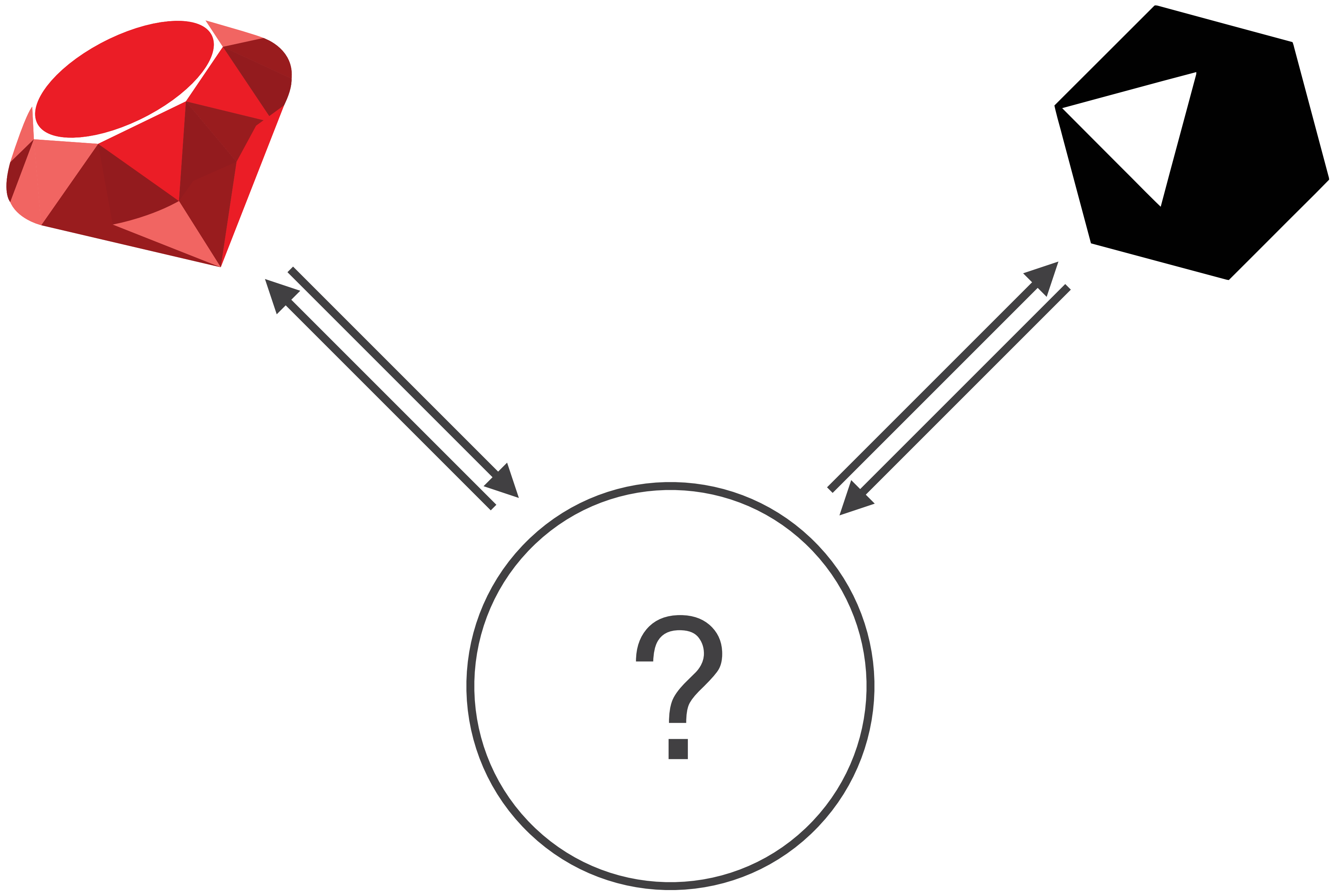
S stal

Google
Russia
Search Google or type URL

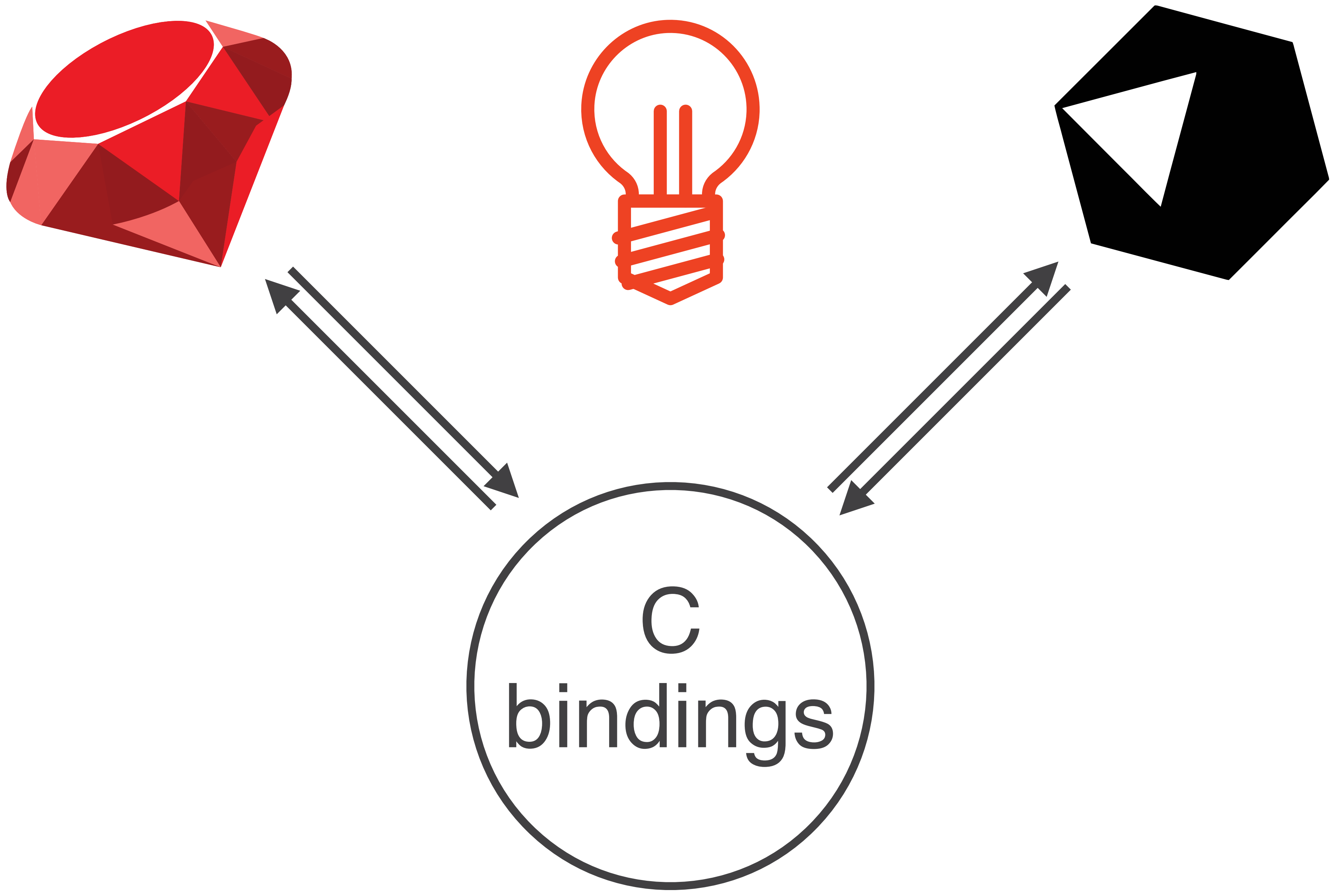
No way



No way



C bindings



C layer

Link Crystal classes and methods to Ruby, so Ruby knows about them

Translate Ruby-types to Crystal-types

So, Crystal lang

