Министерство образования и науки Российской федерации

Федеральное агентство по образованию

Новосибирский государственный университет

Механико-математический факультет

Кафедра программирования

Выпускная квалификационная работа специалиста

КАЛУГИН Михаил Борисович

Управляемый запросами статический анализ для языка Ruby

Научный руководитель:

Кандидат физико-математических наук, старший научный сотрудник

И.Н. Скопин

Новосибирск 2009

Содержание

1 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 приемлемым качеством.

Одними из основных причин, затрудняющих анализ программ на Ruby, являются следующие особенности языка:

- динамическая типизация
- не самая простая система типов (так называемый 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 программами 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 разработанный автором основанный на DDP статический анализатор, 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], вместе с которой 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 соответствие ожиданиям, а также теоретическим оценкам.

2 Задача

2.1 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 существует, и язык полностью определяется существующей реализацией интерпретатора.

Поскольку 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)

Листинг 1 Пример программы на Ruby

```
1  a = 42                # теперь a доступен
2  b = a + 3             # то же самое, что и b = a.+(3)
3  c = d + 3             # ошибка: d неопределен
4  b = foo               # b теперь экземпляр String
5  b.length              # вызов метода length без аргументов
6
7  class Container          # по умолчанию наследник Object
8      def get()        # определение метода
9          @x       # чтения поля, метод возвращает это значение
10     end
11     def set(e)
12         @@last = @x # запись в переменную класса
13         @x = e      # запись в поле
14     end
15     def Container.last()     # метод класса
16         $gl = @@last
17         @@last
18     end
19 end
20 f = Container.new   # создание экземпляра
21 f.set(3)     # можно было также написать f.set 3
22 g = f.get   # g = 3
23 f.set(4)
24 h = Container.new.get   # возвращает nil
25 l = Container.last  # возвращает 3
26 $gl      # возвращает 3
27
28 i = [1, 2, 3]   # литерал массива
29 j = i.collect { |x| x + 1 }      # теперь j равен [2, 3, 4]
30
31 module MyEnumerable # определение модуля
32     def leq(x)
33         (self.twc x) <= 0   # заметим, что метода twc здесь нет
34     end
35 end
36
37 class Container
38     include MyEnumerable     # примешиваем модуль
39     def twc(other)       # объявляем недостающий метод
40         @x.twc other.get
41     end
42 end
43
44 f.twc f # возвращает 0
```

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 буквенно-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 инициализируются 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 интерпретатором 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, или замыканиями). 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 программно изменять классы.

Другими способом изменять классы 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.).

2.2 Специфика интегрированных сред разработки

Такая область применения статического анализа 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.

Третьей 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 анализировать ее заново 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 совместное использование этих структур. 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.

2.3 Существующие решения

2.3.1 Специфичные для Ruby

Static Type Inference for Ruby Авторами предлагается система под названием DRuby для поиска ошибок в программах на языке Ruby. Работа включает алго-

ритм 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 использует 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.

2.3.2 Другие динамические языки

DDP Автором предлагается статический анализатор для языка Smalltalk основанный на управляемом запросами анализе с отсечениями. В его работе 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.

2.4 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 некоторый выбранный среди них файл, являющийся файлом, который непосредственно передается интерпретатору Ruby в качестве входного, и с которого начинается исполнение программы. 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:

*Описание Типа := Тип (∨ Тип)**
Тип := (Базовый Тип, Дополнительные Методы)
Базовый Тип := <конкретное имя класса>
Базовый Тип := <вызов метода new>
Базовый Тип := <произвольный вызов с неизвестным возвращаемым типом>
Базовый Тип := <неопределенная переменная>
*Дополнительные Методы := Метод**

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

Описание типа должно соответствовать типам, которые может принимать указанная переменная в процессе работы программы. Заметим, что такое описание типа является достаточно полным для того, чтобы, имея результат работа статического анализатора, делать заключения о его надежности и возможных подсказках пользователю на его основе.

3 Решение

3.1 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 запросами и эффективным обновлением закешированных данных при изменении анализируемой программы.

3.1.1 Идея

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

Во-первых, почти все современные контекстно-зависимые алгоритмы не масштабируемы.

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, порой более точный чем большинство 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 атомарны.

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 лучшим образом 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 идеи является то, что алгоритм мог бы не искать ответы 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 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. Вопросы должны быть правильно сформулированы. Например, вопрос "какой тип имеет x?" можно легко отсечь, потому что для него существует тривиальный корректный ответ: "Object". Вопрос вида "проанализируй строку n и обнови таблицу b" отсекать невозможно, т.к. это вопросом в общем-то не является. Вообще, рассматривая данные примеры, становится ясным, что к вопросам предъявляются следующие требования:

- существование тривиального корректного ответа
- отсутствие 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 повторных 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.

3.1.2 Алгоритм

Предлагаемый алгоритм реализует идеи описанные выше. Он управляется запросами, позволяет делать отсечения и эффективно кеширует ответы между вопросами.

Листинг 2 Управляемый запросами алгоритм с отсечениями и инкрементальным обновлением вычисленных запросов

```
procedure Evaluate(rootgoal)
    if completed contains rootgoal then
        return GoalAnswer(rootgoal)

    worklist := { rootgoal }

    while worklist != {} do
        if pruner wants to run
        then Prune()
        else UpdateOneGoal()

    return GoalAnswer(rootgoal)

procedure UpdateOneGoal()
    Remove g from worklist
    changed := Update(g)

    if changed then
        deps := GoalsNeeding(g)
        worklist := worklist + deps
        completed := completed - deps

procedure Prune()
    for g in ChoosePrunings() do
        prune g
    worklist := Relevant(rootgoal)

procudure InputAffected(goal)
    changed := Update(g)
    if changed then
        Wipe(goal)

procedure Wipe(goal)
    complete := completed - goal
    deps := GoalsNeeding(goal)
    for g in deps do
        Wipe(g)
```

Псевдокод алгоритма представлен 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 оста-

навливается, когда worklist окажется пустым, а значит ответы на все запросы будут согласованы с ответами на подзапросы.

