Санкт-Петербургский государственный университет

Направление фундаментальной информатики и информационных технологий
Профиль математического и программного обеспечения вычислительных машин, комплексов и компьютерных сетей

Иванов Андрей Васильевич

Особенности вычисления семантики встроенных языков

Магистерская диссертация

Научный руководитель:
к. ф.-м. н., доцент Булычев Д. Ю.

Рецензент:
Ведущий программист ООО ПитерСофтвареХаус Полозов В. С.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Main Field of Study of Fundamental Computer Science and Information Technologies
Area of Specialisation of Software Computers, Complexes and Networks

Andrei Ivanov

Peculiarities of semantic computation of string-embedded languages

Master's Thesis

Scientific supervisor:
PhD Dmitri Boulytchev

Reviewer:
Senior software developer at PiterSoftwareHouse Victor Polozov

Saint-Petersburg
2016

Оглавление

Введение **4**

1. Постановка задачи **6**

2. Обзор **7**

2.1. Подходы к статическому анализу встроенных языков 7

2.2. Существующие методы и инструменты 7

2.3. Проект YaccConstructor 8

2.4. Лес разбора SPPF 10

2.5. Анализ потоков данных и граф потока управления 11

3. Построение графа потока управления по лесу разбора **14**

4. Поиск хорошо определённых переменных в динамически формируемых программах **21**

5. Реализация **28**

5.1. Архитектура 28

5.2. Детали реализации поиска хорошо определённых переменных 29

Заключение **31**

Список литературы **32**

Введение

Существует подход к программированию, в котором одна программа динамически формирует другую программу на некотором языке и передаёт её на выполнение в соответствующее окружение. При этом генерируемый код собирается из строк таким образом, чтобы в момент выполнения результирующая строка представляла собой корректную программу. Далее генерируемый код будем называть *встроенным кодом*, язык, на котором написан встроенный код, – *встроенным языком*, получающиеся программы – *динамически формируемыми программами*. В качестве примера можно привести формирование HTML-страниц в PHP-приложениях, SQL-запросы к базам данных в приложениях на C#, C++, Java. Программы, написанные с использованием такого подхода, обладают высокой производительностью, являются гибкими и выразительными. Благодаря этому подход к программированию, связанный с использованием встроенных языков, получил широкое распространение.

В настоящее время при создании ПО активно используются интегрированные среды разработки (Integrated Development Environment, IDE). Их основная цель – повысить продуктивность работы программиста. Эта цель достигается за счёт того, что IDE интегрирует различные утилиты – компилятор, отладчик, текстовый редактор и т. п., – являясь единственной средой, с помощью которой ведётся разработка. Это избавляет разработчика от необходимости каждый раз вручную переключаться между несколькими программами, что положительно сказывается на его производительности. Также IDE может применять утилиты параллельно и комбинированно. Например, совместная работа текстового редактора и компилятора позволяет реализовать функции автодополнения и навигации по коду, а также статический поиск ошибок, возможность осуществления рефакторинга кода и т. п.

Однако при работе с приложениями, которые используют встроенные языки, трудно реализовать функции IDE, которые указаны выше. Это связано с тем, что компилятор воспринимает встроенный код как обычную строку в основном коде, т. е. встроенный код не анализируется статическим образом. Отсутствие статических проверок ведёт к тому, что об ошибках во встроенном коде станет известно только во время выполнения программы. В некоторых приложениях такая ситуация может оказаться критичной, поскольку ошибка может привести к нарушению целостности данных. Помимо этого в случае, когда встроенный код формируется при помощи условных операторов или циклов основного языка, то его (код) сложно понимать и отлаживать.

Инструмент, осуществляющий статический анализ встроенного кода, позволит избежать описанных выше проблем. Разработка такого инструмента ведётся в рамках проекта YaccConstructor [11, 14, 15, 16], одним из направлений исследований которого является статический анализ встроенных языков. Основной идеей проекта яв-

ляется предположение, что если есть программа, содержащая в себе встроенный код, то в большинстве случаев из неё можно извлечь достаточно информации для проведения статического анализа динамически формируемых программ [14]. В рамках проекта уже проведены исследования в области лексического и синтаксического анализа встроенных языков, предложены и апробированы соответствующие алгоритмы [13, 17]. Однако в проекте отсутствуют средства семантического анализа, которые были бы полезены при решении многих классических задач статического анализа.

1. Постановка задачи

Целью данной работы является изучение особенностей семантического анализа динамически формируемого кода в рамках проекта YaccConstructor. Для её достижения были поставлены следующие задачи.

- Разработать алгоритм построения графа потока управления для динамически формируемого кода, который будет принимать на вход лес разбора.
- Решить задачу поиска хорошо определённых переменных при анализе встроенных языков.
- Реализовать предложенные алгоритмы.

2. Обзор

2.1. Подходы к статическому анализу встроенных языков

Существует два подхода для проведения статического анализа встроенных языков. Первый из них заключается в проверке включения языков, и его единственная цель — дать ответ на вопрос, включается ли язык, который порождается программой, в эталонный язык [4, 8]. Предполагается, что эталонный язык описан пользователем с помощью грамматики или простым перечислением строк, которые он ожидает увидеть в качестве результата работы программы.

Другой подход заключается в последовательном применении алгоритмов аппроксимации, лексического и синтаксического анализа [11]. Вначале производится поиск по исходному коду основной программы возможных точек интереса, или хотспотов (hospot), в которых осуществляется исполнение или передача встроенного кода выделенной компоненте. Точки интереса могут быть заданы как автоматически (если есть информация об используемом фреймворке), так и вручную пользователем. В точках интереса происходит вычисление множества значений динамически формируемого выражения, а также последующая аппроксимация полученного множества. Далее производится лексический анализ, который по автомату над множеством строк строит автомат над множеством лексем. Следующий шаг — синтаксический анализ автомата над токенами, в результате чего строится лес разбора. Преимуществом данного подхода является его гибкость: каждый этап выполняется независимо друг от друга, что позволяет заново использовать существующие реализации других шагов.

2.2. Существующие методы и инструменты

Существует ряд решений, нацеленных на работу со встроенными языками.

- В [7] описан алгоритм статического анализа динамически формируемых строковых выражений на примере статической проверки корректности динамически генерируемого HTML в PHP-программах. На вход платформа принимает data-flow уравнения, полученные при анализе основного кода, LALR(1)-таблицу и правила вычисления семантики. За счёт правил вычисления семантики инструмент может находить такие ошибки, как неуникальное имя идентификатора, ссылки на несуществующий идентификатор и т.п. Несмотря на то что приводятся результаты экспериментов, реализации данного алгоритма на момент написания данной работы в открытом доступе не существовало.

- В [5] предложен метод анализа строковых значений, базирующийся на абстрактной интерпретации. Для основных строковых операций описаны несколько абстрактных доменов, таких как аппроксимация строки её возможным префиксо-

м/суффиксом, аппроксимация строки символами, из которых она должна или может быть составлена (character inclusion) и т. п. Такой подход позволяет гарантировать, например, что все встроенные SQL-запросы начинаются со слова "`SELECT`", а заканчиваются знаком "`;`".

- Alvor[1] [2, 6] – плагин для среды разработки Eclipse, предназначенный для статической проверки корректности кода на SQL, встроенного в Java. Плагин выполняет межпроцедурный анализ кода, поддерживает обработку условных операторов, операций конкатенации и присваивания строк, сообщает о лексических и синтаксических ошибках. К недостаткам плагина можно отнести отсутствие поддержки циклов и строковых операций, отличных от конкатенации.

- IntelliLang[2] – плагин для среды разработки IntelliJ IDEA, который, в частности, расширяет её возможности в области работы со встроенными языками. Плагин осуществляет подсветку и автодополнение встроенного кода, а также для некоторых языков (JavaScipt, XML) находит ошибки. Особенностью работы плагина является то, что он не осуществляет поиск строк со встроенным кодом: программист должен сам указывать, какие строки нужно анализировать.

- PhpStorm[3] – среда разработки для создания Web-приложений на PHP. Программы на PHP часто содержат встроенный код на JavaScript, CSS и HTML, а также на SQL, и данная среда разработки предоставляет подсветку и автодополнение. Однако PhpStorm не обрабатывает строковые операции, в том числе и конкатенацию.

- Varis [9] – плагин для Eclipse, который для языка PHP осуществляет поддержку вложенных HTML, CSS и JavaScript. Varis предлагает следующую функциональность: подсветка встроенного кода, автодополнение, переход к объявлению (jump to declaration). Для встроенного JavaScript также предоставляется функция построения графа вызовов (call graph).

Описанные инструменты поддерживают ограниченный набор встроенных языков, плохо расширяются как в плане поддержки новых встроенных языков, так и в плане добавления новой функциональности. Многие инструменты не осуществляют семантического анализа встроенного кода.