WUT?

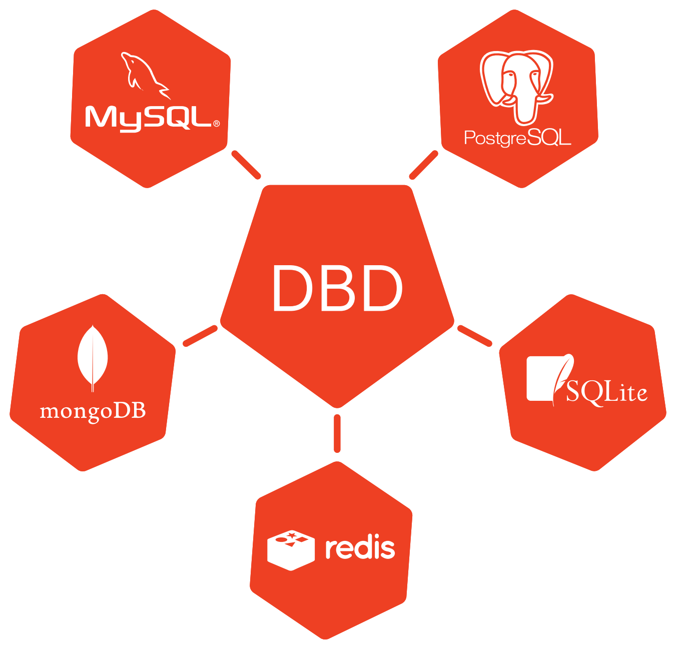
Compile to efficient native code

Statically type-checked

Ruby-inspired syntax

C bindings

Libs



Web Frameworks

- amatista
- amethyst
- kemal
- moonshine

Search

Cache

Testing

Third-party APIs

What for

Ready for production or not?

