1 Введение

Современный мир сильно зависит от различных электронных устройств, к которым мы привыкли и без которых мы уже не мыслим своей жизни. Работа электронных устройств, в свою очередь, сильно зависит от программного обеспечения, которым оно управляется. Поэтому очень актуальной задачей является автоматическая оптимизация исходного кода программы, которая позволяет ускорить скорость выполнения тех или иных программ, что в свою очередь улучшает характеристики электронного устройства.

Стояла задача реализовать программный комплекс, который позволял бы проводить оптимизации над трехадресным кодом программы и компилировать этот трехадресный код в IL-код. IL-код, в свою очередь, должен был быть использован для создания исполняемого файла с помощью соответствующего компилятора, который доступен в платформе *Net*.

В рамках задачи предлагалось реализовать в коде два языка, основанных на различных грамматиках - схожей с языком `C`, и схожей с языком `Pascal`. При этом все различия между грамматиками стирались уже на этапе генерации синтаксического дерева - его структура для обеих грамматик была идентична.

В качестве генераторов синтаксического и лексического анализаторов было решено использовать программы *Yacc* и *Lex*, которые широко используются для подобных целей. Также выбор обусловлен легкостью интеграции данных, генерируемых упомянутыми программами с языком C# - *Yacc* и *Lex* позволяют генерировать код непосредственно для этого языка, и предоставляют для этого удобные механизмы.

Синтаксическое дерево преобразовывалось в трехадресный код, над которым и проводились все оптимизации.

Также в процессе разработки возникла идея интеграции оптимизаций, реализованных в рамках курса, с синтаксическим деревом языка `PascalABC.Net`.

Представленный отчет содержит информацию об использованных или разработанных алгоритмах, которые решали поставленные - а также возникшие в процессе разработки - задачи.

2 Описания языков программирования

2.1 C-подобная грамматика

Реализовано: Циклы (do-while, while), ветвления(if-else), вызовы функций (грамматика самих функций не написана), cout, определение переменных с заданием типа(в одной строке в том числе), грамматика выражений. Циклов for и операций ++ и ей подобных нет.

Данный пример покажет доступные возможности грамматики C.

```
{
    int a = 1, b = 3 * (2 + a);
    cout << 1 << a << endl << 3 << endl;
    some_function_call(a, b, 4);
    while(a < b)
    {
        do
        {
            if (a > 3)
                a = a + 1;
            else a = a - 1;
        }
        while(b < a);
    }
}
```

2.2 Pascal-подобная грамматика

Реализовано: объявление и инициализация переменных различных типов, арифметические выражения, ветвление (if-else), циклы (while-do и repeat-until), write, метки (label, goto), вызов функций.

Пример:

```
begin
    var a : integer := 5.6;
    var b : integer;
    var d : float;
label1:
    b := 89+5*4;
    var c : integer;
    c := 78;
label2:
    d := 23.78;
    goto label1

end.
```

3 Оптимизации

3.1 В пределах базового блока

3.1.1 Генерация трехадресного кода

Общая концепция

Для обхода синтаксического дерева было решено использовать паттерн *Посетитель* (Visitor). Интерфейс для него имеет следующий вид:

```
public interface IVisitor {
  void Visit(BlockNode node);
  void Visit(WhileNode node);
  void Visit(OtherTreeNode node);
  ...
}
```

Количество функций `Visit(..)` этого интерфейса должно покрывать все имеющиеся в дереве узлы.

`BlockNode` является корнем синтаксического дерева, и обход начитается именно с него.

Каждый класс-посетитель дерева должен реализовывать представленный выше интерфейс.

Реализация генератора кода

```
public class Gen3AddrCodeVisitor : IVisitor {
  public iCompiler.ThreeAddrCode CreateCode();
  ...
}
```

В качестве вспомогательной структуры для генерации кода при обходе дерева используется стэк. Запуск обхода происходит путем вызова функции

```
  public void Visit(BlockNode root);
```

аргументом которой является корень сгенерирвоанного ранее синтаксического дерева.

Функции, реализующие обход различных узлов дерева, особого интереса не представляют, и поэтому их описание опускается.

После выполнения обхода всего дерева можно получить трехадресный код с помощью функции `CreateCode()`. При каждом вызове этой функции будет сгенерирован новый экземпляр трехадресного кода (т.е. в визиторе хранятся лишь данные, которые могут быть использованы для этой генерации).

Вывод типов

Типизация переменных происходит в два этапа.

На первом этапе - собственно обход дерева - определяются типы для пользовательских переменных, исходя из их объявления. При попытке использовать пользовательскую переменную, которая не объявлена, выбрасывается исключение (`SemanticException`).

Второй этап происходит после обхода при запросе у визитора трехадресного кода. Выполняется обход имеющегося трехадресного кода, и для временных переменных, тип которых еще не определен, он определяется по следующим правилам: - Всем временным переменным, участвующим в `if`, `while` и т.п. - т.е. являющихся условиями - назначается тип `bool` - Для всех остальных возможных конструкций (унарных операторов, бинарных операторов или тождеств) тип переменной в левой части выводится исходя из таблицы приведения типов. В подобных конструкциях в парвой части на данном этапе будут участвовать либо переменные, тип которых уже определен (либо на первом этапе, либо ранее на этом же), либо числовые значения - в

этом случае сперва выполняется попытка преобразовать это числовое значение в тип `int`, и лишь в случае невозможности такого приведения переменной назначается тип float

Приоритеты типизации: `int` > `float`; `bool` находится несколько особняком

В случае выполнения арифметической операции над переменными разных типов происходит приведение типов к наиболее подходящему (к примеру, `<int> + <float> -> <float>`). Ниже приведена таблица приведения типов (в левой колонке и верхнем ряду - типы операндов бинарного оператора)

	float	int	bool
float	float	float	bool
int	float	int	bool
bool	bool	bool	bool

Пример использования

```
try {
  BlockNode root = FileLoader.LoadFile(file, System.Text.Encoding.UTF8);
  var generator = new Gen3AddrCodeVisitor();
  generator.Visit(root);

  var code = generator.CreateCode();
}
catch (FileNotFoundException) {
  Console.WriteLine("File not found: " + file);
}
catch (LexException e) {
  Console.WriteLine("Lexer error: " + e.Message);
}
catch (SyntaxException e) {
  Console.WriteLine("Syntax error: " + e.Message);
}
catch (SemanticException e) {
  Console.WriteLine("Semantic error: " + e.Message);
}
```