2.3. Проект YaccConstructor

Исследовательский проект YaccConstructor [16] выполняется в лаборатории языковых инструментов JetBrains на математико-механическом факультете СПбГУ. В его

[1] Сайт проекта Alvor: `https://bitbucket.org/plas/alvor`
[2] Сайт IntelliLang: `https://www.jetbrains.com/idea/help/intellilang.html`
[3] Сайт PHPStorm: `http://www.jetbrains.com/phpstorm/`

рамках проводятся эксперименты в области лексических и синтаксических анализаторов, а также разрабатывается платформа для статического анализа встроенного кода. В частности, в инструменте уже реализованы алгоритмы аппроксимации встроенного кода [18], его лексического [17] и синтаксического анализа [13].

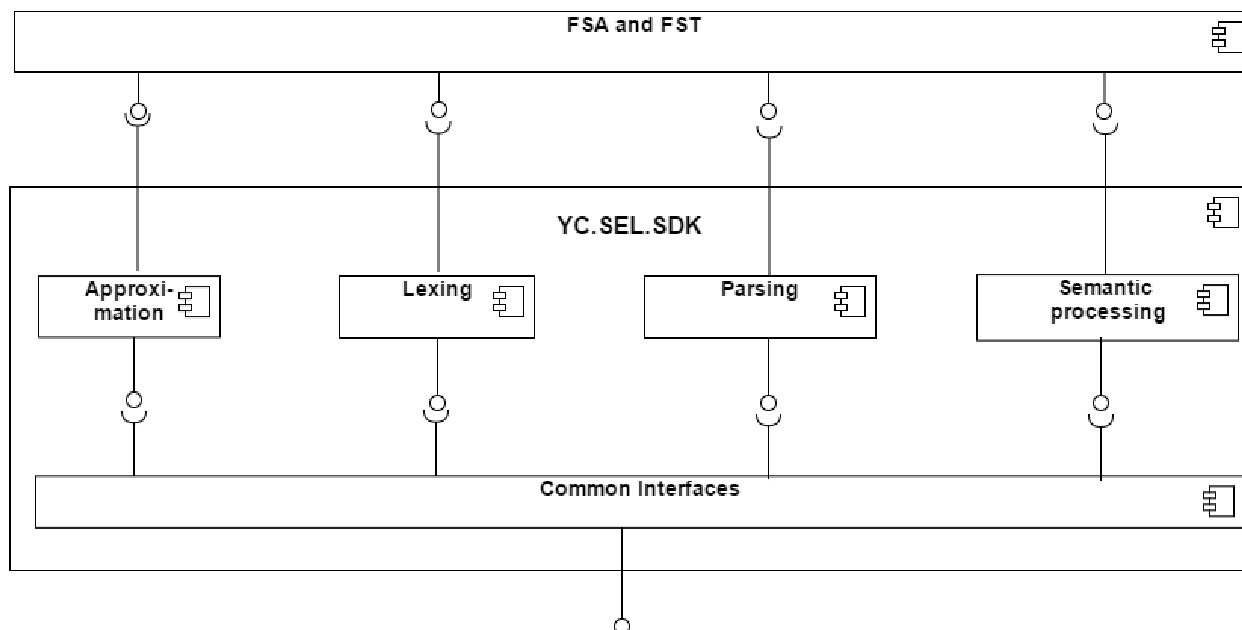


Рис. 1: Диаграмма компонентов в YaccConstructor (рисунок взят из работы [14] и переделан).

Диаграмма компонентов и диаграмма деятельности основного продукта данного проекта представлена на рис. 1 и 2. Компонент **Approximation** отвечает за аппроксимацию динамически формируемого кода. Он принимает на вход дерево разбора основного языка и осуществляет поиск точек интереса. Если точки интереса найдены, то для каждой из них строится регулярная аппроксимация множества возможных значений динамически формируемого выражения. Компонент **Lexing** осуществляет лексический анализ, переводя автомат над символами в автомат над лексемами. Компонент **Parsing** проводит синтаксический анализ полученного после лексического анализа автомата, возвращая в качестве результата лес разбора (Shared Packed Parse Forest, SPPF) [10], компактно представляющий все возможные варианты разбора входной цепочки. Работа, представленная в данной магистерской диссертации, велась в рамках компонента **Sematic Processing**.

Также в YaccConstructor имеется надстройка, позволяющая создавать расширения для ReSharper[4] с целью поддержки встроенных языков.

[4]ReSharper – расширение для Microsoft Visual Studio, предоставляющее пользователю широкий набор дополнительных анализов кода

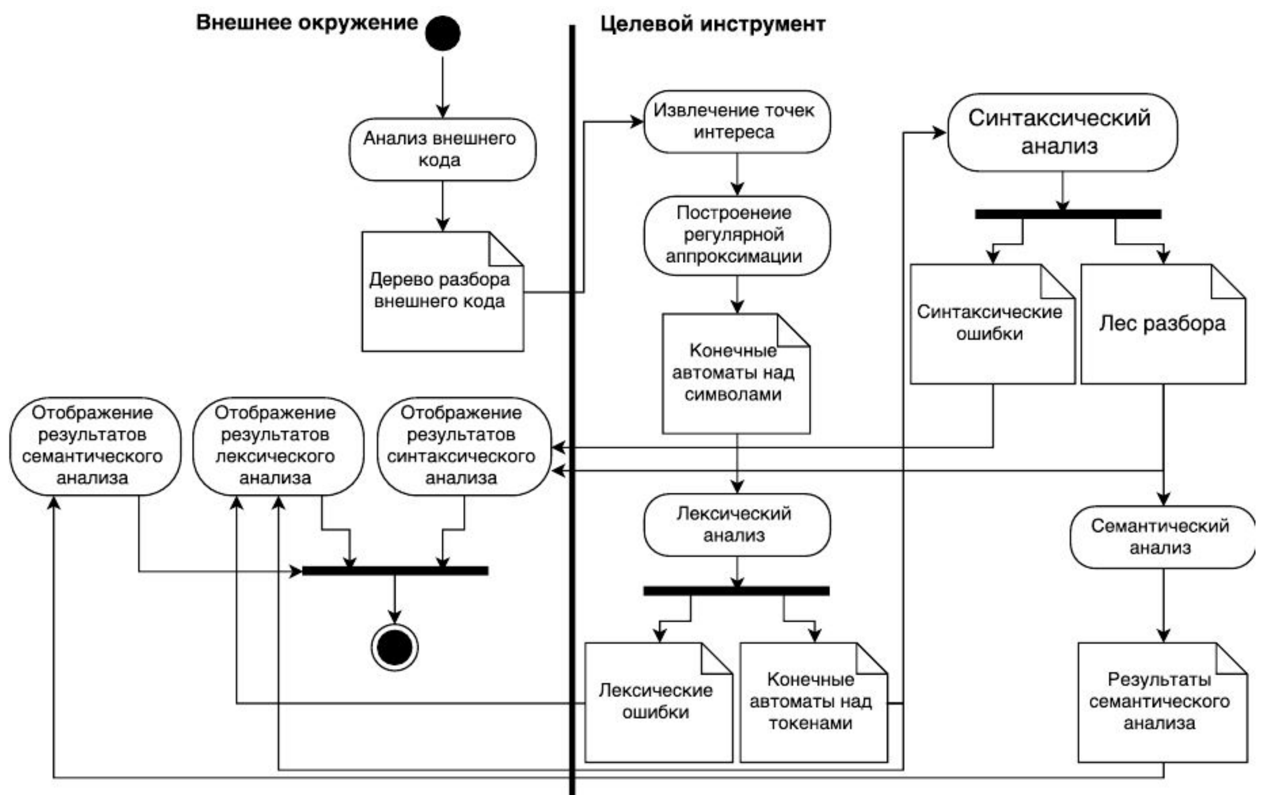


Рис. 2: Диаграмма деятельности обработки встроенных языков в YaccConstructor (рисунок взят из работы [14]).

2.4. Лес разбора SPPF

В инструменте YaccConstructor результатом работы синтаксического анализа является сжатый лес разбора (Shared Packed Parsing Forest, SPPF) [10], компактно представляющий все возможные варианты разбора входной цепочки. SPPF является ориентированным графом и обладает следующей структурой.

1. Корень – вершина, не имеющая входящих дуг. Соответствует стартовому нетерминалу грамматики.

2. Терминальные вершины, не имеющие исходящих дуг, соответствуют либо терминалам грамматики, либо деревьям вывода пустой строки ϵ.

3. Нетерминальные вершины являются корнем дерева вывода некоторого нетерминала грамматики; только вершины-продукции могут быть непосредственно достижимы из таких вершин.

4. Вершины-продукции, представляющие правую часть правила грамматики для соответствующего нетерминала. Вершины, непосредственно достижимые из них, упорядочены и могут являться либо терминальными, либо нетерминальными вершинами.

