Михаил Калугин
Скопин Игорь Николаевич
May 6, 2009
Управляемый запросами статический анализ для языка Ruby

Введение

Понимание кода больших программных систем является сложной задачей для человека. В особенности сложной эта задача является в случае с динамическими языками. Они намного больше способствуют написанию кода с ошибками, с ними ощутимо затруднено чтение незнакомого кода. Если для статически типизированных языков (например, Java) программа считается действительно большой начиная с примерно 1 миллиона строк, то для динамических (например, Ruby) программа в 100 тысяч строк уже является гигантской и невероятно сложной в поддержке.

Тем не менее динамические языки являются крайне популярными, а во многих областях являются стандартом де-факто (например, веб-приложения). Поэтому решение проблем вызванных динамичностью таких языков является крайне актуальным. Человечество постоянно развивает способы борьбы с ними: юнит-тестирование, подробное документирование, системы статического анализа кода, "умные" интегрированные среды разработки. Именно разработке системы статического анализа с её интеграцией в среду разработки и посвящена данная работа. Для языка Ruby (как, впрочем, и для Python, PHP, и др.) данная проблема до сих пор не является решенной с приемлимым качеством анализа.

Одними из основных причин, делающих анализ для Ruby сложным и отпугивающим, являются:

- динамическая типизация
- не самая простая система типов (duck typing)
- метапрограммирование
- поддержка возможностей функционального программирования

Такая область применения как интегрированные среды разработки накладывает специфические ограничения на статический анализатор: быстрое время отклика на вопросы в условиях постоянно изменяющейся программы, большой размер самой программы. В то же время не предъявляется требований к тому, чтобы анализировалась вся программа целиком.

Практически идеальным подходом к анализу в условиях описанных требований является управляемый запросами анализ с отсечениями (DDP), описанный в работе Спуна[]. Данный

подход оперирует с понятием запроса и позволяет производить лишь те вычисления, которые непосредственно необходимы для ответа на поставленный вопрос. Это позволяет эффективно работать с программи любых размеров. Возможность отсечений позволяет гарантировать то, что пользователь не будет ждать результатов анализа часами.

Полезным свойством такого подхода является то, что явно хранятся зависимости между запросами. Поэтому становится возможным эффективно пересчитывать результаты анализа при изменениях в программе, не выбрасывая все полученные ранее результаты.

Данная работа состоит из двух частей: теоретической и практической. Теоретическая часть предлагает модифицированный алгоритм DDP, позволяющий работать с инкрементальными запросами. Также описывается разработнный автором основанный на DDP статический анализ, позволяющий вычислять типы в языке Ruby. Приводятся оценки точности и времени работы.

Практическая часть представляет собой две реализации описанных алгоритмов. Первая реализация является существенно упрощенной (не поддерживается инкрементальное обновление результатов, некоторые возможности языка). Не смотря на кажущиеся недостатки, она интегрирована в среду разработки DLTK Ruby [2], где успешно используется по всему миру. Вторая реализация соответсвует описанной в теоретической части, и подлежит интеграции в DLTK Ruby взамен первой.

Тестирование обеих реализаций показывает жизнеспособность предлагаемых алгоритмов и соотвествие ожиданиям, а также теоретическим оценкам.

Задача

Язык Ruby

Ruby — это динамический, рефлективный объектно-ориентированный язык общего применения. Ruby был создан в 1993 году в Японии. Действительную популярность язык приобрел в начале 2000х, после появления Ruby on Rails — системы программирования, позволяющей быстро разрабатывать веб-приложения. С того момента язык является крайне популярным, а Ruby on Rails, наверное, входит в тройку самых популярных решений для веб-приложений вместе с J2EE и PHP. Такая популярность и область применения открывают огромный спрос на “умные” развитые среды разработки, позволившие бы схожую с Java простоту редактирования кода.

