Automated Type Contracts Generation in Ruby

[1,2] *N. Y. Viuginov <viuginov.nickolay@gmail.com>*
[2] *V. S. Fondaratov <fondarat@gmail.com>*
[1] *St. Petersburg State University,*
13B Universitetskaya Emb., St. Petersburg, 199034, Russia
[2] *JetBrains,*
7-9-11 Universitetskaya Emb., St. Petersburg, 199034, Russia

Abstract. Elegant syntax of the Ruby language pays back when it comes to finding bugs in large codebases. Static analysis is hindered by specific capabilities of Ruby, such as defining methods dynamically and evaluating string expressions. Even in dynamically typed languages, type information is very useful as it ensures better type safety and more reliable checking whether the called method is defined for the object or whether the arguments of the correct types are passed to it. One may annotate the code with YARD (Ruby documentation tool) to declare the input and output types of methods or even declare methods that are added dynamically. These annotations improve the capabilities of 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 contracts. Each method or function is associated with a finite-state automaton consisting of all variants of typed signatures for this method. An effective compression technique is applied to the automaton to reduce the cost of storage and allows to display the collected information in a human-readable form. The exhaustiveness of the contract defined by the generated automaton depends on the diversity of the traced method usages. Therefore, it is also important to be able to merge all the automatons received from users into one, which is further covered in this paper.

Keywords: Ruby; dynamically typed languages; Ruby VM; YARV; method signature; type inference; static code analysis

DOI: 10.15514/ISPRAS-2017-29(4)-1

For citation: Viuginov N.Y., Fondaratov V.S. Automated Type Contracts Generation in Ruby. *Trudy ISP RAN/Proc. ISP RAS*, 2017, vol. 29, issue 4, pp. 7-20. DOI: 10.15514/ISPRAS-2017-29(4)-1

1. Introduction

Developers suffer from time-consuming investigations when trying to understand why a particular piece of code does not work as expected. The dynamic nature of Ruby allows for great possibilities, which has its drawback: the codebase as a whole becomes entangled and investigations become more difficult compared to statically typed languages like Java or C++ [1]. Another downside of its dynamic features is a drastic reduction in static analysis performance 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]. The following code dynamically adds active?, inactive? and pending? methods to the User class:

```
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 to get information about types for such difficult-to-analyze syntactic constructions is using code documentation tools such as RDoc or YARD. @!method annotation defines a method object with a given signature. *@param* and *@return* annotations may help to define the actual types, but 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 define relations between the types. For example, operator *[]=* 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, because all classes in Ruby are inherited from the *Object* and such annotation does not give any additional information about the method.
- from usability perspective, such documentation in some way 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 the needed info about the code they are investigating. This suggests that collecting direct input and output types of all method dispatches during the program execution with postprocessing and

structuring of this data may be considered as a way to automate manual investigations. As a result, it will make up implicit type annotations. As the process is automated, one can retrieve a lot of information about the executed code in the whole project.

Since the quality of the result highly depends on the code coverage of the programs run during the data collection, it is important to be able to merge the result annotations built for the same methods called from different places, projects and even users. These annotations 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 hence the quality of the generated contracts.

Two main contract generation stages can be distinguished:

- During the first stage, the information about called methods and their input and output types is collected throughout the script execution. It is very important to collect the necessary information as quickly as possible not to keep users waiting for script completion much longer compared to 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 the language. *This stage is described in Section 3.*
- During 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 provides the ability to quickly obtain a regular expression that is easily perceived by a human. *This stage is described in Section 4.*

The generated implicit annotations can be built into the static analysis tools [3] to improve existing and provide additional checks and code completion suggestions. *This stage is described in Section 5.*

2. Related works

In *Static Analysis of Dynamic Languages* [7], static analysis techniques for dynamically and statically typed languages are compared. The author notes that the attributes of dynamically typed languages such as flexibility and expressiveness limit the availability of tool-support for those languages. The paper addresses the main problems of analyzing code written in a language with dynamic typing: particularly, the construction of developer tools is difficult due to the lack of static type systems, therefore, many bugs are not discovered until run-time. The use of static analysis, and in particular whole program dataflow analysis, allow static reasoning about programs written in these languages without changing their nature or imposing unrealistic restrictions on the programmers.

In addition, the article mentions the technique called Use Analysis. “Use Analysis: A heuristic for recovering missing dataflow facts, due to missing library code, by

observing how applications objects are used in the application code." An example of such a heuristic is the approach to be described in this article.

For Ruby, as for most dynamically typed languages, there are tools for source code analysis, but they are not capable of statically identifying all errors associated with type mismatch. Here are some of them:

- Rubocop [4] — A Ruby static code analyzer, based on the community-driven Ruby style guide, but it does not allow actual error detection.
- Ruby-lint — A tool for detecting syntax errors, such as undeclared variables, an invalid argument set for calling a method, or unreachable sections of code.
- Diamondback Ruby [5] — an extension to Ruby that aims to bring the benefits of static typing to Ruby. However, at the moment, it is impossible to analyze even the standard Ruby library.

3. Collecting information about method calls

3.1 Calls structure

Method parameters in Ruby have the following structure:

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

An example of calling this method:
m(11,  12,  21,  22,  1,  2,  3,  '1',  '2',  e1:  1,  e2:  2,  e3:  3)  {...}
# a1   a2   b1   b2   ---c----   d1   d2   e1       e2       f        g
```

