Санкт-Петербургский государственный университет
Математическое обеспечение и администрирование информационных систем

Системное программирование

Вьюгинов Николай Юрьевич

Динамическое выделение типов в Ruby

Выпуская квалификационная работа

Научный руководитель:
ст.преп. кафедры системного программирования Я. А. Кириленко

Консультант:
программист ООО "ИнтеллиДжей Лабс" В. С. Фондаратов

Рецензент:
к.ф.-м.н., доцент кафедры системного программирования Д. Ю. Булычев

Санкт-Петербург
2017

SAINT PETERSBURG STATE UNIVERSITY

Software Engineering

Viuginov Nickolay

Dynamic type inference in Ruby

Graduation Project

Scientific supervisor:
Senior Lecturer Iakov Kirilenko

Consultant:
Software Developer at IntelliJ Labs OOO Valentin Fondaratov

Reviewer:
Associate Professor Dmitri Boulytchev

Saint Petersburg
2017

Оглавление

Введение

Ruby — довольно популярный в наше время язык программирования. Динамическая типизация придаёт языку гибкость и позволяет писать выразительный и ёмкий код. Он также позволяет определять поведение и реализацию программ во время исполнения, что явилось причиной порождения множества фреймворков, ускоряющих решение ряда задач. Один из таких фреймворков — Ruby on Rails. Он заметно упрощает и ускоряет создание web-приложений. Однако в наше время приложение не может долгое время оставаться без изменений, оно постоянно подвергается рефакторингу и в него добавляется новая функциональность. В больших приложениях, написанных на языках с динамической типизацией такие мелкие изменения ведут к усложнению кода в совокупности, а также к усложнению отладки и поиска ошибок.

Первыми на помощь пришли языки аннотаций, такие как RDoc [10] и YARD [19]. Они позволяют не только задокументировать возвращаемые и принимаемые типы методов, но даже декларировать создание динамических методов [21]. Полное документирование программ решает проблему, однако противоречит сущности языка — краткости и выразительности. Поэтому необходимо каким-то образом статически анализировать исходный код, чтобы получить информацию о типах аргументов и о возвращаемых типах методов.

Статический анализ тоже не позволяет разрешить все ситуации в силу специфики языка Ruby [12]. Поэтому было решено анализировать не исходный код программы, а фактические типы аргументов в процессе исполнения программы. Проанализировав большой набор таких данных, можно сделать некоторые выводы о типизированной сигнатуре методов. Например, составить неявные типовые аннотации для широко используемых библиотек. Это открывает широкие возможности, как для поиска ошибок в программах, так и для предложения подсказок их авторам при написании.

1. Постановка задачи

Целью данной работы является разработка и реализация модуля вывода контрактов [11] для методов языка Ruby и интеграция сгенерированных контрактов в систему статического анализа.

Для достижения этой цели были сформулированы следующие задачи.

1. Разработать архитектуру.
2. Реализовать окружение для запуска программ, собирающее данные о типах.
3. Разработать систему для вывода контрактов, описывающих сигнатуру метода.
4. Реализовать плагин для IDE, встраивающий контракты в систему статического анализа.
5. Протестировать модуль и провести нагрузочное тестирование.

2. Обзор

Поиск и устранение ошибок занимают значительную часть времени при разработке и сопровождении продукта, поэтому снижение вероятности появления ошибки — очень важная задача. В языках со строгой типизацией, таких как Java или C++ большая часть ошибок, связанных с несоответствием типов выявляется на этапе компиляции. В языках с динамической типизацией этот класс ошибок проявится только во время исполнения, но из-за этого процесс обнаружения и исправления ошибок становится гораздо дольше и сложнее.

Для таких языков, как Python, Ruby и JavaScript появилось довольно много интегрированных сред разразработки и инструментов для статического анализа исходного кода, которые позволяют выявлять довольно широкий класс ошибок во время разработки: Pylint, pyflakes, Муру для Python [13]; JSLint и Flow[5] для JavaScript, RIPS для PHP.

Pylint Инструмент для анализа кода на языке Python, поддерживающий большое количество функций, которые способны заметно уменьшить время разработки, например: проверка кода на соответствие стандартам(PEP8 style guide [9]), поиск ошибок в коде и поиск дубликатов, который сильно помогает при рефакторинге. Pylint готов к работе "из коробки", но может быть полностью сконфигурирован пользователем под его нужды.

Муру Муру [8] — утилита для статической проверки типов. Подразумевается, что код будет проаннотирован в соответствии с синтаксисом Python 3 (PEP 484 нотация).

```
def fib(n: int) -> Iterator[int]:
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

Утилита создана для совмещения плюсов динамической и статической типизации. Проверка типов осуществляется во время компиляции.

Стоит отметить, что в IDE PyCharm уже была реализована функциональность по сбору статистики о типах методов [4]. Собранная информация используется для автодополнения и генерации документации.

Для Ruby тоже существует большое количество инструментов, призванных облегчить работу программиста. Вот некоторые инструменты для статического анализа программы на языке Ruby: RuboCop[1], Reek[18], Brakeman, Ruby-lint, Diamondback Ruby[7],

Rubocop Система для статического анализа кода и проверки выполнения соглашений об оформлении кода с возможностью автоматического исправления ошибок. Несоответствие кода этим соглашениям часто приводит к возникновению ошибок, однако Rubocop не способен обнаружить фактические ошибки, связанные с несоответствием типов даже в самых простых программах.