Далее терминальные и нетерминальные вершины будем называть вершинами грамматики.

На рис. 3 представлен пример леса разбора для следующей грамматики:

```
start := x
x := A
x := B C
```

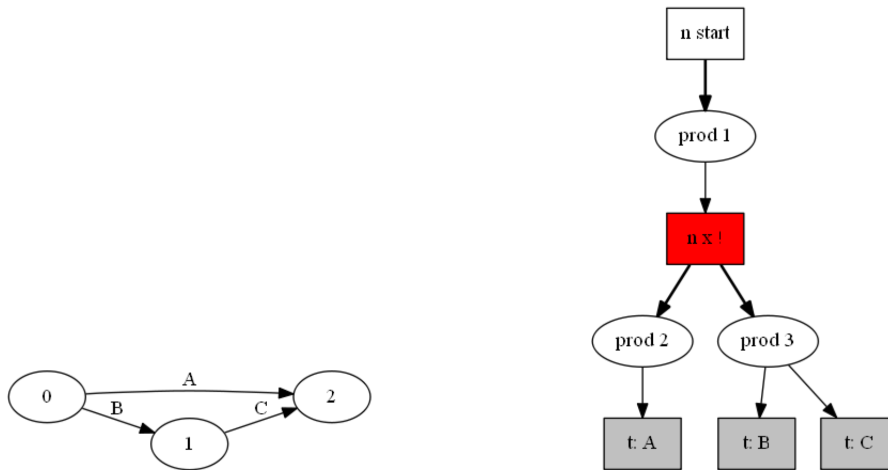


Рис. 3: Пример входного графа и соответствующего леса разбора.

Вершины прямоугольной формы являются вершинами грамматики (терминальные вершины выделены серым фоном), вершины овальной формы – вершинами-продукциями.

2.5. Анализ потоков данных и граф потока управления

В задачу анализа потоков управления (control flow analysis) входит определение свойств передачи управления между операторами программы. Проверка многихсвойств этого вида необходима для решения задач оптимизации, преобразований программ и т. д. Для этого необходимо представить анализируемую программу некоторым графом, а сами задачи сформулировать в теоретико-графовой форме.

Обычно таким графом является граф потока управления (Control Flow Graph, CFG) [1], представляющий собой множество всех возможных путей исполнения программы. Формально граф потока управления G является четвёркой $(V,\ E,\ start,\ stop)$, где:

- $(V,\ E)$ является ориентированным графом;
- $start \in V$, $stop \in V$;
- у вершины *start* нет входящих дуг, у вершины *stop* нет исходящих дуг;

- произвольная вершина принадлежит хотя бы одному пути из вершины *start* в вершину *stop*.

На рис. 4 приведён пример графа потока управления. Вершиной *start* является вершина, помеченная `x = 1`, вершиной *stop* – вершина, помеченная `w = x + y`.

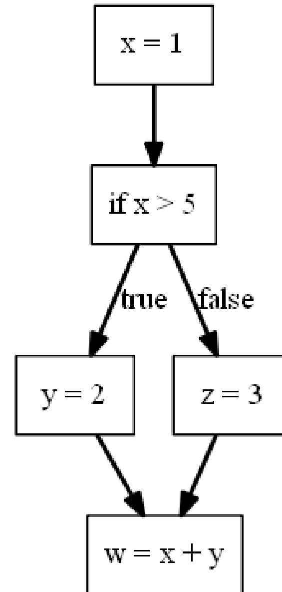


Рис. 4: Граф потока управления.

Общий подход к решению задач анализа потоков данных описывается следующим образом. Формируются следующие структуры данных.

- Полурешётка L конечной высоты. Множество L с операцией $\wedge$ является полурешёткой, если для $\forall\, x, y, z \;\in\; L$ верно:
  - $x \wedge x \;=\; x$ (идемпотентность),
  - $x \wedge y \;=\; y \wedge x$ (коммутативность),
  - $(x \wedge y) \wedge z \;=\; x \wedge (y \wedge z)$ (ассоциативность).

  Полурешётка L представляет собой множество "фактов" о программе, а решёточная операция ($\wedge$) описывает получение общей части нескольких решений.

- Граф потока управления G.

- Разметка $before\colon V \;\to\; L$. Описывает решение задачи анализа потоков данных до исполнения операторов в данной вершине графа.

- Разметка $after\colon V \;\to\; L$. Описывает решение задачи анализа потоков данных после исполнения операторов в данной вершине графа.

- Набор монотонных на L потоковых функций $f_v \colon L \to L$ для каждой вершины v графа G. Потоковые функции переводят "факты", известные непосредственно до исполнения операторов в вершине v, в факты, известные непосредственно после вершины v.

Тогда решением задачи анализа потоков данных является пара наименьших разметок $before$, $after$, являющихся решением системы уравнений.

$$\left\{ \begin{aligned} before(v) &= \bigwedge_{w \in pred(v)} after(w) \\ after(v) &= f_v(before(v)) \end{aligned} \right\}_{v \in V}$$

Данную систему уравнений можно интерпретировать следующим образом. Решением задачи до вершины является общая часть решений всех предшественников данной вершины, а после нее – применение к этой общей части потоковой функции, ассоциированной с данной вершиной.

Примером задачи, которую можно сформулировать и решить таким образом, является поиск хорошо определённых переменных (definite assignment analysis). Переменная x считается хорошо определённой в данной точке программы, если на всех путях от начала программы до данной точки содержится определение переменной x.В данной задаче полурешёткой L является множество всех переменных с операцией пересечения. Потоковые функции $f_v(X)$ возвращают X, если в вершине v не происходит операции присваивания, и возвращают $X \cup x$, если в вершине v происходит присваивание в переменную x. Разметка $before(v)$ будет возвращать множество переменных, определённых непосредственно до вершины v, а $after$ – после.

3. Построение графа потока управления по лесу разбора

В данном разделе предложен алгоритм построения графа потока управления динамически формируемых программ по лесу разбора. Обычно в классических алгоритмах анализа потоков данных граф строится по одному дереву разбора [1]. Однако в ситуации с анализом встроенных языков результатом синтаксического анализа является лес разбора [13, 14]. Подход, основанный на том, чтобы извлекать из леса по одному дереву, по которому строить граф потока управления, слабо применим в данной ситуации, поскольку лес может содержать бесконечное число деревьев. Листинг 1 представляет фрагмент кода, который приводит к такой ситуации.

```
1   string query = "x = 1;";
2   while(cond)
3   {
4       query += "x += 1;";
5       //other code
6   }
7   //other code
8
```

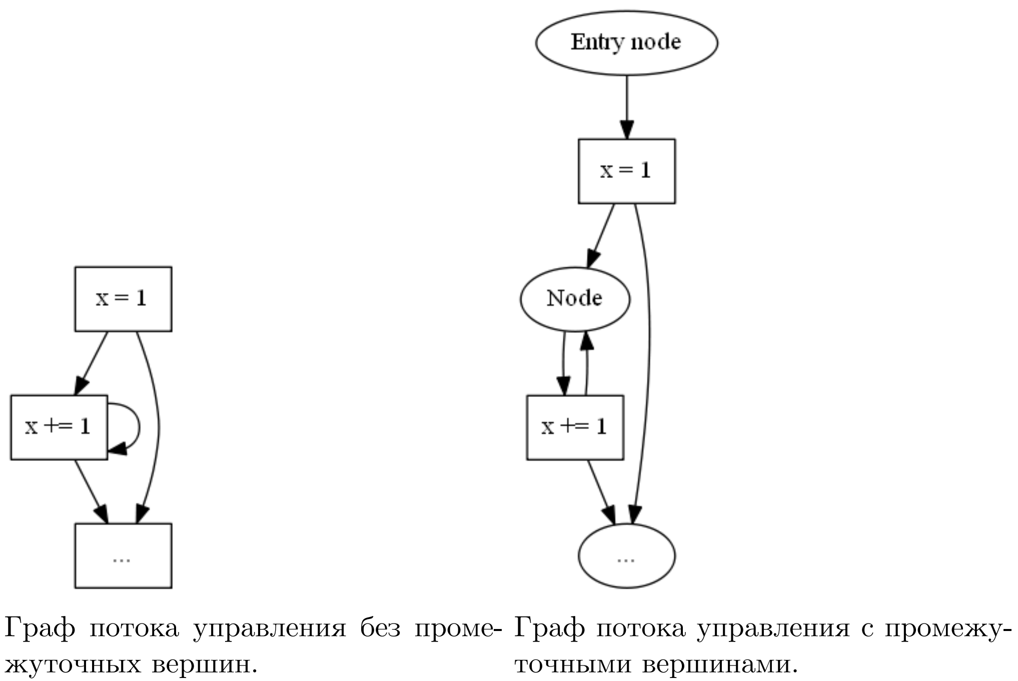
Листинг 1: Лес разбора программы содержит бесконечное число деревьев.