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

Consider a simple Ruby method declaration and handlers set for :call and :return events.

```
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)

The execution output will be:

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

On each method call, the following information is to be obtained:

- method name
- method receiver class
- arity (names and types of parameters)
- types of arguments and return type, hereinafter "**raw type tuple"**
- name and version of gem (ruby library) in which the method was declared
- location of method declaration

3.2 Unspecified arguments

Code analysis often handles direct method calls, so in order to calculate the return type it is important to distinguish which arguments were directly passed to the method by the user, and which were assigned the default values.

Let the following expression occur during the code analysis: `a, b, c = foo, foo('1'), foo(1),` and the following two contracts be generated: `Int → Int, String → String.` If the method cannot be statically analyzed, then we cannot select a contract to apply to the method call without arguments.

Note that default values are assigned to unspecified optional arguments before the :call event is triggered. Therefore, with the standard API, it is impossible to calculate which arguments were passed to the method, and which were not. This poses a problem because it renders detection of the default value types impossible and, therefore, disables the calculation of the expected return type of calls with any

optional parameters unspecified. However, one can build a native extension for the Ruby VM[2] and get this information from an internal stack.

Consider a simple Ruby method with an optional parameter and on appropriate bytecode.

```
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>>==================
...
```

The instruction number 0020, which calls the method *foo*, has information characterizing the number of passed arguments and the list of passed named arguments. Now we need to find a bytecode instruction for the current method dispatch. It is necessary to find the caller control frame and the last executed instruction in this frame. This instruction will correspond to the call of the method that we are interested in.

The big disadvantage of this approach is that the calculation of the full execution context is a time-consuming operation. But later we will only need information about a small part of it. Namely: types of arguments, types and names of method parameters. Creating a native extension for the Ruby VM, which will receive information about the method name directly from YARV instruction list (Fig. 1), will help us to receive information about argument types directly from the internal stack.

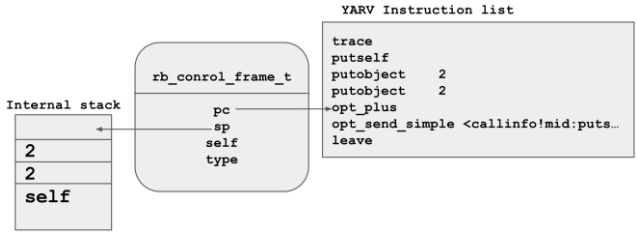

Fig. 1. YARV's internal registers.

4. Tranforming raw call data into contracts

A huge amount of raw data received from the Ruby process must be processed and structured so that it can be easily used and perceived. In our approach, each traced method is associated with a finite-state automaton. This storage structure allows to quickly add raw type tuple obtained from the Ruby process. It can be also easily reduced to a human-readable regular expression.