✗ Web apps

✓ Microservice

- maths calculation
- small but heavy parts of projects

Not so like ruby

Variables

Not so like ruby

The below doesn’t work with instance variables, class variables or global variables.

```
if @a
  # here @a can be nil
end

if @a.is_a?(String)
  # here @a is not guaranteed to be a String
end

if @a.responds_to?(:abs)
  # here @a is not guaranteed to respond to `abs`
end
```

Variables

Not so like ruby

To work with these, first assign them to a variable:

```
if a = @a
  # here a can't be nil
end

# Second option: use `Object#try` found in the
standard library
@a.try do |a|
  # here a can't be nil
end

if (a = @a).is_a?(String)
  # here a is guaranteed to be a String
end
```

Classes and methods

Not so like ruby

Overloading

Return types

Visibility

Finalize

Macros

Not so like ruby

```
class Object
  macro def instance_vars_names : Array(String)
    {{ @type.instance_vars.map &.name.stringify }}
  end
end

class Person
  def initialize(@name, @age)
  end
end

person = Person.new "John", 30
person.instance_vars_names #=> ["name", "age"]
```

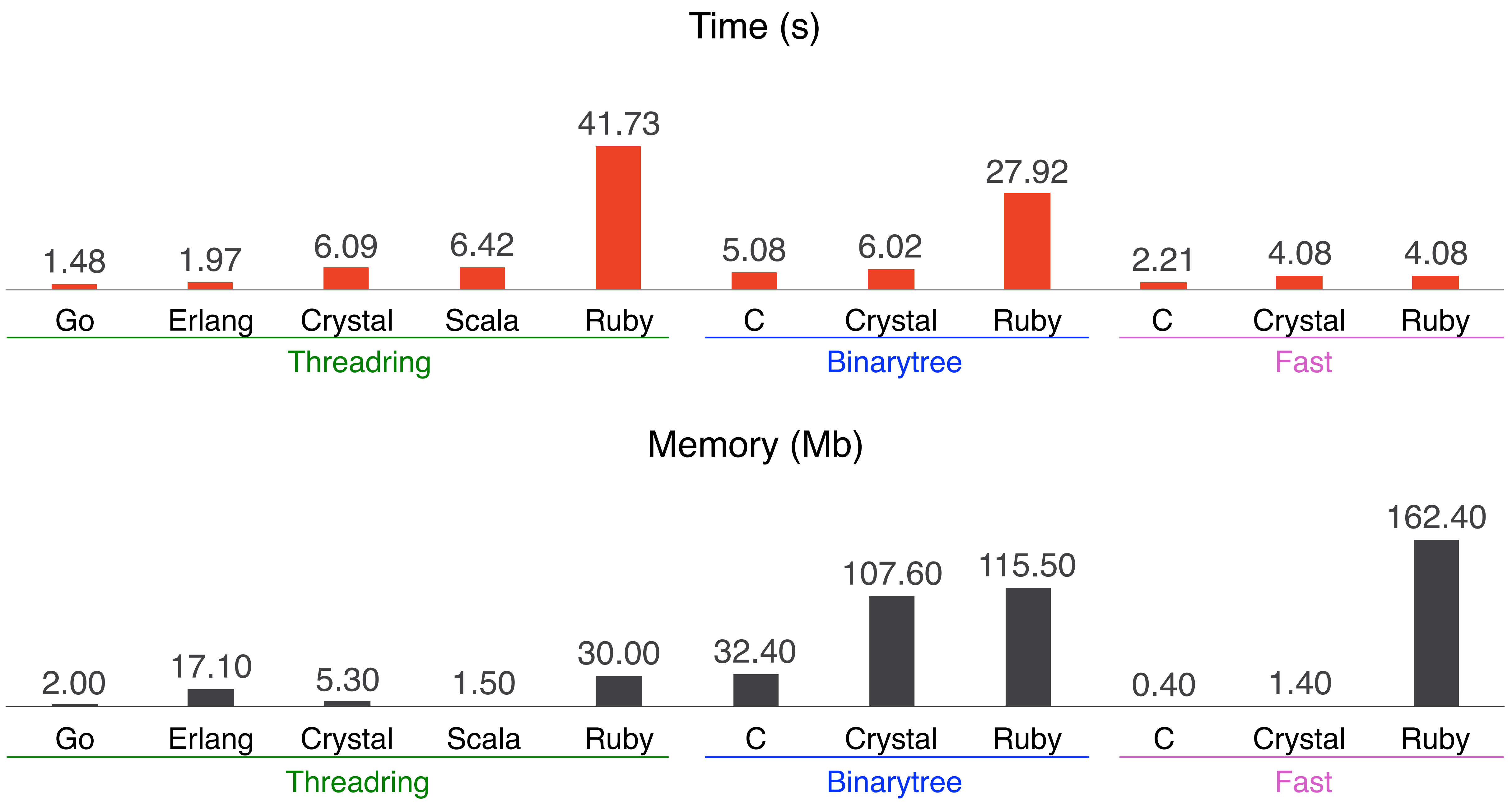
Threads

Not so like ruby

```
def generate(chan)
  i = 2
  loop do
    chan.send(i)
    i += 1
  end
end

def filter(in_chan, out_chan, prime)
  loop do
    i = in_chan.receive
    if i % prime != 0
      out_chan.send(i)
    end
  end
end
```

```
ch = Channel(Int32).new

spawn generate(ch)

100.times do
  prime = ch.receive
  puts prime
  ch1 =
Channel(Int32).new
  spawn filter(ch, ch1,
prime)
  ch = ch1
end
```

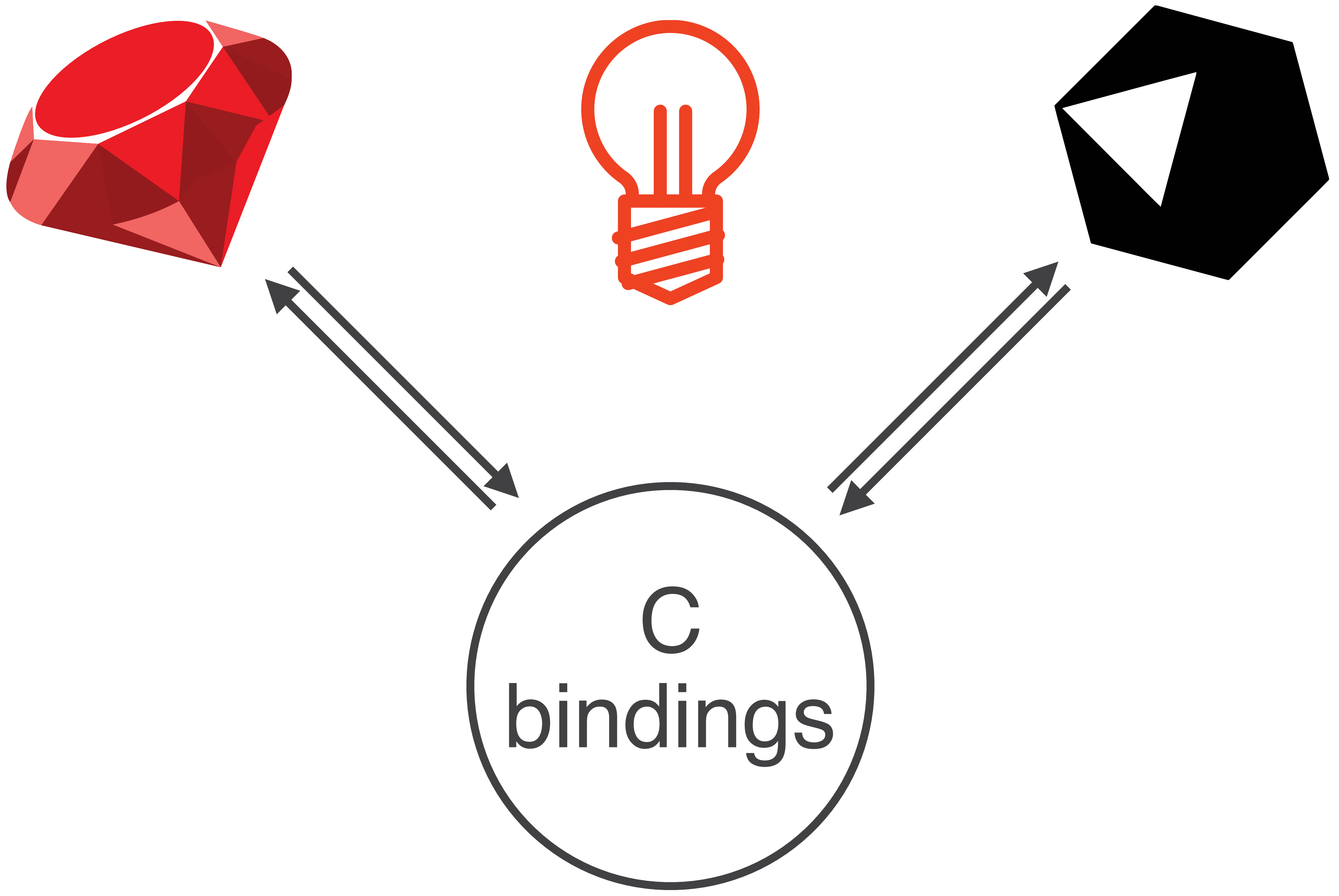
Benchmarking



Back to case of possible bottle neck

```ruby
def fibonacci(n)
    n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2)
end
```

Back to



Less work for C

Passing primitive data structures:

- integer
- string
- boolean
- float / decimal
- etc.

