Contents

Название задачи: создание парсера языка программирования

Выполнено командой: YACT

Постановка задачи:

Необходимо создать парсер языка программирования, используемого для реализации и демонстрации различных оптимизаций. В качестве генератора парсера был использован GPPG. Он генерирует bottom-up парсер, распознающий LALR(1) языки с традиционной системой устранения неоднозначности yacc.

Синтасические конструкции языка программирования зафиксированы в документе language-syntax.md

От каких проектов зависит:

- GPPG

Зависимые проекты:

- Алгоритмы анализа потоков данных

Теория

В данном разделе рассматриваются две составляющие алгоритма парсинга: лексер и парсер.

Лексер

Лексер предназначен для разбиения входного потока символов на *лексемы* - отдельные, осмысленные единицы программы. В данной реализации лексер выполняет следующие функции:

- Выделение идентификаторов и целых чисел
- Выделение символьных токенов (>, <=, & и т.п.)
- Выделение ключевых слов
- Формирование текста синтаксической ошибки

Исходный код лексера назодится в файле SimpleLex.lex

Парсер

Парсер принимает на вход поток лексем и формирует *абстрактное синтаксическое дерево* (AST). В данной реализации были использованы следующие типы для термов:

- *SyntaxNode* в качестве базового узла синтаксического дерева
- *Statement* - для узлов, представляющих оператор
- *Expression* - для узлов, представляющих выражение
- *String* - для метки оператора

Исходный код парсера находится в файле SimpleYacc.y ### Входные данные: - Текст программы на языке, описаном в language-syntax.md

Выходные данные:

- Синтаксическое дерево

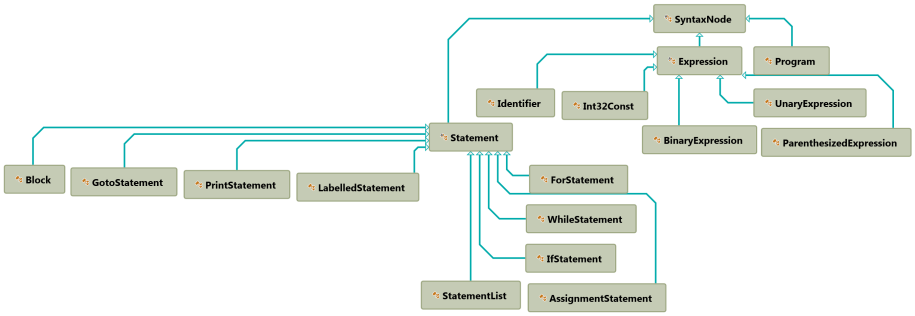
Используемые структуры данных

- List<T> - для хранения списка потомков узла синтаксического дерева

Реализация алгоритма

Формирование синтаксического дерева происходит в действиях, совершаемых парсером в файле SimpleYacc.y. Например, формирование списка операторов выглядит следующим образом:

```
stlist  : statement
    {
        $$ = new StatementList($1);
    }
    | stlist statement
    {
        $$ = ($1 as StatementList).Add($2);
    }
    ;
```



Иерархия синтаксических узлов:

Пример использования

```
string text = @"
a = 40;
b = a + 2;
print(b);";
SyntaxNode root = ParserWrap.Parse(text);
Console.WriteLine(root == null ? "Ошибка" : "Программа распознана");
if (root != null)
```

```
{
    var prettyPrintedProgram = PrettyPrinter.CreateAndVisit(root).FormattedCode;
    Console.WriteLine(prettyPrintedProgram);
}
```

Вывод:

```
a = 40;
b = a + 2;
print(b);
```

Тест

```
string text = @"
a = 40;
b = a + 2;
print(b);";
SyntaxNode root = ParserWrap.Parse(text);

var statements = new StatementList();
statements.Add(new AssignmentStatement(new Identifier("a"), new Int32Const(40)));
statements.Add(new AssignmentStatement(
    new Identifier("b"),
    new BinaryExpression(
        new Identifier("a"),
        Operation.Add,
        new Int32Const(2))));
statements.Add(new PrintStatement(new Identifier("b"), false));
Assert.AreEqual(root, new Program(statements));
```

Название задачи: генерация трехадресного кода по синтаксическому дереву

Выполнено командой: YACT

Постановка задачи:

Необходимо релизовать генератор трехадресного кода по синтаксическому дереву. Трехадресный код состоит из семи видов инструкций:

- x = y op z
- x = y
- x = op z
- **goto** L
- **if** x **goto** L

- no—op
- print

От каких проектов зависит:

- Синтаксическое дерево

Зависимые проекты:

- Базовые блоки

Теория

Перевод синтаксических конструкций в трехадресный код может быть выполнен следующим образом:

- Бинарные выражения сохраняются в отдельных, генерируемых переменных (t0, t1, …)
- Присваивания унарных выражения переводятся напрямую
- Присваивания бинарных выражений преобразуются в присваивания унарных с использованием правил преобразования бинарных выражений
- Циклы и условный оператор могут быть преобразованы с использованием условных/безусловных операторов перехода и пустых операций с метками

Входные данные:

- Синтаксическое дерево

Выходные данные:

- Список команд трехадресного кода

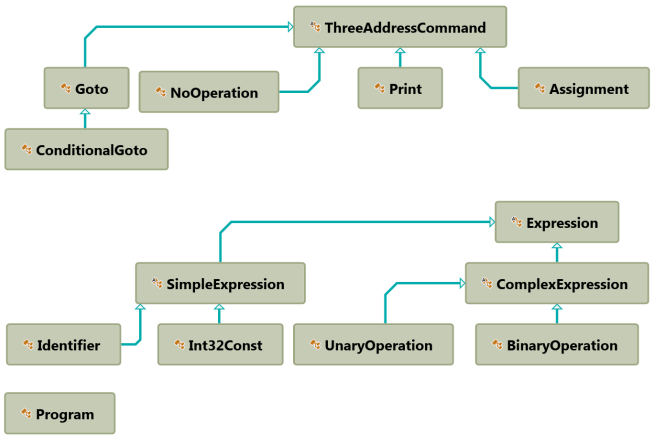
Используемые структуры данных

- List<T> - для хранения списка команд
- Stack<T> - для хранения выражений при преобразовании бинарных выражений

Реализация алгоритма

Генератор трехадресного кода был выполнен с использованием паттерна *визитор*. Для каждого узла синтаксического дерева были определены методы его обработки и в них осуществлялось формирование трехадресных команд.

Исходный код генератора расположен в ThreeAddressCodeGenerator.cs



Иерархия трехадресных команд

Пример использования

```
text = @"
a = 1;
b = 1;
c = a + b;
for i = 1 .. 3
    c = c — 1;
";
SyntaxNode root = ParserWrap.Parse(text);
if (root != null)
{
    var tac = ThreeAddressCodeGenerator.CreateAndVisit(root);
    Console.WriteLine(string.Join(Environment.NewLine, tac.Program.Commands.Select(x
        => x.ToString())));
}
```

Вывод:

```
a = 1
b = 1
t0 = a + b
```

```
c = t0
i = 1
$GL_1: goto $GL_2 if i > 3
t1 = c — 1
c = t1
i = i + 1
goto $GL_1
$GL_2: <no—op>
```

Тест

```
string text = @"
a = 2;
while a > 1
a = a — 1;";
SyntaxTree.SyntaxNodes.SyntaxNode root = ParserWrap.Parse(text);
var threeAddressCode = ThreeAddressCodeGenerator.CreateAndVisit(root).Program;

var expectedCommands = new List<ThreeAddressCommand>
{
    new Assignment("a", new Int32Const(2)),
    new Assignment("t0", new BinaryOperation("a", Operation.Greater, 1)),
    new ConditionalGoto("$GL_2", new BinaryOperation("t0", Operation.Equal, 0)) { Label =
        "$GL_1" },
    new Assignment("t1", new BinaryOperation("a", Operation.Subtract, 1)),
    new Assignment("a", new Identifier("t1")),
    new Goto("$GL_1"),
    new NoOperation("$GL_2")
};

CollectionAssert.AreEqual(threeAddressCode.Commands, expectedCommands);
```

Вычисление множеств Use(B) и Def(B) для активных переменных

Выполнено командой:

PiedPiper (Бергер Анна, Колесников Сергей)

От каких проектов зависит:

1. Базовые блоки

Зависимые проекты:

1. Итерационный алгоритм для активных переменных

Теория

В точке p **x** является активной переменной, если существует путь, проходящий через p, начинающийся присваиванием, заканчивается ее использованием и на всем промежутке нет других присваиваний переменной **x**. - Def(B) – множество переменных, определенных в базовом блоке до любого их использования.

Можно использовать более простое определение Def(B) – множество переменных, определенных в базовом блоке. При таком изменении определения результат работы итерационного алгоритма не изменится. - Use(B) – множество переменных, определенных в базовом блоке до любого их определения.

Входные данные:

- Базовый блок (BasicBlock)

Выходные данные:

- Пара множеств Use(B) и Def(B)

Используемые структуры данных Dictionary, ISet>> SetStorage - словарь, сопоставляющий номеру базового блока его пару его множеств Use(B) и Def(B)

Реализация алгоритма

Метод получения множеств Use(B) и Def(B) по базовому блоку. Он сохраняет полученные множества в SetStorage для избежания повторных вычислений при повторном вызове для блока. C# **public** Tuple<ISet<**string**>, ISet<**string**>> GetDefUseSetsByBlock(BasicBlock block) { **if** (!SetStorage.Keys.Contains(block.BlockId)) { SetStorage.Add(block.BlockId, CreateDefUseSets(block)); } **return** SetStorage[block.BlockId]; } Метод, проходящий по списку команд базового блока и заполняющий множества Use(B) и Def(B). Множество Def(B) пополняется, если встретилась команда типа Assignment. Для пополнения множества Use(B) вызывается вспомогательная функция ExpressionParser. "‘ C# private Tuple, ISet> CreateDefUseSets(BasicBlock block) { ISet Def = new HashSet(); ISet Use = new HashSet();

```
foreach (var command in block.Commands)
{
 if (command.GetType() == typeof(Assignment))
 {
  Def.Add(((Assignment)command).Target.Name);
  ExpressionParser(((Assignment)command).Value, Def, Use);
```

```
 }
 if (command.GetType() == typeof(ConditionalGoto))
 {
  ExpressionParser(((ConditionalGoto)command).Condition, Def, Use);
 }
 if (command.GetType() == typeof(Print))
 {
  ExpressionParser(((Print)command).Argument, Def, Use);
 }
}
return new Tuple<ISet<string>, ISet<string>>(Def, Use);
```

} “‘

Рекурсивная функция, позволяющая обработать части выражений типа Expression и пополнить множество Use(B) C# **private void** ExpressionParser(Expression expr, ISet<**string**> Def, ISet<**string**> Use) { **if** (expr.GetType() == **typeof**(BinaryOperation)) { ExpressionParser(((BinaryOperation)expr).Left, Def, Use); ExpressionParser(((BinaryOperation)expr).Right, Def, Use); } **if** (expr.GetType() == **typeof**(Identifier)) { **if** (!Def.Contains(((Identifier)expr).Name)) { Use.Add(((Identifier)expr).Name); } } **if** (expr.GetType() == **typeof**(UnaryOperation)) { ExpressionParser(((UnaryOperation)expr).Operand, Def, Use); } } ### Пример использования DefUseBlockCalculator DefUseCalc = **new** DefUseBlockCalculator(); var UseDefTuple = DefUseCalc.GetDefUseSetsByBlock(block); var Def = UseDefTuple.Item1; var Use = UseDefTuple.Item2;

Тест Программа:

```
a = 4;
b = 4;
c = a + b;
if 1
   a = 3;
else
   b = 2;
print(c);
```

Базовый блок BlockId = 0

```
Commands:
 a = 4
 b = 4
 t0 = a + b
 c = t0

Def(B) = {a, b, t0, c}
Use(B) = {}
```

Базовый блок BlockId = 3

```
Commands:
 $GL_1: <no—op>
 print c

Def(B) = {}
Use(B) = {c}
```

Название задачи: Iterative Algorithm Parameter for DeadAlive-Variables

Выполнено командой: EndFrame (Аммаев Саид, Пирумян Маргарита)

Постановка задачи:

Необходимо реализовать класс, являющийся параметром обобщенного итерационного алгоритма. Класс должен наследоваться от абстрактного класса BasicIterativeAlgorithmParameters, и в нём необходимо задать поля ForwardDirection, отвечающее за порядок обхода блоков, FirstValue, хранящее значение первого(последнего) блока, StartingValue, хранящее начальное приближение всех блоков, а также методы GatherOperation, являющийся оператором сбора, и TransferFunction, являющийся передаточной функцией.

От каких проектов зависит:

- BasicIterativeAlgorithmParameters
- SetIterativeAlgorithmParameters
- BasicBlock
- DefUseBlockCalculator

Зависимые проекты:

- IterativeAlgorithm

Теория

Переменная x - активная в точке p, если существует путь, проходящий через p, начинающийся присваиванием и заканчивающийся ее использованием, и на всем промежутке нет других присваиваний переменной x. defB - множество переменных, определенных в B до любого их использования useB - множество переменных, использующихся в B до любого их

определения. fB - передаточная функция для блока B. fB(OUT[B]) = IN[B]. IN[B] = useB U (OUT[B] - defB).

Входные данные:

Выходные данные:

- DeadAliveIterativeAlgorithmParameters : SetIterativeAlgorithmParameters

Используемые структуры данных

- IEnumerable> blocks - множество переменных
- BasicBlock block - базовый блок
- ISet input - IN[B]

Реализация алгоритма

```
1  public class DeadAliveIterativeAlgorithmParameters :
       SetIterativeAlgorithmParameters<string>
2  {
3      public override ISet<string> GatherOperation(IEnumerable<ISet<string>> blocks)
4      {
5          ISet<string> union = SetFactory.GetSet((IEnumerable<string>)blocks.First());
6          /* Uпо( всем потомкам) IN[B] */
7          foreach (var block in blocks.Skip(1))
8          {
9              union.UnionWith(block);
10         }
11         return union;
12     }
13     public override ISet<string> GetGen(BasicBlock block)
14     {
15         /* Gen == Use */
16         DefUseBlockCalculator DefUseCalc = new DefUseBlockCalculator();
17         return DefUseCalc.GetDefUseSetsByBlock(block).Item2;
18     }
19     public override ISet<string> GetKill(BasicBlock block)
20     {
21         /* Kill = Def */
22         DefUseBlockCalculator DefUseCalc = new DefUseBlockCalculator();
23         return DefUseCalc.GetDefUseSetsByBlock(block).Item1;
24     }
25     public override ISet<string> TransferFunction(ISet<string> input, BasicBlock block)
```

```
26      {
27          /* useB U (input \ defB) */
28          return SetFactory.GetSet(GetGen(block).Union(input.Except(GetKill(block))));
29      }
30      /* Направление вверх */
31      public override bool ForwardDirection { get { return false; } }
32      /* OUTВыход[] := пустое множество */
33      public override ISet<string> FirstValue { get { return SetFactory.GetSet<string>(); } }
34      /* Начальное приближение — пустое множество */
35      public override ISet<string> StartingValue { get { return SetFactory.GetSet<string>();
        } }
```

Пример использования

```
1 SyntaxNode root = ParserWrap.Parse(text);
2 Graph graph = new
        Graph(BasicBlocksGenerator.CreateBasicBlocks(ThreeAddressCodeGenerator.CreateAndVisit(root).Program));
3 var deadAliveVars = IterativeAlgorithm.Apply(graph, new
        DeadAliveIterativeAlgorithmParameters());
```

Тест

```
1   a = 2;
2   b = 3;
3 1: c = a + b;
4 2: a = 3;
5   b = 4;
6 3: c = a;
```

```
1 OUT[0] = {a, b}
2 OUT[1] = {}
3 OUT[2] = {a}
4 OUT[3] = {}
```

Передаточная функция для задачи распространения констант

Выполнено командой:

EndFrame (Аммаев Саид, Пирумян Маргарита)

Постановка задачи:

Необходимо реализовать функцию для задачи распространения констант.