Ruby поддерживает несколько парадигм программирования: функциональное, объектно-ориентированное, императивное и рефлективное. Также он имеет динамическую типизацию и

автоматическое управление памятью. В настоящий момент спецификации Ruby не сущесвует, и язык полностью определяется существующей реализацией интерпретатора.

Посколько читатель может быть не знаком с деталями языка, представим некоторые ключевые его возможности. В Ruby все является объектом, и каждый объект является экземпляром какого-то конкретного класса. Например, литерал 42 является экземпляром класса Fixnum, true является экземпляром TrueClass, a nil это экземпляр NilClass. Как и в Java, корнем иерархии классов является Object. Классы являются экземплярами класса Class.

Есть несколько видов переменных, различаемых по префиксу:

- локальные переменные не имеют префикса (x, y, z,...)
- переменные экземпляра (поля) имеют префикс @ (@x, @y, @z, ...)
- переменные класса (аналог статических переменных в Java) имеют префикс @@ (@@x, @@y, ...)
- глобальные переменные имеют префикс $ ($var)

```
1 a = 42 # a in scope from here onward
2 b = a + 3 # equivalent to b = a.+(3)
3 c = d + 3 # error : d undefined
4 b = ” foo ” # b is now a String
5 b. length # invoke method with no args
6
7 class Container # implicitly extends Object
8 def get () # method definition
9 @x # field read; method evaluates to expr in body
10 end
11 def set ( e)
12 @@last = @x # class variable write
13 @x = e # field write
14 end
15 def Container. last () # class method
16 $gl = @@last
17 @@last
18 end
19 end
20 f = Container.new # instance creation
21 f . set (3) # could also write as ” f . set 3”
22 g = f . get # g = 3
23 f . set (4)
24 h = Container.new.get # returns nil
25 l = Container. last # returns 3
26 $gl # returns 3
27
28 i = [1, 2, 3] # array literal
29 j = i . collect { | x | x + 1} # j is [2, 3, 4]
30
31 module My enumerable # module creation
32 def leq ( x)
33 (self.twc x) ≤ 0 # note method ”twc” does not exist here
34 end
35 end
36
37 class Container
38 include My enumerable # mix in module
39 def twc(other) # define required method
40 @x.twc other.get
41 end
42 end
43
44 f.twc f # returns 0
```

Листинг 1

Листинг 1 демонстрирует образец кода на Ruby и иллюстрирует некоторые возможности языка. Локальные переменные не видны вне окружающей их определение области (scope). Локальная переменная начинает существовать в момент ее первого определения. Обращаться до определения к переменным нельзя. Поскольку Ruby динамически типизирован, в ходе работы программы переменная может принимать значения различных типов.

Строки 7-19 определяют новый класс Container с методами get и set, методом класса last , переменной экземпляра @x и переменной класса @@last. Возвращаемым значением метода является результат последней вычисленной команды (строка 9). Также возможны явные команды return. Экземпляры классов создаются с помощью метода new.

Синтаксис вызова методов стандартный (строка 21), хотя круглые скобки можно опускать (строки 20 и 22). Имена методов не обязаны быть буквено-числовыми: возможен метод с именем, например “+”. В частности в строке 2 на самом деле вызывается метод (a.+(3)).

В отличие от локальных переменных, переменные экземпляров, классов и глобальные переменные инициализиются значением nil. Переменные класса являются общими для всех экземпляров класса. Говоря о правилах видимости, переменные классов и экземпляров доступны только внутри их класса. Глобальные переменные видны всюду. Также, всегда доступна переменная self, устанавливаемая в зависимости от контекста.

В языке есть константы. Константой является любая переменная, имя которой начинается с заглавной буквы. В частности, имена классов являются константами.

Как и большинство динамических языков, Ruby имеет специальный синтаксис для литералов массивов (строка 28) и хэш-таблиц.