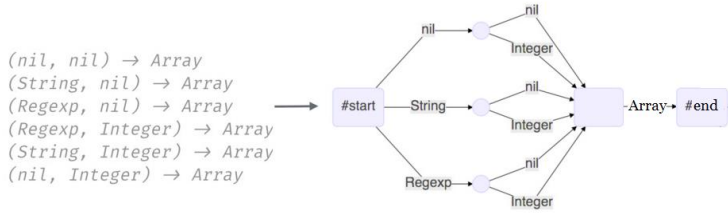

Fig. 2. Example of generating a non-minimized automaton.

In each automaton, there are a single starting vertex, from which the signature begins to be read and a single terminal vertex, in which all edges corresponding to the return types enters. Words obtained by concatenating tuples and corresponding output types are consistently added to the automaton.

```
Data: callArgs, returnType, automaton, parameterList
Result: automaton
tuple ← emptyList
for param : parameterList do
    if ∃arg : arg ∈ callArgs&&arg.getParam == param then
        tuple << arg
    else
        tuple << ϵ
    end
end
node ← automaton.startVertex
for arg : tuple do
    type ← arg.type
    if (node, type) ∉ automaton then
        automaton(node, type) ← newNode
    end
    node ← automaton(node, type)
end
automaton(node, returnType) ← automaton.termVertex
```

Algorithm 1. Adding a tuple to the automaton

Then, the minimization algorithm [7] is applied to the automaton, but it is slightly modified for the automaton of this type (Alg. 2). Note that all the tuples added to the automaton have the same length, so the resulting automaton has a layered structure based on the distance from the starting vertex. And all the edges emerging from the vertices of the i-th layer go to the vertices of i+1-st layer. Note that, after adding a signature to a minimized automaton, each added vertex can be combined only with the vertex of its level (Fig. 3).

***Theorem* 1.** *Only vertices from the same level can be joined during the minimisation.*

Proof. *Consider two vertices a and b from levels i and j ($i \neq j$). The vertices a and b join iff their transition functions coincide. All transitions from the vertices of level i lead to vertices of level $i + 1$, so transitions from a lead to vertices of level $i + 1$, and transitions from vertex b lead to vertices of level $j + 1$. It follows from the fact that the vertices adjacent to a and the vertices adjacent to b lie on different levels that the transition functions for the vertices a and b do not coincide.* □

***Corollary* 1.** *Let n be the number of layers of the automaton, then the computational complexity of the minimization algorithm after adding one tuple to the previously minimized automaton can be estimated as: $O(\sum_{i=1}^{n} automaton.levels[i])$ or $O(automaton.size)$, which is better than $O(automaton.size * n)$, as for the automaton in the general case.*

```
Data: automaton, nodes
Result: automaton
levels ← automaton.levels//splitting automaton for layers
for node : nodes, i++ do
    for nodeForComparison : levels[i] do
        if node.getTransitions = nodeForComparison.getTransitions then
            automaton.joinNodes(node, nodeForComparison);
        end
    end
end
```

Algorithm 2. Automaton minimisation

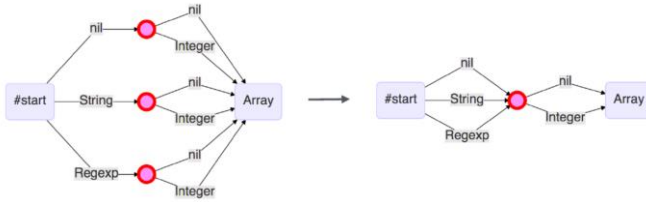

Fig. 3. Joining vertices

Quite often there are situations where types of two or more arguments of the method always coincide or the type of the result coincides with the type of one of the arguments. Consider 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
p equals(:b, :a)   # (Symbol, Symbol) -> FalseClass
p equals(:a, :a)   # (Symbol, Symbol) -> TrueClass
...
```