No allocation memory for complex data types

Don’t do a lot of work for casting variables passed as attributes

Ruby C API

Entry point

Then every C extension has to implement a function named Init_xxxxx (xxxxx being your extension’s name).

It is being executed during the “require” of your extension.

```
void Init_xxxxx() {
  // Code executed during "require"
}
```

Ruby C API
Types

Ruby C API defines a bunch of handful C constants defining standard Ruby objects:

C	Ruby
Qnil	nil
Qtrue	TRUE
Qfalse	FALSE
rb_cObject	Object
rb_mKernel	Kernel
rb_cString	String

Ruby C API

Declaring modules and classes

```
// Creating classes

// rb_define_class creates a new class named name
// and inheriting from super.
// The return value is a handle to the new class.

VALUE rb_define_class(const char *name, VALUE super);


// Creating modules

// rb_define_module defines a module whose name
// is the string name.
// The return value is a handle to the module.

VALUE rb_define_module(const char *name);
```

Ruby C API

Declaring methods

```
// Creating a method

// rb_define_method defines a new instance method in the class
// or moduleklass.
// The method calls func with argc arguments.
// They differ in how they specify the name - rb_define_method
// uses the constant string name

void rb_define_method(VALUE klass, const char *name, VALUE
(*func)(ANYARGS), int argc);

// rb_define_protected_method and rb_define_private_method are
similar to rb_define_method,
// except that they define protected and private methods,
respectively.

void rb_define_protected_method(VALUE klass, const char *name,
VALUE (*func)(ANYARGS), int argc);
void rb_define_private_method(VALUE klass, const char *name,
VALUE (*func)(ANYARGS), int argc);
```

Ruby C API

Ruby objects → C types

```
// Convert Numeric to integer.
long rb_num2int(VALUE obj);

// Convert Numeric to unsigned integer.
unsigned long rb_num2uint(VALUE obj);

// Convert Numeric to double.
double rb_num2dbl(VALUE);

// Convert Ruby string to a String.
VALUE rb_str_to_str(VALUE object);
char* rb_string_value_cstr(volatile VALUE* object_variable);
```

Ruby C API

C types → Ruby objects

```
// Convert an integer to Fixnum or Bignum.
INT2NUM( int );

// convert an unsigned integer to Fixnum or Bignum.
UINT2NUM( unsigned int );

// Convert a double to Float.
rb_float_new( double );

// Convert a character string to String.
rb_str_new2( char* );

// Convert a character string to ID (for Ruby function names,
etc.).
rb_intern( char* );

// Convert a character string to a ruby Symbol object.
ID2SYM( rb_intern(char*) );
```

Ruby C API

Simple example

```ruby
# my_class.rb

class MyClass
  def my_method(param1, param2)
  end
end
```

```c
// my_class_ext.c

static VALUE myclass_mymethod(VALUE rb_self, VALUE rb_param1, VALUE
rb_param2)
{
  // Code executed when calling my_method on an object of class MyClass
}

void Init_xxxxx()
{
  // Define a new class (inheriting Object) in this module
  VALUE myclass = rb_define_class("MyClass", rb_cObject);
  // Define a method in this class, taking 2 arguments,
  // and using the C method "myclass_method" as its body
  rb_define_method(myclass, "my_method", myclass_mymethod, 2);
}
```

Ruby C API

Simple example

```ruby
# my_class.rb

class MyClass
  def my_method(param1, param2)
  end
end
```

```c
// my_class_ext.c

static VALUE myclass_mymethod(VALUE rb_self, VALUE rb_param1, VALUE
rb_param2)
{
  // Code executed when calling my_method on an object of class MyClass
}

void Init_xxxxx()
{
  // Define a new class (inheriting Object) in this module
  VALUE myclass = rb_define_class("MyClass", rb_cObject);
  // Define a method in this class, taking 2 arguments,
  // and using the C method "myclass_method" as its body
  rb_define_method(myclass, "my_method", myclass_mymethod, 2);
}
```

