Contents

Введение

Задача заключалась в разработке *оптимизирующего* компилятора для императивного языка программирования.

В качестве базового языка использовался C#, генератор синтаксического анализатора Yacc и генератор лексического анализатора Lex.

Структура проекта

Проект состоял из нескольких модулей.

1. Парсер и синтаксический анализатор – строит AST (синтаксическое дерево) по исходному коду программы.
2. Генератор трехадресного кода – превращает AST в последовательность команд трехадресного кода
3. Модуль для построения базовых блоков и CFG
4. Модуль, совершающих анализ и преобразование CFG
5. Модуль, реализующий набор итерационных алгоритмов
6. Модуль анализирующий циклы в CFG
7. Юнит тесты и примеры использования.

Описание синтаксиса

Переменные описываются первым присваиванием значения. Пр.: a = 5

; - завершает описание.

Цикл for: **for** b = 1..10 {}

Цикл while: **while** a < b {}

Оператор if: **if** a<b {}

Логические опирации: and, or, not, ==, !=, >, <

Вторая версия синтаксиса

используются endif, endfor, endwhile вместо {}

Генерация трехадресного кода

Интерфейс команды в трехадресном коде

```
public interface IThreeAddressCode
{
    Operation Operation { get; set; }
    IValue LeftOperand { get; set; }
    IValue RightOperand { get; set; }
    StringValue Destination { get; set; }
    LabelValue Label { get; set; }
```

```
}
```

- Operation - бинарная операция или процедура (Print/Println)
- LeftOperand:
- Левый операнд бинарной операции
- Присваиваемое значение в случае операции Assign
- RightOperand
- Правый операнд бинарной операции
- Destination
- Переменная, которой присваивается значения
- Команда, к которой будет осуществлён переход
- Label - метка

Генерация трехадресного кода осуществляется с помощью обхода LinearCodeVisitor

Линейный код

```
a = b + (c + 1);
```

переводится в

```
%ulabel1: $const0 := c + 1
%ulabel0: a := b + $const0
```

Условный оператор

```
if условие<>
{
  oper1;
}
else
{
  oper2;
}
```

переводится в

```
%ulabel0: IF условие<> THEN GOTO %label0
%ulabel1: oper1
%ulabel2: GOTO %label1
%label0: NOP
%ulabel3: oper2
%label1: NOP
```

Циклы

```
for i = 1..n {
  a = 1;
}
```

переводится в

```
%ulabel0: i := 1
%label0: NOP
%ulabel3: $const0 := i <= n
%ulabel4: IF $const0 THEN GOTO %label1
%ulabel5: GOTO %label2
%label1: NOP
%ulabel6: a := 1
%ulabel1: i := i + 1
%ulabel2: GOTO %label0
%label2: NOP
```

Разбиение на базовые блоки

Общие сведения

Определение *Базовый блок* – это последовательность инструкций трехадресного кода, имеющая одну точку входа (первая инструкция) и одну точку выхода (последняя инструкция).

Базовый блок реализует интерфейс IBaseBlock

```
public interface IBaseBlock
{
  bool InsertAfter(IThreeAddressCode after, IThreeAddressCode newElem);
  void Append(IThreeAddressCode newElem);
  bool Remove(IThreeAddressCode elem);
  IEnumerable<IThreeAddressCode> Enumerate();
  string ToString();
}
```

В реализации для хранения команд трехадресного кода использовался LinkedList<IThreeAddressCode> для быстрого удаления(/вставки) из(/на) произвольной позиции в середине списка.

Алгоритм построения

Для построения базовых блоков из необходимо выделить команды-лидеры.

Определение *Команда-лидер* – это: 1. Первая команда 2. Команда, на которую есть переход 3. Команда, следующая за переходом.

Получаем второе определение базового блока:

Определение *Базовый блок* – это последовательность команд от лидера (включая) до лидера (исключая).

Пользуясь этим определением легко реализовать алгоритм разбиения последовательности команд трехадресного кода на базовые блоки.

Для этого первым проходом по последовательности находим команды лидеры:

```
var controlPoints = new HashSet<LabelValue>();
foreach (var lin in lst)
{
   if (forceAdd)
      controlPoints.Add(lin.Label);
   forceAdd = false;

   var op = lin.Operation;

   if (op == Operation.Goto || op == Operation.CondGoto)
   {
      forceAdd = true;
      controlPoints.Add(lin.Destination as LabelValue);
   }
}
```

После этого еще одним проходом собираем команды между лидерами в блоки. На этом разбиение на базовые блоки закончено.

