

Документация по проекту "Разработка оптимизирующего компилятора"

Введение

Целью данного проекта является разработка оптимизирующего компилятора для описанной грамматики. В начале поставленной работы, необходимо было создать парсер или текстовый анализатор.

Написать текстовый анализатор можно множеством различных способов, простейшим из которых является применение lex-инструмента, который конвертирует входную информацию в последовательность лексем. Грамматические правила используются для распознавания последовательности лексем и выполнения подходящих действий. В основном грамматические правила используются в комбинации с lex; оба эти инструмента - yacc и lex - и составляют анализатор.

Проект выполнен на языке C#.

Синтаксис языка

№	Инструкция
1	a = b;
2	{ a = 1; c = d;}
3	read(a)
4	write(a,b,c); writeln(a,b,c);
5	if (a < b) { <операторы>} else { <операторы>}
6	goto 123; 123: a =123;
7	for i=1..5 {write(i, ' ');}
8	while a < b { a = a + 1;}
9	int/bool/float a,b,c;
10	if a != b I a == b { <операторы> }
11	if (a != b I a == b) & b == 7 { <операторы> }

Пример продукции списка выражений

```
exprlist    : expr
             {
               $$ = new ListExprNode($1);
             }
             | exprlist COMMA expr
             {
               $1.Add($3);
               $$ = $1;
             }
```

```
}  
;
```

Оптимизации по синтаксическому дереву

На этом этапе оптимизации используется такая форма представления программы, которая более приспособлена для анализа, дальнейших преобразований и генерации кода. Мы будем переводить текст программы в так называемое синтаксическое дерево. Если синтаксическое дерево построено, то программа синтаксически правильная, и ее можно подвергать дальнейшей обработке. В синтаксическое дерево включаются узлы, соответствующие всем синтаксическим конструкциям языка. Атрибутами этих узлов являются их существенные характеристики. Например, для узла оператора присваивания `AssignNode` такими атрибутами являются `IdNode` - идентификатор в левой части оператора присваивания и `ExprNode` - выражение в правой части оператора присваивания. Синтаксическое дерево программы (или AST - Abstract Syntax Tree) отличается от дерева разбора тем, что в него не добавляются несущественные атрибуты - например, ключевые слова.

В уасс-файл, помимо грамматики, было добавлены семантические правила, которые записаны после правил грамматики в фигурных скобках и являются командами, конструирующими узлы синтаксического дерева. Синтаксическое дерево строится снизу вверх: вначале строятся листовые узлы, не имеющие потомков (например, `IdNode` или `IntNumNode`), затем по ним строятся другие узлы (например, `AssignNode` - по `IdNode` и `ExprNode`). Стратегия построения синтаксического дерева снизу вверх соответствует стратегии разбора снизу-вверх, принятой в парсере `gppg` (точнее, во всех парсерах, поддерживающих LR-грамматики). Корень синтаксического дерева записывается в поле `root` класса `Parser`.

Список задач, которые необходимо реализовать

№	Выражение для оптимизации	Результат оптимизации
1	<code>1 * expr, expr * 1, expr / 1</code>	<code>expr</code>
2	<code>2 * 3</code>	<code>6</code>
3	<code>0 * expr, expr * 0</code>	<code>0</code>
4	<code>0 + expr</code>	<code>expr</code>
5	<code>a - a</code>	<code>0</code>
6	<code>2 < 3</code>	<code>true</code>
7	<code>2 == 4</code>	<code>false</code>
8	<code>a == a, a >= a</code>	<code>true</code>
9	<code>a > a, a != a</code>	<code>false</code>
10	<code>x = x</code>	<code>null</code>
11	<code>if (true) st1; else st2;</code>	<code>st1</code>
12	<code>if (false) st1; else st2;</code>	<code>st2</code>
13	<code>if (expr) null; else null;</code>	<code>null</code>
14	<code>while (false) st;</code>	<code>null</code>

1. Задача на оптимизацию: $1 * \text{expr}$, $\text{expr} * 1$, $\text{expr} / 1$, $\text{expr} + 0 \Rightarrow \text{expr}$

Примечание: Здесь также описана задача №4

1. Команда, реализующая задачу Манукян Г. А.

2. Зависимые и предшествующие задачи

Предшествующая:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

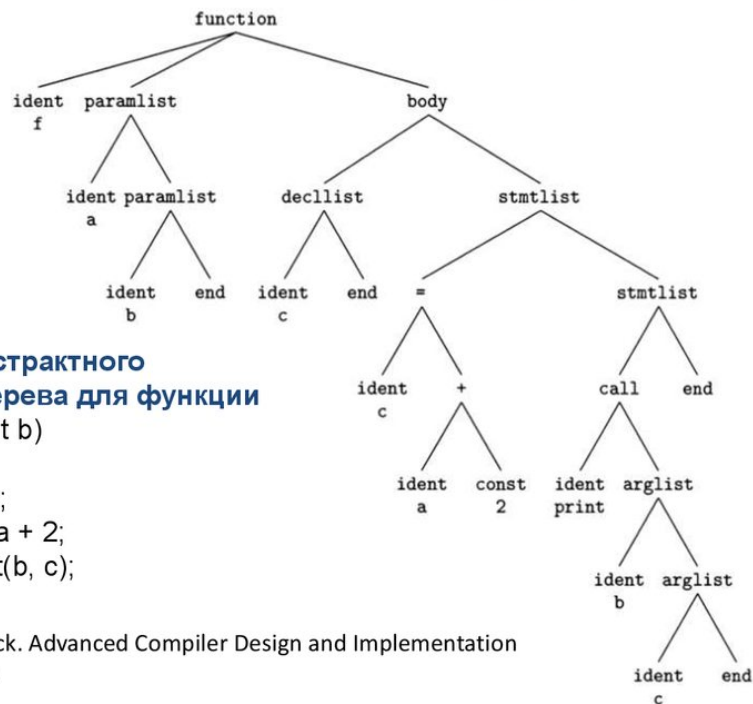
Основная цель данной оптимизации – при помощи Визиторов заменить в узлах синтаксического дерева программы все алгебраически тождественные действия, то есть не меняющие результата после применения операции, на значение переменной (неизменной) – умножение переменной слева (справа) на единицу, деление на единицу, прибавление нуля. Любая комбинация данных действий также устраняется до значения переменных, в условных операторах и так далее:

```
До:
c = c/(c/1);
if 2*1
...
if 1*2 > 0
...
q = 0 + c;
q = c + 0;
После:
c = c/c;
if 2
...
if 2 > 0
...
q = c;
q = c
```

4. Теория

Абстрактное синтаксическое дерево (АСД, англ. AST – Abstract Syntax Tree) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы.

Абстрактные синтаксические деревья



Пример абстрактного синтаксического дерева для функции

```

int f(int a, int b)
{
    int c;
    c = a + 2;
    print(b, c);
}
  
```

Пример из Steven Muchnick. Advanced Compiler Design and Implementation / Morgan Kaufmann. 1997.

Семантическое дерево – «украшенное» синтаксическое, то есть содержащее дополнительную информацию (об операторе, инструкции и так далее) Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

5. Реализация

В соответствии с целью оптимизации, узел дерева, содержащий тождественную операцию, заменяется на новый узел, с другим операндом, который также может содержать тождественную операцию. Для этого реализован паттерн «Визитор», В отдельных классах AlgebraicIdentityProdDiv1Visitor и AlgebraicIdentitySum0Visitor описаны методы Visit, заменяющие (рекурсивно) выражения в узлах AST: **Пример $1 * ex$, $ex * 1$, $ex / 1$.**

```

<...> case BinOpType.Prod:
if (BinOp.Left is IntNumNode && (BinOp.Left as IntNumNode).Num == 1)
{
    BinOp.Right.Visit(this);
    ReplaceExpr(BinOp, BinOp.Right);
}
else if (BinOp.Right is IntNumNode && (BinOp.Right as IntNumNode).Num == 1)
{
    BinOp.Left.Visit(this);
    ReplaceExpr(BinOp, BinOp.Left);
}
else base.Visit(BinOp); break; // После замены вызывается base.визитор - конвейер
case BinOpType.Div:
  
```

```

if (BinOp.Right is IntNumNode && (BinOp.Right as IntNumNode).Num == 1)
{
    BinOp.Left.Visit(this);
    ReplaceExpr(BinOp, BinOp.Left);
}
break;
default:
base.Visit(BinOp); // После замены вызывается base.визитор от BinOp
break;

```

6. Интеграция в общий проект

Применяются вышеописанные методы в конвейере оптимизации (необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, они циклически вызываются каждый раз, когда обновляется AST). После интеграции, понадобились отдельные тесты (NUnit.Framework - TestSuite), где данный оптимизатор вызывается для особых случаев

7. Тесты

Тестирование может проходить в двух вариантах: По заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью обоих визиторов; далее – 1. Заданное дерево (для более простых случаев, когда в программе одна или несколько строк кода) оптимизирующими визиторами ProdDiv и SumDiv - преобразуется в строку и сравнивается с уже приведенным вариантом дерева. 2. По оптимизированному дереву генерируется ТАС-код, который затем сравнивается с предполагаемым ответом. Пример:

```

[TestCase("a = exp * 1;", "a = exp;")]
[TestCase("a = 1 * exp;", "a = exp;")]
[TestCase("a = b / 1;", "a = b;")]
[TestCase("a = c + 0;", "a = c;")]
[TestCase("a = 0 + d;", "a = d;")]
[TestCase("a = c/c/1*1*1/1;", "a = c/c;")]
[TestCase("a = 0 + c + 0*1;", "a = c;")]
public void OneLineTests(string line, string expected)
{
    var parser = GenerateTree(@"{" + @"\n{line}\n" + @"}");
    var expectedTree = GenerateTree(@"{" + @"\n{expected}\n" + @"}");
    var ProdDiv = new AlgebraicIdentityProdDiv1Visitor();
    var Sum = new AlgebraicIdentitySum0Visitor();
    ProdDiv.Visit(parser.root);
    Sum.Visit(parser.root);
    var actual = parser.ToString();
    var expect = expectedTree.ToString();
    Assert.AreEqual(actual, expect);
}

[Test]
public void AllDividedTestCasesTest()
{
    var Text = @"{
a = 1;
b = 1;

```

```

c = 0;
c = c/c/1;
if 1*a + 1*b
{
c = a * b * 1;
}
q = a/1 + 5;
q = c + 0;
}");
    Scanner scanner = new Scanner();
    scanner.SetSource(Text, 0);
    Parser parser = new Parser(scanner);
    parser.Parse();

    var parentFiller = new FillParentsVisitor();
    parser.root.Visit(parentFiller);

    var ProdDiv = new AlgebraicIdentityProdDiv1Visitor();
    var Sum = new AlgebraicIdentitySum0Visitor();

    ProdDiv.Visit(parser.root);
    Sum.Visit(parser.root);
    var prettyPrinter = new PrettyPrinterVisitor();

    parser.root.Visit(prettyPrinter);

    var TAC = GenerateTAC(prettyPrinter.FormattedProgram);

    var expected = new List<string>()
    {
        "a = 1",
        "b = 1",
        "c = 0",
        "#t0 = c / c",
        "c = #t0",
        "#t1 = a + b",
        "if #t1 goto #L0",
        "goto #L1",
        "#L0",
        "#t2 = a * b",
        "c = #t2",
        "#L1",
        "#t3 = a + 5",
        "q = #t3",
        "q = c"
    };

    var actual = TAC.Instructions.Select(instruction =>
instruction.ToString().Trim());
    CollectionAssert.AreEqual(expected, actual);
}

```

2. Свертка констант в синтаксическом дереве.

1. Команда, реализующая задачу Погорелов А. А., Домбровская А. В.

2. Зависимые и предшествующие задачи

Предшествующая:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Данная задача основывается на использовании визиторов и решается путем замены числового выражения на константу. Оптимизация работает для всех операций (сложение, умножение, деление, разность).

```
До:
x = 2 * 3 + 4;
После:
x = 10;
```

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

5. Реализация

При реализации использовался паттерн “Визитор”. Оптимизация была вынесена в отдельный класс-визитор и при обходе синтаксического дерева данным визитором, числовые выражения заменяются на соответствующую константу, вычисленную визитором. Часть кода, отвечающий за такую замену приведена ниже:

```
if (n is BinExprNode node)
{
    if (node.Left is IntNumNode leftInt && node.Right is IntNumNode rightInt)
    {
        if (node.OpType == BinOpType.Div)
        {
            ReplaceExpr(node, new IntNumNode(leftInt.Num / rightInt.Num));
        }
        if (node.OpType == BinOpType.Prod)
        {
            ReplaceExpr(node, new IntNumNode(leftInt.Num * rightInt.Num));
        }
        if (node.OpType == BinOpType.Minus)
        {
            ReplaceExpr(node, new IntNumNode(leftInt.Num - rightInt.Num));
        }
    }
}
```

```

    }
    if (node.OpType == BinOpType.Plus)
    {
        ReplaceExpr(node, new IntNumNode(leftInt.Num + rightInt.Num));
    }
}
}

```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью ConstantFoldingVisitor- оптимизирующий визитор. По оптимизированному дереву генерируется TAC-код, который затем сравнивается с предполагаемым ответом. Пример:

```

[Test]
public void SimpleExample2()
{
    var Text = (
@"
{
x = 7 - 8 * 0 + 3;
}
");
    Scanner scanner = new Scanner();
    scanner.SetSource(Text, 0);
    Parser parser = new Parser(scanner);
    parser.Parse();
    var parentFiller = new FillParentsVisitor();
    parser.root.Visit(parentFiller);

    var optimizer = new ConstantFoldingVisitor();
    parser.root.Visit(optimizer);

    var TACGenerator = new TACGenerationVisitor();
    parser.root.Visit(TACGenerator);
    var TAC = TACGenerator.TAC;

    var expected = new List<string>()
    {
        "x = 10"
    };
    var actual = TAC.Instructions.Select(instruction =>
instruction.ToString().Trim());
    CollectionAssert.AreEqual(expected, actual);
}

```

3. Задача на оптимизацию: $0 * expr, expr * 0 \Rightarrow 0$

1. Команда, реализующая задачу Гарьковенко А. Руднев. Д

2. Зависимые и предшествующие задачи

Предшествующая:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Данная задача основывается на использовании визиторов и решается путем нахождения необходимых условий для преобразований, а именно значение у одного из аргументы 0 и операции умножения. И замене данного выражения на 0;

```
До:
b = 4;
a = b * 0;
После:
b = 4;
a = 0;
```

4. Теория

Абстрактное синтаксическое дерево (АСД) — конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья — с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

5. Реализация

При реализации использовался паттерн “Визитор”. Оптимизация была вынесена в отдельный класс-визитор и при обходе синтаксического дерева данным визитором, выражения типа “ $a = b * 0$ ” и “ $a = 0 * b$ ” константу 0. Часть кода, отвечающий за такую замену приведена ниже:

```
if (binop is BinExprNode bi)
{
    if (bi.Left is ExprNode && bi.Right is IntNumNode
        && ((bi.Right as IntNumNode).Num == 0)
        && bi.OpType == BinOpType.Prod)
    {
        bi.Right.Visit(this);
        ReplaceExpr(bi, new IntNumNode(0));
        IsChanged = true;
    }
    else
    {
```

```
        IsChanged = false;
    }
}
```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью MultiplyOnZero- оптимизирующий визитор. По оптимизированному дереву генерируется TAC-код, который затем сравнивается с предполагаемым ответом. Пример:

```
[Test]
public void Test2()
{
    var text =
@"{
c = 4;
b = 5;
a = b*0 + 0*c;
}";

    /* построение синтаксического дерева */

    var Opt = new MultiplyOnZero();
    parser.root.Visit(Opt);

    var Opt2 = new MultiplyOnZero2();
    parser.root.Visit(Opt2);

    var prettyPrinter = new PrettyPrinterVisitor();
    parser.root.Visit(prettyPrinter);

    var TAC = GenerateTAC(prettyPrinter.FormattedProgram);

    var expected = new List<string>()
    {
        "c = 4",
        "b = 5",
        "#t0 = 0 + 0",
        "a = #t0"
    };
    var actual = TAC.Instructions.Select(instruction =>
instruction.ToString().Trim());
    CollectionAssert.AreEqual(expected, actual);
}
```

5. Задача на оптимизацию: $a - a \Rightarrow 0$

1. Команда, реализующая задачу Гарьковенко А. Руднев. Д.

2. Зависимые и предшествующие задачи

Предшествующие:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Данная задача основывается на использовании визиторов и решается путем нахождения необходимых условий для преобразований, а именно нахождения операции и одинаковых аргумента.

```
До:
b = 4;
a = b - b;
После:
b = 4;
a = 0;
```

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

5. Реализация

При реализации использовался паттерн “Визитор”. Оптимизация была вынесена в отдельный класс-визитор и при обходе синтаксического дерева данным визитором, выражения типа “ $a - b$ ” константу 0. Часть кода, отвечающий за такую замену приведена ниже:

```
if (binop is BinExprNode bi)
{
    if (bi.Left is IdNode && bi.Right is IdNode
        && ((bi.Left as IdNode).Name == (bi.Right as IdNode).Name)
        && bi.OpType == BinOpType.Minus)
    {
        ReplaceExpr(bi, new IntNumNode(0));
        IsChanged = true;
    }
    else {
        IsChanged = false;
    }
}
```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью MinusSelf - оптимизирующий визитор. По оптимизированному дереву генерируется TAC-код, который затем сравнивается с предполагаемым ответом.

```
[Test]
public void Test1()
{
    var Text =
@"{
b = 5;
a = b-b;
}
";

/* построение синтаксического дерева */

var Opt = new MinusSelf();
parser.root.Visit(Opt);

var prettyPrinter = new PrettyPrinterVisitor();
parser.root.Visit(prettyPrinter);
var TAC = GenerateTAC(prettyPrinter.FormattedProgram);
var expected = new List<string>()
{
    "b = 5",
    "a = 0"
};
var actual = TAC.Instructions.Select(instruction =>
instruction.ToString().Trim());
CollectionAssert.AreEqual(expected, actual);
}
```

6. Задача на оптимизацию: $2 < 3 \Rightarrow \text{true}$

1. Команда, реализующая задачу Гуртовой А. И., Остапенко М. В.

2. Зависимые и предшествующие задачи

Предшествующие:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Реализация данной задачи представляет собой замену узла условного оператора сравнения чисел булевой константой с помощью визитора.

4. Теория

Данная задача основывается на использовании визиторов и решается путем нахождения необходимых условий для преобразований, а именно значение у одного из аргументы 0 и операции умножения. И замене данного выражения на 0;

```
До:
b = 4;
a = 2 < 3;
После:
b = 4;
a = true;
```

5. Реализация

В реализации данной задачи использовался отдельный класс-визитор, созданный специально для этой задачи. При обходе дерева, выражения типа "2 < 3" заменяются на узел, содержащий булеву константу. Ниже приведен код, отвечающий за реализацию оптимизации:

```
if (number1 != null && number2 != null)
{
    BoolValNode newExpr = null;
    if (binExpr.OpType == BinOpType.Less)
    {
        newExpr = new BoolValNode(number1 < number2);
        ReplaceExpr(binExpr, newExpr);
    }
    if (binExpr.OpType == BinOpType.Greater)
    {
        newExpr = new BoolValNode(number1 > number2);
        ReplaceExpr(binExpr, newExpr);
    }
    if (newExpr != null)
    {
        isChanged = true;
        base.Visit(newExpr);
    }
}
if (!isChanged)
{
    base.Visit(binExpr);
}
```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их

циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, затем оно оптимизируется с помощью `NumberEqualityVisitor` - оптимизирующий визитор, узел оптимизированного дерева затем сравнивается с предполагаемым ответом. Пример:

```
[TestCase("a = 2 < 3;", "a = true;")]
[TestCase("a = 5 < 7;", "a = true;")]
public void OneLineTests(string line, string expected)
{
    var parser = GenerateTree(@"{" + $"\\n{line}\\n" + @"}");
    var expectedTree = GenerateTree(@"{" + $"\\n{expected}\\n" + @"}");
    var a = new NumberEqualityVisitor();
    a.Visit(parser.root);
    var first = (parser.root.StList[0] as AssignNode).Expr as BoolValNode;
    var second = (expectedTree.root.StList[0] as AssignNode).Expr as BoolValNode;
    Assert.AreEqual(first.Val, second.Val);
}
```

7. Задача на оптимизацию: `2 == 4 => false`

1. Команда, реализующая задачу Чухин А. И., Агафонец Р. Г.