```
x = ''123''
p x.length, x.downcase

x = {:a => '1', :b => '2', :c => '3'}
p x.length, x.downcase
```

```
=>
Inspecting 1 file
C

Offenses:

rubocop_fails.rb:1:5: C: Prefer single-quoted strings when you don't
                         need string interpolation or special symbols.
x = ''123''
    ^^^^^
rubocop_fails.rb:4:5: C: Space inside { missing.
x = {:a => '1', :b => '2', :c => '3'}
    ^
rubocop_fails.rb:4:6: C: Use the new Ruby 1.9 hash syntax.
x = {:a => '1', :b => '2', :c => '3'}
     ^^^^^
rubocop_fails.rb:4:17: C: Use the new Ruby 1.9 hash syntax.
x = {:a => '1', :b => '2', :c => '3'}
                ^^^^^
rubocop_fails.rb:4:28: C: Use the new Ruby 1.9 hash syntax.
x = {:a => '1', :b => '2', :c => '3'}
                           ^^^^^
rubocop_fails.rb:4:37: C: Space inside } missing.
x = {:a => '1', :b => '2', :c => '3'}
```

Видно, что Rubocop не смог детектировать, что для хэш-таблицы x не реализован метод `downcase`.

Reek Система для обнаружения 'дурно пахнущего' (code smells) кода — кода с признаками проблем в системе или архитектуре приложения. Также система предлагает возможные их решения путём рефакторинга.

Brakeman Система для обнаружения уязвимостей в системе безопасности (SQL-инъекциям, Межсайтовому скриптингу и др.) для приложений, написанных на Ruby on Rails.

Ruby-lint Инструмент для детектирования синтаксических ошибок, таких как необъявленные переменные, неправильный набор аргументов для вызова метода или недостижимые участки кода.

Diamondback Ruby Альтернативная платформа для исполнения программ на Ruby, которая привносит преимущества статической типизации без ущерба выразительности языка. Основными особенностями DRuby являются: вывод типов и типовые аннотации.

- Вывод типов: для каждого объекта в программе выводится тип и информируется о любом несоответствии:

```
class A;
    def f();
    end
end
A.new.g# there is no g in A
#...
[ERROR] instance A does not support methods g
  in typing method call g
  at ./method.rb:5
  in typing ::A.new
  at ./method.rb:5
```

- Типовые аннотации: методу можно сопоставить явную типовую аннотацию.

```
class String
  ...
  ##% ''+'' : (String) -> String
  def +(p0); end
  ##% insert : (Fixnum, String) -> String
  def insert(p0, p1); end
  ...
end
```

В официальной документации Diamondback Ruby отмечено, что в настоящий момент система не способна проаннотировать полностью даже стандартные библиотеки.

Ни одно из перечисленных решений не реализует достаточно корректную систему выделения типов и анализа кода. Такая система разрабатывается в IDE RubyMine, но и она не позволяет обрабатывать все конструкции языка. Наиболее часто встречающиеся конструкции, которые не могут быть проанализированы статически без частичной интерпретации исходного кода — это динамическая декларация методов и исполнение Ruby кода, переданного в виде строки (eval[1]).

```
# rails/activerecord/lib/active_record/runtime_registry.rb
[:connection_handler, :sql_runtime].each do |val|
    class_eval %{ def self.#{val}; instance.#{val}; end }, __FILE__, __LINE__
    class_eval %{ def self.#{val}=(x); instance.#{val}=x; end }, __FILE__, __LINE__
end
```

Выбранный подход подразумевает сохранение информации о методах в момент работы программы и последующий её анализ. Анализ статистики запусков позволяет строить предположения о поведении метода при разных наборах входных данных. Это позволяет показывать пользователю список типов аргументов, которые он может передать в метод, чтобы его работа завершилась корректно и статически детектировать довольно широкий класс ошибок.

[1]https://apidock.com/ruby/Kernel/eval

3. Архитектура

Приложение имеет модульную архитектуру и состоять из четырёх основных частей.

- Первый модуль — это окружение, которое запускается одновременно с приложением и перехватывает события вызова метода и выхода из него. После чего информация о вызовах методов сериализуется и в виде последовательности JSON-ов пересылается на локальный сервер.
- Второй модуль — локальный сервер, обрабатывающий поток данных, полученный из Ruby процесса. На локальном сервере информация о разных вызовах одного и того же метода объединяется в контракт и отправляется на удалённый сервер.
- Третий модуль — это облачное хранилище, которое получает контракты от пользователей. В этом хранилище методы разбиты по наименованиям и версиям библиотек, в которых они декларируются. Пользователь IDE по готовому списку библиотек, которые он хочет использовать получает архив, в котором содержатся типовые аннотации методов из этой библиотеки.
- Четвёртый модуль — это плагин для IDE RubyMine, который использует полученные контракты для анализа кода и других функций IDE.

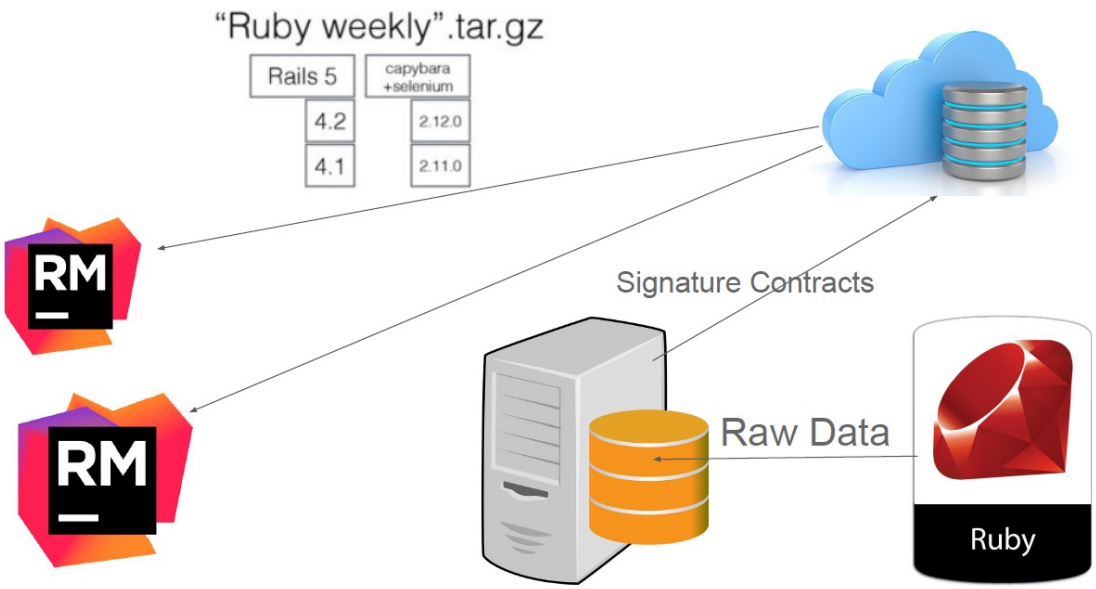

Рис. 1: Схема работы

4. Окружение для запуска программ

4.1. Опциональные аргументы

В языке Ruby существует TracePoint API[14], которое позволяет установить обработчики на ряд системных событий:

- :line — переход исполнения на новую строку;
- :class — начало декларации класса или модуля;
- :end — конец декларации класса или модуля;
- :call — вызов метода;
- :return — выход из метода;
- :thread_begin — старт нового потока;
- :thread_end — завершение потока и тд.

Для получения информации о типах, переданных в метод будет использоваться событие :call, а для получения возвращаемого типа — событие :return.

В процессе анализа кода приходится анализировать вызовы метода с некоторым набором параметров, а значит важно знать, к какому типу относится каждый параметр метода, какие из них обязательны для передачи, а какие нет, в каком порядке передаются аргументы и какие они имеют имена.

Аргументы методов в языке Ruby имеют следующую структуру:

```
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
```

Текущая версия TracePoint API не располагает необходимой информацией об аргументах метода. А именно, он не предоставляет информацию о том, какие опциональные аргументы (opt, keyword) были непосредственно переданы в метод, а каким было присвоено значение по умолчанию. Можно было бы изменить стандартное API, но тогда это изменение войдёт только в последующие версии интерпретатора и виртуальной машины, а приложение должно работать и с предыдущими версиями Ruby. Поэтому для получения этой информации необходимо создать расширение для виртуальной машины YARV (Yet another Ruby VM)[15].

Каждый блок исполнения [16] имеет ссылку на верхушку стека данных и ссылку на исполняемую в данный момент инструкцию (рис. 2). Для получения аргументов,

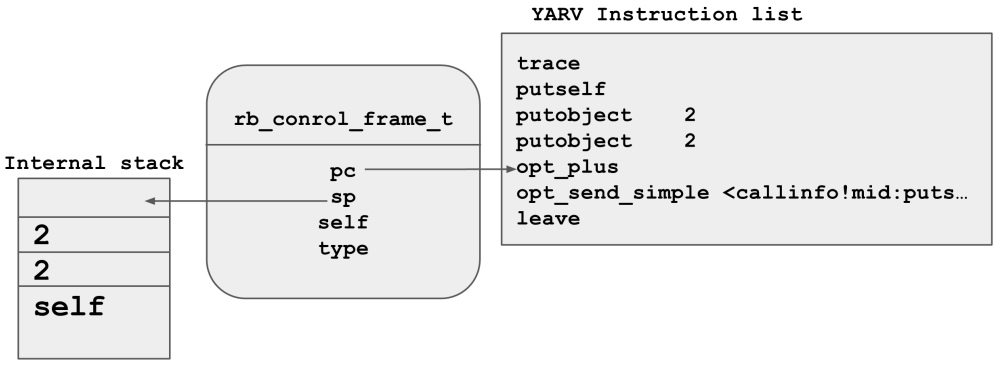

Рис. 2: YARV — виртуальная машина с двумя стеками.

непосредственно переданных в метод необходимо найти кадр управления, предшествующий кадру, отвечающему за исполнение соответствующего метода. В этом кадре pc указывает на инструкцию, идущую после интересующего нас вызова метода. В инструкции, соответствующей вызову исследуемого метода присутствует поле, описывающее количество переданных аргументов (argc) и список переданных keyword аргументов. Именно эти поля использует интерпретатор при передаче переменных в метод и сопоставлении их с аргументами. Кроме того по списку байт-код инструкций, приведенному ниже можно заметить, что инструкции, проставляющие значения по умолчанию в не проинициализированные аргументы находятся перед инструкцией 'trace', которая и вызывает соответствующее событие :call в TracePoint API, поэтому от его использования и пришлось отказаться.

```
def foo(a, b=42, kw1: 1, kw2:, kw3: 3)
    #...
end
foo(1, kw1: '1', kw2: '2')