While adding the next transition from the vertex to the automaton, let's compare the symbol of the transition we want to add with all the previous symbols of the current tuple. In case there is at least one match, instead of a regular edge with a type symbol, edge with a bit mask is added. The length of this mask equals to the ordinal number of the current type within the tuple decreased by 1. i-th bit is 1 iff the i-th type in the tuple equals to the type to be added (Fig. 4).

```
Data: tuple, type, typeIndex
Result: mask
mask = 0
for i : [1..typeIndex − 1] do
    if tuple[i] == type then
        mask[i] = 1
    end
end
```

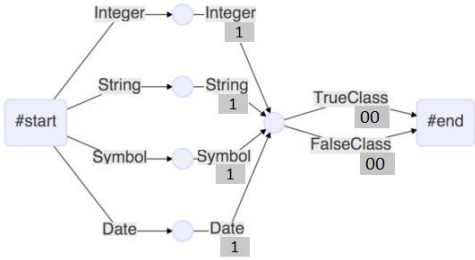

Fig. 4. Automaton with counted bit masks

When reading the signature, each following type is compared to the previous signature types and if a nonzero mask is obtained, one goes through the transition with the mask received.

```
for arg : tuple, i++ do
    type ← arg.type
    mask ← calculate_mask(tuple, type, i)
    if mask > 0 then
        if (node, mask) ∉ automaton then
            automaton(node, mask) ← newNode
        end
        node ← automaton(node, mask)
    else
        if (node, type) ∉ automaton then
            automaton(node, type) ← newNode node ← automaton(node, type)
        end
        node ← automaton(node, type)
    end
end
automaton(node, returnType) ← automaton.termVertex
```

Algorithm 1'. Adding a tuple to the automaton with masks

Theorem 2. *Before the minimization from the vertex cannot be the transition with a type symbol and the transition with an appropriate mask simultaneously.*

Proof. Consider two cases: the transition with the symbol was added before the transition with the mask and vice versa.
1) If an transition with a symbol was added before the transition with the mask, then when it was added, a non-zero mask should have been produced. Then instead of the usual transitions had to be added a transition with a mask.
2) If the transition with a mask was added first, then instead of adding an edge with a symbol, we should just go through the existing transition with mask. □

Corollary 1. *After minimization, the automaton with masks remains deterministic, that is, for every vertex and any type it is impossible to find both conventional and mask edges corresponding to that type simultaneously.*

Automata received from different users need to be merged. The following algorithm is used for this:

```
Data: automaton, additionalAutomaton
Result: automaton
bfsQueue.push(automaton.getStartNode, additionalAutomaton.getStartNode)
while !bfsQueue.empty do
    (oldNode, newNode) = bfsQueue.pop
    for transition : newNode.getTransitions do
        node = createNewNode
        oldNode.addTransition(transition, node)
        if transition ∈ oldNode.getTransitions then
            nodeToClone = oldNode.goByTransition(transition)
            node.getTransitions.add(nodeToClone.getTransitions)
        end
        nodes = (node, newNode.goByTransition(transition))
        if !used(nodes) then
            used.add(nodes)
            bfsQueue.push(nodes)
        end
    end
end
automaton.minimize
```

Algorithm 3. Automatons merge

In Ruby, Duck Typing [8] is quite heavily used. As a consequence, variables of various types that implement a set of methods can be passed as arguments to a method. Hence, many multiple edges corresponding to these classes appear in the automaton. These multiple edges can be replaced by one edge containing information about the interface that all these classes satisfy. Then, to jump on this edge, the next type from the signature must implement this interface. In case this common interface is empty on the edge, it is enough to write the type Object, since it is the parent class for all objects.