От каких проектов зависит:

1. Трехадресного кода
2. Базового блока

Зависимые проекты:

1. Итерационный алгоритм

Теория

```
После
 применения передаточной функции к значению потока данных получаются новые.Поток

 данных — mНовое
 значение потока данных — m'Свойства
 значения потока данных и  правила вычисления передаточной функции:
1. m1 ^ m2 = m <=> m1(v) ^ m2(v) = m(v), для любых v
2. m1 < m2 <=> m1(v) < m2(v) , для любых v
  fs(m) = m', где s — команда трехадресного кода
  2.1. если s — не  присваивание, то m' = m
  2.2. если s — присваивание, то для любого v != x : m'(v) = m(v)
3. если v = x
  3.1. x := c
     m'(x) = c
  3.2. x := y + z
     m'(x) = m(y) + m(z), если m(y) и m(z) — const
     m'(x) = NAC, если m(y) или m(z) — NAC
     m'(x) = UNDEF — в остальных случаях
  3.3. x := f(...), где f — оператор вызова функции
     m'(x) = NAC
```

Входные данные:

- Dictionary input - значение потока данных
- BasicBlock block - базовый блок(ББЛ)
- int commandNumber - номер команды в ББЛ

Выходные данные:

- Dictionary - новое значение потока данных

Реализация алгоритма:

“‘ C# /* реализация передаточной функции в классе public class ConstantsPropagationParameters : CompositionIterativeAlgorithmParameters> */

public override Dictionary CommandTransferFunction(Dictionary input, BasicBlock block, int commandNumber) { //получение команды по номеру из ББЛ ThreeAddressCommand command = block.Commands[commandNumber]; //если присваивание if (command.GetType() == typeof(Assignment)) { string newValue = NAC; Expression expr = (command as Assignment).Value; if (expr.GetType() == typeof(Int32Const) || expr.GetType() == typeof(Identifier)) newValue = getConstantFromSimpleExpression(input, (expr as SimpleExpression)); //если унарная операция else if (expr.GetType() == typeof(UnaryOperation)) { UnaryOperation operation = (expr as UnaryOperation); newValue = calculateVal(getConstantFromSimpleExpression(input, operation.Operand), operation.Operation); } //если бинарная операция else if (expr.GetType() == typeof(BinaryOperation)) { BinaryOperation operation = (expr as BinaryOperation); newValue = calculateVal(getConstantFromSimpleExpression(input, operation.Left), getConstantFromSimpleExpression(input, operation.Right), operation.Operation); } string leftOperand = (command as Assignment).Target.Name; input[leftOperand] = newValue; } //если не присваивание, вернется входной поток данных return input; }

```
//вспомогательная функция, которая возвращает константу по значению входного
    потока данных и выражению
string getConstantFromSimpleExpression(Dictionary<string, string> input,
    SimpleExpression expr)
{
    string result = NAC;
    if (expr.GetType() == typeof(Int32Const))
        result = (expr as Int32Const).ToString();
    else if (expr.GetType() == typeof(Identifier))
    {
        string var = (expr as Identifier).ToString();
        if (!input.ContainsKey(var))
            input[var] = UNDEF;
        result = input[var];
    }
    return result;
}

...
```

} “‘

//вспомогательные функции calculateVal реализованы другой командой

Пример использования

```
Dictionary<string, string> m = new Dictionary(/* ... */);
BasicBlock block = new BasicBlock(/*...*/);
Dictionary<string, string> m2 = CommandTransferFunction(m, block, 0)
```

Тест

Программа

```
x = y + z;
```

```
m: x = UNDEF; y = ”2”; z = ”3”;
m’: x = ”5”; y = ”2”; z = ”3”;
```

Базовые блоки и алгоритм их получения.

Выполнено командой:

EndFrame (Аммаев Саид, Пирумян Маргарита)

Постановка задачи:

Необходимо реализовать класс Базовый блок(ББЛ). Необходимо реализовать класс Список Базовых блоков. Необходимо реализовать алгоритм построения списка ББЛ по имеющемуся списку трехадресных команд.

От каких проектов зависит:

1. Трехадресного кода

Зависимые проекты:

1. Граф управления потока

Теория

Базовый блок(ББЛ) - максимальная последовательность, идущих друг за другом команд, удовлетворяющий младующим свойствам: 1. Поток управления может входить в блок только через первую команду; 2. Управление покидает блок без останова или ветвления за исключением, возможно, последней команды.

Алгоритм нахождения: 1. Определить команды-лидеры: - первая команда программы; - любая команда, на которую есть переход; - любая команда, следующая за командой перехода; 2. Сформировать ББЛ - набор команд от лидера до лидера.(первый включается, второй нет)

Входные данные:

- Объект класса Program из ThreeAddressCode.Model, в котором хранится список трехадресных команд (program)

Выходные данные:

- Список базовых блоков (BasicBlocksList)

Используемые структуры данных: - BasicBlock - класс ББЛ (созданный нами класс) - BasicBlocksList - класс списка ББЛ (созданный нами класс)

- BasicBlocksList basicBlocks - список ББЛ
- List commands - список трехадресных команд (ThreeAddressCommand структура из ThreeAddressCode.Model)
- Dictionary labels - словарь соответствия команды в трехадресном коде и метки, присвоенной ей
- List firstCommandsOfBlocks - список для номеров комманд, являющимися первыми в ББЛ
- List lastCommandsOfBlocks - список для номеров комманд, являющимися последними в ББЛ
- int[] BlockByFirstCommand - соответствие номера блока номеру команды в трехадресном коде, которая является первой командой в ББЛ
- int[] BlockByLastCommand - соответствие номера блока номеру команды в трехадресном коде, которая является последней командой в ББЛ
- List[] PreviousBlocksByBlockID - список номеров блоков, входящих в каждый ББЛ
- List[] NextBlocksByBlockID - списко номеров блоков, исходящих из каждого ББЛ

Реализация алгоритма:

“‘ C# // построение списка ББЛ по списку трхадресных команд public static BasicBlocksList CreateBasicBlocks(Program program) { BasicBlocksList basicBlocks = new BasicBlocksList(); List commands = program.Commands; Dictionary labels = new Dictionary(); List firstCommandsOfBlocks = new List(); List lastCommandsOfBlocks = new List();

```
    //поиск командлидеров— и присваивание меток всем коммандам(labels)

    //первая комманда в программе
    firstCommandsOfBlocks.Add(0);
    for (int i = 0; i < commands.Count; ++i)
    {
        ThreeAddressCommand currentCommand = commands[i];
        //если есть метка в трехадресном коде
        if (currentCommand.Label != null)
        {
            labels[currentCommand.Label] = i;
            //если не первая команда в программе
```

```
            if (i > 0)
            {
                //команда с меткой
                firstCommandsOfBlocks.Add(i);
                //команда предшествующая команде с меткой
                lastCommandsOfBlocks.Add(i - 1);
            }
        }
        //если есть переход
        if (currentCommand is Goto && i < commands.Count - 1)
        {
            //следующая команда за командой перехода
            firstCommandsOfBlocks.Add(i + 1);
            //сама команда перехода
            lastCommandsOfBlocks.Add(i);
        }
    }
    lastCommandsOfBlocks.Add(commands.Count - 1);
    firstCommandsOfBlocks = firstCommandsOfBlocks.Distinct().ToList();
    lastCommandsOfBlocks = lastCommandsOfBlocks.Distinct().ToList();

    int[] BlockByFirstCommand = new int[commands.Count];
    int[] BlockByLastCommand = new int[commands.Count];
    for (int i = 0; i < firstCommandsOfBlocks.Count; ++i)
        BlockByFirstCommand[firstCommandsOfBlocks[i]] = i;
    for (int i = 0; i < lastCommandsOfBlocks.Count; ++i)
        BlockByLastCommand[lastCommandsOfBlocks[i]] = i;

    List<int>[] PreviousBlocksByBlockID = new List<int>[commands.Count];
    List<int>[] NextBlocksByBlockID = new List<int>[commands.Count];
    for (int i = 0; i < commands.Count; ++i)
    {
        PreviousBlocksByBlockID[i] = new List<int>();
        NextBlocksByBlockID[i] = new List<int>();
    }

    //формирование списков номеров блоков, входящих и исходящих из каждого ББЛ
    foreach (var currentNumOfLastCommand in lastCommandsOfBlocks)
    {
        ThreeAddressCommand currentCommand =
     commands[currentNumOfLastCommand];
        int numOfCurrentBlock = BlockByLastCommand[currentNumOfLastCommand];
        if ((currentCommand.GetType() != typeof(Goto)) && (currentNumOfLastCommand <
     commands.Count - 1))
        {
            int numOfNextBlock = numOfCurrentBlock + 1;
```

```
            NextBlocksByBlockID[numOfCurrentBlock].Add(numOfNextBlock);
            PreviousBlocksByBlockID[numOfNextBlock].Add(numOfCurrentBlock);
        }
        if (currentCommand is Goto)
        {
            int numOfNextBlock = BlockByFirstCommand[labels[(currentCommand as
    Goto).GotoLabel]];
            NextBlocksByBlockID[numOfCurrentBlock].Add(numOfNextBlock);
            PreviousBlocksByBlockID[numOfNextBlock].Add(numOfCurrentBlock);
        }
    }

    //добавление команд в каждый ББЛ
    for (int i = 0; i < firstCommandsOfBlocks.Count; ++i)
    {
        BasicBlock block = new BasicBlock();
        block.Commands = new
    List<ThreeAddressCommand>(commands.Take(lastCommandsOfBlocks[i] +
    1).Skip(firstCommandsOfBlocks[i]).ToList());
        basicBlocks.Blocks.Add(block);
        basicBlocks.BlockByID[block.BlockId] = block;
    }

    //добавление полученных списков входящих и исходящих ББЛ в каждый блок
    for (int i = 0; i < basicBlocks.Count(); ++i)
    {
        for (int j = 0; j < PreviousBlocksByBlockID[i].Count; ++j)
            PreviousBlocksByBlockID[i][j] =
    basicBlocks.Blocks[PreviousBlocksByBlockID[i][j]].BlockId;
        for (int j = 0; j < NextBlocksByBlockID[i].Count; ++j)
            NextBlocksByBlockID[i][j] = basicBlocks.Blocks[NextBlocksByBlockID[i][j]].BlockId;
        basicBlocks.Blocks[i].InputBlocks.AddRange(PreviousBlocksByBlockID[i]);
        basicBlocks.Blocks[i].OutputBlocks.AddRange(NextBlocksByBlockID[i]);
    }

    return basicBlocks;
  }
  ...
```

} “‘

Пример использования

```
...
var b = BasicBlocksGenerator.CreateBasicBlocks(threeAddressCode);
```

```
...
```

Тест

Программа

```
b = 1;
if 1
  b = 3;
else
  b = 2;
```

Список базовых блоков

```
BlockId = 0
Commands:
  b = 1
  goto $GL_2 if 1 == 0
InputBlocksNumbers = {}
OutputBlocksNumbers = {1; 2}
BlockId = 1
Commands:
  b = 3
  goto $GL_1
InputBlocksNumbers = {0}
OutputBlocksNumbers = {3}
BlockId = 2
Commands:
  $GL_2: <no—op>
  b = 2
InputBlocksNumbers = {0}
OutputBlocksNumbers = {3}
BlockId = 3
Commands:
  $GL_1: <no—op>
InputBlocksNumbers = {1; 2}
OutputBlocksNumbers = {}
```

Построение графа потока управления CFG (Control Flow Graph)

Выполнено командой:

PiedPiper (Бергер Анна, Колесников Сергей)

От каких проектов зависит:

1. Базовые блоки

Зависимые проекты:

1. Построение трехадресного кода по графу потока управления
2. Итерационный алгоритм (и вспомогательные алгоритмы для его работы)
3. Построение дерева доминаторов

Теория

Граф потока управления – множество всех возможных путей исполнения программы, представленное в виде графа. В графе потока управления каждый узел графа соответствует базовому блоку. Ребро из блока B в блок C идет тогда и только тогда, когда первая команда блока C может следовать непосредственно за последней командой блока B. Тогда говорим, что B – предшественник C, а C – преемник B.

Зачастую к графу добавляются два узла, именуемые входом и выходом и не соответствующие выполнимым промежуточным командам. Существует ребро от входа к первому выполнимому узлу графа, т.е. к базовому блоку, который начинается с первой команды промежуточного кода. Если последняя команда программы не является безусловным переходом, то выходу предшествует блок, содержащий эту последнюю команду программы. В противном случае таковым предшествующим блоком является любой базовый блок, имеющий переход к коду, не являющемуся частью программы.

Входные данные:

- Список базовых блоков (BasicBlocksList)

Выходные данные:

- Граф потока управления (Graph)

Используемые структуры данных - BidirectionalGraph> - граф потока управления (структура данных из пакета QuickGraph) - Dictionary blockMap - соответствие между узлами графа и уникальными идентификаторами базовых блоков (blockId)

Реализация алгоритма

“‘ C# // построение графа потока управления по списку базовых блоков public Graph(BasicBlocksList listBlocks) { CFG.AddVertexRange(listBlocks.Blocks);

```
1 foreach (BasicBlock block in listBlocks.Blocks)
2 {
```

```
    blockMap.Add(block.BlockId, block);
}

foreach (var block in listBlocks.Blocks)
{
    foreach (var numIn in block.InputBlocks)
    {
        CFG.AddEdge(new Edge<BasicBlock>(this.getBlockById(numIn), block));
    }
}
...
```

} “‘ Для удобства использования графа в нем реализованы следующие методы: - public BasicBlock getBlockById(int id) – по идентификатору базового блока возвращается базовый блок - узел графа - public BasicBlocksList getChildren(int id) – по идентификатору базового блока возвращается список базовых блоков-преемников - public BasicBlocksList getParents(int id) – по идентификатору базового блока возвращается список базовых блоков-предшественников - BasicBlock getRoot() – точка входа в программу - public int GetCount() – количество вершин в графе - public IEnumerable> GetEdges() – список ребер в графе - public IEnumerable GetVertices() – список вершин в графе - public bool Contains(BasicBlock block) – проверка, содержится ли базовый блок в графе - public bool IsAncestor(int id1, int id2) – проверка является ли блок с blockId = id1

Пример использования

```
...
var b = BasicBlocksGenerator.CreateBasicBlocks(threeAddressCode);
Graph g = new Graph(b);
...
```

Тест

Программа

```
b = 1;
if 1
  b = 3;
else
  b = 2;
```

Список базовых блоков

```
BlockId = 0
Commands:
   b = 1
   goto $GL_2 if 1 == 0
```

```
InputBlocksNumbers = {}
OutputBlocksNumbers = {1; 2}
BlockId = 1
Commands:
  b = 3
  goto $GL_1
InputBlocksNumbers = {0}
OutputBlocksNumbers = {3}
BlockId = 2
Commands:
  $GL_2: <no—op>
  b = 2
InputBlocksNumbers = {0}
OutputBlocksNumbers = {3}
BlockId = 3
Commands:
  $GL_1: <no—op>
InputBlocksNumbers = {1; 2}
OutputBlocksNumbers = {}
```

Граф потока управления

```
0:
<——
——> 1 2
1:
<—— 0
——> 3
2:
<—— 0
——> 3
3:
<—— 1 2
——>
```

Название задачи: Построение трехадресного кода по CFG

Выполнено командой:

Yet yet another team(Горелов Антон, Кочерга Михаил)

Постановка задачи:

Требуется реализовать алгоритм, который по данному графу потока управления восстанавливает текст программы в трехадресном коде.

От каких проектов зависит:

- Граф управления потока(CFG)

Зависимые проекты:

- Отсутствует

Теория

Трехадресный код - это последовательность операторов одного из следующих видов. x = y op z x = y //команды копирования x = op y //x, y, z - имена, константы или временные объекты. goto L if x goto L if x goto L if False x goto L **Трехадресный код** представляет собой линеаризованное представление синтаксического дерева или ориентированного ациклического графа, в котором явные имена соответствуют внутренним узлам графа.

