Принципы статического анализа исходного кода

Ахундов Алексей Назимович, БПИ205

Декабрь, 2022

Содержание

Метрики ПО

Цикломатическая сложность

Рассматривается граф потока управления программы (Control-flow graph):

- узлы - неделимые группы команд программы (базовые блоки) без передачи управления от или к (кроме первой инструкции)
- ориентированные ребра - зависимость: если вторая группа может быть выполнена непосредственно после группы команд первого узла, они соединяются

Цикломатическая сложность - количество линейно незавимсимых маршрутов через этот граф:

$$M = E - N + 2 \times P$$

$$\begin{aligned} M &= \text{цикломатическая сложность} \\ E &= \text{количество ребер в графе} \\ N &= \text{количество вершин в графе} \\ P &= \text{количество компонент связности} \end{aligned}$$

Связанность

- Cohesion - мера степени функциональной связанности компонент одного модуля (как хорошо связанны отдельные части модуля для решения одной задачи), чем больше - тем лучше
  Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.
- Coupling - мера степени зависимости между модулями, чем меньше - тем лучше
  Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

Лексический анализ

Пример грамматики для AntLR, регулярные выражения для термов и последовательностей:

```
grammar While;

program : seqStatement;

seqStatement: statement (';' statement)* ;

statement: ID ':=' expression                                  # attrib
         | 'skip'                                              # skip
         | 'if' bool 'then' statement 'else' statement         # if
         | 'while' bool 'do' statement                         # while
         | 'print' Text                                        # print
         | 'print' expression                                  # write
         | '{' seqStatement '}'                                # block
         ;

expression: INT                                                # int
          | 'read'                                             # read
          | ID                                                 # id
          | expression '*' expression                          # binOp
          | expression ('+'|'-') expression                    # binOp
          | '(' expression ')'                                 # expParen
          ;

bool: ('true'|'false')                                         # boolean
    | expression '=' expression                                # relOp
    | expression '<=' expression                               # relOp
    | 'not' bool                                               # not
    | bool 'and' bool                                          # and
    | '(' bool ')'                                             # boolParen
    ;

INT: ('0'..'9')+ ;
ID: ('a'..'z')+;
Text: '"' .*? '"';
Space: [ \t\n\r] -> skip;
```

Статический анализ

Пример грамматики

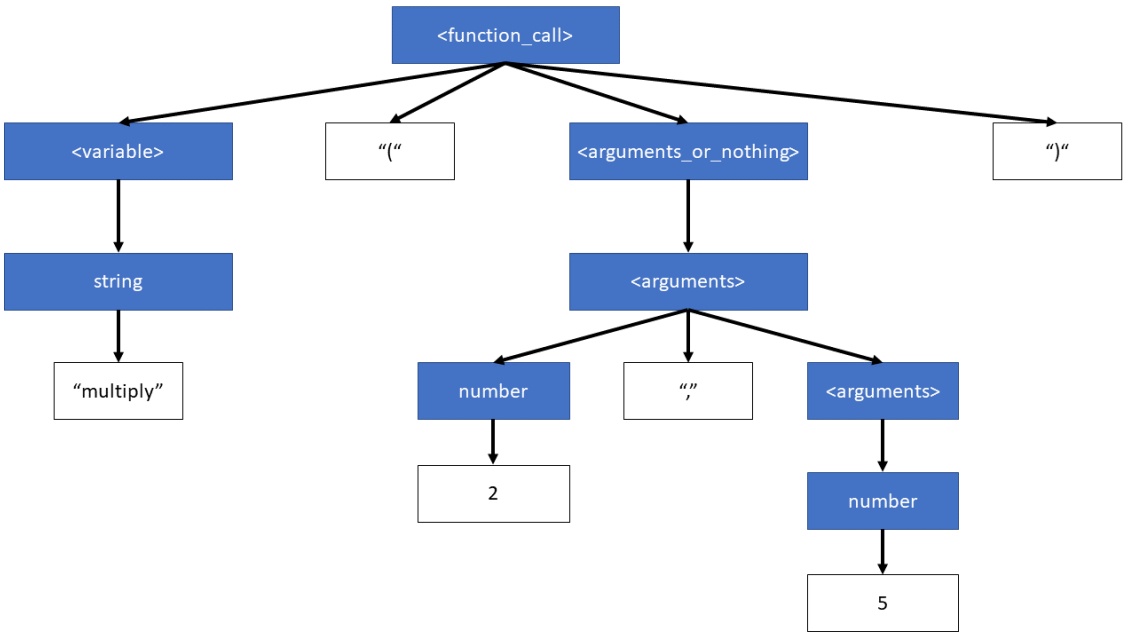
```
<function_call> ::=
  <variable> '(' <arguments_or_nothing> ')'

<arguments_or_nothing> ::= <arguments> |

<arguments> ::=
   number ',' <arguments>
   | number

<variable> ::= string
```