В этом примере встроенный код содержится в переменной `query`. В зависимости от значения параметра `cond` во встроенном коде будет содержаться выражение `x += 1;` n раз, где n принимает значения от 0 до бесконечности раз. Из-за этого в лесе разбора встроенного языка будет содержаться бесконечное число деревьев, каждое из которых будет соответствовать ситуации, когда тело цикла выполнилось n раз.

Есть другой подход: строить граф потока управления сразу по лесу разбора, без извлечения деревьев. Граф потока управления, полученный таким способом, будет содержать в себе несколько графов потока управления, каждый из которых будет соответствовать какому-то дереву из леса разбора. В частности, для приведённого выше кода граф потока управления динамически формируемой программы мог бы выглядеть одним из двух способов, показанных на рис. 5.

Далее описан алгоритм, строящий по лесу разбора граф потока управления динамически формируемой программы (см. `Algorithm` 1 и `Algorithm` 2).

Алгоритм принимает на вход корень леса разбора и совершает обход в глубину, вызывая две взаимно-рекурсивные функции `HandleGrammarNode` и `HandleProductionNode`, которые обрабатывают вершины грамматики и вершины-продукции соответственно. При этом предполагается наличие вспомогательных функций-обработчиков, которые могут строить блок графа потока управления по поддеревьям специализированного вида. Например, функция `processAssignment` может принимать поддерево с корнем

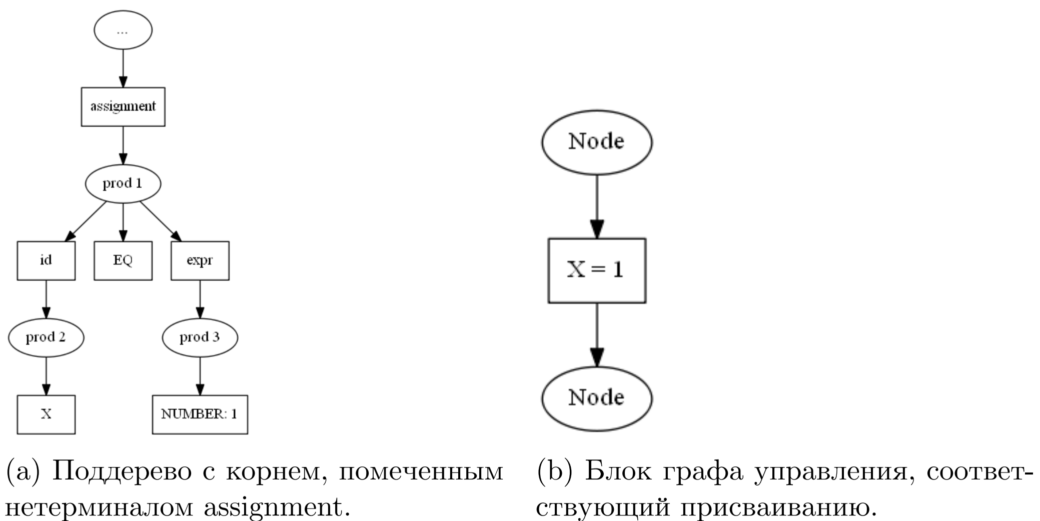


Граф потока управления без промежуточных вершин.

Граф потока управления с промежуточными вершинами.

Рис. 5: Два способа представления графа потока управления встроенных языков: без промежуточных вершин и с ними.

в нетерминальной вершине, помеченной `assignment`, и возвращать блок графа потока управления, соответствующий присваиванию (см. рис. 6). Таким образом, если в процессе обхода встретилась вершина `v`, для которой существует функция `f`, которая может эту вершину обработать, то происходит вызов функции `f` (см. строки 3-4 в функции `handleGrammarNode` в `Algorithm` 1).



(a) Поддерево с корнем, помеченным нетерминалом assignment.

(b) Блок графа управления, соответствующий присваиванию.

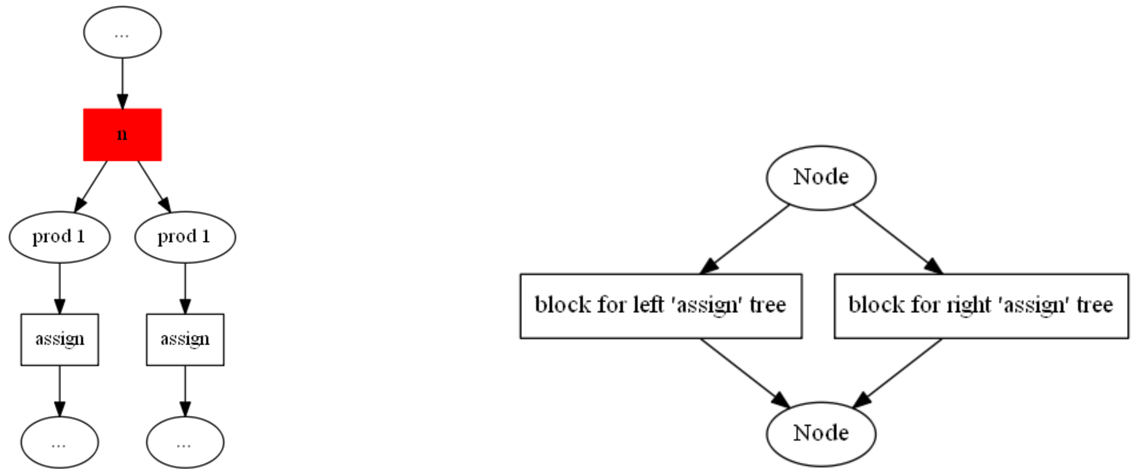
Рис. 6: Пример обработки поддеревьев.

Если же для рассматриваемой вершины подходящей функции-обработчика не нашлось, то алгоритм продолжит обход в глубину. При этом учитываются возможные

Algorithm 1 Алгоритм построения графа потока управления. Обработка вершин грамматики.

```
 1: function HandleGrammarNode(graph, node)
 2:     if handler-function for node exists then
 3:         block ← call corresponding handler-function
 4:         graph.Add(block)
 5:     else
 6:         commonStart ← graph.Current
 7:         endVertices ← new Set()
 8:         for child ∈ node.children do
 9:             graph.Current ← commonStart
10:             handleProductionNode(graph, child)
11:             endVertices.Add(graph.Current)
12:         commonEnd ← graph.CreateNewVertex
13:         replace all endVertices on commonEnd vertex
14:         graph.Current ← commonEnd
```

неоднозначности, из-за которых у нетерминала может быть несколько порождающих деревьев (а значит, несколько непосредственных дочерних вершин). В таких случаях поддеревья должны быть обработаны независимо друг от друга, но при этом соответствующим им блокам добавляется по искусственной общей конечной вершине (см. строки 6-14 в функции `HandleGrammarNode` в `Algorithm` 1). Например, на рис. 7 у нетерминала `n` имеется несколько порождающих деревьев. В таком случае алгоритм обрабатывает каждое поддерево по отдельности, после чего назначает полученным блокам общую стартовую и общую конечную вершины.



(a) Поддерево с неоднозначностью. (b) Блоки графа потока управления.

Рис. 7: Пример обработки неоднозначностей.

В отличии от функции `HandleGrammarNode`, обрабатывающей только вершины грамматики, функция `HandleProductionNode` обрабатывает вершины-продукции (см. `Algorithm` 2)

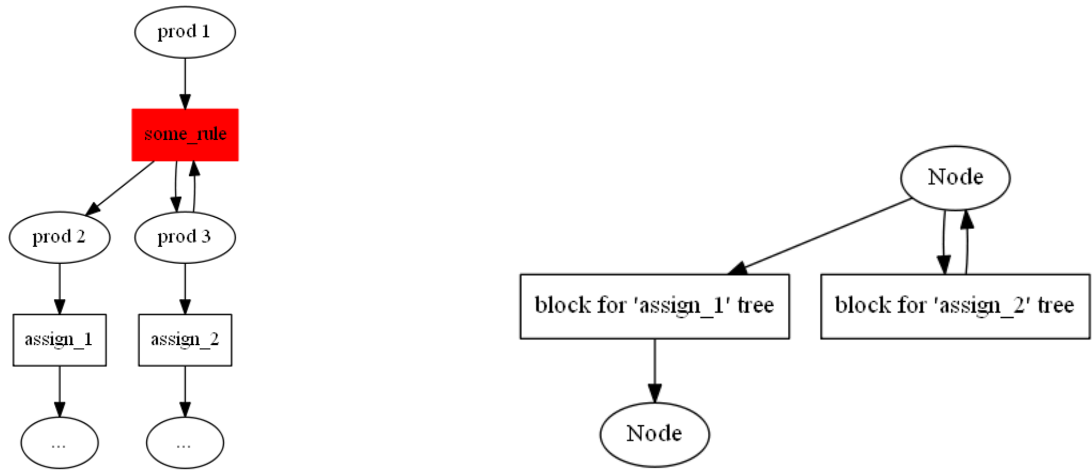
Её основная основная задача – избегать повторной обработки поддеревьев, для которых уже был построен блок графа потока управления. Для этого функция запоминает состояния вершин-продукций. Каждая вершина-продукция может находиться в одном из трёх состояний, описанных ниже.