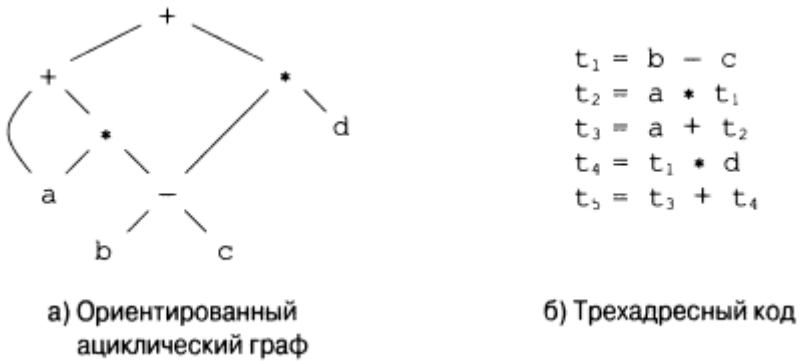


Figure 1:

Ориентированный ациклический граф и соответствующий ему трехадресный код.

Входные данные:

- Граф потока управления

Выходные данные:

- Трехадресный код.

Используемые структуры данных

- BidirectionalGraph - граф потока управления (структура данных из пакета Quick-Graph)
- Dictionary blockMap - соответствие между узлами графа и уникальными идентификаторами базовых блоков (blockId)
- List - для хранения результата
- Stack - для хранения порядка обработки ББл.
- HashSet - для хранения обрабтанных ББл.

Реализация алгоритма

```
public List<ThreeAddressCommand>  transformToThreeAddressCode()
      {
         var res = new List<ThreeAddressCommand>();
         var done = new HashSet<BasicBlock>();
         var stack = new Stack<BasicBlock>();
         stack.Push(getBlockById(0));
         BasicBlock cur = null;

         while (done.Count < this.GetCount())
         {
            cur = stack.Pop();
            done.Add(cur);

            foreach (var c in cur.Commands)  { res.Add(c); }

            switch (cur.OutputBlocks.Count)
            {
               case 0:
                  continue;
                case 1:
                  //добавить в стек выходной ББЛ
                case 2:
                  //если в else последний оператор goto, то добавить в стек ББл, на
      который указывает goto
                  //добавить в стек ветку else
                  //добавить в стек ветку if
               default:
                  throw new Exception("There cannot be more than two output blocks!");
```

```
            }
        }
```

Пример использования

```
SyntaxTree.SyntaxNodes.SyntaxNode root = ParserWrap.Parse(sourseText);
            var sourceThreeAddressCode =
    ThreeAddressCodeGenerator.CreateAndVisit(root).Program;
            BasicBlocksList bbl =
    BasicBlocksGenerator.CreateBasicBlocks(sourceThreeAddressCode);
            Graph cfg = new Graph(bbl);
            List<ThreeAddressCommand> resultThreeAddressCode =
    cfg.transformToThreeAddressCode();
```

Тест

```
if 1
  if 1
  {
    a = 5;
    a = 1;
  }
  else
  {
    for i = 1..5
      a = 1;
  }

while 1
  a = 1;
a = 2;

for i = 1 .. 5
  for j = 1 .. 5
    a = 1;

for i = 1 .. 5
  for j = 1 .. 5
    if 1
      a = 1;
    else
    {
      a = 1;
      a = 2;
    }
```

Построение множества gen S и kill S для одной команды

Выполнено командой:

NoName (Скиба Кирилл, Борисов Сергей)

От каких проектов зависит:

1. Граф управления потока

Зависимые проекты:

1. Вычисление передаточной функции, как композиции передаточных функций для одной команды
2. Множества gen B и kill B(на основе передаточной функции по явным формулам)

Постановка задачи:

Необходимо реализовать алгоритм, вычисляющий множества *gen* и *kill* для одной команды

Теория

Множеством *gen S* называется множество определений, генерируемых в базовом блоке *B*. Множеством *kill S* называется множество всех прочих определений той же переменной во всей остальной программе.

Эти множества входят в передаточную функцию для задачи о достигающих определениях, поэтому перед запуском итерационного алгоритма для нее необходимо вычислить эти множества для каждого блока, а для этого нужно вычислить эти множества для каждой команды в этом блоке.

Входные данные

- Базовый блок в котором содержится команда
- ID команды

Выходные данные

- Множество gen
- Множество kill

Используемые структуры данных

- CommandNumber Gen - множество gen (так как команда одна, то это номер этой команды)
- ISet<CommandNumber> - множество kill
- Dictionary<**string**, ISet<CommandNumber>> - хранилище команд
- Graph graph - граф потока управления

Реализация

В реализации данного модуля был использован пакет NuGet QuickGraph.

```
1  // класс содержащий множества gen и kill.
2  public class GenKillOneCommand
3  {
4      public CommandNumber Gen { get; set; }
5
6      public ISet<CommandNumber> Kill { get; set; }
7
8      public GenKillOneCommand(CommandNumber gen, ISet<CommandNumber> kill)
9      {
10         Gen = gen;
11         Kill = kill;
12     }
13 }
14
15 // Алгоритм построения множеств gen и kill
16 public class GenKillOneCommandCalculator
17 {
18     private CommandNumber Gen;
19     private ISet<CommandNumber> Kill;
20     private Dictionary<string, ISet<CommandNumber>> CommandStorage;
21     private Graph graph;
22
23     public GenKillOneCommandCalculator(Graph g)
24     {
25         graph = g;
26         CommandStorage = new Dictionary<string, ISet<CommandNumber>>();
27         foreach (BasicBlock block in graph)
28             for (int i = 0; i < block.Commands.Count(); i++)
29             {
30                 var command = block.Commands[i];
31                 if (command.GetType() == typeof(Assignment))
32                     if (CommandStorage.ContainsKey((command as Assignment).Target.Name))
```

```
                    CommandStorage[(command as Assignment).Target.Name].Add(new
    CommandNumber(block.BlockId, i));
                else
                    CommandStorage.Add((command as Assignment).Target.Name,
                        SetFactory.GetSet(new CommandNumber(block.BlockId, i)));
            }
        Kill = SetFactory.GetSet<CommandNumber>();
    }

    public GenKillOneCommand CalculateGenAndKill(BasicBlock block,
        ThreeAddressCommand command)
    {
        Kill.Clear();
        if (command.GetType() == typeof(Assignment))
        {
            Gen = new CommandNumber(block.BlockId, block.Commands.IndexOf(command));
            string target = (command as Assignment).Target.Name;
            foreach (var c in CommandStorage[target])
                if (c.BlockId != block.BlockId && c.CommandId !=
    block.Commands.IndexOf(command))
                    Kill.Add(new CommandNumber(c.BlockId, c.CommandId));
        }
        return new GenKillOneCommand(Gen, Kill);
    }

    public GenKillOneCommand CalculateGenAndKill(int blockId, int commandId)
    {
        var block = graph.getBlockById(blockId);
        return CalculateGenAndKill(block, block.Commands[commandId]);
    }

    public GenKillOneCommand CalculateGenAndKill(BasicBlock block, int commandId)
    {
        return CalculateGenAndKill(block, block.Commands[commandId]);
    }
}
```

Пример использования

```
calculator = new GenKillOneCommandCalculator(g);
GenKillOneCommand genKill = calculator.CalculateGenAndKill(block, commandNumber);
```

Построение множеств gen B и kill B (на основе передаточной функции по явным формулам)

Выполнено командой:

Ampersand (Золотарёв Федор, Маросеев Олег)

От каких проектов зависит:

1. Множества gen S и kill S для одной команды

Зависимые проекты:

1. Итерационный алгоритм
2. Достигающие определения

Постановка задачи:

В данной задаче требуется разработать класс, который реализует функции построения множеств Gen и Kill по входному графу потока. Вычисления множеств осуществляются на основе передаточной функции по явным формулам.

Теория

Множество gen(B) - множество операций в блоке B. Множество kill(B) - множество остальных определений этих переменных в программе.

Формулы для построения множеств: Gen(B) = gen[n] + (gen[n-1] - kill[n]) + (gen[n-2] - kill[n-1] - kill[n]) + … + (gen[1] - kill[2] - … - kill[n]) Kill(B) = kill[1] + … + kill[n]

Используемые структуры данных

- List<**int**> OutputBlocks - номера выходных базовых блоков

Реализация

```
1 class ExplicitTransferFunction : SetIterativeAlgorithmParameters<CommandNumber>
2   {
3     private GenKillOneCommandCalculator commandCalc;
4
5     public ExplicitTransferFunction(Graph g)
```

```
6       {
7           commandCalc = new GenKillOneCommandCalculator(g);
8       }
9
10      public override ISet<CommandNumber>
        GatherOperation(IEnumerable<ISet<CommandNumber>> blocks)
11      {
12          ISet<CommandNumber> res = SetFactory.GetSet<CommandNumber>();
13          foreach (var command in blocks)
14              res.UnionWith(command);
15          return res;
16      }
17
18      public override ISet<CommandNumber> GetGen(BasicBlock block)
19      {
20          //list of kill—sets for every command in block
21          List<ISet<CommandNumber>> listKill = new
        List<ISet<CommandNumber>>(block.Commands.Count);
22          foreach (var command in block.Commands)
23              listKill.Add(commandCalc.CalculateGenAndKill(block, command).Kill);
24
25          ISet<CommandNumber> genB = SetFactory.GetSet<CommandNumber>();
26
27          for (int i = block.Commands.Count — 1; i >= 0; i——)
28          {
29              ISet<CommandNumber> genS = SetFactory.GetSet<CommandNumber>();
30              genS.Add(commandCalc.CalculateGenAndKill(block, block.Commands[i]).Gen);
31              for (int j = i; j < block.Commands.Count — 1; j++)
32              {
33                  genS.IntersectWith(listKill[i]);
34              }
35              genB.UnionWith(genS);
36          }
37
38          return genB;
39      }
40
41      public override ISet<CommandNumber> GetKill(BasicBlock block)
42      {
43          ISet<CommandNumber> killB = SetFactory.GetSet<CommandNumber>();
44          foreach (var command in block.Commands)
45              killB.UnionWith(commandCalc.CalculateGenAndKill(block, command).Kill);
46          return killB;
47      }
48  }
```

Пример использования

```
public override ISet<Expression> TransferFunction(ISet<Expression> input, BasicBlock
    block)
{
    var kill = GetKill(block).Cast<Identifier>();
    var result = SetFactory.GetSet(
        input.Where(inputExpression =>
        !kill.Any(inputExpression.HasIdentifiedSubexpression)));
    result.UnionWith(GetGen(block));

    return result;
    /*
     * Non—LINQ realization
    var difference = SetFactory.GetSet();
    var foundInKill = false;
    foreach (var inputExpression in input)
    {
        foreach (var killExpression in kill)
        {
            if (!inputExpression.HasIdentifiedSubexpression(killExpression as Identifier))
                continue;
            foundInKill = true;
            break;
        }
        if (!foundInKill)
            difference.Add(inputExpression);
        foundInKill = false;
    }
    return SetFactory.GetSet(GetGen(block).Union(difference));
    */
}
```

Вычисление передаточной функции, как композиции передаточных функций для одной команды

Выполнено командой:

Google Dogs (Александр Василенко, Кирилл Куц)

От каких проектов зависит:

1. Получение базовых блоков
2. Получение множеств *gen* и *kill* для одной команды

Зависимые проекты:

1. Итерационный алгоритм для достигающих определений

Постановка задачи:

Используя множества *gen* и *kill* для одной команды, необходимо вычислить множества *gen* и *kill* для каждого блока.

Теория

Предположим, что блок *B* состоит из инструкций *s1*, …, *sn* в указанном порядке. Если *s1* — первая инструкция базового блока *B*, то IN[*B*] = IN[*s1*]. Аналогично, если *sn*— последняя инструкция базового блока *B*, то OUT[*B*] = OUT[*sn*]. Передаточная функция базового блока *B*, которую мы обозначим как *fB*, может быть получена как композиция передаточных функций инструкций базового блока. Пусть *fsi* — передаточная функция для инструкции *si*. Тогда *fB* = *fsn* ○ … ○ *fs2* ○ *fs1*.

Входные данные

- Базовые блоки

Выходные данные

- Множества *gen* и *kill* для блока

Используемые структуры данных

- GenKillOneCommandCalculator commandCalc - вычислитель множеств *gen* и *kill* для одной команды

Реализация алгоритма

```
public override ISet<CommandNumber> CommandTransferFunction
  (ISet<CommandNumber> input, BasicBlock block,
    int commandNumber)
{
  GenKillOneCommand genKill =
    calculator.CalculateGenAndKill(block, commandNumber);
  var result = SetFactory.GetSet<CommandNumber>(input);
  result.ExceptWith(genKill.Kill);
```

```
    result.UnionWith(genKill.Gen == null
      ? SetFactory.GetSet<CommandNumber>()
      : SetFactory.GetSet(genKill.Gen));
    return result;
}
```

Пример использования

```
public override T TransferFunction
  (T input, BasicBlock block)
{
    return Enumerable.Range(0, block.Commands.Count)
      .Aggregate(input, (result, c) =>
        CommandTransferFunction(result, block, c));
}
```

Итерационный алгоритм для доступных выражений

Выполнено командой:

Google Dogs (Александр Василенко, Кирилл Куц)

От каких проектов зависит:

1. Построение базовых блоков
2. Построение трехадресного кода
3. Построение множеств *e_gen* и *e_kill*

Зависимые проекты:

отсутствуют

Постановка задачи

Написать класс-наследник общего итерационного алгоритма, переопределить методы для поиска доступных выражений

Теория

Выражение *x* + *y* **доступно** (available) в точке *p*, если любой путь от входного узла к *p* вычисляет *x* + *y* и после последнего такого вычисления до достижения *p* нет последующих присваиваний переменным *x* и *y*. Когда речь идет о *доступных выражениях*, мы говорим,

что блок уничтожает выражение *x* + *y*, если он присваивает (или может присваивать) *x* и *y* и после этого не вычисляет *x* + *y* заново. Блок генерирует выражение *x* + *y*, если он вычисляет *x* + *y* и не выполняет последующих переопределений *x* и *y*.