Рассматриваем строку этого языка *multiply(2,5)*

Дерево разбора

Дерево разбора - результат грамматического анализа - корневое дерево синтаксиса со всеми специальными символами: скобки, запятые, точки и прочее

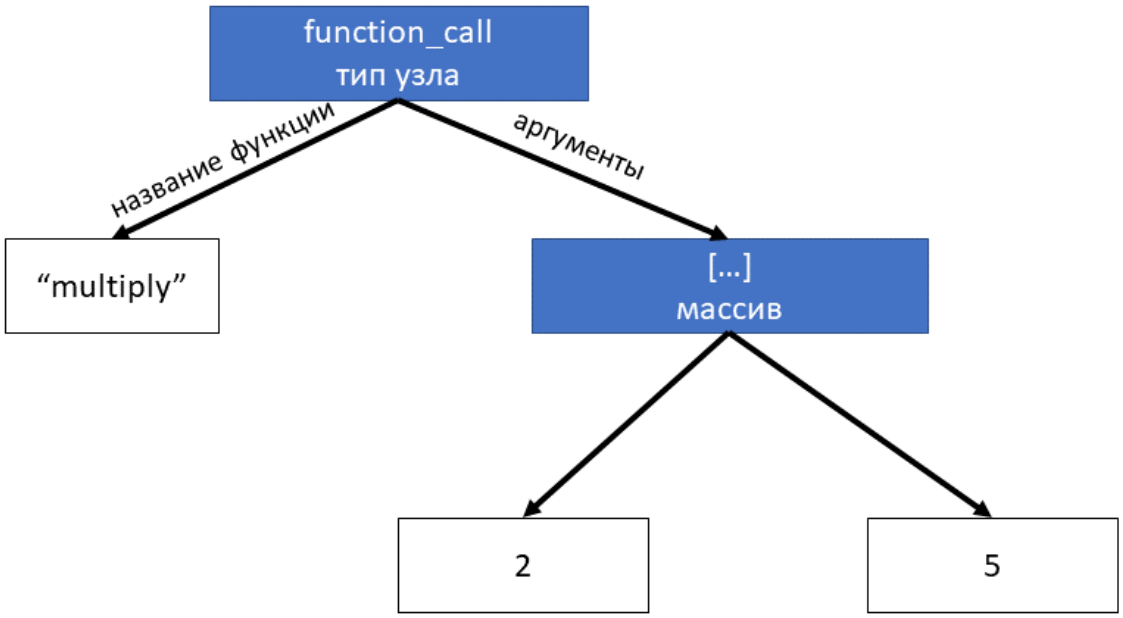
Рис. 1: Дерево разбора для строки *multiply(2, 5)*



Абстрактное синтаксическое дерево

Абстрактное синтаксическое дерево - конечное корневое ориентированное дерево, вершины - синтаксические единицы (операторы, переменные, литералы и прочее), ребра - операнды (переменные и константы). Дерево очищено от ребер для тех синтаксических правил, которые не влияют на семантику

Рис. 2: AST для строки *multiply(2, 5)*



Преобразование дерева разбора в АСТ

Для преобразования parse tree в некий заранее спроектированный вариант AST требуется сделать ряд типовых операций:

- Переименовать некоторые узлы;
- Удалить «служебные» узлы;
- Свернуть порождённое рекурсией поддерево <arguments> в линейный список/массив.

Рассмотрим, для примера, следующую грамматику (упрощённый оператор присвоения):

```
1 <assignment> ::= letter "=" number
```

Где letter — буква латинского алфавита, number — число.
Данная грамматика будет соответствовать строкам типа x=5, y=21 и т. д.

Предположим, что в нашем варианте AST мы для каждого присвоения хотим сохранять массив вида [”x 5], [”y 21] и т. д. (В данном случае мы, говоря AST, имеем в виду в первую очередь внутреннюю структуру данных языка разработки, а не её графическое изображение.) Тогда мы можем добавить код (в фигурных скобках) к данному правилу в следующей манере:

```
<assigment> ::= letter "=" number ; { [$1, $3] }
```

В системах автоматической генерации парсеров подобный код даёт следующее указанию парсеру: «когда обработаешь данное правило вывода (построишь соответствующее ему parse tree), возьми первый символ (letter) и третий (number) и выполни с ними указанный код (объедини в массив из двух элементов) — это и будет AST».