Создание из ББЛ CFG

Был создан класс CFGraph, который представляет собой граф потока управления. Он базируется на библиотеке QuickGraph, которая предоставляет понятный и многофункциональный интерфейс работы с графами.

```
public class CFGraph
{
   public BidirectionalGraph<CFGNode, Edge<CFGNode>> graph =
      new BidirectionalGraph<CFGNode, Edge<CFGNode>>();

   public readonly List<IBaseBlock> Blocks;

   public CFGraph(List<IBaseBlock> blocks);
   public CFGNode GetRoot();
   public int NumberOfVertices();
   public IEnumerable<CFGNode> GetVertices();
}
```

Граф потока управления строится из базовых блоков следующим образом: 1. Для каждого базового блока строим узел его содержащий: CFGNode. В отличии от базового блока, узлы хранят ссылка на следующие базовые блоки. 1. Добавляем в граф все узлы 2. Основываясь на связях в узлах, добавляем в граф ребра

```
List<CFGNode> cfg_nodes = new List<CFGNode>(blocks.Count);
for (int i = 0; i < blocks.Count; i++)
{
    cfg_nodes.Add(new CFGNode(blocks[i]));
}

/// Create connections between CFGNode
/// ...

/// Create graph
graph.AddVertexRange(cfg_nodes);
foreach (var node in cfg_nodes)
{
    if (node.directChild != null)
    {
        graph.AddEdge(new Edge<CFGNode>(node, node.directChild));
    }

    if (node.gotoNode != null)
    {
        graph.AddEdge(new Edge<CFGNode>(node, node.gotoNode));
    }
}
```

Входные данные

Конструктору класса CFGraph подаётся на вход список базовых блоков трёхадресного кода (List<IBaseBlock> blocks).

Выходные данные

Экземпляр класса CFGraph хранит в себе граф из библиотеки - BidirectionalGraph<CFGNode, Edge<CFGNode>> graph, а также прочую служебную информацию необходимую для дальнейших преобразований.

Пример применения

```
var blocks = LinearToBaseBlock.Build(code);
var cfg = new CFGraph(blocks);
```

Доступные выражения внутри ББЛ(+оптимизации)

Постановка задачи

Найти доступные выражения внутри базовых блоков.

Описание алгортима

Выражение X+У доступно в точке Р, если любой путь от начального узла к Р вычисля- ет X+У и после такого вычисления и до достижения Р нет последующих присвоений X или У. Блок генерирует выражение x+y, если он вычисляет x+y и не выполняет последующих переопределений x и y.

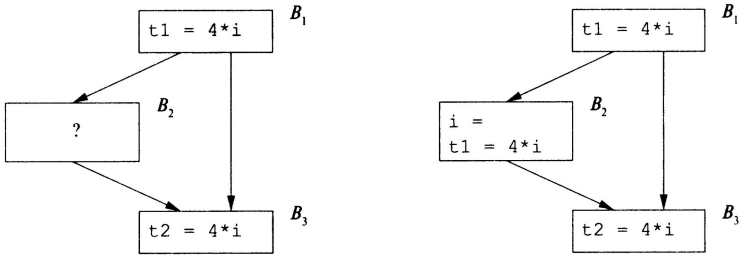

Рис. 9.17. Потенциальные общие подвыражения, пересекающие границы блоков

Figure 1: enter image description here

Входные данные

Базовый блок

Выходные данные

Множество выражений

Реализация алгоритма

```
1 public Expression(Operation op, IValue leftOper, IValue rightOper)
2     {
3         Op = op;
```

```
4         LeftOper = leftOper;
5         RightOper = rightOper;
6     }
7
8     public Expression()
9     {
10        LeftOper = null;
11        RightOper = null;
12        Op = Operation.NoOperation;
13    }
14
15    public override bool Equals(object obj)
16    {
17        if (obj is Expression)
18        {
19            Expression Other = (Expression)obj;
20
21            return Other.Op == this.Op &&
22                //Коммутативный случай
23                ((LinearHelper.IsBinOp(this.Op) || this.Op == Operation.Mult || this.Op ==
      Operation.Plus) &&
24                    (Other.LeftOper.Equals(this.LeftOper) &&
      Other.RightOper.Equals(this.RightOper) || Other.LeftOper.Equals(this.RightOper) &&
      Other.RightOper.Equals(this.LeftOper)) ||
25                    //Некоммутативный случай
26                    Other.LeftOper.Equals(this.LeftOper) &&
      Other.RightOper.Equals(this.RightOper));
27        }
28        else
29            return false;
30    }
```

Достигающие определения: вычисление `gen` и `kill`

Определение *Множеством gen_B* называется множество определений, генерируемых в базовом блоке B. *Множеством kill_B* называется множество всех прочих определений тех же переменный во всей остальной программе.

Эти множества входят в передаточную функцию для задачи о достигающих определениях, поэтому перед запуском итерационного алгоритма для нее необходимо вычислить эти множества для каждого блока.