== disasm: #<ISeq:<compiled>@<compiled>>================================
...
0020 opt_send_without_block <callinfo!mid:foo, argc:3, kw:[kw1,kw2], FCALL|KWARG>, <callcache>
0023 leave #<- PC
== disasm: #<ISeq:foo@<compiled>>=======================================
local table (size: 7, argc: 1 [opts: 1, rest: -1, post: 0, block: -1, kw: 3@1, kwrest: -1])
[ 7] a<Arg>     [ 6] b<Opt=0>   [ 5] kw2        [ 4] kw1        [ 3] kw3        [ 2] ?
0000 putobject        42                                            (   1)
0002 setlocal_OP__WC__0 6
0004 trace            8
0006 putnil
0007 trace            16                                            (   3)
0009 leave
```

Видно, что байт-код инструкция 0020 хранит информацию о количестве переданных аргументов (argc:3) и наборе именованных параметров (kw:[kw1,kw2]). Нам осталось только найти кадр, из которого был вызван интересующий нас метод, найти в нём байт-код инструкцию, осуществившую вызов и получить всю необходимую информацию. Данные об имени метода, названии и версии библиотеки, в которой он был декларирован, а также точный путь к файлу и номер строки декларации могут быть получены из стандартного API.

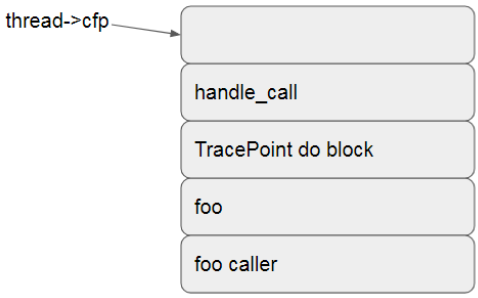

Рис. 3: Кадры управления

Вызов события :call происходит во время начала исполнения соответствующего метода, поэтому на момент поиска необходимого нам кадра, структура верхних кадров стека всегда одна и та же (рис. 3). После кадра, отвечающего за работу метода, в том же самом потоке появляется несколько кадров, порождённых нашим окружением для запуска. Поэтому необходимо определённое количество раз перейти по ссылкам на предыдущий кадр и получить кадр, из которого был вызван наш метод.

О каждом вызове метода сохраняется следующая информация:

- имя метода,
- количество переданных аргументов,
- список именованных параметров, переданных в метод,
- тип объекта, у которого был вызван метод,
- информация о сигнатуре метода (имена, типы параметров и типы аргументов),
- выходной тип метода,
- имя гема (библиотеки) в котором был задекларирован метод,
- версия гема,
- видимость метода,
- путь к файлу, где был задекларирован метод,
- номер строки, где был задекларирован метод.

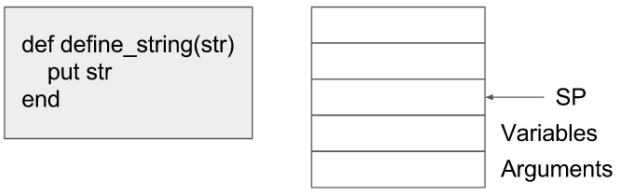

Рис. 4: Содержимое стека.

4.2. Ускорение сбора данных

Для сбора статистики очень важно время работы, чтобы не заставлять пользователя ждать завершения исполнения скрипта со сбором данных во много раз дольше, чем при обычном исполнении. Оказалось, что API языка Ruby, которое предоставляет информацию о сигнатуре метода и типах аргументов работает довольно медленно, так как оно вычисляет и сохраняет полный контекст исполнения, из которого нами будет использована лишь малая часть, а именно входные и выходные типы для методов. Поэтому типы аргументов мы также будем получать непосредственно из виртуальной машины. На рис. 2 показано, что в структуре `rb_control_frame_t` хранится указатель на верхушку стека данных. На момент вызова метода в этом стеке лежат только аргументы метода, поэтому нам достаточно вручную пройти по стеку данных и вычислить типы переменных (рис. 4).

Для каждого параметра необходимо определить, к какому типу он относится. Кадр управления, отвечающий за выполнение каждого метода содержит флаги и переменные, соответствующие наличию и количеству аргументов каждого типа, по которым однозначно можно вычислить тип, к которому относится каждый параметр:

```
//ruby/vm_core.h
lead_num     = M
opt_num      = N
rest_start   = M+N
post_start   = M+N+(*1)
post_num     = O
keyword_num  = K
block_start  = M+N+(*1)+O+K
keyword_bits = M+N+(*1)+O+K+(&1)
size         = M+N+O+(*1)+K+(&1)+(**1) // parameter size.
```

Знать сигнатуру метода важно для того, чтобы при статической проверке кода выявлять некорректные вызовы методов, тела которых генерируются динамически, и поэтому они не могут быть проанализированы статически.

Ниже приведено время работы bencmark тестов приложения Puppet[1] при обычном исполнении, исполнении со сбором статистики при использовании обычного API и самописного расширения для получения типов аргументов. Видно, что при использовании стандартного API исполнение замедляется примерно в 60 раз, а при использовании самописного в 16. Значит произведена оптимизация в 3.75 раз.

Обычное исполнение:

	user	system	total	real
Run 1	1.040000	0.010000	1.050000	1.065770
Run 2	1.010000	0.010000	1.020000	1.018101
Run 3	1.010000	0.030000	1.040000	1.030037
Run 4	1.000000	0.010000	1.010000	1.017340
Run 5	1.000000	0.030000	1.030000	1.022257
Run 6	1.000000	0.020000	1.020000	1.028633
Run 7	0.980000	0.010000	0.990000	0.986009
Run 8	1.060000	0.020000	1.080000	1.079832
Run 9	0.950000	0.020000	0.970000	0.970303
Run 10	0.950000	0.000000	0.950000	0.948032
total:	10.000000	0.160000	10.160000	10.166314
avg:	1.000000	0.016000	1.016000	1.016631

Исполнение со сбором статистики при использовании стандартного API:

user	system	total	real	
Run 1	57.970000	0.050000	58.020000	58.087346
Run 2	51.900000	0.010000	51.910000	51.925019

Исполнение со сбором статистики при использовании самописного расширения:

	user	system	total	real
Run 1	16.590000	0.040000	16.630000	16.630727
Run 2	14.850000	0.030000	14.880000	14.870517
Run 3	15.500000	0.000000	15.500000	15.518889
Run 4	15.740000	0.020000	15.760000	15.752735
Run 5	15.900000	0.020000	15.920000	15.922776
Run 6	19.960000	0.040000	20.000000	20.003364
Run 7	15.400000	0.040000	15.440000	15.438530
Run 8	15.490000	0.010000	15.500000	15.501521
Run 9	16.460000	0.020000	16.480000	16.484746
Run 10	15.750000	0.030000	15.780000	15.782393
total:	161.640000	0.250000	161.890000	161.906198
avg:	16.164000	0.025000	16.189000	16.190620

[1]https://puppet.com/

5. Вывод контрактов

Большое количество сырых сигнатур, полученных на первом этапе необходимо структурировать, чтобы их можно было легко использовать и сохранять. Каждому исследуемому методу сопоставляется конечный автомат с выделенной стартовой и одной терминальной вершиной. Если какой-то из опциональных аргументов не был передан, то в качестве его типа указывается специальный символ, который не совпадает ни с одним типом. В автомат последовательно жадно добавляются слова, полученные конкатенацией входных и выходных типов. То есть, если при добавлении очередного ребра из текущей вершины уже есть переход с таким же символом, то ребро не добавляется и мы переходим по существующему ребру. Ребро, соответствующее выходному типу метода всегда ведёт в выделенную терминальную вершину.

Исходные параметры: callArgs, returnType, automaton, parameterList
Результат: automaton
$signature \leftarrow emptyList$
for $param : parameterList$ **do**
  if $\exists arg : arg \in callArgs \&\& arg \mapsto param$ **then**
    $signature << arg$
  else
    $signature << \epsilon$
  end
end
$node \leftarrow automaton.startVertex$
for $arg : signature$ **do**
  $type \leftarrow arg.type$
  if $(node, type) \notin automaton$ **then**
    $automaton(node, type) \leftarrow newNode$
  end
  $node \leftarrow automaton(node, type)$
end
$automaton(node, returnType) \leftarrow automaton.termVertex$

Алгоритм 1: Добавление сигнатуры к автомату

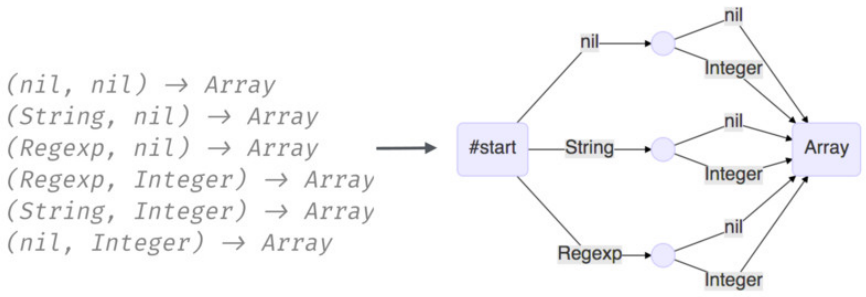

Рис. 5: Генерация не минимизированного автомата.

Затем к автомату применяется алгоритм минимизации [6], но он немного модифицируется для автоматов такого типа(алг. 2). Заметим, что все сигнатуры, добавляемые в автомат имели одинаковую длину, значит полученный автомат имеет слоистую структуру относительно удалённости от стартовой вершины. И все рёбра, ведущие из вершин i-го уровня ведут в вершины i+1-го уровня. Докажем, что после добавления сигнатуры к минимизированному автомату каждая добавленная вершина может быть объединена только с вершиной своего уровня (рис. 6).

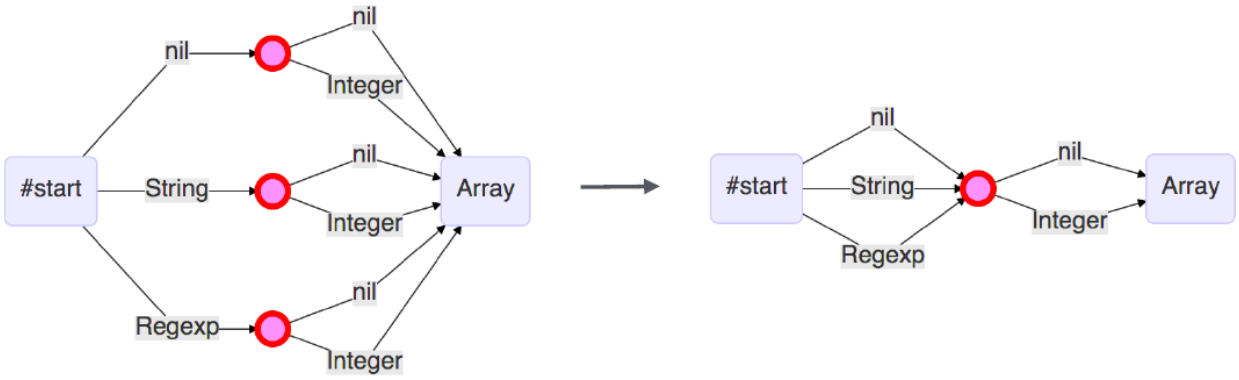

Рис. 6: Объединение вершин

***Утверждение* 1.** *При минимизации объединяются только вершины одного уровня.*

Доказательство. Рассмотрим две вершины a и b с уровней i и j соответственно($i \neq j$). Вершины a и b объединяются в случае, если функции переходов для вершин a и b совпадают. По построению автомата все переходы из вершин уровня i ведут в вершины уровня $i + 1$, значит переходы из a ведут в вершины уровня $i + 1$, а переходы из вершины b ведут в вершины уровня $j + 1$. Из того, что вершины, смежные с a, и вершины, смежные с b лежат на разных уровнях следует, что функции переходов для вершин a и b не совпадают. □

***Следствие* 1.** *Пусть n — высота автомата, тогда время работы алгоритма минимизации после добавления одной сигнатуры к заранее минимизированному автомату можно оценить как: $O(\sum_{i=1}^{n} automaton.levels[i])$ или $O(automaton.size)$, а не $O(automaton.size * n)$, как для автомата в общем случае.*

Важно, что минимизация сохраняет набор слов, который принимается автоматом.

Исходные параметры: automaton, signatureNodes
Результат: automaton
$levels \leftarrow automatron.levels$//разбиение автомата на слои
for $node : signatureNodes,\ i{+}{+}$ **do**
  for $nodeForComparison : levels[i]$ **do**
    if $node.getTransitions = nodeForComparison.getTransitions$ **then**
      automatron.joinNodes(node, nodeForComparison);
    end
  end
end

Алгоритм 2: Минимизация автомата

Когда во время анализа кода понадобится вычислить выходной тип метода и при этом типы всех аргументов будут вычислены, достаточно будет прочитать в автомате слово, полученное конкатенацией входных типов, и все рёбра ведущие из вершины, в которую мы попадём и будут соответствовать выходным типам.

Довольно часто типы двух или более аргументов всегда совпадают, в этом случае автомат можно ещё сильнее минимизировать. Рассмотрим эту оптимизацию на примере метода `equals`. При корректном завершении работы метода типы аргументов совпадают.

