Документация проекта Оптимизирующий компилятор Pile

Содержание

Введение

Была поставлена задача разработки оптимизирующего компилятора для языка, описываемого выбранной грамматикой.

В связи с тем, что для кода проекта был выбран язык C#, было решено использовать генератор синтаксического анализатора Yacc и генератор лексического анализатора Lex.

Структура проекта

Проект состоит из трех основных компонентов:

- библиотеки, реализующей оптимизации;
- консольного приложения;
- реализации графического интерфейса
- проекта с тестами

Описание грамматики

Была поставлена задача разработки оптимизирующего компилятора для языка, описываемого грамматикой:

Типы данных:

- целое число
- вещественное число
- строка

```
// SimpleYacc.y

        public double dVal;
        public int iVal;
        public string sVal;
```

Поддерживаемые операции:

- операции сравнения (<, >, <=, >=, ==, !=)
- арифметические операции (+, -, /, *)

```
bin_sign    : LS { $$ = BinSign.LS; } // <
            | GT { $$ = BinSign.GT; } □
            | LE { $$ = BinSign.LE; }
            | GE { $$ = BinSign.GE; }
            | EQ { $$ = BinSign.EQ; }
            | NE { $$ = BinSign.NE; }
```

Поддерживаемые операторы:

- оператор присваивания
- блок кода
- условный оператор (if-then, if-then-else)

- условный оператор (if-then, if-then-else)
- циклы
    - cycle - выполнение указанного кол-ва итераций
    - for
    - repeat-until
    - while

```
// SimpleYacc.y

statement   : assign SEMICOLON { $$ = $1; }
            | block { $$ = $1; }
            | cycle { $$ = $1; }
            | if_st { $$ = $1; }
            | for_st { $$ = $1; }
            | rep_unt { $$ = $1; }
            | while_st { $$ = $1; }
            ;
```

Операторы отделяются с помощью точки с запятой (;)

Блоки выделяются с помощью begin и end :

```
block       : BEGIN stlist END { $$ = $2; }
```

Вспомогательные задачи

Парсер языка + AST

Задача является основой всей последующей работы

Постановка задачи

Эта задача подразумевает создание парсера языка и построения абстрактного синтаксического дерева по тексту программы

Описание алгортима

При разработке использовались следующие средства: *GPLex* - лексический анализатор *GPPG* - синтаксический анализатор

Входные данные

Текст программы

Выходные данные

Абстрактное синтаксическое дерево

Пример применения

//в переменной text находится текст программы

```
Scanner scanner = new Scanner();
scanner.SetSource(text, 0);

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

if (!b)
    Console.WriteLine("Ошибка");
else Console.WriteLine("Программа распознана");

var prettyVisitor = new PrettyPrintVisitor();
parser.root.Accept(prettyVisitor);
Console.WriteLine(prettyVisitor.Text);
```

Пример правильной программы

```
{
   b = -2;
   a = 3;
   a = -b;
   c = b + 2;
   d = a * b + c * d + 3;

   if 1
      b = 1;
   else
   {
    b = 0;
   }

   if a
      a = 1;

   while x <= 10 do
   {
    x = x + 1;
   }

   cycle 3
   {
    a = a + a;
   }

   for i=1 to 10 do
    a = a + i;
}
```

Трехадресный код

Представление трехадресного кода

Это задача требуется для выполнения Задачи Генерация 3-адресного кода по AST.

Постановка задачи

Написать классы для представления 3-адресного кода

Входные данные

3-адресный код

Выходные данные

Класс ThreeAddressCode

Пример применения

```
var threeAddressCode = new ThreeAddressCode
    {
        Operation = Operation.Minus,
        Destination = new IdentificatorValue("variable_name1"),
        LeftOperand = new IdentificatorValue("variable_name2"),
        RightOperand = new NumericValue(12)
    };
```

Генерация трехадресного кода

Интерфейс для работы с трехадресным кодом

```
// ../Library/ThreeAddressCode/IThreeAddressCode.cs

using OptimizingCompilers2016.Library.ThreeAddressCode.Values.Base;

namespace OptimizingCompilers2016.Library.ThreeAddressCode
{
    using Values;
    public interface IThreeAddressCode
    {
        Operation Operation { get; set; }
        IValue LeftOperand { get; set; }
        IValue RightOperand { get; set; }
        StringValue Destination { get; set; }
        LabelValue Label { get; set; }
    }
}
```

Каждая команда трехадресного кода имеет следующий вид: > Label: Destination = LeftOperand Operation RightOperand

где

- Label - метка
- Destination
    - либо переменная, которой присваивается значение (ее имя - идентификатор),
    - либо команда, к которой будет осуществлен переход (по метке)
- Operation - операция/оператор
    - бинарные
        - ♠ арифметические: -, +, -, /, *
        - ♠ сравнение: <, <=, >, >=, ==, !=
    - унарные
        - ♠ знак (+, -)
    - пустая операция (NoOp) - для реализации переходов goto
    - присваивание
    - оператор перехода Goto
    - оператор условного перехода CondGoto
- LeftOperand

    - либо левый операнд бинарной операции
    - либо единственный операнд унарной операции
- RightOperand - правый операнд бинарной операции

Пример // если поле не указано, то оно равно null Оператор присваивания:

а) label : dest = leftOp op rightOp б) label : dest = leftOp в) label : dest = op leftOp где op - арифметическая или бинарная операция Оператор перехода: а) label: dest goto

Обход синтаксического дерева

Для обхода синтаксического дерева используется паттерн Visitor:

- класс LinearCode.cs реализует генерацию линейного кода (реализация интерфейса iVisitor)
- класс PrettyPrintVisitor используется для печати трехадресного кода.