3.1.2 Иерархия классов-строк трехадресного кода

Иерархия наследования

```
Line
|
|--- EmptyLine
|
|--- NonEmptyLine
        |
        |--- FunctionCall
        |    - IsVoid()  // есть ли у функции возвращаемый параметр
        |
```

```
        |--- FunctionParam
        |    - ParamIsNumber()
        |    - ParamIsIntNumber()
        |
        |--- GoTo
        |    |
        |    |--- ConditionalJump
        |         - ConditionIsNumber() // условие - число?
        |
        |--- Expr
             |
             |--- BinaryExpr
             |    - IsBoolExpr() // является ли строка логическим выр-ем
             |    - IsArithmExpr() // является ли строка арфим. выр-ем
             |    - FirstParamIsNumber()
             |    - FirstParamIsIntNumber()
             |    - SecondParamIsNumber()
             |    - SecondParamIsIntNumber()
             |
             |--- UnaryExpr
             |    - IsBoolExpr() // является ли строка логическим выр-ем
             |    - IsArithmExpr() // является ли строка арфим. выр-ем
             |    - ArgIsNumber()
             |    - ArgIsIntNumber()
             |
             |--- Identity
                      - RightIsNumber()
                      - RightIsIntNumber
```

- У каждого наследника `Line` есть поле `label`, котрое определяет метку строки
- У каждого наследника `Line` есть метод `HasLabel()`, который позволяет проверить, есть ли у указанной строки кода метка
- У каждого наследника `Line` есть метод `IsEmpty()`, который позволяет проверить, является ли строка пустым оператором (т.н. nop)
- У каждого наследника `Expr` есть метод `IsEqualRightSide(Expr other)`, который позволяет проверить, равны ли правые части у двух выражений. Этот метод перегружен всеми наследниками Expr (`BinaryExpr`, `UnaryExpr`, `Identity`).

3.1.3 Def-Use информация о переменных

Постановка задачи

Вычисление Def-Use информации о переменных внутри базового блока.

Вход

0. Трехадресный код.
1. Задача решена с помощью метода класса Block, он должен быть создан.
2. Номер строки трехадресного кода.

Выход

1. Возвращает список живых переменных на данной строке
2. Проверяет, жива ли переменная на данной строке

Описание алгоритма

Анализ представлен тремя методами класса Block:

1. `void CalculateDefUseData()`, проходящую по всем строкам 3-х адресного кода данного блока и заполняющую внутреннюю структуру живыми переменными для каждой строки 3-х адресного кода. Метод должен вызываться перед первым вызовом `IsVariableAlive`, `GetAliveVariables`, также требуется повторный вызов, если блок был изменен.

2. `bool IsVariableAlive(string variable, int step)`, возвращает false, если переменная мертва на данном шаге, в противном случае 1.

3. `HashSet<string> GetAliveVariables(int step)`, возвращает множество живых переменных для заданного шага.

Пример использования

```
        block.CalculateDefUseData();
        Hashset<string> alive_vars = block.GetAliveVariables(current_line);
        bool is_alive_a = block.IsVariableAlive("a", current_line);
```

3.1.4 Устранение мертвого кода

Постановка задачи

Удаление мёртвого кода в пределах базового блока.

Описание алгоритма

Код называется мёртвым, если его исполнение не влияет на вывод программы. Резудьтатом вычислений мёртвого кода являются переменные, которые не используются в дальнейшем в программе. Обход строк базового блока производится снизу вверх. Производится проверка: если переменная встретилась впервые, то она заносится в ассоциативный массив в качестве кандидата на мёртвую. Если эта же переменная встретится ещё раз без использования в выражении, то удаляем её, как мёртвую. Если она уже была использована до того, как встретилась во второй раз, то делаем её "живой". Также удаляются выражения вида 'x=x'.

Входные данные

Трёхадресный код, разбитый на базовые блоки.

Выходные данные

Трёхадресный код с применённой оптимизацией.

Пример использования

```
{
```

```
    int a = 2;
    int x = 45, y = 5, t;
    if (1) {
        int k = y;
        y = 2;
        k = 9;
    }

    int i = x;
    int l = 90;
    t = 9;
    y = 9;
    int o = l, e;
    o = 4;
    o = a;
    a = k + i;
    o = a;
}
```

Результат работы

Исходный код:

Результат оптимизации:

3.1.5 Свертка констант и алгебраические тождества

Постановка задачи

Необходимо реализовать алгоритм, осуществляющий свертку найденных в трехадресном коде констант и алгебраических тождеств

Основная идея алгоритма

Основная идея заключена в проходе по блокам треахдресного кода и разборе(преобразовании) каждой строки, содержащей константы или алгебраические тождества, по некоторому набору правил:

1. Сумма, разность, произведение, деление двух чисел заменяется на их результат.
2. Сумма, разность некоторого числа с числом 0 заменяется на само число, с соответствующим знаком.
3. Произведение, деление некоторого числа с числом 0 заменяется на число 0, кроме случая деления на 0.
4. Произведение, деление некоторого числа с числом 1 заменяется на само число, кроме случая деления 1 на данное число.