Внутреннее представление

Абстрактное синтаксическое дерево может являться промежуточным представлением между деревом разбора и внутренним представлением.

Таблица символов

Структура данных для поддержания информации о символах в программе: типы данных, скоуп доступа (где они находятся в терминах синтаксиса), имена, переменные, имена функций, классы, объекты и прочее.

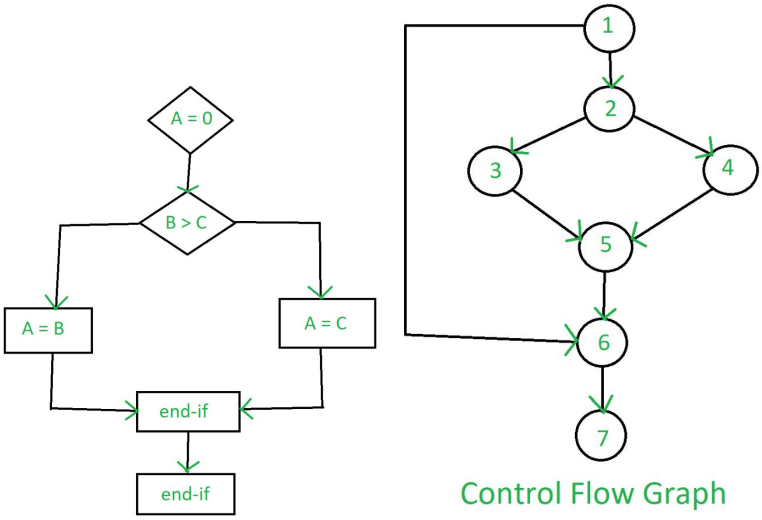
- Лексический анализ - наполнение таблицы изначальными конструкциями: токены
- Синтаксический анализ - наполнение типом, скоупом, строкой кода использования
- Семантический анализ - использование таблицы для проверки типов, конверсий
- Генерация промежуточного кода - использование для рассчета размера рантайм памяти, временная информация
- Оптимизации - машинные оптимизации
- Генерация конечного кода - присвоение и использование адресов символов

Граф потока управления

Вершина - базовый блок, ребра - передача управления + блок входа и блок выхода. Пример:

```
1 if  A = 10 then
2   if B > C
3      A = B
4   else A = C
5   endif
6   endif
7 print A, B, C
```

Рис. 3: FLOWCHART и CFG для кода выше



Доминатор (Dominator)

Блок M называется доминирующим над блоком N, если любой путь от входного блока к блоку N проходит через блок M. Входной блок доминирует над всеми остальными блоками графа.

Непосредственный доминатор (Immediate dominator)

Блок M называется непосредственно доминирующим блок N, если блок M доминирует блок N, и не существует иного промежуточного блока P, который бы доминировался блоком M и доминировал над блоком N. Другими словами, M — последний доминатор в любых путях от входного блока к блоку N. У каждого блока кроме входного есть единственный непосредственный доминатор.

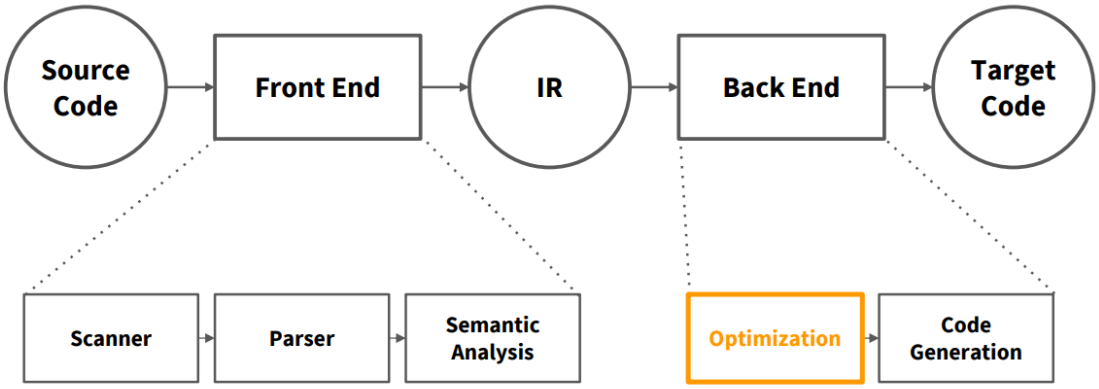
Дерево доминаторов (Dominator Tree)

Вспомогательная структура данных типа дерево, содержащая информацию об отношениях доминирования. Ветка от блока M к блоку N создаётся тогда и только тогда, когда блок M является непосредственным доминатором N. Структура данных является деревом, поскольку любой блок имеет уникального непосредственного доминатора. Корнем дерева является входной узел. Для построения используется эффективный алгоритм Lengauer-Tarjan's.