- `New`. Если вершина находится в состоянии `New`, это означает, что она ещё не обрабатывалась. В начале работы алгоритма все вершины-продукции находятся в этом состоянии.
- `InProgress`. Если вершина находится в этом состоянии, то её разбор стартовал, но ещё не завершён.
- `Processed`. Обработка вершин, находящихся в данном состоянии, уже завершена. С данным состоянием ассоциированы две вершины: одна отвечает за начало блока, а другая – за конец блока.

Algorithm 2 Алгоритм построения графа потока управления. Обработка вершин-продукции

```
 1: function HandleProductionNode(graph, node)
 2:     if node.state is new then
 3:         start ← graph.Current
 4:         node.state ← InProgress(start)
 5:         for all child ∈ node.Children do
 6:             HandleGrammarNode(graph, child)
 7:         node.state ← Processed(start, graph.Current)
 8:     else if node.state is inProgress(start) then          ▷ Cycle in SPPF is detected
 9:         graph.AddEdge(Epsilon, start)
10:         endVertex ← try find last vertex
11:         if endVertex is not null then
12:             node.state ← Processed(start, endVertex)
13:             graph.Current ← endVertex
14:     else                                     ▷ node.state is Processed(start, target)
15:         graph.AddEdge(Epsilon, start)
16:         graph.Current ← target
```

Наличие состояний у вершин-продукций помогает при обработке циклов внутри леса разбора. Если в функции `HandleProductionNode` производится обработка вершины, находящейся в состоянии `InProgress`, то в этот момент обнаружен цикл. В таких ситуациях алгоритм предпринимает попытку найти вершину, которая будет являться выходом из цикла. Если такая вершина найдена, то она объявляется конечной, а вершина-продукция считается обработанной (см. строки 9-15 в `HandleProductionNode` в `Algorithm` 2). В противном случае нужно продолжить обход.



(a) Поддерево с циклом. (b) Блоки графа потока управления.

Рис. 8: Пример обработки циклов.

Рассмотрим процесс поиска вершины, являющейся выходом из цикла, более подробно на примере вышеприведённого фрагмента леса разбора (рис. 8). Предположим, вершины-продукция `prod 1`, `prod 3` находятся в состоянии `InProgress`, вершина-продукция `prod 2` – в состоянии `Processed`. Алгоритм находится в вершине-продукции `prod 3`. Поскольку данная вершина находится в состоянии `InProgress`, то он понимает, что обнаружил цикл в лесе разбора, и пытается найти выход из цикла. Для этого осуществляется просмотр состояний всех вершин, достижимых из данной вершины.

- Если среди них найдутся вершины с состоянием `New` (то есть вершины, которые ещё не обрабатывались), то вернуть *null* в качестве концевой вершины.

- Иначе если среди них найдутся вершины с состоянием `Processed`то предпринять следующие действия. Если такая вершина всего одна, то вернуть в качестве результата её концевой узел. Если таких вершин несколько, то создать общую концевую вершину `v`, добавить дуги из найденных концевых узлов вершин с состоянием `Processed` и вернуть `v` в качестве результата.

- Если не выполнено ни одно из описанных выше условий (т.е. все вершины находятся в состоянии `InProgress`), то вернуть в качестве концевой вершины начальную вершину.

В данном примере есть достижимая вершина-продукция с состоянием `Processed` – это вершина-продукция `prod 2`. Поэтому концевая вершина `prod 3` будет совпадать с концевой вершиной `prod 2`, что приведёт к блоку графа потока управления, который показан на рис. 8.

Описанный алгоритм обрабатывает только те неоднозначности в лесе разбора, которые возникают на стыке конструкций встроенного языка. При этом обработка

других неоднозначностей (т.е. *внутри* конструкций языка), делегируется вспомогательным функциям-обработчикам. На листинге 2 в правой части присваивания переменной `x` может оказаться как потенциально бесконечное выражение "y + 1 + 1 + ...", так и одна переменная `y` (в случае, если тело цикла ни разу не выполнялось). Непосредственной обработки такой ситуации не произойдёт, поскольку до этого алгоритм обнаружит нетерминальную вершину, помеченную нетерминалом `assignment`, и поручит обработку этого поддерева вспомогательной функции `processAssignment`.

```
1    string query = "x = y";
2    while(cond)
3    {
4        query += "+ 1;";
5        //other code
6    }
7    //other code
8
```

Листинг 2: Пример программы, порождающей бесконечные выражения

Ниже описан алгоритм 3, позволяющий обрабатывать такие ситуации. Его идея во многом сходна с идеей алгоритма построения графа потока управления. Разница в том, что данный алгоритм строит граф, содержащий множество возможных значений выражения, а не граф потока управления.

Algorithm 3 Алгоритм разбора выражений. Обработка вершин грамматики

1: **function** HandleGrammarNode(*graph*, *node*)
2:     **if** *node* is Terminal t **then**
3:         $newVertex \leftarrow$ graph.CreateNewVertex
4:         $current \leftarrow graph.Current$
5:         graph.AddEdge(Tag(t), *current*, *newVertex*)
6:         $graph.Current \leftarrow newVertex$
7:     **else if** *node* is nonterminal n **then**
8:         $commonStart \leftarrow graph.Current$
9:         $endVertices \leftarrow$ new Set()
10:         **for** $child \in node.children$ **do**
11:             $graph.Current \leftarrow commonStart$
12:             handleProductionNode(*graph*, *child*)
13:             endVertices.Add(*graph.Current*)
14:         $commonEnd \leftarrow$ graph.CreateNewVertex
15:         **for all** $vertex \in endVertices$ **do**
16:             graph.Add(Epsilon, *vertex*, *commonEnd*)
17:         $graph.Current \leftarrow commonEnd$

На вход алгоритм принимает граф нетерминальную вершину `exprRoot`. В ходе своей работы алгоритм совершает обход в глубину, начиная с вершины `exprRoot`, вызывая две взаимно-рекурсивные функции `HandleGrammarNode` (см. `Algorithm 3`)

и HandleProductionNode. Алгоритм, используемый в HandleProductionNode, полностью совпадает с Algorithm 2.

Функция HandleGrammarNode (Algorithm 3) добавляет новый токен в граф, если вершина оказалась терминальной (см. строки 3-6). В противном случае она продолжает обход. При этом также учитывается ситуация, когда нетерминал имеет несколько поддеревьев вывода. В таких случаях поддеревья должны быть обработаны независимо друг от друга, но при этом соответствующим им участкам графа добавляется по искусственной общей конечной вершине (см. строки 8-17 в функции HandleGrammarNode в Algorithm 3).

4. Поиск хорошо определённых переменных в динамически формируемых программах

На графе потока управления можно решать задачи классического статического анализа: поиск неопределённых переменных, поиск использований (find usages), переход к определению (go to definition) и т. п.

Рассмотрим в качестве примера задачу поиска хорошо определённых переменных (definite assignment analysis). В данной задаче переменная x считается хорошо определённой, если любому использованию x предшествует присваивание в переменную x какого-нибудь значения. В терминах графа потока управления это означает, что переменная определена в данной точке тогда и только тогда, когда на всех путях от начала программы до данной точки содержится определение данной переменной.

```
1    string query = "a = 1;";
2    query += cond? "b" : "c";
3    query += " = 2;";
4    query += "x = a + b + d;";
5    Eval(query);
6
```

Листинг 3: Пример программы со встроенным языком

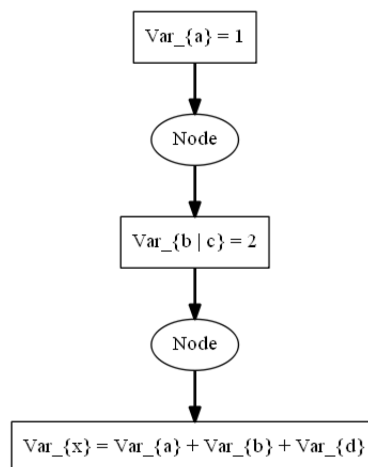


Рис. 9: Граф потока управления для листинга 3

Рассмотрим фрагмент кода, представленный на листинге 3, и соответствующий ему граф потока управления (рис. 9). Здесь и далее переменные встроенного языка будут обозначены как Var_w, где Var означает, что данный тип токена является переменной, а w означает множество возможных идентификаторов. Например, во втором блоке объявляется переменная, идентификатором которой может быть b или c.