2. Зависимые и предшествующие задачи

Предшествующие:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Данная задача основывается на использовании визиторов и решается путем замены в узле условного оператора оператора равенства двух чисел на булеву константу `false`. Помимо указанного `2 == 4`, были учтены все остальные варианты операторы сравнения двух чисел (`>=`, `<=`, `!=`, `>`, `<`).

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

```
До:
b = 4;
a = 2 == 4;
```

```
После:  
b = 4;  
a = false;
```

5. Реализация

При реализации использовался паттерн “Визитор”. Оптимизация была вынесена в отдельный класс-визитор и при обходе синтаксического дерева данным визитором, выражения, приведенного выше типа, заменялись на новый узел, содержащий булево значение. Фрагмент кода, класса, отвечающего за данную задачу приведен ниже:

```
public class NumberEqualityVisitor : ChangeVisitor  
{  
    public override void Visit(BinExprNode binExpr)  
    {  
        double? number1 = null, number2 = null;  
        bool isChanged = false;  
        if (binExpr.Left is IntNumNode || binExpr.Left is FloatNumNode)  
        {  
            if (binExpr.Left is IntNumNode i)  
                number1 = i.Num;  
            if (binExpr.Left is FloatNumNode f)  
                number1 = f.Num;  
        }  
        if (binExpr.Right is IntNumNode || binExpr.Right is FloatNumNode)  
        {  
            if (binExpr.Right is IntNumNode i)  
                number2 = i.Num;  
            if (binExpr.Right is FloatNumNode f)  
                number2 = f.Num;  
        }  
        if (number1 != null && number2 != null)  
        {  
            BoolValNode newExpr = null;  
            if (binExpr.OpType == BinOpType.Equal)  
            {  
                newExpr = new BoolValNode(number1 == number2);  
                ReplaceExpr(binExpr, newExpr);  
            }  
            if (binExpr.OpType == BinOpType.NotEqual)  
            {  
                newExpr = new BoolValNode(number1 != number2);  
                ReplaceExpr(binExpr, newExpr);  
            }  
            if (binExpr.OpType == BinOpType.Less)  
            {  
                newExpr = new BoolValNode(number1 < number2);  
                ReplaceExpr(binExpr, newExpr);  
            }  
            if (binExpr.OpType == BinOpType.Greater)  
            {  
                newExpr = new BoolValNode(number1 > number2);
```

```

        ReplaceExpr(binExpr, newExpr);
    }
    if (binExpr.OpType == BinOpType.LessOrEqual)
    {
        newExpr = new BoolValNode(number1 <= number2);
        ReplaceExpr(binExpr, newExpr);
    }
    if (binExpr.OpType == BinOpType.GreaterOrEqual)
    {
        newExpr = new BoolValNode(number1 >= number2);
        ReplaceExpr(binExpr, newExpr);
    }
    if (newExpr != null)
    {
        isChanged = true;
        base.Visit(newExpr);
    }
}
if (!isChanged)
{
    base.Visit(binExpr);
}
}
}

```

6. Интеграция в общий проект

Класс наследуются от общего для всех подобных оптимизаций предка `ChangeVisitor`.

7. Тесты

До	После
"a = 5 > 2;"	"a = true;"
"a = 2 < 3;"	"a = true;"
"a = 2 > 2;"	"a = false;"
"a = 5 >= 5;"	"a = true;"
"a = 5 < 5;"	"a = false;"
"a = 5 <= 5;"	"a = true;"
"a = 5 == 2;"	"a = false;"
"a = 5 == 5;"	"a = true;"
"a = 5.123 == 5.123;"	"a = true;"
"a = 5.124 == 5.123;"	"a = false;"

8. Задача на оптимизацию: `a == a`, `a >= a => true`

1. Команда, реализующая задачу Погорелов А. А., Домбровская А. В.

2. Зависимые и предшествующие задачи

Предшествующие:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Данная задача основывается на использовании визиторов и решается путем замены в узле условного оператора оператора равенства на булеву константу `true`. Помимо указанных `a == a` и `a >= a`, был также учтен вариант `a <= a`.

```
До:
if a == a
{
    st1;
}
После:
if true
{
    st1;
}
```

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

5. Реализация

При реализации использовался паттерн "Визитор". Оптимизация была вынесена в отдельный класс-визитор и при обходе синтаксического дерева данным визитором, выражения типа `"a==a"`, `"a>=a"` и `"a<=a"` заменялись на новый узел, инкапсулирующий константу `true`. Часть кода, отвечающий за такую замену приведена ниже:

```
if (node.Left is IdNode leftIdNode
    && node.Right is IdNode rightIdNode
    && leftIdNode.Name.Equals(rightIdNode.Name)
    && (node.OpType == BinOpType.Equal
        || node.OpType == BinOpType.LessOrEqual
        || node.OpType == BinOpType.GreaterOrEqual))
{
    ReplaceExpr(node, new BoolValNode(true));
    IsChanged = true;
}
else
{
```

```
    IsChanged = false;
    base.Visit(node);
}
```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью `TrueConditionOptVisitor` - оптимизирующий визитор. По оптимизированному дереву генерируется TAC-код, который затем сравнивается с предполагаемым ответом. Пример:

```
[Test]
public void SimpleIf()
{
    var Text =
@"
{
    if a == a
    {
        b = a;
    }
}
";

    Scanner scanner = new Scanner();
    scanner.SetSource(Text, 0);
    Parser parser = new Parser(scanner);
    parser.Parse();
    var parentFiller = new FillParentsVisitor();
    parser.root.Visit(parentFiller);

    var trueOpt = new TrueConditionOptVisitor();
    parser.root.Visit(trueOpt);
    var prettyPrinter = new PrettyPrinterVisitor();
    parser.root.Visit(prettyPrinter);
    var TAC = GenerateTAC(prettyPrinter.FormattedProgram);

    var expected = new List<string>()
    {
        "if True goto #L0",
        "goto #L1",
        "#L0",
        "b = a",
        "#L1"
    };

    var actual = TAC.Instructions.Select(instruction =>
        instruction.ToString().Trim());
```

```
CollectionAssert.AreEqual(expected, actual);  
}
```

9. Задача на оптимизацию: $a > a$, $a \neq a \Rightarrow \text{false}$.

1. Команда, реализующая задачу Османян В. А., Маслова О. В.

2. Зависимые и предшествующие задачи

Предшествующие:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Данная задача основывается на использовании визиторов и решается путем замены в узле условного оператора больше на булеву константу false. Помимо указанных $a > a$ и $a \neq a$, был также учтен вариант $a < a$.

```
До:  
if a > a  
{  
    st2;  
}  
После:  
if (false)  
{  
    st2;  
}  
До:  
if a != a  
{  
    st2;  
}  
После:  
if (false)  
{  
    st2;  
}
```

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

5. Реализация

При реализации использовался паттерн "Визитор". Оптимизация была вынесена в отдельный класс-визитор и при обходе синтаксического дерева данным визитором, выражения типа "a > a", "a != a" и "a < a" заменялись на новый узел, инкапсулирующий константу false. Часть кода, отвечающий за такую замену приведена ниже:

```
if (node.Left is IdNode leftIdNode
    && node.Right is IdNode rightIdNode
    && leftIdNode.Name.Equals(rightIdNode.Name)
    && (node.OpType == BinOpType.Greater
        || node.OpType == BinOpType.Less
        || node.OpType == BinOpType.NotEqual))
{
    ReplaceExpr(node, new BoolValNode(false));
    IsChanged = true;
}
else
{
    IsChanged = false;
    base.Visit(node);
}
```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью SelfNotEqualTest - оптимизирующий визитор. По оптимизированному дереву генерируется ТАС-код, который затем сравнивается с предполагаемым ответом. Пример:

```
[Test]
public void SelfNotEqualTest()
{
    var parser = GenerateTree(
@"{
if (a!=a)
{
a = 5;
}
else
{
b = 1;
}
}");
    var expectedTree = GenerateTree(
@"{
if (false)
```

```

{
a = 5;
}
else
{
b = 1;
}
}");
    var a = new FindFalseVisitor();
    a.Visit(parser.root);
    var first = (parser.root.StList[0] as AssignNode);
    var second = (expectedTree.root.StList[0] as AssignNode);
    Assert.AreEqual(first, second);
}

```

10. Задача на оптимизацию: `a = a => null`

1. Команда, реализующая задачу Чубинидзе Н. Р., Романченко Р. Д.
2. Зависимые и предшествующие задачи

Предшествующие:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Данная задача основывается на использовании визиторов и решается путем замены узла оператора присваивания на `null`.

```

До:
a = a
После:
null

```

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

5. Реализация

При реализации использовался паттерн “Визитор”. Оптимизация была вынесена в отдельный класс-визитор и при обходе синтаксического дерева данным визитором, выражения типа “`a = a`” заменялись на пустой узел, который затем удалялся из дерева. Часть кода, отвечающий за такую замену приведена ниже:

```

if (n is AssignNode a)
{
    if (a.Expr is IdNode idNode && idNode.Name == a.Id.Name)
    {
        ReplaceStatement(a, new EmptyStatement());
    }
}
else if (n is BlockNode bl)
{
    bl.StList = bl.StList.Where(x => !(x is EmptyStatement)).ToList();
}

```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью SameAssignmentOptVisitor- оптимизирующий визитор, после чего из кода удаляются пустые выражения RemoveEmptyStatementVisitor. По оптимизированному дереву генерируется ТАС-код, который затем сравнивается с предполагаемым ответом. Тест с последовательным применением визиторов представлен ниже:

```

[Test]
public void SimpleTest1()
{
    var Text =
@"
{
    a = a;
}
";
    // . . . заполнение предков дерева

    var optimizers = new List<ChangeVisitor>
    {
        new SameAssignmentOptVisitor(),
        new RemoveEmptyStatementVisitor()
    };

    int countOptimization = 0;
    while (countOptimization < optimizers.Count)
    {
        parser.root.Visit(optimizers[countOptimization]);
        if (optimizers[countOptimization].IsChanged)
        {
            optimizers[countOptimization].IsChanged = false;

```

```

        countOptimization = 0;
    }
    else countOptimization++;
}
// . . . генерация ТАС-кода
var expected = new List<string>();
var actual = TAC.Instructions.Select(instruction =>
instruction.ToString().Trim());
CollectionAssert.AreEqual(expected, actual);
}

```

11. Задача на оптимизацию: if (true) st1; else st2; => st1;

1. Команда, реализующая задачу Погорелов А. А., Домбровская А. В.
2. Зависимые и предшествующие задачи

Предшествующие:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Данная задача решается путем замены узла условного оператора на выражение, находящееся по ветке True. Данная замена производится только при наличии в условии булевой константы True. Это задача может быть связана с предыдущей оптимизацией, поэтому данную оптимизацию стоит запускать после нее.

```

До:
if true
{
    st1;
}
После:
st1;

```

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

5. Реализация

При реализации использовался паттерн “Визитор”. Оптимизация была вынесена в отдельный класс-визитор и при обходе синтаксического дерева данным визитором, булевы константы “true” заменялись на узел, соответствующий выражению в первой ветке условного оператора. Часть кода, отвечающий за такую замену приведена ниже:

```

if (node.Condition is BoolValNode bv && bv.Val)
{
    Visit(node.Stat);
    ReplaceStatement(node, node.Stat);
    IsChanged = true;
}
else
{
    IsChanged = false;
    base.Visit(node);
}

```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью `TrueIfOptVisitor` - оптимизирующий визитор. По оптимизированному дереву генерируется ТАС-код, который затем сравнивается с предполагаемым ответом. Пример:

```

[Test]
public void SimpleIf()
{
    var Text =
@"
{
    if true
    {
        b = a;
    }
}
";

    Scanner scanner = new Scanner();
    scanner.SetSource(Text, 0);
    Parser parser = new Parser(scanner);
    parser.Parse();
    var parentFiller = new FillParentsVisitor();
    parser.root.Visit(parentFiller);

    var trueIfOpt = new TrueIfOptVisitor();
    parser.root.Visit(trueIfOpt);

    var prettyPrinter = new PrettyPrinterVisitor();
    parser.root.Visit(prettyPrinter);

    var TACGenerator = new TACGenerationVisitor();

```



```

parser.root.Visit(TACGenerator);
var TAC = TACGenerator.TAC;

var expected = new List<string>()
{
    "b = a"
};
var actual = TAC.Instructions.Select(instruction =>
instruction.ToString().Trim());
CollectionAssert.AreEqual(expected, actual);
}

```

Примечание: Данная задача также реализована командой Чухин А. И., Агафонцев Р. Г. Однако интегрировано в общий конвейер было решение, описанное выше. Тесты и реализация данной команды представлены в проекте.

12. Задача на оптимизацию: `if (false) st1; else st2; => st2`

1. Команда, реализующая задачу Османьян В. А., Маслова О. В.

2. Зависимые и предшествующие задачи

Предшествующие:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Данная задача решается путем замены узла условного оператора на выражение, находящееся по ветке `False`. Данная замена производится только при наличии в условии булевой константы `False`. Это задача может быть связана с предыдущей оптимизацией, поэтому данную оптимизацию стоит запускать после нее.

```

До:
if (false)
{
    st1;
}
else
{
    st2;
}
После:
st2;

```

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и

константы. Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

5. Реализация

При реализации использовался паттерн "Визитор". Оптимизация была вынесена в отдельный класс-визитор и при обходе синтаксического дерева данным визитором, булевы константы "false" заменялись на узел, соответствующий выражению в первой ветке условного оператора. Часть кода, отвечающий за такую замену приведена ниже:

```
if (node.Condition is BoolValNode bv && !bv.Val)
{
    if (node.ElseStat != null)
    {
        Visit(node.ElseStat);
        ReplaceStatement(node, node.ElseStat);
    }
    else
    {
        ReplaceStatement(node, new EmptyStatement());
    }
    IsChanged = true;
}
else
{
    IsChanged = false;
    base.Visit(node);
}
```

Так же для данной оптимизации прописан PostVisit

```
public override void PostVisit(Node n)
{
    if (n is BlockNode bl)
    {
        bl.StList = bl.StList.Where(x => !(x is EmptyStatement)).ToList();
    }
}
```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью SelfNotEqualTest - оптимизирующий визитор. По оптимизированному дереву генерируется ТАС-код, который затем сравнивается с предполагаемым ответом. Пример

```
[Test]
public void ifFalseTest()
{
    var parser = GenerateTree(
@"{
if (false)
{
a = 5;
}
else
{
b = 1;
}
}");
    var expectedTree = GenerateTree(
@"
{
{
b = 1;
};
}");
    var a = new IfFalseVisitor();
    a.Visit(parser.root);
    var first = (parser.root.StList[0] as AssignNode);
    var second = (expectedTree.root.StList[0] as AssignNode);
    Assert.AreEqual(first, second);
}
}
```

13. Задача на оптимизацию: if (ex) null; else null; => null;

1. Команда, реализующая задачу Чубинидзе Н. Р., Романченко Р. Д.

2. Зависимые и предшествующие задачи

Предшествующие:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Данная задача решается путем замены узла условного оператора на пустой оператор. Данная замена производится только при условии, что внутри данного оператора находится пустой оператор. Представленная оптимизация работает как для одного условного оператора без ветки else, так и с ее наличием.

```
До:
if true
{
    null;
```

```
}  
else  
{  
    null;  
}  
После:  
    null;
```

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Один визитор - одна группа действий, позволяющая выполнять простейшую оптимизацию, над узлами дерева. Контекст в данной задаче не важен.

5. Реализация

При реализации использовался паттерн “Визитор”. Оптимизация была вынесена в отдельный класс-визитор и при обходе синтаксического дерева данным визитором, условные операторы, содержащие пустые операторы полностью заменялись на пустые операторы. Часть кода, отвечающий за такую замену приведена ниже:

```
if (n is IfNode ifN)  
{  
    if ((ifN.Stat.StList.Count == 0 && ifN.ElseStat == null)  
        || (ifN.Stat.StList.Count == 0  
            && ifN.ElseStat.StList.Count == 0))  
    {  
        ReplaceStatement(ifN, new EmptyStatement());  
    }  
    if (ifN.ElseStat != null && ifN.ElseStat.StList.Count == 0)  
    {  
        ReplaceStatement(ifN, new IfNode(ifN.Condition, ifN.Stat));  
    }  
}  
else if (n is BlockNode bl)  
{  
    var prevLength = bl.StList.Count;  
    bl.StList = bl.StList.Where(x => !(x is EmptyStatement)).ToList();  
    IsChanged = prevLength != bl.StList.Count;  
}
```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью `NullIfElseOptVisitor` - оптимизирующий визитор, после чего из кода удаляются пустые выражения `RemoveEmptyStatementVisitor`. В данном тесте потребовалось использовать также `SameAssignmentOptVisitor` для получения пустого оператора. По оптимизированному дереву генерируется ТАС-код, который затем сравнивается с предполагаемым ответом. Тест с последовательным применением визиторов представлен ниже Пример:

```
[Test]
public void SimpleTest1()
{
    var Text =
@"
{
    if (a > b)
    {
        a = a;
    }
}
";

    // . . . заполнение предков дерева

    var optimizers = new List<ChangeVisitor>
    {
        new SameAssignmentOptVisitor(),
        new NullIfElseOptVisitor(),
        new RemoveEmptyStatementVisitor()
    };

    int countOptimization = 0;
    while (countOptimization < optimizers.Count)
    {
        parser.root.Visit(optimizers[countOptimization]);
        if (optimizers[countOptimization].IsChanged)
        {
            optimizers[countOptimization].IsChanged = false;
            countOptimization = 0;
        }
        else countOptimization++;
    }

    // . . . генерация ТАС-кода
    var expected = new List<string>();
    var actual = TAC.Instructions.Select(instruction =>
instruction.ToString().Trim());
    CollectionAssert.AreEqual(expected, actual);
}
```

14. Задача на оптимизацию: `while (false) st => null`

1. Команда, реализующая задачу Гуртовой А. И., Остапенко М. В.

2. Зависимые и предшествующие задачи

Предшествующие:

- Построение синтаксического дерева

Зависимые: -

3. Аннотация

Реализация данной задачи представляет собой замену узла оператора цикла while на пустой узел с помощью визитора.

```
До:
while false
{
    a = 3;
}
После:
null;
```

4. Теория

Визитор – группа действий, выполняющая простейшую оптимизацию над узлами AST-дерева.

5. Реализация

В реализации данной задачи использовался отдельный класс-визитор, созданный специально для этой задачи. При обходе дерева циклы while типа “while (false) st” заменяются на узел, представляющий собой пустой узел, содержащий в себе “null”. Ниже приведен код, отвечающий за реализацию оптимизации:

```
public class EmptyStatement : StatementNode
{
    public EmptyStatement() { }
    public override void Visit(Visitor v)
    {

    }
}

. . .

if (node.Condition is BoolValNode bv && bv.Val == false)
{
    ReplaceStatement(node, new EmptyStatement());
}
else
{
    base.Visit(node);
}
```

6. Интеграция в общий проект

Конвейер оптимизации по синтаксическому дереву устроен следующим образом: необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, после чего происходит их циклическое применение до тех пор, пока дерево программы изменяется. Для интеграции отдельного оптимизатора, достаточно добавить его в нужную коллекцию оптимизаторов.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке строится синтаксическое дерево, заполняются все предки, затем оно оптимизируется с помощью `WhileFalseVisitor` - оптимизирующий визитор. По оптимизированному дереву генерируется TAC-код, который затем сравнивается с предполагаемым ответом.

```
[Test]
public void Test1()
{
    var Text =
@"
{
a = 3;
while (false)
{
    b = a;
}
}";

    Scanner scanner = new Scanner();
    scanner.SetSource(Text, 0);
    Parser parser = new Parser(scanner);
    parser.Parse();
    var parentFiller = new FillParentsVisitor();
    parser.root.Visit(parentFiller);
    var whileFalse = new WhileFalseVisitor();
    parser.root.Visit(whileFalse);
    var prettyPrinter = new PrettyPrinterVisitor();
    parser.root.Visit(prettyPrinter);
    var TACGenerator = new TACGenerationVisitor();
    parser.root.Visit(TACGenerator);
    var TAC = TACGenerator.TAC;

    var expected = new List<string>()
    {
        "a = 3"
    };
    var actual = TAC.Instructions.Select(instruction =>
instruction.ToString().Trim());
    CollectionAssert.AreEqual(expected, actual);
}
```

Генерация трехадресного кода. Построение базовых блоков и `ControlFlowGraph`, задачи для интеграции.

