Automated Type Contracts Generation in Ruby

Nickolay Viuginov
Software engineer chairs
St.-Petersburg State University
St.-Petersburg, Russia
Email: viuginov.nickolay@gmail.com

Valentin Fondaratov
Team Lead in RubyMine
JetBrains
St.-Petersburg, Russia
Email: fondarat@gmail.com

***Abstract*—Elegant syntax of Ruby language pays back when it comes to finding bugs in large codebases. Static analysis is hindered[1] by specific capabilities of Ruby, such as defining methods dynamically and evaluating string expressions. Even in dynamically typed languages, type information is very useful, because of better type safety and more reliable checks. One may annotate the code with YARD (Ruby documentation tool) which also enables improved tooling such as code completion.**

This paper reports a new approach to type annotations generation. We trace direct method calls while the program is running, evaluate types of input and output variables and use this information to derive implicit type annotations.

Each method or function is associated with a finite-state automaton, to which all variants of typed signatures are added. Then an effective compression technique is applied to the automaton, which reduces the cost of storage and allows to display the collected information in a human-readable form.

***Index Terms*—Ruby, Dynamically typed languages, Ruby VM, YARV, Method signature, Type inference, Static code analysis.**

I. Introduction

Developers suffer from time-consuming investigations with a goal to understand why the particular piece of code does not work as expected. The dynamic nature of Ruby allows for great possibilities which has a drawback — the codebase as a whole becomes entangled and investigations become more difficult compared to the statically typed languages like Java or C++. Another downside of its dynamic features is a drastic static analysis performance reduction due to inability to resolve some symbols reliably.

Consider the dynamic method creation which is often done with `define_method` call. Names and bodies of dynamically created methods may be calculated at runtime[2].

```
class User
  ACTIVE = 0
  INACTIVE = 1
  PENDING = 2

  attr_accessor :status

  def self.states(*args)
    args.each do |arg|
      define_method "#{arg}?" do
        self.status == User.const_get(arg.upcase)
      end
    end
  end
  states :active, :inactive, :pending
end
```

One of the possible workarounds is using code documentation tools like RDoc or YARD. As `@param` and `@return` annotations may help, they have several drawbacks, too:

- the type system used for documenting attributes, parameters and return values is pretty decent however it is not clear how to connect the types with each other. For example, `def []=` for array usually returns the same type as the second arg taking any type so in YARD this will look like `@param value [Object]`, `@return [Object]` which is not really helpful.
- from some perspective, such documentation itself a kind of contradicts the purpose of Ruby — to be as short, natural and expressive as possible.

The proposed approach is inspired by the way people tackle this problem manually: one may run or debug the program to inspect some valuable info about the code they're interested in. This suggests that collecting direct input and output types of the method dispatches during the program execution, post-processing and structuring of this data will make up implicit type annotations. As the process is automated, one can retrieve a plenty of information about the covered code.

The collected information not only might be used for YARD annotations generation but also could be stored in a public database to be shared and reused by different users in order to maximize the coverage of the analyzed code and the quality of the results. Moreover, generated implicit annotations can be built into the static analysis tools[3]) to improve existing and provide additional checks and code completion suggestions.

The project implementation can be divided into two main parts:

- At the first stage, the information about called methods and their input and output types is being collected though-out the script execution. It is very important to collect the necessary information as quickly as possible not to force users to wait for script completion many times longer compared to the case of the regular execution. To achieve this we implement a native extension, which receives all the necessary information directly from the internal stack of the virtual machine instead of using the standard API provided by language.
- At the second stage, the data obtained in the first stage is structured, reduced to a finite-state automaton and prepared for further use in code insight. This storage

scheme was chosen because of the ability to quickly obtain a regular expression that is easily perceived by a human.

