v0dro / ra_grant_final.md Secret
Last active 10 minutes ago

Ruby Association Grant 2016 Final Report

ra_grant_final.md

Ruby Association Grant 2016 final term report

Author: Sameer Deshmukh

E-Mail: sameer.deshmukh93@gmail.com

GitHub: @v0dro

Contents

Overview

The CRuby interpreter, due to various reasons, is slow when it comes to raw performance. This limitation of speed is often circumvented by using C extensions that interface with external C libraries. For example, the Nokogiri gem, a popular library for XML/HTML parsing, makes use of the libxml C library for all the actual heavy lifting. As another notable example, the fast_blank gem is written completely as a C extension since C code is magnitudes faster than Ruby code.

However, C extensions force the programmer to face many problems:

- Difficult and irritating to write.
- Time consuming to debug.
- Tough to trace memory leaks.
- Change of mindset from high level to low level language.
- Familiarity with MRI C API.
- Changes in MRI C API breaks the whole gem.
- Need to care about small things.

Rubex is a new language that aims to simplify this process and bring writing Ruby C extensions within the grasp of even novice programmers. It is syntactically very similar to Ruby, and compiles to runnable C code that implicity interfaces with the CRuby C API.

All the original objectives as stated in grant proposal have been successfully completed. I will now elaborate on how the code that has been written so far can be used. At the end, I have included a small section on the future scope and plan for Rubex.

Installation

Rubex is still under heavy development, but you can still try Rubex by installing it from source.

Use the following commands:

```
git clone https://github.com/v0dro/rubex.git
cd rubex
rake install
```

This will install a binary called `rubex` on your system. In order to compile a Rubex file, issue the following command:

```
rubex path/to/file.rubex
```

Results

The proposed objectives to be met for the final term report were as follows:

- Ruby-style if-elsif-else blocks.
- Creating C static arrays.
- Looping support with Rubex's own syntax for for-loops and while loops.
- Support for struct, enum, union and typedefs.
- Interface for building C extensions.

All the objectives as mentioned in the grant proposal have been successfully completed.

Rubex code compilation

The following Rubex code uses all of the above Rubex constructs.

```
lib "<math.h>" do
  double cos(double)

  struct exception do
    int type
    char *name
  end

  double pow(double, double)

  alias exec = struct exception
end

def maths(double a, double b, c)
  alias int_64 = i64
  int_64 p = cos(a)
  int_64 rr

  rr = 332

  exec e
  e.type = 3

  struct new_struct do
    double a,b,c
    char* str
  end

  new_struct s
  s.a = a
  s.b = b

  if (s.a > s.b)
    s.str = c
  end

  return pow(6.7, s.a)
end
```

If you add the above code in a file called `c_bindings.rubex` , you can compile to C code using the command `rubex` c_bindings.rubex . This will generate a folder called c_bindings that will contain a file called c_bindings.c and a Ruby file called extconf.rb . Running the extconf.rb file will generate a Makefile that can be used for compiling c_bindings.c using the make command.

The full sequence of commands for compilation are as follows:

```
rubex c_bindings.rubex
cd c_bindings
ruby extconf.rb
make
```

This will create a shared object file called c_bindings.so that can be eventually used in any Ruby script.

You can now create a file called test.rb that will call the Ruby method maths that exists in the file c_bindings.rubex . This file can look like this:

```
require 'c_bindings.so'

puts maths(3, 5, "hello")
```