Функция UpdateOneGoal обновляет ответ на переданный ей запрос так, 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 тривиальным ответом и добавляются в worklist. Если обновление ответа привело к тому, что ответ действительно изменился, то UpdateOneGoal помещает в worklist все запросы, запрашивавшие 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. 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 compeleted. 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 не было подзапросов (а значит он непосредственно зависел 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.

3.1.3 Пример работы

Проиллюстрируем работу алгоритма на конкретном примере. Рассмотрим случай анализа типов в программе, приведенной в Листинге 3.

Листинг 3 Пример анализируемой программы. Нас интересует тип $\$x$ в строке 7.

```
1  class MyClass
2
3          def foo(p1)
4                  $x = $y
5                  doStuff()
6                  $x = p1
7                  $x
8          end
9          def baz
10                 self.foo $y
11         end
12         def boz
13                 self.foo 45
14         end
15         def qux
16                 $y = 45
17         end
18
19 end
```

В данной программе нас интересует тип переменной $x в строке 7. Данный вопрос будет корневым запросом при запуске алгоритма.

Приведенные ниже схемы (рис. 1 — рис. 10) иллюстрируют работу алгоритма по шагам. На каждой схеме обозначены все существующие запросы. Запрос обозначается тремя строками:

1. Предмет запроса
2. Состояние: completed означает, что запрос находится в списке completed. Пустая строка означает, что запрос находится в списке worklist.
3. Ответ на запрос

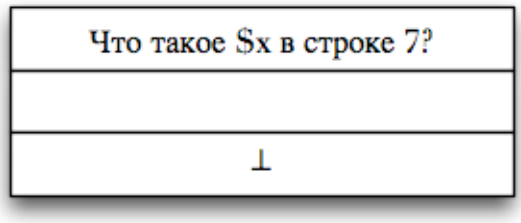


Рис. 1: Начало работы над запросом “Что такое $x в строке 7?”. Запрос находится в рабочем списке и имеет пустой ответ, обозначаемый как ⊥.

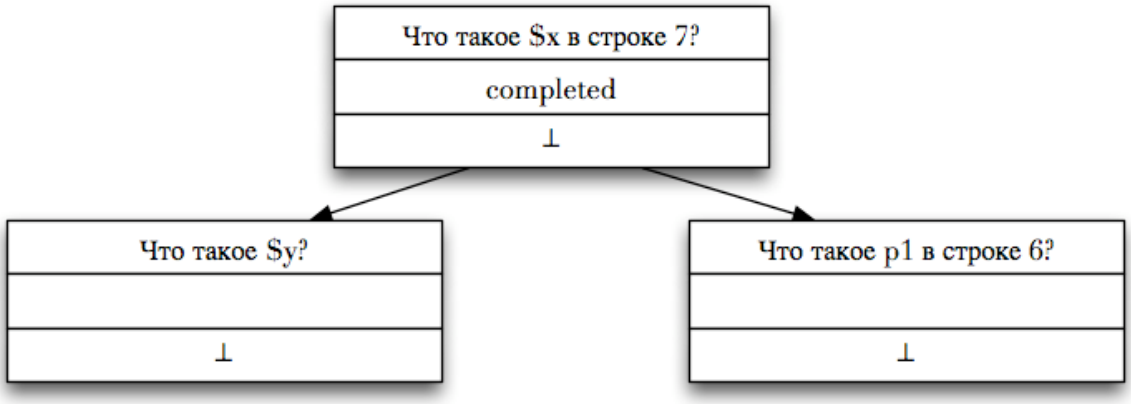


Рис. 2: Корневой запрос обработан. Для него потребовалось два новых запроса, которые помещены в рабочий список. Ответ на корневой запрос согласован с ответами на подзапросы, и поэтому запрос перемещен в список completed.

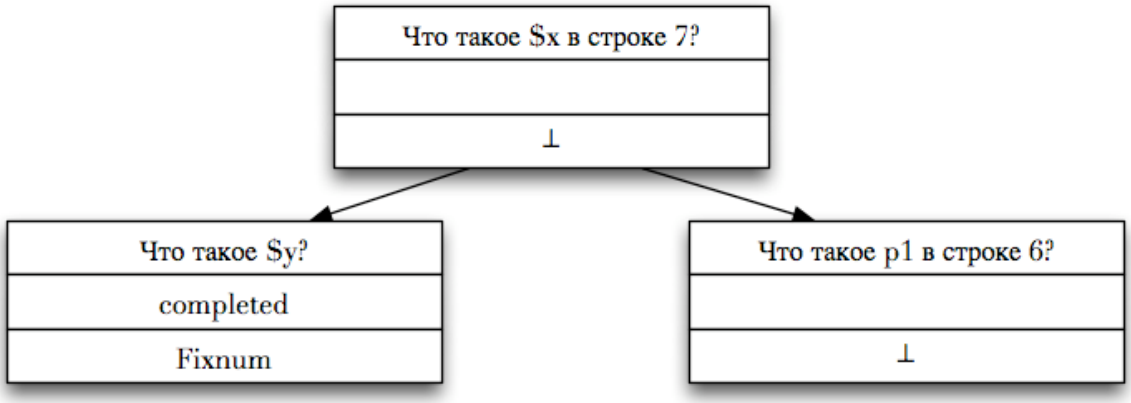


Рис. 3: Запрос “Что такое $y?” обработан и имеет непустой результат. Поскольку корневой запрос зависит от него, корневой запрос перемещен из completed в рабочий список.