Поищем неопределённые переменные в данном графе потока управления. В последнем блоке переменная с идентификатором a гарантированно определена, посколь-

ку происходит присваивание в эту переменную в первом блоке. Присваивания в переменную d не происходит, поэтому о ней будет просигнализировано, как об ошибке. Состояние переменной b зависит от параметра $cond$ основного языка, а не встроенного. Если $cond$ имеет значение $true$, то в строку $query$ будет добавлена часть, инициализирующая переменную b. В результате во втором блоке графа потока управления будет происходить присваивание в переменную b, что сделает её состояние "определена". В противном случае (когда параметр $cond$ имеет значение $false$) переменная b будет находиться в состоянии "не определена".

Проблема возникла из-за того, что с одной переменной было ассоциировано несколько идентификаторов, что невозможно при классическом анализе программ. Поэтому необходимо расширить алгоритм поиска неинициализированных переменных на такой класс случаев.

Отметим, что множество идентификаторов может быть аппроксимировано сверху регулярным выражением [3]. Действительно, конкатенация строк соответствует конкатенации регулярных выражений, ветвления в циклах – операции объединения, циклы или рекурсия – операции замыкания. Далее будем считать, что с каждой переменной ассоциирован конечный автомат, задающий множество идентификаторов. На листинге 4 у формируемой переменной неограниченное число имён идентификаторов. Однако все они принадлежат регулярному языку $xy*$.

```
1   string query = "x";
2   for(int i = 0; i < limit; i++)
3   {
4       query += "y";
5   }
6   query += " = 1;";
7   //other code
8
```

Листинг 4: Программа, порождающая переменную с неограниченным числом идентификаторов.

Другая особенность анализа потоков данных динамически формируемых программ состоит в том, что выражения представляются графом, а не линейным потоком. Например, в правой части присваивания могут быть ветвления и циклы. В некоторых задачах анализа потоков данных это должно находить своё отражение. Например, если рассмотреть задачу поиска констант, то переменной, чтобы считаться константой, недостаточно иметь в правой части константы и операции над ними. Необходимо ещё, чтобы в правой части присваивания отсутствовали ветвления и циклы. Однако в некоторых видах анализа нелинейное представление выражений не играет особой роли. Например, в задаче поиска хорошо определённых переменных достаточно пройтись по дугам графа, извлечь переменные и провести анализ переменной.

Ниже предлагается алгоритм, который расширяет задачу поиска хорошо определённых переменных на случай, когда одной переменной соответствует конечный автомат идентификаторов. Идея базируется на том, чтобы свести данную задачу к задаче о разрешимости булевой формулы. Ниже представлено формальное описание.

Полурешётка. В качестве полурешётки возьмём множество булевых формул с операцией пересечения ∇, определённой следующим образом. Пусть Γ_1 и Γ_2 – множества аксиом. Тогда $\Gamma_3 = \Gamma_1 \nabla \Gamma_2$ будет содержать те формулы из Γ_1, что выводимы из Γ_2, плюс те формулы из Γ_2, что выводимы из Γ_1.

$$\Gamma_3 = \Gamma_1 \nabla \Gamma_2 = \{\Phi_i \colon (\Phi_i \in \Gamma_1) \wedge (\Gamma_2 \models \Phi_i)\} \cup \{\Phi_j \colon (\Phi_j \in \Gamma_2) \wedge (\Gamma_1 \models \Phi_j)\}$$

Покажем, что это действительно полурешётка. Для этого операция ∇ должна удовлетворять всем необходимым условиям.

- Идемпотентность. $\Gamma \nabla \Gamma = \Gamma$

  Идемпотентность вытекает из того, что любая формула Φ_i из Γ выводима из Γ (поскольку Φ_i будет содержаться и в левой, и в правой части секвенции).

- Коммутативность. $\Gamma_1 \nabla \Gamma_2 = \Gamma_2 \nabla \Gamma_1$ Коммутативность верна, потому что

  $$\Phi \in \Gamma_1 \nabla \Gamma_2 \Leftrightarrow \Gamma_1 \models \Phi \wedge \Gamma_2 \models \Phi \Leftrightarrow \Gamma_2 \models \Phi \wedge \Gamma_1 \models \Phi \Leftrightarrow \Phi \in \Gamma_2 \nabla \Gamma_1$$

- Ассоциативность. $\Gamma_1 \nabla (\Gamma_2 \nabla \Gamma_3) = (\Gamma_1 \nabla \Gamma_2) \nabla \Gamma_3$ Аналогичным способом, как и коммутативность, показывается ассоциативность.

Выразим основные операции над регулярными языками в виде предикатов второго порядка. Ниже представлены предикаты для конкатенации двух символов (`Concat`), альтернативы (`Alter`) и замыкания (`Closure`). Под ϵ подразумевается пустая строка. С помощью композиции данных трёх операций выражаются все слова, принадлежащие классу регулярных выражений.

$$Concat_{ab}\,(x) \rightleftharpoons \exists\, y(x = ay \wedge y = b)$$

$$Alter_{a|b}\,(x) \rightleftharpoons x = a \vee x = b$$

$$Closure_{a*}\,(x) \rightleftharpoons x = \epsilon \vee \exists\, y\,(x = ay \wedge Closure_{a*}(y))$$

Теперь опишем, как будут выглядеть разметки $before(v)$ и $after(v)$, а также потоковая функция $f_v(x)$.

Потоковая функция. Потоковая функция $f_v(X)$ будет иметь вид $f_v(X) = X \cup D$, где D:

- пустое множество, если вершина v не является блоком присваивания;
- множество, состоящее из формулы, соответствующей конечному автомату идентификаторов левой части.

Для листинга 5 потоковая функция отработает следующим образом.

```
1   query = "a";
2   while(cond)
3   {
4       query += "b";
5       //other code
6   }
7   query += " = 1";
8   //other code
9
```

Листинг 5: Фрагмент программы, порождающей переменную с бесконечным число идентификаторов.

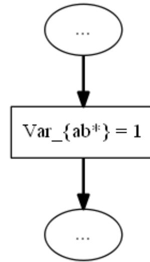


Рис. 10: Граф потока управления для листинга 5.

Поскольку в данном блоке происходит присваивание значения в переменную, то нужно добавить новую формулу в список аксиом. Гарантированно известно, что в данном блоке произошла инициализация какого-то идентификатора, но неизвестно, какого именно. Таким образом, для данного блока v потоковая функция $f_v(X)$ будет добавлять формулу (1) в список аксиом.

$$\exists\, w\ (IsWord_{ab*}(w)\ \wedge\ w\ \in\ Defined) \quad (1)$$

$$IsWord_{ab*}\ (x)\ \rightleftharpoons\ \exists\, y\ (x\ =\ ay\ \wedge\ IsWord_{b*}(y))$$

$$IsWord_{b*}(y)\ \rightleftharpoons\ y\ =\ \epsilon\ \vee\ \exists\, z\ (y\ =\ bz\ \wedge\ IsWord_{b*}(z))$$

Разметка $after$ будет иметь вид $after(v) = f_v(before(v))$. Т.е. во множество аксиом, определённых непосредственно перед вершиной v, добавляются формулы, порождённые потоковой функцией.

Разметка $before$ будет иметь вид

$$before(v) = \bigwedge_{u \in pred(v)} after(u)$$

где $\wedge$ – это решёточная операция ∇, определённая ранее.

Процесс определения состояния переменной в вершине v происходит следующим образом.

- Для переменной x составляется предикат $IsWord_{...}$, соответствующий конечному автомату, ассоциированному с переменной x.
- Составляется секвенция, приведённая ниже. Γ – множество аксиом (в точности совпадает с разметкой $before(v)$).

$$\Gamma \models \forall\, w\ (IsWord_{...}(w) \Rightarrow w \in Defined)$$

- Если секвенция выводима (т. е. на листьях дерева вывода одни аксиомы), то переменная считается хорошо определённой. В противном случае переменная не считается хорошо определённой. При этом на тех листьях дерева вывода, что не являются аксиомами, будут находиться те значения идентификаторов, которые могут быть не определены в ходе выполнения программы.

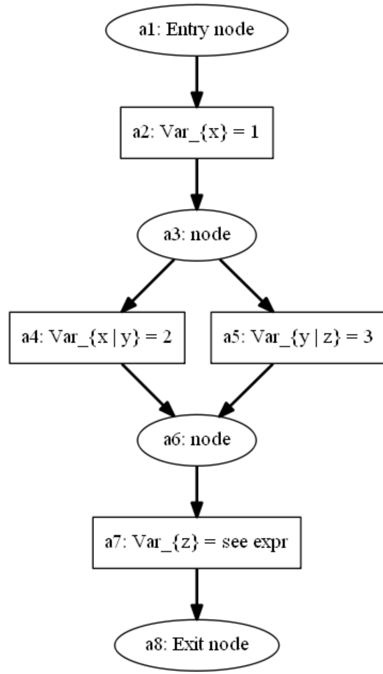
В качестве примера приведём решение задачи поиска хорошо определённых переменных для графа потока управления, изображённого на рис. 11. Для простоты дальнейших рассуждений все вершины помечены значениями от a_1 до a_8.