Для интеграции задач в общий проект, необходимо было решить следующие задачи.

№	Задания
1	Архитектура системы оптимизаций
2	Генерация трехадресного кода для всех конструкций
3	Создание структуры ББл и CFG - графа ББл
4	Разбиение на ББл + слияние ББл
5	Архитектура системы тестирования оптимизаций
6	Вычисление передаточной функции для достигающих определений композицией передаточных функций команд

1. Архитектура Системы Оптимизаций

1. Команда, реализующая задачу Гарьковенко А., Руднев. Д

2. Зависимые и предшествующие задачи

Предшествующие:

- генерация трехадресного кода;
- все оптимизации по дереву;
- разбиение на базовые блоки.

Зависимые: -

3. Аннотация

Решение данной задачи отвечает за реализацию общего конвейера оптимизаций. Решение данной задачи отвечает за реализацию общего конвейера оптимизаций. Решение данной задачи отвечает за реализацию общего конвейера оптимизаций.

4. Теория

Все оптимизации необходимо выполнять до тех пор, пока есть изменений в коде, в определенном порядке: сначала выполняется первая оптимизация до тех пор, пока она актуальна, затем выполняется вторая, после этого вновь первая, если первая не дала изменений - вторая и т.д. Так же, что касается трехадресных блоков, то необходимо сначала выполнить все возможные оптимизации для каждого базового блока, затем выполнить определенные оптимизации, которые независимы от базового блока.

5. Реализация

Были реализованы два класса, один Общий для трехадресных оптимизаций, и один для оптимизаций по дереву, они оба содержат списки всех, возможных оптимизаций. Ниже представлен код, отвечающий за генерацию списка оптимизаций по синтаксическому дереву:

```
public static List<ChangeVisitor> ChangeVisitorsOptimization = new List<ChangeVisitor>
{
    new SameAssignmentOptVisitor(),
    new NumberEqualityVisitor(),
```



```

new MinusSelf(),
new FindFalseVisitor(),
new WhileFalseVisitor(),
new MultiplyOnZero(),
new MultiplyOnZero2(),
new TrueConditionOptVisitor(),
new TrueIfOptVisitor(),
new NullIfElseOptVisitor(),
new RemoveEmptyStatementVisitor(),
new ConstantFoldingVisitor(),
new IfFalseVisitor(),
new AlgebraicIdentityProdDiv1Visitor(),
new AlgebraicIdentitySum0Visitor()
};

```

Для организации работы все оптимизаторов была написана функция, которая последовательно проверяет, отработал ли какой-нибудь из оптимизаторов, если да, то итеративный процесс начинается заново:

```

public static void Optimization(Parser parser)
{
    int countOptimization = 0;
    while (countOptimization < ChangeVisitorsOptimization.Count)
    {
        parser.root.Visit(ChangeVisitorsOptimization[countOptimization]);
        if (ChangeVisitorsOptimization[countOptimization].IsChanged)
        {
            ChangeVisitorsOptimization[countOptimization].IsChanged = false;
            countOptimization = 0;
        }
        else countOptimization++;
    }
}

```

Аналогичный конвейер реализован для оптимизаций по ТАС-коду.

6. Интеграция в общий проект

Для интеграции со всеми оптимизациями, были реализован список оптимизаций

7. Тесты

Тестирование проходит следующим образом: сначала по заданной программе генерируется ТАС-код, затем к нему применяется общий конвейер оптимизаций, который запускается командой `Optimization()`. Затем оптимизированный ТАС-код сравнивается с предполагаемым ответом. Пример:

```

[Test]
public void WhileFalseOptimization()
{
    var Text =
@"

```

```

{
x = 13;
b = 14;
while (7 - 7 == 3 - 1)
{
    x = x * (a - a);
    b = b;
}
}
";

/* построение синтаксического дерева */

AllVisitorsOptimization.Optimization(parser);

var prettyPrinter = new PrettyPrinterVisitor();
parser.root.Visit(prettyPrinter);

var TACGenerator = new TACGenerationVisitor();
parser.root.Visit(TACGenerator);
var TAC = TACGenerator.TAC;

var expected = new List<string>()
{
    "x = 13",
    "b = 14"
};
var actual = TAC.Instructions.Select(instruction =>
instruction.ToString().Trim());
CollectionAssert.AreEqual(expected, actual);
}

```

2. Генерация трехадресного кода для всех конструкций

1. Команда, реализующая задачу Погорелов А. А., Домбровская А. В.
2. Зависимые и предшествующие задачи

Предшествующие:

- построение AST дерева.

Зависимые:

- учет алгебраических тождеств;
- живые и мертвые переменные, удаление мертвого кода;
- протяжка констант;
- протяжка копий;
- оптимизация общих подвыражений;
- устранение переходов через переходы;

- свертка констант;
- создание структуры ББл.

3. Аннотация

Данная задача важна для всех последующих, так как она закладывает основу для оптимизаций, которые неэффективно или невозможно применять на AST.

4. Теория

Трехадресный код - это линейаризованное AST-дерево. Используется при оптимизации в тех случаях, когда использование AST-дерева неэффективно. В данном проекте используется следующее подмножество команд:

```
x = y op z;
x = y;
x = op y;
if x goto L
goto L.
```

где op - любая алгебраическая операция, L - метка. Представление в виде пятерок:

Метка	op	arg1	arg2	result	Команда
L	+	a	b	x	x = a + b
	-	a		x	x = - a
	assign	a		x	x = a
	goto	L1			goto L1
L1	ifgoto	x	L		if x goto L

5. Реализация

Трехадресный код принято хранить в виде списка трехадресных инструкций. При этом, трехадресная инструкция представляет собой пятерку: метку инструкции, операцию, первый аргумент, второй аргумент и возможная переменная-результат. Для генерации списка трехадресных инструкций, был использован паттерн "Визитор". При обходе каждого конкретного узла синтаксического дерева, генерируется соответствующая ему инструкция. Ниже представлен пример обхода узла условного выражения:

```
var genCond = GenerateTACExpr(node.Condition);
var label1 = GenerateTempLabel();
var label2 = GenerateTempLabel();
AddInstruction("if goto", genCond, label1, "");
if (node.ElseStat != null)
node.ElseStat.Visit(this);
AddInstruction("goto", label2, "", "");
AddInstruction("", "", "", "", label1);
node.Stat.Visit(this);
AddInstruction("", "", "", "", label2);
```

Однако, при обходе узла выражения, использовалась специальная рекурсивная функция `GenerateTACExpr`. Это сделано для того, чтобы эффективно и просто генерировать очередную временную переменную, если в правой части выражения содержится более 2 аргументов. Часть кода этой функции, отвечающую за генерацию трехадресных инструкций для бинарных выражений, можно увидеть ниже:

```
var bin = ex as BinExprNode;
string tmp1 = GenerateTACExpr(bin.Left);
string tmp2 = GenerateTACExpr(bin.Right);
string tmp = GenerateTempName();
AddInstruction(bin.OpType.ToFriendlyString(), tmp1, tmp2, tmp);
return tmp;
```

6. Интеграция в общий проект

Генерация трехадресного кода заложила основу для других заданий. При обходе синтаксического дерева специальным визитором `TACGenerationVisitor`, создается экземпляр специального класса `ThreeAddressCode`, позволяющий получать доступ к отдельным инструкциям и базовым блокам.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке генерируется TAC-код, затем полученный код сравнивается с предполагаемым ответом. Тестами покрыты все возможные в нашем языке операторы. Пример:

```
[Test]
public void SimpleGeneration()
{
    var TAC = GenerateTAC(
@"
{
    int x;
    x = 14;
    y = 2 * (a + b) - c;
    x = x + x;
}
");
    var expected = new List<string>()
    {
        "x = 14",
        "#t0 = a + b",
        "#t1 = 2 * #t0",
        "#t2 = #t1 - c",
        "y = #t2",
        "#t3 = x + x",
        "x = #t3"
    };
    var actual = TAC.Instructions.Select(instruction =>
instruction.ToString().Trim());
    CollectionAssert.AreEqual(expected, actual);
}
```

3. Создание структуры ББл и CFG - графа ББл

1. Команда, реализующая задачу Османян В., Маслова О.

2. Зависимые и предшествующие задачи

Предшествующие

- Генерация трехадресного кода для всех конструкций

Зависимые

- Построение CFG (Control Flow Graph)
- все итерационные алгоритмы.

3. Аннотация

В этом задании вводится представление промежуточного кода в виде графа, полезное при рассмотрении генерации кода (даже если алгоритм генерации кода не строит граф явным образом).

4. Теория

Представление строится следующим образом:

1) Промежуточный код разделяется на базовые блоки (basic blocks), представляющие собой максимальные последовательности следующих друг за другом трехадресных команд, обладающие приведенными ниже свойствами:

а) поток управления может входить в базовый блок только через первую команду блока, т.е. переходы в середину блока отсутствуют;

б) управление покидает блок без останова или ветвления, за исключением, возможно, в последней команде блока.

2) Базовые блоки становятся узлами графа потока (CFG), ребра которого указывают порядок следования блоков.

Данная оптимизация позволяет в дальнейшем рассматривать трансформации графов потоков, которые преобразуют исходный промежуточный код в "оптимизированный" промежуточный код, позволяющий генерировать более качественный целевой код. "Оптимизированный" промежуточный код превращается в машинный код с помощью методов генерации трехадресного кода (ThreeAddressCode.cs).

5. Реализация

Исходя из определений, приведенных в источнике Дж. Ульман, А. Ахо "Компиляторы: принципы, технологии и инструментарий", была реализована структура в виде классов, содержащих связь ББл и CFG:

```
public class BasicBlock
{
    private static int index;

    private List<TACInstruction> instructions = new List<TACInstruction>();
```

```

private List<BasicBlock> _Out = new List<BasicBlock>();
private List<BasicBlock> _In = new List<BasicBlock>();

public int Index { get; private set; }

public List<TACInstruction> Instructions
{
    get
    {
        return instructions;
    }
}

public List<BasicBlock> In
{
    get
    {
        return _In;
    }
    set
    {
        _In = value;
    }
}

public List<BasicBlock> Out
{
    get
    {
        return _Out;
    }
    set
    {
        _Out = value;
    }
}

public BasicBlock()
{
    instructions = new List<TACInstruction>();
    _Out = new List<BasicBlock>();
    _In = new List<BasicBlock>();
    Index = index++;
}

public BasicBlock(List<TACInstruction> instr)
{
    instructions = instr;
    _Out = new List<BasicBlock>();
    _In = new List<BasicBlock>();
    Index = index++;
}

```

```

    }

    public override int GetHashCode()
    {
        return Index;
    }

    public override bool Equals(object obj)
    {
        if (obj == null)
        {
            return false;
        }
        var second = obj as BasicBlock;
        if (second == null)
        {
            return false;
        }
        return second.Index == Index;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        foreach (var i in Instructions)
        {
            builder.Append(i.ToString().Trim());
            builder.Append('\n');
        }
        return builder.ToString();
    }

    /// <summary>
    /// Обнуляет внутренний статический счетчик индексов
    /// </summary>
    public static void clearIndexCounter()
    {
        index = 0;
    }
}

```

6. Интеграция в общий проект

Команда, реализовавшая алгоритм разбиения программы на ББл, воспользовалась алгоритмом определения структуры ББл. Данная задача является опорной, поскольку от её качества реализации зависит в целом результат выполнения заданий остальными командами.

7. Тесты

Также необходимая архитектура тестов была имплементирована и задокументирована:

```

public class TACTestsBase
{

```

```

protected ThreeAddressCode GenerateTAC(string sourceCode)
{
    Scanner scanner = new Scanner();
    scanner.SetSource(sourceCode, 0);

    Parser parser = new Parser(scanner);
    parser.Parse();

    var parentFiller = new FillParentsVisitor();
    parser.root.Visit(parentFiller);

    var TACGenerator = new TACGenerationVisitor();
    parser.root.Visit(TACGenerator);
    return TACGenerator.TAC;
    /*
    var TACBlocks = new TACBaseBlocks(TACGenerator.Instructions);
    TACBlocks.GenBaseBlocks();
    var cfg = new ControlFlowGraph(TACBlocks.blocks);
    */
}
}

```

4. Разбиение на ББл + слияние ББл

1. Команда, реализующая задачу Чубинидзе Н. Р., Романченко Р. Д.

2. Зависимые и предшествующие задачи

Предшествующие:

- генерация трехадресного кода;
- создание структуры ББл.

Зависимые:

- построение CFG;
- все итерационные алгоритмы.

3. Аннотация

Разбиение на базовые блоки лежит в основе построения CFG и всех итерационных алгоритмов.

4. Теория

Базовый блок - это максимальная последовательность команд трехадресного кода, удовлетворяющая следующим условиям:

- поток управления может входить в ББл только через первую команду;
- управление покидает ББл без останова или ветвления, за исключением, возможно, последней команды. Базовый блок также может быть определен как блок от лидера до лидера, где команда-лидер это:
 - первая команда;
 - любая команда, на которую есть переход;

- любая команда, непосредственно следующая за переходом.

5. Реализация

Реализация данной задачи базируется структуре ББл, определенной другой командой. Генерация блоков проходит в соответствии с определением, заявленным выше. Сначала в список блоков добавляется первая команда, затем инструкции, на которые есть переход:

```
if (instructions[i - 1].Operation.Contains("goto"))
{
    var label = "";
    if (instructions[i - 1].Operation.Equals("goto"))
        label = instructions[i - 1].Argument1;
    else if (instructions[i - 1].Operation.Equals("if goto"))
        label = instructions[i - 1].Argument2;
    for (int j = 1; j < instructions.Count; ++j)
    {
        if (instructions[j].HasLabel && instructions[j].Label.Equals(label))
            list.Add(j);
    }
    list.Add(i);
}
```

А затем операции, следующие сразу за переходом:

```
if (instructions[instructions.Count - 1].Operation.Equals("goto"))
    for (int j = 1; j < instructions.Count; ++j)
    {
        if (instructions[j].HasLabel && instructions[j]
            .Label.Equals(instructions[instructions.Count - 1].Argument1))
            list.Add(j);
    }
```

Наконец, объединение базовых блоков происходит следующим образом:

```
foreach (var block in blocks)
{
    foreach (var instr in block.Instructions)
    {
        merging.Add(instr);
    }
}
```

6. Интеграция в общий проект

Генерация трехадресного кода заложила основу для CFG и итерационных алгоритмов. При последовательном обходе инструкций ТАС, создается экземпляр специального класса `TASBaseBlocks`, позволяющий получать доступ к отдельным инструкциям и базовым блокам.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке генерируется ТАС-код, затем на основе полученного списка инструкций генерируются базовые блоки. В конце результат разбиения на блоки сравнивается с предполагаемым ответом. Пример:

```
[Test]
public void SimpleOneBlockTest1()
{
    var TAC = GenerateTAC(
@"
{
    int x;
    x = 14;
    y = 2 * (a + b) - c;
    x = x + x;
}
");
    var blocks = new TACBaseBlocks(TAC.Instructions);
    blocks.GenBaseBlocks();
    Assert.AreEqual(blocks.blocks.Count, 1);

    var expected = new List<List<string>>()
    {
        new List<string>()
        {
            "x = 14",
            "#t0 = a + b",
            "#t1 = 2 * #t0",
            "#t2 = #t1 - c",
            "y = #t2",
            "#t3 = x + x",
            "x = #t3"
        }
    };
    var actual = blocks.blocks
        .Select(block => block.Instructions
            .Select(instr => instr.ToString().Trim()).ToList());
    Assert.AreEqual(expected, actual);
}
```

5. Архитектура системы тестирования оптимизаций

1. Команда, реализующая задачу Гуртовой А. И., Остапенко М. В.

2. Зависимые и предшествующие задачи

Предшествующие: - Зависимые: -

3. Аннотация

Архитектура системы тестирования оптимизаций позволяет выявить недостатки оптимизаций и проблемы совместимости этих оптимизаций. В качестве главного инструмента тестирования было выбрано расширение для Visual Studio Unit Test - NUnit3TestAdapter.

4. Теория

-

5. Реализация

Для того, чтобы получить доступ к возможностям NUnit3TestAdapter, необходимо было сначала добавить его к пакетам NuGet проекта. Затем в тестовых файлах каждой оптимизации нужно было подключить NUnit.Framework. После этого каждый класс для тестирования оптимизаций был помечен как [TestFixture]. Все тестирующие функции для каждого отдельного случая помечаются как [Test]. Ниже приведен пример содержимого тестового файла:

```
[TestFixture]
class OptimizerNameTests: TestsBase
{
    [Test]
    public void Test1() { /* содержимое теста */ }
    [Test]
    public void Test2() { /* содержимое теста */ }
    /* другие тесты */
}
```

В конце каждой тестовой функции происходит сравнение результатов и в случае их несовпадения, данный тест помечается, как провалившийся во вкладке TestExplorer. Иначе, тест помечается, как успешно пройденный. Задача всех групп сделать так, чтобы все случаи были покрыты тестами и все тесты были помечены, как успешно пройденные.

6. Интеграция в общий проект

Был добавлен отдельный проект для тестов и подключен фреймворк NUnit.

7. Тесты

-

6. Вычисление передаточной функции для достигающих определений композицией передаточных функций команд.

1. Команда, реализующая задачу Манукян Г. А.

2. Зависимые и предшествующие задачи

Предшествующая:

- Построение графа потока управления (CFG – Control Flow Graph)
- Вычисление передаточной функции для достигающих определений
- Создание структуры базовых блоков, графа CFG
- Разбиение на ББл (от лидера до лидера) + слияние ББл в единый список

Зависимые:

- Итерационный алгоритм для достигающих определений, ~в обобщенной структуре

3. Аннотация

Цель оптимизации – в каждой инструкции 1,2,3...n базового блока вычислить свою передаточную функцию, объединить их в композицию, потом проверить вычисление общей передаточной функции. С точки зрения обхода CFG, выделяют прямой и обратный обходы. В каждой отдельной ситуации значение передаточной функции композиции функции вычисляется единообразно.

4. Теория

Передаточная функция f_B блока B по определению равна композиции передаточных функций его инструкций I_1, \dots, I_n :

$$f_B(x) = f_{I_n}(f_{I_{n-1}}(\dots f_{I_1}(x)\dots)) = (f_{I_1} \circ f_{I_2} \circ \dots \circ f_{I_n})(x)$$

или

$$f_B = f_{I_1} \circ f_{I_2} \circ \dots \circ f_{I_n}$$

Пример: $d: u = v + w$ $f_d(X) = \text{gend}_d \sqcup (X - \text{kill}_d)$, где $\text{gend}_d = \{d\}$, $\text{kill}_d = \{\}$
 gen_B – мн-во определений, генерируемых и не переопределённых базовым блоком B . kill_B – мн-во остальных определений переменных, определяемых в опр-ях gen_B , в других ББл.