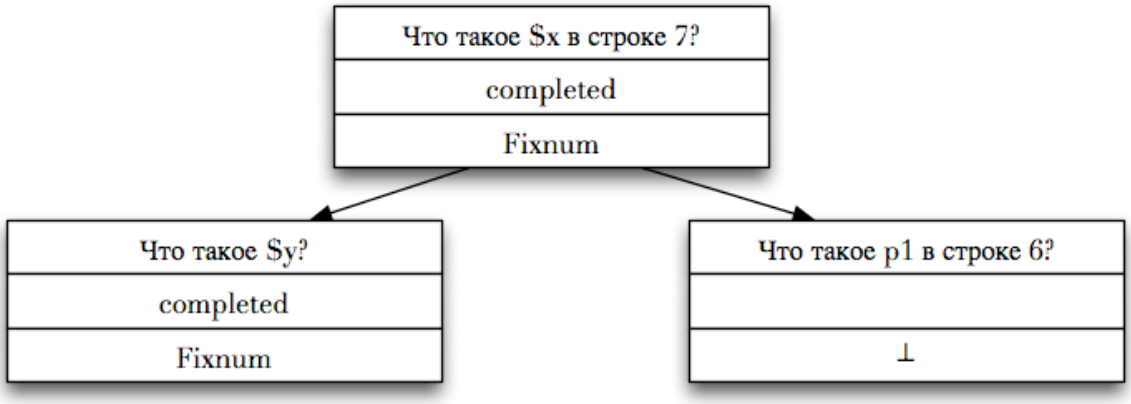


Рис. 4: Корневой запрос обработан второй раз. Теперь его ответ снова согласован с подзапросами и равен Fixnum. Корневой запрос снова перемещен в completed.

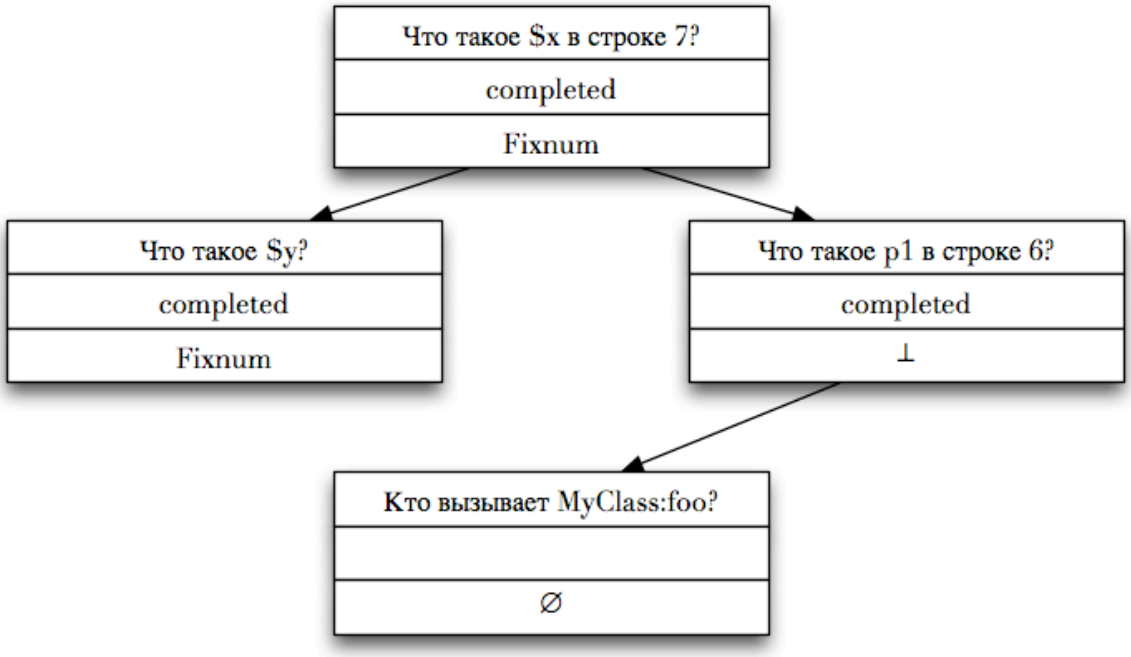


Рис. 5: Обработан второй подзапрос корневого запроса. Для него потребовался ответ на запрос “Кто вызывает MyClass:foo”, который помещен в рабочий список.