Анализ абстрактных синтаксических деревьев

При проходе по ACT имеется возможность просмотреть наличие импорта (или в целом) определенного типа при определении его для переменной

Рис. 4: Pipeline трансляции



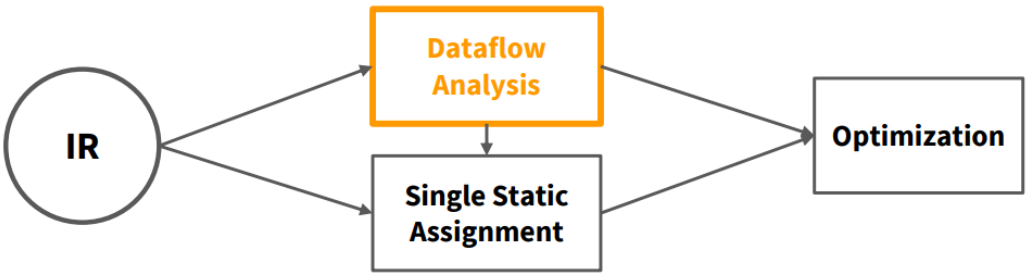
Анализ потока выполнения

???

Анализ потока данных

Анализ потока данных - анализ информации о объявлении и использовании информации в программе. Поток данных программы получается из потока управления программы - те же вершины, но ребра ставятся по принципу записи-чтения (в таком порядке) по каждой переменной:

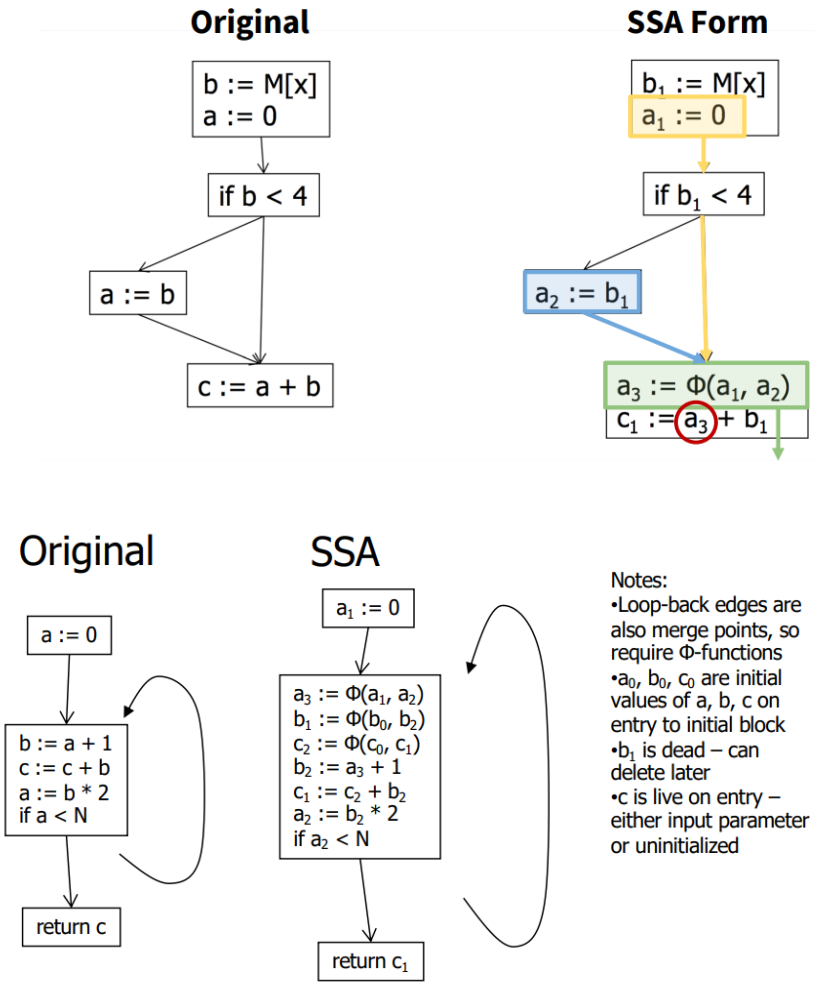
Рис. 5: IR->Optimization



SSA

Static single-assignmnent form. Промежуточное представление, используемое компиляторами, в котором каждой переменной значение присваисвается лишь единожды. Переменные разбиваются на версии (суффексы) так, что каждое присвание - уникальной версии переменной. При неоднозначности (между вариантами - при if-е, например), используем Ф функцию:

Рис. 6: SSA



UW CSE 401/M501 Spring 2021

Data Flow Graph

```
1                              //          a     b     r
2  int max(int a, int b) {     // 1        W     W
3    if (a > b)                // 2        R     R
4      r = a;                  // 3        R           W
5    else
6      r = b;                  // 4              R     W
7    return r;                 // 5                    R
```

Находим крайнюю точку чтения переменной, затем поднимаемся вверх поиском в ширину и расставляем ребра

```
1                              //              data edges
2                              //          a         b         r
3  int max(int a, int b) {     // 1        1->2      1->2
4                              //          1->3      1->4
5    if (a > b)                // 2
6      r = a;                  // 3                              3->5
7    else
8      r = b;                  // 4                              4->5
9    return r;                 // 5
```

Пример с циклами

```
1                                 //          a     b
2  unsigned gcd(unsigned a,       // 1        W     W
3               unsigned b) {
4
5    while (a != b)               // 2        R     R
6      if (b > a)                 // 3        R     R
7        b = b - a;               // 4        R     W,R
8      else
9        a = a - b;               // 5        W,R   R
10
11   return a;                    // 6        R
12 }
```

```
1                                 //          data edges
2                                 //          a      b
3  unsigned gcd(unsigned a,       // 1        1->2   1->2
4               unsigned b) {                 1->3   1->3
5                                             1->4   1->4
6                                             1->5   1->5
7                                             1->6   1->6
8
9    while (a != b)               // 2
10     if (b > a)                 // 3
11       b = b - a;               // 4                4->4
12                                //                   4->3
13                                //                   4->2
14                                //                   4->5
15     else
16       a = a - b;               // 5        5->5
17                                //          5->6
18                                //          5->2
19                                //          5->3
20                                //          5->4
21
22   return a;                    // 6
23 }
```

Другие примеры

```
1                                  //        CFG          a        DFG(a)
2 for (a=0;                        // 1      1->2         W
3           a<3;                   // 2      2->4         R        1->2, 3->2
4                                             2->exit
5                 a++) {           // 3      3->2         W,R      1->3, 3->3
6   b = b + a;                     // 4      4->3         R        1->4, 3->4
7 }
```

```
1                                  //        CFG          a        DFG(a)
2 switch(a) {                      // 1      1->2         R        initial->1
3                                             1->3
4                                             1->4
5   case '1': b = a;               // 2      2->exit      R        initial->2
6             break;
7   case '2': c = a;               // 3      3->exit      R        init->3
8             break;
9   default : b = 0;               // 4      4->exit
10 }
```

```
1                                        //      CFG          a        DFG(a)
2  a = 1;                                // 1    1->2         W
3  do {
4     a = a - 1;                         // 2    2->3         R,W      1->2, 2->2
5  } while (a != 0);                     // 3    3->2         R        2->3
6                                        //      3->exit
```

Межпроцедурный анализ

???

Символическое исполнение и SMT решатели

Символическое исполнение

Аналогично символьному вычислению мат. выражений (решению мат. задач) - см. sympy.
Средство анализа программы для определения того, какие входные данные вызывают выполнение каждой части программы. Интерпретатор следует программе, принимая символические значения для входных данных, а не получая фактические входные данные, как при нормальном выполнении программы. Таким образом, он приходит к выражениям в терминах этих символов для выражений и переменных в программе и к ограничениям в терминах этих символов для возможных результатов каждой условной ветви.

SMT

SMT - проблема определения выполнения определенного предиката. SMT - формула логики первого порядка для некоторого языка: $3x + 2y - z \geq 4$, $f(f(u, v), v) = f(u, v)$ (f - не задана). SMT решатель пытается проверить, выполняется ли SMT. Используется, например, для генерации тестовых данных

Satisfiability Modulo theories (SMT) is an area of automated deduction that studies methods for checking the satisfiability of first-order formulas with respect to some logical theory T of interest. It differs from general automated deduction in that the background theory T need not be finitely or even first-order axiomatizable, and specialized inference methods are used for each theory. By being theory-specific and restricting their language to certain classes of formulas (such as, typically but not exclusively, quantifier-free formulas), these specialized methods can be implemented in solvers that are more efficient in practice than general-purpose theorem provers.