Генерация линейного кода

Имена меток - "%l" Имена временных переменных - "$t"

Условный оператор:

Исходный код

```
if условие then оп_1 else оп_2
```

Линейный код

```
%l1: $t1 = условие
%l2: if $t1 goto %l5
%l3: оп_2
%l4: goto %l6
%l5: оп_1 □
%l6: noOp
```

Оператор цикла while:

Исходный код

```
while условие do  оп_1
```

Линейный код

```
  %l1: $t1 = условие
  %l2: if $t1 goto %l5
  %l3: goto %l6
  %l4: оп_1
  %l5: goto %l1
  %l6: noOp
```

Оператор цикла cycle

Исходный код

```
cycle 5 do оп_1
```

Линейный код

```
  %l1: $t1 = 1
  %l2: $t2 = t1 GT 5
  %l3: if $t1 goto %l7
  %l4: оп_1
  %l5: $t1 = $t1 + 1
  %l6: goto %l2
  %l7: no op
```

Базовые блоки

Представление Базового Блока

Класс, представляющий Базовый Блок.

Постановка задачи

Создать представление для Базового Блока.

Входные данные

Имя блока, трехадресный код, входной блок (опционально), выходной блок (опционально)

Выходные данные

Класс BaseBlock

Пример применения

```
BaseBlock b1 = new BaseBlock();
b2.Name = "B2";
```

Задача разбиения на базовые блоки

От этой задачи зависит большинство последующих задач.

Постановка задачи

Разбить программу на базовые блоки.

Описание алгоритма

Началом базового блока считаются метки, концом базового блока считаются переходы. После прохода алгоритма могут появиться путсые блоки, поэтому в конце алгоритма выполняется удаление лишних пустых блоков.

Входные данные

Выходные данные

Граф потока управления, построенный на основе получившихся блоков.

Пример использования

```
var linearCode = new LinearCodeVisitor();
parser.root.Accept(linearCode);
var blocks = BaseBlockDivider.divide(linearCode.code);
```

Граф потока управления: представление и построение

Класс, представляющий граф потока упраления.

Постановка задачи

Выбрать представление для графа потока управления.

В качестве внутреннего представления используется обобщенная реализация двунаправленного графа, предоставляемая библиотекой QuickGraph

Описание алгоритма

Построение CFG:

1. Добавляем в граф все узлы (базовые блоки)
2. Устанавливаем связи между узлами (исходя из связей, указанных в ББ)

```
// Добавление вершин в граф
CFG.AddVertexRange(blocks);
// Теперь, когда граф содержит вершины, можно добавить и дуги
foreach (var block in blocks)
{
    if (block.Output != null)
    {
        CFG.AddEdge(new Edge<BaseBlock.BaseBlock>(block, block.Output));
    }
    if (block.JumpOutput != null)
    {
        CFG.AddEdge(new Edge<BaseBlock.BaseBlock>(block, block.JumpOutput));
    }
}
```

Входные данные

Конструктору класса CFG подаётся на вход список базовых коллекция базовых блоков трёхадресного кода (`IEnumerable<BaseBlock.BaseBlock>`).

Выходные данные

Конструктор инициализирует поле класса типа `BidirectionalGraph<BaseBlock.BaseBlock, Edge<BaseBlock.BaseBlock>>`

Пример применения

```
var blocks = BaseBlockDivider.divide(linearCode.code);
var CFG = new ControlFlowGraph(blocks);
```

Оптимизации внутри базового блока

Def-Use информация внутри блока

Описание задачи

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

Описание алгоритма

Построение def-use цепочек

1. Проходим по командам блока (линейный код)
2. Берем все вхождения переменных из левых частей инструкций линейного кода.
3. Ищем все вхождения этой переменной в правых частях инструкций.

Построение use-def цепочек

1. Проходим по командам блока (линейный код)
2. Ищем все вхождения переменных в правых частях инструкций.
3. Берем все вхождения этих переменных из левых частей инструкций линейного кода.

Входные данные

Базовый блок

Выходные данные

def-use

Словарь, ставящий в соответствие определениям переменной ее использованиям.

use-def

Словарь, ставящий в соответствие использованию переменной ее определение.

```
// def-use
 public Dictionary<IntraOccurence, HashSet<IntraOccurence>> defUses { get; set; } = new Dicti
 □
 // use-def
 public Dictionary<IntraOccurence, HashSet<IntraOccurence>> useDefs { get; set; } = new Dicti
 □
 // where intraOcurence is
 /  System.Tuple<OptimizingCompilers2016.Library.BaseBlock, System.Tuple<int, int, Optimizing
```

Пример использвоания

```
foreach (var block in blocks)
{
   InblockDefUse DU = new InblockDefUse(block);
   foreach (var item in DU.defUses)
   {
   // ...
   }
}
```

Удаление мертвого кода

Задача является основой для удаления мертвого кода

Постановка задачи

Эта задача подразумевает удаление мертвого кода в рамках одного базового блока

Описание алгортима

Проходимся по элементам трехадресного кода в рамках блока (в обратном направлении) переменная может быть активной и неактивной, изначально все переменные считаются актиными (они могут использоваться в других блоках) удаляем элементы трехадресного кода, которые явдяются присвоением и переменная, которой присваивается значение, является неактивной если переменной присваивается значение, она помечается как неактивная если переменная используется в правой части, она помечается как активная