II. Collecting information about method calls

Method parameters in Ruby have the following structure:

```
def m(a1, a2, ..., aM,  # mandatory(req)
    b1=(...), ..., bN=(...), # optional(opt)
    *c,                 # rest
    d1, d2, ..., dO,    # post
    e1:(...), ..., eK:(...), # keyword
    **f,                # keyword_rest
    &g)                 # block
```

TracePoint is a API allowing to hook several Ruby VM events like method calls and returns and get any data through Binding, which encapsulates the execution context(variables, methods) and retain this context for future use.

```
def foo(a, b = 1)
  b = '1'
end

TracePoint.trace(:call, :return) do |tp|
  binding = tp.binding
  method = tp.defined_class.method(tp.method_id)
  p method.parameters
  puts tp.event, (binding.local_variables.map do |v|
    "#{v}->#{binding.local_variable_get(v).inspect}"
  end.join ', ')
end

foo(2)
```

```
[[:req, :a], [:opt, :b]]
call
a->2, b->1
[[:req, :a], [:opt, :b]]
return
a->2, b->"1"
```

The big disadvantage of this approach is that calculation of full execution context is a time-consuming operation. But later we will need information only about a small part of it. Namely: types of arguments, types and names of method parameters. Creating a native extension for the Ruby VM[4], which will receive information about the method name directly from YARV instruction list (Figure 1) and receive information about argument types directly from the internal stack. Code analysis often handles direct method calls, so it is important to separate which arguments were directly passed to the method by the user, and which ones were assigned the default values. When Ruby VM hook the call event, all not specified optional arguments already initialized with default values. So we need to build one more native extension and gem this information from internal stack. Lets take a look at simple Ruby method with optional parameter and on appropriate bytecode.

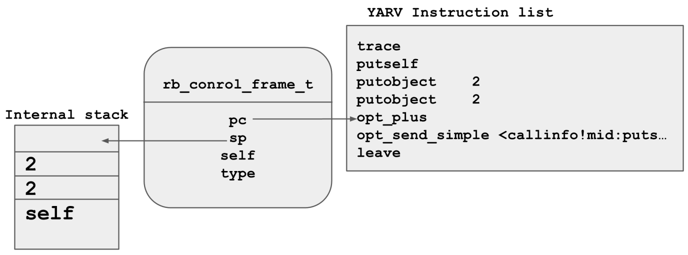


Fig. 1. YARV's internal registers.

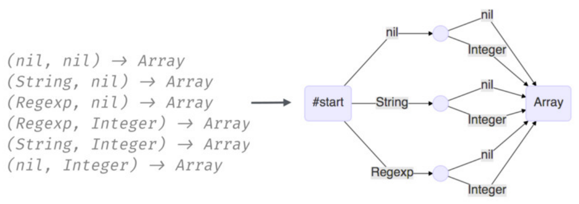
```
def foo(a, b=42, kw1: 1, kw2:, kw3: 3)
  #...
end

foo(1, kw1: '1', kw2: '2')
```

```
== disasm: #<ISeq:<compiled>@<compiled>>============
0000 trace      1
0002 putspecialobject 1
0004 putobject   :foo
0006 putiseq     foo
0008 opt_send_without_block
    <callinfo!mid:core#define_method, argc:2,
    ARGS_SIMPLE>
0011 pop
0012 trace      1
0014 putself
0015 putobject_OP_INT2FIX_O_1_C_
0016 putstring   "1"
0018 putstring   "2"
0020 opt_send_without_block <callinfo!mid:foo,
    argc:3, kw:[kw1,kw2], FCALL|KWARG>
0023 leave
== disasm: #<ISeq:foo@<compiled>>===================
0000 putobject   42
0002 setlocal_OP__WC__0 6
0004 trace      8
0006 putnil
0007 trace      16
0009 leave
```

So we need to find bytecode instruction for current method dispatch. For this, it is necessary to find caller control frame, and get last executed instruction in this frame. Thats how we get number of arguments(argc:) and list of key words(kw:[])

III. Transforming raw call data into contracts

Huge amount of raw signatures received from the Ruby process must be structured and processed so that it can be easily used and perceived. Each traced method is associated with a finite-state automaton. This storage structure allows you to quickly add raw data obtained from the Ruby process. It is also can easily be reduced to a human-readable regular expression.



```
(nil, nil) → Array
(String, nil) → Array
(Regexp, nil) → Array
(Regexp, Integer) → Array
(String, Integer) → Array
(nil, Integer) → Array
```

Fig. 2. Example of generating a non-minimized automaton.

