Генерация сигнатур для языка программирования Ruby

Вьюгинов Н. Ю., 4 курс кафедры системного программирования СПбГУ, viuginov.nickolay@gmail.com
Кириленко Я. А., ст. преп. кафедры системного программирования СПбГУ, jake@math.spbu.ru

Аннотация

Ruby - красивый, искусный язык, в то же время он удобен и практичен. Обратной стороной такого удобства языка является сложность отладки и поиска ошибок в больших кодовых базах. Анализ кода на Ruby значительно усложняется засчёт некоторых возможностей языка[5]. Даже в динамически типизированных языках программирования информация очень полезна, ведь она позволяет реализовать большее количество статических проверок.

В статье описывается новый подход к генерации типовых аннотаций. Предлагается отслеживать непосредственные вызовы метода во время исполнения и генерировать на основе входных и выходных типов контракты, описывающие сигнатуру метода.

Введение

Разработчики, работающие с большими кодовыми базами, тратят довольно много времени на то, чтобы понять, почему тот или иной фрагмент кода не работает так, как запланированно. Динамическая природа языка Ruby позволяет решать некоторые задачи быстро и ёмко, но у этой простоты есть недостаток: кодовая база в целом становится запутанной и сложной для изучения в сравнении с некотроыми другими языками, такими как Java или C++. Например в Ruby довольно часто используют методы, имена и тела которых вычисляются только в процессе исполнения программы [1].

Документационные утилиты, такие как RDoc [6]/YARD [7] могут решить эту проблему, но и у них есть ряд недочётов:

- Система типов, используемая при документации в целом корректна, но она не позволяет указать взаимосвязь между типами. Например метод `def []=` для массивов возвращает объект того же

типа, что и второй аргумент, но в аннотации YARD это будет выглядеть как `@param value [Object], @return [Object]` .

- Полное документирование кода противоречит самой природе языка Ruby: естественности и выразительности.

Поставленную задачу можно решать таким же способом, каким программисты обычно отлаживают свои программы: запустить сам скрипт и получить всю интересующую информацию непосредственно в момент работы программы. Выбранный подход предполагает запуск скрипта и сбор входных типов, переданых в метод при каждом его вызове, и соответствующих им выходных типов. Все собранные данные обрабатываются и на их основе строятся неявные типовые аннотации. Так как этот процесс автоматизирован, проанатированными окажутся все методы, до которых дошёл процесс исполнения.

Сгенерированные аннотации используются не только для составления YARD аннотаций, также они могут быть встроены в систему статического анализа [2] IDE, что позволит улучшить статические проверки и рефакторинги.

Реализация проекта может быть разбита на две основные части:

- Первая часть заключается в реализации окружения для запуска Ruby скриптов. Это окружение будет собирать всю необходимую информацию о вызовах методов и выходах из них. Очень важно, чтобы это окружение работало быстро, чтобы пользователю не проходилось ждать завершения исполнения во много раз дольше чем при обычном иполнении. Во многом из-за этого пришлось отказаться от API, реализованного в языке и реализовать расширение, получающее все необходимые данные напрямую из внутреннего стека виртуальной машины.

- Вторая часть заключается в обработке данных, полученных на первом этапе. Все сырые данные, относящиеся к одному методу будут приведены к конечному автомату. Такая схема хранения позволяет быстро редактировать содержимое, она не затратна по памяти и может быть лего приведена к регулярному выражению, оторое легко воспринимается человеком.

Сбор данных о вызовах методов

Аргументы методов в 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
```

TracePoint это API, предоставляемое виртуальной машиной Ruby [3], оно позваляет обрабатывать события, такие как вызов метода и выход из него, и получать любые данные через `binding`, который инкапсулирует весь контекст исполнения(переменные, методы) и хранит его для последующего использования.

```
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"
```

Большим недостатком этого подходя является то, что подсчёт и инкапсуляция контекста исполнения это довольно трудоёмкая задача. Но впоследствие нам понадобятся только типы аргументов. Реализовав расширение для виртуальной машины YARV мы сможем получить названия типов аргументов напрямую из внутреннего стека (Рис 1).

Анализ кода чаще всего работает с инструкциями вызова методов, поэтому важно понять, какие аргументы были переданы в метод пользователем, а каким были присвоены значения по умолчанию. На

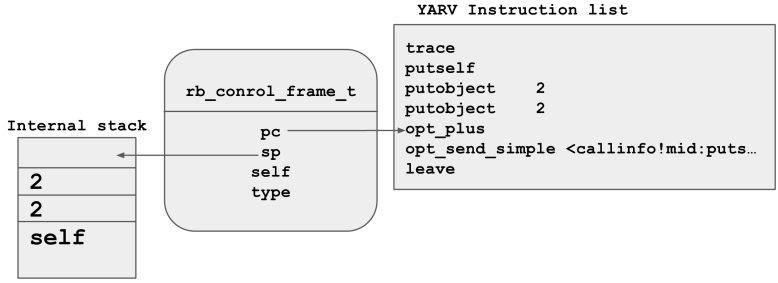

Рис. 1: Внутренние регистрые YARV

момент, когда виртуальная машины вызывает событие вызова метода всем аргументам, которые не были переданы в метод уже проставлены значения. Поэтому было решено реализовать ещё одно расширение, которое будет получать эти данные из внутреннего списка инструкция. Давайте посмотреи на метод с опциональными аргументами и его дизасембляцию:

```
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
```

Видно, что байт-код инструкция 0020 хранит информацию о колличестве переданных аргументов(argc:3) и наборе именованных параметров(kw:[kw1,kw2]). Нам осталось только найти фрэйм, из которого был вызван интересующий нас метод, найти в нём байт-код инструкцию, осуществившую вызов и получить всю необходимую информацию.

Генерация контрактов

Большое колличество 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. В автомат последовательно жадно добавляются слова, полученные склеиванием входных и выходных типов. Затем к автомату применяется алгоритм минимизации [4].

Когда во время анализа кода понадобится вычислить тип, возвращаемый методом, в автомате просто нужно будет прочитать слово, составленное из входных аргументов метода и тогда все переходы из вершины, в которой мы окажемся и будут соответствовать выходным типам.

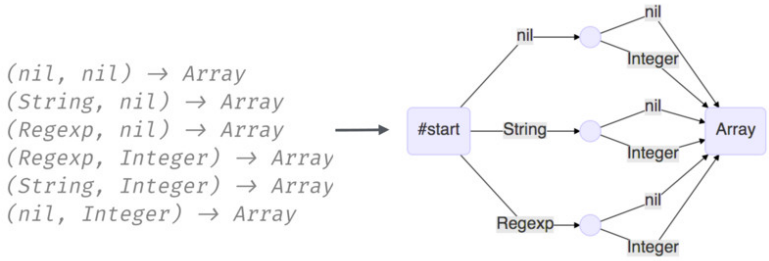

Рис. 2: Генерация неминимизированного автомата

Довольно часто типы двух или более аргументов всегда совпадают, в этом случае автомат можно уменьшить ещё сильнее. Рассмотрим эту оптимизацию на примере метода `equals`:

```
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
...
```

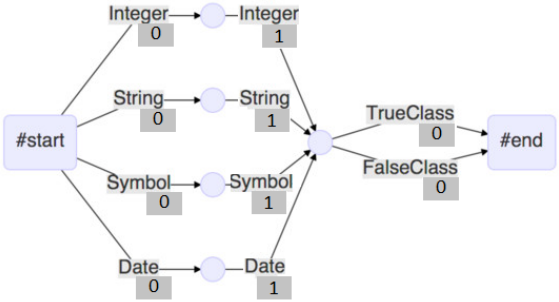

Рис. 3: Автомат с подсчитанными масками

В этом случае при корректном завершении работы метода типы аргументов совпадают. Оптимищация состоит в замене типа, написанного на ребре битовой маской, единица в которой обозначает совпадение типа, написанного на соответствующем ребре с одним из предыдущих типов (Рис 3.). После этого автомат повторно минмимзируется.

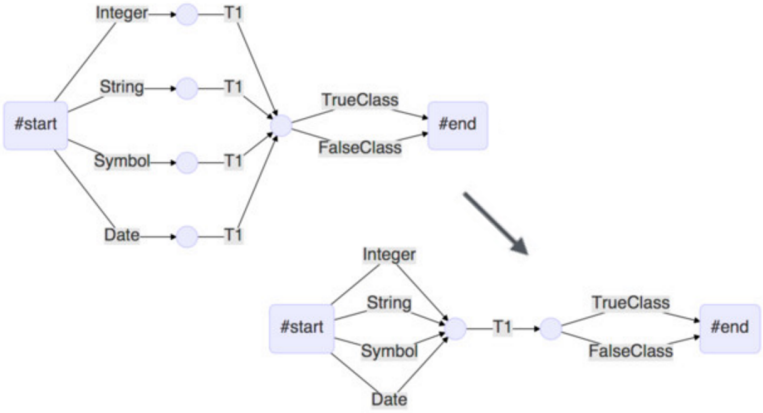

Рис. 4: Минимизация автомата с рёбрами-масками

Заключение

В работе был описан новый подход генерации неявных типовых аннотация. В будующем необходимо будет ещё сильнее соптимизировать время сбора данных о методах, протестировать подход, провести нагрузочное тестирование и встроить сгенерированные контракты в существующий алгоритм статического анализа IDE.

Литература

[1] Gradual Type Checking for Ruby — blog.codeclimate.com/blog/2014/05/06/gradual-type-checking-for-ruby

[2] O. Shivers Control flow analysis in scheme// ACM SIGPLAN 1988 conference on Programming language design and implementation. — https://cs.purdue.edu/homes/suresh/502-Fall2008/papers/shivers-cfa.pdf

[3] Pat Shaughnessy. Ruby Under a Microscope// No Starch Press. — http://patshaughnessy.net/ruby-under-a-microscope

[4] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. Introduction to Automata Theory // Addison-Wesley.

[5] Brianna M. Ren. The Ruby Type Checker — http://cs.umd.edu/ jfoster/papers/oops13.pdf

[6] RDoc - Documentation from Ruby Source Files — rdoc.sourceforge.net

[7] http://yardoc.org