The C code that is generated by Rubex looks like this:

```
/* C extension for c_bindings.
This file in generated by Rubex. Do not change!
*/
#include <ruby.h>
#include <stdint.h>
#include <math.h>

typedef struct exception exec;

VALUE __rubex_f_maths (int argc, VALUE* argv, VALUE __rubex_arg_self);
VALUE __rubex_f_maths (int argc, VALUE* argv, VALUE __rubex_arg_self)
{
  typedef int64_t int_64;
  typedef struct new_struct
  {
    double __rubex_v_a;
    double __rubex_v_b;
    double __rubex_v_c;
    char* __rubex_ptr_str;
  } __rubex_t_new_struct;

  double __rubex_arg_a;
  double __rubex_arg_b;
  VALUE __rubex_arg_c;
  int_64 __rubex_v_p;
  int_64 __rubex_v_rr;
  exec __rubex_v_e;
  __rubex_t_new_struct __rubex_v_s;
  if (argc != 3)
  {
    rb_raise(rb_eArgError, "Need 3 args, not %d", argc);
  }
  __rubex_arg_a=NUM2DBL(argv[0])  ;
  __rubex_arg_b=NUM2DBL(argv[1])  ;
  __rubex_arg_c=argv[2]  ;
  __rubex_v_p = (int_64)(cos(__rubex_arg_a));

/* Rubex file location: spec/fixtures/c_bindings/c_bindings.rubex:19 */
  __rubex_v_rr = 332;

/* Rubex file location: spec/fixtures/c_bindings/c_bindings.rubex:22 */
  __rubex_v_e.type = 3;

/* Rubex file location: spec/fixtures/c_bindings/c_bindings.rubex:30 */
  __rubex_v_s.__rubex_v_a = __rubex_arg_a;

/* Rubex file location: spec/fixtures/c_bindings/c_bindings.rubex:31 */
  __rubex_v_s.__rubex_v_b = __rubex_arg_b;
  if (( __rubex_v_s.__rubex_v_a > __rubex_v_s.__rubex_v_b ))
  {

/* Rubex file location: spec/fixtures/c_bindings/c_bindings.rubex:34 */
    __rubex_v_s.__rubex_ptr_str = StringValueCStr(__rubex_arg_c);

  }

/* Rubex file location: spec/fixtures/c_bindings/c_bindings.rubex:37 */
  return   rb_float_new(pow(6.7,__rubex_v_s.__rubex_v_a));
}

void Init_c_bindings (void);
void Init_c_bindings (void)
{
  rb_define_method(rb_cObject ,"maths", __rubex_f_maths, -1);
}
```

Code Walkthrough

Now let us briefly walk through the Rubex code written above. I will elaborate on each construct of the code and its significance with the whole.

The first construct is the `lib` block. The `lib` block tells the Rubex compiler that functions from a C library are to be used in the program, and that it should include `<math.h>` in the C file that it generates. The block attached to `lib` contains functions and structs that will be used in the Rubex program. These declarations are parsed by Rubex, which gives it an understanding of the function prototypes and symbols.

The `lib` block also contains the `alias` statement. The `alias` statement is similar to `typedef` in C. It tells Rubex that from now on, the keyword `exec` shall be used in place of `struct exception` .

Moving on the `maths` method, this method is function that accepts 3 arguments: the first, `double a` is of the C type `double` , so the second argument `double b` . The type of the third argument has been left out, and Rubex assumes that a Ruby object will be passed as the third argument `c` .

We then use another alias statement for demonstration purposes. This particular alias will `typedef` the C `int64_t` as `int_64` . After that are a bunch of self-explanatory C type declarations and initializations. Notice in particular the `int_64 p = cos(a)` statement. It is calling the `cos()` function from the `math.h` header file and setting the value into `p` . The typecast from `double` (which is returned by `cos()`) to `int_64` (which is the type of `p`) is done automatically by the Rubex compiler as can be seen in this line from the generated C code:

```
__rubex_v_p = (int_64)(cos(__rubex_arg_a));
```

We are then defining a C struct called `new_struct` . It contains some variables of type `double` and `char*` . Variables of type `new_struct` can be defined by simply specifying the name of the type and followed by the name of the variable (just like any other variable declaration). Variables of this type can be declared the way any other variable can be (for example `new_struct s`).

You can then see that some of the elements within the struct have been assigned to other values. The way to do this is straight forward and is very similar to C code (using the dot (`.`) operator). In retrospect, the Rubex code for assigning values to certain elements in a struct will be translated to the following C code:

```
/* Rubex file location: spec/fixtures/c_bindings/c_bindings.rubex:30 */
  __rubex_v_s.__rubex_v_a = __rubex_arg_a;

/* Rubex file location: spec/fixtures/c_bindings/c_bindings.rubex:31 */
  __rubex_v_s.__rubex_v_b = __rubex_arg_b;
```

We then have an `if` statement. This `if` statement executes the block of code enclosed within it if the condition is true. The block of code contains something interesting. When the Ruby object `c` is assigned to a `char*` variable, Rubex assumes that `c` is a `String` and automatically converts the Ruby String to a C String.

To demonstrate, the Rubex code for this purpose looks something like this:

```
if (s.a > s.b)
```

```
  s.str = c
end
```

The compiled C code looks like this:

```
if (( __rubex_v_s.__rubex_v_a > __rubex_v_s.__rubex_v_b ))
{
  __rubex_v_s.__rubex_ptr_str = StringValueCStr(__rubex_arg_c);
}
```

As you can see, an implicit conversion from Ruby string to C string has been performed by Rubex using the `StringValueCStr` macro from the Ruby C API.