Мы можем найти доступные выражения методом, напоминающим метод вычисления достигающих определений. Предположим, что *U* — “универсальное” множество всех выражений, появляющихся в правой части одной или нескольких инструкций программы. Пусть для каждого блока *B* множество IN[*B*] содержит выражения из *U*, доступные в точке непосредственно перед началом блока *B*, а OUT[*B*] — такое же множество для точки, следующей за концом блока *B*. Определим *e_genB* как множество выражений, генерируемых *B*, а *e_killB* — как множество выражений из *U*, уничтожаемых в *B*. Заметим, что множества IN, OUT, *e_gen* и *e_kill* могут быть представлены в виде битовых векторов. Неизвестные множества IN и OUT связаны друг с другом и с известными *e_gen* и *e_kill* следующими соотношениями:

OUT[Вход] =

□

и для всех базовых блоков *B*, отличных от входного:

OUT[*B*] = *e_genB*

□

(IN[*B*] - *e_killB*)

IN[*B*] =

∩

OUT[*P*], где *P* - предшественник *B*

Реализация алгоритма

```
1  public AvailableExpressionsCalculator(Graph g)
2  {
3     Graph = g;
4  }
5
6  public override ISet<Expression> GatherOperation
7   (IEnumerable<ISet<Expression>> blocks)
8  {
9     ISet<Expression> intersection =
          SetFactory.GetSet((IEnumerable<Expression>)blocks.First());
10    foreach (var block in blocks.Skip(1))
11      intersection.IntersectWith(block);
12    return intersection;
13 }
14
```

```
15 public override ISet<Expression> GetGen(BasicBlock block)
16 {
17    return SetFactory.GetSet(
18       block.Commands
19         .OfType<Assignment>()
20         .Select(x => x.Value));
21 }
22
23 public override ISet<Expression> GetKill(BasicBlock block)
24 {
25    return SetFactory.GetSet(
26      block.Commands
27       .OfType<Assignment>()
28       .Select(x => x.Target as Expression));
29 }
30
31 public override ISet<Expression> TransferFunction
32  (ISet<Expression> input, BasicBlock block)
33 {
34    var kill = GetKill(block).Cast<Identifier>();
35    var result = SetFactory.GetSet(
36       input.Where(inputExpression =>
37       !kill.Any(inputExpression.HasIdentifiedSubexpression)));
38    result.UnionWith(GetGen(block));
39    return result;
40 }
41
42 public override bool ForwardDirection => true;
43
44 public override ISet<Expression> FirstValue => SetFactory.GetSet();
45
46 public override ISet<Expression> StartingValue
47 {
48    get
49    {
50       ISet<Expression> result = SetFactory.GetSet();
51       foreach (BasicBlock b in Graph)
52         foreach (ThreeAddressCommand c in b.Commands)
53           if (c.GetType() == typeof(Assignment))
54           {
55              result.Add((c as Assignment).Value);
56              result.Add((c as Assignment).Target);
57           }
58        return result;
59    }
60 }
```

Пример использования

```
var availableExprs = IterativeAlgorithm.Apply(g,
  new AvailableExpressionsCalculator(g));
var outExpressions = availableExprs.Out.Select(
  pair => $"{pair.Key}: {string.Join(", ",
  pair.Value.Select(ex => ex.ToString()))}");
```

Название задачи: вычисление `e_gen(b)` и `e_kill(b)` для доступных выражений

Выполнено командой: YACT

Постановка задачи:

Необходимо по данному графу потока данных найти множества e_gen и e_kill для каждого базового блока. Эти данные используются алгоритмом анализа доступных выражений.

От каких проектов зависит:

- CFG
- Базовые блоки
- Обобщенный итеративный алгоритм
- Трехадресный код

Зависимые проекты:

- Анализатор доступных выражений

Теория

Доступные выражения – алгоритм анализа, определяющий для каждой точки в программе множество выражений, которые не требуется перевычислять. Такие выражения называются *доступными* в данной точке. Для того, чтобы быть доступными в точке программы, операнды выражения не должны быть изменены на любом пути от вхождения этого выражения до точки программы.

Входные данные:

- Базовый блок

Stmt	Gen	Kill
x := a + b	a + b	
y := a * b	a * b	
a := a + 1		a + 1, a + b, a * b

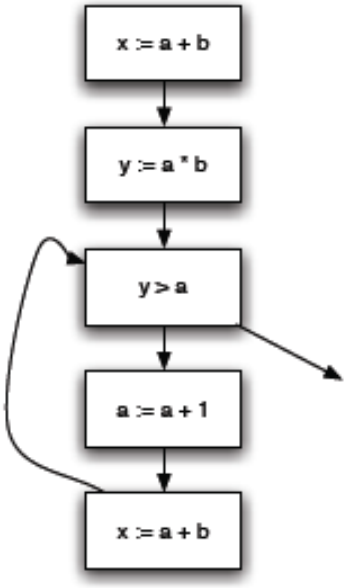


Figure 2: alt text

Выходные данные:

- Множество выражений

Используемые структуры данных

- HashSet<Expression> – множество выражений

Реализация алгоритма

```
// Метод получения множества Gen
public override ISet<Expression> GetGen(BasicBlock block)
{
    return SetFactory.GetSet(
        block.Commands
            .OfType<Assignment>() // оставляем только присваивания
            .Select(x => x.Value)); // из них берем правые части
}

// Метод получения множества Kill
public override ISet<Expression> GetKill(BasicBlock block)
```

```
{
    return SetFactory.GetSet(
        block.Commands
            .OfType<Assignment>() // оставляем только присваивания
            .Select(x => x.Target as Expression)); // из них берем левые части
}
```

Пример использования

Трехадресный код программы:

```
1: <no—op>
t0 = a + b
x = t0
2: <no—op>
t1 = a * b
y = t1
3: <no—op>
t2 = a + 1
a = t2
```

Вызов алгоритма

```
// — — — —
// фомирование графа g
// — — — —
...
var availableExprs = IterativeAlgorithm.Apply(g, new AvailableExpressionsCalculator(g));
var outExpressions = availableExprs.Out.Select(
    pair => $"{pair.Key}: {string.Join(", ", pair.Value.Select(ex => ex.ToString()))}");
```

Вывод outExpressions:

```
// Доступные выражения
0: a + b, t0
1: a + b, t0, a * b, t1
2: t0, t1, a + 1, t2
```

Тест

```
string programText = @"
a = 4;
b = 4;
c = a + b;
if 1
  a = 3;
```

```
else
  b = 2;
print(c);
";

// Формирование CFG и вызов алгоритма
SyntaxNode root = ParserWrap.Parse(programText);
var threeAddressCode = ThreeAddressCodeGenerator.CreateAndVisit(root).Program;
var basicBlocks = BasicBlocksGenerator.CreateBasicBlocks(threeAddressCode);
Graph g = new Graph(basicBlocks);

var availableExprs = IterativeAlgorithm.Apply(g, new AvailableExpressionsCalculator(g));
var outExpressions = availableExprs.Out.Select(
    pair => $"{pair.Key}: {string.Join(", ", pair.Value.Select(ex => ex.ToString()))}");

// Сравнение с эталоном
int startIndex = availableExprs.Out.Keys.Min();
Assert.IsTrue(availableExprs.Out[startIndex]
    .SetEquals(
        new HashSet<Expression>(new Expression[]
        {
            new Int32Const(4),
            new BinaryOperation(new identifier("a"), Operation.Add, new identifier("b")),
            new identifier("t0")
        })));

Assert.IsTrue(availableExprs.Out[startIndex + 1]
    .SetEquals(
        new HashSet<Expression>(new Expression[]
        {
            new Int32Const(4),
            new Int32Const(3),
            new identifier("t0")
        })));

Assert.IsTrue(availableExprs.Out[startIndex + 2]
    .SetEquals(
        new HashSet<Expression>(new Expression[]
        {
            new Int32Const(4),
            new Int32Const(2),
            new identifier("t0")
        })));

Assert.IsTrue(availableExprs.Out[startIndex + 3]
    .SetEquals(
```

```
53      new HashSet<Expression>(new Expression[]
54      {
55          new Int32Const(4),
56          new identifier("t0")
57      }));
```

Итерационный алгоритм для задачи распространения констант

Выполнено командой:

Ampersand (Золотарёв Федор, Маросеев Олег)

От каких проектов зависит:

1. Общий итерационный алгоритм
2. Передаточная функция для задачи распостронения констант

Постановка задачи:

Создать класс, реализующий итерационный алгоритм для задачи распостронения констант

Теория

Распространение констант – хорошо известная проблема глобального анализа потока данных. Цель распространения констант состоит в обнаружении величин, которые являются постоянными при любом возможном пути выполнения программы, и в распространении этих величин так далеко по тексту программы, как только это возможно. Выражения, чьи операнды являются константами, могут быть вычислены на этапе компиляции. Поэтому использование алгоритмов распространения констант позволяет компилятору выдавать более компактный и быстрый код.

Исходный алгоритм:

□ t □ tuples lattice(t) = T unvisited(t) = true

Visit all basic blocks B in the program Visit all tuples t within B if unvisited(t) then propagate(t)

propagate (tuple t) unvisited(t) = false if ssa_link(t) ≠ 0 then if unvisited(ssa_link(t)) then propagate(ssa_link(t)) lattice(t) = lattice(t) П lattice(ssa_link(t)) endif if unvisited(left(t)) then propagate(left(t)) if unvisited(right(t)) then propagate(right(t)) case on type (t) constant C: lattice(t) = C arithmetic operation: if all operands have constant lattice value then lattice(t) = arithmetic

result of lattice values of operands else lattice(t) = □ endif store: lattice(t) = lattice(RHS) φ-function: lattice(t) = Π of φ-arguments of t γ-function: if lattice(predicate) = C then lattice(t) = lattice value of γ-argument corresponding to C else lattice(t) = Π of all γ-arguments of t endif μ-function: lattice(t) = □ η-function: lattice(t) = lattice(η-argument) default: lattice(t) = □ end case end propagate

Используемые структуры данных

- Dictionary<Edge<BasicBlock>, EdgeType> - словарь классифицированных ребер

Реализация

```
public class ConstantPropagation : IOptimization
  {
     public Graph Apply(Graph graph)
     {
        var constants = IterativeAlgorithm.Apply(graph, new
     ConstantsPropagationParameters());

        return graph;
     }
  }

public class ConstantsPropagationParameters :
     CompositionIterativeAlgorithmParameters<Dictionary<string, string>>
  {
     public const string NAC = "NAC";
     public const string UNDEF = "UNDEF";

     public override bool ForwardDirection { get { return true; } }

     public override Dictionary<string, string> FirstValue { get { return new
     Dictionary<string, string>(); } }

     public override Dictionary<string, string> StartingValue { get { return new
     Dictionary<string, string>(); } }

     public override Dictionary<string, string>
     CommandTransferFunction(Dictionary<string, string> input, BasicBlock block, int
     commandNumber)
     {
        ThreeAddressCommand command = block.Commands[commandNumber];
        if (command.GetType() == typeof(Assignment))
        {
```

```
27              string newValue = NAC;
28              Expression expr = (command as Assignment).Value;
29              if (expr.GetType() == typeof(Int32Const) || expr.GetType() == typeof(Identifier))
30                  newValue = getConstantFromSimpleExpression(input, (expr as
    SimpleExpression));
31              else if (expr.GetType() == typeof(UnaryOperation))
32              {
33                  UnaryOperation operation = (expr as UnaryOperation);
34                  newValue = calculateVal(getConstantFromSimpleExpression(input,
    operation.Operand), operation.Operation);
35              }
36              else if (expr.GetType() == typeof(BinaryOperation))
37              {
38                  BinaryOperation operation = (expr as BinaryOperation);
39                  newValue = calculateVal(getConstantFromSimpleExpression(input,
    operation.Left), getConstantFromSimpleExpression(input, operation.Right),
    operation.Operation);
40              }
41              string leftOperand = (command as Assignment).Target.Name;
42              input[leftOperand] = newValue;
43          }
44          return input;
45      }
46
47      string getConstantFromSimpleExpression(Dictionary<string, string> input,
    SimpleExpression expr)
48      {
49          string result = NAC;
50          if (expr.GetType() == typeof(Int32Const))
51              result = (expr as Int32Const).ToString();
52          else if (expr.GetType() == typeof(Identifier))
53          {
54              string var = (expr as Identifier).ToString();
55              if (!input.ContainsKey(var))
56                  input[var] = UNDEF;
57              result = input[var];
58          }
59          return result;
60      }
61
62      public override bool AreEqual(Dictionary<string, string> t1, Dictionary<string,
    string> t2)
63      {
64          return t1.Count == t2.Count && t1.Keys.All(key => t2.ContainsKey(key) && t1[key]
    == t2[key]);
65      }
```

```
66      string calculateVal(string x1, Operation op)
67      {
68          return calculateVal(x1, "0", op);
69      }
70
71      string calculateVal(string x1, string x2, Operation op)
72      {
73          if (x1 == NAC || x2 == NAC)
74              return NAC;
75          else if (x1 == UNDEF || x2 == UNDEF)
76              return UNDEF;
77          else
78          {
79              int lx = int.Parse(x1);
80              int rx = int.Parse(x2);
81              return ArithmeticOperationCalculator.Calculate(op, lx, rx).ToString();
82          }
83      }
84      string gatherVal(string x1, string x2)
85      {
86          if (x1 == x2 || x2 == UNDEF || x1 == NAC)
87              return x1;
88          else if (x1 == UNDEF || x2 == NAC)
89              return x2;
90          else
91              return NAC;
92      }
93      public override Dictionary<string, string>
        GatherOperation(IEnumerable<Dictionary<string, string>> blocks)
94      {
95          return blocks.Aggregate(new Dictionary<string, string>(), (result, x) =>
96          {
97              foreach(KeyValuePair<string, string> pair in x)
98                  result[pair.Key] = result.ContainsKey(pair.Key) ? gatherVal(result[pair.Key],
        pair.Value) : pair.Value;
99              return result;
100         });
101     }
102 }
```

Пример использования

```
1  [TestClass]
2    public class ConstantPropagationIterativeAlgorithmTests
3    {
4        [TestMethod]
```

```
5       public void ConstantsPropagation1()
6       {
7           string programText = @"
8   a = 4;
9   b = 4;
10  c = a + b;
11  if 1
12    a = 3;
13  else
14    b = 2;
15  print(c);
16  ";
17          SyntaxNode root = ParserWrap.Parse(programText);
18          var threeAddressCode = ThreeAddressCodeGenerator.CreateAndVisit(root).Program;
19          Trace.WriteLine(threeAddressCode);
20
21          var basicBlocks = BasicBlocksGenerator.CreateBasicBlocks(threeAddressCode);
22          Trace.WriteLine(Environment.NewLine + "Базовые блоки");
23          Trace.WriteLine(basicBlocks);
24
25          Trace.WriteLine(Environment.NewLine + "Управляющий граф программы");
26          Graph g = new Graph(basicBlocks);
27          Trace.WriteLine(g);
28
29
30
31          Trace.WriteLine(Environment.NewLine + "Распространение констант");
32          var consts = IterativeAlgorithm.Apply(g, new ConstantsPropagationParameters());
33          var outConsts = consts.Out.Select(
34              pair => $"{pair.Key}: {string.Join(", ", pair.Value.Select(ex => ex.ToString()))}");
35
36          foreach (var outInfo in outConsts)
37          {
38              Trace.WriteLine(outInfo);
39          }
40
41          int startIndex = consts.Out.Keys.Min();
42          Assert.IsTrue(consts.Out[startIndex]["a"]=="4" &&
43              consts.Out[startIndex]["b"] == "4" &&
44              consts.Out[startIndex]["t0"] == "8" &&
45              consts.Out[startIndex]["c"] == "8");
46
47          Assert.IsTrue(consts.Out[startIndex + 1]["a"] == "3" &&
48              consts.Out[startIndex + 1]["b"] == "4" &&
49              consts.Out[startIndex + 1]["t0"] == "8" &&
50              consts.Out[startIndex + 1]["c"] == "8");
```

```
51
52          Assert.IsTrue(consts.Out[startIndex + 2]["a"] == "4" &&
53              consts.Out[startIndex + 2]["b"] == "2" &&
54              consts.Out[startIndex + 2]["t0"] == "8" &&
55              consts.Out[startIndex + 2]["c"] == "8");
56
57          Assert.IsTrue(consts.Out[startIndex + 3]["a"] == "NAC" &&
58              consts.Out[startIndex + 3]["b"] == "NAC" &&
59              consts.Out[startIndex + 3]["t0"] == "8" &&
60              consts.Out[startIndex + 3]["c"] == "8");
61      }
62
63      [TestMethod]
64      public void ConstantsPropagation2()
65      {
66          string programText = @"
67    e=10;
68    c=4;
69    d=2;
70    a=4;
71    if 0
72        goto 2;
73    a=c+d;
74    e=a;
75    goto 3;
76  2:  a=e;
77  3:  t=0;
78  ";
79          SyntaxNode root = ParserWrap.Parse(programText);
80          var threeAddressCode = ThreeAddressCodeGenerator.CreateAndVisit(root).Program;
81          Trace.WriteLine(threeAddressCode);
82
83          var basicBlocks = BasicBlocksGenerator.CreateBasicBlocks(threeAddressCode);
84          Trace.WriteLine(Environment.NewLine + "Базовые блоки");
85          Trace.WriteLine(basicBlocks);
86
87          Trace.WriteLine(Environment.NewLine + "Управляющий граф программы");
88          Graph g = new Graph(basicBlocks);
89          Trace.WriteLine(g);
90
91          Trace.WriteLine(Environment.NewLine + "Распространение констант");
92          var consts = IterativeAlgorithm.Apply(g, new ConstantsPropagationParameters());
93          var outConsts = consts.Out.Select(
94              pair => $"{pair.Key}: {string.Join(", ", pair.Value.Select(ex => ex.ToString()))}");
95
96          foreach (var outInfo in outConsts)
```

```
97          {
98              Trace.WriteLine(outInfo);
99          }
100
101         int startIndex = consts.Out.Keys.Min();
102         Assert.IsTrue(consts.Out[startIndex]["a"] == "4" &&
103             consts.Out[startIndex]["e"] == "10" &&
104             consts.Out[startIndex]["d"] == "2" &&
105             consts.Out[startIndex]["c"] == "4");
106
107         Assert.IsTrue(consts.Out[startIndex + 1]["a"] == "4" &&
108             consts.Out[startIndex + 1]["e"] == "10" &&
109             consts.Out[startIndex + 1]["d"] == "2" &&
110             consts.Out[startIndex + 1]["c"] == "4");
111
112         Assert.IsTrue(consts.Out[startIndex + 2]["e"] == "6" &&
113             consts.Out[startIndex + 2]["c"] == "4" &&
114             consts.Out[startIndex + 2]["d"] == "2" &&
115             consts.Out[startIndex + 2]["a"] == "6" &&
116             consts.Out[startIndex + 2]["t0"] == "6");
117
118         Assert.IsTrue(consts.Out[startIndex + 3]["a"] == "10" &&
119             consts.Out[startIndex + 3]["e"] == "10" &&
120             consts.Out[startIndex + 3]["d"] == "2" &&
121             consts.Out[startIndex + 3]["c"] == "4");
122
123         Assert.IsTrue(consts.Out[startIndex + 4]["e"] == "NAC" &&
124             consts.Out[startIndex + 4]["c"] == "4" &&
125             consts.Out[startIndex + 4]["d"] == "2" &&
126             consts.Out[startIndex + 4]["t"] == "0" &&
127             consts.Out[startIndex + 4]["t0"] == "6");
128     }
129 }
```