Повторяем вышеизложенные действия пока количество элементов в трехадресном коде не перестанет меняться

Входные данные

Базовый блок

Выходные данные

Базовый блок без мертвого кода

Пример применения

```
foreach (var block in blocks)
{
    Console.WriteLine(block.ToString());
    DeadCodeDeleting.optimizeDeadCode(block);
    Console.WriteLine("After optimization:");
    Console.WriteLine(block.ToString());
    Console.WriteLine("-------");
}
```

Пример1

```
{
  b = 2;
  a = 3;
```

```
  a = b;
  c = b + 2;
  d = a * b;
}
```

Результат:

Block B0 Ins: [] b := 2 a := 3 a := b c := b + 2 d := a * b JumpOut: Out:

After optimization: Block B0 Ins: [] b := 2 a := b c := b + 2 d := a * b JumpOut: Out:

Пример2

```
{
  b = 2;
  a = 3;
  a = 5;
  c = b + 2;
  b = 4;
  d = a * b;
  c = 3;
}
```

Результат:

```
Block B0
Ins: []
          b := 2
          a := 3
          a := 5
          c := b + 2
          b := 4
          d := a * b
          c := 3
JumpOut: □
Out: □

After optimization:
Block B0
Ins: []
          a := 5
          b := 4
          d := a * b
          c := 3
JumpOut: □
Out:
```

Свертка констант

Постановка задачи

Если в линейном коде, в выражении, правый и левый операнд являются константами, производим вычисление значения выражения и заменяем выражение на его результат.

Описание алгоритма

1. Проходим по всем инструкциям линейного кода в блоке
2. Если в инструкции оба операнда являются константами, производим вычисление значения

выражения
3. Заменяем выражение на его результат

Входные данные

Базовый блок.

Выходные данные

Видоизмененный блок.

Пример использования

```
ConstantFolding.transform(block);
```

Протяжка констант

Задача позволяет протянуть константы внутри базового блока

Постановка задачи

Эта задача подразумевает протяжку констант в рамках одного базового блока

Описание алгоритма

1. Находим первое присваивание константы какой-либо перемонной.
2. Заменяем все вхождения данной переменной в правой части, пока не дойдет до присваивания этой переменной нового значения или не дойдем до конца блока.
3. Повторять пункты 1-2, пока количество замен больше 0.

Входные данные

Базовый блок

Выходные данные

Базовый блок, где протянуты все константы

Пример применения

```
var blocks = BaseBlockDivider.divide(linearCode.code);
IOptimizator optimizator = new ConstantPropagationOptimizator();
foreach (var block in blocks)
{
    optimizator.Optimize(block);
}
```

Учёт алгебраических тождеств

Постановка задачи

Реализовать учет алгебраических тождеств внутри Базового Блока. Список алгебраических тождеств:

Реализовать учет алгебраических тождеств внутри базового блока. Список алгебраических тождеств:

```
1 * a = a
a * 1 = a
0 + a = a
a + 0 = a
a - 0 = a
a / 1 = a
```

Входные данные

Базовый Блок

Выходные данные

Модифицированный Базовый Блок, внутри которого проведены оптимизации по учету алгебраических тождеств.

Пример применения

```
  var opt = new AlgebraicIdentityOptimizator();
  foreach (var block in blocks)
    opt.Optimize(block);
```

Оптимизация общих подвыражений

Постановка задачи

Задача подразумевает оптимизацию одинаковых выражений в рамках одного базового блока.

Описание алгортима

Алгоритм реализован в 2 прохода. При первом проходе мы ищем одинаковые подвыражения среди элементов трёхадресного кода базового блока и запоминаем их местоположение в базовом блоке. Далее, при втором проходе происходит создание вспомогательной переменной и замена общего подвыражения. Для хранения подвыражений создана структура `BinaryExpression`, которая также отвечает за сравнение подвыражений.

Входные данные

Базовый блок

Выходные данные

Базовый блок с замененными общими подвыражениями

Пример применения

```
{
    var block = new BaseBlock();
    ...
    var cse = new CommonExperssions();
    Console.WriteLine(block);
    cse.Optimize(block);
    Console.WriteLine("After optimization:");
```

```
    Console.WriteLine("After optimization: ");
    Console.WriteLine(block);
}
```

Пример

```
{
  ...
  b = 2 + a;
  a = 3;
  c = 2 + a;
  d = a + 2;
}
```

Результат:

```
  b = 2 + a;
  a = 3;
  tmp0 = 2 + a;
  c = tmp0;
  d = tmp0;
```

Оптимизации между базовыми блоками

Общий итерационный алгоритм

Постановка задачи

Реализовать базовый итерационный алгоритм и предусмотреть возможности для его использования в связанных задачах.

Описание класса

Абстрактный класс `BaseIterationAlgorithm<T>` находится в пространстве имён `OptimizingCompilers2016.Library.Analysis`.

На тип `T` поставлено ограничение `IClonable`.

Класс реализует интерфейс `Semilattice`.