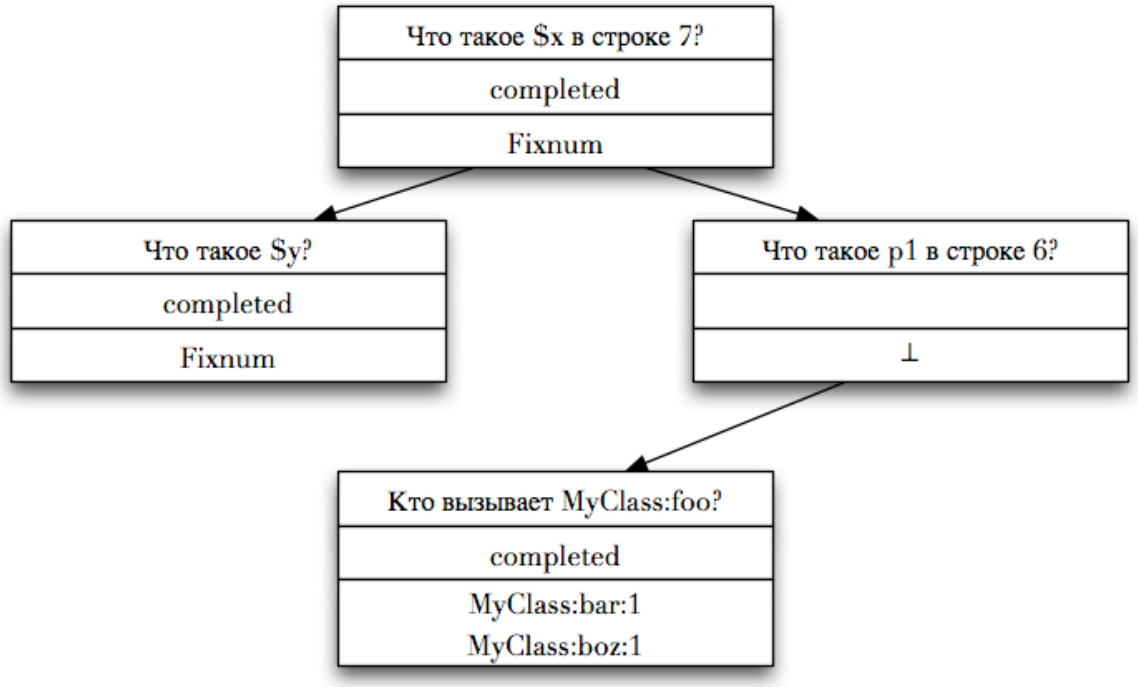


Рис. 6: Обработан запрос “Кто вызывает MyClass:foo” и теперь он имеет ответ из двух точек вызова. От него зависит запрос “Что такое p1 в строке 6?”, поэтому тот снова перемещается в рабочий список для обработки.

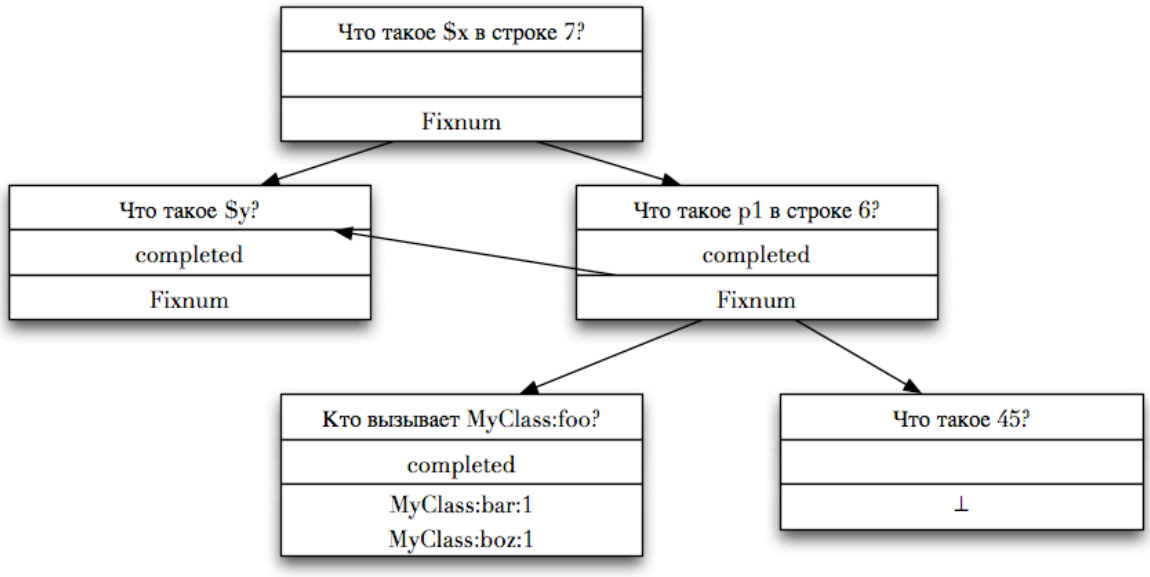


Рис. 7: Повторное вычисление запроса “Что такое p1 в строке 6?” потребовало два подзапроса. Один из них уже вычислялся и имеет ответ Fixnum. Поэтому ответ на запрос изменился. От него зависит корневой запрос, поэтому тот помещается в рабочий список.