5. Using of contracts in static analysis algorithms

The contract is used to calculate the type returned when the method is called with a certain set of arguments. It is worth noting that the types of arguments are not always uniquely defined. Sometimes there is a set of types to which the variable may belong. To calculate the type returned by the method, it is necessary to go successively along the edges of the automaton calculating a set of vertices reachable by reading some sequence of types. The unspecified optional arguments types are imitated with a special non-alphabetic character so that the length of a tuple is lower than the automaton height by 1.

```
Data: argumentTypes, automaton
Result: returnType
node ← automaton.startVertex
index = 0
for type : argumentTypes, index++ do
    mask = calculate_mask(argumentTypes, type, index)
    if mask ∈ node.getTransitions then
        node = node.goByTransition(mask)
        continue
    end
    if type ∈ node.getTransitions then
        node = node.goByTransition(type)
        continue
    end
    return UNKNOWN_SET_OF_ARGUMENTS
end
returnType << node.getTransitions.types
```

Algorithm 4. Output type calculation

The generated contracts complement the type selection system because they allow to calculate the types returned from methods which were not successfully analyzed using standard tools. This expands the class of variables for which it is possible to statically compute a type.

The collected information for the methods makes it possible to significantly accelerate the existing control flow analysis because the methods for which a sufficiently representative contract is generated do not require additional analysis. Contracts allow to extend the applicability of some of the features that are supported in most modern IDEs. The functions considered are applicable to method calls for which it was possible to select the class of the object to which they were applied and for this class there is a contract corresponding to the method with that name and configuration of parameters. Functions in which contracts are applied:

- **Go To Declaration/Find Usages.** At the execution time information about method declaration was collected. This information can be used for navigation from method call to declaration and vice versa.
- **Autocompletion.** A list of methods implemented for an object can be supplemented with methods for which the contract was found.
- **'Incorrect method arguments' Inspection**. Information about the method parameters can be used to detect incorrect calls.

6. Conclusion

The paper describes the approach to the generation of implicit type contracts. This approach provides information containing type signatures of methods that cannot be obtained by static analysis using the source code given it is possible to understand in which library the method was declared and to resolve the method receiver. This approach is useful for analyzing programs which heavily utilize dynamic features like dynamic methods creation or when there are complex syntactic constructions in

methods implementations. In addition, this approach can be applied to other languages with dynamic typing, such as Python or JavaScript.

Several problems remain unsolved, such as Duck Typing and handling an ambiguous resolve of the argument type in a static analysis.

The problem with duck typing is that, during the execution of the program, it is impossible to save all the methods implemented for the object. Therefore, it is difficult to find the largest common interface for a group of classes.

The problem with arguments with types ambiguous according to the static analysis is that they cannot be read in the automaton.

References

[1]. Brianna M. Ren., J. Toman, T. Stephen Strickland and Jeffrey S. Foster. The ruby type checker. Available: http://www.cs.umd.edu/~jfoster/papers/oops13.pdf
[2]. blog.codeclimate. Gradual type checking for ruby, 2014. [Online]. Available: blog.codeclimate.com/blog/2014/05/06/gradual-type-checking-for-ruby/
[3]. O. Shivers. Control flow analysis in scheme. ACM SIGPLAN 1988 conference on Programming language design and implementation, 1988.
[4]. Bozhidar Batsov. Rubocop, 2017. [Online]. Available: http://batsov.com/rubocop/
[5]. Jeff Foster, Mike Hicks, Mike Furr, David An. Diamond-back ruby guide, 2009.[Online]. Available: http://www.cs.umd.edu/projects/PL/druby/manual/manual.pdf
[6]. Pat Shaughnessy. Ruby Under a Microscope. No Starch Press, 2013.
[7]. Madsen M. Static Analysis of Dynamic Languages. Available: http://pure.au.dk/ws/files/85299449/Thesis.pdf
[8]. Duck Typing [Online]. Available: http://rubylearning.com/satishtalim/duck_typing.html