Входные данные

На вход алгоритму подается трехадресный код, разбитый на базовые блоки

Выходные данные

Трехадресный код, с примененной оптимизацией

Пример использования

```
    ThreeAddrCode code;

    // ...

    ConstantFolding cs = new ConstantFolding(code);
    cs.Optimize();

    // Дальнейшее использование code
```

Пример оптимизации

Начальный код:

```
a = 2 + 3;
b = 4 * 1;
c = 5 + 0;
d = 3 / 1;
```

Результат оптимизации:

```
a = 5;
b = 4;
c = 5;
d = 3;
```

3.1.6 Оптимизация общих подвыражений

Постановка задачи

Устранение общих подвыражений в каждом базовом блоке

Входные данные

На вход алгоритму подается трехадресный код, разбитый на базовые блоки

Выходные данные

Трехадресный код, с применненой оптимизацией

Описание алгоритма

Для работы алгоритма были созданы классы Value, Operation, UnaryOperation. Алгоритм начинается с инициализации словаря, который в ставит в соответствии с идентификатором класс Value. Алгоритм заключается в повторении для каждой строки кода следующих операций: Найти для каждой переменной, соответствующий ей класс Value из словаря, а если такого не найдено, то добавить новый элемент в словарь. Для соответствущей строки кода (Identity, BinaryExpr, UnaryExpr) создается соответствующей ей элемент класса Value (Value, Operation, UnaryOperation). Затем, производится поиск похожего элемента в словаре. Если похожий элемент найден, то

производится переприсваивание переменной в выражении слева нового найденного значения. Иначе элемент класса Value просто добавляется в словарь.

Пример использования

```
    ThreeAddrCode code;

    // ...

    CommonSubexpressionsOptimization cso = new
CommonSubexpressionsOptimization(code);
    cso.Optimize();

    // Дальнейшее использование code
```

Пример оптимизации

Начальный код:

```
a1 = z
a = a1 + a2
b = a1 + a2
c = a1
d = c + a2
```

Результат оптимизации:

```
a1 = z
a = z + a2
b = a
c = z
d = a
```

3.1.7 Протяжка констант

Постановка задачи

Протяжка констант в каждом базовом блоке

Входные данные

На вход алгоритму подается трехадресный код, разбитый на базовые блоки

Выходные данные

Трехадресный код, с примененной оптимизацией

Описание алгоритма

Для реализации алгоритма используется def-use информация о переменных внутри базовых блоков. Идея алгоритма заключается в проходе каждой строки трехадресного кода и поиска константных значений для переменных, учавствующих в данной строке, по следующему алгоритму:

```
1. Фиксирование рассматриваемой строки трехадресного кода.

2. Анализ def-use информации для каждой перменной, учавствующей в операции, и
выбор строки, в которой располагается её константное значение.

3. Протяжка константного значения на место переменной в трехадресном коде.
```

Пример использования

```
    ThreeAddrCode code;

    // ...

    DraggingConstantsOptimization dco = new DraggingConstantsOptimization(code);
    dco.Optimize();

    // Дальнейшее использование code
```

Пример оптимизации

Начальный код:

```
x = 2
y = x + z
c = 5
b = a + x
z = c + x
s = c
```

Результат оптимизации:

```
x = 2
y = 2 + z
c = 5
b = a + 2
z = 5 + 2
s = 5
```

3.1.8 Выделение базовых блоков

Постановка задачи

Выделение базовых блоков

От каких задач зависит, входные данные

Класс трехадресного кода

```
ThreeAddrCode threeAddrCode
```

Алгоритм берет данные из параметра

```
List<Block> blocks
```

Для каких задач нужна, выходные данные

Алгоритм меняет класс 3-х адресного кода, поданного на вход. Изменяется только свойство

```
List<Block> blocks
```

Данный алгоритм является основой практически для все аглоритмов, выполняющих оптимизации как в, так и вне базового блока.

Реализация

Алгоритм реализован в виде единственного статического класса с единственным статическим методом

```
static class BaseBlocksPartition
{
    public static void Partition(ThreeAddrCode threeAddrCode);
}
```

Пример использования:

```
BaseBlocksPartition.Partition(threeAddrCode);
```

Описание(что делает и как реализовано)

- Алгоритм разбивает весь код программы на отдельные базовые блоки
- Код программы подается как первый элемент свойства blocks
- На выходе blocks содержит отдельные базовые блоки в порядке их следования в основном тексте программы
- Алгоритм анализирует все строки кода на наличие команд goto

Команда Альфа

3.2 Построение графа потока управления

Постановка задачи

Построить граф потока управления

От каких задач зависит, входные данные

На вход алгоритму подается список базовых блоков, который хранится в классе 3-х адресного кода

```
IList<Block> blocks
```

Алгоритм использует выхдные данные алгоритма разбиения на базовые блоки

Для каких задач нужна, выходные данные

Граф используется внутри других алгоритмов только для навигации между соседними вершинами графа потока управления

- Получить список всех блоков, в которые есть вход из блока `source`

```
foreach (var block in graph.OutEdges(source)) {...}
```

- Получить список всех блоков, из которых есть вход в блок `sink`

```
foreach (var block in graph.InEdges(sink)) {...}
```

Реализация

```
public class ControlFlowGraph : IGraph<Block>
{
    public ControlFlowGraph(IList<Block> blocks);
    public IEnumerable<Block> OutEdges(Block block);
    public IEnumerable<Block> InEdges(Block block);
}
```

Интерфейс IGraph необходим, т.к. является основой для написания тестовых графов потока управления для других алгоритмов.

Пример использования:

```
var graph = new ControlFlowGraph(threeAddrCode.blocks);
foreach (var block in graph.OutEdges(threeAddrCode.blocks[0])) {}
foreach (var block in graph.InEdges(threeAddrCode.blocks[1])) {}
```

Описание

- Класс хранит граф потока управления для конкретного списка базовых блоков
- Алгоритм мспользует упорядоченность блоков в списке и анализ меток в 3-х адресном коде

Команда Альфа

3.3 Итерационные алгоритмы

3.3.1 Достигающие определения

3.3.1.1 Вычисление множеств gen и kill

Постановка задачи

Генерация Gen-Kill множеств для достигающих определений

Вход

Трехадресный код.

Выход

Список множеств Gen и Kill для каждого блока.