Ruby C API

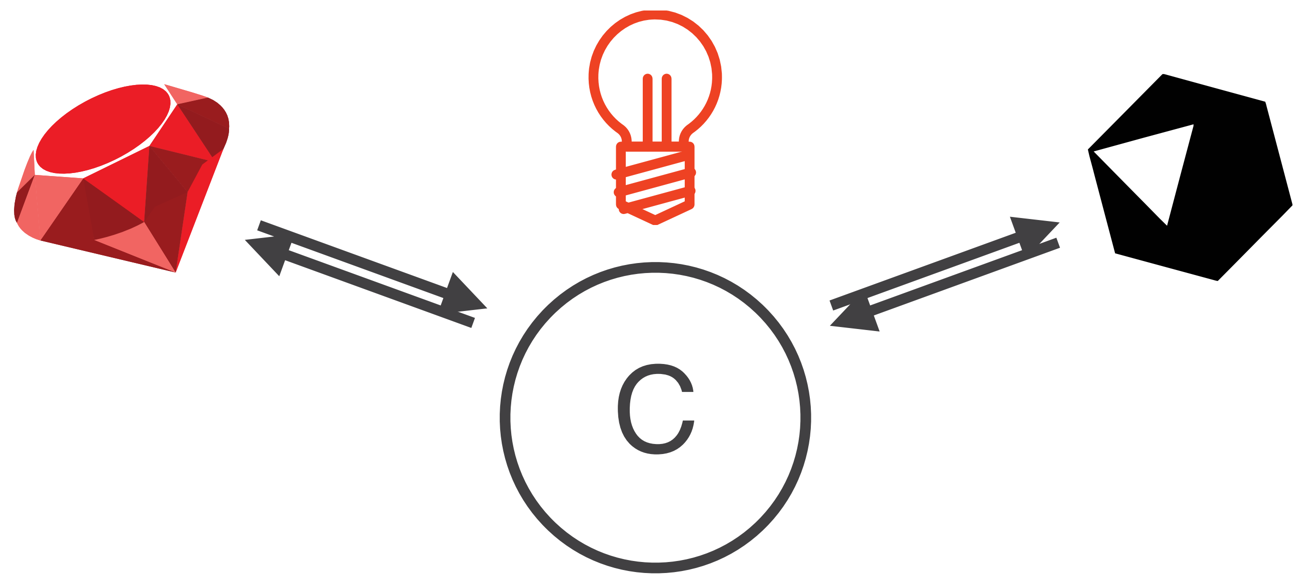
Simple example

```
# my_class.rb

class MyClass
  def my_method(param1, param2)
  end
end


// my_class_ext.c

static VALUE myclass_mymethod(VALUE rb_self, VALUE rb_param1, VALUE
rb_param2)
{
  // Code executed when calling my_method on an object of class MyClass
}

void Init_xxxxx()
{
  // Define a new class (inheriting Object) in this module
  VALUE myclass = rb_define_class("MyClass", rb_cObject);
  // Define a method in this class, taking 2 arguments,
  // and using the C method "myclass_method" as its body
  rb_define_method(myclass, "my_method", myclass_mymethod, 2);
}
```

Back to case of possible bottle neck

```ruby
def fibonacci(n)
  n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2)
end
```

And solution



Crystal extension

It's easy

A lib declaration groups C functions and types that belong to a library.

```
lib CrRuby
  type VALUE = Void*

  $rb_cObject : VALUE
end
```

Crystal extension

It’s easy

Every ruby object is treated as type VALUE in C.

```
lib CrRuby
  type VALUE = Void*

  $rb_cObject : VALUE
end
```

Crystal extension

It’s easy

rb_cObject is a C constant defining standard Ruby Object class.

```
lib CrRuby
  type VALUE = Void*

  $rb_cObject : VALUE
end
```

Crystal extension

It's easy

A fun declaration inside a lib binds to a C function.

We bind rb_num2int & rb_int2inum, to use it later.

```
lib CrRuby
  type VALUE = Void*

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE
end
```

Crystal extension

It's easy

We bind rb_define_class & rb_define_method.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_class(name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func:
METHOD_FUNC, argc: Int32)
end
```

Crystal extension

It's easy

Our new and shiny fibonacci method in Crystal.

```
def fibonacci_cr(n)
  n <= 1 ? n : fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

Do you see the difference with implementation in Ruby?

```
def fibonacci(n)
  n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2)
end
```

Crystal extension

It's easy

Let's create a wrapper for this method, used for:

- convert inbound Ruby-type parameter to Crystal;
- convert outbound result back to Ruby type.

```
def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)

  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end
```

Crystal extension

It’s easy

Now we are done with our functions and C bindings.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_class(name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end

def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

Crystal extension

It’s easy

Now we are done with our functions and C bindings.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_class(name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end

def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

Crystal extension

It’s easy

Now we are done with our functions and C bindings.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_class(name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end

def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

Crystal extension

It’s easy

Now we are done with our functions and C bindings.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_class(name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end

def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

Crystal extension

It’s easy

We bind **init** function, the first one to be called.