Итерационный алгоритм

Выполнено командой:

Ampersand (Золотарёв Федор, Маросеев Олег)

От каких проектов зависит:

1. Граф управления потока

Зависимые проекты:

1. Итерационный алгоритм для задачи распространения констант
2. Итерационный алгоритм для доступных выражений
3. Итерационный алгоритм для активных переменных

Постановка задачи:

В данной задаче требуется разработать класс, который реализует общий итерационный алгоритм

Теория

пока out(B) меняется { для каждого B - базового блока { in(B) = / out(p);// где p - непосредственно предшествующее out(B) = func(in(B)); } }

Реализация

```
public static class IterativeAlgorithm
  {
     public static IterativeAlgorithmOutput<V> Apply<T, V>(Graph graph,
     BasicIterativeAlgorithmParameters<V> param) where T :
     BasicIterativeAlgorithmParameters<V>
     {
        IterativeAlgorithmOutput<V> result = new IterativeAlgorithmOutput<V>();

        foreach (BasicBlock bb in graph)
           result.Out[bb.BlockId] = param.StartingValue;

        bool changed = true;
        while (changed)
        {
           changed = false;
           foreach (BasicBlock bb in graph)
           {
              result.In[bb.BlockId] = param.GatherOperation((param.ForwardDirection ?
     graph.getParents(bb.BlockId) : graph.getAncestors(bb.BlockId)).Blocks.Select(b =>
     result.Out[b.BlockId]));
              V newOut = param.TransferFunction(result.In[bb.BlockId], bb);
              changed = changed || !param.Compare(result.Out[bb.BlockId], newOut);
              result.Out[bb.BlockId] = param.TransferFunction(result.In[bb.BlockId], bb);
           }
        }
```

```
        if (!param.ForwardDirection)
            result = new IterativeAlgorithmOutput<V> { In = result.Out, Out = result.In };
        return result;
    }
  }
```

Пример использования

```
[TestClass]
  public class AvailableExpressionTest
  {
    [TestMethod]
    public void AvailableExpressionsTest()
    {
      string programText = @"
a = 4;
b = 4;
c = a + b;
if 1
 a = 3;
else
 b = 2;
print(c);
";
        SyntaxNode root = ParserWrap.Parse(programText);
        var threeAddressCode = ThreeAddressCodeGenerator.CreateAndVisit(root).Program;
        Trace.WriteLine(threeAddressCode);

        var basicBlocks = BasicBlocksGenerator.CreateBasicBlocks(threeAddressCode);
        Trace.WriteLine(Environment.NewLine + "Базовые блоки");
        Trace.WriteLine(basicBlocks);

        Trace.WriteLine(Environment.NewLine + "Управляющий граф программы");
        Graph g = new Graph(basicBlocks);
        Trace.WriteLine(g);



        Trace.WriteLine(Environment.NewLine + "Доступные выражения");
        var availableExprs = IterativeAlgorithm.Apply(g, new
      AvailableExpressionsCalculator(g));
        var outExpressions = availableExprs.Out.Select(
          pair => $"{pair.Key}: {string.Join(", ", pair.Value.Select(ex => ex.ToString()))}");

        foreach (var outInfo in outExpressions)
        {
```

```
38              Trace.WriteLine(outInfo);
39          }
40
41          int startIndex = availableExprs.Out.Keys.Min();
42          Assert.IsTrue(availableExprs.Out[startIndex]
43              .SetEquals(new Expression[]
44                  {
45                      new Int32Const(4),
46                      new BinaryOperation(new identifier("a"), Operation.Add, new identifier("b")),
47                      new identifier("t0")
48                  }));
49
50          Assert.IsTrue(availableExprs.Out[startIndex + 1]
51              .SetEquals(new Expression[]
52                  {
53                      new Int32Const(4),
54                      new Int32Const(3),
55                      new identifier("t0")
56                  }));
57
58          Assert.IsTrue(availableExprs.Out[startIndex + 2]
59              .SetEquals(
60                  new Expression[]
61                  {
62                      new Int32Const(4),
63                      new Int32Const(2),
64                      new identifier("t0")
65                  }));
66
67          Assert.IsTrue(availableExprs.Out[startIndex + 3]
68              .SetEquals(
69                  new Expression[]
70                  {
71                      new Int32Const(4),
72                      new identifier("t0")
73                  }));
74      }
75  }
```

Построение дерева доминаторов

Выполнено командой:

NoName (Скиба Кирилл, Борисов Сергей)

От каких проектов зависит:

1. Итерационный алгоритм
2. Граф управления потока

Зависимые проекты:

1. Алгоритм классификации ребер графа

Постановка задачи:

Необходимо создать класс для дерева доминаторов и построить это дерево, используя итеративный алгоритм.

Теория

Узел *d* графа потока доминирует над узлом *n*, если любой путь от входного узла графа потока к *n* проходит через *d*. При таком определении каждый узел доминирует над самим собой.

Дерево доминаторов - это дерево, в котором входной узел является корнем, а каждый узел *d* доминирует только над своими потомками в дереве.

Существование деревьев доминаторов следует из свойства доминаторов: каждый узел *n* имеет единственный непосредственный доминатор *m*, который является последним доминатором *n* на любом пути от входного узла до *n*.

Входные данные

- Граф потока управления

Выходные данные

- Дерево доминаторов

Используемые структуры данных

- AdjacencyGraph<**int**, Edge<**int**>> Tree - дерево доминаторов (структура данных из пакета NuGet QuickGraph)
- Dictionary<**int**, **int**> Map - словарь: номер базового блока - его непосредственный доминатор

Реализация

В реализации данного модуля был использован пакет NuGet QuickGraph.

```
// класс дерево доминаторов
public class DominatorsTree : IEnumerable
{
    private AdjacencyGraph<int, Edge<int>> Tree = new AdjacencyGraph<int, Edge<int>>();
    private Dictionary<int, int> Map;

    public DominatorsTree(Graph g)
    {
        Map = ImmediateDominator.FindImmediateDominator(g);

        Tree.AddVertexRange(Map.Keys);

        foreach (int key in Map.Keys.Skip(1))
            Tree.AddEdge(new Edge<int>(Map[key], key));
    }

    public int GetParent(int id)
    {
        return Map[id];
    }

    public List<int> GetAncestors(int id)
    {
        return Map.Where(x => x.Value == id).Select(x => x.Key).ToList();
    }

    public override string ToString()
    {
        string res = "";
        foreach (var v in Tree.Vertices)
            if (Tree.OutEdges(v).Count() > 0)
                foreach (var e in Tree.OutEdges(v))
                    res += v + " --> " + e.Target + "\n";
        return res;
    }

    public IEnumerator GetEnumerator()
    {
        return Map.Values.GetEnumerator();
    }

    public Dictionary<int, int> GetMap()
```

```
43      {
44          return Map;
45      }
46  }
47
48  // Класс непосредственный "доминатор"
49  public static class ImmediateDominator
50  {
51      public static Dictionary<int, int> FindImmediateDominator(Graph g)
52      {
53          var _out = IterativeAlgorithm.IterativeAlgorithm.Apply(g, new
            DominatorsIterativeAlgorithmParametrs(g)).Out;
54
55          int min = _out.Keys.Min();
56
57          return _out.Select(x => new KeyValuePair<int, int>(x.Key,
58                                     x.Key > min ? _out[x.Key].Take(_out[x.Key].Count — 1).Last() : min))
59                                    .ToDictionary(x => x.Key, x => x.Value);
60      }
61  }
62
63  // наследник общего класса параметры "итерационного алгоритма"
64  public class DominatorsIterativeAlgorithmParametrs :
        BasicIterativeAlgorithmParameters<ISet<int>>
65  {
66      private Graph graph;
67
68      public DominatorsIterativeAlgorithmParametrs(Graph g)
69      {
70          graph = g;
71      }
72
73      public override ISet<int> GatherOperation(IEnumerable<ISet<int>> blocks)
74      {
75          ISet<int> intersection = SetFactory.GetSet((IEnumerable<int>)blocks.First());
76          foreach (var block in blocks.Skip(1))
77              intersection.IntersectWith(block);
78
79          return intersection;
80      }
81
82      public override ISet<int> TransferFunction(ISet<int> input, BasicBlock block)
83      {
84          return SetFactory.GetSet<int>(input.Union(new int[] { block.BlockId }));
85      }
86
```

```
    public override bool AreEqual(ISet<int> t1, ISet<int> t2)
    {
        return t1.IsSubsetOf(t2) && t2.IsSubsetOf(t1);
    }

    public override ISet<int> StartingValue { get { return
        SetFactory.GetSet<int>(Enumerable.Range(graph.GetMinBlockId(), graph.Count())); } }

    public override ISet<int> FirstValue { get { return
        SetFactory.GetSet<int>(Enumerable.Repeat(graph.GetMinBlockId(), 1)); } }

    public override bool ForwardDirection { get { return true; } }

}

// Итеративный алгоритм, использующий для построения дерева доминаторов
public static IterativeAlgorithmOutput<V> Apply<V>(Graph graph,
        BasicIterativeAlgorithmParameters<V> param, int[] order = null)
{
    IterativeAlgorithmOutput<V> result = new IterativeAlgorithmOutput<V>();

    foreach (BasicBlock bb in graph)
        result.Out[bb.BlockId] = param.StartingValue;
    IEnumerable<BasicBlock> g = order == null ? graph : order.Select(i =>
        graph.getBlockById(i));
    bool changed = true;
    while (changed)
    {
        changed = false;
        foreach (BasicBlock bb in g)
        {
            BasicBlocksList parents = param.ForwardDirection ? graph.getParents(bb.BlockId) :
        graph.getChildren(bb.BlockId);
            if (parents.Blocks.Count > 0)
                result.In[bb.BlockId] = param.GatherOperation(parents.Blocks.Select(b =>
        result.Out[b.BlockId]));
            else
                result.In[bb.BlockId] = param.FirstValue;
            V newOut = param.TransferFunction(result.In[bb.BlockId], bb);
            changed = changed || !param.AreEqual(result.Out[bb.BlockId], newOut);
            result.Out[bb.BlockId] = param.TransferFunction(result.In[bb.BlockId], bb);
        }
    }
    if (!param.ForwardDirection)
        result = new IterativeAlgorithmOutput<V> { In = result.Out, Out = result.In };
    return result;
```

```
}
```

Пример использования

```
DominatorsTree tree = new DominatorsTree(g);
foreach (Edge<BasicBlock> e in g.GetEdges())
{
   if (dfn[e.Source.BlockId] >= dfn[e.Target.BlockId])
      edgeTypes.Add(e, EdgeType.Retreating);
   else if (g.IsAncestor(e.Target.BlockId, e.Source.BlockId) &&
       tree.GetParent(e.Target.BlockId) == e.Source.BlockId)
      edgeTypes.Add(e, EdgeType.Advancing);
   else
      edgeTypes.Add(e, EdgeType.Cross);
}
```

Тест

```
Программа
**:
b = 1;
if 1
 b = 3;
else
 b = 2;
```

Граф потока управления:

Вывод программы:

Формат вывода: “номер базового блока - его непосредственный доминатор”

0 -> 0
1 -> 0
2 -> 0
3 -> 0

Построение глубинного остовного дерева

Выполнено командой:

PiedPiper (Бергер Анна, Колесников Сергей)

От каких проектов зависит:

1. Граф управления потока

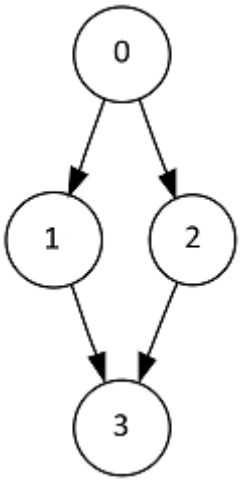


Figure 3:

Зависимые проекты:

1. Нахождение естественных циклов

Теория

Остовное дерево графа состоит из минимального подмножества рёбер графа, таких, что из любой вершины графа можно попасть в любую другую вершину, двигаясь по этим рёбрам. Остовное дерево может быть построено практически любым алгоритмом обхода графа, например поиском в глубину или поиском в ширину. Оно состоит из всех пар рёбер (u,v), таких, что алгоритм, просматривая вершину u, обнаруживает в её списке смежности новую, не обнаруженную ранее вершину v.

Поиск в графе в глубину (depth-first search) однократно посещает все узлы графа, начиная с входного узла и посещая, в первую очередь, насколько это возможно, узлы, максимально удаленные от входного. Путь поиска в глубину образует глубинное остовное дерево (охватывающее вглубь дерево) (depth-first spanning tree — DFST). Обход в прямом прядке посещает узел перед посещением любого из его дочерних узлов, которые затем рекурсивно посещаются в порядке слева направо. Обход в обратном порядке сначала рекурсивно слева направо посещает узлы, дочерние по отношению к текущему, а затем посещает сам текущий узел.

Входные данные:

- Не требуются, остовное дерево строится в процессе построения CFG

Выходные данные:

- Ассоциативный массив

Реализация алгоритма

Узлы графа потока управления обходятся в прямом порядке и в процессе обхода нумеруются.

```
private void dfs(BasicBlock block, Dictionary<BasicBlock, bool> visited, ref int c)
{
 visited[block] = true;
 foreach (var node in getChildren(block.BlockId).Blocks)
 {
  if (!visited[node])
  {
   spanTree.AddEdge(new Edge<BasicBlock>(block, node));
   dfs(node, visited, ref c);
  }
 }
 spanTreeOrder[block.BlockId] = c;
 c——;
}
```

Пример использования

```
var b = BasicBlocksGenerator.CreateBasicBlocks(threeAddressCode);
Graph g = new Graph(b);
Dictionary<int, int> dfn = g.GetDFN();
```

Тест

Программа:

```
b = 1;
if 1
 b = 3;
else
 b = 2;
```

Управляющий граф программы

```
0:
<——
——>  1 2
```

```
1:
<—— 0
——> 3
2:
<—— 0
——> 3
3:
<—— 1 2
——>
```

Новое нумерование

```
3——>4
1——>3
2——>2
0——>1
```

Классификация рёбер графа

Выполнено командой:

Google Dogs (Александр Василенко, Кирилл Куц)

От каких проектов зависит:

1. Построение CFG
2. Построение дерева доминаторов
3. Построение глубинного остовного дерева

Зависимые проекты:

1. Установить, все ли отступающие ребра являются обратными
2. Формирование последовательности областей в восходящем порядке

Постановка задачи:

Необходимо классифицировать ребра графа потока управления на три вида:

1. *Наступающие* (advancing) ребра идут от узла *m* к истинным преемникам *m* в дереве.
2. *Отступающие* (retreating) ребра идут от узла *m* к предку *m* в дереве (возможно, к самому *m*).