Определения хранятся в виде пары, где первый элемент – метка команды с определением, вторая – имя определяемой переменной.

```
using Definition = Tuple<LabelValue, IdentificatorValue>;

public readonly Dictionary<IBaseBlock, List<Definition>> Gen;

public readonly Dictionary<IBaseBlock, List<Definition>> Kill;
```

Сначала реализуем функцию для выбора всех определений из блока:

```
static IEnumerable<Tuple<LabelValue, IdentificatorValue>> CalcGen(IBaseBlock block)
{
  return block.Enumerate()
    .Where(IsDefinition)
    .Select(t => Tuple.Create(t.Label, t.AsDefinition()));
}
```

Тогда алгоритм построения множеств gen и kill будет прямо следовать из определений:

```
foreach (var block in blocks)
{
  var gen = CalcGen(block).ToList();

  var vars = new HashSet<IdentificatorValue>(gen.Select(e => e.Item2));
  var kill = blocks.Where(b => b != block).SelectMany(CalcGen).Where(e =>
      vars.Contains(e.Item2));

  Gen[block] = gen.ToList();
  Kill[block] = kill.ToList();
}
```

Итерационный алгоритм для достигающих определений

Задача поиска достигающих определений важна в задаче анализа потоков данных и полезна при проведении многих оптимизаций, таких как поиск неинициализированных переменных или констант в конкретных точках программы.

Определение Определение d *достигает* точки p, если существует путь от непосредственно следующей за d точкой к p на котором d не уничтожается

Для поиска достигающих определений используется итеративный алгоритм для прямой задачи потока данных, на вход которому подается граф потока управления и множества gen (генерирующиеся определения) и kill (уничтожение определеий) для каждого блока.

Для хранения достигающих определений используется структура данных HashSet<LabelValue>

Передаточная функция имеет вид $f(x) = gen_b \cup (x \setminus kill_b)$

```
protected override HashSet<LabelValue> TransferFunc(CFGNode node)
{
  var res = GenKill.GenLabels(node.Value);
  res.UnionWith(In[node].Except(GenKill.KillLabels(node.Value)));
  return res;
}
```

Оператор сбора для итерационного алгоритма – объединение:

```
protected override HashSet<LabelValue> MeetOp(List<CFGNode> nodes)
{
  return new HashSet<LabelValue>(nodes.SelectMany(n => Out[n]));
}
```

Таким образом, на выходе мы имеем множества In и Out достигающих опрелений для входа и выхода каждого блока.

Живые переменные между ББЛ, множества use_B и def_B

Описание задачи

Построение def-use и use-def цепочки для блока.

Описание алгоритма

Для каждого ББЛ: 1. берем левый операнд, если он не содержится в def - добавляем его в use. 2. Аналогично для правого операнда. 3. Добавляем в def переменную, если идет присваивание значения.

Реализация алгоритма

```
public DefUseBuilder(List<IBaseBlock> blocks)
{
  foreach (var block in blocks)
  {
    var defsB = new List<Definition>();
    var usesB = new List<Definition>();

    foreach (var t in block.Enumerate())
    {
      var leftOperand = t.LeftOperand as IdentificatorValue;
      var rightOperand = t.RightOperand as IdentificatorValue;

      if (leftOperand != null && !defsB.Select(e => e.Item2).Contains(leftOperand))
      {
        usesB.Add(Tuple.Create(t.Label, leftOperand));
```

```
        }
        if (rightOperand != null && !defsB.Select(e => e.Item2).Contains(rightOperand))
        {
        usesB.Add(Tuple.Create(t.Label, rightOperand));
        }

        var def = t.AsDefinition();
        if (def != null)
        {
           defsB.Add(Tuple.Create(t.Label, def));
        }
     }

     Use[block] = usesB;
     Def[block] = defsB;
  }
}
```

Пример использования

```
DefUse = new DefUseBuilder(g.Blocks);
```

Живые переменные - итерационный алгоритм

Постановка задачи

Определить какие переменные являются активными(живыми) для каждого базового блока.

```
public sealed class ActiveVarsIterAlg : IterativeCommonAlg<HashSet<LabelValue>>
{
   private readonly DefUseBuilder DefUse;
   public ActiveVarsIterAlg(CFGraph g) : base(g)
}
```

Реализация алгоритма

Сначала получаем *Def*/*Use* информацию. Затем выполняем итерационный алгоритм.

```
public ActiveVarsIterAlg(CFGraph g) : base(g)
{
   DefUse = new DefUseBuilder(g.Blocks);
   ReverseRun();
}
```

```
public virtual void ReverseRun()
{
   foreach (var b in graph.GetVertices().Reverse())
      In[b] = Top;

   var nodes = new HashSet<CFGNode>(graph.GetVertices().Reverse());

   var cont = true;

   while (cont)
   {
      cont = false;
      foreach (var node in nodes)
      {
         var childNodes = new List<CFGNode>();
         if (node.directChild != null)
         {
            childNodes.Add(node.directChild);
         }
         Out[node] = MeetOp(childNodes);
         var prevIn = In[node];
         var newIn = In[node] = TransferFunc(node);

         if (ContCond(prevIn, newIn))
            cont = true;
      }
   }
}
```