Описание алгоритма

Генерация представлена вызовом метода класса ThreeAddrCode `GetGenKillInfoData()`, возвращающего `List<GenKillInfo>`. Каждый элемент списка представляет собой множества Gen и Kill для соответствующего блока.

Класс `GenKillInfo` содержит два поля `HashSet<Index>` с именами `Gen` и `Kill`, представляющие соответствующие множества для блока.

Класс Index содержит информацию о переменной из множеств `Gen-Kill`, содержит поля `int mBlockInd` с номером блока, `int mInternalInd` - номер строки в блоке, `mVariableName` - имя переменной.

Пример использования

```
  Console.WriteLine("GenKillInfo");
  var a = codeGenerator.Code.GetGenKillInfoData();
  for (int i = 0; i < a.Count; ++i)
  {
        Console.WriteLine("Block: " + i);
        Console.WriteLine("Gen");
        foreach (ThreeAddrCode.Index ind in a[i].Gen)
            Console.WriteLine(ind.ToString());
            Console.WriteLine("Kill");
            foreach (ThreeAddrCode.Index ind in a[i].Kill) {
                Console.WriteLine(ind.ToString());
            }
            Console.WriteLine();
  }
```

3.3.1.2 Передаточная функция

Передаточная функция - это, вообще говоря, отображение значений из одного множества в значения другого множества. Согласно известной теореме:

Суперпозиция передаточных функций определенного вида не выводит из класса таких функций, т.е. является функцией того же вида

Таким образом, передаточная функция реализует интерфейс

```
public interface ITransferFunction<T> {
    IEnumerable<T> Map(IEnumerable<T> x); // отображение
    ITransferFunction<T> Map(ITransferFunction<T> f1); // суперпозиция
}
```

Для большинства задач, связанных с анализом потока данных, передаточная функция имеет вид

```
F(x) = A `UNION` (x - B)
```

Класс TransferFunction, наследующий интерфейс ITransferFunction, при реализации функции `Map(IEnumerable<T> x)` выполняет именно это действие.

Суперпозиция передаточных функций

Как было сказано выше, композиция передаточных функций не выводит из класса передаточных функций. В случае необходимости получить композицию некоторого набора передаточных функций, можно поступить двумя способами:

1. Последовательное применение суперпозиции поочередно ко всем функциям набора.

2. Оказывается, существует формула, которая позволяет бычислить композицию набора передаточных функций. Ниже продемонстрирован вид этой формулы для множеств gen/kill.

```
f[n] . ... . f[2] . f[1] = g `UNION` (x - k), где

k = kill[1] `UNION` kill[2] `UNION` ... `UNION` kill[n]

g = gen[n] `UNION` (gen[n-1] - kill[n]) `UNION`
    (gen[n-2] - kill[n-1] - kill[n]) `UNION ... `UNION`
    (gen[1] - kill[2] - ... - kill[n])
```

3.3.1.3 Реализация алгоритма

Алгоритм, вычисляющий множества IN[block]/OUT[block] для достигающих определений использует существующий модуль итерационного алгоритма. Для его применения требуются две вещи:

- Определить полурешетку.

Полурешетка для данной задачи наследует стандартную полурешетку с оператором сбора - объединением, и верхним элементом - пустым множеством.

```
class ReachableDefSemilattice : UnionSemilattice<ThreeAddrCode.Index>
{

}
```

Более от полурешетки ничего не требуется.

- Определить семейство передаточных функций.

Общий вид передаточных функций базовых блоков для данного алгоритма не отличается от стандартного (A UNION (x - B)). В качестве множества A служит множество gen блока, в качестве множества B - kill.

За генерацию семейства передаточных функций для данной задачи отвечает статическая функция TransferFuncFactory.TransferFuncsForReachDef(code)

3.3.1.4 Тесты

Постановка задачи

Составить набор тестов для проверки работы алгоритма достигающих определений

Определение

Мы говорим, что 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. Мы уничтожаем определение переменной x, если существует иное определение x где-то вдоль пути.

d1 достигает определение d2:

```
    d1 : y := 3
    d2 : x := y
```

d1 не достигает определения d3, т.к. определение d2 уничтожает определение переменной y в d1.

```
    d1 : y := 3
    d2 : y := 4
    d3 : x := y
```

Пример теста

```
{
  int a1 = 1;
  int a2 = 2;
  int a3 = a2;
  do
  {
    a1 = 11;
    int c1 = a2;
    int c2 = a3;
  } while (5 < 1);
}
```

3.3.2 Активные переменные

3.3.2.1 Анализ

Основная идея алгоритма

Большое количество улучшающих преобразований зависит от информации, вычисляемой в направлении, обратном потоку управления программы. Именно к такому типу преобразований относится анализ активных переменных.

Анализ активных переменных состоит в том, что для данной переменной x и точки p мы хотим узнать, может ли значение x в точке p быть использовано вдоль некоторого, начинающегося в p пути графа потока. Если да, то мы говорим, что переменная x активная, или живая в точке p; в противном случае переменная x в точке p неактивна, или мертва.

Определим `IN[B]` как множество переменных, активных в точке непосредственно перед блоком `B`, а `OUT[B]` - как такое же множество в точке, непосредственно следующей за

блоком. Обозначим через DEF[B] множество переменных, которым в блоке B значения присваиваются до их использования, а через USE[B] - множество переменных, значения которых могут использоваться в B до их определения. Тогда уравнения, связывающее DEF и USE с неизвестными IN и OUT, выглядит следующим образом:

```
IN[B] = USE[B]<объединение>(OUT[B] - DEF[B])
OUT[B] = <объединение по P - непосредственным потомкам B>IN[P]
```

Реализация включает отдельную функцию, вычисляющую множества DEF и USE для каждого блока, а также обобщенный итерационный алгоритм, реализующий описанные уравнения и возвращающий множества IN и OUT для каждого блока трехадресного кода.