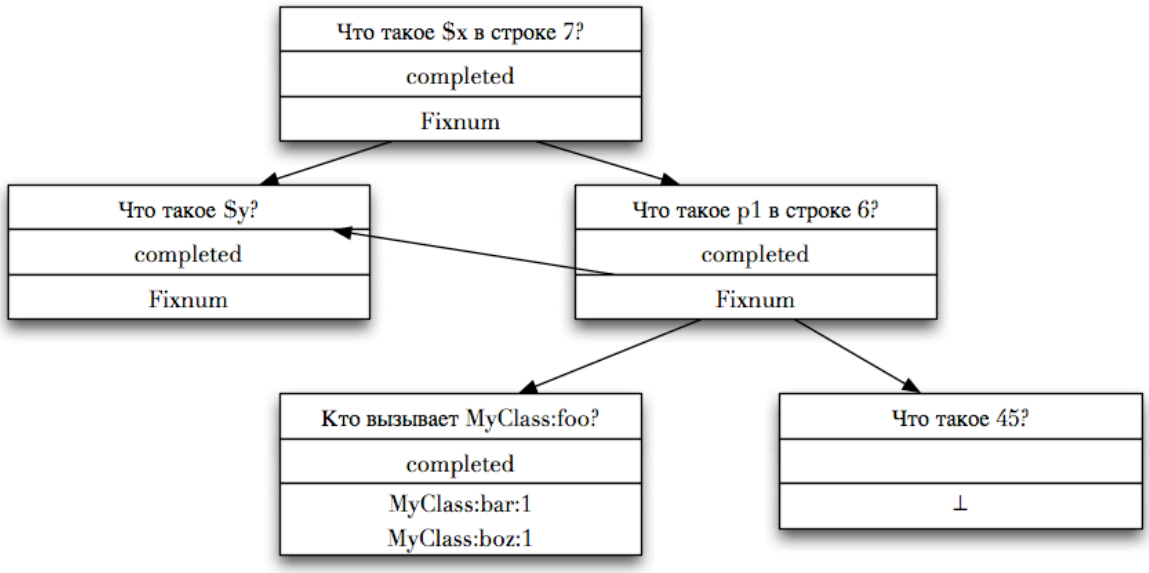


Рис. 8: Корневой запрос снова согласован с подзапросами и помещен в completed.

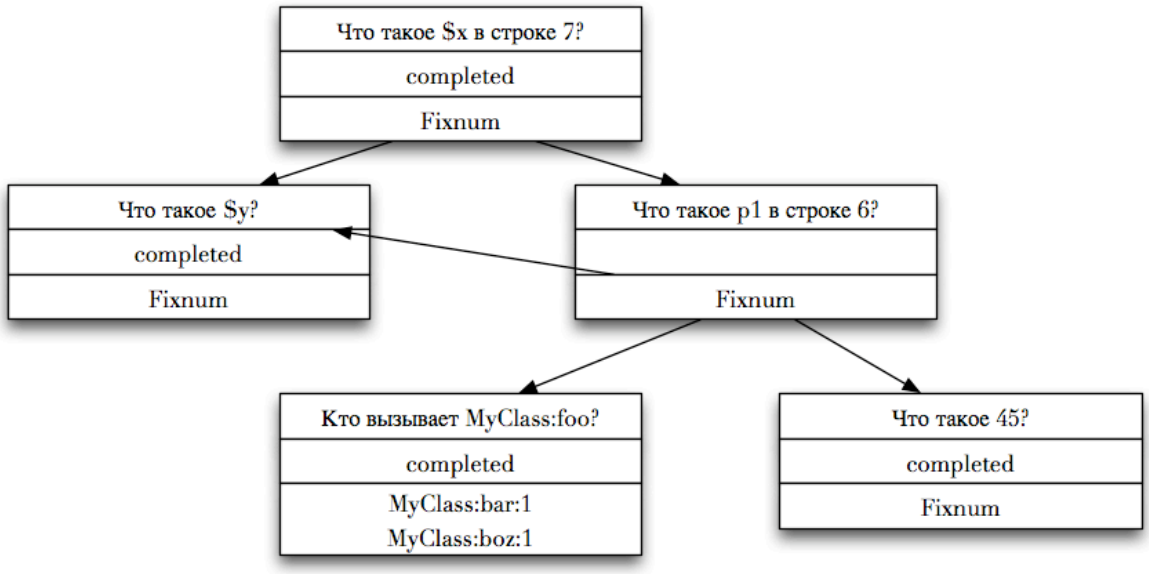


Рис. 9: Запрос “Что такое 45?” обработан. От него зависел “Что такое p1 в строке 6?”, поэтому он помещен снова в рабочий список.

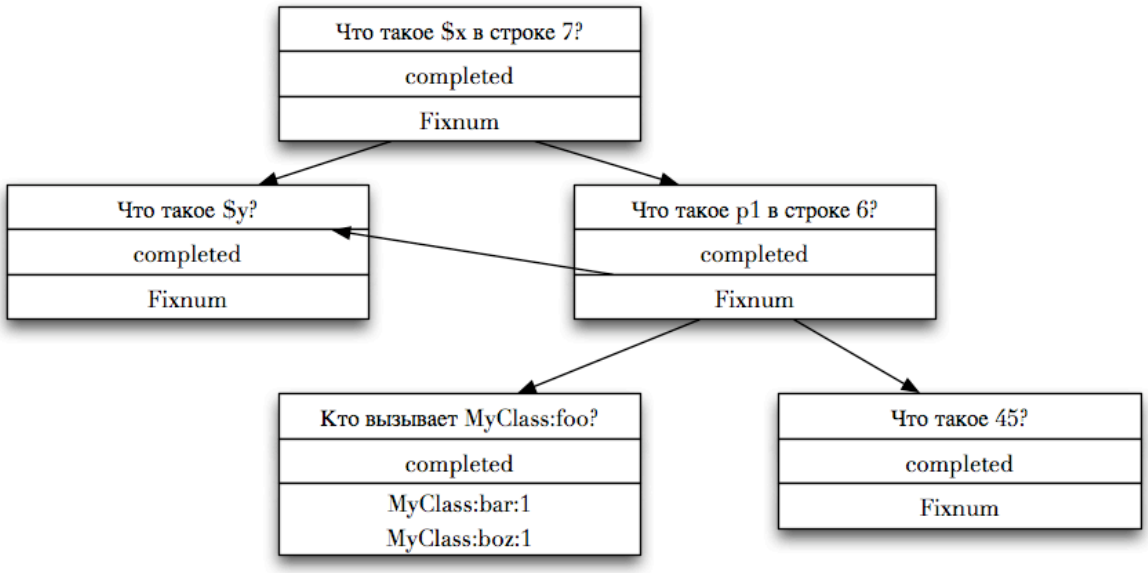


Рис. 10: Запрос “Что такое p1 в строке 6?” обработан снова. Его ответ не изменился. Рабочий список пустой, алгоритм останавливается.