```
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
...
```

Во время добавления к автомату очередного перехода из вершины сравним тип, переход по которому мы хотим добавить, со всеми предыдущими типами добавляемой сигнатуры. В случае, если найдётся хотя бы одно совпадение, вместо обычного ребра с типом добавим ребро с маской, длина которой равна порядковому номеру добавляемого типа, а каждая единица соответствует предыдущему типу в сигнатуре, который совпал с типом, который мы хотели добавить(рис. 7). При чтении сигнатуры очередной тип аналогично нужно будет сравнить с предыдущими типами сигнатуры и перейти по соответствующему ребру при наличие перехода по полученной маске.

Докажем, что после минимизации автомат с переходами-масками останется детерменированным, то есть не может быть такого, что из некоторой вершины может быть осуществлён переход и по обычному ребру и по ребру с маской одновременно.

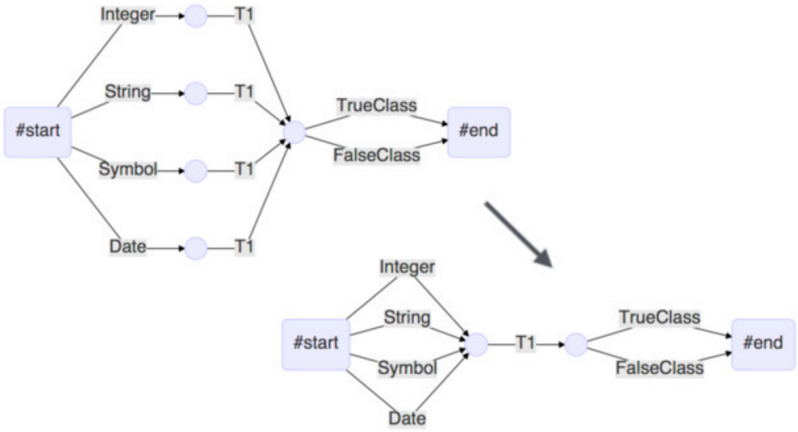

Рис. 7: Автомат с подсчитанными масками

***Утверждение* 2.** *После минимизации автомат остаётся детерминированным.*

Доказательство. Пусть после минимизации автомата появилась вершина v, такая что при чтении автоматом очередного типа s из некоторой сигнатуры sign из v может быть осуществлён переход по ребру с типом s и ребру с некоторой маской m. Тогда до минимизации в автомате присутствовали две вершины v1 и v2, которые были объединены в v. Значит чтение sign в автомате до минимизации проходило либо через вершину v1, либо через вершину v2 (не умаляя общности через v1). У вершины v есть переходы по s и mask, значит они были у v1. Тогда при добавлении перехода по s к вершине v1 была получена маска m, тогда вместо добавления обычного ребра должно было быть добавлено ребро с маской. Значит у v1 не могло быть переходов s и m одновременно. □

В Ruby довольно активно используется Duck Typing [3]. В следствие этого в качестве аргумента в метод могут быть переданы переменные самых разных типов, реализующих некоторый набор методов. В следствие этого в автомате появится множество кратных рёбер, соответствующих этим классам. Эти кратные рёбра могут быть заменены одним ребром, содержащем информацию об интерфейсе, которому удовлетворяют все эти классы. В случае, если этот общий интерфейс пуст на ребре достаточно написать тип `Object`, так как он является родительским классом для всех объектов. Тогда для перехода по этому ребру очередной тип из сигнатуры должен реализовывать этот интерфейс.

По постренному автомату легко может быть построено регулярное выражение, описывающее сигнатуру метода — это и будет контрактом. Например автомату из рис. 7 соответствует выражение
`(Integer|String|Symbol|Date;T#1)->(TrueClass|FalseClass)`.

6. Облачное хранилище для типовых контрактов

Контракты, сгенерированные на локальном сервере, отправляются на удалённый сервер в виде сериализованного списка рёбер. На сервере с определенной периодичность генерируются пакеты для каждой библиотеки, составленные из контрактов, соответствующих методам из этой библиотеки. Создание облачного хранилища обусловлено тем, что разные пользователи могут использовать одни и те же методы поразному, поэтому необходимо собирать информацию о вызовах разных пользователей в одном месте. Сбор данных от пользователей позволяет увеличить количество проаннотированых методов и покрытие функциональности каждого метода, что делает соответствующие контракты более репрезентативными.

На основе Gemfile-a, в котором содержатся названия всех используемых в приложении библиотек с указанием версий, пользователь получает набор пакетов для всех необходимых библиотек. С целью минимизации затрат на пересылку данных от пользователя на сервер достаточно отправлять только новые вершины и рёбра автомата, образовавшиеся за время работы пользователя. На сервере эта разница добавляется к предыдущей версии автомата. В процессе приближения контракта репрезентативному объём принимаемых от пользователя данных будет падать. Стоит отметить, что пользователь может локально работать с набором своих контрактов, не отправляя их на сервер, в случае, если он работает с проприетарным приложением.

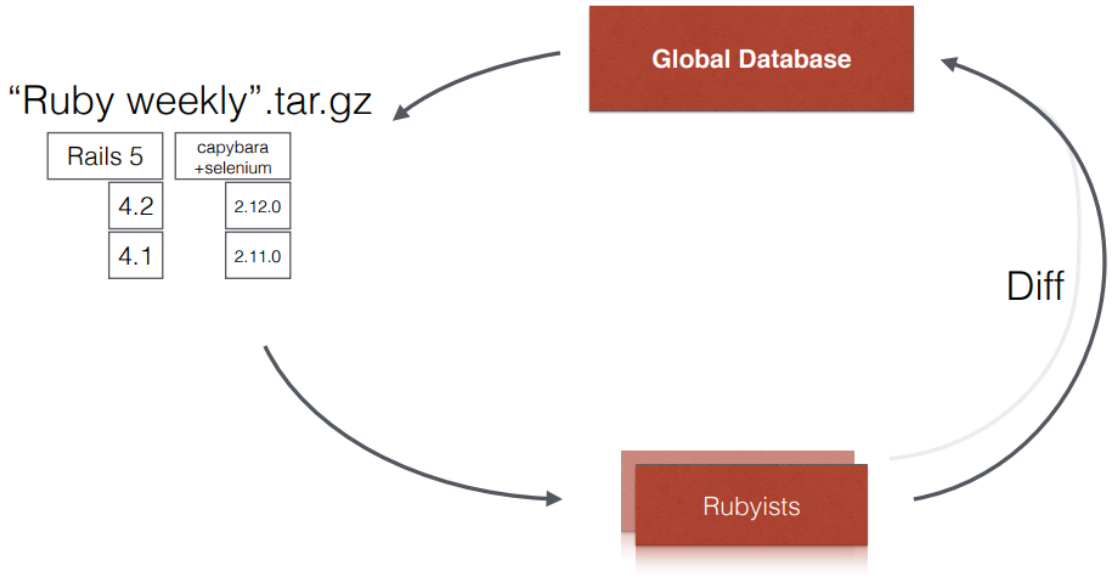

Рис. 8: Схема пополнения облачного хранилища