Также Ruby поддерживает функции высшего порядка, называемые Ruby-блоками (или просто блоками). Строка 29 показывает вызов метода collect, который создает новый массив применением переданного ему блока к каждому элементу исходного массива. Между вертикальными скобками указываются аргументы блока. Надо заметить, что в отличие от методов, блоки имеют доступ к локальным переменным окружающей их области. Любой метод в Ruby может принимать один блок в качестве последнего аргумента и вызывать его с помощью команды yield(v1,...vn). Блоки можно передавать и в качестве обычных аргументов, однако это не тривиально и редко применяется.

Ruby поддерживает единственное наследование. Синтаксис class Foo < Bar означает, что класс Foo наследуется от класса Bar. Если суперкласс не указан, то он предполагается равным Object.

Аналогом множественного наследования в Ruby являются модули (modules, mixins). Например, строки 31-35 определяют модуль My_enumerable, который определяет метод leq в терминах другого метода twc. В строках 37-42 мы "примешиваем" модуль My_enumerable с помощью команды include и затем определяем необходимый метод twc. Начиная со строки 44 мы можем вызывать Container.leq. Заметим, что в строке 37 мы дополняем определение класса Container и примешиваем модуль и добавляем метод. Это один из способов программ изменять классы.

Другими способом изменять класс являются методы alias и define_method. Вызывая alias у класса, мы можем создать копию какого-либо метода под другим именем. Вызов define_method принимает в качестве аргументов имя и блок, из которых под заданным именем создается метод.

Возможно создавать методы, специфичные для конкретного экземпляра (eigen methods или singleton methods). Синтаксис def obj.meth() ... end объявляет метод meth, который будет доступен только у экземпляра obj. Аналогичным по действию является синтаксис class << obj, позволяющий объявить несколько методов за раз и создающий область с переменной self равной obj. Возможность создавать методы специфичные для конкретных экземпляров означает, что тип в Ruby определяется не столько классом, сколько набором доступных у экземпляра методов.

Наконец, язык поддерживает возможность исполнения строк как кода на Ruby (eval, instance_eval, etc.).

Специфика интегрированных сред разработки

Такая область применения как интегрированные среды разработки имеет свою специфику, отличную от специфики применения статических анализаторов в трансляторах и верификаторах. Рассматривая самые востребованные действия пользователя, которые могут потребовать анализа программы, можно выделить следующие:

1. Подсказка при вводе метода (code completion). Реализация данной функции требует знать список имен методов, которые можно вызвать в указанной точке программы. А значит, это требует вычисления типа объекта, у которого вызывается метод.

2. Переход к определению (jump to declaration). Реализация перехода к определению требует зная имя метода, перейти к его определению. Для этого опять же нужно знать тип объекта, у которого вызывается метод.

Первой особенностью, которую можно заметить из данных кратких описаний, является формат обращения к анализу: в обоих случая это вопрос об одной единственной конкретной переменной (или выражении, что не важно).

Второй особенностью, и в частности требованием, является необходимость в малом времени на ответ на такие вопросы. Пользователь может ждать секунду-две, но не больше. При этом он не сильно огорчится, если результат окажется не полным или даже не корректным.

Треьей особенностью является то, что редактируя код, пользователь изменяет программу. А значит если статический анализатор сохраняет какие-либо результаты вычислений для оптимизации (а это логично делать), то после изменения программы, он должен эффективно их пересчитать. Можно привести простой практический пример: программа состоит из большой библиотеки и небольшого клиентского кода, ее использующего. Если пользователь работает над клиентским кодом, не изменяя код библиотеки, не эффективно переанализировать ее при каждом запросе к анализу. Здесь нужно заметить, что длительный анализ в момент запуска среды разработки не возбраняется. Тем не менее при изменении кода, задержки на анализ должны быть пропорциональны масштабности изменений.