В классе `BaseIterationAlgorithm` определены следующие методы:

- Оператор сбора:

  ```
  public abstract T Collect(T x, T y);
  ```

- Передаточная функция:

  ```
  protected abstract T Transfer(T x, BaseBlock b);
  ```

-

Функция, необходимая для установки начального значения элемента данных:

```
protected abstract T SetStartingSet();
```

- Функция, необходимая для заполнения сущностей `generators` и `killers`, используемых в связанных задачах:

```
protected abstract void FillGeneratorsAndKillers(List<BaseBlock> blocks);
```

- Функция, необходимая для заполнения вспомогательных сущностей, общих для связанных задач

```
protected void FillSupportingStructures(List<BaseBlock> blocks);
```

- Публичный метод, необходимый для запуска анализа на основе итерационного алгоритма

```
public abstract void RunAnalysis(List<BaseBlock> blocks);
```

- Итерационный алгоритм

```
protected virtual void IterationAlgorithm(List<BaseBlock> blocks);
```

Описание алгоритма

Работа алгоритма заключается в следующем:

```
foreach B - базовый блок + Вход
{
    OUT[B] = {}  //инициализируем пустым множеством
    while OUT меняется
    {
        foreach B - базовый блок do
        IN[B] =  OUT[P], где P - предшественники B, - оператор сбора
        OUT[B] = fb(IN[B]), где fb - передаточная функция
    }
}
```

Входные данные

Список базовых блоков программы.

Выходные данные

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

```
protected Dictionary<BaseBlock, T> outs = new Dictionary<BaseBlock, T>();
protected Dictionary<BaseBlock, T> ins = new Dictionary<BaseBlock, T>();
```

Оператор сбора

Интерфейс, классы, реализующие этот интерфейс должны предоставлять реализацию оператора сбора (Collect)

Постановка задачи

Проектирование интерфейса для работы с оператором сбора.

Входные данные

Типовый параметр T --- множество, на котором вводится полурешётка (например: мн-во всех подмножеств множества определений в случае задачи о достигающих определениях)

```
interface Semilattice<T>
{
    T Collect(T x, T y);
}
```

Передаточная функция

Классы, наслудуемые от BaseIterationAlgorithm, должны определять передаточную функцию (Transfer)

Постановка задачи

Проектирование абстрактной функции для представления оператора сбора

Входные данные

Типовый параметр T --- множество, на котором вводится полурешётка (например: мн-во всех подмножеств множества определений в случае задачи о достигающих определениях)

```
protected abstract T Transfer(T x, BaseBlock b);
```

Глобальные def-use цепочки

Постановка задачи

Построение def-use и use-def цепочек между блоками.

Описание алгоритма

Def-Use

1. Используется базовый итерационный алгоритм для проведения анализа
2. На основе анализа запускаем еще раз внутриблочные def-use для каждого блока B, передавая на вход IN[B], где IN[B] - это список генераторов, достигших этого блока

Use-Def

Аналогично случаю с def-use.

Входные данные

Список блоков.

Выходные данные

```
// Def-Use:
public Dictionary<IntraOccurence, HashSet<IntraOccurence>> GetUseDefs();
// Use-Def
public Dictionary<IntraOccurence, HashSet<IntraOccurence>> GetDefUses();
```

Пример использования

```
var gdu = new GlobalDefUse();
gdu.runAnalys(blocks);
gdu.getDefUses();
```

Задачи оптимизации

Анализ активных переменных между базовыми блоками

Постановка задачи

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

```
class ActiveVariables
{
    public new Dictionary<string, HashSet<IdentificatorValue>> result;

    private Dictionary<string, HashSet<IdentificatorValue>> IN;
    private Dictionary<string, HashSet<IdentificatorValue>> OUT;
    private Dictionary<string, HashSet<IdentificatorValue>> Def;
    private Dictionary<string, HashSet<IdentificatorValue>> Use;
    private List<BaseBlock> blocks;

    public ActiveVariables(List<BaseBlock> blocks)
}
```

Описание алгоритма

Сначала получаем Def-Use информацию. Затем выполняем итерационный алгоритм.

```
// итерационный алгоритм
    foreach B - базовый блок + Выход
    {
        IN[B] = {}  //инициализируем пустым множеством
        while IN[B] - меняется
        {
            foreach B - базовый блок do
            OUT[B] = U IN[P], где P - потомки B
            IN[B] = USE[B] U (OUT[B] - DEF[B])
        }
     }
```

Входные данные

Граф потока управления.

Выходные данные

Объект класса ActiveVariables, хранящий для каждого базового блока информацию об активных переменных.

Пример использования

```
var AV = new ActiveVariables(new ControlFlowGraph(blocks));
AV.runAnalys(); 
//AV.result - Dictionary, ключ - имя блока, значение - список активных переменных для данно
```

Оптимизация по результатам анализа активных переменных

Это задача требует выполнение задачи анализ активных переменных.

Постановка задачи

Провести оптимизацию после выполнения анализа активных переменных

Описание алгоритма

В оптимизации по удалению мертвого кода мы не предполагаем, что переменныя живая. Это значение мы берем из анализа активных переменных

```
if (!viewed.Contains(DU.defUses.ElementAt(i).Key.Item2.Item2) && !activeVars.Contains(DU.defU
{
    viewed.Add(DU.defUses.ElementAt(i).Key.Item2.Item2);
}
else
{
    if (DU.defUses.ElementAt(i).Value.Count == 0)
    {
        toDelete.Add(block.Commands[DU.defUses.ElementAt(i).Key.Item2.Item1]);
    }

}
```

Входные данные

Базовые блоки и результат анализа активных переменных

Выходные данные

Оптимизированные базовые блоки

Пример применения

```
var blocks = BaseBlockDivider.divide(linearCode.code);
var AV = new ActiveVariables(new ControlFlowGraph(blocks));
AV.runAnalys();
foreach (var block in blocks)
{
    DeadCodeDeleting.optimizeDeadCode(block, AV.result[block]);
}
```

Доступные выражения

Анализ доступности выражений между базовыми блоками

Постановка задачи

Определить какие выражения являются доступными для каждого базового блока.