Символические решатели

SMT-LIB

SMT-LIB (Satisfiability Modulo Theories LIBrary) is a interface language intended for use by programs designed to solve SMT (Satisfiability Modulo Theories)
SMT-LIB (Satisfiability Modulo Theories Library) is a library of benchmark formulas for testing satisfiability modulo theories (SMT) solvers. In simple terms, it is a collection of formulas written in a standardized format that can be used to evaluate the performance of SMT solvers. These formulas are designed to cover a range of different logical theories and to test the solvers' ability to reason about them. SMT-LIB is used as a standard benchmark for comparing the performance of different SMT solvers and for evaluating their capabilities.

```
1 ; Basic Boolean example
2 (set-option :print-success false)
3 (set-logic QF_UF)
4 (declare-const p Bool)
5 (assert (and p (not p)))
6 (check-sat) ; returns 'unsat'
7 (exit)
```

```
; Getting values or models
(set-option :print-success false)
(set-option :produce-models true)
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(assert (= (+ x (* 2 y)) 20))
(assert (= (- x y) 2))
(check-sat)
; sat
(get-value (x y))
; ((x 8) (y 6))
(get-model)
; ((define-fun x () Int 8)
;  (define-fun y () Int 6)
; )
(exit)
```

Z3

Z3 is an SMT (Satisfiability Modulo Theories) solver developed by Microsoft Research. It is a high-performance tool for automatically proving the satisfiability (SAT) or unsatisfiability (UNSAT) of logical formulas, or generating satisfying assignments for SAT formulas. Z3 can handle formulas from a variety of logical theories, including those with linear real arithmetic, non-linear real arithmetic, bit-vectors, and arrays. It is used in a variety of applications, including software verification, hardware verification, and automated theorem proving. Z3 has a number of advanced features, such as support for quantifiers, interpolation, and incremental solving, which make it a powerful tool for solving complex logical problems.

Дедуктивная верификация

Дедуктивная верификация - это метод проверки корректности системы, путем доказательства того, что она соответствует спецификации. Решение SMT-задачи - это процесс определения удовлетворимости заданной логической формулы. SMT-решатель - это инструмент, который может автоматически решать SMT-задачи. На практике решение SMT-задачи может использоваться как шаг в процессе дедуктивной верификации, но это не единственный шаг.

JML

Язык формальной спецификации для задание спецификации под поведение программ на Java. Используется в различных автоматизированных тулах, которые проверяют соответствие программы заданной спецификации или генерируют тестовые данные:

```
public class Counter {
  /*@ public invariant count >= 0; */
  private int count;

  /*@ ensures \result == count; */
  public int getCount() {
    return count;
  }

  /*@ ensures count == \old(count) + 1; */
  public void increment() {
    count++;
  }

  /*@ ensures count == \old(count) - 1;
    @ signals (Exception e) count == \old(count); */
  public void decrement() throws Exception {
    if (count > 0) {
      count--;
    } else {
      throw new Exception("Cannot decrement counter below 0");
    }
  }
}
```

@public invariant задает спецификацию на весь класс (для всех объектов обязано выполняться), @ensures задает пост-условия для методов

Why3

Платформа для дедуктивныой верификации программ. Для этого используется язык WhyML и внешние пруверы: как автоматизированные, так и интерактивные. Есть целочисленные, действиетльные арифтитические теории, булевы операции, сеты и мапы, массивы, очереди, хеш таблицы и пр.
Теормему можно описать следующим образом:

```
theory HelloProof
  goal G1: true
  goal G2: (true -> false) /\ (true \/ false)
  use int.Int
  goal G3: forall x:int. x * x >= 0
end
```

Доказать: why3 prove −P Alt−Ergo hello_proof.why −T HelloProof −G G2 −G G3

Coq

Такая же штука как Why3 только древнее и со своим языком (тут нужно самим описывать доказательства):

```
From Coq Require Import List.
Import ListNotations.
Lemma rev_snoc_cons A :
  forall (x : A) (l : list A), rev (l ++ [x]) = x :: rev l.
Proof.
  induction l.
  - reflexivity.
  - simpl. rewrite IHl. simpl. reflexivity.
Qed.
From Coq.Program Require Import Basics.
From Coq Require Import FunctionalExtensionality.
Open Scope program_scope.
Theorem rev_invol A : rev (A:=A)  rev (A:=A) = id.
Proof.
  apply functional_extensionality.
  intro x.
  unfold compose, id. rewrite rev_rev.
  reflexivity.
Qed.
```

Анализ на основе майнинга данных. Автоматическое исправление ошибок

???