Наконец, нужно сказать про реализационные особенности. Так как среда разработки часто использует промежуточное представление и индексы, аналогичные тем, что могут понадобиться анализу, рациональным является переиспользование этих структур между друг другом. А это означает, что анализ также должен быть реализован на языке, на котором возможна среда разработки. Рассматривая существующие среды, которые позволяют интеграцию в них анализа для Ruby, таким языком является Java.

Еще одним аспектом, ограничивающим выбор языка программирования для реализации, является скорость. К сожалению удобные для написания статического анализа языки, такие как Prolog, сам Ruby, Python, OCaml исполняются несравнимо более медленно чем Java.

Существующие решения

Ruby

Static Type Inference for Ruby

Авторами предлагается система под названием DRuby для поиска ошибок в программах на языке Ruby. Работа включает алгоритм для вычисления типов. Их алгоритм основывается на итеративном решении построенной системы ограничений и предлагается его реализация на OCaml. Авторы утверждают, что их алгоритм выдает точные ответы во всех возможных случаях. Однако время работы на тестовой программе в 800 строк составляет 36.1 секунды, что является слишком медленным, чтобы использовать их алгоритм в интерактивных средствах. Одним из результатов их работы являются типовые аннотации для стандартной библиотеки Ruby. Представленная здесь работа переиспользует этот результат.

RadRails, RubyMine

RadRails и RubyMine это популярные среды разработки для Ruby с поддержкой помощи пользователю (code completion). Документальных описаний статического анализа в них не известно. Однако экспериментальная проверка показала, что авторами скорее всего используется тривиальный внутрипроцедурный анализ.

Другие динамические языки

DDP

Автором предлагается статический анализатор для языка Smalltalk основанный на управляемом запросами анализе с отсечениями. В ней показывается применимость данного подхода к задачам статического анализа, а также выигрышность по сравнению с классическими (итеративными) подходами. Именно данная работа послужила источником идей для написания представленной, т.к. Smalltalk является во многом похожим на Ruby.

Формальная постановка задачи

Задача состоит в разработке программной системы (статического анализатора), принимающего на вход обозначенные ниже данные и выдающие обозначенный ниже результат.

Вход.

На вход анализатора поступает программа на Ruby: набор .rb файлов R, а также выбранный среди них файл m \in R, являющийся файлом, который непосредственно

передается интерпретатору Ruby в качестве входного. Этот файл может включать остальные с помощью команды require.

Также анализатору указывается файл из входного набора и позиция в нем. На указанной позиции в этом файле должен находиться литерал, ссылающийся на какую-либо переменную.

Выход.

На выход алгоритм должен выдать описание типа обозначенной на входе переменной. Описание типа должно следовать следующей структуре:

ОписаниеТипа := Тип (или Тип)*

Тип := (Источник, Методы)

Источник := <конкретное имя класса>

Источник := <вызов метода new>

Источник := <произвольный вызов с неизвестным возвращаемым типом>

Источник := <неопределенная переменная>

Методы := Метод*

Метод := <описание сигнатуры метода: имя и агрументы>

Описание типа должно соответствовать типам, которые может принимать указанная переменная в процессе работы программы.

Решение

Управляемый запросами анализ

Большинство из описанных ниже идей были заимствованы из работы Спуна[], которые им в свою очередь были заимствованы из экспертных систем. Таким образом общая идея управляемых запросами алгоритмов не нова. Данная работа расширяет существующие решение поддержкой кеширования результатов между вопросами и эффективным обновлением закешированных данных при изменении входных данных.

Идея

Рассмотрим несколько наблюдений над существующими работами и над природой самой задачи. Данные наблюдения позволят лучше понять как решать поставленную задачу.

Во-первых, все современные контекстно-зависимые алгоритмы не масштабируемы. 0-CFA, CPA и k-CFA [] не способны справиться с программами в несколько десятков тысяч строк.