7. Встраивание контрактов в алгоритм выделения типов

Контракт используется для вычисления типа, возвращаемого при вызове метода с некоторым набором аргументов. Стоит отметить, что типы аргументов не всегда однозначно определены, иногда некоторое множество типов, к которому может относиться переменная. Для вычисления типа, возвращаемого методом необходимо последовательно переходить по рёбрам автомата, поддерживая множество вершин, в которые мы могли бы попасть при некоторой последовательности типов. На места опциональных аргументов, которые не передаются проставлен специальный неалфавитный символ, то есть длина сигнатуры на один отличается от рёберной высоты автомата.

Исходные параметры: argumentUnionTypes, automaton
Результат: returnType
$nodes \leftarrow automaton.startVertex$
for $unionType : argumentUnionTypes$ **do**
  $nextLayer \leftarrow emptySet$
  for $node : nodes$ **do**
    for $type : unionType$ **do**
      if $(node, type) \in automaton$ **then**
        $nextLayer << automaton(node, type)$
      end
    end
  end
  $nodes \leftarrow nextLayer$
end
for $node : nodes$ **do**
  $returnType << node.getTransitions.types$
end

Алгоритм 3: Вычисление выходного типа

Сгенерированные контракты дополняют систему выделения типов, ведь они позволяют вычислять типы, возвращаемые из методов, которые не получалось проанализировать стандартными средствами. Засчёт этого расширяется класс переменных, для которых возможно статически вычислить тип.

Собранная информация о методах позволяет значительно ускорить существующий анализ потока управления [17], ведь методы, для которых сгенерирован достаточно репрезентативный контракт не нуждаются в дополнительном анализе. Контракты позволяют расширить применимость некоторых функций, поддерживаемых в большинстве современных IDE. Рассмотренные функции применимы к вызовам методов, для которых удалось выделить класс объекта, к которому они были применены и для этого класса существует контракт, соответствующий методу с таким именем и

конфигурацией параметров. Функции, в которых применяются контракты:

Go To Declaration/Find Usages В момент исполнения метода была сохранена информация о месте его декларации. Эти данные могут быть использованы для навигации от вызова метода к его реализации и наоборот. В случае, если метод добавлен динамически или в дереве разбора отсутствует вершина, соответствующая объявлению метода — создаётся синтетических символ, положение которого определяется путём к файлу и номером строки, полученными из контракта.

Autocompletion Список методов, реализованных для объекта класса может быть дополнен методами, для которых нашёлся контракт.

'Incorrect method arguments' Inspection Информация о конфигурации параметров метода использована для обнаружения некорректных вызовов.

Method tooltip По контракту может быть сгенерировано регулярное выражение, которое является аннотацией для метода.

8. Тестирование и апробация

Для тестирование функциональности модуля была использована инспекция IDE под название Unresolved Ruby Reference, которая отображает участки кода, которые не удалось обработать (в частности, вызовы методов, для которых не удалось найти реализацию). Для примера была проанализирована директория `/spec/controllers` приложения discourse [2] — популярной платформы, реализованной на Ruby On Rails. При анализе без сгенерированных контрактов было найдено 1084 предупреждения в 72-х файлах. После запуска этого набора тестов и генерации набора автоматов количество предупреждений упало до 429. Большинство из них связано с подключением файлов, которые не удаётся найти, использованием классов, декларация которых не найдена и вызовом методов у объектов, тип которых не удалось выделить.

В файле `spec/controllers/admin/backups_controller_spec.rb.rb` все предупреждения были устранены. Например, устранены все предупреждения о том, что для класса `SiteSettings < Class:ActiveRecord::Base` не реализован метод `disable_emails`, который добавляется динамически. Но в процессе исполнения тестов был сгенерирован контракт `disable_emails: ()->(TrueClass|FalseClass)` для родительского класса `ActiveRecord::Base`, благодаря которому и была устранена проблема.