3.1.4 Свойства

Приведенный алгоритм обладает несколькими полезными свойствами. Во-первых, время его работы зависит только от количества обрабатываемых запросов и того, сколько раз каждый из них обрабатывается. А значит, алгоритм должен работать одинаково быстро, если количество запросов ограниченно (это позволяют гарантировать отсечения). Предполагая, что с ростом размера программы, сложность взаимосвязей в ней растет слабо, ограничение числа запросов не должно вызывать значительных потерь в точности. То есть утверждается, что алгоритм сможет масштабироваться.

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

Наконец, алгоритм может быть настроенным 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” наверх и уточнять корневой запрос до рассмотрения остальных подзапросов. Однако, 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. Можно сформулировать следующее очевидное утвреждение:

Утв. (корректность): Для любой стратегии выбора остановка алгоритма озна-

чает, что все ответы на все запросы согласованы с ответами на подзапросы.

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 зависимости 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 (количеству вершин) подграфа подзапроса. ▶

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

Все же попробуем придумать условия, при которых можно гарантировать остановку без помощи отсечений. Для этого введем некоторый порядок на результатах запросов. Пусть все возможные результаты какого-либо запроса сравнимы, т.е. для любых двух ответов a_1 и a_2 можно было бы сказать $a_1 < a_2$, $a_1 = a_2$ или $a_1 > a_2$.

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

Тогда можно сформулировать следующее утверждение:

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 ввилу конечности цепи результатов, оставаясь согласованным со своими подзапросами. А значит также рано или поздно все запросы окажутся согласованными и алгоритм остановится. ▶

3.2 Первоначальные преобразования

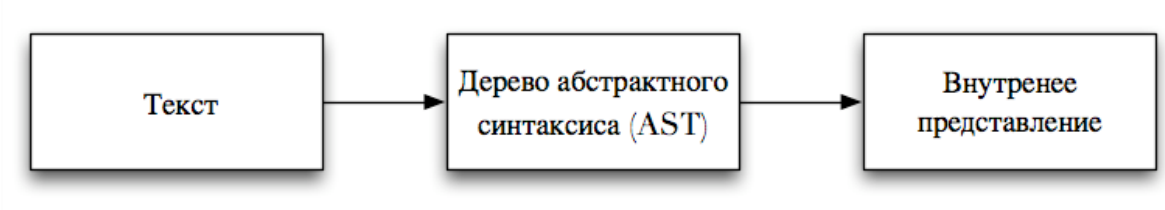
В одной из предыдущих частей мы описали поставленную задачу следующих образом:

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 дерево абстрактного синтаксиса (ДАС или, как оно будет обозначаться далее, AST).

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

- AST содержит слишком много конструкций, многие из которых равнозначны при вычислении типов;
- AST не отражает аспектов касающихся потоков данных и потока исполнения программы.

Поэтому AST преобразуется в структуры данных, более удобные для анализа. Под внутренним представлением мы будем понимать именно эти структуры.

3.3 Внутреннее представление

На основе AST каждого входного файла, статическим анализатором строится внутреннее представление. Структуры данных, образующие ВП, можно разделить на две группы:

1. Основные. Основным представлением программы являются управляющие графы (УГ) для всех ее блоков (к блоками здесь также относятся и тела методов, помимо просто блоков, ввиду их одинаковой семантики) и кода верхнего уровня всех файлов.
2. Вспомогательные.
   (a) Внутрипроцедурная информация об использованиях и определениях переменных (DU&UD chains).
   (b) Индексы. Индексы хранят глобальную информацию о местах определений и присваиваний переменным, а также местах вызова и определения методов.

Рассмотрим каждую из структур более подробно.

3.3.1 Управляющие графы

Управляющие графы строятся для следующих элементов программы:

- верхнеуровневый код файлов
- код методов
- код блоков (замыканий)

Если вернуться к представлению в виде AST, то будет видно, что структурно вторые и третьи графы могут быть вложены друг в друга, а также все они обязательно вложены в графы первого вида. Информация о том, где именно в файле находится определение метода или блока, является важной. Более конкретно, важно, имея, например, только УГ замыкания эффективно определять, в каком контексте оно было определено. Поэтому введем понятие *контекстного управляющего графа* (КУГ): пары вида (G, v), где G — собственно граф, а v — вершина какого-то другого графа, имеющая параметром данный (вершины УГ будут определены ниже).

Для того, чтобы описать устройство УГ, нам нужно описать множество вершин V и множество ребер E этих графов.

Вершинами УГ будут являться пары вида (c, G), где c — какая-то *инструкция* языка, а G — КУГ, содержащий данную вершину. Заметим, что в нашем случае управляющие графы обходятся без базовых блоков, храня непосредственно инструкции.

Определим *инструкции* возможные в КУГ. Каждая инструкция может иметь некоторые параметры (например, КУГ определяемого метода). Эти параметры будем называть также и параметрами соответствующей вершины графа.