Во-вторых, в любых программах, даже самых больших, большинство типов всегда вычисляется очень просто. Достаточно посмотреть на расположенные рядом присваивания, чтобы найти подходящий литерал. Любой человек таким образом взглянув на программу способен дать результат, попой более точный чем любой из существующих анализаторов.

В-третьих, инкрементальный анализ традиционно считается крайне сложной задачей. В то же время, идея удаления ставших некорректными результатов, затем тех, которые зависят от них и т.д. выглядит тривиальной, если известны зависимости между результатами и результаты достаточно мелки.

Наконец, рассмотрим такой пример ситуации. В любой достаточно большой программе всегда найдется метод, который вызывается из тысячи мест. Практически любой из существующих анализаторов потратил бы время на рассмотрение каждого из них. В то же время, для точного ответа на поставленный ему вопрос скорее всего достаточно рассмотрения лишь десятка из них.

Главной идеей является то, что статический анализатор мог бы, имея ограниченные ресурсы, потратить их на поиск ответа на заданный вопрос. Если ресурсов оказалось недостаточно, то он должен выдать тривиальный корректный ответ. При этом для разных типов вопросов он применял бы различные стратегии вычисления. Тогда на простые вопросы он мог бы отвечать быстро и хорошо, а на сложные хуже.

Управляемый запросами алгоритм следует описанной идее. На каждый поставленный ему вопрос выбирается своя стратегия вычисления. При этом в поисках ответа на вопрос он может задавать новые вопросы.

Естественным расширением этой идее является то, что алгоритм мог бы не искать ответа на некоторые возникшие вопросы. В таком случае, если изначальный вопрос приводит к очень сложным вопросам, ответ на которые результат не улучшит, он мог бы не вычислять только их, Это позволило бы успешно ответить на изначальный вопрос в условиях ограниченных ресурсов. Данное уточнение позволяет называть алгоритм управляемым запросами с отвечениями.

Важно заметить, что отсечения можно делать не с любыми вопросами. Поэтому вопросы должны быть правильно сформулированы. Например, вопрос “какой тип имеет x?” можно легко отсечь, потому что для него существует тривиальный корректный ответ: “Ob-

ject". Вопрос вида "проанализируй строку n и обнови таблицу b" отсекать невозможно, т.к. это вопрос в общем-то не является. Вообще, рассматривая данные примеры, становится ясным, что к вопросам предъявляются следующие требования:

- существование тривиального корректного ответа

- отсутсвие побочных эффектов от их рассмотрения (то есть каждый вопрос рассматривается изолированно)

Наконец, представим ситуацию, в которой алгоритму постоянно задают вопросы. Логично, что многие вопросы возникающие в процессе анализа будут повторяться. Поэтому естественным является записывать ответы на них, чтобы при повторых подобных вопросах не искать на них ответ.

Данная оптимизация, тем не менее, затрудняется если входные данные алгоритма меняются время от времени. Это означает, что старые ответы уже не корректны. Однако, бывает, решив какую-то задачу на бумаге, а потом получив аналогичную, вы переиспользуете большинство своих записей. Вы последовательно зачеркиваете устаревшие записи и записываете новые вместо них, затем те, на что те повлияли и т.д. Естественным решением проблемы, которое использует человек, является запоминание того, как вопросы зависят друг от друга и от входных данных. Подобную стратегию можно применять и при статическом анализе.

Алгоритм

Предлагаемы алгоритм использует подход описанный выше. Он управляется запросами, позволяет делать делать отсечения и эффективно кеширует ответы между вопросами.

Псевдокод алгоритма преставлен в Листинге 2. На вход алгоритм принимает goal — исходный запрос. Выходом алгоритма является ответ на поставленный запрос.

Главной частью алгоритма является множество worklist содержащее набор запросов, ответы на которые нужно обновить. Алгоритм последовательно выбирает из worklist запросы и обновляет их ответы. Если после обновления ответа как какой-то запрос, ответ действительно изменился, тогда все запросы зависящие от данного помещаются в worklist, так как теперь их ответ тоже мог измениться. Алгоритм останавливается, когда worklist окажется пустым, а значит ответы на все запросы будут согласованы с ответами на подзапросы.