Пример использования

```
var cfg = ListBlocksToCFG.Build(blocks);
var DefUse = new ActiveVarsIterAlg(cfg);
```

Доступные выражения между базовыми блоками - множества e_gen(b) и e_kill(b)

Постановка задачи

Необходимо произвести анализ выражений для каждого блока с целью построения множеств E_Gen и E_Kill

Описание алгортима

Мы проходим по списку всех блоков и для каждого собираем множества. Результирующие множества для каждого блока записываем в словарь множеств, где ключом выступает имя блока. Каждый блок обрабатывается процедурой, которая хранит ассоциативный массив переменных и списков выражений, их использующих.

Входные данные

Базовый блок

Выходные данные

Множество выражений

Реализация алгоритма

E_Gen - множество выражений, генерируемых в базовом блоке. E_Kill - множество выражений, уничтожаемых в базовом блоке

Вызывается genKills = **new** GenKillExprs(cfg); и после этого в genKills.Gen[block] содержатся выражения, которые генерируются для этого блока В genKills.Remove[block] содержатся выражения, которые удаляются в этом блоке

```
1  public GenKillExprs(CFGraph cfg)
2      {
3          var blocks = cfg.Blocks;
4
5          foreach (var block in blocks)
6          {
7              BlockDefs[block] = new List<StringValue>();
8              Gen[block] = new List<Expression>();
9              Remove[block] = new List<Expression>();
10
11             var countOfElems = block.Enumerate().Count();
12
13             foreach (var elem in block.Enumerate().Reverse())
14             {
15                 if (elem.IsBinOp())
16                 {
17                     BlockDefs[block].Add(elem.Destination);
18
19                     if (elem.Operation != Operation.NoOperation)
20                     {
```

```
                var expr = new Expression(elem.Operation, elem.LeftOperand,
    elem.RightOperand);

                var hasThisExpr = AllExpressions.Any(iexpr => iexpr.Equals(expr));
                if (!hasThisExpr)
                {
                   AllExpressions.Add(expr);
                }

                if (!BlockDefs[block].Contains(elem.LeftOperand) &&
    !BlockDefs[block].Contains(elem.RightOperand))
                {
                   Gen[block].Add(expr);
                }
             }

          }
       }

    }
```

Доступные выражения, итерационный алгоритм

Постановка задачи

Найти доступные выражения внутри базовых блоков.

Описание алгортима

```
OUT ВХОД[] = 0;
for каждый( базовый блок B, отличный от входного) OUT B[] = U;
while внесены( изменения в OUT)
for каждый( базовый блок B, отличный от входного) {
IN B[] = Прпредшественник— B OUT [P];
OUT B[] = e_genB U (IN B[] — екШв_);
}
```

Входные данные

Вход: граф потока, у которого для каждого блока B вычислены e_killB и e_genB.

Выходные данные

Словари ins и outs, содержашие в себе множества для каждого блока, получившиеся на последней итерации алгоритма.

```
InBlocks[vertce.Value] = UnionNodes(InBlocks[vertce.Value], OutBlocks[parent.Value]);

OutBlocks[vertce.Value] = new List<Expression>(genKills.AllExpressions);
```

Реализация алгоритма

```
class AvailableExprAnalyzer

  {
     public readonly Dictionary<IBaseBlock, List<Expression>> InBlocks = new
      Dictionary<IBaseBlock, List<Expression>>();
     public readonly Dictionary<IBaseBlock, List<Expression>> OutBlocks = new
      Dictionary<IBaseBlock, List<Expression>>();
     public readonly GenKillExprs genKills;

     protected readonly CFGraph cfg;

     public AvailableExprAnalyzer(CFGraph _cfg)
     {

        cfg = _cfg;
        // Генерируем e_gen и e_kills
        genKills = new GenKillExprs(cfg);

        // Инициализируем для всех ББЛ пустые списки с входными выражениями
        foreach (var vertice in cfg.GetVertices())
        {
           InBlocks.Add(vertice.Value, new List<Expression>());
        }
     }

     public void analyze()
     {
        var firstBlock = cfg.GetVertices().First();
        var vertces = cfg.GetVertices();
        // Инициализируем для кажлого ББЛ кроме прервого список выходных
       выражений равный U множество( всех выражений)
        foreach (var vertce in vertces)
        {
           if (firstBlock != vertce.Value)
```

```
32          {
33              OutBlocks[vertce.Value] = new List<Expression>(genKills.AllExpressions);
34          }
35      }
36
37      bool hasChanges = true;
38
39      while (hasChanges)
40      {
41          hasChanges = false;
42
43          foreach (var vertce in vertces)
44          {
45              if (firstBlock != vertce.Value)
46              {
47                  // Добавляем во входные выражения все выражения, которые выходят
    из родительских блоков
48                  foreach (var parent in vertce.ParentsNodes)
49                  {
50                      InBlocks[vertce.Value] = UnionNodes(InBlocks[vertce.Value],
    OutBlocks[parent.Value]);
51                  }
52
53                  //Копируем старый список выходных выражений для сравнения в
    будущем
54                  var oldOut = new List<Expression>(OutBlocks[vertce.Value]);
55
56                  // IN[B] — e_kill_B — убиваем в списке входных выражений те, которые
    будут убиты в текущем базовом блоке
57                  var resNodes = ResidualNodes(InBlocks[vertce.Value],
    genKills.Remove[vertce.Value]);
58
59                  // объединяем оставшиеся IN[B] с e_gen_B, те.. все что осталось после
    кровавой резни между IN и e_kill объединяем с тем, что родилось в данном ББЛ
60                  // охраняемC это все как выходные выражения для текущего базового
    блока
61                  OutBlocks[vertce.Value] = UnionNodes(genKills.Gen[vertce.Value], resNodes);
62
63                  // Если мы поменяли список выходных вырожений, то мы обязаны снова
    пройти по всем блокам и пересчитать входвыход/, ибо наш вход опирается на
    выходы других блоков, которые могли быть изменены
64                  if (!EqualNodes(OutBlocks[vertce.Value], oldOut))
65                  {
66                      hasChanges = true;
67                  }
68              }
```

```
69          }
70        }
71      }
```