The last statement is a `return` statement that returns the value of `6.7` raised to `s.a`. This code is translated as follows:

```
return   rb_float_new(pow(6.7,__rubex_v_s.__rubex_v_a));
```

Here it can be seen that Rubex is implicity converting the `double` value returned by `pow` into a Ruby `Float` using the `rb_float_new()` function. This ensures that the `maths` method always returns a Ruby object.

Real life use case

For demonstration purposes, let us see a real life use case of Rubex where we port the fast_blank gem, which is written in C, to Rubex. You can see the C code for the fast_blank gem here. We will try to port most of the functionality of fast_blank to Rubex.

This is what the Rubex implementation of fast_blank will look like:

```
lib "<ruby.h>" do
  char* RSTRING_END(object)
  int RSTRING_LEN(object)
  int RSTRING_PTR(object)
end

lib "<ruby/encoding.h>" do
  struct rb_encoding do

  end

  rb_encoding *rb_enc_from_index(int)
  int ENCODING_GET(object)
  unsigned int rb_enc_codepoint(char *, char *, rb_encoding *)
  int rb_enc_codelen(int, rb_encoding *)
  int rb_isspace(int)
end


def blank?(string)
  rb_encoding *enc
  char *s
  char *e
  int n
  unsigned int cc

  enc = rb_enc_from_index(ENCODING_GET(string))
  s = RSTRING_PTR(string)
  e = RSTRING_END(string)

  if !s || (RSTRING_LEN(string) == 0)
    return true
  end

  while s < e do
    cc = rb_enc_codepoint(s, e, enc)
    n = rb_enc_codelen(cc, enc)

    if !rb_isspace(cc) && cc != 0
      return false
    end

    s += n
  end

  return true
end
```

Here is the code used for benchmarking the above program against fast_blank and the native Ruby implementation:

```
  require_relative 'blank.so'
  require 'fast_blank'
  require 'benchmark/ips'

  a = "    "*666 + "があるん"
  Benchmark.ips do |x|
    x.report("fast_blank") do
      a.blank?
    end

    x.report("Rubex blank?") do
      blank?(a)
    end

    x.report("strip!") do
      a.strip! == ""
    end

    x.report("strip") do
      a.strip == ""
    end

    x.compare!
  end

  # Benchmarks

  # Warming up --------------------------------------
  #           fast_blank     3.329k i/100ms
  #         Rubex blank?     2.286k i/100ms
  #               strip!   233.477k i/100ms
  #                strip   196.615k i/100ms
  # Calculating -------------------------------------
  #           fast_blank      8.552M (± 4.7%) i/s -     42.605M in   4.994504s
  #         Rubex blank?      8.346M (± 4.9%) i/s -     41.550M in   4.993115s
  #               strip!      5.330M (± 2.7%) i/s -     26.850M in   5.041363s
  #                strip      3.604M (± 2.6%) i/s -     18.089M in   5.022386s

  # Comparison:
  #           fast_blank:  8551816.4 i/s
  #         Rubex blank?:  8345618.8 i/s - same-ish: difference falls within error
  #               strip!:  5329965.1 i/s - 1.60x  slower
  #                strip:  3604111.5 i/s - 2.37x  slower

  # [Finished in 28.2s]

  # Conclusion: fast_blank and Rubex version of blank? are almost the same in terms of speed.
```

As can be seen in the above benchmarks, Rubex is somewhat slower than the native fast_blank implementation since it is not yet capable of outputting highly optimized C code. However, it is significantly faster than using String#strip or String#strip!.

Future Work

The future plan for Rubex has already been laid out and can be found on this wiki.

Work for implementing this has already started and should be ready in a few months.

Once the v0.1 goals are complete the above Rubex code should become significantly simpler since many of the functions from `ruby.h` will be shipped with the Rubex gem upon installation and Rubex will be made aware of simple Ruby data types like strings, hashes and arrays.

Talks

I had the opportunity to talk about the current progress of Rubex at Ruby Conf India 2017. You can find the video of the talk here, and the slides here.

Acknowledgements

I would like to sincerely thank the Ruby Association and Mr. Koichi Sasada for the resources and mentorship that they provided for this project.