Функция UpdateOneGoal обновляет ответ на переданный ей запрос та, чтобы он учитывал ответы на подзапросы. Например, для запроса "тип x?" ответ может уточниться с "Fixnum" до "Fixnum or Float". Функция Update выполняет данное уточнение и возвращает булево значение, означающее изменился ответ по сравнению с предыдущим или нет.

Точное поведение Update зависит от конкретных типов вычислений, производимых с помощью данного алгоритма, и применительно к анализу Ruby рассматривается далее. Заметим, что если обновляемому запросу потребовались новые подзапросы, ранее не существовавшие, то тогда они создаются, инициализируются минимальным ответом и добавляются в worklist. Если обновление ответа привело к тому, что ответ действительно изменился, то UpdateOneGoal помещает в worklist все запросы, запрашивающие его как подзапрос.

Время от времени алгоритм вызывает Prune и отсекает некоторые подзапросы. ChoosePrunings выбирает какие запросы должны быть отсечены. ChoosePrunings это эвристика, и существует множество возможных стратегий по ее реализации. Всем запросам, которые отсекаются, назначается тривиальный ответ. После этого worklist очищается от отсеченных запросов, так чтобы он содержал только исходный запрос и все непосредственные и косвенные его подзапросы.

Для того чтобы GoalsNeeding не нарушала эффективности алгоритма, нужно чтобы после отсечений она не возвращала отсеченных запросов. Иначе это будет фактически отменять отсечения. Также логично хранить дополнительный набор completed. Он должен хранить все запросы, которые обработаны и ответы на которые уже согласованы с подзапросами. Непосредственно после обновления запроса он должен помещаться в completed. И когда запрос помещается в worklist, он должен исключаться из completed. Таким образом во время работы алгоритма запросы перемещаются из completed в worklist и обратно, всегда находясь в одном из них. Отсеченные запросы удаляются из обоих наборов. GoalsNeeding же может возвращать только запросы, находящиеся каком-то из этих множеств.

Функция InputAffected вызывается между вызовами Evaluate, то есть предполагается, что все обработанные ранее запросы находятся в completed. InputAffected перевычисляет потенциально изменившийся запрос, при условии что у него было подзапросов (а значит он непосредственно зависил от входных данных и только). В случае, если его ответ действительно изменился, запрос, а также все прямо и косвенно зависящие от него удаляются из completed. Теперь при следующем обращении к Evaluate их потребуется вычислить заново.

Пример исполнения

TODO

Свойства

Приведенный алгоритм обладает несколькими полезными свойствами. По-первым время его работы зависит только от количества обрабатываемых запросов и того, сколько раз каждый из них обрабатывается. По крайней мере прямой зависимости от того, насколько большая программа анализуется не ожидается. А значит алгоритм должен работать одинаково быстро вне зависимости от размера входной программы, если ограничить количество запросов (это позволяют сделать отсечения).

Затем, алгоритм в простых случаях находит ответ моментально. Так, вопросы приводящие через один-два шага к литералам, гарантированно будут найдены.

Наконец, алгоритм может быть настроеным под конкретную ситуацию. Выбирая стратегии отсечения можно контролировать соотношение скорости/качества. Также, обратим внимание на то, что не было указано, по какому принципу выбирается следующий запрос из worklist. Это также влияет на свойства алгоритма.

Если worklist работает по принципу очереди, то сначала будут рассматриваться более новые подзапросы. А значит алгоритм будет больше уходить “в глубину”, обновляя ответы на вопросы близкие к корневому в последнюю очередь. Если же worklist организовать по принципу стека, то появление конкретного ответа на какой-то подвопрос будет в первую очередь “просачиваться” на верх и уточнять корневой запрос до рассмотрения остальных подзапросов. Однакой ценой за такие скорые ответы является то, что корневой вопрос будет пересчитываться каждый раз, когда обновится какой-то из подзапросов, что может привести к общему замедлению работы.