Построить на CFG глубинное остовное дерево, перенумеровав элементы в порядке обратном посфиксному

Был создан класс DepthSpanningTree, который представляет собой остовное дерево CFG. Он базируется на библиотеке QuickGraph, которая предоставляет понятный и многофункциональный интерфейс работы с графами.

```
1  public class DepthSpanningTree
2  {
3    public Dictionary<CFGNode, int> Numbers { get; }
4    public BidirectionalGraph<CFGNode, Edge<CFGNode>> Tree { get; }
5    public DepthSpanningTree(CFGraph cfg);
6
7    // Finds back path from source to target, true if it is.
8    public bool FindBackwardPath(CFGNode source, CFGNode target);
9
10   public override string ToString();
11 }
12 }
```

Остовное дерево CFG строится из CFGraph следующим образом: узлы графа потока управления обходятся в прямом порядке, в процессе прохода они нумеруются.

Конструктор остова запускает рекурсивную функцию от корня CFGraph:

```
1  private void BuildTree(CFGNode текущийузел_, ref int счетчик)
2  {
3    1. Пометить текущий узел как просмотренный
4    2. Если нет потомков, то занумеровать и выйти
5    3. Добавить в остов текущий узел
6    4. Для каждого потомка:
7      1. Если он не посещен:
8        1. Если его нет в остове, добавить его
9        2. Добавить ребро к потомку
10       3. BuildTreeпотомок(, счетчик)
11     2. Занумеровать потомка
12 }
```

Входные данные

Конструктору класса DepthSpanningTree подаётся на вход CFGraph

Выходные данные

Экземпляр класса DepthSpanningTree хранит в себе граф из библиотеки - BidirectionalGraph<CFGNode, Edge<CFGNode>> Tree, а также прочую служебную информацию необходимую для дальнейших преобразований.

Пример применения

```
var blocks = LinearToBaseBlock.Build(code);
var cfg = new CFGraph(blocks);
var dst = new DepthSpanningTree(cfg);
```

Передаточная функция для реализации распространения констант. Распространение констант внутри базового блока

Постановка задачи

1. Реализовать передаточную функцию для распространения констант.
2. Реализовать распространение констант внутри базового блока.

- Передаточная функция для базового блока определяется следующим образом:

```
s — statementЕсли
 s — не присваивание => fb = I тождественная( функция)Если
 s — присваивание => {Для
 всех v => {
   Если v!=x => m'(v) = m(v)
   Если v = x {
      Если (statement: x := c){
         m'(x) = CONSTANT(c)
      }
      Если (statement: x := y <operation> z){
         m'(x) = m(y) <operation> m(z), если (m(z) == CONSTANT) && (m(y) == CONSTANT)

         m'(x) = NAC, если (m(y) == NAC) || (m(z) == NAC)

         m'(x) = UNDEF, иначе
      }
   }
}
}
```

На вход реализованной функции поступает строка адресного кода блока и текущая таблица констант. Возвращается новое значение переменной, ее принадлежность к одному из трех видов, в зависимости от данных в таблице в текущий момент: * NAC * UNDEF * CONSTANT

Передаточная функция

```
private VariableValue CalculateTransmitionFunc(IThreeAddressCode line,
        VariableConstantMap currentTable)
```

Для корректной проверки работы передаточной функции, в данной задаче было реализовано распространение констант внутри базового блока.