Описание алгоритма

Сначала генерируем множества gen и kill для каждого базового блока. Затем выполняем итерационный алгоритм.

```
// итерационный алгоритм
    foreach B - базовый блок + Выход
    {
        OUT[B] = 0 // инициализируем пустым множеством
    }
        while OUT[B] - меняется
        {
            foreach B - базовый блок do
            IN[B] = U OUT[P], где P -  предок B
            OUT[B] = gen[B] U (x - kill[B]) // x = IN[B]
        }
     }
```

Входные данные

Список базовых блоков

Выходные данные

Список доступных выражений для каждого базового блока

Пример применения

```
AvailabilityAnalysis AA = new AvailabilityAnalysis();
AA.RunAnalysis(blocks);
```

Оптимизация по результатам анализа доступных выражений

Постановка задачи

По результатам анализа доступных выражений выполнить оптимизацию по удалению общих подвыражений.

Описание алгоритма

Производится проход по каждому базовому блоку. Если для базового блока IN[B] есть какие-либо общие подвыражения, то проходим по элементам трёхадресного кода в поиске данных подвыражений. В случае, если мы нашли совпадающее подвыражение, то производим его замену согласно следующему алгоритму (метод `replaceAllOccurences`): Просматривается каждый предшественник Pred базового блока Cur.

a.

Если Gen[Pred] содержит данное подвыражение, ищем его, начиная с последней инструкции. Найдя инструкцию, создаём вспомогательную переменную, проинициализировав её нашим подвыражением и заменяем вхождение данного выражения в блоке-предшественнике.

b. Если же Gen[Pred] не содержит подвыражения, то применяем данный алгоритм для базового блока Pred.

Более подробное описание см. в файле Library/InterBlockOptimizators/CommonExpressions.cs

Входные данные

Граф потока управления

```
ControlFlowGraph cfg;
```

Выходные данные

Работа алгоритма изменяет базовые блоки cfg

Пример использования

```
var cse = new CommonExpressions();
cse.Optimize(blocks);
```

Пример

```
{
  ...
  b = 2 + a;
  if (a)
  {
    b = 2 + a;
  }
  else
  {
    c = a + 2;
  }
  a = a + 2;
  d = a + 2;
}
```

Результат:

```
  ...
  tmp0 = 2 + a;
  b = tmp0;
  if (a)
  {
    b = tmp0;
  }
  else
  {
    c = tmp0;
  }
  a = tmp0;
  d = a + 2;
```

Задача распространения констант между базовыми блоками

Задача распространения констант между базовыми блоками

Постановка задачи

1. Выполнить анализ на основе базового итерационного алгоритма для задачи распространения констант.
2. По результатам анализа выполнить оптимизацию по распространению и свертке констант.

Описание алгоритма

- Элементом данных в данной задаче является отображение `m: Vars -> {NAC, CONSTANT(int), UNDEF}`
- Класс, реализующий распространение констант является наследником базового итерационного алгоритма и предоставляет свою реализацию оператора сбора и передаточной функции.
- Оператор сбора определяется следующим образом:

`m □ m/' = m'' <=> m(v) □ m'(v) = m''(v), для всех v - переменных`

Причем `m(v) □ m'(v)` вычисляется по следующим правилам: `* UNDEF □ v = v * NAC □ v = NAC * CONSTANT(c) □ CONSTANT(c) = CONSTANT(c) * CONSTANT(c1) □ CONSTANT(c2) = NAC`

- Передаточная функция fb(x) определяется следующим образом:

```
s - statement
Если 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) == CONSTAN

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

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

- После построения анализа, для каждого блока формируется список констант, пришедших с других блоков и вызывается внутриблочная функция протягивания и свертки констант

```
foreach (var block in blocks)
{
    Dictionary<IdentificatorValue, int> constants = new Dictionary<IdentificatorValue, int>()
    //...
    ConstantPropagationOptimizator cpo = new ConstantPropagationOptimizator(constants);
    cpo.Optimize(block);
}
```

Входные данные

Список базовых блоков программы.

```
List<BaseBlock> blocks
```

Выходные данные

Работа алгоритма изменяет базовые блоки, поступившие на вход.

Пример использования

```
var constantPropagation = new GlobalConstantPropagation();
constantPropagation.RunAnalysis(blocks);
```

Оптимизация общего итерационного алгоритма

Задача об определении доминаторов

Задача является основой для оптимизаций, связанных с определением доминаторов

Постановка задачи

Эта задача подразумевает определение всех отношений доминирования

Описание алгортима

Изначально для всех узлов (кроме входного, для входного множество пустое) определяем множество out как множество всех узлов и множество in (пустое для всех узлов)

Проходим по всем узлам, в цикле множество in для узла определяется как пересечение множеств out для всех предшественников узла множество out для узла определяется как in объединенное с текущим узлом

Предыдущий цикл выполняется пока множества in и out для узлов изменяются множества out для узлов являются результатом

Входные данные

Список узлов и Входной узел

Выходные данные

Словарь (ключ - узел, значение - список его доминаторов)

Пример применения

Код программы:

```
{
    a := 3;
```

```
   a := 3;
   a := b;
   x := 5;
  □
   if a then
     a := 1;
  □
   while x <= 10 do
   {
     x := x + 1;
   }
}
```

Вывод:

Blocks:

```
Block B0
Ins: []
          a := 3
          a := b
          x := 5
          if a goto %l0
JumpOut: B2
Out: B1


Block B1
Ins: [B0, ]
          goto %l1
JumpOut: B3
Out: B2


Block B2
Ins: [B1, B0, ]
%l0:      nop
          a := 1
JumpOut: □
Out: B3


Block B3
Ins: [B2, B1, ]
%l1:      nop
JumpOut: □
Out: B4


Block B4
Ins: [B3, B6, ]
%l2:      nop
          $t0 := x <= 10
          if $t0 goto %l3
JumpOut: B6
Out: B5


Block B5
Ins: [B4, ]
          goto %l4
JumpOut: B7
Out: B6


Block B6
```

```
Block B6
Ins: [B5, B4, ]
%l3:      nop
          x := x + 1
          goto %l2
JumpOut: B4
Out: B7


Block B7
Ins: [B6, B5, ]
%l4:      nop
JumpOut: □
Out:

D(B1) = {B1}
D(B2) = {B1, B2}
D(B3) = {B1, B3}
D(B4) = {B1, B3, B4}
D(B5) = {B1, B3, B4, B5}
D(B6) = {B1, B3, B4, B6}
D(B7) = {B1, B3, B4, B7}
D(B8) = {B1, B3, B4, B7, B8}
D(B9) = {B1, B3, B4, B7, B8, B9}
D(B10) = {B1, B3, B4, B7, B8, B10}
```

Непосредственные доминаторы, построение дерева доминаторов

Задача является основой для оптимизаций, связанных с определением доминаторов

Постановка задачи

Эта задача подразумевает определение непосредственных доминаторов и построение дерева доминаторов

Описание алгортима

Построение списка непосредственных доминаторов Изначально считаем все доминаторы узлов непосредственными Помечаем на удаление связи удовлетворяющие условию: если список доминаторов доминатора узла отличается от списка доминаторов потомка не на самого потомка

По списку непосредственных доминаторов строится дерево(список непосредственных доминаторов и текущий узел) Строим узел дерева по текущему узлу Для рекурсивного вызова создаем новый список непосредственных доминаторов (в котором нет текущего узла в качестве доминатора) Для каждого элемента из списка непосредственных доминаторов, где текущий узел являтся доминатором вызываем рекурсивно построение потомка для текущего узла

Входные данные

Словарь отношений доминирования (ключ - узел, значение - список его доминаторов), Входной узел

Выходные данные

Список непосредственных отношений доминирования Дерево доминаторов

Пример применения

Код программы:

```
  .. .  .

{
   a := 3;
   a := b;
   x := 5;
   if a then
     a := 1;

   while x <= 10 do
   {
     x := x + 1;
   }
}
```

Вывод:

```
Blocks:
Block B0
Ins: []
          a := 3
          a := b
          x := 5
          if a goto %l0
JumpOut: B2
Out: B1


Block B1
Ins: [B0, ]
          goto %l1
JumpOut: B3
Out: B2


Block B2
Ins: [B1, B0, ]
%l0:      nop
          a := 1
JumpOut: □
Out: B3


Block B3
Ins: [B2, B1, ]
%l1:      nop
JumpOut: □
Out: B4


Block B4
Ins: [B3, B6, ]
%l2:      nop
          $t0 := x <= 10
          if $t0 goto %l3
JumpOut: B6
Out: B5


Block B5
Ins: [B4, ]
          goto %l4
JumpOut: B7
Out: B6
```

```
Block B6
Ins: [B5, B4, ]
%l3:      nop
          x := x + 1
          goto %l2
JumpOut: B4
Out: B7


Block B7
Ins: [B6, B5, ]
%l4:      nop
JumpOut: □
Out:
```

Дерево непосредственных доминаторов:

```
  B1
      B2

      B3
          B4
              B5

              B6

              B7
                  B8
                      B9

                      B10
```

Построение фронта доминирования (DF) и итерационного фронта доминирования (IDF)

Постановка задачи

Фронт доминирования узла n

DF(n) - множество узлов m:

1. n доминирует над p, где p - предшественник m
2. n не доминирует над m или n = m

Итерационный фронт доминирования

S - множество узлов

DF(S) = U DF(x), где x - узел из S

$IDF^1(S) = DF(S)$

$IDF^{i+1}(S) = DF(S \cup DF^i(S))$

```
class DominanceFrontier
{
    public Dictionary<string, HashSet<string>> DF = new Dictionary<string, HashSet<string>>()
    public Dictionary<string, HashSet<string>> IDF = new Dictionary<string, HashSet<string>>(
```

```
    private List<BaseBlock.BaseBlock> blocks;

    public DominanceFrontier(List<BaseBlock.BaseBlock> blocks)
}
```

Описание алгоритма

Вычисление всех DF(x):

```
    foreach n - узел
    {
        if n имеет больше одного предшественника
        foreach p - предшественник n
        {
            r = p
            while r != IDom(n) // IDom(n) - непосредственный доминатор n
            {
                DF(r) += n
                r = IDom(r)
            }
        }
    }
```

Входные данные

Список базовых блоков.

Выходные данные

Объект класса DominanceFrontier, хранящий для каждого базового блока информацию об его фронте доминирования.

Пример использования

```
var block = new BaseBlock();
var blocks = new List<BaseBlock>();
var domFront = new DominanceFrontier(blocks);
//domFront.DF - Dictionary, ключ - имя блока, значение - список узлов, входящих во фронт доми
var IDF = new HashSet<string>();
IDF = domFront.ComputeIDF(block); // итерационный фронт доминирования блока.
IDF = domFront.ComputeIDF(blocks); // итерационный фронт доминирования множества блоков.
```

Отношение доминирования

Построение глубинного остовного дерева

Постановка задачи

Построение глубинного остовного дерева на основе данного графа потока управления, нумерация блоков.