Итак, стратегию выбора следующего запроса можно выбирать в зависимости от наличия ресурсов и конкретной задачи. При этом на результат данный выбор не влияет.

Утв. (корректность): Для любой стратегии выбора остановка алгоритма означает, что все ответы на все запросы согласованы с ответами на подзапросы.

> очевидно <

Теперь давайте вернемся к словам про то, от чего зависит время работы алгоритма. Как было замечено, запросы могут обрабатываться несколько раз — фактически, каждый раз, когда изменятся ответ какого-то из их подзапроса. Возникает логичный вопрос о том, как часто один запрос может обновляться, и вообще останавливается ли алгоритм.

Во-первых, как уже было замечено, выбор стратегии может влиять на то как часто запросы перевычисляются. Это является отдельным полем для исследования, и наверняка можно выбрать как стратегию с минимальным количеством пересчетов, так и с максимальным. Однако нам важен лишь тот факт, но данный выбор не влияет на результат.

Во-вторых, давайте представим запросы и завимости между ними как орграф. Граф может быть двух видов: циклический и ациклический. Можно доказать следующее утверждение:

Утв. (критерий остановки №1): Если запросы образуют ациклический граф, то для любых стратегий выбора и отсечения алгоритм останавливается.

> Индукция по размеру (к-ву вершин) подграфа подзапроса. <

Если граф циклический, алгоритм может не останавлиться. Но так как предусмотрены отсечения, то выбрав подходящую стратегию отсеения можно алгоритм остановить.

Все же попробуем придумать условия, при которых можно гарантировать остановку. Пусть все возможные результаты какого-либо запроса упорядочены, т.е. для любых двух ответов a1 и a2 можно было бы сказать a1 < a2, a1 = a2 или a2 > a1.

Опр.: Запрос будем называть монотонным, если при каждом последующем его обновлении предыдущий результат всегда больше (всегда меньше) предудущего.

Тогда можно сформулировать следующее утвреждение:

Утв. (критерий остановки №2): Если все запросы монотонны и множество результатов каждого конечно, то алгоритм всегда останавливается.

> Для любого запроса рано или поздно наступит момент, когда он больше не будет изменять свой ответ, оставаясь согласованным со своими подзапросами. А значит также рано или поздно все запросы окажутся согласнованными и алгоритм остановится. <

Первоначальные преобразования

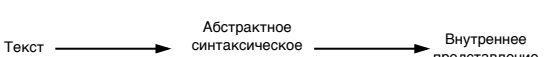
В одной из предыдущих частей мы описали поставленную задачу следующих образом:

Вход.

На вход анализатора поступает программа на Ruby: набор .rb файлов R, ...

Иными словами, на вход разрабатываемого статического анализатора поступает текст программы. Очевидно, что производить анализ оперируя с текстом невероятно сложно и глупо.

Поэтому до начала непосредственно анализа программа проделывает следующий путь:



Первым преобразованием является лексический и синтаксический анализ. Выходом этого преобразования является абстрактное синтаксическое дерево программы (АСД).

Теоретически, анализ программы уже можно проводить на данном представлении. Однако, это не является удобным ввиду следующих причин:

- АСД содержит слишком много конструкций, многие из которых равнозначны при вычислении типов

- АСД не отражает аспектов касающихся потоков данных и потока исполнения программы

Поэтому, АСД преобразуется дальше в структуры данных, удобные для нашего анализа, которые мы и называем внутренним представлением.