Пример использования протяжки констант внутри базового блока

```
ConstantPropagation constProp = new ConstantPropagation();
        var resBlock = constProp.OptimizeBlock(blocks[0] as BaseBlock);
```

Классификация ребер графа: наступающие, отступающие, поперечные

Задача заключается в том, чтобы классифицировать все ребра CFGraph. Для решения данной задачи необходимо DST

Ребра бывают трех типов: 1. Наступающее ребро: если в существует такое же ребро в DST 2. Отступающее ребро: если в DST существует прямой путь из target в source 3. Поперечное ребро в иных случаях

Для решения этой задачи были созданы следующие классы:

```
public enum EdgeType
{
  Coming = 1,
  Retreating = 2,
  Cross = 3
}

public class EdgeTypes : Dictionary<Edge<CFGNode>, EdgeType>
{
  public override string ToString();
}
```

Класс CFGraph был расширен:

```
public class CFGraph
{
  ...
```

```
    public EdgeTypes EdgeTypes { get; }
    ...
}
```

Классификация ребер CFGraph происходит следующим образом: 1. Строим глубинное островное дерево для CFGraph 2. Проходим по всем ребрам в CFGraph и классифицируем их согласно определению

Она вызывается в конструкторе CFGraph.

Входные данные

Конструктору класса CFGraph подаётся на вход список базовых блоков трёхадресного кода (List<IBaseBlock> blocks).

Выходные данные

Экземпляр класса CFGraph хранит в себе граф из библиотеки - BidirectionalGraph<CFGNode, Edge<CFGNode>> graph, а также прочую служебную информацию необходимую для дальнейших преобразований. В том числе поле EdgeTypes, содержащее словарь с классификацией ребер.

Пример применения

```
var blocks = LinearToBaseBlock.Build(code);
var cfg = new CFGraph(blocks);
Console.WriteLine(cfg.EdgeTypes.ToString());
```

Являются ли все ли отступающие ребра обратными

Общие сведения

Ребро (a -> b) является обратным, если b доминирует над a. Узел a доминирует над узлом b, если любой путь от входного узла графа к b проходит через a.

Постановка задачи

Необходимо было проверить, являются ли все отступающие рёбра обратными

Входные данные

Для решения используются отступающие рёбра графа (классификация была проведена в другой задаче) и дерево доминаторов (было построено в другой задаче)

Выходные данные

Функция возвращает True, если все отступающие ребра обратные, иначе False

Решение задачи

Из определений следует, что для решения задачи необходимо проверить, что для всех отступающих ребер (a -> b) узел b доминирует над узлом a.

Для решения в класс графа управления (CFGraph) был добавлен метод allRetreatingEdgesAreBackwards()

```
1     public bool allRetreatingEdgesAreBackwards()
2     {
3        return EdgeTypes.Where(
4        edgeType => edgeType.Value == EdgeType.Retreating)
5           .Select(edgeType => edgeType.Key).ToList()
6           .All(edge => isDominate(edge.Target, edge.Source));
7     }
```

Для определения, является ли узел a доминирующим по отношению к узлу b, используется дерево доминаторов. Если существует путь от узла a к узлу b, то a доминирует над b.

Для поиска пути в дереве доминаторов был реализован алгоритм DFS.

```
1     private bool isWayExists(DominatorTreeNode from, DominatorTreeNode to)
2     {
3        if (from.Equals(to))
4           return true;
5
6        visited.Add(from);
7
8        var nodes = graph.Edges.Where(dtn => dtn.Source.Equals(from))
9  .Select(dtn => dtn.Target)
10 .Where(dtn => !visited.Contains(dtn)).ToList();
11
12       foreach (var node in nodes)
13          if (isWayExists(node, to))
14             return true;
15       return false;
16    }
```

Для каждого обратного ребра найти естественный цикл

Общие сведения

Ребро (a -> b) является обратным, если b доминирует над a. Для обратного ребра (a -> b) естественный цикл - множество узлов, которые могут достичь a, не проходя через b.

Постановка задачи

Для каждого обратного ребра необходимо найти его естественный цикл

Входные данные

Для решения используются граф потока управления и его обратные ребра

Выходные данные

Функция возвращает естественные циклы для всех обратных ребер графа

Решение задачи

Для каждого обратного ребра (a -> b) запустим алгоритм DFS, начиная с узла a по обратному графу , причём изначально b помечается уже посещенный, чтобы через него не проходил поиск

В класс графа потока управления (CFGRaph) добавлена функция, находящая все обратные ребра, и определяющая для них естественные циклы

```
public List<List<CFGNode>> getNaturalCyclesForBackwardEdges()
{
  return EdgeTypes
    .Where(edgeType => edgeType.Value == EdgeType.Retreating)
    .Select(edgeType => edgeType.Key)
    .Where(edge => isBackwardEdge(edge))
    .Select(edge =>
      naturalCycleGraph.findBetween(edge.Source, edge.Target))
    .ToList();
}
```