Входные данные

Граф потока с вычисленными для каждого блока множествами DEF и USE

Выходные данные

Множества IN и OUT для каждого блока

Пример использования

```
ThreeAddrCode code;

// ...

var InOut = DataFlowAnalysis.GetActiveVariables(code);

// ...
```

3.3.2.2 Набор тестов

Постановка задачи

Необходимо реализовать набор тестов для алгоритма анализа активных переменных

Результаты

Было разработано несколько тестов для проверки корректности работы алгоритма анализа активных переменных. Данные тесты расположены среди общего набора файлов для тестирования, а именно в файлах: test-avo-1.cn и test-avo-2.cn.

Пример использования

```
List<string> files = new List<string>();
files.Add(@"..\..\tests\test-avo-1.cn");
files.Add(@"..\..\tests\test-avo-2.cn");

// ...

List<Testing.RootTest> root_tests = new List<Testing.RootTest>();
```

```
// ...

List<Testing.CodeTest> code_tests = new List<Testing.CodeTest>();

// ...

Testing.PerformTests(files, root_tests, code_tests);

// ...
```

3.3.2.3 Оптимизации

Оптимизации с использованием множеств IN/OUT для активных переменных

Алгоритм доступен в классе `ActiveVarsOptimization`. Этот класс является наследником `DeadCodeElimination` и лишь предоставляет для каждого блока заполненный словарь `<переменная>-<живучесть>`, который используется в алгоритме `DeadCodeElimination`. Ддя оригинального алгоритма этот словарь для каждого блока - пустой, т.е. оптимизации осуществляются им только в пределах базового блока.

Учитывая же данные из анализа графа потока управления, которые можно получить с помощью итерационного алгоритма, мы может получить инофрмацию о "живучести" переменной вне текущего блока - таким образом, оптимизация будет более полной.

Определение живучести переменной, основываясь на данных `IN/OUT`

- Все переменные в левых частях выражений в последнем блоке считаются живыми. Это позволяет не удалять **все** переменные программы, потому что они больше не используются

Возможно, в данном случае даже нужно было считать живыми все переменные программы, а не только переменные последнего блока

- Для всех остальных базовых блоков рассматривается множество `OUT` для этого блока: все переменные, которых нет в этом множестве (`OUT`) считаются мертвыми.

3.3.3 Доступные выражения

3.3.3.1 Анализ

Класс `ReachableExprsGenerator` имеет несколько важных статических методов, перечисленных ниже.

Генерация множеств `e.gen-e.kill`

```
Dictionary<Block, ExpressionGenKill> BuildExpressionsGenKill(ThreeAddrCode code)
```

Упомянутая функция принимает на вход трехадреснный код и генерирует множества `e.gen` и `e.kill`, используемые в итерационном алгоритме. Эти множества формируются по следующим правилам:

- множество `e.gen` - множество выражений, которые генерирует базовый блок и далее переменные входящие эти выражения не переопределяются до конца базового блока.
- множество `e.kill` - множество выражений все программы, что переменные входящие в них переопредяются в текущем базового блоке и потом эти выражения не генерируются базовым блоком заново.

Конвертация из внутреннего представления множеств `e.gen-e.kill` в представление, совместимое с остальными решениями

```
InOutData<Line.Expr> BuildReachableExpressionsGenKill(ThreeAddrCode code)
```

(детали реализации)

Генерация семейства передаточных функций:

```
Dictionary<Block, ...> BuildTransferFuncsForReachableExprs(ThreeAddrCode code)
```

Функция в соответствии с общими правилами строит семейство передаточных функций с множествами `e.gen-e.kill` для доступных выражений.

Важно отметить, что применение полученной передаточной функции в итерационном алгоритме осуществляется совместно с полурешеткой, оператором сбора в которой является операция пересечения, а верхним элементом - множество всех выражений программы.

3.3.3.2 Набор тестов

Тесты, демонстрирующие независимость определения множества `e.kill` от взаимной ориентации базовых блоков

```
{
  int a, b, c, d;
  if (1)
  {
    int p = b + c;
  }

  b = a - d;
  int u = b + c;
}
```

Здесь во множество `e.kill` для "верхнего" базового блока попадет выражение `b + c`.

Тесты, демонстрирующие зависимость расположения выражений внутри базового блока:

```
{
  int a, b, c, d;
  if (1)
  {
    int p = b + c;
  }
```

```
  int u = b + c;
  b = a - d;
}
```

Заключение Для тестирование написаны соответствующие методы, которые выводят ответы в доступной форме.

Оба алгоритма (генерация `e.gen-e.Kill` и итерационный алгоритм) проверялись на примерах `test-exprgenkill-1.cn` ... `test-exprgenkill-5.cn`.

3.3.3.3 Оптимизации

Оптимизации с использованием множеств IN/OUT для доступных выражений

Алгоритм доступен в классе `ReachExprOptimization`. Этот класс является наследником `CommonSubexpressionsOptimization` и лишь предоставляет для каждого блока заполненный словарь выражений, который используется в алгоритме `CommonSubexpressionsOptimization`. Для оригинального алгоритма этот словарь для каждого блока - пустой, т.е. оптимизации осуществляются им только в пределах базового блока.

Учитывая же данные из анализа графа потока управления, которые можно получить с помощью итерационного алгоритма, мы может получить инофрмацию о выражениях вне текущего блока - таким образом, оптимизация будет более полной.

Полноту оптимизаций можно увидеть в сравнении с оригинальным алгоритмом `CommonSubexpressionsOptimization` ниже

```
// Оригинал трехадресного кода
      a = b + c
      if 1 goto @l0

      goto @l1

@l0:  d = b + c
      e = b + c
      h = b + c

// Оптимизация с помощью `CommonSubexpressionsOptimization`
      a = b + c
      if 1 goto @l0

      goto @l1

@l0:  d = b + c
      e = d
      h = d

// Оптимизация с помощью `ReachExprOptimization`
      a = b + c
      if 1 goto @l0

      goto @l1
```

```
@l0:  d = a
      e = a
      h = a
```

Можно заметить, что для оптимизации теперь используются и данные из других блоков.