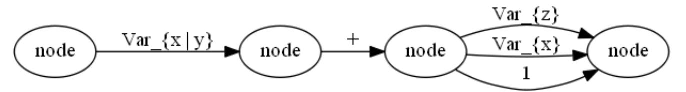
На первом этапе происходит разметка вершин данного графа. С каждой вершиной будем ассоциировать множество аксиом. Для этого нужно определить, как множество аксиом будут меняться при переходе от одной вершины к другой. Эти переходы представлены ниже.

Вершина a_1 является стартовой, до этого состоянии переменные не могут быть определены. Поэтому с a_1 всегда ассоциировано пустое множество предикатов.

Т. к. в вершине a_1 не происходит присваивания значения в переменную, то множество определённых идентификаторов перед вершиной a_2 совпадает со множеством идентификаторов, определённых до вершины a_1. Аналогичное рассуждение верно к вершинам a_4, a_5, a_7.

Рассмотрим вершину a_3. У неё один родитель: вершина a_2, в которой происходит присваивание в переменную с идентификатором x. Значит, непосредственно перед a_3





(a) Граф выражения для вершины a_7.

Рис. 11: Граф потока управления.

определены те же вершины, что и перед a_2, плюс новая переменная x. Аналогичное рассуждение верно и в отношении вершины a_8.

$$\begin{aligned} a_1 &= \emptyset \\ a_2 &= a_1 \\ a_3 &= a_2 \cup \Phi_1 \\ a_4 &= a_3 \\ a_5 &= a_3 \\ a_6 &= (a_4 \cup \Phi_2)\nabla(a_5 \cup \Phi_3) \\ a_7 &= a_6 \\ a_8 &= a_7 \cup \Phi_4 \end{aligned}$$

$$\Phi_1 \rightleftharpoons \exists\, w\, (IsWord_x(w) \wedge w \in Defined)$$

$$\Phi_2 \rightleftharpoons \exists\, w\, (IsWord_{x|y}(w) \wedge w \in Defined)$$

$$\Phi_3 \rightleftharpoons \exists\, w\, (IsWord_{y|z}(w) \wedge w \in Defined)$$

$$\Phi_4 \rightleftharpoons \exists\, w\, (IsWord_z(w) \wedge w \in Defined)$$

Рассмотрим вершину a_6. У неё два родителя: a_4 и a_5. Перед a_6 определены те идентификаторы, которые содержатся и непосредственно после вершины a_4, и непосредственно после вершины a_5. Для этого необходимо выполнить пересечение ассоциированных с этими вершинами предикатов. Этот процесс необходимо осуществить в

соответствии с решёточной операцией $\wedge$, т. е. с ∇, которая была определена ранее.

После того, как переходы составлены, начинается итеративный поиск неподвижной точки. Таблица по итерациям представлена ниже.

№	0	1	2	3	4	5	6
a_1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
a_2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
a_3	$\emptyset$	$\{\Phi_1\}$	$\{\Phi_1\}$	$\{\Phi_1\}$	$\{\Phi_1\}$	$\{\Phi_1\}$	$\{\Phi_1\}$
a_4	$\emptyset$	$\emptyset$	$\{\Phi_1\}$	$\{\Phi_1\}$	$\{\Phi_1\}$	$\{\Phi_1\}$	$\{\Phi_1\}$
a_5	$\emptyset$	$\emptyset$	$\{\Phi_1\}$	$\{\Phi_1\}$	$\{\Phi_1\}$	$\{\Phi_1\}$	$\{\Phi_1\}$
a_6	$\emptyset$	$\emptyset$	$\emptyset$	$\{\Phi_1,\ \Phi_2\}$	$\{\Phi_1,\ \Phi_2\}$	$\{\Phi_1,\ \Phi_2\}$	$\{\Phi_1,\ \Phi_2\}$
a_7	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{\Phi_1,\ \Phi_2\}$	$\{\Phi_1,\ \Phi_2\}$	$\{\Phi_1,\ \Phi_2\}$
a_8	$\emptyset$	$\{\Phi_4\}$	$\{\Phi_4\}$	$\{\Phi_4\}$	$\{\Phi_4\}$	$\{\Phi_1,\ \Phi_2,\ \Phi_4\}$	$\{\Phi_1,\ \Phi_2,\ \Phi_4\}$

В начале работы алгоритма с каждой вершиной ассоциировано пустое множество аксиом. Далее множества меняются в соответствии с определённым выше списком переходов (тавтологии при этом игнорируются). Например, переход $a_4 = a_3$ означает, что значение a_4 на i-ой итерации будет совпадать со значением a_3 на $i-1$-ой итерации. Алгоритм заканчивает свою работу на 6-ой итерации, поскольку никаких изменений по сравнению с предыдущей итерацией не произошло. Таким образом, разметка, полученная на момент остановки алгоритма, является неподвижной точкой.

Теперь посмотрим, как с помощью полученной разметки определять состояние переменных. Рассмотрим правую часть присваивания в вершине a_7 (рис. 11,a). Она представлена графом. Ищем в графе дуги, помеченные переменными. В ней есть две переменные: с одной ассоциирован конечный автомат, принимающий слова $x|y$, со второй — автомат, принимающий слово z, с третьей – автомат, принимающий слово x. Для них будут составлены предикаты, указанные ниже. В левой части секвенции фигурирует множество формул, состоящее из Φ_1, Φ_2, поскольку именно этим множеством помечена вершина a_7 в неподвижной точке.

$$\Phi_1,\ \Phi_2\ \models\ \forall\, w\ (IsWord_{x|y}(w)\ \Rightarrow\ w\ \in\ Defined)$$

$$\Phi_1,\ \Phi_2\ \models\ \forall\, w\ (IsWord_{z}(w)\ \Rightarrow\ w\ \in\ Defined)$$

$$\Phi_1,\ \Phi_2\ \models\ \forall\, w\ (IsWord_{x}(w)\ \Rightarrow\ w\ \in\ Defined)$$

Заметим, что только третья формула является выводимой. В первом случае идентификатор y не принадлежит множеству определённых идентификаторов, а во втором случае – идентификатор z. Поэтому переменные $Var_{x|y}$ и Var_z могут оказаться неинициализированными в ходе выполнения динамически формируемой программы.

5. Реализация

5.1. Архитектура

Предложенные алгоритмы были реализованы как часть проекта YaccConstructor; языком разработки являлся F# [12]. Работа велась в рамках модуля, отвечающего за семантику (**Semantic processing** на рис. 12). Созданные в ходе данной работы модули выделены цветом.

Компонент **Common** содержит в себе вспомогательную информацию, необходимую для построения графа потока управления, а также работы с ним. Примером такой информации может быть соответствие между нетерминалами и конструкциями языка (т.е. что поддереву с корнем, помеченным нетерминалом `assignment`, нужно сопоставлять блок графа потока управления, соответствующий присваиванию).

Компонент **Control Flow Graph Builder** отвечает за построение графа потока управления. В своей работе он пользуется модулем **Common.AST**, который содержит в себе лес разбора **SPPF**, а также компонентом **Common**. Построение графа потока управления происходит согласно алгоритму, описанному в разделе 3.

Компонент **Control Flow Graph** содержит в себе граф потока управления и функции для работы с ним. В частности, есть реализация поиска неопределённых переменных.

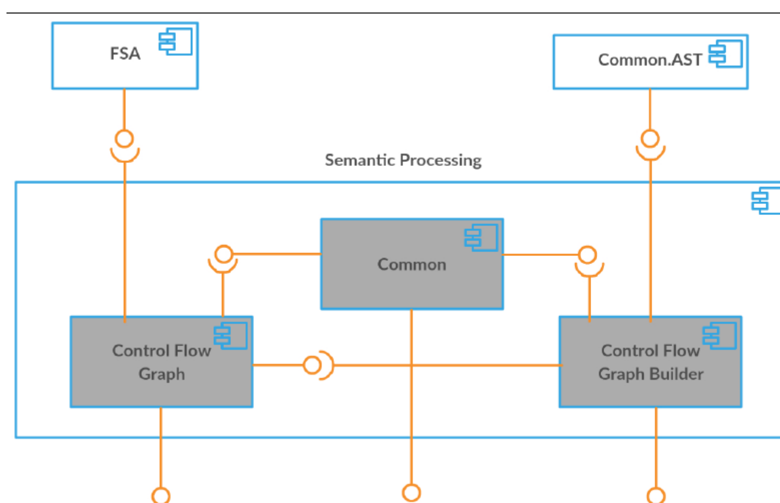


Рис. 12: Архитектура модуля **Semantic Processing**.