Для определения естественного цикла, был реализован алгоритм DFS, который запоминает все пройденные узлы

```
public List<CFGNode> findBetween(CFGNode from, CFGNode to)
{
   visitedNodes.Clear();
   visitedNodes.Add(to);

   backDFS(from);

   return visitedNodes;
}
private void backDFS(CFGNode currentNode)
{
   if (!visitedNodes.Contains(currentNode))
   visitedNodes.Add(currentNode);

   getNodes(currentNode).ToList()
      .Where(node => !visitedNodes.Contains(node)).ToList()
      .ForEach(node => backDFS(node));
}
```

Построение дерева доминаторов

Общие сведения

Определение *Дерево доминаторов* – это дерево, в котором входной узел является корнем, а каждый узел ddоминирует только над своими потомками в дереве.

Для каждого узла графа потока управления при помощи итерационного алгоритма для задачи потока данных находятся все *доминаторы* – такие узлы графа, которые присутствуют в любом пути от начала программы до данного узла. Далее, для каждого узла (кроме корня графа потока управления) выбирается непосредственный доминатор, который и объявляется родителем в дереве доминаторов.

Таким образом, для построения дерева доминаторов на вход алгоритму необходимо подать только граф потока управления.

Реализация

В реализации данного модуля был использован пакет NuGet QuickGraph.

Узлами дерева доминаторов являются объекты класса DominatorTreeNode:

```
public class DominatorTreeNode
{
   public DominatorTreeNode ParentNode;
   public List<DominatorTreeNode> ChildrenNodes;
   public CFGNode CFGNode;
```

```
6
7     ...
8 }
```

Сама реализация алгоритма построения дерева выглядит следующим образом:

```
1  public DominatorTree(CFGraph cfg)
2  {
3      var doms = new DominatorsIterAlg(cfg).Dom; // 1
4      var vertices = doms.Keys.Select(x => new DominatorTreeNode(x)).ToList();
5      graph.AddVertexRange(vertices); // 2
6
7      foreach (var node in vertices) // 3
8      {
9          var dominatedBy = doms[node.CFGNode].ToList();
10         dominatedBy.Reverse();
11         var cfgClosestDominator = dominatedBy.Skip(1).FirstOrDefault(); // 3.1
12         if (cfgClosestDominator != null)
13         {
14             var domClosestDominator = vertices
15                 .FirstOrDefault(x => x.CFGNode == cfgClosestDominator);
16
17             node.ParentNode = domClosestDominator; // 3.2
18             domClosestDominator.AddChild(node);
19
20             graph.AddEdge(new Edge<DominatorTreeNode> // 3.3
21                     (domClosestDominator, node));
22         }
23     }
24 }
```

1. Используется итерационный алгоритм нахождения доминаторов.
2. Преобразованные вершины графа потока управления объявляются вершинами дерева доминаторов.
3. Для каждой вершины: 1) находится ближайший доминатор (непосредственный) 2) доминатор объявляется предком текущей вершины, а сама вершина – потомком доминатора 3) построенная дуга добавляется в дерево доминаторов

Пример использования

Для использования дерева доминаторов достаточно лишь передать в конструктор класса DominatorTree в качестве аргумента граф потока управления программы:

```
1 var dt = new LYtest.DominatorTree.DominatorTree(cfg);
```

Класс DominatorTree предоставляет доступ к корню и вершинам дерева, а также к самой структуре QuickGraph.BidirectionalGraph, поэтому можно использовать все реализованные для этого класса алгоритмы. Кроме того, так как информация о предках и потомках узлов дерева хранится в самих узлах, то можно передвигаться по дереву непосредственно.

Формирование поседовательности областей R1,…,Rn в восходящем порядке

Описание задачи

Необходимо построить восходящую последовательность областей R1,…,Rn, где первые k областей R1,…,Rk - области-листья, представляющие собой один узел графа. Затем естественные циклы упорядочиваются изнутри наружу, т.е. начиная с наиболее внутренних циклов. В итоге все естественные циклы сводятся к отельным узлам.

Входные данные

Приводимый граф потока. ### Выходные данные Список областей графа потока ### Описание алгоритма 1. Построить список листьев-областей, состоящий из отдельных узлов графа. 2. Неоднократно выбрать естественный цикл L, такой, что если существуют либые естественные циклы, содержащиеся в L, то тела и области этих цклов уже внесены в список. Добавить область тела, состоящую из тела L (без обратных ребер к заголовку L). Затем добавить область цикла L (содержит обратные ребра к заголовку L). 3. Если весь граф не представляет собой естественный цикл, добавляем в конец списка область, состоящую из всего графа потока целиком.

Описание реализации

Для решения задачи был создан класс RegionSequence со следующим интерфейсом

```
1 public class RegionSequence
2 {
3   List<Region> regions;
4   public List<Region> Regions { get { return regions; } }
5   public RegionSequence(CFGraph cfg) { ... }
6   ...
7 }
```

Построение последовательности происходит в конструкторе класса, на вход которому подается приведенный граф управления.