3.3.4 Распространение констант

Постановка задачи

Межблочное распространение констант. Задача распространения констант состоит в передаче информации о текущем значении переменных в следующие блоки. Если на каком-либо участке кода переменная принимает одинаковые значения для всех путей графа потока, то алгоритм должен заменить ее вхождения в правой части выражений на константу.

Вход

Трехадресный код

Выход

Трехадресный код, с примененной оптимизацией

Описание алгоритма

Оптимизация состоит из следующих шагов:

1. Анализ значений всех переменных в левых частях выражений для каждого блока.
2. Оптимизация с использованием информации, полученной на шаге 1.

Алгоритм выполняется до тех пор, пока в коде что-то меняется.

Подробное описание шагов алгоритма

1. Анализ значений переменных. Вход Трехадресный код

Выход Список множеств ConstNACinfo для каждого блока

Описание шага 1

Класс ConstNACinfo содержит в себе 3 поля Type, VName, Value.

Перечислимый тип VariableConstType содержит в себе 3 значения: UNDEFINED, CONSTANT, NOT_A_CONSTANT

VarType - поле перечислимого типа VariableConstType, содержит информацию о состоянии переменной(не определена, константа, не константа) VarName - имя переменной ConstVal - значение переменной, если она является константой.

Метод MaxLowerBound, возвращающий своего рода наибольшую нижнию границу для двух экземпляров класса ConstNACinfo.

Определим: c1 = v1.ConstVal, c2 = v2.ConstVal, c1 != c2;

Тогда результат выполнения метода будет следущим:

v1.VarType.VarType|UNDEF| c2 | NAC | UNDEF |UNDEF| c2 | NAC | c1 | c1 | NAC | NAC | NAC | NAC | NAC | NAC |

Если же c1 = c2, то:
v1.VarType.VarType|UNDEF| c2 | NAC | UNDEF |UNDEF| c2 | NAC | c1 | c1 | c1 | NAC | NAC | NAC | NAC | NAC |

Определим `IN[B]` как множество ConstNACinfo, непосредственно перед блоком `B`, `OUT[B]` - как такое же множество в точке, следующей сразу за блоком `B`.

Пусть `GEN[B]` - множество, состоящее из ConstNACinfo (тип, переменная, значение) для каждой переменной из левых частей выражений блока `B`. В нем учитываются результаты последних присваиваний для каждой из переменных `x`. Если присваивание имеет вид: `x=<константа>`, то переменнаях во множестве `GEN[B]` помечается, как `CONSTANT`, иначе - `NOT_A_CONSTANT`.

Генерация данного множества для каждого из блоков представлена вызовом метода класса ThreeAddrCode `GetConstInfo()`, возвращающего `List<HashSet<ConstNACinfo>>`

Определим `join_operator` как оператор на множестве ConstNACinfo: `X <join_operator> (Y) = X <объединение, которое при совпадении имен берет результат работы MaxLowerBound> (Y)` где `Z` множество элементов `y` из `Y`: существует `x` из `X`: `y`.VarName = `x`.VarName.

Определим `operator` как оператор на множестве ConstNACinfo: `X <operator> (Y) = X <объединение> (Y - Z)` где `Z` множество элементов `y` из `Y`: существует `x` из `X`: `y`.VarName = `x`.VarName.

Тогда уравнения, связывающее `GEN[B]`, `OUT[B]` и `IN[B]` выглядят следующим образом:

```
    IN[B] = <join_operator по P - непосредственным предкам B>OUT[P]
    OUT[B] = GEN[B]<operator>(IN[B])
```

Реализация включает обобщенный итерационный алгоритм, реализующий описанные уравнения и возвращающий множества `IN` и `OUT` для каждого блока трехадресного кода. для проверки полученных множеств можно использовать `TestConstantsPropagation_sets(ProgramTree.BlockNode root)`

Шаг 2 оптимизации

Вход

Трехадресный код, множество `ConstNACinfo` для каждого блока.

Выход

трехадресный код, в котором протянуты и свернуты все известные константы

Описание

В начало каждого блока дописываются все известные для этого блока константы, выполняются оптимизации протяжки и сворачивания констант внутри блока. Затем удаляются добавленные строки.

алгоритм распространения констант завершается, если после шага 2 код не изменился

Тестирование оптимизации распространения констант В проекте реализован класс `ConstantsPropagationOptimization` - наследник `IOptimizer`. Для его работы необходимо передать ему трехадресный код ~~{.cs} ConstantsPropagationOptimization opt = new ConstantsPropagationOptimization(code); opt.Optimize();~~

Пример оптимизации

Начальный код:

{ int x = 0; int y = 0; int z = 0; int p = 5; y = 3; while (x == 5) { x = p + 4; p = 7; while (p == 5) { y = x + y; z = p + 1;
} p = 5; } }

Преобразованный код:

{ int x = 0; int y = 0; int z = 0; int p = 5; y = 3; while (x == 5) { x = 9; p = 7; while (7 == 5) { y = 9 + y; z = 8;
} p = 5; } }

3.3.5 Итерационный алгоритм

Как общеизвестно, итерационный алгоритм для каждого вида оптимизаций имеет похожий вид - различаются лишь конкретные участки. Общая схема этого алгоритма (в видепсевдокода) приведена ниже.

```
OUT[Вход] = <пустое-множество>
Для всех базовых блоков B, кроме `Вход`
  OUT[B] = <верхний-элемент>

Пока OUT[B] по всем базовым блокам B меняются
  Для всех базовых блоков B
    IN[B] = <оператор-сбора> по OUT[P] для всех P - предш. B
    OUT[B] = <передаточная-функция>(IN[B])
```

Верхний элемент и оператор сбора (плюс направление анализа потока данных и область определения) образуют т.н. полурешетку.

Таким образом можно написать обобщенную версию итерационного алгоритма, параметрами которого будут полурешетка и семейство передаточных функций.