```
class DepthSpanningTree
{
        HashSet<BaseBlock> Visited;
        Dictionary<BaseBlock, int> Numbers;
        BidirectionalGraph<BaseBlock, Edge<BaseBlock>> SpanningTree;
```

```
        public DepthSpanningTree(ControlFlowGraph controlFlowGraph)
}
```

Описание алгоритма

Узлы графа потока управления обходятся в прямом порядке, в процессе прохода они нумеруются.

```
// шаг построения дерева
build(узел n, счетчик c) {
    помечаем узел n как просмотренный
    foreach m - потомок n
    {
        если m не посещен
        {
            добавляем в дерево дугу из n в m
            build(m)
        }
        присваиваем узлу n номер c
        уменьшаем счетчик c на 1
     }
 }

build(корневой узел, кол-во узлов в графе)
```

Входные данные

Граф потока управления.

Выходные данные

Объект класса DepthSpanningTree, хранящий нумерацию, множество посещенных узлов и граф - остовное дерево.

Пример использования

```
var depthSpanningTree = new DepthSpanningTree(cfg); 
// остовное дерево лежит в поле depthSpanningTree.SpanningTree
// нумерация - маппинг между блоком и его номером depthSpanningTree.Numbers
```

Классификация ребёр: наступающие, отступающие, поперечные

Данная задача нужны для определения типов ребер в графе потока управления (далее CFG).

Постановка задачи

Данн CFG. Необходимо классифицировать каждое его ребро. Для решения данной задачи необходимо глубинное островное дерево

Типы ребер

Был создан enum тип для представления типов ребер

```
public enum EdgeType
{
    Unknown = 0,
    Coming = 1,
```

```
    Retreating = 2,
    Cross = 3,
    Backward = 4
}
```

Так же был создан класс, который представляет собой Dictionary, где Key имеет тип Edge, а Value - EdgeType

```
public class EdgeTypes: Dictionary<Edge<BaseBlock>, EdgeType>
{
    public override string ToString()
    {
        return string.Join("\n", this.Select(ed => $"[{ed.Key.Source.Name} -> {ed.Key.Target.
    }
}
```

В класс ControlFlowGraph было добавленно свойство EdgeTypes, которое по каждому ребру в CFG возвращает его тип

```
public class ControlFlowGraph
{
    ...
    public EdgeTypes EdgeTypes { get; set; }
    ...
}
```

Описание алгоритма

1. Строим глубинное островное дерево для CFG
2. Проходим по всем ребрам в CFG и классифицирем их согласно определению

Входные данные

Список базовых блоков

Выходные данные

CFG с классифицированными ребрами

Пример применения

```
var blocks = BaseBlockDivider.divide(linearCode.code);
Console.WriteLine("Edge Types:");
Console.WriteLine(blocks.EdgeTypes);
```

Нахождение обратных рёбер

Постановка задачи

Определить обратные ребра в CFG

Входные данные

Список классифицированных(наступающие, отступающие, поперечные) ребер CFG, дерево доминаторов.

Выходные данные

Список обратных ребер CFG(поле BackwardEdges в классе ControlFlowGraph)

Пример применения

```
ClassificateEdges();
var dominatorsTree = DOM.DOM_CREAT(baseBlocks, baseBlocks.ElementAt(0));
FindBackwardEdges(dominatorsTree);
```

Граф потока управления: проверка приводимости

Экземплярный метод класса CFG

Постановка задачи

Определение Граф потока управления является приводимым, если в глубинном остовном дереве графа любое отступающее ребро является обратным.

По данному графу потока управления проверять, является ли он приводимым или нет.

Описание алгоритма

1. Сконструировать глубинное остовное дерево данного графа потока управления
2. Получить множество обратных рёбер дерева.
3. Получить множество отступающих дерева.
4. Сравнить полученные множества на равенство.

Входные данные

Объект класса CFG

Выходные данные

Логическое значение приводимости CFG

Граф потока управления: выделение естественных циклов

Класс `NaturalLoop` в пространстве имён `OptimizingCompilers2016.Library.ControlFlowGraph`.

Алгоритм построение запускается вызовом конструктора. Конструктор инициализирует поле `loop` типа `HashSet<BaseBlock.BaseBlock>`, значение которого в дальнейшем доступно через свойство `Loop`.

Постановка задачи

Свойства естественных циклов:

1. Цикл должен иметь единственную входную точку, называемую "заголовком" (header). Эта входная точка доминирует над всеми узлами в цикле.
2. Должен быть как минимум один путь итерации цикла, т.е. как минимум один путь назад к заголовку.

По данному графу потока управления проверять, является ли он приводимым или нет.

По данному графу потока управления проверять, является ли он приводимым или нет.

Описание алгоритма

Основная работа происходит в функции `BuildNaturalLoop`, которая работает с обращённым графом потока управления и совершает поиск в глубину (начиная с входного узла обратной дуги) и собирает узлы цикла.