Автоматизированная генерация типовых контрактов для языка Ruby

[1, 2] Н. Ю. Вьюгинов <viuginov.nickolay@gmail.com>
[2] В. С. Фондаратов <fondarat@gmail.com>
[1] СПбГУ,
199034, Россия, Санкт-Петербург, Университетская наб., 13В
[2] JetBrains,
199034, Россия, Санкт-Петербург, Университетская наб., 7-9-11

Аннотация. Элегантный синтаксис языка Ruby заметно усложняет поиск ошибок в больших кодовых базах. Статический анализ усложняется специфическими возможностями языка, такими как динамическое создание методов и исполнение строковых выражений. Даже в языках с динамической типизацией информация о типах важна, так как она позволяет улучшить типобезопасность и производить более надёжные статические проверки того, определён ли метод для объекта и передан ли метода корректный набор аргументов. Одним из путей решения проблемы является использование YARD нотаций. Они позволяют задокументировать входные и выходный типы методов или даже декларировать методы, добавляемые динамически.

Такие аннотации позволяют улучшить анализ кода и автодополнение. В статье описывается новый подход к генерации типовых аннотаций. Мы отслеживаем непосредственные вызовы метода во время исполнения программы и сохраняем типы аргументов и выходной тип. На основе собранной информации для каждого метода строится неявная типовая аннотация. Каждому автомату сопоставляется конечный автомат, составленный из различных типовых сигнатур метода. К автомату применяется эффективный алгоритм минимизации с целью снизить затраты на хранение и позволяет привести автомат к виду, который может быть легко представлен в виде регулярного выражения. В сгенерированном автомате учитывается только та функциональность метода, которая была покрыта программой, которую исполнил пользователь. Поэтому в подходе предусмотрено объединение автоматов, полученных у разных пользователей с целью увеличения репрезентативности и покрытия функциональности метода.

Ключевые слова: Ruby; динамически типизированные языки; Ruby VM; YARV; сигнатура метода; наследование типов; статический анализ кода

DOI: 10.15514/ISPRAS-2017-29(4)-1

Для цитирования: Вьюгинов Н. Ю., Фондаратов В. С. Автоматизированная генерация типовых контрактов для языка Ruby. Труды ИСП РАН, том 29, вып. 4, 2017 г., стр. 7-20 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(4)-1

Список литературы

[1]. Brianna M. Ren., J. Toman, T. Stephen Strickland and Jeffrey S. Foster. The ruby type checker. Доступно по ссылке: http://www.cs.umd.edu/~jfoster/papers/oops13.pdf
[2]. blog.codeclimate. Gradual type checking for ruby, 2014. [Online]. Доступно по ссылке: blog.codeclimate.com/blog/2014/05/06/gradual-type-checking-for-ruby/
[3]. O. Shivers. Control flow analysis in scheme. ACM SIGPLAN 1988 conference on Programming language design and implementation, 1988.
[4]. Bozhidar Batsov. Rubocop, 2017. [Online]. Доступно по ссылке: http://batsov.com/rubocop/
[5]. Jeff Foster, Mike Hicks, Mike Furr, David An. Diamond-back ruby guide, 2009.[Online]. Доступно по ссылке: http://www.cs.umd.edu/projects/PL/druby/manual/manual.pdf
[6]. Pat Shaughnessy. Ruby Under a Microscope. No Starch Press, 2013.
[7]. Madsen M. Static Analysis of Dynamic Languages. Доступно по ссылке: http://pure.au.dk/ws/files/85299449/Thesis.pdf
[8]. Duck Typing [Online]. Доступно по ссылке: http://rubylearning.com/satishtalim/duck_typing.html