Композиция передаточных функций:

$$f_1(X) = \text{gen}_1 \cup (X - \text{kill}_1)$$

$$f_2(X) = \text{gen}_2 \cup (X - \text{kill}_2)$$

$$f_2(f_1(X)) = \text{gen}_2 \cup (\text{gen}_1 \cup (X - \text{kill}_1) - \text{kill}_2) =$$

$$= \text{gen}_2 \cup (\text{gen}_1 - \text{kill}_2) \cup (X - (\text{kill}_1 \cup \text{kill}_2)) = G \cup (X - K)$$

В общем случае:

$$f_B(X) = \text{gen}_B \cup (X - \text{kill}_B), \text{ где}$$

$$\text{kill}_B = \text{kill}_1 \cup \text{kill}_2 \cup \dots \cup \text{kill}_n$$

$$\text{gen}_B = \text{gen}_n \cup (\text{gen}_{n-1} - \text{kill}_n) \cup (\text{gen}_{n-2} - \text{kill}_{n-1} - \text{kill}_n) \cup \dots$$

$$\cup (\text{gen}_1 - \text{kill}_2 - \dots - \text{kill}_n)$$

5. Реализация

Так как речь идет о передаточной функции инструкции определенного базового блока, естественно сформировать таблицы для множеств из определения функции:

```
private ILookup<string, TACInstruction> def_b;
private ILookup<BasicBlock, TACInstruction> gen_b;
private ILookup<BasicBlock, TACInstruction> kill_b;
```

Теперь в два прохода по блокам $O(\text{Len}_B)$ заполняем таблицы, то есть для каждой инструкции, в которой есть оператор „=“ мы добавляем ее во множество defs:

```
List<TACInstruction> defs = new List<TACInstruction>();
foreach (var block in blocks)
    foreach (var instruction in block.Instructions)
        if (instruction.Operation == "=")
            defs.Add(instruction);
def_b = defs.ToLookup(x => x.Result, x => x);
```

Множества `gen`, `kill` получены, как и следует из определения, следующим образом:

1. Каждое сгенерированное блоком `B` определение переменной ("`=`"), и не переопределенное им же;
2. Если в таблице `defs` встретилось определение, то оно автоматически добавляется в `kill`, искл. `gen`:

```
List<(BasicBlock, TACInstruction)> gen = new List<ValueTuple<BasicBlock,
TACInstruction>>();
List<(BasicBlock, TACInstruction)> kill = new List<ValueTuple<BasicBlock,
TACInstruction>>();
foreach (var block in blocks)
{
    var flag = new HashSet<string>();
    foreach (var instruction in block.Instructions.Reverse<TACInstruction>())
    {
        if (!flag.Contains(instruction.Result) && instruction.Operation == "=")
        {
            gen.Add((block, instruction));
            flag.Add(instruction.Result);
        }
        foreach (var exclude_def in def_b[instruction.Result].Where(x => x !=
instruction))
            kill.Add((block, exclude_def));
    }
}
gen_b = gen.ToLookup(x => x.Item1, x => x.Item2);
kill = kill.Distinct().ToList();
kill_b = kill.ToLookup(x => x.Item1, x => x.Item2);
```

Для дополнительной функциональности, добавлены функции возвращения всех трех множеств, а также применения – `ApplyTransferFunc` к списку `IEnumerable`.

6. Интеграция в общий проект

В классе, отведенном под граф потока управления добавлена следующая функциональность: список всех потомков и список всех предков текущего базового блока (ББЛ). В задаче «Итерационный алгоритм в обобщенной структуре» данная функциональность необходима» В задаче на обобщенный итерационный алгоритм, необходимый для написания итерационного алгоритма распространения констант, использует данную функцию, а также модифицированный CFG. Таким образом, передаточная функция для достигающих определений композицией передаточных функций команд полностью интегрирована и участвует в реализации итерационного алгоритма.

7. Тесты

Унаследовавшись от базового класса `IterAlgoGeneric` – обобщенного базового алгоритма (см. задание 4), применена передаточная функция для дост. опр. композ. пер. ф-ций команд: Класс-наследник – итерационный алгоритм:

```
public class SampleClassIterAlgoForTransferFunc :
IterAlgoGeneric<IEnumerable<TACInstruction>>
{
```

```

/// <inheritdoc/>
    public override Func<IEnumerable<TACInstruction>, IEnumerable<TACInstruction>,
IEnumerable<TACInstruction>> CollectingOperator => (a, b) => a.Union(b);

/// <inheritdoc/>
    public override Func<IEnumerable<TACInstruction>, IEnumerable<TACInstruction>,
bool> Compare
        => (a, b) => !a.Except(b).Any() && !b.Except(a).Any();

/// <inheritdoc/>
    public override IEnumerable<TACInstruction> Init { get =>
Enumerable.Empty<TACInstruction>(); protected set { } }

/// <inheritdoc/>
    public override Func<BasicBlock, IEnumerable<TACInstruction>,
IEnumerable<TACInstruction>> TransferFunction { get; protected set; }

    public override InOutData<IEnumerable<TACInstruction>> Execute(
ControlFlowGraph graph)
    {
        TransferFunction = new ReachingTransferFunc(graph).Transfer;
        return base.Execute(graph);
    }

    public override void Run()
    {
        this.Execute(Cfg);
    }
}

```

Тесты – среднее время компиляции – 3 мс, всего 6 тестов, из них проходят – 6.

```

using InOutInfo = InOutData<IEnumerable<TACInstruction>>;
[TestFixture]
class TransferFuncForIterativeAlgorithm : CFGTestsBase
{
    protected (List<BasicBlock> basicBlocks, InOutInfo inOutInfo)
GenGraphAndGetInOutInfo(string program)
    {
        var TAC = GenerateTAC(program);
        var TACBlocks = new TACBaseBlocks(TAC.Instructions);
        var cfg = new ControlFlowGraph(TACBlocks.blocks);
        var inOutInfo = new SampleClassIterAlgoForTransferFunc().Execute(cfg);
        return (TACBlocks.blocks, inOutInfo);
    }

    [Test]
    public void TestMultipleIfStatements()
    {
        (var blocks, var inOutInfo) = GenGraphAndGetInOutInfo(
@"

```

```

{
int a, b;
a = 5;
if a > 0
{
a = 0;
}
else
{
a = 1;
}
b = a;
}
");
Assert.AreEqual(6, inOutInfo.Count);

var falseBranch = blocks[1].Instructions.Take(1);
var trueBranch = blocks[2].Instructions.Take(1);
var lastBlock = blocks[3].Instructions.Skip(1);
CollectionAssert.AreEqual(falseBranch.Concat(trueBranch),
inOutInfo[blocks[3]].In);
CollectionAssert.AreEqual(falseBranch.Concat(trueBranch).Concat(lastBlock),
inOutInfo[blocks[3]].Out);
}

[Test]
public void BasicTest()
{
    (var blocks, var inOutInfo) = GenGraphAndGetInOutInfo(
@"
{
int one;
one = 1;
}
");

// only one basic block + entry and exit
Assert.AreEqual(3, inOutInfo.Count);

Assert.AreEqual(0, inOutInfo[blocks[0]].In.Count());
Assert.AreEqual(1, inOutInfo[blocks[0]].Out.Count());
Assert.AreEqual(blocks[0].Instructions, inOutInfo[blocks[0]].Out);
}

```

Оптимизации базовых блоков

После построения трехадресного кода и разбиения его на базовые блоки появится задача оптимизации этих базовых блоков. Их можно разделить на локальные и глобальные оптимизации.

Локальная и глобальная оптимизации:

- Оптимизация в пределах базового блока – локальная

- Оптимизация между базовыми блоками – глобальная
- Локальная оптимизация существенно проще и позволяет во многих случаях существенно увеличить производительность малой ценой

Для оптимизаций базовых блоков необходимо решить следующие задачи:

№	Задание
1	Живые и мертвые переменные и удаление мертвого кода (замена на пустой оператор)
2	Учет алгебраических тождеств
3	Протяжка констант. Протяжка копий
4	Оптимизация общих подвыражений
5	Устранение переходов через переходы
6	Def-Use информация: накопление информации и удаление мертвого кода на ее основе
7	Свертка констант
8	Устранение переходов к переходам. Удаление пустых операторов

1. Живые и мертвые переменные и удаление мертвого кода (замена на пустой оператор)

1. Команда, реализующая задачу Гарьковенко А., Руднев. Д.

2. Зависимые и предшествующие задачи

Предшествующие:

- генерация трехадресного кода;

Зависимые: -

3. Аннотация

Данная задача основывается на анализе трехадресного кода внутри базового блока. Все переменные в конце блока объявляются живыми, и дальше начинается обход инструкций снизу вверх и переменная встречается слева от присваивания она объявляется мертвой. В конце мертвые переменные заменятся на блок "Empty".

4. Теория

Use - множество всех использований переменной. Каждый use хранит ссылку на def переменной либо null, в случае, когда переменная не определяется в пределах ББл. Def - множество всех определений переменной. Каждый def хранит список use данной переменной в пределах ББл. Пример:

```

c1:    x = a;           // определение x (def)
      . . .
c2:    y = x + z;       // использование x (use)

```


Если в промежутке между `s1` и `s2` `x` никак не переопределялось, то `x` является живой на этом участке кода, иначе - мертвой.

5. Реализация

Оптимизация была вынесена в отдельный класс, в котором при проходе по инструкциям снизу вверх переменным присваивается булево значение, отвечающее за то, является переменная живой или мертвой. В начале, поскольку алгоритм идет снизу вверх, определяется тип последней инструкции, как представлено ниже:

```
var last = Instructions.Last();
newInstructions.Add(last);
assignmentInfo.Add(last.Result, false);
if (!int.TryParse(last.Argument1, out _))
    && last.Argument1 != "True"
    && last.Argument1 != "False")
    assignmentInfo[last.Argument1] = true;
if (!int.TryParse(last.Argument2, out _))
    && last.Argument2 != "True"
    && last.Argument2 != "False")
    assignmentInfo[last.Argument2] = true;
```

Далее запускается итерационный процесс по оставшимся инструкциям:

```
var inst = Instructions[i];
if (inst.Operation == "Empty")
{
    newInstructions.Add(inst);
    continue;
}
if (assignmentInfo.ContainsKey(inst.Result) && !assignmentInfo[inst.Result])
{
    newInstructions.Add(new TACInstruction("Empty", null, null, null, inst.Result));
    continue;
}
/* определение типа, как показано выше, для inst*/
```

6. Интеграция в общий проект

Процедура оптимизации с использованием всех оптимизаторов ТАС, проходит в специальном классе, инкапсулирующем в себе конвейер оптимизаций. Для интеграции отдельного оптимизатора в общий проект, необходимо просто добавить экземпляр данного оптимизатора в коллекцию оптимизаторов конвейера.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке генерируется ТАС-код, затем к нему применяется оптимизатор `DeadAliveOptimize`, который запускается командой `Run()`. Затем оптимизированный ТАС-код сравнивается с предполагаемым ответом. Пример:

```

[Test]
public void OneBlock1()
{
    var TAC = GenerateTAC(
@"
{
    a = d;
    x = a;
    a = e;
    x = b;
    y = x + z;
}
");

    var optimizer = new DeadAliveOptimize(TAC);
    optimizer.Run();
    var actual = optimizer.Instructions;
    var expected = new List<TACInstruction>()
    {
        new TACInstruction("Empty", null, null, null, "a"),
        new TACInstruction("Empty", null, null, null, "x"),
        new TACInstruction("=", "e", "", "a", ""),
        new TACInstruction("=", "b", "", "x", ""),
        new TACInstruction("+", "x", "z", "#t0", ""),
        new TACInstruction("=", "#t0", "", "y", ""),
    };
    Assert.AreEqual(actual, expected);
}

```

2. Учет алгебраических тождеств

1. Команда, реализующая задачу Погорелов А. А., Домбровская А. В.

2. Зависимые и предшествующие задачи

Предшествующие:

- генерация трехадресного кода;
- создание структуры ББл;
- разбиение на ББл.

Зависимые: -

3. Аннотация

Учет алгебраических тождеств - это замена следующих выражений на переменные и константы:

```

x + 0 = x;
0 + x = x;
x - 0 = x;
x - x = 0;

```

```
x * 1 = x;  
1 * x = x;  
x * 0 = 0;  
x / 1 = x;  
x / x = 1.
```

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Трехадресный код принято хранить в виде списка трехадресных инструкций. При этом, трехадресная инструкция представляет собой пятерку: метку инструкции, операцию, первый аргумент, второй аргумент и возможная переменная-результат.

5. Реализация

Т.к. блок трехадресного кода представляет собой список трехадресных инструкций, оптимизатор блока представляет собой отдельный класс, который проходит по такому списку инструкций и изменяет их в соответствии с задачей. Так, оптимизатор алгебраических тождеств заменяет, например, инструкцию `"a = b + 0"` на инструкцию `"a = b"`. Ниже приведен фрагмент кода, отвечающий за оптимизацию инструкций, содержащих оператор `"+"`:

```
if (c.Operation.Equals("+"))  
{  
    if (c.Argument1.Equals("0"))  
    {  
        c.Argument1 = c.Argument2;  
        c.Argument2 = "";  
        c.Operation = "=";  
    }  
    else if (c.Argument2.Equals("0"))  
    {  
        c.Argument2 = "";  
        c.Operation = "=";  
    }  
}
```

6. Интеграция в общий проект

Процедура оптимизации с использованием всех оптимизаторов ТАС, проходит в специальном классе, инкапсулирующем в себе конвейер оптимизаций. Для интеграции отдельного оптимизатора в общий проект, необходимо просто добавить экземпляр данного оптимизатора в коллекцию оптимизаторов конвейера.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке генерируется ТАС-код, затем к нему применяется оптимизатор `AlgebraicIdentitiesOptimizer`, который запускается командой `Run()`. Затем оптимизированный ТАС-код сравнивается с предполагаемым ответом. Пример:

```
[Test]
public void SimpleExample()
{
    var TAC = GenerateTAC(
@"
{
    x = x + 0;
}
");
    var AIOptimizer = new AlgebraicIdentitiesOptimizer(TAC);
    AIOptimizer.Run();
    var expected = new List<string>()
    {
        "#t0 = x", "x = #t0"
    };
    var actual = AIOptimizer.TAC.Instructions.Select(instruction =>
                                                    instruction.ToString().Trim());
    CollectionAssert.AreEqual(expected, actual);
}
```

3. Протяжка констант. Протяжка копий.

1. Команда, реализующая задачу Чухин А. И., Агафонов Р. Г.

2. Зависимые и предшествующие задачи

Предшествующие:

- Трехадресный код

Зависимые: -

3. Аннотация

Протянуть копии и константы где это возможно.

4. Теория

И в протяжке копий и в протяжке констант совершает проход по всем инструкциям трехадресного кода, в процессе заполняются множества известных переменных и их значений. С их помощью происходит замена копий и констант. При любом изменении области видимости множества обнуляются.

5. Реализация

Участок кода, отвечающий за работу с константами:

```
public void OptimizeConstants()
{
    var knownConstants = new HashSet<string>();
    var values = new Dictionary<string, string>();
    foreach (var current in Instructions)
    {
        if (!(current.Result.Equals("""))))
        {
```

```

        if (knownConstants.Contains(current.Argument1))
        {
            current.Argument1 = values[current.Argument1];
        }
        if (knownConstants.Contains(current.Argument2))
        {
            current.Argument2 = values[current.Argument2];
        }
        if (!current.Result.Contains("#"))
        {
            if (current.Argument2 == ""
                && double.TryParse(current.Argument1, out double c))
            {
                if (!current.Argument1.Contains("#"))
                {
                    knownConstants.Add(current.Result);
                    values[current.Result] = current.Argument1;
                }
                else
                {
                    knownConstants.Remove(current.Result);
                }
            }
        }
    }
else
{
    if (current.Operation is "if goto"
        || current.Operation is "goto")
    {
        knownConstants.Clear();
        values.Clear();
    }
}
}
}

```

Участок кода, отвечающий за работу с копиями:

```

public void OptimizeCopy()
{
    var knownVariables = new HashSet<string>();
    var values = new Dictionary<string, string>();
    foreach (var current in Instructions)
    {
        if (!current.Result.Equals(""))
        {
            if (knownVariables.Contains(current.Argument1))
            {
                current.Argument1 = values[current.Argument1];
            }
        }
    }
}

```

```

        if (knownVariables.Contains(current.Argument2))
        {
            current.Argument2 = values[current.Argument2];
        }

        if (!current.Result.Contains("#"))
        {
            // если это буква
            if (current.Argument2 == ""
                && !double.TryParse(current.Argument1, out _))
            {
                if (!current.Argument1.Contains("#"))
                {
                    knownVariables.Add(current.Result);
                    values[current.Result] = current.Argument1;
                }
                else
                {
                    knownVariables.Remove(current.Result);
                }
            }
        }
    }
    else
    {
        if (current.Operation is "if goto"
            || current.Operation is "goto")
        {
            knownVariables.Clear();
            values.Clear();
        }
    }
}

```

6. Интеграция в общий проект

Созданный класс является наследником общего для подобных оптимизаций абстрактного класса `TASOptimizer`, переопределенная функция `Run()` проводит оптимизации.

7. Тесты

- Тесты для проверки на протягивание констант:

```

до
@"
{
a = b;
c = 0;
d = c + 1;
e = d * b;
a = x - y;

```

```

k = c + a;
}
"
после
@"
{
a = b;
c = 0;
d = 0 + 1;
e = d * b;
a = x - y;
k = 0 + a;
}
"

```

-
- Тесты для проверки на протягивание копий:
-

```

до
@"
{
a = b;
c = b - a;
d = c + 1;
e = d * a;
a = x - y;
k = c + a;
}
"
после
@"
{
a = b;
c = b - b;
d = c + 1;
e = d * b;
a = x - y;
k = c + a;
}
"

```

4. Оптимизация общих подвыражений

1. Команда, реализующая задачу Османян В. А., Маслова О. В.

2. Зависимые и предшествующие задачи

Предшествующая:

- Генератор трехадресного кода

Зависимые: -

3. Аннотация

В том случае, если в каких то присваиваниях в трех адресном коде присутствуют одинаковые подвыражения, то вместо последующего повторения подвыражения производится подстановка переменной, которой уже присвоили искомое значение. Исключения: случаи когда переменные участвующие в расчетах, изменяют свои значения между такими расчетами.

До:

$a = b + c$

$b = a - d$

$c = b + c$

$d = a - d$

После:

$a = b + c$

$b = a - d$

$c = a$

$d = b$

4. Теория

Абстрактное синтаксическое дерево (АСД) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы. Трехадресный код принято хранить в виде списка трехадресных инструкций. При этом, трехадресная инструкция представляет собой пятерку: метку инструкции, операцию, первый аргумент, второй аргумент и возможная переменная-результат. Выражение E называется общим подвыражением, если E было ранее вычислено и значение переменных в E с того времени не изменилось.

5. Реализация

Т.к. блок трехадресного кода представляет собой список трехадресных инструкций, оптимизатор блока представляет собой отдельный класс, который проходит по такому списку инструкций и изменяет их в соответствии с задачей. Так, оптимизатор алгебраических тождеств заменяет, например, инструкцию " $a = b + c$ " на инструкцию " $a = d$ ", где d соответствующая переменная. Фрагмент кода, выполняющий это:

```
if (Instructions.Count == 0)
    return;
for(int i=1; Instructions.Count>i;++i )
{
    if((Instructions[i].Argument1.Length>0) &&(Instructions[i].Argument2.Length > 0))
    {
        var op = Instructions[i].Operation;
        var arg1 = Instructions[i].Argument1;
        var arg2 = Instructions[i].Argument2;
        for (int j = i-1; 0 <= j; --j)
        {
            if((Instructions[j].Operation == op)
                &&(Instructions[j].Argument1 == arg1)
                && (Instructions[j].Argument2 == arg2))
            {
                // replacement logic
            }
        }
    }
}
```



```

        && Check_Varieble(Instructions[j].Result,j+1, i)
        && Check_Varieble(Instructions[j].Argument1,j+1, i )
        && Check_Varieble(Instructions[j].Argument2,j+1, i))
    {
        Instructions[i].Operation = "=";
        Instructions[i].Argument1 = Instructions[j].Result;
        Instructions[i].Argument2 = "";
        break;
    }
}
}
}

```

6. Интеграция в общий проект

Процедура оптимизации с использованием всех оптимизаторов ТАС, проходит в специальном классе, инкапсулирующем в себе конвейер оптимизаций. Для интеграции отдельного оптимизатора в общий проект, необходимо просто добавить экземпляр данного оптимизатора в коллекцию оптимизаторов конвейера.

7. Тесты

Для тестирования сначала исходный код теста подается на вход парсеру, после чего генерируется трехадресный код. Сгенерированный трехадресный код оптимизируется с помощью оптимизатора общих подвыражений. Образчиком выступает написанный вручную, с учетом априорно верных оптимизаций, отрывок трехадресного кода. Сгенерированный код и образчик сравниваются. Ниже можно увидеть пример теста для оптимизатора доступных выражений: Пример:

```

[Test]
public void Simple()
{
    var TAC = GenerateTAC(
@"
{
a = 3 + 5;
b = 3 + 5;
f = a + b;
e = 3 * a;
r = a + b;
}
");
    var optimizer = new CommonExpressionsOptimizer(TAC);
    optimizer.Run();

    var actual = optimizer.Instructions.Select(i => i.ToString().Trim()).ToList();
    var expected = new List<string>()
    {
        "a = 3 + 5",
        "b = a",
        "f = a + b",
        "e = 3 * a",
        "r = f"
    }
}

```

```
};
Assert.AreEqual(expected, actual);
}
```

5. Устранение переходов через переходы

1. Команда, реализующая задачу Чубинидзе Н. Р., Романченко Р. Д.
2. Зависимые и предшествующие задачи

Предшествующие:

- генерация трехадресного кода;.