Параметрами могут являться:

- Переменная: x (локальная), или $@x$ (поле), или $@@x$ (класса), или $\$x$ (глобальная), или X (константа), или $self$;
- Литерал: целое число, строка, массив вида $[v_1, ..., v_n]$ (где v_i — либо другой литерал, либо переменная), хэш-таблица вида $\{k_1 => v_1, ..., k_m => v_m\}$ (где k_i, v_i— какие-либо литералы или переменные);
- Объявляемые параметры: $a_1, \ldots, a_n, [o_1, \ldots, o_m, [r, [b]]]$, где a_i, o_j, r, b — какие-то имена, соответствующие обязательным агрументам, опциональным, массиву с остальными аргументами и блоковому аргументу. o_j, r, b могут отсутствовать в последовательности, заданной квадратными скобками;
- Замыкание (блок): $(CFG, params)$ — пара состоящая из КУГ и объявляемых параметров;
- Вызов
  - вида $var.methodName(v_1, \ldots, v_n, closure)$, где var, v_i — переменные, а $closure$ — замыкание или переменная (может отсутствовать);
  - вида $yield(v_1, \ldots, v_n)$, где v_i — переменные.

Теперь мы можем, наконец, определить набор инструкций:

- Присваивание: $var = rhs$, где var — переменная, rhs — литерал, переменная или вызов;
- Проверка условия: $if(var)$, где var — переменная;
- Определение обычного метода: $InstDef(var, name, CFG, params)$, где var — переменная равная классу/модулю у которого определяется метод, $name$ — имя метода, $params$ — объявляемые аргументы, CFG — КУГ с содержимым метода;

- Определение синглетон метода: $SinglDef(var, name, CFG, params)$, где var — переменная, $name$ — имя метода, $params$ — объявляемые аргументы, CFG — КУГ с содержимым метода;
- Возврат из метода: $Return(var)$, где var — переменная;
- Примешивание модуля: $Include(var, var_m)$, где var — переменная равная классу, а var_m— переменная или литерал, означающие модуль.
- Включение файла: $Require(v)$, где v — переменная или литерал с именем включаемого файла.
- Управляющие инструкции: $break$, $next$, $redo$.
- Вспомогательные инструкции: ВХОД и ВЫХОД. В каждом графе существуют в единственном экземпляре и обозначают входную и выходую вершины соответственно.

Необходимо прокомментировать соответствие определенного набора с командами и конструкциями языка. Во-первых, среди приведенных инструкций отсутствуют инструкции определения класса или модуля. Это вызвано тем, что в языке Ruby определение класса или модуля фактически равно присваиванию константе с именем класса экземпляра класса $Class$ или $Module$ соответственно. Поэтому определение класса заменяется на присваивание константе, а команды находившиеся внутри определения, оказываются на уровень выше.

Во-вторых, нужно заметить, что среди инструкций нет вызовов. Вызовы, стоящие в исходной программе “просто так”, преобразуются в присваивания новым вспомогательным переменным. Это сделано для удобства определения возвращаемых значений методов/замыканий, которые могут быть равны последней выполненной команде.

Затем, так как среди инструкций нет циклов, то циклы в программе преобразуются контур из условного оператора и тела цикла с выходящим ребром соответствующим $else$.

Наконец, заметим, что практически все инструкции имеют параметрами либо переменные либо литералы. Таким образом, вложенные вызовы и сложные выражения будут развернуты в несколько последовательных инструкций с использованием новых вспомогательных переменных.

Про ребра УГ нужно сказать, что они соотвествуют естественному порядку исполнения и легко могут быть построены читателем.

Рис. 11 демонстрирует пример управляющего графа для программы из листинга 4.

Листинг 4

```
require 'socket'
module Boz
end
class Foo
        def bar(x)
                if isGood(x)
                        y = Foo.zzz(x)
                else
                        puts "oops"
                end
        end
        def self.zzz(x)
                56
        end
        include Boz
end
$f = Foo.new
```