3. *Поперечные* (cross) - все остальные ребра.

Пример:

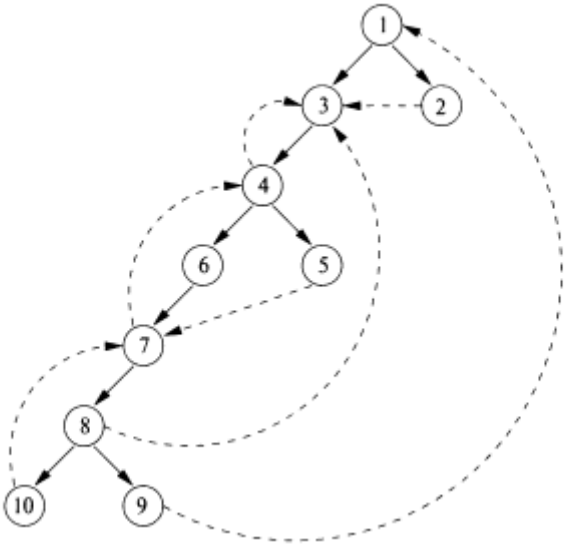


Figure 4:

Рёбра 1

→

3, 6

→

7, 9

→

10 являются наступающими, рёбра 4

→

3, 7

→

4, 8

→

3 являются отступающими, рёбра 2

→

3, 5

→

7 - поперечными.

Теория

Все ребра, попадающие в глубинное остовное дерево, являются *наступающими*. В примере выше такие ребра отмечаются сплошной стрелкой. Однако, если бы на графе потока управления существовало ребро, например, 1

→

4, оно бы тоже считалось *наступающим*, несмотря на то, что не входило бы в DFST.

Ребро *m

→

n* является *отступающим* тогда и только тогда, когда *dfn*[*m*]

≥

dfn[*n*]. Если *m* является потомком *n* в глубинном остовном дереве, то *search*(*m*) завершается до *search*(*n*), поэтому *dfn*[*m*]

≥

dfn[*n*]. *И* наоборот, если *dfn*[*m*]

≥

dfn[*n*], то *search(*m*) завершается до* search*(*n*), или* m* = *n*. Но вызов *search*(*n*) должен начинаться до *search*(*m*), если существует ребро *m

→

n*, иначе тот факт, что* n* является преемником *m*, должен сделать *n* потомком *m* в DFST. Таким образом, время активности *search*(*m*) представляет собой подынтервал времени активности *search*(*m*), откуда следует, что *n* является предком *m* в DFST.

Важным свойством *поперечных* ребер является то, что если изобразить DFST так, чтобы дочерние узлы некоторого узла располагались слева направо в порядке, в котором они добавлялись в дерево, то все *поперечные* ребра будут идти справа налево. В нашем языке *поперечные* ребра могут быть получены только в случае, если на графе потока управления можно найти так называемый “ромб”: то есть из одного базового блока можно попасть в другой (не непосредственный потомок первого) по двум и более путям.

Входные данные

- Граф потока управления

Выходные данные

- Пары “ребро - тип ребра”

Используемые структуры данных

- Dictionary<Edge<BasicBlock>, EdgeType> edgeTypes - хранение результата
- Dictionary<int, int> dfn - порядок узлов
- DominatorsTree tree - дерево доминаторов

Реализация алгоритма

```
public static Dictionary<Edge<BasicBlock>, EdgeType>
    ClassifyEdge(ControlFlowGraph.Graph g)
{
    // Инициализация пустым словарем
    Dictionary<Edge<BasicBlock>, EdgeType> edgeTypes
    = new Dictionary<Edge<BasicBlock>, EdgeType>();
    // Получение порядка узлов
    Dictionary<int, int> dfn = g.GetDFN();
    // Получение дерева доминаторов
    DominatorsTree tree = new DominatorsTree(g);

    // Выполнение для всех ребер графа
    foreach (Edge<BasicBlock> e in g.GetEdges())
    {
        /* Если порядок начала больше порядка конца,
           то ребро отступающее */
        if (dfn[e.Source.BlockId] >= dfn[e.Target.BlockId])
            edgeTypes.Add(e, EdgeType.Retreating);
        // Если предок в дереве, то наступающее
        else if (g.IsAncestor(e.Target.BlockId, e.Source.BlockId)
        && tree.GetParent(e.Target.BlockId) == e.Source.BlockId)
            edgeTypes.Add(e, EdgeType.Advancing);
        // Иначе поперечное
        else
            edgeTypes.Add(e, EdgeType.Cross);
    }

    return edgeTypes;
}
```

Пример использования

```
var edgeClassify = EdgeClassification.ClassifyEdge(g);
foreach (var v in edgeClassify)
  // В консоль выводятся вход и выход обратного ребра, а также тип ребра
  Console.WriteLine(v.Key.Source.BlockId + " —> "
  + v.Key.Target.BlockId + " : " + v.Value);
```

Тест

Программа:

```
b = 1;
if 1
 b = 3;
else
 b = 2;
```

Граф потока управления:

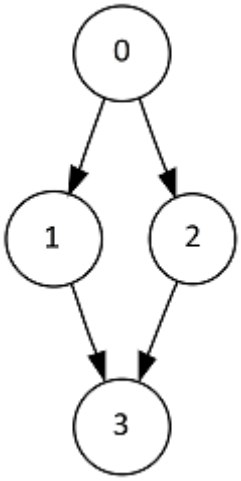


Figure 5:

Ребра *0

→

1* и *0

→

2* являются наступающими, а ребра *1

→

3* и *2

→

3* являются поперечными.

Вывод программы:

Формат вывода: “выход обратного ребра - вход обратного ребра - тип ребра”

0 -> 1 : Advancing
0 -> 2 : Advancing
1 -> 3 : Cross
2 -> 3 : Cross # Название задачи: FindReverseEdges

Выполнено командой: EndFrame (Аммаев Саид, Пирумян Маргарита)

Постановка задачи: Необходимо реализовать функцию, которая находит в заданном графе потока управления обратные ребра.

От каких проектов зависит:

- ControlFlowGraph
- EdgeClassification
- ImmediateDominator

Зависимые проекты:

- CheckRetreatingIsReverse

Теория

> Отступающее ребро в графе потока управления - ребро, идущее от потомка к предку. Вершина графа d доминирует над a, если любой путь от начальной вершины графа до вершины a, проходит через d. Свойства доминирования: d доминирует над d; если d доминирует над b, a b доминирует над a, то d доминирует над a. Отступающее ребро, направленное из a в b, называется обратным, если b доминирует над a

Входные данные:

- Graph - граф потока управления

Выходные данные:

- ISet> - множество дуг базовых блоков

Используемые структуры данных

- Dictionary, EdgeType> ClassifiedEdges - соответствие к ребру его типа
- Dictionary Dominators - соответсвие к узлу графа его непосредственного доминатора
- Dictionary, EdgeType> RetreatingEdges - отступающие ребра со своим типом

Реализация алгоритма

```
1   public static ISet<Edge<BasicBlock>> FindReverseEdges(ControlFlowGraph.Graph g)
2   {
3     ISet<Edge<BasicBlock>> res = SetFactory.GetEdgesSet();//new SortedSet<Edge
    <BasicBlock>>();
4     Dictionary<Edge<BasicBlock>, EdgeType> ClassifiedEdges =
    EdgeClassification.EdgeClassification.ClassifyEdge(g);
5     Dictionary<int, int> Dominators = ImmediateDominator.FindImmediateDominator(g);
6     var RetreatingEdges = ClassifiedEdges.Where(x => x.Value ==
    EdgeType.Retreating);
7     foreach (var edg in RetreatingEdges)
8     {
9       var edge = edg.Key;             // текущее отступающее ребро
10      int key = edge.Source.BlockId;   // текущий узел ставится на начало
    отступающего ребра
11      int value = edge.Target.BlockId;  // конец отступающего ребра
12      bool isReverse = false;          // является ли текущий узел концом
    отступающего ребра
13      /* цикл, пока у текущего узла есть непосредственный доминатор, он не начало
    дерева доминаторов,
14      и текущий узел не попал на конец отступающего ребра */
15      while (Dominators.ContainsKey(key) && Dominators[key] != key && !isReverse)
16      {
17        key = Dominators[key];     // текущий узел перемещается на
    непосредственного доминатора
18        isReverse = (key == value); // конец отступающего доминирует над началом,
    то есть ребро — обратное
19      }
20      if (isReverse)       // если ребро обратное — добавляем
21        res.Add(edge);
22    }
23    return res;
24  }
```

Пример использования

```
var ReverseEdges = FindReverseEdge.FindReverseEdges(g);
```

Тест

```
for i = 1 + 2 * 3 .. 10
  println(i);
```

```
Полученный
 CFG:
0: 1
1: 2, 3
2: 1
3:
```

```
Обратное
 ребро: 2 ——> 1
```

Установить, все ли отступающие ребра являются обратными

Выполнено командой:

Ampersand (Золотарёв Федор, Маросеев Олег)

От каких проектов зависит:

1. Алгоритм классификации ребер графа
2. Алгоритм нахождения обратных ребер графа

Постановка задачи:

В данной задаче требуется разработать класс, который реализует функцию проверки того факта, что все отступающие ребра графа являются обратными.

Теория

Отступающее ребро - ребро от потомков к предку остовного графа.

Ребро от A к B называется обратным, если B доминирует над A.

Используемые структуры данных

- Dictionary<Edge<BasicBlock>, EdgeType> - словарь классифицированных ребер.

Реализация

```
namespace DataFlowAnalysis.IntermediateRepresentation.CheckRetreatingIsReverse
{
    class CheckRetreatingIsReverse
    {
        public static bool CheckReverseEdges(ControlFlowGraph.Graph g)
        {
            Dictionary<Edge<BasicBlock>, EdgeType> ClassifiedEdges =
        EdgeClassification.EdgeClassification.ClassifyEdge(g);
            var RetreatingEdges = ClassifiedEdges.Where(x => x.Value ==
        EdgeType.Retreating).Select(x => x.Key);

            var ReverseEdges = FindReverseEdge.FindReverseEdges(g);

            return ReverseEdges.IsSubsetOf(RetreatingEdges);
        }
    }
}
```

Пример использования

```
[TestClass]
    public class RetreatingNotReverseEdgesTests
    {
        [TestMethod]
        public void RetreatingNotReverseEdges1()
        {
            var programText = File.ReadAllText("./../../RetreatingNotReverseEdgesEx1.txt");


            SyntaxNode root = ParserWrap.Parse(programText);
            Graph graph = new Graph(
                BasicBlocksGenerator.CreateBasicBlocks(
                    ThreeAddressCodeGenerator.CreateAndVisit(root).Program));

            Assert.IsFalse(CheckRetreatingIsReverse.CheckReverseEdges(graph));
        }

        [TestMethod]
        public void RetreatingNotReverseEdges2()
        {
            var programText = File.ReadAllText("./../../RetreatingNotReverseEdgesEx2.txt");

            SyntaxNode root = ParserWrap.Parse(programText);
            Graph graph = new Graph(
```

```
            BasicBlocksGenerator.CreateBasicBlocks(
                ThreeAddressCodeGenerator.CreateAndVisit(root).Program));

        Assert.IsFalse(CheckRetreatingIsReverse.CheckReverseEdges(graph));
    }

    [TestMethod]
    public void RetreatingNotReverseEdges3()
    {
        var programText = File.ReadAllText("./../../RetreatingNotReverseEdgesEx3.txt");

        SyntaxNode root = ParserWrap.Parse(programText);
        Graph graph = new Graph(
            BasicBlocksGenerator.CreateBasicBlocks(
                ThreeAddressCodeGenerator.CreateAndVisit(root).Program));

        Assert.IsFalse(CheckRetreatingIsReverse.CheckReverseEdges(graph));
    }
}
```

Поиск естественных циклов

Выполнено командой:

Google Dogs (Александр Василенко, Кирилл Куц)

От каких проектов зависит:

1. Построение CFG
2. Алгоритм классификации рёбер CFG

Зависимые проекты:

1. Формирование последовательности областей в восходящем порядке

Постановка задачи:

В данной задаче требуется найти на графе потока управления (CFG) *естественные циклы*. Эти циклы обладают двумя важными свойствами: 1. Цикл имеет единственный входной узел - базовый блок, который называется *заголовком*. Данный узел доминирует над всеми узлами этого цикла, в противном случае узел не является единственной точкой входа в цикл. 2. Обязано существовать обратное ребро, которое ведет в *заголовок*. Иначе данная структура не может считаться циклом.

Пример:

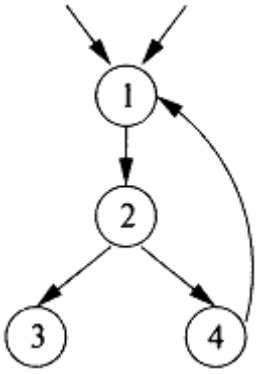


Figure 6:

Узлы 1, 2, 4 образуют естественный цикл.

Теория

Пусть существует обратное ребро *n*

→

d. Для этого ребра естественный цикл определяется как множество узлов, в которое входят: 1. Сам узел *d* 2. Все узлы, которые могут достичь *n*, не проходя через *d*. В данном случае узел *d* является заголовком цикла.

На первом шаге алгоритма необходимо найти все обратные рёбра, далее для каждого ребра выполнить следующий алгоритм:

Алгоритм построения естественного цикла обратной дуги:

Вход: граф потока *G* и обратная дуга *n*

→

d.

Выход: множество *loop*, состоящее из всех узлов естественного цикла *n*

→

d.

Метод: вначале *loop* инициализируется множеством { *n*, *d* }. Узел *d* помечается как “посещеннный”, чтобы поиск не проходил дальше *d*. Выполним поиск в глубину на

обратном графе потока, начиная с *n*. Внесем все посещенные при этом поиске узлы в *loop*. Такая процедура позволяет найти все узлы, достигающие *n*, минуя *d*.