Зависимые: -

3. Аннотация

Устранение переходов через переходы - это глобальная оптимизация, которая преобразует код следующим образом:

```

До:
if (cond) goto L1
goto L2
L1: st1;
L2:
После:
if (!cond) goto L2
st1;
L2;

```

4. Теория

goto - это оператор безусловного перехода. При оптимизации устранения переходов через переходы условие в условном операторе заменяется на противоположное, таким образом идет избавление от излишних переходов.

5. Реализация

Т.к. блок трехадресного кода представляет собой список трехадресных инструкций, оптимизатор блока представляет собой отдельный класс, который проходит по такому списку инструкций и изменяет их в соответствии с задачей. Так, при устранении переходов через переходы идет замена следующим образом: если встречается переход по условному оператору, а следующая операция - переход, то условие в операторе меняется на обратное, а последующие две инструкции удаляются. Реализация данного алгоритма представлена ниже:

[illegible]

```

        Instructions[i].Argument1 = tempName;
        Instructions[i].Argument2 = Instructions[i + 1].Argument1;
        Instructions[i + 1] = null;
        Instructions[i + 2] = null;
        Instructions.Insert(i, notInstruction);
    }
}

```

6. Интеграция в общий проект

Процедура оптимизации с использованием всех оптимизаторов TAC, проходит в специальном классе, инкапсулирующем в себе конвейер оптимизаций. Для интеграции отдельного оптимизатора в общий проект, необходимо просто добавить экземпляр данного оптимизатора в коллекцию оптимизаторов конвейера.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке генерируется TAC-код, затем к нему применяется оптимизатор `GotoOptimizer`, который запускается командой `Run()`. Затем оптимизированный TAC-код сравнивается с предполагаемым ответом. Пример:

```

[Test]
public void GoTotoGoToSimple()
{
    var sourceCode =
@"
{
a = 1;
b = 5;
if a > b
{
    goto 6;
}
6: a = 4;
}
";
    var TACGen = GenerateTAC(sourceCode);

    var GoTotoGoToOptimizer = new GoTotoGoTo_EmptyOptimizer(TACGen);
    GoTotoGoToOptimizer.Run();
    var TACGenAfter = GenerateTAC(sourceCode);
    var actual = TACGenAfter.Instructions
        .Select(instruction => instruction.ToString().Trim());
    var expected = new List<string>()
    {
        "a = 1",
        "b = 5",
        "#t0 = a > b",
        "if #t0 goto #L0",
        "goto #L1",
        "#L0",
        "goto 6",
        "#L1",
    }
}

```

```
        "6",  
        "a = 4"  
    };  
    CollectionAssert.AreEqual(expected, actual);  
}
```

6. Def, Use информация.

1. Команда, реализующая задачу Гуртовой А. И., Остапенко М. В.

2. Зависимые и предшествующие задачи

Предшествующие:

- генерация трехадресного кода;

Зависимые:

- удаление мертвого кода;
- протяжка констант;
- протяжка копий.

3. Аннотация

В данной задаче необходимо для каждой переменной в базовом блоке сохранить информацию о том, где переменная определяется и какие использования затрагивают данное определение переменной. Кроме того, была реализована функция удаления мертвого кода в пределах одного базового блока.

4. Теория

Use - множество всех использований переменной. Каждый use хранит ссылку на def переменной либо null, в случае, когда переменная не определяется в пределах ББл. Def - множество всех определений переменной. Каждый def хранит список use данной переменной в пределах ББл.

Пример:

```
x = 7;           // определение x (def)  
. . .  
y = x + z;       // использование x (use)
```

Оптимизации на основе использования Def-Use информации:

- удаление мертвого кода, если список Use пуст;
- протяжка констант, если def - константа, то заменяем;
- протягивание копий, если def - переменная, то заменяем

5. Реализация

Для данной задачи был реализован оптимизатор, который инкапсулирован в специальный класс DefUseOptimizer. При вызове метода Run() данного оптимизатора, первым делом создаются множество Use и список Def для текущего базового блока. Код для их создания приведен ниже:

```

DefList = new List<Def>();
for (int i = 0; i < commands.Count; ++i)
{
    if (operations.Contains(commands[i].Operation))
        DefList.Add(new Def(i, commands[i].Result));
    AddUse(commands[i].Argument1, commands[i], i);
    AddUse(commands[i].Argument2, commands[i], i);
}

```

После этого для каждой команды базового блока в обратном порядке происходит удаление мертвого кода, как представлено ниже:

```

if (curDefInd != -1 && DefList[curDefInd].Uses.Count == 0
    && (c.Result[0] != '#' ? curDefInd != lastDefInd : true))
{
    DeleteUse(commands[i].Argument1, i);
    DeleteUse(commands[i].Argument2, i);
    result.Add(new TACInstruction("", "", "", "", commands[i].Label));
    isChange = true;
}
else
    result.Add(commands[i]);

```

Наконец, в зависимости от того, был ли удален хоть один участок кода, преобразуется список инструкций ББл и на место удаленной инструкции помещается пустая строка.

6. Интеграция в общий проект

Процедура оптимизации с использованием всех оптимизаторов ТАС, проходит в специальном классе, инкапсулирующем в себе конвейер оптимизаций. Для интеграции отдельного оптимизатора в общий проект, необходимо просто добавить экземпляр данного оптимизатора в коллекцию оптимизаторов конвейера.

7. Тесты

Для тестирования сначала исходный код теста подается на вход парсеру, после чего генерируется трехадресный код. Далее оптимизатору на вход подается список команд трехадресного кода, который оптимизируется при запуске функции Run(). Полученный результат сравнивается с предполагаемым ответом. Ниже разобран простейший из вариантов:

```

[Test]
public void SimpleExample()
{
    var TAC = GenerateTAC(
@"
{
    a = b;
    a = c;
}
");
    var optimizer = new DefUseOptimizer(TAC);

```

```

optimizer.Run();
var expected = new List<string>()
{
    "",
    "a = c"
};
var actual = optimizer.TAC.Instructions
    .Select(instruction => instruction.ToString().Trim());
CollectionAssert.AreEqual(expected, actual);
}

```

7. Свертка констант в трехадресном коде

1. Команда, реализующая задачу Погорелов А. А., Домбровская А. В.
2. Зависимые и предшествующие задачи

Предшествующая:

- генерация трехадресного кода

Зависимые: -

3. Аннотация

Данная задача решается путем замены в трехадресном коде числового выражения на константу. Оптимизация работает для всех операций (сложение, умножение, деление, разность).

```

До:
x = 2 * 3 + 4;
После:
#t0 = 6;
x = #t0 + 4;

```

4. Теория

Свертка констант - оптимизация, уменьшающая избыточные вычисления, путём замены константных выражений на их значения.

5. Реализация

Оптимизация была вынесена в отдельный класс, в котором при проходе по инструкциям трехадресного кода числовые выражения заменяются на соответствующую константу. Часть кода, отвечающий за такую замену приведена ниже:

```

foreach (var c in Instructions)
{
    if (c.Operation.Equals("+")
        || c.Operation.Equals("-")
        || c.Operation.Equals("*")
        || c.Operation.Equals("/"))
    {
        var arg1 = 0.0;
    }
}

```

```

var arg2 = 0.0;
var arg1IsDigit = double.TryParse(c.Argument1, out arg1);
var arg2IsDigit = double.TryParse(c.Argument2, out arg2);
if (arg1IsDigit && arg2IsDigit)
{
    double res = 0.0;
    switch (c.Operation)
    {
        case "+":
            res = arg1 + arg2;
            break;
        ...
    }
    c.Argument1 = res.ToString();
    c.Argument2 = "";
    c.Operation = "=";
}
}
}

```

6. Интеграция в общий проект

Процедура оптимизации с использованием всех оптимизаторов ТАС, проходит в специальном классе, инкапсулирующем в себе конвейер оптимизаций. Для интеграции отдельного оптимизатора в общий проект, необходимо просто добавить экземпляр данного оптимизатора в коллекцию оптимизаторов конвейера.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке генерируется ТАС-код, затем к нему применяется оптимизатор `ConstantFoldingOptimizer`, который запускается командой `Run()`. Затем оптимизированный ТАС-код сравнивается с предполагаемым ответом. Пример:

```

[Test]
public void SubsequentOptimizations()
{
    var TAC = GenerateTAC(
@"
{
    y = 6 / 2 - 2 * 1;
    x = y * 4;
}
");
    var optimizer = new ConstantFoldingOptimizer(TAC);
    optimizer.Run();

    var expected = new List<string>()
    {
        "#t0 = 3",
        "#t1 = 2",
        "#t2 = #t0 - #t1",
        "y = #t2",
    }
}

```

```

        "#t3 = y * 4",
        "x = #t3"
    };
    var actual = TAC.Instructions
        .Select(instruction => instruction.ToString().Trim());
    CollectionAssert.AreEqual(expected, actual);
}

```

8. Устранение переходов к переходам. Удаление пустых операторов.

1. Команда, реализующая задачу Манукян Г. А.

2. Зависимые и предшествующие задачи

Предшествующие:

- построение AST.
- генерация трехадресного кода;

Зависимые: -

3. Аннотация

Устранение переходов к переходам, удаление пустых операторов - это локальная оптимизация над трехадресным кодом, преобразующая его следующим образом:

```

1)   До:
      goto L1;
      ...
      L1: goto L2;
      После:
      goto L2;
      ...
      L1: goto L2;

2)   До:
      if (/*усл*/) goto L1;
      ...
      L1: goto L2;
      После:
      if (/*усл*/) goto L2;
      ...
      L1: goto L2;

3)   До:
      goto L1;
      ...
      L1: if (/*усл*/) goto L2;
      L3:
      После:
      ...
      if (/*усл*/) goto L2;
      goto L3;

```



```
...  
L3:
```

4. Теория

goto - это оператор безусловного перехода. При оптимизации устранения переходов через переходы условие в условном операторе заменяется на противоположное, таким образом идет избавление от излишних переходов.

5. Реализация

При помощи специальной структуры public struct GoToScan – сканера стало возможным поддерживать параметр номера команды в трёхадресном коде, для обеспечения поиска сложности $O(1)$. Поскольку при устранении переходов к переходам идет замена метки в трёхадресном коде на которой стоит goto или ifgoto, на соответствующую метку, на которую существует goto стоящий в коде предыдущей метки:

```
{  
    var wasChanged = false; // флаг, проведенной оптимизации  
    List<GoToScan> list = new List<GoToScan>(); // Список всех переходов  
    List<TACInstruction> TACcode = new List<TACInstruction>(); // Трёхадресный код  
}
```

Функция устранения переходов к переходам:

```
for (int i = 0; i < commands.Count; i++)  
{  
    tmpcommands.Add(commands[i]);  
    if (commands[i].Operation == "goto")  
    {  
        list.Add(new GoToScan(i, commands[i].Label, commands[i].Argument1));  
    }  
  
    if (commands[i].Operation == "ifgoto")  
    {  
        list.Add(new GoToScan(i, commands[i].Label, commands[i].Argument2));  
    }  
}  
  
for (int i = 0; i < tmpcommands.Count; i++)  
{  
    if (tmpcommands[i].Operation == "goto")  
    {  
        for (int j = 0; j < list.Count; j++)  
        {  
            if (list[j].labelto == tmpcommands[i].Argument1)  
            {  
                if (tmpcommands[i].Argument1.ToString() ==  
list[j].labelfrom.ToString())  
                {  
                    changed |= false;  
                }  
            }  
        }  
    }  
}
```

```

        else
        {
            changed |= true;
            tmpcommands[i] = new TACInstruction(tmpcommands[i].Label, "goto",
list[j].labelfrom.ToString(), "", "");
        }

    }

}

if (tmpcommands[i].Operation == "ifgoto")
{
    for (int j = 0; j < list.Count; j++)
    {
        if (list[j].labelto == tmpcommands[i].Argument2)
        {
            if (tmpcommands[i].Argument2.ToString() ==
list[j].labelfrom.ToString())
            {
                changed |= false;
            }
            else
            {
                tmpcommands[i] = new TACInstruction(tmpcommands[i].Label,
"ifgoto", tmpcommands[i].Argument1, list[j].labelfrom.ToString(), "");
                changed |= true;
            }
        }
    }
}
}
}

```

Довольно проста и тривиальна идея удаления оператора: здесь необходимо найти поор-команду, либо переход на команду с поор-флагом, а затем удалить метку:

```

if (currentCommand.Operation == "noop" && currentCommand.Label == "") changed = true;
else if (currentCommand.Operation == "noop")
{
    if (commands[i + 1].Label == "")
    {
        var nextCommand = commands[i + 1];
        changed = true;
        result.Add(new TACInstruction(
            currentCommand.Label,
            nextCommand.Operation,
            nextCommand.Argument1,
            nextCommand.Argument2,
            nextCommand.Result

```

```

        )
    );
    i += 1;
    if (i == commands.Count - 1)
    {
        toAddLast = false;
    }
}
else
{
    var nextCommand = commands[i + 1];
    changed = true;
    var currentLabel = currentCommand.Label;
    var nextLabel = nextCommand.Label;

    result = result.Select(com => com.Operation == "goto"
        && com.Argument1 == currentLabel
        ? new TACInstruction(com.Label, com.Operation,
nextLabel, com.Argument2, com.Result): com).ToList();

    for (var j = i + 1; j < commands.Count; j++)
    {
        commands[j] = commands[j].Operation == "goto" && commands[j].Argument1
== currentLabel ? new TACInstruction(
            commands[j].Label,
            commands[j].Operation,
            nextLabel,
            commands[j].Argument2,
            commands[j].Result
        ): commands[j];
    }
}
}

```

6. Интеграция в общий проект

Применяются вышеописанные методы в конвейере оптимизации (необходимые классы-оптимизаторы помещаются в коллекцию оптимизаторов, они циклически вызываются каждый раз, когда обновляется трехадресный код). Разработаны тесты (NUnit.Framework - TestSuite), где данный оптимизатор вызывается для особых случаев. Все, что потребовалось для интеграции данной оптимизации – реализовать метод `public abstract void Run();` класса `TACOptimizer`. Поскольку данная оптимизация может быть оттестирована в отдельности, как и в предыдущей задаче, есть два варианта тестирования – с интеграцией и без.

7. Тесты

В тестах проверяется, что применение оптимизации устранения переходов к переходам к заданному трехадресному коду, возвращает ожидаемый результат (в каждом тесте – работа непосредственно с трехадресным кодом, без интеграции):

```

[Test]
public void GoToToGoToSimple()
{

```

```

    var sourceCode = @"
{
a = 1;
b = 5;
if a > b
{
goto 6;
}
6: a = 4;
}
";

    var TACGen = GenerateTAC(sourceCode);
    var GoTotoGoToOptimizer = new GoTotoGoTo_EmptyOptimizer(TACGen);
    GoTotoGoToOptimizer.Run();
    var TACGenAfter = GenerateTAC(sourceCode);
    var actual = TACGenAfter.Instructions.Select(instruction =>
instruction.ToString().Trim());
    var expected = new List<string>()
    {
        "a = 1",
        "b = 5",
        "#t0 = a > b",
        "if #t0 goto #L0",
        "goto #L1",
        "#L0",
        "goto 6",
        "#L1",
        "6",
        "a = 4"
    };
    CollectionAssert.AreEqual(expected, actual);
}

[Test]
public void GoTotoGoToIfElse()
{
    var sourceCode = @"
{
int a, b;
b = 5;
if a > b
{
goto 6;
} else
{
goto 4;
}
6: a = 4;
4: a = 6;
}
";
    <...>

```

```

var expected = new List<string>()
{
    "b = 5",
    "#t0 = a > b",
    "if #t0 goto #L0",
    "goto 4",
    "goto #L1",
    "#L0",
    "goto 6",
    "#L1",
    "6",
    "a = 4",
    "4",
    "a = 6",
};
CollectionAssert.AreEqual(expected, actual);
}

```

Итерационные алгоритмы

На данном этапе решаются задачи потока данных, состоящие в поиске решения для множества ограничений, накладываемых на $IN[S]$ и $OUT[S]$, где через IN и OUT обозначаются значения потока данных до и после каждой инструкции s . Существует два типа ограничений: основанные на семантике (передаточные функции) и на потоке управления. Суть использования первого вида ограничений заключается в следующем: значения потока данных перед инструкцией и после неё ограничены семантикой этой инструкции. Пусть анализ потока данных - определение константного значения переменных в точках. Например, переменная a имеет значение val перед выполнением $a = b$, то обе переменные - a , b после выполнения инструкции имеют значение val . Соотношение между потоками данных до и после инструкции присваивания - передаточная функция. Второе множество ограничений связано с потоком управления. Внутри базового блока поток управления очень простой. Если базовый блок B состоит из инструкций $s_1, s_2, s_3, \dots, s_n$, то значение потока управления на выходе s_i такое же, что и значение потока управления на входе s_{i+1} . Более подробно о ребрах графа потока управления **между** базовыми блоками рассматривается в следующем разделе.

Реализованные задачи в рамках данного этапа:

№	Название задачи
1	Итерационный алгоритм в структуре распространения констант
2	Доступные выражения
3	Разработка альтернативной реализации хранения IN и OUT в виде битовых векторов. Интеграция этого представления в итерационный алгоритм о достигающих определениях
4	Активные переменные
5	Достигающие определения
6	Передаточная функция в структуре распространения констант
7	Итерационный алгоритм в обобщённой структуре

1. Итерационный алгоритм в структуре распространения констант и передаточная функция.

1. Команда, реализующая задачу Гарьковенко А. Руднев. Д.

2. Зависимые и предшествующие задачи

Предшествующая:

- разбиение на базовые блоки
- построение Control Flow Graph
- Итерационный алгоритм

Зависимые:

- Определение натуральных циклов

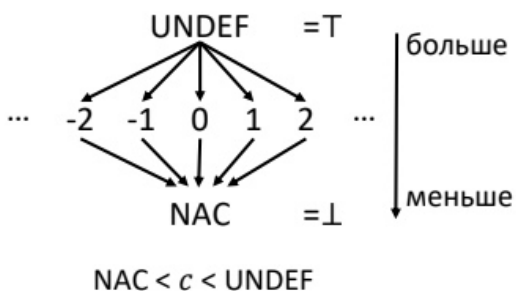
3. Аннотация задачи

Заменить в программе переменные, имеющие константное значение, на константу.

4. Теория

Полурешетка и оператор сбора:

Значения полурешетки V_i



Оператор сбора на полурешетке V_i

$UNDEF \wedge v = v$ (v – переменная)

$NAC \wedge v = NAC$

$c \wedge c = c$

$c_1 \wedge c_2 = NAC$

Итерационный Алгоритм:

```

 $Out[Вход] = (UNDEF, UNDEF, \dots, UNDEF) = \top$ 
foreach  $B - \text{ББл}, B \neq \text{Вход}$ 
     $Out[B] = (UNDEF, UNDEF, \dots, UNDEF) = \top$ 
while (внесены изменения в  $Out$ )
{
    foreach  $B - \text{ББл}, B \neq \text{Вход}$ 
    {
         $IN[B] = \bigwedge_{P-\text{предш } B} OUT[P]$ 
         $OUT[B] = f_B(IN[B])$ 
    }
}

```

5. Реализация

Была создана структура Полурешетки и оператор сбора для неё:

```

public SemilatticeValue collecting(SemilatticeValue second) =>
    Type == SemilatticeData.NAC || second.Type == SemilatticeData.NAC
    ? new SemilatticeValue(SemilatticeData.NAC)
    : Type == SemilatticeData.UNDEF
    ? second
    : second.Type == SemilatticeData.UNDEF
    ? this
    : ConstValue == second.ConstValue
    ? second
    : new SemilatticeValue(SemilatticeData.NAC);

```

Далее класс итерационного алгоритма был унаследован от обобщенного итерационного алгоритма и переопределены функции: порядка выполнения, сравнения блоков, и так далее:

```

public override Func<Dictionary<string, SemilatticeValue>, Dictionary<string, SemilatticeValue>, bool> Compare
=> (a, b) => !a.Where(entry => b[entry.Key] != entry.Value).Any();

public static Dictionary<string, SemilatticeValue> Collect(Dictionary<string, SemilatticeValue> first, Dictionary<string, SemilatticeValue> second)
{
    var result = new Dictionary<string, SemilatticeValue>(first.Count, first.Comparer);
    foreach (var elem in second)
    {
        result[elem.Key] = first[elem.Key].collecting(elem.Value);
    }
}

```

```
    }  
    return result;  
}
```

И после этого было вызвано исполнение базового класса. На основе полученной полурешетки была выполненная оптимизация протяжки констант.

```
private void Optimize(InOutData<Dictionary<string, SemilatticeValue>> insOuts)  
{  
    var blocks = Cfg.blocks;  
    foreach (var b in blocks)  
    {  
        var curIn = insOuts[b].Item1;  
        foreach (var i in b.Instructions)  
        {  
            if (curIn.ContainsKey(i.Argument1) && curIn[i.Argument1].Type ==  
SemilatticeData.CONST)  
            {  
                i.Argument1 = curIn[i.Argument1].ConstValue;  
            }  
            if (curIn.ContainsKey(i.Argument2) && curIn[i.Argument2].Type ==  
SemilatticeData.CONST)  
            {  
                i.Argument2 = curIn[i.Argument2].ConstValue;  
            }  
            if (curIn.ContainsKey(i.Result) && curIn[i.Result].Type ==  
SemilatticeData.CONST)  
            {  
                if (i.Argument2 == "" && int.TryParse(i.Argument1, out var temp))  
                {  
                    curIn[i.Result] = new SemilatticeValue(SemilatticeData.CONST,  
temp);  
                }  
                else  
                {  
                    curIn[i.Result] = new SemilatticeValue(SemilatticeData.NAC);  
                }  
            }  
        }  
    }  
}
```

6. Интеграция

Поскольку данный оптимизатор представляет собой обособленный класс, который наследуется от абстрактного класса оптимизаций, для интеграции в общий проект необходимо просто добавить создание его экземпляра в классе конвейера оптимизаций по трехадресному коду.