Анализ Java байткода

Байткод

Рассмотрим следующую программу на Java

```
1  import java.util.Scanner;
2
3  public class ControlFlowGraph {
4    public static void main(String[] args) {
5      Scanner scanner = new Scanner(System.in);
6      int input = scanner.nextInt();
7
8      // Branch based on input value
9      if (input > 0) {
10       System.out.println("Input is positive");
11     } else {
12       System.out.println("Input is non-positive");
13     }
14
15     // Loop based on input value
16     for (int i = 0; i < input; i++) {
17       System.out.println("Iteration: " + i);
18     }
19
20     scanner.close();
21   }
22 }
```

Ее байткод:

```
1  public class ControlFlowGraph {
2    public ControlFlowGraph();
3      Code:
4         0: aload_0
5         1: invokespecial #1    // Method java/lang/Object."<init>":()V
6         4: return
7
8    public static void main(java.lang.String[]);
9      Code:
10        0: new           #7    // class java/util/Scanner
11        3: dup
12        4: getstatic     #9    // Field java/lang/System.in:Ljava/io/InputStrear
13        7: invokespecial #15   // Method java/util/Scanner."<init>":(Ljava/io/I
14       10: astore_1
15       11: aload_1
16       12: invokevirtual #18   // Method java/util/Scanner.nextInt:()I
17       15: istore_2
18       16: iload_2
```

```
19        17: ifle          31
20        20: getstatic     #22   // Field java/lang/System.out:Ljava/io/PrintStrea
21        23: ldc           #26   // String Input is positive
22        25: invokevirtual #28   // Method java/io/PrintStream.println:(Ljava/lan
23        28: goto          39
24
25        39: iconst_0
26        40: istore_3
27        41: iload_3
28        42: iload_2
29        43: if_icmpge     64
30        46: getstatic     #22   // Field java/lang/System.out:Ljava/io/PrintStrea
31        49: iload_3
32        50: invokedynamic #36,  0   // InvokeDynamic #0:makeConcatWithConstants:
33        55: invokevirtual #28       // Method java/io/PrintStream.println:(Ljava
  58: iinc          3, 1
34        61: goto          41
35        64: aload_1
36        65: invokevirtual #40       // Method java/util/Scanner.close:()V
37        68: return
38 }
```

Анализ Java байткода при помощи ObjectWeb ASM

Пример кода с использованием этой библиотеки для нахождения количества вызов методов:

```
import org.ow2.asm.ClassReader;
import org.ow2.asm.ClassVisitor;
import org.ow2.asm.MethodVisitor;
import org.ow2.asm.Opcodes;

public class MethodInvocationCounter extends ClassVisitor {
  private int methodInvocationCount = 0;

  public MethodInvocationCounter() {
    super(Opcodes.ASM9);
  }

  @Override
  public MethodVisitor visitMethod(
      int access,
      String name,
      String descriptor,
      String signature,
      String[] exceptions) {
    return new MethodVisitor(Opcodes.ASM9) {
      @Override
      public void visitMethodInsn(int opcode,
      String owner,
      String name,
      String descriptor,
      boolean isInterface) {
        methodInvocationCount++;
      }
    };
  }

  public int getMethodInvocationCount() {
    return methodInvocationCount;
  }
}

// usage
byte[] bytecode = ...; // obtain bytecode of class to analyze
ClassReader classReader = new ClassReader(bytecode);
MethodInvocationCounter counter = new MethodInvocationCounter();
classReader.accept(counter, 0);
int methodInvocationCount = counter.getMethodInvocationCount();
```

При этом сам байткод можно получить несколькими способами:

- ClassLoader для какого-то из наших классов:

```
byte[] bytecode =
    ((InstrumentationClassLoader) MyClass.class.getClassLoader())
        .getClassBytes(MyClass.class.getName());
```

- Из .class файла:

```
Path classFile = Paths.get("path/to/MyClass.class");
byte[] bytecode = Files.readAllBytes(classFile);
```

- Из .jar файла с указанием пути к классу:

```
Path jarFile = Paths.get("path/to/my.jar");
JarFile jar = new JarFile(jarFile.toFile());
JarEntry entry = jar.getJarEntry("path/to/MyClass.class");
InputStream input = jar.getInputStream(entry);
byte[] bytecode = input.readAllBytes();
```