Входные данные

- Граф потока управления

Выходные данные

- Пары “обратное ребро - соответствующее этому ребру множество базовых блоков, входящих в естественный цикл”

Используемые структуры данных

- ISet<int> Loop - множество номеров базовых блоков, входящих в цикл
- Stack<int> Stack - стек
- Dictionary<Edge<BasicBlock>, EdgeType> classify - ассоциативный массив, хранящий классификацию ребер
- Dictionary<Edge<BasicBlock>, ISet<int>> result - ассоциативный массив, хранящий результат

Реализация алгоритма

```
1  // Вспомогательный алгоритм
2  private static void Insert(int m)
3  {
4      if (!Loop.Contains(m))
5      {
6          Loop.Add(m); // Добавление узла в цикл
7          Stack.Push(m); // Добавление узла в стек
8      }
9  }
10
11 // Основной алгоритм
12 public static Dictionary<Edge<BasicBlock>, ISet<int>>
           FindAllNaturalLoops(ControlFlowGraph.Graph g)
13 {
14     var classify = EdgeClassification.EdgeClassification.ClassifyEdge(g); // Выполнение
           классификации рёбер
15     var result = new Dictionary<Edge<BasicBlock>, ISet<int>>(); // Инициализация рёбер
16     foreach (var pair in classify)
17     {
```

```
18      if (pair.Value == EdgeClassification.Model.EdgeType.Retreating) // Алгоритм
          выполняется только для отступающих рёбер в( данном случае, и обратных)
19      {
20        Stack = new Stack<int>(); // Инициализация стека
21        Loop = SetFactory.GetSet(new int[] { pair.Key.Target.BlockId });
22        Insert(pair.Key.Source.BlockId); // Добавление в цикл и стек узлов, входящих в
          обратное ребро
23        while (Stack.Count() > 0)
24        {
25          int m = Stack.Pop();
26          foreach (BasicBlock p in g.getParents(m)) // Добавление в цикл и стек всех
          непосредственных предков узла
27              Insert(p.BlockId);
28        }
29        result.Add(pair.Key, Loop);
30      }
31    }
32    return result;
33  }
```

Пример использования

```
1  var allNaturalLoops = SearchNaturalLoops.FindAllNaturalLoops(g);
2  foreach (var loop in allNaturalLoops)
3  {
4    // В консоль выводятся вход и выход обратного ребра, а также номера блоков,
         входящих в естественный цикл
5    Console.Write(loop.Key.Source.BlockId + " —> " + loop.Key.Target.BlockId + " : ");
6    foreach (int node in loop.Value)
7      Console.Write(node.ToString() + " ");
8    Console.WriteLine("");
9  }
```

Тест

Программа:

```
1  i = 10;
2  1: i = i — 1;
3  if i > 0
4   goto 1;
```

Граф потока управления:

Естественный цикл в данном случае один, в него входят узлы 1 и 2.

Вывод программы:

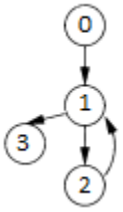


Figure 7:

Формат вывода: “выход обратного ребра - вход обратного ребра - список узлов из цикла”

2 -> 1 : 1 2 # Разработка класса “область” и разновидности: класс “область тела” и класс “область цикла”

Выполнено командой:

NoName (Скиба Кирилл, Борисов Сергей)

От каких проектов зависит:

1. Базовые блоки и алгоритм получения

Зависимые проекты:

1. Формирование последовательности областей в восходящем порядке
2. Разработка класса для хранения передаточных функций для алгоритма анализа на основе областей
3. Восходящая часть алгоритма анализа на основе областей
4. Нисходящая часть алгоритма анализа на основе областей

Постановка задачи:

В данной задаче требуется разработать класс “область”. Также необходимо разработать разновидности класса “область”: класс “область тела” и класс “область цикла”. Так как базовый блок рассматривается как область, то необходимо также разработать класс “область-лист”

Теория

В процессе анализа на основе областей программа рассматривается как иерархия которые грубо можно считать частями графа потока, имеющими единственную точку

входа. Каждая инструкция в блочно структурированной программе является областью, поскольку поток управления может достичь инструкции только через ее начало. Каждый уровень вложенности инструкций соответствует уровню в иерархии областей.

Формально область графа потока представляет собой набор узлов *N* и ребер *E*, таких, что 1. Существует заголовок *h

□

N*, доминирующий над всеми узлами в* N*; 2. Если некоторый узел* m* может достичь узла *n

□

N*, минуя* h*, то* m* также входит в *N*; 3. *E* - множество всех ребер потока управления между узлами *n1* и *n2* из *N*, за исключением, возможно, некоторых ребер, входящих в *h*.

Тело цикла *L* (все узлы и ребра, за исключением обратных ребер к заголовку) замещаем узлом, представляющим область *R*. Ребра, ведущие к заголовку *L*, входят теперь в узел R. Ребро из любого выхода из *L* замещается ребром от *R* в то же самое место назначения. Однако если ребро является обратным, то оно становится петлей у *R*. Назовем *R областью тела*.

Единственное отличие области тела от области цикла заключается в том, что последняя включает обратные ребра к заголовку цикла L.

Используемые структуры данных

- List<int> OutputBlocks - номера выходных базовых блоков

Реализация

```
// Абстрактный класс область""
public abstract class Region
{
    public abstract List<int> OutputBlocks { get; }
}

// Вспомогательный класс IntermediateRegion
public class IntermediateRegion : Region
{
    public BasicBlock Header { get; set; }

    List<int> outputBlocks;
    public override List<int> OutputBlocks { get { return outputBlocks; } }
    public IntermediateRegion(BasicBlock header, List<int> outputBlocks)
    {
        Header = header;
```

```
        this.outputBlocks = outputBlocks;
    }

    protected bool Equals(IntermediateRegion other)
    {
        return Equals(Header, other.Header) &&
         OutputBlocks.SequenceEqual(other.OutputBlocks);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;
        if (obj.GetType() != this.GetType()) return false;
        return Equals((IntermediateRegion) obj);
    }

    public override int GetHashCode()
    {
        unchecked
        {
            return ((Header != null ? Header.GetHashCode() : 0) * 397) ^ (OutputBlocks != null
         ? OutputBlocks.GetHashCode() : 0);
        }
    }
}

// Класс область"тела"
public class BodyRegion : IntermediateRegion
{
    public List<Region> Regions { get; set; }

    public BodyRegion(BasicBlock header, List<int> outputBlocks, List<Region> regions) :
         base(header, outputBlocks)
    {
        Regions = regions;
    }

    protected bool Equals(BodyRegion other)
    {
        return Regions.SequenceEqual(other.Regions);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
```

```
60          if (ReferenceEquals(this, obj)) return true;
61          if (obj.GetType() != this.GetType()) return false;
62          return Equals((BodyRegion) obj);
63      }
64
65      public override int GetHashCode()
66      {
67          return (Regions != null ? Regions.GetHashCode() : 0);
68      }
69  }
70
71  // Класс область" цикла"
72  public class LoopRegion : IntermediateRegion
73  {
74      public BodyRegion Body { get; set; }
75
76      public LoopRegion(BasicBlock header, List<int> outputBlocks, BodyRegion body) :
            base(header, outputBlocks)
77      {
78          Body = body;
79      }
80
81      public LoopRegion(BodyRegion body) : base(body.Header, body.OutputBlocks)
82      {
83          Body = body;
84      }
85
86      protected bool Equals(LoopRegion other)
87      {
88          return base.Equals(other) && Equals(Body, other.Body);
89      }
90
91      public override bool Equals(object obj)
92      {
93          if (ReferenceEquals(null, obj)) return false;
94          if (ReferenceEquals(this, obj)) return true;
95          if (obj.GetType() != this.GetType()) return false;
96          return Equals((LoopRegion) obj);
97      }
98
99      public override int GetHashCode()
100     {
101         unchecked
102         {
103             return (base.GetHashCode() * 397) ^ (Body != null ? Body.GetHashCode() : 0);
104         }
```

```
    }
}

// Класс область"лист". Каждый базовый блок рассматривается как областьлист—
public class LeafRegion : Region
{
    public BasicBlock Block { get; set; }

    public override List<int> OutputBlocks
    {
        get
        {
            return Block.OutputBlocks.Count > 0 ? new List<int> { Block.BlockId } : new
        List<int>() ;
        }
    }

    public LeafRegion(BasicBlock block)
    {
        Block = block;
    }

    protected bool Equals(LeafRegion other)
    {
        return Equals(Block, other.Block);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;
        if (obj.GetType() != this.GetType()) return false;
        return Equals((LeafRegion) obj);
    }

    public override int GetHashCode()
    {
        return (Block != null ? Block.GetHashCode() : 0);
    }
}
```

Пример использования

```
foreach (Region r in regions)
{
    LeafRegion leaf = r as LeafRegion;
```

```
    if(leaf != null)
    {
        result[r, RegionDirection.In, r] = Identity;
        result[r, RegionDirection.Out, r] = input => param.TransferFunction(input, leaf.Block);
    }
    BodyRegion body = r as BodyRegion;
    if(body != null)
        foreach(Region s in body.Regions)
        {
            result[r, RegionDirection.In, s] = input => GatherFunctionsResults(input, result, r,
        body.Header.InputBlocks, graph, param);
            CalculateForOutputBlocks(result, r, s, s.OutputBlocks, graph);
        }
    LoopRegion loop = r as LoopRegion;
    if(loop != null)
    {
        result[r, RegionDirection.In, loop.Body] = input =>
        SetFactory.GetSet<V>(input.Union(GatherFunctionsResults(input, result, loop.Body,
        loop.Header.InputBlocks, graph, param)));
        CalculateForOutputBlocks(result, r, loop.Body, loop.OutputBlocks, graph);
    }
}
```

Восходящая часть алгоритма анализа на основе областей

Выполнено командой:

Ampersand (Золотарёв Федор, Маросеев Олег)

От каких проектов зависит:

1.

Зависимые проекты:

Постановка задачи:

В данной задаче требуется разработать класс, который реализует функцию проверки

Теория

Используемые структуры данных

- Dictionary<Edge<BasicBlock>, EdgeType> - словарь классифицированных ребер

Реализация

Пример использования

Формирование восходящей последовательности областей

Выполнено командой

PiedPiper (Бергер Анна, Колесников Сергей)

От каких проектов зависит

1. Граф потока управления
2. Задача определения приводимости графа
3. Поиск естественных циклов
4. Разработка класса “область”

Зависимые проекты

1. Восходящая часть алгоритма анализа на основе областей

Теория

В процессе анализа на основе областей программа рассматривается как иерархия областей(регионов), которые грубо говоря, можно считать частями графа потока, имеющими единственную точку входа. Формально область графа представляет собой набор узолв *N* и ребер *E* таких, что

- существует заголовок *h
  □
  N*, доминирующий над всеми узлами в* N*
- Если некоторый узел *m* может достичь узла *n
  □
  N*, минуя* h*, то* m* также входит в *N*

- *E* - множество всех ребер потока управления между узлами *n1* и *n2* из *N*, за исключением, возможно, некоторых ребер, входящих в *h*

В процессе построения иерархии областей идентифицируем естественные циклы. Процесс начинается с того, что каждый базовый блок рассматривается как область (LeafRegion). Затем естественные циклы упорядочиваются изнутри наружу, т.е. начиная с наиболее внутренних цикла.

Для каждого цикла выделяются две области

- R – область тела (BodyRegion). Тело цикла *L* (все узлы и ребра, за исключением обратных ребер к заголовку) замещаем узлом, представляющим область *R*. Ребра, ведущие к заголовку *L*, входят теперь в узел R. Ребро из любого выхода из *L* замещается ребром от *R* в то же самое место назначения. Однако если ребро является обратным, то оно становится петлей у *R*.
- LR – область цикла (LoopRegion). Единственное отличие области тела от области цикла заключается в том, что последняя включает обратные ребра к заголовку цикла L.

Входные данные

Граф потока управления Graph g

Выходные данные

Список областей в восходящем порядке (List < Region > regionList)

Используемые структуры данных

- List<Region> regionList – список областей в восходящем порядке
- Dictionary<BasicBlock, Region> basicBlockLastRegion – словарь, хранящий соответствие базовый блок - самый внешний регион, в который он вошел
- Dictionary<Edge<BasicBlock>, **bool**> regionMade– словарь, хранящий соответствие обратная дуга - верно ли, что цикл по ней уже сформирован

Реализация алгоритма

Проверка циклов на вложенность осуществляется с помощью вспомогательного метода

```
1 private bool checkLoopInclusion(KeyValuePair<Edge<BasicBlock>, ISet<int>> curLoop,
2             KeyValuePair<Edge<BasicBlock>, ISet<int>> loopOther, bool regionMade)
3 {
4  if (loopOther.Value.IsSubsetOf(curLoop.Value) && !regionMade)
5   return true;
```

```
 else
  return false;
}
```

Основная работа по построению восходящей последовательности областей проходит в методе **public** List<Region> CreateSequence(Graph g)

```
public List<Region> CreateSequence(Graph g)
{
```

Проверка приводимости графа

```
  if (!CheckRetreatingIsReverse.CheckRetreatingIsReverse.CheckReverseEdges(g)){
     Console.WriteLine("there are some retreating edges which aren't reverse");
     Environment.Exit(0);
  }
```

Добавление базовых блоков как LeafRegion в список областей

```
  var basicBlockLastRegion = new Dictionary<BasicBlock, Region>();

  foreach (var v in g.GetVertices())
  {
     var newReg = new LeafRegion(v);
     regionList.Add(newReg);
     basicBlockLastRegion[v] = newReg;
  }

  var loops = SearchNaturalLoops.FindAllNaturalLoops(g);

  var regionMade = new Dictionary<Edge<BasicBlock>, bool>();

  foreach (var loop in loops)
  {
     regionMade[loop.Key] = false;
  }
```

Пока остались незанесенные в список областей циклы, метод продолжает работу

```
  while (regionMade.ContainsValue(false))
  {
     foreach (var loop in loops)
     {
```

Проверка цикла на существование вложенных в него и еще не обработанных циклов

```
        bool anyInsideLoops = false;
        foreach (var loopOther in loops)
```

```
        {
            anyInsideLoops = anyInsideLoops || checkLoopInclusion(loop, loopOther,
    regionMade[loopOther.Key]);
        }
        if (!anyInsideLoops) continue;
```

Если все вложенные циклы уже обработаны или их нет, приступаем к формированию новой области Для формирования BodyRegion требуется - BasicBlock header - им является Target обратной дуги, формирующей цикл - List curRegions - формируется с помощью словаря basicBlockLastRegion - List outputBlocks (блоки, из которых есть дуги в другие регионы - формируется с помощью массива OutputBlocks каждого блока

```
        regionMade[loop.Key] = true;

        var header = loop.Key.Target;

        var curRegions = new List<Region>();
        var outputBlocks = new List<int>();
        foreach (var blockId in loop.Value)
        {
            var block = g.getBlockById(blockId);
            if (!curRegions.Contains(basicBlockLastRegion[block]))
                curRegions.Add(basicBlockLastRegion[block]);

            foreach (var outputBlock in block.OutputBlocks)
            {
                if (!loop.Value.Contains(outputBlock))
                {
                    outputBlocks.Add(outputBlock);
                    break;
                }
            }
        }

        var bodyReg = new BodyRegion(header, outputBlocks, curRegions);
        regionList.Add(bodyReg);

        var loopReg = new LoopRegion(bodyReg);
        regionList.Add(loopReg);

        foreach (var blockId in loop.Value)
        {
            var block = g.getBlockById(blockId);
            basicBlockLastRegion[block] = loopReg;
        }
    }
```

```
  }
```

Если программа не является циклом, то формируется последний регион, включающий в себя всю программу

```
  foreach (var block in basicBlockLastRegion)
  {
     if (block.Value.GetType() == typeof(LeafRegion))
     {
        var header = g.getRoot();
        var outputBlocks = new List<int>();
        var curRegions = new List<Region>();
        foreach (var curblock in basicBlockLastRegion)
        {
           if (!curRegions.Contains(curblock.Value))
              curRegions.Add(curblock.Value);
        }
        var newReg = new BodyRegion(header, outputBlocks, curRegions);
        regionList.Add(newReg);
        break;
     }
  }
```

Тест
Программа
```
i = 1;
j = 4;
a = 2;
while i < 20
{
i = i + 1;
j = j + 1;
if i > a
a = a + 5;
while j < 5
{
a = 4;
}
i = i + 1;
}
```

Граф потока управления, соответствующий программе

```
0:
<——
```

```
——> 1
1:
<—— 0 7
——> 2 8
2:
<—— 1
——> 3 4
3:
<—— 2
——> 4
4:
<—— 2 3
——> 5
5:
<—— 4 6
——> 6 7
6:
<—— 5
——> 5
7:
<—— 5
——> 1
8:
<—— 1
——>
```

Список областей в восходящей последовательности

```
R0 Leaf — 0
R1 Leaf — 1
R2 Leaf — 2
R3 Leaf — 3
R4 Leaf — 4
R5 Leaf — 5
R6 Leaf — 6
R7 Leaf — 7
R8 Leaf — 8
R9 Body — R5, R6
R10 Loop — R9
R11 Body — R1 R2 R3 R4 R10 R7
R12 Loop — R11
R13 Body — R0 R11 R8
```

Нисходящая часть алгоритма анализа на основе областей.