7. Тесты

Тестирование проходит следующим образом: сначала по заданной строке генерируется TAC-код, затем к нему применяется оптимизатор `ConstantPropogationIter`, который запускается командой `Optimize()`. Затем оптимизированный TAC-код сравнивается с предполагаемым ответом.

```
[Test]
public void Nested()
{
    var TAC = GenerateTAC(
@"
{
b = 3;
if a
{
goto 1;
}
b = 3;
1: r = b;
}
");
    var blocks = new TACBaseBlocks(TAC.Instructions);
    blocks.GenBaseBlocks();
    var cfg = new ControlFlowGraph(blocks.blocks);
    var optimizer = new ConstantPropogationIter();
    optimizer.Cfg = cfg;
    optimizer.Instructions = TAC.Instructions;
    optimizer.Blocks = blocks.blocks;
    optimizer.Run();
    var actual = optimizer.Instructions.Select(i => i.ToString().Trim()).ToList();
    var expected = new List<string>()
    {
        "b = 3",
        "if a goto #L0",
        "goto #L1",
        "#L0",
        "goto 1",
        "#L1",
        "b = 3",
        "1",
        "r = 3"
    };
    Assert.AreEqual(expected, actual);
}
```

2. Доступные выражения

1. Команда, реализующая задачу Погорелов А. А., Домбровская А. В.
2. Зависимые и предшествующие задачи

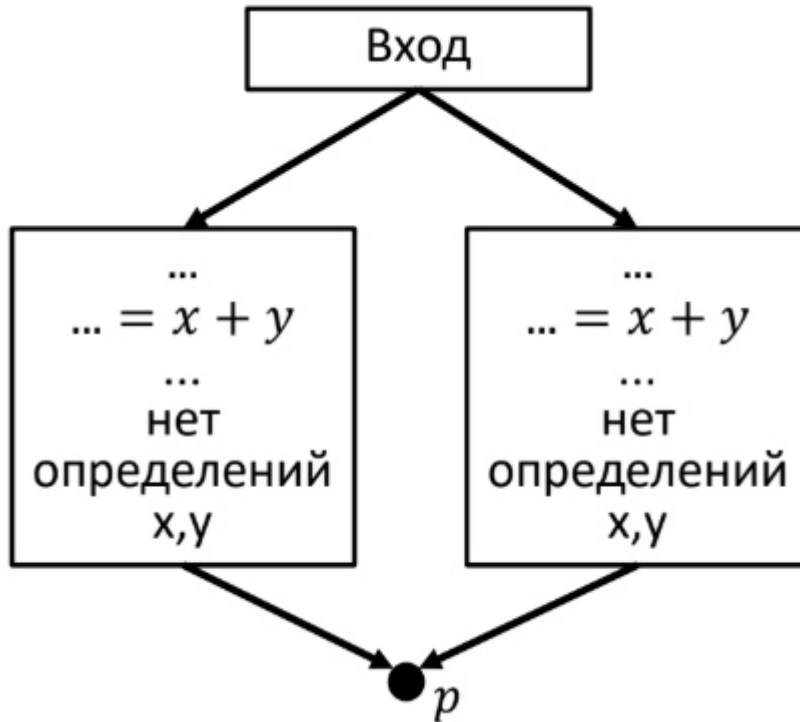
Предшествующие:

- построение CFG.

Зависимые: -

3. Аннотация

В данной задаче необходимо при прямом проходе алгоритма определить доступные выражения на входе и на выходе каждого из ББл. Для этой задачи также потребуются определения множеств, уничтожаемых блоком В, и множеств, генерируемых блоком В.



4. Теория

Выражение доступно в точке p , если любой путь от входа к p вычисляет это выражение и после последнего вычисления до достижения p не никаких изменений этого выражения. Блок В уничтожает выражение $x + y$, если он изменяет значение x или y и затем не перевычисляет значение выражения $x + y$. Блок В генерирует выражение $x + y$, вычисляет значение выражения $x + y$ и затем не переопределяет значения x и y .

Алгоритм: Вход: граф потока управления, в котором для каждого ББл вычислены уничтожаемые и генерируемые множества выражений. **Выход:** множества выражений, доступных на входе $IN[V]$ и на выходе $OUT[V]$ для всех ББл В.

```

OUT[ВХОД] = ∅;
for (каждый блок В, кроме входного)
    OUT[V] = U;
while (OUT изменяется)
    for (каждый блок В, кроме входного)
        IN[V] = ∩ OUT[P] // где P - предшественник В
        OUT[V] = e_gen[V] ∪ (IN[V] - e_kill[V]);
  
```

5. Реализация

Для данной задачи был реализован итерационный алгоритм, который был инкапсулирован в специальный класс `AvailableExpressionsOptimizer`. При вызове метода `Run()` данного оптимизатора, первым делом создаются множества `Gen` и `Kill` для каждого базового блока. Код для их создания приведен ниже:

```
foreach (var instruction in block.Instructions)
{
    if (instruction.Operation.IsArithmetic())
    {
        gen.AddNewExpression(instruction);
    }
    if (instruction.Result.IsVariable())
    {
        gen.RemoveExpression(instruction.Result);
    }
}
```

После этого вычисляются множества `In` и `Out`, используя специальный класс `AvailableExpressionsTable`, инкапсулирующий в себе таблицу доступных выражений. Основной код, вычисляющий множества `In` и `Out` приведен ниже:

```
while (changed)
{
    changed = false;
    foreach (var block in blocks)
    {
        var prevOuts = new List<AvailableExpressionsTable>();
        foreach (var prev in block.In)
        {
            prevOuts.Add(Out[prev]);
        }
        In[block] = AvailableExpressionsTable.Intersection(prevOuts);
        var oldOut = Out[block];
        Out[block] = CalculateOut(In[block], block);
        if (!oldOut.Equals(Out[block]))
        {
            changed = true;
        }
    }
}
```

Наконец, после вычисления множеств `In` и `Out` происходит оптимизация внутри каждого блока с учетом доступных выражений в каждой точке программы.

7. Интеграция

Поскольку оптимизатор доступных выражений представляет собой обособленный класс, который наследуется от абстрактного класса оптимизаций, для интеграции в общий проект

необходимо просто добавить создание его экземпляра в классе конвейера оптимизаций по трехадресному коду.

7. Тесты

Для тестирования сначала исходный код теста подается на вход парсеру, после чего генерируется трехадресный код и строится граф потока управления программы. С помощью графа потока управления трехадресный код оптимизируется с помощью оптимизатора доступных выражений. Образчиком выступает написанный вручную, с учетом априорно верных оптимизаций, отрывок трехадресного кода. Сгенерированный код и образчик сравниваются. Ниже можно увидеть пример теста для оптимизатора доступных выражений:

```
var TAC = GenerateTAC(
@"
{
x = 1 + y;
t = 1 + y;
y = 2;
z = 1 + y;
}
");
var blocks = new TACBaseBlocks(TAC.Instructions);
blocks.GenBaseBlocks();
var cfg = new ControlFlowGraph(blocks.blocks);
var optimizer = new AvailableExpressionsOptimizer();
optimizer.Run(cfg, blocks.blocks);
var actual = blocks.blocks.Select(b => b.ToString().Trim());
var expected = new List<string>() {
    "#t0 = 1 + y\n" +
    "x = #t0\n" +
    "#t1 = #t0\n" +
    "t = #t1\n" +
    "y = 2\n" +
    "#t2 = 1 + y\n" +
    "z = #t2"
};
Assert.AreEqual(actual, expected);
```

3. Разработка альтернативной реализации хранения IN и OUT в виде битовых векторов. Интеграция этого представления в итерационный алгоритм о достигающих определениях.

1. Команда, реализующая задачу Чухин А. И., Агафонцев Р. Г.
2. Зависимые и предшествующие задачи

Предшествующие задачи:

- Достигающие определения

Зависимые: -

3. Аннотация

Для данных IN и OUT необходимо построить битовые вектора, однозначно отображающие их содержимое.

4. Теория

Составив список всех присваиваний в коде, IN и OUT можно представить как битовые векторы, где значение true будет означать наличие данного присваивания в IN или OUT, а false соответственно его отсутствие.

Представим определения битовыми векторами

Block B	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

$$\begin{aligned}
 IN[B_2]^1 &= OUT[B_1]^1 \cup OUT[B_4]^0 \\
 &= 111\ 0000 + 000\ 0000 = 111\ 0000 \\
 OUT[B_2]^1 &= gen[B_2] \cup (IN[B_2]^1 - kill[B_2]) \\
 &= 000\ 1100 + (111\ 0000 - 110\ 0001) = 001\ 1100
 \end{aligned}$$

5. Реализация

Для хранения вектора был использован стандартный класс BitArray. Над ним реализуется класс-обертка InOutVector с переопределенными операциями сложения, вычитания и сравнения. Также реализуется отдельный класс InOutVectorCreator для перевода IN и OUT в векторный вид, хранящий массив присваиваний. Класс InOutVectorCreator:

```

public class InOutVectorCreator
{
    static List<TACInstruction> assignList;
    public InOutVectorCreator(ThreeAddressCode a)
    {
        changeCode(a);
    }

    public void changeCode(ThreeAddressCode a)
    {
        assignList = new List<TACInstruction>();
        foreach (var instr in a.Instructions)
            if (!(instr.Result.Equals("") || instr.Result.Contains("#")))
                assignList.Add(instr);
    }
}

```

Класс InOutVector:

```

public class InOutVector
{
    BitArray data;
    public BitArray Data => data;
    public InOutVector(IEnumerable<TACInstruction> assigns, List<TACInstruction>
assignList)
    {
        data = new BitArray(assignList.Count, false);
        foreach (var assign in assigns)
            data[assignList.IndexOf(assign)] = true;
    }
}

```

6. Интеграция

Векторное представление было интегрировано в алгоритм о достигающих определениях в отдельном файле, с заменой стандартного вычисления IN и OUT на векторные.

7. Тесты

Тестируется равенство стандартных и векторных IN и OUT, а также основные операции класса InOutVector.

```

[Test]
public void Operations()
{
    var array1 = new InOutVector(new BitArray(new[] {true,true,false,true}));
    var array2 = new InOutVector(new BitArray(new[] { false, false, false, true }));
    var array3 = new InOutVector(new BitArray(new[] { true, false, true, false }));
    var array1copy = array1;
    // -a
    Assert.AreEqual((-array1).Data, new BitArray(new[] { false, false, true, false
}));
    // []
    Assert.AreEqual(array1[0], true);
    Assert.AreEqual(array1[2], false);
    // == !=
    Assert.False(array1 == array2);
    Assert.True(array2 != array3);
    Assert.True(array1copy == array1);
    Assert.True(array1copy != array2);
    // + -
    Assert.AreEqual((array1 + array2).Data, new BitArray(new[] { true, true, false,
true }));
    Assert.AreEqual((array2 + array3).Data, new BitArray(new[] { true, false, true,
true }));
    Assert.AreEqual((array1 - array2).Data, new BitArray(new[] { true, true, false,
false }));
    Assert.AreEqual((array3 - array2).Data, new BitArray(new[] { true, false, true,
false }));
}

```

4. Активные переменные

1. Команда, реализующая задачу Османян В., Маслова О.

2. Зависимые и предшествующие задачи

Предшествующие:

- Построение CFG;

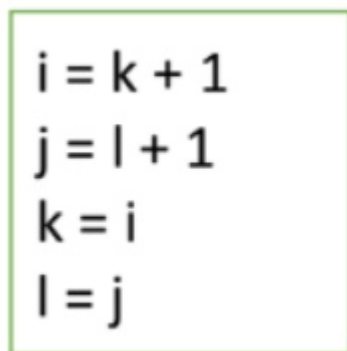
Зависимые: -

3. Аннотация

В данной задаче необходимо при прямом проходе алгоритма определить активные переменные для каждого ББл. Для этой задачи также потребуются определения множеств def и use .

4. Теория

Переменная x активна в точке p , если значение x из точки p может использоваться вдоль некоторого пути. $use[B]$ - множество переменных, значения которых могут использоваться в B до любого их определения. Пример:



```
i = k + 1
j = l + 1
k = i
l = j
```

$$def_B = \{i, j\}$$
$$use_B = \{k, l\}$$

Любая переменная из $use[B]$ - активная на входе в B .

Алгоритм: **Вход:** граф потока управления, в котором для каждого ББл вычислены $def[B]$ и $use[B]$. **Выход:** множества переменных, активных на входе $IN[B]$ и на выходе $OUT[B]$ для всех ББл B .

```
IN[Выход] = ∅;
for (каждый блок B, кроме входного)
    IN[B] = ∅;
while (IN изменяется)
    for (каждый блок B, кроме входного)
        OUT[B] = ∅ IN[S] ; // где S - приемник B
        IN[B] = use[B] ∪ (OUT[B] - def[B]);
```

5. Реализация

Алгоритм итеративно работает на всех блоках. Конструируется множество all_B , простым перебором всех инструкций в блоке. Для каждого элемента из all_B , производятся

следующие манипуляции: Происходит перебор инструкций с конца к началу в обратном порядке. Если не встречено ни одного использования, и встречено определение переменной, то оно добавляется в use_B . Если встречено, определение переменной, то оно добавляется во множество def_B . Функция, отвечающая за сбор usedef информации, является рекурсивной и ее содержимое выглядит следующим образом:

```
if (cur_b != cfg.end)
{
    if (cur_b != cfg.start && !cheked.Contains(cur_b.Index))
    {
        Filling(cur_b);
        cheked.Add(cur_b.Index);
        foreach (var blok in cur_b.Out)
        {
            Find_usedef(blok, cfg, cheked);
        }
    }
}
```

Далее, согласно вышеизложенному алгоритму, циклически производятся вычисления. После остановки вычислений имеем 2 множества In_Akt_Ver(IN[B]) и Out_Akt_Ver(OUT[B]) . Переменные активные на входе и выходе из каждого блока. Основной участок кода для нахождения активных переменных выглядит следующим образом:

```
do
{
    flag = true;
    foreach (var blok in cfg.blocks)
    {
        var in_ = new List<string>();
        foreach (var s in use_B[blok.Index])
            in_.Add(s);
        foreach (var s in Out_Akt_Ver[blok.Index])
            if(!def_B[blok.Index].Contains(s)
                && !in_[blok.Index].Contains(s))
                in_.Add(s);
        foreach (var s in in_)
            flag &= In_Akt_Ver[blok.Index].Contains(s);
        foreach (var s in In_Akt_Ver[blok.Index])
            flag &= in_.Contains(s);
        In_Akt_Ver[blok.Index] = in_;
        var Out_ = new List<string>();
        foreach (var chld in blok.Out)
        {
            foreach (var s in In_Akt_Ver[chld.Index])
                if (!Out_.Contains(s))
                    Out_.Add(s);
        }
        Out_Akt_Ver[blok.Index] = Out_;
    }
}
```

```
}  
while (!flag);
```

6. Интеграция

В процессе сборки проекта объекту класса `ActiveVariableOptimizer`, Подается на вход Граф Потока Управления. По полученному ГПУ производится расчет активных переменных, к которым можно обратиться в других файлах проекта.

7. Тесты

Для тестирования сначала исходный код теста подается на вход парсеру, после чего генерируется трехадресный код. Далее оптимизатору на вход подается список команд трехадресного кода, который оптимизируется при запуске функции `Run()`. Полученный результат сравнивается с предполагаемым ответом. Ниже приведен пример простейшего из вариантов:

```
var TAC = GenerateTAC(  
  @"  
{  
  b = 7;  
  a = b + 1;  
  a = c;  
  b = 22;  
}  
");  
var expected = new List<string>()  
{  
  "b = 7",  
  "a = b + 1"  
};
```

5. Достигающие определения

1. Команда, реализующая задачу Чубинидзе Н. Р., Романченко Р. Д.

2. Зависимые и предшествующие задачи

Предшествующие:

- построение CFG;

Зависимые: -

3. Аннотация

Достигающие определения позволяют определить, является переменная константой в точке `p` или является ли эта переменная неинициализированной в точке `p`.

4. Отрезок теории

Определение. `d` достигает точки `p`, если существует путь от точки, непосредственно следующей за `d`, к точке `p` такой, что `d` не уничтожается вдоль этого пути.



Пример:

Алгоритм: **Вход:** граф потока управления, в котором для каждого ББл вычислены уничтожаемые и генерируемые множества выражений. **Выход:** множества достигающих определений на входе `IN[B]` и на выходе `OUT[B]` для всех ББл `B`.

```
OUT[ВХОД] = {};  
for (каждый блок B, кроме входного)  
    OUT[B] = {};  
while (OUT изменяется)  
    for (каждый блок B, кроме входного)  
        IN[B] = {} OUT[P] // где P - предшественник B  
        OUT[B] = gen[B] U (IN[B] - kill[B]);
```

5. Реализация

Для данной задачи был реализован итерационный алгоритм, который был инкапсулирован в специальный класс `ReachingDefinitionOptimizer`. При вызове метода `Run()` данного оптимизатора, первым делом создаются множества `Gen` и `Kill` для каждого базового блока. Код для создания множества `Gen` приведен ниже:

```
foreach (var instr in graph.blocks[i].Instructions)  
{  
    if (!setGen.Contains(instr.Result)  
        && !(instr.Result.Equals("")  
            || instr.Result.Contains("#")))  
    {  
        setGen.Add(instr.Result);  
        currGen.Add(instr);  
    }  
}  
gen.Add(graph.blocks[i], currGen);
```

Код для создания множества `Kill`:

```
for (int j = 0; j < graph.blocks.Count; ++j)  
{  
    if (j == i)
```

```

        continue;
    foreach (var instr in graph.blocks[j].Instructions)
    {
        if (setGen.Contains(instr.Result))
        {
            currKill.Add(instr);
        }
    }
}
kill.Add(graph.blocks[i], currKill);

```

После этого вычисляются множества In и Out согласно алгоритму, приведенному выше. Основной код, вычисляющий множества In и Out приведен ниже:

```

while (change)
{
    for (int i = 0; i < graph.blocks.Count; ++i)
    {
        change = false;
        var t = graph.blocks[i].In.SelectMany(n => OUT[n]);
        IN[graph.blocks[i]] = new HashSet<TACInstruction>(t);
        var prevOut = OUT[graph.blocks[i]];
        OUT[graph.blocks[i]] = new HashSet<TACInstruction>(gen[graph.blocks[i]]);
        OUT[graph.blocks[i]]
            .UnionWith(IN[graph.blocks[i]]
                .Except(kill[graph.blocks[i]]));
        if (!prevOut.SetEquals(OUT[graph.blocks[i]]))
            change = true;
    }
}

```

6. Интеграция

Поскольку оптимизатор достигающих определений представляет собой обособленный класс, для интеграции в общий проект необходимо просто добавить его создание и вызов процедуры Run() после конвейера оптимизаций по трехадресному коду.