Можно также просмотреть информацию о полях и методах, сколько раз переменная была прочтена или записана:

```
import org.ow2.asm.MethodVisitor;
import org.ow2.asm.Opcodes;

public class VariableCounter extends MethodVisitor {
  private final int variableIndex;
  private int readCount = 0;
  private int writeCount = 0;

  public VariableCounter(int variableIndex) {
    super(Opcodes.ASM9);
    this.variableIndex = variableIndex;
  }

  @Override
  public void visitVarInsn(int opcode, int var) {
    if (var == variableIndex) {
      if (opcode == Opcodes.ILOAD
      || opcode == Opcodes.FLOAD
      || opcode == Opcodes.LLOAD
      || opcode == Opcodes.DLOAD
      || opcode == Opcodes.ALOAD) {
        readCount++;
      } else if (opcode == Opcodes.ISTORE
      || opcode == Opcodes.FSTORE
      || opcode == Opcodes.LSTORE
      || opcode == Opcodes.DSTORE
      || opcode == Opcodes.ASTORE) {
        writeCount++;
      }
    }
  }

  public int getReadCount() {
    return readCount;
  }

  public int getWriteCount() {
    return writeCount;
  }
}

// usage
byte[] bytecode = ...; // obtain bytecode of class to analyze
ClassReader classReader = new ClassReader(bytecode);
classReader.accept(new ClassVisitor(Opcodes.ASM9) {
```

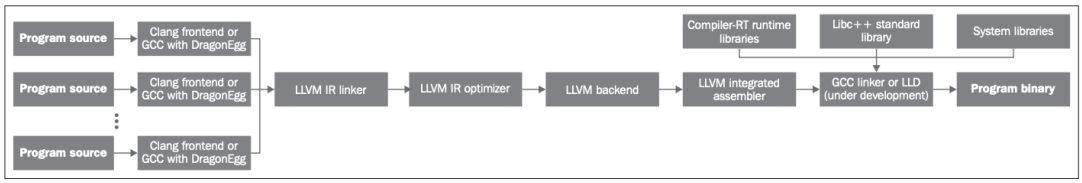
```
46    @Override
47    public MethodVisitor visitMethod(int access,
48    String name,
49    String descriptor,
50    String signature,
51    String[] exceptions) {
52      return new VariableCounter(0); // count accesses to local variable
53                                     // with index 0
54    }
55  }, 0);
```

LLVM и Clang Static Analyzer

LLVM использует следующие структуры данных для работы:

- Abstract Syntax Tree (AST) при транслировании программы с C и C++ в LLVM IR (intermediate representation)
- Directed Acyclic Graph (DAG) при транслировании с LLVM IR в ассемблер
- MCModule при линковке

Рис. 7: Архитектура LLVM



Инструменты, которые могут использоваться при компиляции:

- opt - оптимизирует программы на уровне IR. Принимает и выдаёт файлы в LLVM bitecode
- llc - используется для перевода LLVM bitecode в ассмеблеровские инструкции для специфичной машины (или можно сразу же в объектный файл)
- llvm-mc - используется для транслирования ассмеблеровский инструкций в объектный файл (ELF, MachO, and PE)
- lli - интерпретатор и JIT компилятор для LLVM IR

- llvm-link - используется для линковки нескольких LLVM bitcode файлов
- llvm-as - переводит человекочитаемые LLVM IR файлы (LLVM assemblies) в LLVM bitcodes
- llvm-dis - переводит LLVM bitcodes в LLVM asseblies

Основные библиотеки, которые используются в LLVM и Clang:

- libLLVMCore - библиотека для работы с LLVM IR (классы, функции и т.д.)
- libLLVMAnalysis - библиотека для анализа LLVM IR (lias analysis, dependence analysis, constant folding, loop info, memory dependence analysis, and instruction simplify)
- libLLVMCodeGen - библиотека, которая генерит машинонезависимый код и реализует machine level (the lower level version of the LLVM IR) анализы и трансформации
- libLLVMTarget - библиотека, в которой хранятся абстракции для связывания бэкенда, который реализован в libLLVMCodeGen, и машинозависимой логики, реализованной в libLLVMX86CodeGen/LLVMARMCodeGen/LLVMMipsCo
- libLLVMSupport - предоставляет общие инструменты для работы с ошибками, числами с плавающей точкой, парсинг аргументов из command-line, debugging, работа с файлами и со строками
- libclang - интерфейс для работы с Clang (frontend) на C
- libclangDriver - используется для работы compiler driver tool (парсинг аргуемнтов как в GCC, назаначение jobs и т.д.)
- libclangAnalysis - множество различных анализов для Clang

EOLANG и Polystat