In each automaton there is a single starting vertex and a single terminal vertex. To the automaton consistently added words obtained by concatenating signatures and corresponding output types. Then the minimization algorithm[5] is applied to this automaton. Quite often there are situations when the types of the two or more arguments of the method always coincide. Or even the type of the result coincides with type of one of the arguments. Consider an algorithm of processing such methods using method `equals` as an example.

```
def equals(a, b)
  raise StandardError if a.class != b.class
  a == b
end
p equals(1, 1) # (Integer, Integer) -> TrueClass
p equals(1, 2) # (Integer, Integer) -> FalseClass
...
```

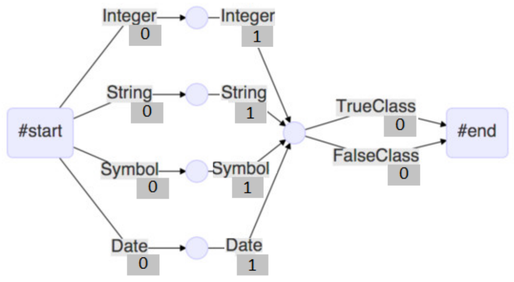


Fig. 3. Automaton with counted bit masks.

In this case, the types of arguments are the same, so the automaton can be markedly reduced. The algorithm consists in storing a bit mask on each automaton transition. Bit with one indicates a match of the type written on the edge with the corresponding previous type (Fig 3). In case the mask is greater than 0, the type written on the edge can be replaced with a mask. Subsequently, the transition along the edge will be carried out only when a readable signature is applied to the mask. After that, minimization algorithm is applied to the automaton one more time.

When during the code analysis it will be necessary to calculate the type returned by the method, we simply read through the automaton the set of input types of this method, and all the transitions from the vertex in which we turn out to be the desired output types.

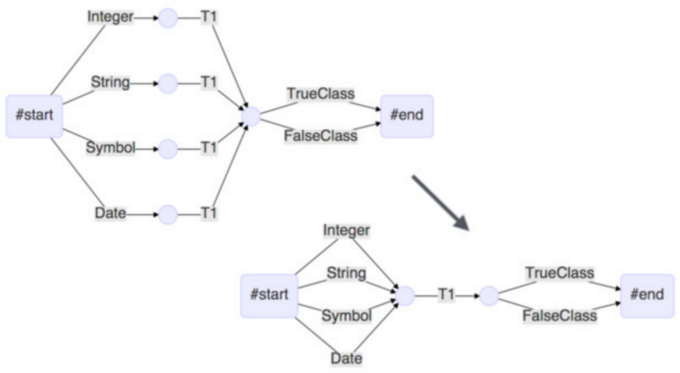


Fig. 4. Minimizing the automaton with reference edges.

IV. Conclusion

The paper describes the approach to generation of implicit type annotations.

This approach provides information about the types of methods that can not be obtained by static analysis of the source code in case if it is possible to understand in which library the method was declared and resolve the method receiver. This approach is useful for methods that are declared dynamically or in their body there are complex syntactic constructions, for example evaluating a string variable.

At the moment, we are working on optimizing the time of collecting of statistics. In the future, it will be necessary to test the approach, carry out load testing and implement the prototype in the existing static analysis of the Ruby language.

References

[1] Brianna M. Ren. The ruby type checker.
[2] blog.codeclimate. Gradual type checking for ruby, 2014.
[3] O. Shivers. *Control flow analysis in scheme*. ACM SIGPLAN 1988 conference on Programming language design and implementation, 1988.
[4] Pat Shaughnessy. *Ruby Under a Microscope*. No Starch Press, 2013.
[5] Jeffrey D. Ullman John E. Hopcroft, Rajeev Motwani. *Introduction to Automata Theory*. Addison-Wesley, 2001.