7. Тесты

Для тестирования сначала исходный код теста подается на вход парсеру, после чего генерируется трехадресный код и строится граф потока управления программы. С помощью графа потока управления определяются достигающие определения. Полученные результаты затем сравниваются с предполагаемым ответом. Пример теста представлен ниже:

```

[Test]
public void OneBlockTest1()
{
    var TAC = GenerateTAC(
@"
{
    x = z + y;
    u = x;

```

```

t = z + y;
}
");
var blocks = new TACBaseBlocks(TAC.Instructions);
blocks.GenBaseBlocks();
var cfg = new ControlFlowGraph(blocks.blocks);
var optimizer = new ReachingDefinitionOptimizer(cfg);
optimizer.Run();
Assert.AreEqual(optimizer.IN[cfg.start].Count, 0);
Assert.AreEqual(optimizer.IN[cfg.blocks[0]].Count, 0);
Assert.AreEqual(optimizer.OUT[cfg.start].Count, 0);
Assert.AreEqual(optimizer.OUT[cfg.blocks[0]].Count, 3);
var expected = new List<string>() {
    "t = #t1",
    "u = x",
    "x = #t0"
};
var actual = optimizer.OUT[cfg.blocks[0]]
    .Select(x => x.ToString().Trim()).ToList();
Assert.AreEqual(actual, expected);
}

```

6. Передаточная функция в структуре распространения констант

1. Команда, реализующая задачу Гарьковенко А., Руднев. Д.

2. Зависимые и предшествующие задачи

Предшествующие:

- разбиение на базовые блоки;
- построение Control Flow Graph;
- итерационный алгоритм.

Зависимые: - .

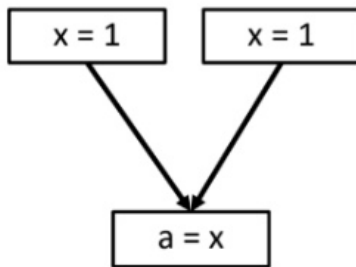
3. Аннотация

Для выполнения данной задачи требуется описать передаточную функцию в структуре распространения констант.

4. Теория

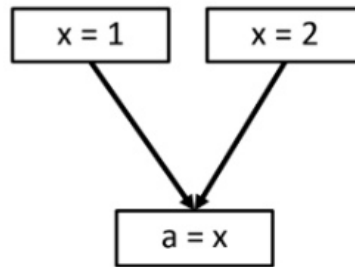
Замена константы может быть произведена только в том случае, когда ни по какому из путей не происходит переопределение.

Пример 1.



Можно

Пример 2.

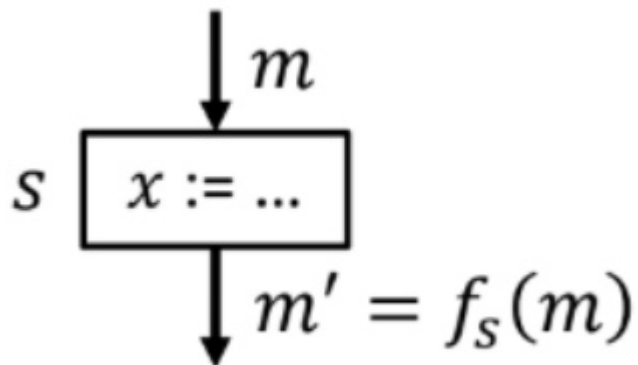


Нельзя

Для одной переменной x_i – значения в полурешётке V_i :

- все константы данного типа
- NAC (Not A Constant) – либо переменной было присвоена не константа, либо по разным веткам – разные константы
- UNDEF (неизвестно пока, является ли константой)

Рассмотрим передаточную функцию одной команды:



- Если s – не присваивание, то f_s – тождественная: $f_s(m) = m$
- Если $s: x := \dots$, то $\forall v \neq x \ m'(v) = m(v)$, а $m'(x)$ определяется так:
 - если $x := c$, то $m'(x) = c$
 - если $x := y + z$, то

$$m'(x) = \begin{cases} m(y) + m(z), & \text{если } m(y) + m(z) = \text{const} \\ \text{NAC}, & \text{если } m(y) = \text{NAC} \text{ или } m(z) = \text{NAC} \\ \text{UNDEF} & \text{в остальных случаях} \end{cases}$$

- если $x := g(\dots)$, то $m'(x) = \text{NAC}$ (консервативно)

5. Реализация

За передаточную функцию в структуре распространения констант отвечает следующий участок кода:

```
OUT[instrs[i].Result] = OUT[first].Type == SemilatticeData.UNDEF
    ? new SemilatticeValue(SemilatticeData.UNDEF)
    : OUT[first].Type == SemilatticeData.NAC
    || OUT[second].Type == SemilatticeData.NAC
    ? new SemilatticeValue(SemilatticeData.NAC)
    : new SemilatticeValue(SemilatticeData.UNDEF);
```

6. Интеграция в общий проект

Данная функция используется в итерационном алгоритме распространения констант.

7. Тесты

Тестирование функции проходило вместе с тестированием самого алгоритма. Пример такого теста представлен ниже:

```
[Test]
public void Simple()
{
    var TAC = GenerateTAC(
@"
{
a = 3;
b = 3;
c = a + b;
a = d;
e = a;
}
");
    var blocks = new TACBaseBlocks(TAC.Instructions);
    blocks.GenBaseBlocks();
    var cfg = new ControlFlowGraph(blocks.blocks);
    var optimizer = new ConstantPropagationIter();
    optimizer.Cfg = cfg;
    optimizer.Instructions = TAC.Instructions;
    optimizer.Blocks = blocks.blocks;
    optimizer.Run();
    var actual = optimizer.Instructions.Select(i => i.ToString().Trim()).ToList();

    var expected = new List<string>()
    {
        "a = 3",
        "b = 3",
        "c = 3 + 3",
        "a = d",
        "e = a"
    };
};
```

```
Assert.AreEqual(expected, actual);  
}
```

7. Итерационный алгоритм в обобщённой структуре

1. Команда, реализующая задачу Манукян Г. А.

2. Зависимые и предшествующие задачи

Предшествующая:

- Построение CFG.
- Обход потомков и обход предков для каждого ББл

Зависимые:

- Вычисление передаточной функции для достигающих определений композицией передаточных функций команд
- Передаточная функция в структуре распространения констант
- Итерационный алгоритм в структуре распространения констант

3. Аннотация

В рамках этой задачи реализован обобщенный итерационный алгоритм. Необходим он для связки задач:

- Проверки CFG на неприводимость
- Поиска неиспользуемого кода, поиска в циклах переходов goto (not natural loops)
- Цикловых оптимизаций в CFG
- Использование и определение переменных в заданной точке программы (дост. опр-я)
- Анализ указателей (алиасов)

4. Теория

Входы итерационного алгоритма:

- Граф потока данных с помеченными входными и выходными узлами
- Направление потока данных
- Множество значений V
- Оператор сбора \square
- Множество функций f где $f(b)$ из F представляет собой передаточную функцию для блока b
- Константное значение v вход или v выход из V , представляющее собой граничное условие для прямой и обратной структуры соответственно.

Выходы итерационного алгоритма:

- Значения из V для $IN(b)$ и $OUT(b)$ для каждого блока b в CFG

В задаче на передаточную функцию для достигающих определений команд – алгоритм находит IN и OUT множества для каждого блока как ряд последовательных приближений.

Служит для избежания базового итеративного алгоритма для каждой структуры потока данных используемой на стадии оптимизации и используется в прямой и обратной задачах потока данных (Data Flow Optimizations in relation to CFG).

Его использование предоставляет ряд полезных свойств, собственно, на примере решения задач команд:

- Гарьковенко + Руднев {итерационный алгоритм для задачи распространения констант}
- Манукян {Вычисление передаточной функции для достигающих определений композицией передаточных функций команд}.

Итерационный алгоритм для задачи распространения констант

```
Out[Вход] = (UNDEF, UNDEF, ..., UNDEF) =  $\top$ 
foreach  $B \in \text{ББл}, B \neq \text{Вход}$ 
    Out[B] = (UNDEF, UNDEF, ..., UNDEF) =  $\top$ 
while (внесены изменения в Out)
{
    foreach  $B \in \text{ББл}, B \neq \text{Вход}$ 
    {
         $IN[B] = \bigwedge_{P \rightarrow B} OUT[P]$ 
         $OUT[B] = f_B(IN[B])$ 
    }
}
```

Так как передаточная функция монотонна и множество V с операцией \wedge является полурешёткой, то итерационный алгоритм сходится.

Итеративный алгоритм для достигающих определений

Вход: граф потока управления, в котором для каждого ББл вычислены gen_B и $kill_B$

Выход: Множества достигающих определений на входе $IN[B]$ и на выходе $OUT[B]$ для каждого ББл B

```
OUT[Вход] = ∅;  
for (каждый базовый блок  $B$ , отличный от входного)  $OUT[B] = ∅$ ;  
while (внесены изменения в  $OUT$ )  
    for (каждый базовый блок  $B$ , отличный от входного) {  
         $IN[B] = \bigcup_{P-\text{предшественник } B} OUT[P]$ ;  
         $OUT[B] = gen_B \cup (IN[B] - kill_B)$ ;  
    }
```

Сходимость алгоритма: на каждом шаге $IN[B]$ и $OUT[B]$ не уменьшаются для всех B и ограничены сверху, поэтому алгоритм сходится.

Максимальное время сходимости: за каждую итерацию внутреннего цикла вносится изменение в один $OUT[B] - (\text{кво ББл}) * (\text{кво определений})$

5. Реализация

Класс, необходимый для генерации OUT – множества выходных данных:

```
public class InOutData<T> : Dictionary<BasicBlock, (T In, T Out)>  
    where T : IEnumerable  
{  
    public override string ToString()  
    {  
        <...>  
    }  
    public InOutData() { }  
    public InOutData(Dictionary<BasicBlock, (T, T)> dictionary)  
    {  
    }  
}
```

Тип прохода задается перечислением `directed`, он должен быть определен ещё до запуска алгоритма:

```
public enum directed { forward, back }
```

Основной класс реализует интерфейс `IEnumerable`, для совместимости с `HashSet<>`, `ValueTuple`2`, `List<>`.

`GetInitData` - функция инициализации данных относительно вида прохода алгоритма, в случае обратного прохода берутся блоки с конца, в обратном случае с начала, проводится инициализация первого элемента полурешетки, свойств для каждого типа прохода, а также каждого типа полурешетки:

```

public abstract class IterAlgoGeneric<T> : TACOoptimizer where T : IEnumerable
{
    public abstract Func<T, T, T> CollectingOperator { get; }
    public abstract Func<T, T, bool> Compare { get; }
    public abstract T Init { get; protected set; }
    public virtual T InitFirst { get => Init; protected set { } }
    public abstract Func<BasicBlock, T, T> TransferFunction { get; protected set; }
    public virtual directed directed => directed.forward;
    public virtual InOutData<T> Execute(ControlFlowGraph graph)
    {
        GetInitData(graph, out var blocks, out var data,
            out var InitBlocks, out var InitVals, out var combine);
        var outChanged = true;
        while (outChanged)
        {
            outChanged = false;
            foreach (var block in blocks) // основной цикл итерационного алгоритма
            {
                var inset = InitBlocks(block).Aggregate(Init, (x, y) =>
                    CollectingOperator(x, InitVals(y)));
                var outset = TransferFunction(block, inset);
                if (!Compare(outset, InitVals(block)))
                    outChanged = true;
                data[block] = combine(inset, outset);
            }
        }
        return data;
    }
}

```

6. Интеграция в общий проект

Для нужд команды, реализовавшей решение итерационного алгоритма для распространения констант, унаследованный класс был протестирован, все тесты были пройдены успешно. В единой структуре вызываются итерационные алгоритмы для различных типов передаточных функций (в классе задаются методом-параметром).

7. Тесты

В тестах проверялось использование итерационных алгоритмов в обобщенной структуре, результаты – 2 из 2 тестов. Ниже приведен один из этих тестов:

```

public void SampleClassIterAlgoForTransferFunc()
{
    var TAC = GenerateTAC(@"
{
    int a,b,c;
    b = 10;
    a = b + 1;
    if a < c
    {
        c = b - a;
    } else

```

```

{
c = b + a;
}
write(c);
}");
var TACBlocks = new TACBaseBlocks(TAC.Instructions);
var cfg = new ControlFlowGraph(TACBlocks.blocks);
var TransferFunc = new SampleClassIterAlgoForTransferFunc();
var resultTransferFunc = TransferFunc.Execute(cfg);
var In = new HashSet<string>();
var Out = new HashSet<string>();
var actual = new List<(HashSet<string> IN, HashSet<string> OUT)>();
foreach (var x in cfg.blocks.Select(z => resultTransferFunc[z]))
{
    foreach (var y in x.In)
    {
        In.Add(y.ToString());
    }
    foreach (var y in x.Out)
    {
        Out.Add(y.ToString());
    }
    actual.Add((new HashSet<string>(In), new HashSet<string>(Out)));
    In.Clear(); Out.Clear();
}
var expected = new List<(HashSet<string> IN, HashSet<string> OUT)>()
{
    (new HashSet<string>(){ "c" }, new HashSet<string>(){ "c" }),
    (new HashSet<string>(){ "c" }, new HashSet<string>(){ "a", "b" }),
    (new HashSet<string>(){ "a", "b" }, new HashSet<string>(){ "c" }),
    (new HashSet<string>(){ "a", "b" }, new HashSet<string>(){ "c" }),
    (new HashSet<string>(){ "c" }, new HashSet<string>(){ }),
    (new HashSet<string>(){ }, new HashSet<string>(){ })
};
AssertSet(expected, actual);
}

```

Циклы в графах потоков управления

Пока мы не учитывали наличие циклов в CFG. Циклы дают потенциально бесконечное число путей в программах.

Для этого необходимо реализовать следующее:

№	Задание
1	Построение дерева доминаторов - итерационный алгоритм для определения D(B) + непосредственные доминаторы
2	Алгоритм упорядочения в глубину с построением глубинного остовного дерева
3	Классификация ребер графа: наступающие, отступающие, поперечные (по построенному остовному дереву)

4	Обратные ребра и определение того, что CFG является приводимым
5	Определение всех естественных циклов

1. Построение дерева доминаторов

1. Команда, реализующая задачу Гарьковенко А., Руднев. Д.

2. Зависимые и предшествующие задачи

Предшествующие:

- разбиение на базовые блоки;
- построение Control Flow Graph;
- Итерационный алгоритм

Зависимые:

- определение натуральных циклов

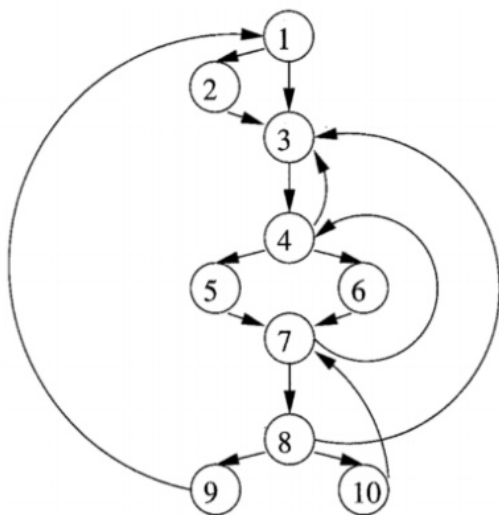
3. Аннотация

В данной задаче необходимо построить дерево доминаторов.

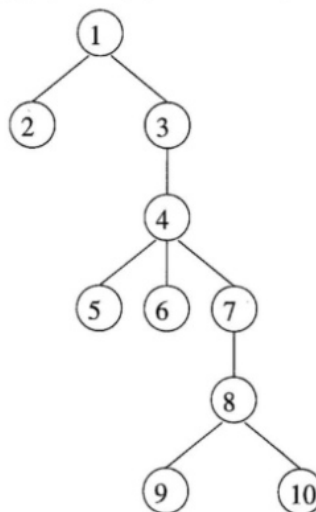
4. Теория

Пусть d, n - вершины CFG. Вершина d доминирует на n , если любой путь от входного узла к n проходит через d .

Пример. Граф потока управления



Дерево доминаторов



Множество доминаторов узла (кроме него самого) – это пересечение доминаторов всех его предшественников.

Алгоритм:

Вход: граф потока управления. **Выход:** $OUT[V]$ - множество доминаторов над V .

```

OUT[ВХОД] = {ВХОД};
for (каждый блок В, кроме входного)
    OUT[В] = U;
while (OUT изменяется)
    for (каждый блок В, кроме входного)
        IN[В] =  $\cap$  OUT[P]          // где P - предшественник В
        OUT[В] = IN[В]  $\cup$  {В};

```

5. Реализация

Для хранения дерева доминаторов, был использован словарь, ключом которого являлся определенный базовый блок, а значением - список всех его доминаторов. Сначала полагаем, что все блоки доминируют над всеми:

```

for (int i = 1; i < controlFlow.BlockCount; i++)
    dominators.Add(controlFlow.blocks[i], controlFlow.blocks.ToList());

```

Затем удаляем неподходящие блоки, как представлено ниже:

```

for (int i = 1; i < controlFlow.BlockCount; i++){
    previos = dominators[controlFlow.blocks[i]];
    //пересечение блоков доминаторов предшественников
    if (controlFlow.blocks[i].In.Count == 1)
        temp = dominators[controlFlow.blocks[i].In[0]].ToList();
    else
        for (int j = 0; j < controlFlow.blocks[i].In.Count - 1; j++)
        {
            var intersect = dominators[controlFlow.blocks[i].In[j]]
                .Intersect(dominators[controlFlow.blocks[i].In[j + 1]]).ToList();
            if (temp.Count != 0)
                temp = temp.Intersect(intersect).ToList();
            else
                temp = intersect;
        }
    //сам блок
    temp.Add(controlFlow.blocks[i]);
    dominators[controlFlow.blocks[i]] = temp.ToList();
    if (dominators[controlFlow.blocks[i]].SequenceEqual(previos))
        changedCount++;
}

```

6. Интеграция в общий проект

Т.к. публичная функция для нахождения всех доминаторов помещена в класс DominatorsTree, ею можно воспользоваться из любого места компилятора.

7. Тесты

При тестировании исходный текст программы подавался на вход парсеру, после чего генерировался трехадресный код, создавался граф потока управления и дерево доминаторов. После чего, доминаторы, полученные в ходе работы программы и доминаторы-образчики, созданные вручную, сравнивались на равенство. Ниже приведен пример теста

```

[Test]
public void Test1()
{
    var cfg = GenerateCFG(
@"{
if (true)
{
a = 5;
}
else
{
b = 1;
}
}");

    /*
     * 4 блока
     * 0 {if True goto #L0}
     * 1 {b = 1; goto #L1}
     * 2 {#L0 a = 5;}
     * 3 {#L1}
     */

    var dominatorsTree = new DominatorsTree(cfg);
    dominatorsTree.GenDominatorsTree();
    var Actual = dominatorsTree.dominators;

    var Expect = new Dictionary<BasicBlock, List<BasicBlock>>();
    Expect.Add(cfg.blocks[0], new List<BasicBlock>(){cfg.blocks[0]});
    Expect.Add(cfg.blocks[1], new List<BasicBlock>() { cfg.blocks[0], cfg.blocks[1]
});
    Expect.Add(cfg.blocks[2], new List<BasicBlock>() { cfg.blocks[0], cfg.blocks[2]
});
    Expect.Add(cfg.blocks[3], new List<BasicBlock>() { cfg.blocks[0], cfg.blocks[3]
});
    CollectionAssert.AreEqual(Actual, Expect);
}

```

2. Алгоритм упорядочения в глубину с построением глубинного остовного дерева.

1. Команда, реализующая задачу Османян В. А., Маслова О. В.