Также данная функциональность встроена в плагин для ReSharper для поддержки встроенных языков.

На рис. 13 переменная встроенного языка `varY` подчёркнута на 22-й строке, поскольку она может оказаться неопределённой. Это может случиться, если параметр `cond` принимает значение $false$.

```
17  public static void Execute(bool cond)
18  {
19      string query = "varX = 1;";
20      if (cond)
21          query += "varY = 2;";
22      query += "varZ = varX + varY;";
23      Program.ExtEval(query);
24  }
```

Рис. 13: Пример работы плагина.

5.2. Детали реализации поиска хорошо определённых переменных

В реализации поиск хорошо определённых переменных происходит в два этапа.

На первом этапе происходит разметка вершин графа. В реализации вершины графа размечаются двумя конечными автоматами. Первый автомат (`Surely`) содержит в себе гарантированно определённые идентификаторы, а второй (`Maybe`) – те идентификаторы, которые встречались в левых частях присваиваний и "определённость" которых зависит от потока исполнения основной программы. Т. е. если с инициализируемой переменной ассоциирован только один идентификатор, то он попадёт в автомат `Surely`, в противном случае – в автомат `Maybe`.

Операции разметок тогда будут выглядеть следующим образом. Разметка $before(v)$ является пересечением разметок $after(u)$, где вершина `u` – родитель вершины `v`. Пересечение происходит поэлементно, т. е. результирующий автомат `Surely` является пересечением автоматов `Surely` родительских вершин, а результирующий автомат `Maybe` – пересечением автоматов `Maybe`.

Пусть $q = (S, M)$, где S – автомат, соответствующий гарантированно определённым идентификаторам, а M – автомат, соответствующий Maybe идентификаторам. Тогда потоковая функция $f_v(q)$ возвращает следующие значения:

- разметку $(S,\ M)$, если вершина `v` не является блоком присваивания;
- разметку $(S \cup x, M \setminus x)$, если совершается присваивание в переменную, с которой ассоциирован один идентификатор x;
- разметку $(S, M \cup X \setminus S)$, если совершается присваивание в переменную, с которой ассоциировано множество идентификаторов $X(|X| > 1)$.

Разметка $after(v)$ имеет вид $after(v) = f_v(before(v))$.

Вместо использования подхода, связанного с поиском неподвижной точки на полурешётках, используется алгоритм с рабочим списком [1]. Его идея в том, чтобы

производить итеративное пересчёт разметок, используя рабочий список вершин. Он содержит те вершины, для предшественников которых значение разметок было изменено на предыдущем шаге. Опустошение списка будет свидетельствовать о том, что достигнута неподвижная точка, что приведёт к остановке алгоритма.

На втором этапе с помощью полученных разметок происходит анализ состояния переменных. Для каждой переменной x каждой вершины `v` графа происходит проверка: является ли данная переменная хорошо определённой. Для этого нужно сделать следующие шаги.

1. Пусть FSA_{ids} – автомат, ассоциированный с переменной x, а $FSA_{surely_defined}$ – автомат, содержащий в себе множество гарантированно определённых идентификаторов, $FSA_{maybe_defined}$ – автомат, содержащий в себе множество идентификаторов, принадлежащих `Maybe`.

2. "Вычесть" из автомата FSA_{ids} автомат $FSA_{surely_defined}$ (под вычитанием понимается пересечение автомата FSA_{ids} с дополнением автомата $FSA_{surely_defined}$). Пусть результатом этой операции будет автомат FSA_{res}.

3. Если автомат FSA_{res} является пустым (т.е. в нём не существует пути из начального состояния в конечное), то переменная x является хорошо определённой.

4. В противном случае переменная x не является хорошо определённой и может оказаться неинициализированной перед первым использованием. Отметим, что если верно $FSA_{res} \subset FSA_{maybe_defined}$, то инициализированность переменной x зависит только от того, в какой именно идентификатор происходило присваивание значения.

Заключение

В ходе данной работы получены следующие результаты.

- Разработан алгоритм построения графа потока управления для динамически формируемого кода, который принимает на вход лес разбора
- Решена задача поиска хорошо определённых переменных при анализе встроенных языков.
- Выполнена реализация получившихся алгоритмов в рамках исследовательского проекта YaccConstructor.
- Результаты работы вошли в статью “On Development of Static Analysis Tools for String-Embedded Languages” (CEE-SECR’15 Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia).

Проект можно найти на сайте `https://github.com/YaccConstructor/YaccConstructor`, автор принимал участие под учётной записью IvanovAndrew.

В дальнейшем планируется исследовать другие задачи анализа потоков данных: задача поиска достижимые определения (use-def chains), живых переменных (def-use chains). Решение первой задачи может быть использовано при реализации такой функции IDE, как “перейти к определению” (go to definition), в то время как решение второй может быть полезно при реализации функции “перейти к использованию” (find usages).

Список литературы

[1] Aho A. V., Sethi R., Ullman J. D. Compilers: Principles, Techniques, and Tools. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] Annamaa A., Breslav A., Vene V. Using Abstract Lexical Analysis and Parsing to Detect Errors in String-Embedded DSL Statements // Proceedings of the 22nd Nordic Workshop on Programming Theory. — 2010. — P. 20–22.

[3] Automata-based Symbolic String Analysis for Vulnerability Detection / F. Yu, M. Alkhalaf, T. Bultan, O. H. Ibarra // Form. Methods Syst. Des. — 2014. — Vol. 44, no. 1. — P. 44–70.

[4] Christensen A. S., Møller A., Schwartzbach M. I. Precise Analysis of String Expressions // Proc. 10th International Static Analysis Symposium (SAS). — Vol. 2694 of LNCS. — Springer-Verlag, 2003. — P. 1–18.

[5] Constantini J., Ferrara P., Cortesi A. Static Analysis of String Values // Formal Methods and Software Engineering. 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings. — Springer Berlin Heidelberg, 2011. — P. 505–521.

[6] An Interactive Tool for Analyzing Embedded SQL Queries / A. Annamaa, A. Breslav, J. Kabanov, V. Vene // Programming Languages and Systems. — 2010. — P. 131–138.

[7] Kim H., Doh K.-G., Schmidt D. Static validation of dynamically generated HTML documents based on abstract parsing and semantic processing // Static Analysis. 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. — Springer Berlin Heidelberg, 2013. — P. 194–214.

[8] Minamide Y. Static Approximation of Dynamically Generated Web Pages // Proceedings of the 14th International Conference on World Wide Web. — WWW '05. — New York, NY, USA : ACM, 2005. — P. 432–441.

[9] Nguyen H. V., Kästner C., Nguyen T. N. Varis: IDE Support for Embedded Client Code in PHP Web Applications // Proceedings of the 37th International Conference on Software Engineering (ICSE). — Vol. 2. — 2015. — P. 693–696.

[10] Rekers J. G. Parser Generation for Interactive Environments : Ph. D. thesis / J. G. Rekers ; Universiteit van Amsterdam. — 1992.

[11] String-embedded Language Support in Integrated Development Environment / S. Grigorev, E. Verbitskaia, A. Ivanov et al. // Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia. — CEE-SECR '14. — New York, NY, USA : ACM, 2014. — P. 21:1–21:11.

[12] Syme D., Granicz A., Cisternino A. Expert F# (Expert's Voice in .Net).

[13] Verbitskaia E., Grigorev S., Avdyukhin D. Relaxed Parsing of Regular Approximations of String-Embedded Languages // Preliminary Proceedings of the PSI 2015: 10th International Andrei Ershov Memorial Conference. — PSI'15. — 2015. — P. 1–12.

[14] Григорьев С.В. Синтаксический анализ динамически формируемых программ : Дисс... кандидата наук / С.В. Григорьев ; Санкт-Петербургский государственный университет. — 2015.

[15] Инструментальная поддержка встроенных языков в интегрированных средах разработки / С. В. Григорьев, Е. А. Вербицкая, М.И. Полубелова и др. // Моделирование и анализ информационных систем. — 2014. — Т. 21, № 6. — С. 131–143. — URL: http://mais-journal.ru/jour/article/view/77.

[16] Кириленко Я.А, Григорьев С.В., Авдюхин Д.А. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем // Научно-технические ведомости Санкт-Петербургского государственного политехнического университета. Информатика. Телекоммуникации. Управление. — 2013. — Т. 174, № 3. — С. 94–98. — URL: http://ntv.spbstu.ru/telecom/article/T3.174.2013_11/.

[17] Полубелова М.И. Лексический анализ динамически формируемых строковых выражений // Бакалаврская работа кафедры системного программирования СПбГУ. — 2015. — URL: http://se.math.spbu.ru/SE/diploma/2015/bmo/444-Polubelova-report.pdf.

[18] Хабибуллин М.Р. Построение регулярной аппроксимации встроенных языков. — 2015. — Магистерская диссертация.