На первом этапе строятся области-листья для каждого узла графа.

```
List<CFGNode> allNodes = cfg.GetVertices().ToList();
List<Edge<CFGNode>> edges = cfg.EdgeTypes.Select(e => e.Key).ToList();
foreach (var node in cfg.GetVertices())
{
   var edgesFromNode = edges.FindAll(e => e.Target == node);
   regions.Add(new LeafRegion(node, edges, NextName()));
}
```

Затем ищутся все естествеенные циклы с помощью встроенного метода графа (getNaturalCyclesForBackwardEdges()). Также создаются множества cyclesHeaders (все заголовки найденных циклов) и addedCyclesHeaders (заголовки циклов, которые внесены в список областей) для удобства учета обработанных циклов.

```
var nc = cfg.getNaturalCyclesForBackwardEdges();
List<Edge<CFGNode>> edges = cfg.EdgeTypes.Select(e => e.Key).ToList();
HashSet<CFGNode> cyclesHeaders = new HashSet<CFGNode>(nc.Select(c => c[0]));
HashSet <CFGNode> addedCyclesHeaders = new HashSet<CFGNode>();
```

После чего, пока есть циклы, не добавленные в список областей: 1. Выбираются циклы, подходящие для обработки (заголовок принадлежит множеству заголовков добавленных циклов) 2. Для каждого найденного цикла в список областей добавляется область тела, а затем бласть цикла. 3. Обработанные циклы удаляются из списка недобавленных.

```
while (nc.Count > 0)
{
   List<List<CFGNode>> cyclesToAdd = nc.FindAll(c => c.Skip(1).All(node =>
   {
      return cyclesHeaders.Contains(node) ? addedCyclesHeaders.Contains(node) : true ;
   }));
   foreach (var cycle in cyclesToAdd)
   {
      var nodes = new HashSet<CFGNode>(cycle);
      AddCycle(cycle, edges, nodes);
      addedCyclesHeaders.Add(cycle[0]);
   }
   nc.RemoveAll(c => addedCyclesHeaders.Contains(c[0]));
}
```

Последний шаг: если граф не представляется собой естественный цикл, то внести в список область, состоящую из всего графа (все узлы и ребра).

```
bool cfgIsNaturalCycle = cyclesHeaders.Contains(allNodes[0]);
if (!cfgIsNaturalCycle)
 regions.Add(new BodyRegion(allNodes[0], allNodes, edges, NextName()));
```

Пример использования

Для создания последовательности областей нужно создать объект RegionSequence и передать на вход граф потока данных.

```
var rs = new RegionSequence(cfg);
```

Разработка класса “область” с разновидностями “область тела” и “область цикла”

Описание задачи

Определение *Область* графа потока представляется собой набор узлов *N* и ребе *E*, таких, что: 1. Существует заголовок *h* из *N*, доминирующий над всеми узлами в *N*; 2. Если некоторый узел m может достичь узла *n* из *N*, минуя *h*, то *m* также входит в *N*; 3. *E* - множество всех ребер потока управления между узлами *n1* и *n2* из *N*, за исключением, возможно, некоторых ребер, входящих в *h*.

Области подразделяются на следующие виды: 1. Каждый базовый блок рассматривается как область. Такая область называется *область-лист*. 2. *Область тела* представляет собой все узлы и ребра тела естественного цикла, за исключением обратных ребер к заголовку. 3. *Область цикла*. Отличается от области *области тела* тем, что включает в себя обратные ребра, ведущие к заголовку.

Описание реализации

Базовый класс для областей - Region. Хранит свой ездинственный узел, он же является заголовком, и ребра, выходящие из этого узла.

```
public class Region
{
 List<Edge<CFGNode>> edges;
 public List<Edge<CFGNode>> Edges { get { return edges; } }
 public CFGNode Header { get; }
 public string Name { get; set; }
}
```

От данного класса наследюся классы LeafRegion и NonLeafRegion для листовых и нелистовых областей соответственно. Их отличие состоит в том, что NonLeafRegion хранит список узлов, в то время как LeafRegion один узел - заголовок, который и является соответствующим узлом графа. NonLeafRegion также хранит список всех дуг графа, выходящих из узлов, принадлежащих области.

```
public class LeafRegion : Region
{ }
```

```
3
4  public class NonLeafRegion : Region
5  {
6     List<CFGNode> nodes;
7     public List<CFGNode> Nodes { get { return nodes; } }
8  }
```

От NonLeafRegion наследуются классы для *области тела* и *области цикла*: BodyRegion и LoopRegion (хранит в себе область тела).

```
1  public class BodyRegion : NonLeafRegion
2  { }
3
4  public class LoopRegion : NonLeafRegion
5  {
6     BodyRegion bodyReg;
7  }
```