2. Зависимые и предшествующие задачи

Предшествующие:

- построение CFG;

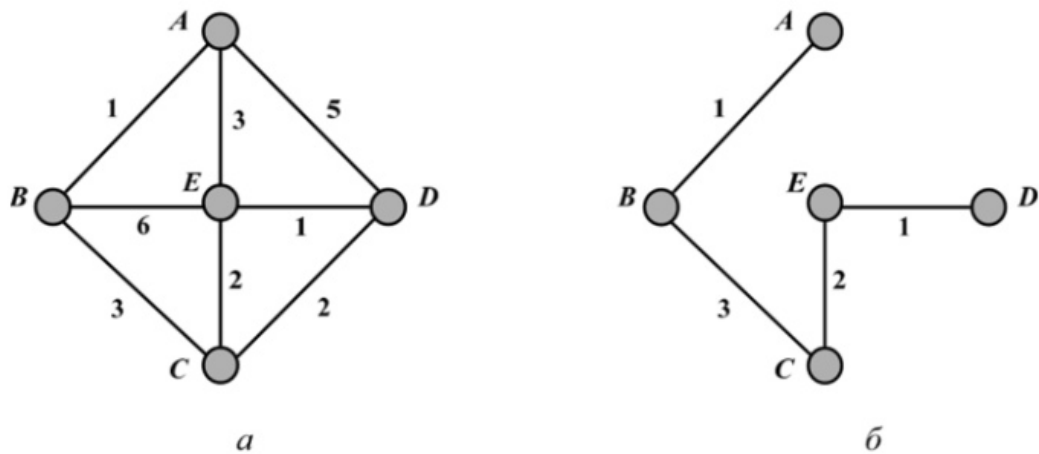
Зависимые: -

3. Аннотация

В данной задаче необходимо написать алгоритм упорядочения в глубину и, используя его, написать алгоритм построения глубинного остовного дерева.

4. Теория

Остовное дерево графа – это дерево, подграф данного графа, с тем же числом вершин, что и у исходного графа. Остовное дерево получается из исходного графа удалением максимального числа рёбер, входящих в циклы, но без нарушения связности графа. Пример:



б - остовное дерево, построенное для графа а

Обход в глубину - один из основных методов обхода графа, используемый в данной задаче для проверки связности и поиска цикла. Общая идея алгоритма состоит в следующем: для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них.

5. Реализация

Первым делом для построения остовного дерева нужно было написать алгоритм упорядочения, как указано в задании, в глубину. Ниже можно увидеть участок кода, отвечающий за этот алгоритм, где dfst отвечает за список ребер остовного дерева:

```
int Traverse(BasicBlock n,int c)
{
    p[n.Index] = true;
    foreach(BasicBlock s in n.Out)
        if(!p[s.Index])
        {
            dfst.Add(new IndexEdge(n.Index, s.Index));
            c=Traverse(s,c);
        }
    dfn[n.Index] = c--;
    return c;
}
```

Затем построение остовного дерева осуществлялось при помощи функции Block_Cheking(cur_b, cfg, cheked), которая отвечает за подсчет количества блоков CFG. Ниже приведен участок кода, отвечающий за построение необходимого дерева:

```

public SpanningTree(ControlFlowGraph cfg)
{
    var cur_b = cfg.start;
    List<int> cheked = new List<int>();
    c = Block_Cheking(cur_b, cfg, cheked);
    foreach (var block in cfg.blocks)
        p[block.Index] = false;
    p[cfg.end.Index] = false;
    Traverse(cur_b, c);
}

```

6. Интеграция в общий проект

Т.к. список ребер остоного дерева является публичным полем нестатического класса SpanningTree, то для того, чтобы получить к ним доступ, необходимо создать экземпляр класса SpanningTree и обратиться к его полю dfst.

7. Тесты

При тестировании исходный текст программы подавался на вход генератору CFG. После этого по полученному графу строилось остоное дерево. Далее ребра полученного дерева сравнивались с предполагаемым ответом. Ниже приведен пример теста без циклов в исходном коде:

```

[Test]
public void Simple()
{
    BasicBlock.clearIndexCounter();
    var cfg = GenerateCFG(
@"
{
a = 3;
b = 2;
c = a + b;
}
");

    var spanningTree = new SpanningTree(cfg);
    var actual = spanningTree.dfst;
    var expected = new List<IndexEdge>()
    {
        new IndexEdge(1, 0),
        new IndexEdge(0, 2)
    };
    Assert.AreEqual(expected, actual);
}

```

3. Классификация ребер в CFG: наступающие, отступающие, поперечные (по остоному дереву)

1. Команда, реализующая задачу Чухин А. И., Агафонцев Р. Г.
2. Зависимые и предшествующие задачи

Предшествующие:

- Алгоритм упорядочения в глубину
- Построение глубинного остовного дерева
- Построение CFG

Зависимые: -

3. Аннотация

Дан ControlFlowGraph. Все его рёбра необходимо классифицировать на три группы:

- наступающие (advancing) рёбра идут от узла к его истинному потомку
- отступающие (retreating) рёбра идут от узла к его предку
- поперечные (cross) - все остальные рёбра.

4. Теория

Рёбра графа при построении глубинного остовного дерева

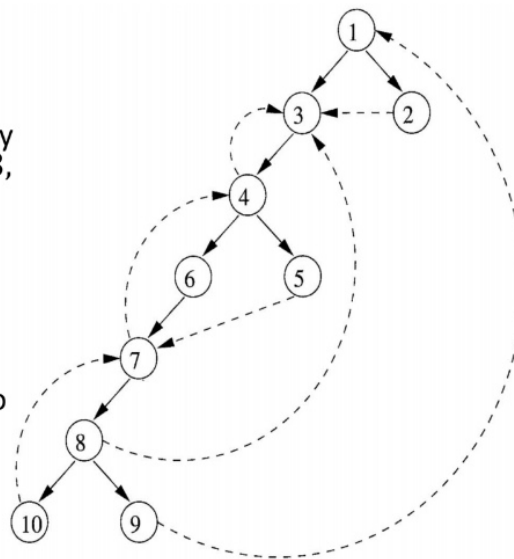
Наступающие (advancing) – от m к потомкам в дереве (все рёбра глубинного остовного дерева + возможно другие – например, $4 \rightarrow 8$)

Отступающие (retreating) – от m к предку в дереве (возможно, к самому m): $4 \rightarrow 3$, $7 \rightarrow 4$, $10 \rightarrow 7$, $8 \rightarrow 3$, $9 \rightarrow 1$

Поперечные (cross) – рёбра $m \rightarrow n$, такие что ни m ни n не являются предками друг друга: $2 \rightarrow 3$, $5 \rightarrow 7$

Свойство: если $m \rightarrow n$ – отступающее, то $dfn[m] \geq dfn[n]$.

Доказательство. По алгоритму $Traverse(m)$ завершается раньше $Traverse(n)$



5. Реализация

Реализуем перечисление EdgeType для хранения типа ребра. Далее с помощью функции-расширения classifyEdges получаем словарь, где ключами выступают грани, а значениями – их типы EdgeType.

```
public enum EdgeType
{
    None = 0,
    Advancing = 1,
```

```

    Retreating = 2,
    Cross = 3
}

public static class CFGExtension
{
    static bool isPathExists(int start, int end, IEnumerable<IndexEdge>
ostovTreeEdges)
    {
        var visitedChildren = new HashSet<int>();
        var children = new HashSet<int>();
        children.Add(start);
        while (children.Count != 0)
        {
            visitedChildren.UnionWith(children);

            var oldChildren = new HashSet<int>(children);
            foreach (var child in oldChildren)
                children.UnionWith(ostovTreeEdges.Where(e => e.start == child
                                                                || e.end == child)
                                                                .Select(e => e.start == child? e.end : e.start));

            if (children.Contains(end))
                return true;

            children.ExceptWith(visitedChildren);
        }

        return false;
    }

    public static Dictionary<IndexEdge, EdgeType> classifyEdges(this ControlFlowGraph
c)
    {
        var types = new Dictionary<IndexEdge, EdgeType>();
        var ostovTree = new Ostov_Tree(c);

        var edges = new HashSet<IndexEdge>();
        edges.Add(new IndexEdge(c.start.Index, c.blocks[0].Index));
        for (int i = 0; i < c.blocks.Count; i++)
        {
            edges.UnionWith(c.blocks[i].Out.Select(b =>
                new IndexEdge(c.blocks[i].Index, b.Index)));
        }

        foreach (var edge in edges)
        {
            if (!ostovTree.dfn.ContainsKey(edge.start)
                || !ostovTree.dfn.ContainsKey(edge.end))
                types[edge] = EdgeType.Cross;
            else if (ostovTree.dfst.Any(e => e.start == edge.start && e.end ==
edge.end))

```

```

        || (ostovTree.dfn[edge.start] < ostovTree.dfn[edge.end]
        && isPathExists(edge.start, edge.end, ostovTree.dfst)))
    {
        types[edge] = EdgeType.Advancing;
    }
    else if (ostovTree.dfn[edge.start] >= ostovTree.dfn[edge.end]
    && isPathExists(edge.start, edge.end, ostovTree.dfst))
    {
        types[edge] = EdgeType.Retreating;
    }
    else
    {
        types[edge] = EdgeType.Cross;
    }
}
return types;
}
}

```

6. Интеграция в общий проект

Классификация ребер происходит в соответствии с реализацией построения остова дерева, в частности ключами получаемого словаря выступают пары индексов базовых блоков, так же хранятся ребра в остова дерева

7. Тесты

Программа

```

@"{
if (true)
{
a = 5;
}
else
{
b = 1;
}
2:
goto 2;
}"

```

Блоки (пара индекс-блок)

```

@"
0-{if True goto #L0}
1-{b = 1; goto #L1}
2-{#L0 a = 5;}
3-{#L1}
4-{2: goto 2}
"

```

Результат

```

0->1 - Advancing

```

```
0->2 - Advancing
1->3 - Advancing
2->3 - Advancing
3->4 - Advancing
4->4 - Retreating
```

4. Обратные ребра и определение того, что CFG является приводимым

1. Команда, реализующая задачу Чубинидзе Н. Р., Романченко Р. Д.

2. Зависимые и предшествующие задачи

Предшествующие:

- построение CFG.
- Вычисление доминаторов
- Классификация ребер графа

Зависимые:

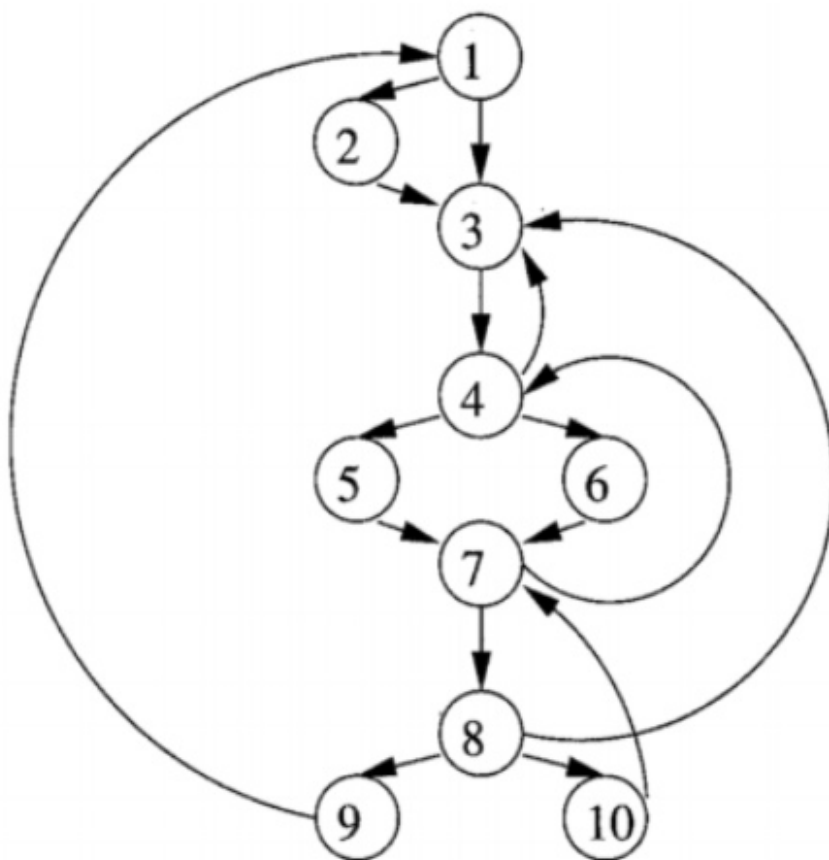
- определение всех естественных классов.

3. Аннотация

В данной задаче необходимо было определить обратные ребра и является ли CFG приводимым.

4. Теория

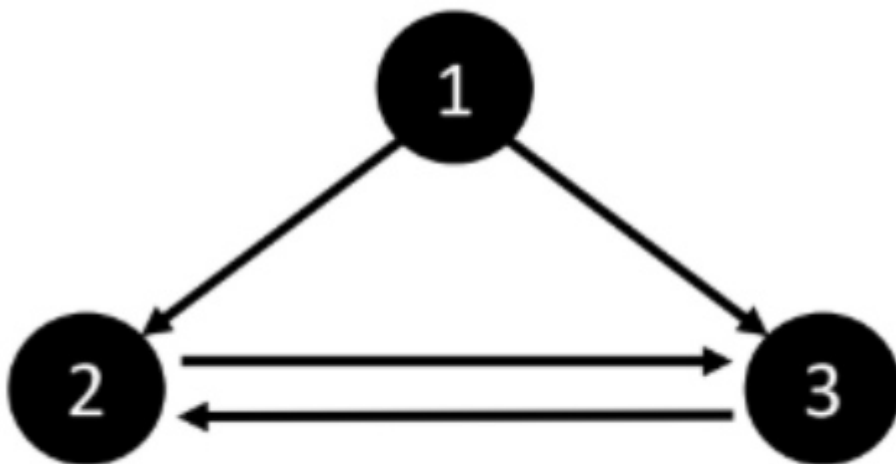
Обратное ребро - это в CFG такое ребро $a \rightarrow d$, у которого d доминирует над a . При этом любое обратное ребро является отступающим, но не всегда верно обратное. Пример:



Обратные ребра: $4 \rightarrow 3$, $7 \rightarrow 4$, $8 \rightarrow 3$, $9 \rightarrow 1$, $10 \rightarrow 7$

Граф потока управления приводим, если все его отступающие ребра являются обратными. Приведенный выше граф является примером приводимого графа. Ниже показан пример

неприводимого графа. Пример:



5. Реализация

Для данной задачи был реализован статический класс TACUtils, который инкапсулирует статический метод GetBackEdges(CFG, Dominators). Для нахождения обратных ребер был использован алгоритм обхода в ширину.

```
var result = new List<BasicBlock, BasicBlock>();
var queue = new Queue<BasicBlock>();
var used = new HashSet<BasicBlock, BasicBlock>();
queue.Enqueue(cfg.start);
while (queue.Count != 0) //просто обход в ширину
{
    var cur = queue.Peek();
    queue.Dequeue();
    foreach (var edgeEnd in cur.Out)
    {
        if (edgeEnd == cfg.start || edgeEnd == cfg.end)
        {
            continue;
        }

        if (!used.Contains((cur, edgeEnd)))
        {
            if (dominators.ContainsKey(cur) && dominators[cur].Contains(edgeEnd)
                //обратное ребро)
            {
                result.Add((cur, edgeEnd));
            }
            used.Add((cur, edgeEnd));
            queue.Enqueue(edgeEnd);
        }
    }
}
```

```
    }  
    return result;
```

Также имеется функция, которая возвращает булево значение - Является ли граф приводимым?

```
var edgesType = CFGExtension.classifyEdges(cfg);  
foreach (var x in edgesType)  
{  
    if (x.Value.Equals(EdgeType.Retreating))  
    {  
        bool flag = false;  
        foreach (var b in backEdges)  
        {  
            if (x.Key.Equals(new IndexEdge(b.Item1.Index, b.Item2.Index)))  
                flag = true;  
        }  
        if (!flag)  
            return false;  
    }  
}  
return true;
```

6. Интеграция в общий проект

Для нахождения обратных ребер используется ранее построенный граф CFG, а так же дерево доминаторов. При проверке приводимости графа используется классификация ребер. В дальнейшем нахождение обратных ребер графа необходимо для определения всех естественных циклов. 7. Тесты

При тестировании исходный текст программы подавался на вход парсеру, после чего генерировался трехадресный код, создавался граф потока управления и дерево доминаторов. После этого происходила проверка графа на приводимость, Ниже приведен пример теста с приводимым графом :

```
[Test]  
public void SimpleTest2()  
{  
    BasicBlock.clearIndexCounter();  
    var cfg = GenerateCFG(  
@"{  
a = 5;  
1:  
b = c;  
2:  
goto 2;  
goto 1;  
d = a;  
goto 1;  
}");  
/*
```

```

    * 5 блоков
    0 {a = 5}
    1 {1: b = c}
    2 {2: goto 2}
    3 {goto 1}
    4 {d = a; goto 1}
    */

    var dominators = (new DominatorsTree(cfg)).GenDominatorsTree();
    Assert.IsTrue(ReducibleGraph.IsReducible(cfg, dominators));
}

```

5. Определение всех естественных циклов

1. Команда, реализующая задачу Погорелов А. А., Домбровская А. В.

2. Зависимые и предшествующие задачи

Предшествующие:

- построение CFG;
- обратные ребра;
- определение того, что граф приводим.

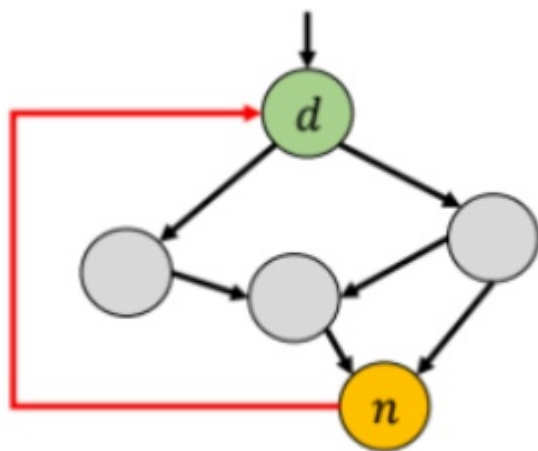
Зависимые: -

3. Аннотация

Естественные циклы позволяют определить, возможность применения простых оптимизаций.

4. Теория

Пусть граф потока управления приводим, тогда для данного обратного ребра $n \rightarrow d$ определим естественный цикл ребра как d плюс множество узлов, которые могут достичь n не проходя через d . Узел d - заголовок цикла. Естественные циклы либо не пересекаются либо один из них вложен в другой. Пример естественного цикла:



Алгоритм:

```
loopSet := {n, d};
used[d] = true;
used = BFS(G, n);    // где G - обратный CFG,
// n - вершина, с которой начинается поиск.
// used - все посещенные вершины
loopSet.Add(used);
```

5. Реализация

Первым этапом для определения естественных циклов необходимо найти все обратные дуги в графе потока управления. Алгоритм нахождения обратных дуг был вынесен в отдельную функцию статического класса TACUtils, которая просматривает все дуги графа и с помощью дерева доминаторов определяет, является ли дуга обратной. После нахождения всех обратных дуг, для каждой из них отработывает алгоритм нахождения естественного цикла C для обратной дуги (v1, v2). Для этого используется простой обход обратного графа потока управления в ширину. Ниже можно увидеть участок кода, отвечающий за обход в ширину:

```
while (queue.Count != 0) //просто обход в ширину
{
    var cur = queue.Peek();
    queue.Dequeue();
    result.Blocks.Add(cur);
    foreach (var edgeStart in cur.In)
    {
        if (edgeStart == cfg.start || edgeStart == cfg.end)
        {
            continue;
        }

        if (!used.Contains(edgeStart))
        {
            used.Add(edgeStart);
            queue.Enqueue(edgeStart);
        }
    }
}
```

6. Интеграция в общий проект

Т.к. публичная функция для нахождения всех естественных циклов помещена в статический класс TACUtils, ею можно воспользоваться из любого места компилятора.

7. Тесты

При тестировании исходный текст программы подавался на вход парсеру, после чего генерировался трехадресный код, создавался граф потока управления и дерево доминаторов. После этого вызывалась функция TACUtils.GetLoops, для нахождения всех циклов в тестируемом коде. Как образчик использовались созданные вручную экземпляры класса NaturalLoop. После чего, циклы, полученные в ходе работы программы и циклы-

образчики, созданные вручную, сравнивались на равенство. Ниже приведен пример теста с одним естественным циклом в исходном коде:

```
var actual = GetLoops(
@"
{
z = 0;
1: x = 0;
y = 1;
goto 2;
2: x = 1;
goto 1;
y = 123;
}
");
var loop1 = GetLoopFromBlocksCode(
@"
{
1: x = 0;
y = 1;
goto 2;
}
",
@"
{
2: x = 1;
goto 1;
}
");
var expected = new List<NaturalLoop>()
{
    loop1
};
Assert.IsTrue(LoopsAreEqual(actual, expected), "Loops are not equal");
```