Выполнено командой:

EndFrame (Аммаев Саид, Пирумян Маргарита)

Постановка задачи:

Необходимо реализовать нисходящую часть алгоритма анализа на основе областей в классе RegionsAlgorithm.

От каких проектов зависит:

1. Region
2. Graph
3. IterativeAlgorithm
4. IterativeAlgorithmParameters

Зависимые проекты:

1. IterativeAlgorithm

Теория

```
Нисходящая
 часть алгоритма:
IN[Rn] = INВхода[]для
 каждого региона Ri в нисходящем порядке
 {
   IN[Ri] = fR(i—1), IN[Ri](IN[R(i—1)])
   OUT[Ri] = transferfunction(IN[Ri])
 }
```

Входные данные:

- List regions - список регионов
- TransferFunctionStorage> functions - передаточные функции
- SetIterativeAlgorithmParameters param - параметры итерационного алгоритма
- Graph graph - граф программы

Выходные данные:

- IterativeAlgorithmOutput> - множества входов и выходов для каждого блока

Реализация алгоритма:

“‘ C# /* реализация нисходящей части алгоритма анализа на основе областей. в классе RegionsAlgorithm */

static IterativeAlgorithmOutput> ApplyDescendingPart(List regions, TransferFunctionStorage> functions, SetIterativeAlgorithmParameters param, Graph graph) { Dictionary> regionsInputs = new Dictionary>(); IterativeAlgorithmOutput> result = new IterativeAlgorithmOutput>();

```
    regionsInputs[regions.Count — 1] = param.FirstValue;

    Dictionary<Region, Region> parents = new Dictionary<Region, Region>();

    for (int i = regions.Count — 1; i >= 0; ——i)
    {
        BodyRegion body = regions[i] as BodyRegion;
        if(body != null)
            foreach (Region r in body.Regions)
                parents[r] = body;

        LoopRegion loop = regions[i] as LoopRegion;
        if (loop != null)
            parents[loop.Body] = loop;
        if (parents.ContainsKey(regions[i]))
        {
            Region parent = parents[regions[i]];
            regionsInputs[i] = functions[parent, RegionDirection.In,
 regions[i]](regionsInputs[regions.IndexOf(parent)]);
        }
    }

    int numOfBlocks = graph.Count();

    for(int i = 0; i < numOfBlocks; ++i)
    {
        var curBlock = regions[i].Header;
        int curBlockId =  curBlock.BlockId;

        result.In[curBlockId] = regionsInputs[i];
        result.Out[curBlockId] = param.TransferFunction(regionsInputs[i], curBlock);
    }

    return result;
}
```

```
35    ...
```

```

**Пример использования**

**Тест**

Программа

```
i = 1;
  j = 4;
  a = 2;
  while i < 20
  {
     i = i + 1;
     j = j + 1;
     if i > a
        a = a + 5;
     i = i + 1;
  }
```

## Название задачи: хранение передаточных функций

**Выполнено командой: YACT**

**Постановка задачи:**

Необходимо релизовать хранилище для передаточных функций, позволяющее сохранять и получать передаточные функции по ключу Region1, Direction, Region2.

**От каких проектов зависит:**

- Базовые блоки
- Регионы

**Зависимые проекты:**

- Алгоритм оптимизации с помощью областей
```

Теория

Для сохранения функций и их получения необходимо создать ключ, по которому будут идентифицироваться передаточные функции. Ключ состоит из двух регионов и направления. Для реализации сравнения ключей используются методы Equals(obj) и GetHashCode(). Подобные методы должны быть реализованы в классах регионов и в самом ключе.

Входные данные:

- Передаточные функции

Выходные данные:

- Передаточные функции

Используемые структуры данных

- Dictionary<TKey, TValue> - для хранения передаточных функций

Реализация алгоритма

Для классов BodyRegion, IntermediateRegion, LeafRegion, LoopRegion были перегружены методы Equals и GetHashCode. При реализации учитывались следующие факторы:

- Регионы, состоящие из одного блока сравниваются по блоку
- Регионы, включающие в себя список блоков, сравниваются с помощью метода SequenceEqual
- Для классов-наследников IntermediateRegion необходимо вызывать также **base**.Equals
- GetHashCode реализовывались с учетом полей, учитывающихся при сравнении на равенство с попыткой переполнить **int**, чтобы получить наиболее уникальный хеш.

Пример использования

```
// setting functions
foreach (Region r in regions)
{
   LeafRegion leaf = r as LeafRegion;
   if(leaf != null)
   {
      result[r, RegionDirection.In, r] = Identity;
      result[r, RegionDirection.Out, r] = input => param.TransferFunction(input, leaf.Block);
```

```
    }
...
}

// getting functions
foreach (var r in regions.Reverse<Region>())
{
    int curIndex = RegionIndexes[r];
    if (curIndex != lastIndex)
    {
        regionsInputs[curIndex] = functions[regions[prevIndex], RegionDirection.In,
         regions[curIndex]](regionsInputs[prevIndex]);
    }
}
```

Тест

```
string text = @"
    i = 1;
    j = 4;
    a = 2;
    while i < 20
    {
        i = i + 1;
        j = j + 1;
        if i > a
            a = a + 5;
        i = i + 1;
    }";

SyntaxNode root = ParserWrap.Parse(text);
var graph = new
        Graph(BasicBlocksGenerator.CreateBasicBlocks(ThreeAddressCodeGenerator.CreateAndVisit(root).Program));
var regions = new RegionSequence().CreateSequence(graph);
var storage = new AbstractTransferFunctionStorage<int>();
for (var i = 0; i < regions.Count — 1; i++)
{
    storage[regions[i], RegionDirection.Out, regions[i + 1]] = 2 * i;
    storage[regions[i], RegionDirection.In, regions[i + 1]] = 2 * i + 1;
}

for (var i = 0; i < regions.Count — 1; i++)
{
    Assert.IsTrue(storage[regions[i], RegionDirection.Out, regions[i + 1]] == 2 * i);
    Assert.IsTrue(storage[regions[i], RegionDirection.In, regions[i + 1]] == 2 * i + 1);
}
```

Название задачи: Meet Over All Paths

Выполнено командой: YACT

Постановка задачи:

Необходимо реализовать алгоритм Meet Over All Paths на ациклическом графе потока данных для реализации оптимизаций. А также сравненить результаты работы этого алгоритма с результатами итерационного алгоритмом для двух конфигураций.

Решить задачу потока данных, методом обхода всех путей от входной точки до каждого базового блока. На примерах показать, что для дистрибутивных схем передачи потока данных результат работы Meet Over All Paths совпадает с результатом работы итерационного алгоритма, а для не дистрибутивных(распространение констант) не совпадает.

От каких проектов зависит:

- Control Flow Graph
- Basic Block Model
- Iterative Algorithm Parameters

Зависимые проекты:

нет

Теория

Рассмотрим входную точку базового блока B. Идеальное решение начинается с поиска всех возможных путей выполнения, ведущих от входной точки программы к к началу B. Путь “возможен”, только если некоторое вычисление программы следует в точности по этому пути. Идеальное решение должно вычислить значение потока данных в конце каждого возможного пути и применить к найденным значениям оператор сбора для получения их наибольшей нижней границы. Тогда никакое выполнение программы не в состоянии привести к меньшему значению для данной её точки.

Определение: Для каждого пути *p*, состоящего из базовых блоков *Bi* рассмотрим *fp(VBход)*, где *fp* это композиция передаточных функций базовых блоков на пути *p*. Идеальным решением назовём

IDEAL[B] = □ fp(VBход), где □ это оператор сбора по возможным путям от входа к B

Интуитивно значение, более близкое к идеальному, является более точным. Чтобы увидеть, почему любые решения должны не превосходить идеальное, заметим, что для любого блока любое решение, большее, чем IDEAL, может быть получено путем

игнорирования некоторого пути выполнения, по которому может пойти программа; мы не можем гарантировать, что вдоль этого пути не произойдёт что-то, что сделает недействительным все улучшения программы, которые мог ли бы быть сделанны на основе большего решения. Но поиск всех возможных путей задача неразрешимая. Рассмотрим пример №1. Из этого примера видно, что не зная ничего о sqr нельзя сказать какие пути возможны, а какие нет. Поэтому будем рассматривать Meet Over All Paths(MOP).

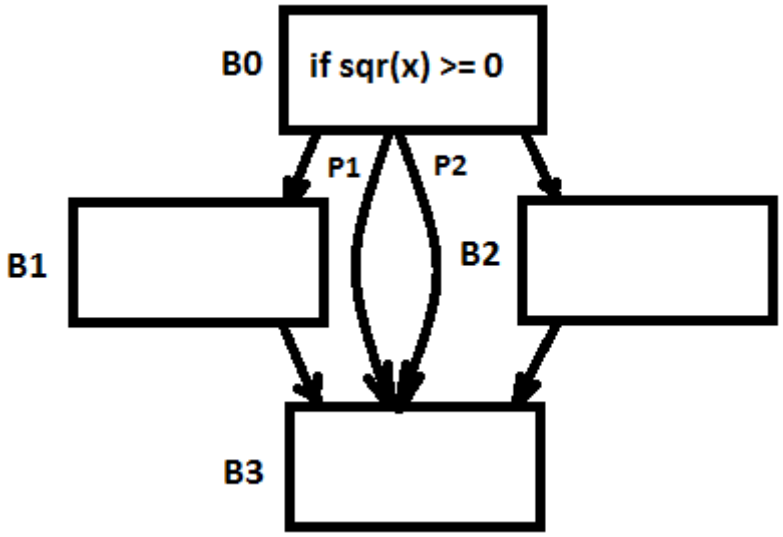


Figure 8:

Meet Over all Paths приближённое решение. Рассматриваемые в решении MOP пути представляют собой надмножество всех путей, которые могут быть выполнены. Таким образом, решение MOP собирает вместе не только значения потоков даных всех выполнимых путей, но и дополнительные значения, связанные с путями, которые не могут быть выполнены. MOP[B] = □ fp(VBход), где □ это оператор сбора по всем путям от входа к B

Сравним Meet Over All Paths с итерационным алгоритмом(Maximum Fix Point) на примере следующего графа

MOP[B] = fB3(fB1(VBход)) □ fB3(fB2(VBход))

MFP[B] = fB3(fB1(VBход) □ fB2(VBход))

Утверждение: Если схема передачи потока данных дистрибутивна, то **MFP[B] = MOP[B]**

Замечание 1: Схема передачи данных дистрибутивна <=> f - дистрибутивна

Замечание 2: Всегда имеет место **MFP[B]≤MOP[B]≤IDEAL[B]**

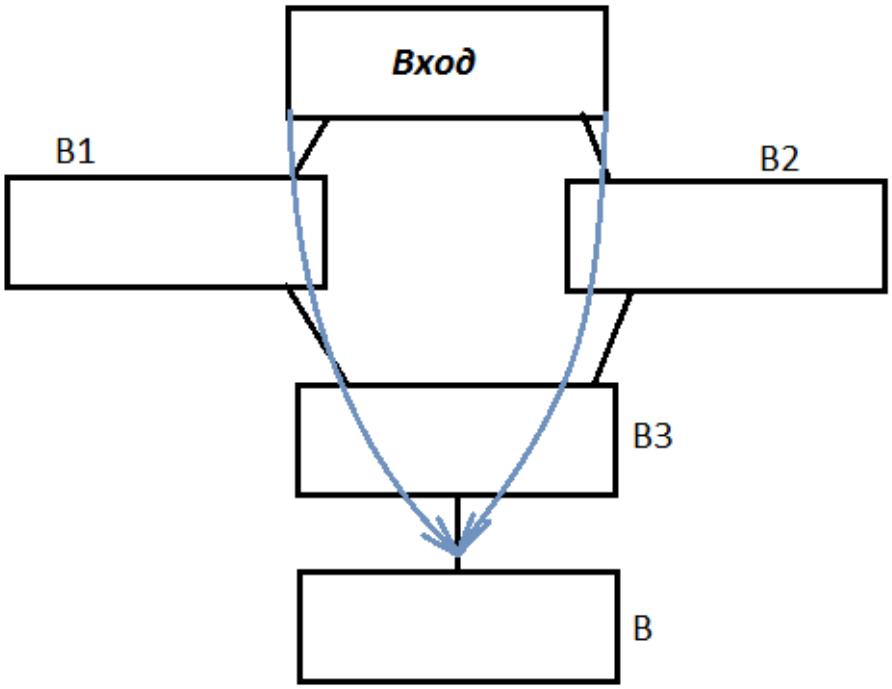


Figure 9:

Входные данные:

- Ациклический граф потока данных программы
- Параметры конкретной оптимизации

Выходные данные:

- Решение задачи потока данных для конкретной оптимизации

Используемые структуры данных

- Словарь из идентификатора блока в результат работы передаточной функции
- Граф потока данных
- Абстрактный класс параметров алгоритма

Реализация алгоритма

```
var MOP = new Dictionary<int, V>(); // Результат

foreach (BasicBlock blockTo in graph) // Для каждого блока получаемзначение потокаданных
{
  MOP[blockTo.BlockId] = param.StartingValue;
  foreach (var path in GraphAlgorithms.FindAllPaths(graph, blockTo.BlockId)) // Для каждого пути от начала до конкретного блока получаем композицию передаточных функций и собираем результаты опертором сбора
  {
    var value = path.Aggregate(param.FirstValue, param.TransferFunction);
    MOP[blockTo.BlockId] = param.GatherOperation(new List<V> { MOP[blockTo.BlockId], value});
  }
}
```

Пример использования

Программа на языке:

```
if 1
{
   x = 2;
   y = 3;
}
else
{
```

```
    x = 3;
    y = 2;
}
z = x + y;
```

Вызов алгоритма:

```
// Парсинг программы и получение CFG
...
var constantPropagationMOP = MeetOverPaths.Apply(graph, new
        ConstantsPropagationParameters());
var it = constantPropagationMOP.Out.Select(
  pair => $"{pair.Key}: {string.Join(", ", pair.Value.Select(ex => ex.ToString()))}");
```

Вывод:

```
50:
51: [x, 2], [y, 3]
52: [x, 3], [y, 2]
53: [x, NAC], [y, NAC], [t0, 5], [z, 5]
```

Тест

```
// Программа взята из тестов оптимизации AvailableExpression
string programText = @"
a = 4;
b = 4;
c = a + b;
if 1
  a = 3;
else
  b = 2;
print(c);
";

// Получение CFG
SyntaxNode root = ParserWrap.Parse(programText);
var threeAddressCode = ThreeAddressCodeGenerator.CreateAndVisit(root).Program;
var basicBlocks = BasicBlocksGenerator.CreateBasicBlocks(threeAddressCode);
Graph g = new Graph(basicBlocks);

// Получаем результаты разными алгоритмами
var availableExprsIterative = IterativeAlgorithm.Apply(g, new
        AvailableExpressionsCalculator(g));
var availableExprsMOP = MeetOverPaths.Apply(g, new AvailableExpressionsCalculator(g));

```

```
// Вывод полученных результатов
var it = availableExprsIterative.Out.Select(
  pair => $"{pair.Key}: {string.Join(", ", pair.Value.Select(ex => ex.ToString()))}");
foreach (var outInfo in it)
{
  Trace.WriteLine(outInfo);
}
var mop = availableExprsMOP.Select(
  pair => $"{pair.Key}: {string.Join(", ", pair.Value.Select(ex => ex.ToString()))}");
Trace.WriteLine("====");
foreach (var outInfo in mop)
{
  Trace.WriteLine(outInfo);
}

// Сравниваем на совпадение полученные структуры данных
Assert.IsTrue(availableExprsIterative.Out.OrderBy(kvp => kvp.Key).
  Zip(availableExprsMOP.OrderBy(kvp => kvp.Key), (v1, v2) => v1.Key == v2.Key &&
      v1.Value.SetEquals(v2.Value)).All(x => x));
```