Реализация обобщенного алгоритма

```
class IterativeAlgo<AData, TrFunc>
    where TrFunc : ITransferFunction<AData>
{
  public ISemilattice<AData> Semilattice; // полурешетка
  public Dictionary<Block, TrFunc> TransferFuncs; // передаточные функции
```

```
  public void Run(ThreeAddrCode code);
  public void RunOnReverseGraph(ThreeAddrCode code);
```

Здесь `Semilattice` - полурешетка для типа `AData` (т.е. верхний элемент и оператор сбора для этого типа), `TransferFuncs` - набор передаточных функций для каждого базового блока кода.

Функции `Run` и `RunOnReverseGraph` запускают итерационный алгоритм в прямом (сверху вниз) и обратном (снизу вверх) направлениях анализа соответственно.

Стоит отметить, что в рамках курса разработаны варианты полурешеток для различных алгоритмов (к примеру, `IntersectSemilattice`, верхним элементом которой является множество всех значений типа, а оператором сбора - пересечение, или `UnionSemilattice` с верхним элементом - пустым множеством и оператором сбора объединением), а в статическом классе `TransferFuncFactory` можно найти функции для получения семейств передаточных функций для различных типов.

И для полурешеток, и для передаточных функций созданы интерфейся, которые упомятуные ваше экземпляры и реализуют.

```
interface ISemilattice<T> {
  IEnumerable<T> Join(IEnumerable<T> lhs, IEnumerable<T> rhs); // оператор сбора
  IEnumerable<T> Top(); // T - верхний элемент
}

public interface ITransferFunction<T> {
  IEnumerable<T> Map(IEnumerable<T> x);
  ITransferFunction<T> Map(ITransferFunction<T> f1); // композиция передат-х ф-й
}
```

Обратное направление анализа

Как было замечено выше, общая схема итерационного алгоритма однотипна, как для различных оптимизаций - так и для направления обхода графа. Ниже представлена схема алгоритма для обратного направления анализа.

```
IN[Выход] = <пустое-множество>
Для всех базовых блоков B, кроме `Выход`
  IN[B] = <верхний-элемент>

Пока IN[B] по всем базовым блокам B меняются
  Для всех базовых блоков B
    OUT[B] = <оператор-сбора> по OUT[P] для всех P - потомков B
    IN[B] = <передаточная-функция>(OUT[B])
```

Для того, чтобы реализовать возможность анализа с помощью итерационного алгоритма в обратном направлении, используя имеющиеся наработки, было решено просто на вход в этом случае подавать инвертированный граф. Кроме того, после выполнения алгоритма множества IN и OUT меняются местами (учитывая схему алгоритма при обратном обходе).

Запустить алгоритм в обратном направлении анализа можно с помощью функции `RunOnReverseGraph(code)` основного класса `IterativeAlgo`.

3.3.6 Отношение доминирования

3.3.6.1 Итерационный алгоритм

Используется полурешетка с операцией пересечения в качестве оператора сбора. Элементами являются блоки. Верхним элементом полурешетки является множество всех блоков.

Передаточная функция представляет собой множество `Gen` с одним единственным элементом - данным блоком и пустое множество `Kill`.

```
var blockFuncGen = new HashSet<Block>() { block };
funcs[block] = new TransferFunction<Block>(blockFuncGen, emptyKill);
```

На выходе получается набор множеств `Dom` для блоков. По смыслу, каждое множество `Dom` - это множество узлов, которые "доминируют" над данным. Под доминированием подразумевается то, что при обходе графа потока управления каждый элемент из множества Dom будет присутствовать на любом из путей из начала графа в данный блок.

Это хорошо видно на множестве тестов: `test-dom-1.cn` ... `test-dom-2.cn`.

3.3.6.2 Построения графа доминатора

Постановка задачи

Построение графа доминатора(графа, в котором узлы будут связанны отношением доминирования). Говорят, что вершина A доминирует(dom) над B(A dom B), если любой путь в CFG из входа в B проходит через A.

Входные данные

На вход алгоритму подается трехадресный код, разбитый на базовые блоки.

Выходные данные

Граф, в котором базовые блоки будут связаны отношением доминирования. Либо специальная структура(DomTree), которая предоставляет интерфейс для работы с графом доминатора.

Описание алгоритма

Для построения графа доминатор используется выходные данные итерационного алгоритма вычисления dom множеств для базовых блоков. Сам алгоритм построения состоит из двух основных частей: 1. Опеределение корневого блока. 2. Обход всех dom множеств блоков и дальнейшее построение дерева.

Пример использования

```
Поддерживается дальнейшая возможность работы с dom графом напрямую используя
функцию GenerateDomTree:
```

```
    ThreeAddrCode code;

    // ...

    Dictionary<Block, List<Block>> tree = DomGraph.GenerateDomTree(code);

    // Дальнейшее использование tree
```

Либо возможность воспользоваться специальным классом для dom графа DomTree, который реализует интерфейс IDominatorRelation<Block>:

```
    ThreeAddrCode code;

    // ...

    DomTree domTree = new DomTree(code);
    Block blockA = code.blocks[0];
    Block blockB = code.blocks[1];

    // ...

    domTree.FirstDomSeccond(blockA, blockB); // = blockA dom blockB

    domTree.UpperDominators(blockA); // = List<Block>

    domTree.DownDominators(blockA); // = List<Block>
```

Пример работы алгоритма

Трехадресный код:

```
Block 1 :
        i1 = 1
        if 1 goto @l0

Block 2 :
        goto @l1

Block 3 :
@l0:    i3 = 3
        if 3 goto @l2

Block 4 :
        goto @l1

Block 5 :
@l2:    iu = 0

Block 6 :
@l1:    <empty statement>
```