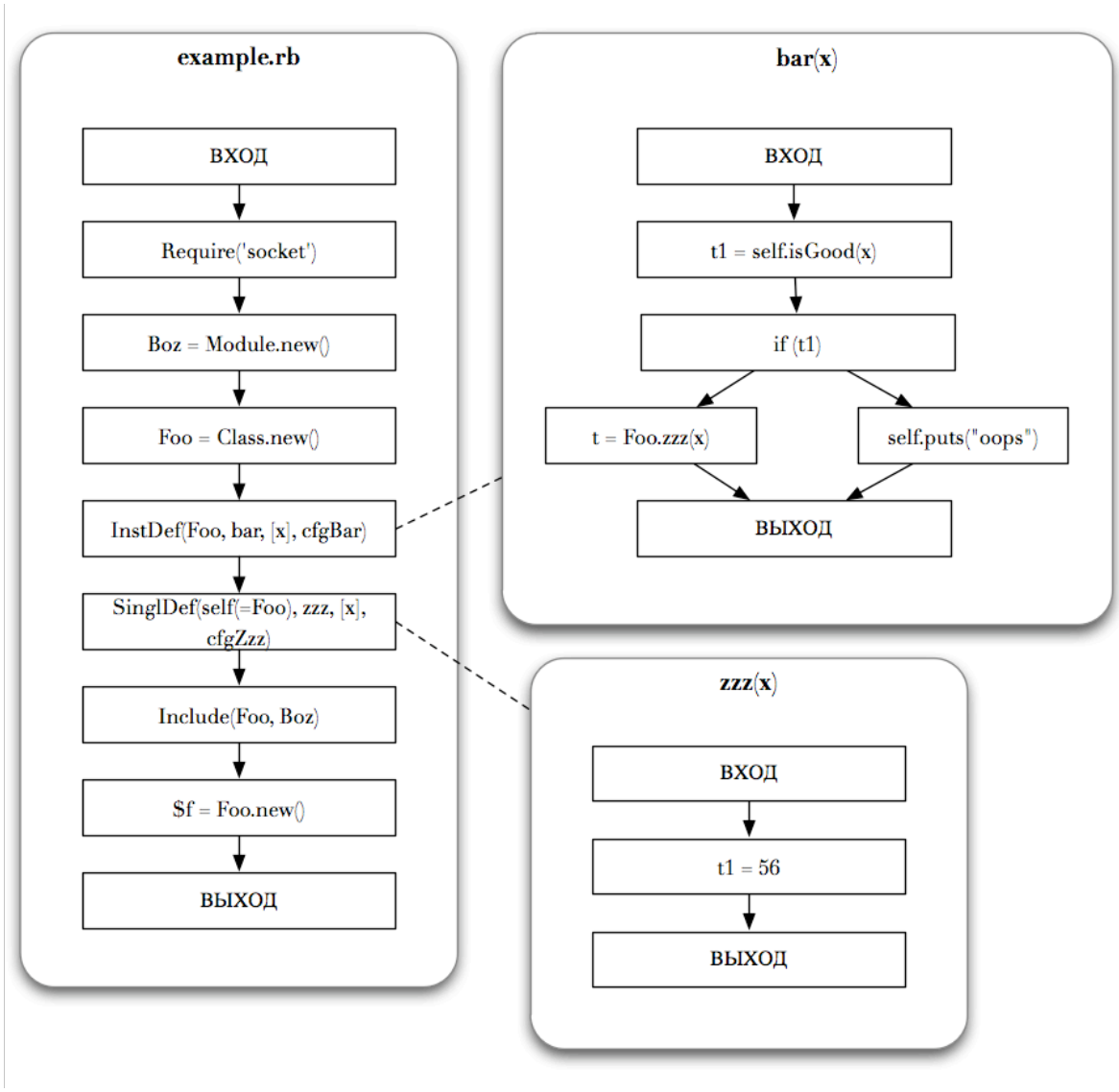


Рис. 11: КУГ для программы из листинга 3

3.3.2 Использования и определения переменных

Будем говорить, что инструкция v *использует* переменную var, если выполнено хотя бы одно из условий:

- v содержит переменную var как параметр;
- v является вызывом, и КУГ передаваемого замыкания содержит инструкцию, использующую var.

Будем говорить, что инструкция v *определяет* переменную var, если инструкция является присваиванием этой переменной.

Таким образом, для каждой переменной var можно обозначить множества $Def(var) \subseteq V$ и $Use(var) \subseteq V$ — множества инструкций определяющих и использующих переменную var.

Пусть $d \in Def(var)$. Тогда определим *множество использований* данного *определения* следующим образом:

$$Use_d = \{u \in Use(var) | \exists p, p \in Paths(d, u) \text{ и } p \cap Def(var) = \{d\}\}$$

, где $Paths(d, u)$ — множество всех путей от d до u в УГ.

Пусть теперь $u \in Use(var)$. Определим *множество определений достигающих использование* u следующим образом:

$$Def_u(var) = \{d \in Def(var) | \exists p, p \in Paths(d, u) \text{ и } p \cap Def(var) = \{d\}\}$$

Множества $Def_u(var)$ и Use_d являются необходимыми и крайне часто используемыми при внутрипроцедурном анализе. Поэтому имеет смысл предпросчитывать их до начала работы анализа, сразу после построения УГ. Сделать это можно с помощью алгоритма, приведенного в листинге 5.

Листинг 5 Построение множеств Def_u и Use_d

```
procedure UpdateReachingDefs(G, v, reachingDefs)
        rd = ReachingDefs[v] + reachingDef
        if visited[v] && rd == reachingDefs
                return

        visited[v] = true

        if v defines var then
                reachingDefs[var] = v

        for u in Succ(v) do
                UpdateReachingDefs(G, u, reachingDefs)

procedure BuildDefUse(G)
        for v in V(G) do
                rd = reachingDefs[v]
                for var in VarsUsedBy(v)
                        if rd[var]
                                Use[rd[var]] += v
                                Def[v, var] += rd[var]
```

Данный алгоритм строит множества $Def_u(var)$ и Use_d. При этом время его работы можно оценить как $O(|V| * A)$, где A — количество обратных двуг в УГ.

3.3.3 Индексы

Для оптимизации поиска в процессе глобального анализа, имеет смысл хранить следующие структуры данных:

- Для каждого имени глобальной переменной, поля, переменной класса и константы — все присваивания в программе переменной с таким именем.
- Для каждого имени глобальной переменной, поля, переменной класса и константы — все инструкции в программе, использующие перменные с таким именем.
- Для каждого имени метода список инструкций в программе, которые вызывают метод с таким именем.

- Для каждого имени метода список инструкций в программе, определеляющих метод с таким именем.

Данные списки строятся интуитивно понятным образом.
// дописать про инкрементальное обновление

3.4 Запросы

// TODO

3.4.1 Запросы о значениях

// TODO

3.4.2 Запросы об использованиях

// TODO

3.4.3 Запросы о точках вызова

// TODO

3.4.4 Запросы о вызываемом коде

// TODO

4 Результаты и эксперименты

// TODO

5 Заключение

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

Была произведена следующая работа:

- разработан, описан и реализован модифицированный алгоритм DDP с поддержкой кеширования между запросами и инкрементальным обновлением результатов;
- исследован и описан язык Ruby;
- в рамках проекта DLTK Eclipse Foundation был реализован упрощенный статический анализатор для языка Ruby, а также алгоритм DDP без поддержки инкрементальности;
- разработан, описан и реализован основанный на модифицированном DDP статический анализ для языка Ruby, позволяющий вычислять типы.

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 анализа к задачам биоинформатики, социологии, экономики.

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 приемлемый уровень точности анализа. Однако были выявлены не учтенные изначально особенности языка, требующие 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 делающей работу проще для 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.

Список литературы