```
public NaturalLoop(ControlFlowGraph CFG, Edge<BaseBlock.BaseBlock> BackEdge) {
    reverseCFG = new ReversedBidirectionalGraph<BaseBlock.BaseBlock, Edge<BaseBlock.BaseBlock
    loop.Add(BackEdge.Target);
    loop.Add(BackEdge.Source);
    BuildNaturalLoop(BackEdge.Source);
}

void BuildNaturalLoop(BaseBlock.BaseBlock Source) {
    foreach (var edge in reverseCFG.OutEdges(Source)) {
        if (!loop.Contains(edge.Target))
        {
            loop.Add(edge.Target);
            BuildNaturalLoop(edge.Target);
        }
    }
}
```

Входные данные

Объект класса `CFG` G и обратная дуга n -> d (значение типа Edge)

Выходные данные

Множество всех узлов естественного цикла (значение типа `HashSet<BaseBlock.BaseBlock>`)

Построение восходящей последовательности областей

Постановка задачи

Необходимо разбить граф потока управления на области и расположить их в порядке вложенности. То есть, если R1 - примитивная область, соответствующая базовому блоку, а R2 непосредственно содержит R1, то R1 должна идти раньше, чем R2.

Области создаются либо на основе базовых блоков, либо на основе циклов. Таким образом, каждый новый уровень иерархии содержит меньше циклов, чем предыдущий.

Логику поиска областей реализует класс RegionHierarchy: Hierarchy - список областей, упорядоченный по возрастанию количества уровней вложенности.

```
public class RegionHierarchy
    {
        /// ascending sequence of regions
        public List<Region> Hierarchy { get; }
    }
```

Виды областей: - базовый класс области (Region) - примитивная область (BaseBlockRegion) - область тела цикла (CycleBodyRegion) - область цикла (CycleRegion)

Базовая область

Содержит информацию о - родительской области - дочерних областях (граф внутренности области)

В данном случае, внутренность области будет хранить области, составляющие тело цикла.

```
public class Region
    {
        public Region ParentRegion { get; set; } /// область, непосредственно содержащая данн
        public BidirectionalGraph HierarchyLevel { get; set; } /// граф непосредственных пото
        public IEnumerable<Edge<Region>> ChildEdges { get { return HierarchyLevel.Edges; } }

        public Region() { HierarchyLevel = new BidirectionalGraph(); }
    }
```

Примитивная область

Область, соответствующая некоторому BaseBlock.

```
public class BaseBlockRegion : Region
    {
        public BaseBlock Block { get; }

        public BaseBlockRegion(BaseBlock block): base() { Block = block; }
    }
```

Область тела цикла

Область, соответствующая телу цикла (без обратного ребра)

CycleBodyStart - заголовок цикла. Тело цикла должно храниться в поле Region::HierarchyLevel

```
public class CycleBodyRegion : Region
    {
        // region that dominates all the other regions in the cycle
        public Region CycleBodyStart { get; }
        public CycleBodyRegion(Region cycleBodyStart) : base()
        {
            CycleBodyStart = cycleBodyStart;
        }
    }
```

Область цикла

Область цикла - это тело цикла + обратное ребро

```
public class CycleRegion : Region
    {
        /// cycle entry
        public Region CycleStart { get; }
        /// Regions, which are the start of reverse edges
        /// TODO: consider removing this - as in this case the only reverse edge is from cycl
        public List<Region> ReverseEdgeSources { get; }
        public List<Edge<Region>> ReverseEdges { get; }
    }
```

Описание алгоритма

Если граф является приводимым, строим первый уровень иерархии - создаем области, соответствующие базовым блокам. Находим обратные ребра, каждому из них соответствует естетсвенный цикл. Если цикл не имеет вложенных циклов, создаем область тела цикла, добавляем ее в иерархию. Создаем область цикла, добавляем ее в иерархию. И так до тех пор, пока циклы не закончатся. Оставшиеся области формируют последний уровень иерархии.

Входные данные

Граф потока данных

Выходные данные

Объект класса REgionHierarchy, хранящий список областей Region, который представляет собой восходящую последовательность областей.

Оптимизированная Версия Итерационного Алгоритма

Постановка задачи

Оптимизация итерационного алгоритма.

Описание алгоритма

1. Проверяем граф потока данных на приводимость
2. Если он неприводим, выход (нельзя проводить оптимизацию)
3. Сортируем блоки так, чтобы для обратных и отступающих дуг номер блока входной блок имел больший номер, чем выходной блок.
4. Запускаем обычный итерационный алгоритм на отсортированных блоках

Данный алгоритм может быть использован только для тех задач, в которых информация распространяется вдоль ациклических путей, то есть для задач: - анализ активных переменных - анализ достигающий определений - анализ доступных выражений

Для задачи распростанения констант данный алгоритм не может быть использован.

Реализация сортировки блоков:

```
// sort blocks
private int CompareBlocks(BaseBlock b1, BaseBlock b2)
{
    if (b1 == b2) return 0;
    return domRelations[b1].Contains(b2) ? 1 : -1;
}
```

Пример использования

```
// sort blocks
 if (useImprovedAlgorithm) □
 {
    domRelations = DOM.DOM_CREAT(blocks, blocks[0]);
    blocks.Sort((b1, b2) => CompareBlocks(b1, b2));
 }
```