As you remember, a function named Init_xxxxx is being executed during the “require” of your extension.

```
fun init = Init_cr_math
end
```

Crystal extension

It’s easy

Firstly, we start garbage collector.

We need to invoke Crystal's "main" function, the one that initializes all constants and runs the top-level code.

We pass 0 and null to argc and argv.

```
fun init = Init_cr_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)
end
```

Crystal extension

It’s easy

We define class CrMath.

```
fun init = Init_cr_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)

  cr_math = CrRuby.rb_define_class("CrMath", CrRuby.rb_cObject)
  CrRuby.rb_define_method(cr_math, "fibonacci", ->fibonacci_cr_wrapper, 1)
end
```

Crystal extension

It’s easy

We define class CrMath.

Attach method fibonacci to class.

```
fun init = Init_cr_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)

  cr_math = CrRuby.rb_define_class("CrMath", CrRuby.rb_cObject)
  CrRuby.rb_define_method(cr_math, "fibonacci", ->fibonacci_cr_wrapper, 1)
end
```

Crystal extension

Ready to compile

We have Crystal library with C bindings.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_class(name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end
```

```
def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end

fun init = Init_cr_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)

  cr_math = CrRuby.rb_define_class("CrMath", CrRuby.rb_cObject)
  CrRuby.rb_define_method(cr_math, "fibonacci", ->fibonacci_cr_wrapper, 1)
end
```

Crystal extension

Ready to compile

We have method fibonacci_cr and wrapper for it.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_class(name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end
```

```
def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

```
fun init = Init_cr_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)

  cr_math = CrRuby.rb_define_class("CrMath", CrRuby.rb_cObject)
  CrRuby.rb_define_method(cr_math, "fibonacci", ->fibonacci_cr_wrapper, 1)
end
```

Crystal extension

Ready to compile

We have entry point.

```
lib CrRuby
  type VALUE = Void*
  type METHOD_FUNC = VALUE, VALUE -> VALUE

  $rb_cObject : VALUE

  fun rb_num2int(value : VALUE) : Int32
  fun rb_int2inum(value : Int32) : VALUE

  fun rb_define_class(name: UInt8*, super: VALUE) : VALUE
  fun rb_define_method(klass: VALUE, name: UInt8*, func: METHOD_FUNC, argc: Int32)
end

def fibonacci_cr_wrapper(self : CrRuby::VALUE, value : CrRuby::VALUE)
  int_value = CrRuby.rb_num2int(value)
  CrRuby.rb_int2inum(fibonacci_cr(int_value))
end

def fibonacci_cr(n)
  n <= 1 ? n :  fibonacci_cr(n - 1) + fibonacci_cr(n - 2)
end
```

```
fun init = Init_cr_math
  GC.init
  LibCrystalMain.__crystal_main(0, Pointer(Pointer(UInt8)).null)

  cr_math = CrRuby.rb_define_class("CrMath", CrRuby.rb_cObject)
  CrRuby.rb_define_method(cr_math, "fibonacci", ->fibonacci_cr_wrapper, 1)
end
```

Crystal extension

Ready to compile

Makefile

```
CRYSTAL = crystal
UNAME = "$(shell uname -ms)"
LIBRARY_PATH = $(shell brew --prefix crystal-lang)/embedded/lib

TARGET = cr_math.bundle

$(TARGET): cr_math.o
	$(CC) -bundle -L$(LIBRARY_PATH) -o $@ $^

cr_math.o: cr_math.cr
	$(CRYSTAL) build --cross-compile $(UNAME) $<
```

Crystal extension

Using in Ruby

```
$ irb
2.1.6 :001 > require './cr_math'
 => true
2.1.6 :002 > CrMath.new.fibonacci(20)
 => 6765
```

Crystal extension

Benchmarking

```ruby
# benchmark.rb

require 'benchmark'
require './cr_math' # We have compiled cr_math.bundle
require './rb_math'

iterations = 10_000
number = 20

Benchmark.bm do |bm|
  bm.report("rb") do
    iterations.times { RbMath.new.fibonacci(number) }
  end

  bm.report("cr") do
    iterations.times { CrMath.new.fibonacci(number) }
  end
end
```

```ruby
# rb_math.rb

class RbMath
  def fibonacci(n)
    return n if n <= 1

    fibonacci(n - 1) +
      fibonacci(n - 2)
  end
end
```

Crystal extension

Benchmarking