```
describe ".restore" do

  it "starts a restore" do
    expect(SiteSetting.disable_emails).to eq(false)
                                          :restore!).with( *expected_para
    xhr :post, :restore, id: backup_filename, client_id: "foo"

    expect(SiteSetting.disable_emails).to eq(true)
    expect(response).to be_success
  end

end
```

Cannot find 'disable_emails' for type 'SiteSetting' more... (Ctrl+F1)

Рис. 9: Unresolved Reference

Алгоритм генерации контрактов был протестирован на примере библиотеки activerecord 4.2.7.1[20]. Для сбора статистики были выбраны три набора тестов из discourse. Наборы тестов: /spec/jobs (210 тестов), /spec/models (1583 теста) и /spec/controllers (1202 теста). В столбце ni отображено количество вызовов метода после i-го набора тестов, в столбце si — размер автомата после исполнения i-го набора тестов. Из таблицы, приведённой ниже, видно, что рост автомата замедляется по мере его приближения к достаточной репрезентативности. Новые сигнатуры всё чаще принимаются автоматом и в этом случае он не нуждается в модификации.

Название класса (ActiveRecord::)	Название метода	n1	s1	n2	s2	n3	s3
SpawnMethods	merge	120	42	440	79	688	81
SpawnMethods	spawn	112	2	411	2	639	2
Relation	initialize_copy	112	3	411	3	639	3
Relation	reset	112	2	411	2	639	2
Relation::Merger	initialize	94	67	362	127	568	134
SpawnMethods	merge!	93	65	362	122	568	128
ModelSchema::ClassMethods	table_name=	85	3	268	3	431	3
Core::ClassMethods	relation	78	2	261	2	401	2
Scoping::Named::ClassMethods	all	78	2	269	2	414	2
Relation::Merger	merge	76	2	300	2	87	2
Scoping::Named::ClassMethods	default_scoped	76	2	259	2	399	2
Scoping::ScopeRegistry	set_value_for	62	6	218	6	332	6
Scoping::Default::ClassMethods	unscoped	60	2	247	2	378	2
Scoping::ClassMethods	current_scope=	60	3	213	3	323	3
Querying	where	60	4	291	4	459	4
QueryMethods	limit!	59	4	234	4	356	4
QueryMethods	limit	59	4	234	4	356	4
Associations::AssociationScope	add_constraints	58	36	273	56	431	68
Associations::AssociationScope	last_chain_scope	58	60	272	109	428	110
Relation	load	47	2	190	2	297	2
Associations::AssociationScope	bind	44	13	181	13	290	13
Associations::Association	scope	43	2	172	2	268	2
QueryMethods	where	39	6	166	6	265	6
Associations::Association	target_scope	38	2	156	2	245	2
Reflection::AssociationReflection	initialize	37	7	143	7	217	7
SpawnMethods	relation_with	37	3	108	3	152	3
Associations::AssociationScope	scope	37	10	166	10	264	10
Associations::Association	association_scope	36	2	150	2	234	2
Associations::Association	inverse_reflection_for	31	6	130	6	196	6
Type::Value	changed?	31	30	112	36	181	37
Core	init_with	30	3	148	3	226	3
Core::ClassMethods	allocate	30	2	148	2	226	2
Attribute	initialize	29	13	109	17	175	17
Core::ClassMethods	find_by	28	3	137	3	209	3
FinderMethods	apply_join_dependency	27	12	96	24	138	24
Associations::Association	set_inverse_instance	27	3	109	3	162	3
Relation	scoping	25	2	96	2	149	2

9. Заключение

В ходе данной работы получены следующие результаты.

1. Разработана архитектура.
2. На языке Ruby/C реализовано окружение для запуска программ, собирающее данные о типах.
3. На языке Java разработан модуль для вывода контрактов, описывающих сигнатуру метода.
4. Создано облачное хранилище для сгенерированных контрактов.
5. Реализован плагин для IDE, встраивающий контракты в систему статического анализа.
6. Модуль протестирован и проведено нагрузочное тестирование.

В дальнейшем планируется активно использовать разработанный модуль в IDE RubyMine и поддерживать базу аннотаций для популярных библиотек.

Список литературы

[1] Batsov Bozhidar. RuboCop. — 2017. — URL: batsov.com/rubocop (online; accessed: 24.04.2017).

[2] Discource. — URL: https://github.com/discourse/discourse (online; accessed: 12.05.2017).

[3] Duck Typing. — URL: http://rubylearning.com/satishtalim/duck_typing.html (online; accessed: 12.05.2017).

[4] Dynamic runtime type inference in PyCharm 2.7. — URL: https://blog.jetbrains.com/pycharm/2013/02/dynamic-runtime-type-inference-in-pycharm-2-7/ (online; accessed: 07.05.2017).

[5] Flow Documentation. — URL: https://flow.org/en/docs/ (online; accessed: 07.05.2017).

[6] John E. Hopcroft Rajeev Motwani Jeffrey D. Ullman. Introduction to Automata Theory. — Addison-Wesley, 2001.

[7] Mike Furr David An Jeff Foster Mike Hicks. Diamondback Ruby Guide. — 2009. — URL: http://www.cs.umd.edu/projects/PL/druby/manual/manual.pdf (online; accessed: 24.04.2017).

[8] Mypy documentation. — URL: http://mypy.readthedocs.io/en/latest/ (online; accessed: 07.05.2017).

[9] PEP 8 – Style Guide for Python Code. — URL: https://www.python.org/dev/peps/pep-0008/ (online; accessed: 07.05.2017).

[10] RDoc - Documentation from Ruby Source Files. — URL: http://rdoc.sourceforge.net/ (online; accessed: 07.05.2017).

[11] Ralf Hinze Johan Jeuring, LÃndres.

[12] Ren Brianna M. The Ruby Type Checker. — URL: http://www.cs.umd.edu/~jfoster/papers/oops13.pdf (online; accessed: 21.12.2016).

[13] Review of Python Static Analysis Tools. — URL: https://blog.codacy.com/review-of-python-static-analysis-tools-ff8e7e27f972 (online; accessed: 07.05.2017).

[14] Ruby Documentation - TracePoint. — URL: http://ruby-doc.org/core-2.0.0/TracePoint.html (online; accessed: 12.05.2017).

[15] Sasada Koichi. YARV: Yet Another Ruby VM. — 2006. — URL: http://www.atdot.net/yarv/ (online; accessed: 24.04.2017).

[16] Shaughnessy Pat. Ruby Under a Microscope. — No Starch Press, 2013.

[17] Shivers O. Control flow analysis in scheme. — ACM SIGPLAN 1988 conference on Programming language design and implementation, 1988.

[18] Szotkowski Piotr. How to Find Ruby Code Smells with Reek. — 2017. — URL: batsov.com/rubocop (online; accessed: 26.02.2017).

[19] YARD. A Ruby Documentation Tool. — URL: http://yardoc.org/ (online; accessed: 07.05.2017).

[20] activerecord 4.2.7.1. — URL: https://rubygems.org/gems/activerecord/versions/4.2.7.1 (online; accessed: 12.05.2017).

[21] blog.codeclimate. Gradual Type Checking for Ruby. — 2014. — URL: blog.codeclimate.com/blog/2014/05/06/gradual-type-checking-for-ruby (online; accessed: 21.12.2016).