Внутреннее представление программы

По АСД строятся следующие структуры:

- графы управления для инструкций находящихся на верхнем уровне в файле (иными словами то, что будет исполнено, когда файл будет подан на вход интерпетатора)

- графы управления для методов и блоков (то, что может быть исполнено в процессе работы программы)

- индексы

В качестве вершин графов управления используется значительно упрощенный набор команд, определяемый ниже:

```
CodeBlock
        CFG cfg;
        CodeBlock parent;

CFG := Graph<CFGNode>

CFGNode :=
        VAR = RHS |
        break |
        next |
        redo |
        if VAR |
```

include VAR |
require VAR |
def method(PARAM) CodeBlock |
def VAR.method(PARAM) CodeBlock |
return VAR

RHS := CALL | LITERAL | VAR
VAR := x | @x | @@x | $x | X
LITERAL := fixnum | string | array | ...
CALL := VAR.method(VAR1, ... , VARk) CodeBlock |
VAR.call(VAR1, ..., VARk) CodeBlock |
super(VAR1, ..., VARk) CodeBlock

PARAM := arg1, ..., argN, optarg1, ..., optargM, restarg, blockarg

Индексы

TODO

- места вызовов
- места определений блоков и методов
- места присваиваний
- места использований

Запросы

TODO

Значения

TODO

values (x)

if x is localval

if x is instancevar

if x is classvar

if x is globalvar

if x is const

if x is call

if x is .new

if x is x[...]

Использования

if x is localval

if x is instancevar

if x is classvar

if x is globalvar

if x is const

if x is x[...]

Места вызова

if method

if block

Вызываемый код

if method

if yield

if call

if super

TODO

Результаты и эксперименты

TODO

sadr analysis engine

800 строк: 43% точно, < 1 секунды

Заключение

В данной работе была описана система для статического анализа программ на языке Ruby подходящая для использования в интегрированных средах разработки, тем самым удовлетворяющая поставленной задаче.

Была произведена следующая работа:

- разработан, описан и реализован модифицированный алгоритм DDP с поддержкой кеширования между запросами и икрементальным обновлением результатов

- исследован и описан язык Ruby с точки зрения проблем статического анализа

- разработан, описан и реализован основанный на модифицированном DDP статический анализ для языка Ruby, позволяющий вычислять типы

- в рамках проекта DLTK Eclipse Foundation был реализован упрощенный статический анализатор для языка Ruby, а также алгоритм DDP без поддержки инкрементальности

Вместе с работой Спуна, данная работа показывает актуальность применения алгоритмов основанных на запросах в статическом анализе. Данный подход позволяет успешно решать проблему масштабирования, остро стоящую для многих алгоритмов анализа, при этом не значительно теряя в точности. Важно заметить, что такой подход может быть применен не только к анализу программ на языке Ruby, но и к анализу любых программ вообще. Более того, такой подход может быть применен для решения задач не только анализа программ. Крайне интересны возможные применения управляемого запросами анализ к задачам биоинформатики, социологии, экономики связанных с системами взаимосвязанных элементов.

Наконец, еще одним существенным преимуществом такого подхода является предрасположенность к распараллеливанию. Если вычисляемые наборы запросов независимы, их можно эффективно вычислять на различных ядрах процессора. Решение задачи распараллеливания является одним из следующих шагов данной работы.

Тестирование реализации анализа для языка Ruby показало приемлимый уровень точности анализа. Однако были выявленны не учтенные изначально особенности языка, требущие более подробного рассмотрения: коллекции, исключения. Также было обнаружено, что точность описания библиотек и нативных функций оказывает крайне существенное влияние на точность анализа.

Реализация в рамках проектах Eclipse DLTK является достаточно популярной и достаточно порадовавшей многих разрабочиков на Ruby []. Также эта реализация используется в среде разработки для Ruby On Rails 3rd Rail от CodeGear (бывшая Borland).

Наконец, данная работа является примером возможности создания практически полезных статических анализаторов для динамически типизированных языков. В частности, она может быть полезна при создании анализаторов для других динамических языков, таких как Python или PHP.

Литература

TODO

Приложения

TODO