```ruby
# benchmark.rb

require 'benchmark'
require './cr_math' # We have compiled cr_math.bundle
require './rb_math'

iterations = 10_000
number = 20

Benchmark.bm do |bm|
  bm.report("rb") do
    iterations.times { RbMath.new.fibonacci(number) }
  end

  bm.report("cr") do
    iterations.times { CrMath.new.fibonacci(number) }
  end
end
```

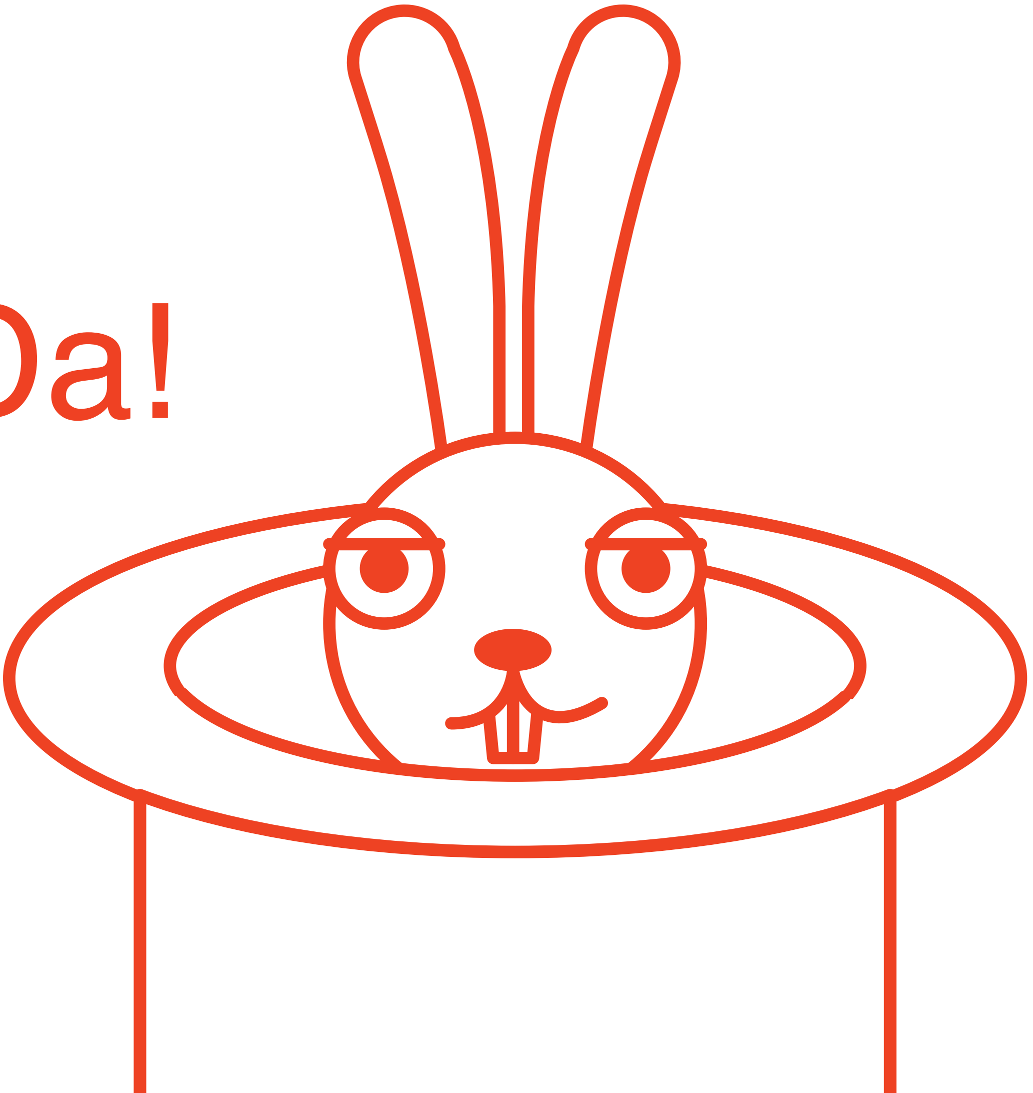
```ruby
# rb_math.rb

class RbMath
  def fibonacci(n)
    return n if n <= 1

    fibonacci(n - 1) +
      fibonacci(n - 2)
  end
end
```

Crystal extension

Benchmarking

```
$ ruby benchmark.rb
       user     system      total        real
rb 10.270000   0.030000  10.300000 ( 10.314595)
cr  0.460000   0.000000   0.460000 (  0.464087)
```

Ta-Da!



Thank you

https://twitter.com/gaar4ica

https://github.com/gaar4ica/ruby_ext_in_crystal_math

http://crystal-lang.org/

https://github.com/manastech/crystal

https://github.com/5t111111/ruby_extension_with_crystal

http://blog.jacius.info/ruby-c-extension-cheat-sheet/