Результат работы алгоритма:

```
Dom Tree Algorithm

    Block(1) <===> Childs: 2; 3; 6;
    Block(2) <===> Childs:
    Block(3) <===> Childs: 4; 5;
    Block(4) <===> Childs:
    Block(5) <===> Childs:
    Block(6) <===> Childs:

Dom Tree Class

    1 dom 1 = True
    1 dom 2 = True
    1 dom 3 = True
    1 dom 4 = True
    1 dom 5 = True
    1 dom 6 = True
    2 dom 1 = False
    2 dom 2 = True ...

    UpperDominators(1) = ( 1 )
    UpperDominators(2) = ( 1 2 )
    UpperDominators(3) = ( 1 3 )
    UpperDominators(4) = ( 1 3 4 )
    UpperDominators(5) = ( 1 3 5 )
    UpperDominators(6) = ( 1 6 )

    DownDominators(1) = ( 1 6 3 5 4 2 )
    DownDominators(2) = ( 2 )
    DownDominators(3) = ( 3 5 4 )
    DownDominators(4) = ( 4 )
    DownDominators(5) = ( 5 )
    DownDominators(6) = ( 6 )
```

3.4 Анализ графа для алгоритма выделения областей

3.4.1 Обход в глубину с нумерацией

см. **Построение остовного дерева**

3.4.2 Построение остовного дерева

Постановка задачи

Построение остовного дерева

От каких задач зависит, входные данные

На вход алгоритму подаются:

- Список базовых блоков

```
IEnumerable<T> blocks
```

- Граф потока управления

```
IGraph<T> graph
```

Для каких задач нужна, выходные данные

Выходные данные алгоритма:

- Номера блоков

```
tree.Numbers
```

- Остовное дерево

```
tree.Data
```

Реализация

Шаблон типа необходим для проведения тестирования алгоритма с помощью типа int, но алгоритм предназначен для использования с типом Block.

```
public abstract class SpanningTree<T>
{
    public SpanningTree(IEnumerable<T> blocks, IGraph<T> graph);
    public Dictionary<T, int> Numbers { get; protected set; }
    public Dictionary<T, List<T>> Data { get; protected set; }
}
public class SpanningTreeWithoutRecursive<T> : SpanningTree<T>
{
    public SpanningTreeWithoutRecursive(IEnumerable<T> blocks, IGraph<T> graph)
            : base(blocks, graph) { }
}
```

Пример использования:

- Инициализация

```
SpanningTree<Block> tree = new SpanningTreeWithoutRecursive<Block>(code.blocks,
code.graph);
```

- Получить номер блока b (от 0 до n-1)

```
int i = tree.Numbers[b];
```

- Получить всех потомков блока b в остовном дереве tree

```
List<Block> blocks = tree.Data[b];
```

Описание

- Алгоритм строит и хранит произвольное остовное дерево графа и нумерацию блоков при обходе в глубину
- Алгоритм обходит в глубину граф потока управления

Тесты

Для прогона тестов необходимо вызвать

```
SpaningTreeTesting.Test();
```

Команда Альфа

3.4.3 Классификация рёбер

Постановка задачи

Необходимо реализовать алгоритм, осуществляющий классификацию ребер графа потока управления. Ребра классифицируются по двум признакам: (прямые, обратные) и (наступающие, отступающие, поперечные).

Основная идея алгоритма

Основная идея заключена в анализе связей и отношений доминирования между вершинами графа. В зависимости от этих признаков ребро относится к тому или иному классу по следующему набору правил:

1. Если вершина A доминирует над вершиной B, то ребро (A,B) называется прямым ребром.
2. Если вершина B доминирует над вершиной A, то ребро (A,B) называется обратным ребром.
3. Если вершина A является предком вершины B, то ребро (A,B) называется наступающим ребром.
4. Если вершина A является потомком вершины B, то ребро (A,B) называется отступающим ребром.
5. Если ребро (A,B) не является ни наступающим, ни наступающим ребром, то ребро (A,B) называется поперечным ребром.

Входные данные

- Остовное дерево графа

```
SpanningTree<T> spanningTree
```

- Коллекция, содержащая отношения доминирования

```
Dictionary<T, IEnumerable<T>> Dom
```

Выходные данные

Список ребер заданного типа. Каждому типу ребер соответствует свой метод.

Пример использования

- Инициализация

```
GraphEdges<Block> graphEdges = new GraphEdges<Block>(spanningTree, blockDoms);
```

- Получить список наступающих ребер

```
List<DomGraph.ValPair<Block>> listEdges = graphEdges.AdvancingEdges() as
List<DomGraph.ValPair<Block>>;
```

- Получить список отступающих ребер

```
List<DomGraph.ValPair<Block>> listEdges = graphEdges.RetreatingEdges() as
List<DomGraph.ValPair<Block>>;
```

- Получить список поперечных ребер

```
List<DomGraph.ValPair<Block>> listEdges = graphEdges.CrossingEdges() as
List<DomGraph.ValPair<Block>>;
```

- Получить список прямых ребер

```
List<DomGraph.ValPair<Block>> listEdges = graphEdges.StraightEdges(blockDoms) as
List<DomGraph.ValPair<Block>>;
```

- Получить список обратных ребер

```
List<DomGraph.ValPair<Block>> listEdges = graphEges.ReversedEdges(blockDoms) as
List<DomGraph.ValPair<Block>>;
```

Тесты

Для запуска теста необходимо вызвать

```
Testing.TestGraphEdges(ProgramTree.BlockNode root)
```

3.4.4 Определить обратные рёбра в CFG

Постановка задачи

Определить обратные рёбра в графе потоков управления.

Описание алгоритма

Ребро из A в B называется обратным, если B доминирует над A. Выполняется обход вершин в словаре отношения доминирования. Если существует ребро из A в B, и B доминирует над A, то эти вершины заносим в список рёбер.

Входные данные

Остовное дерево CFG и отношения доминирования в виде словаря.

Выходные данные

Список обратных рёбер.

Пример использования

```

```

Результат работы

Исходный код:

Результат оптимизации:

3.4.5 Определить приводимость графа

Постановка задачи

Определить, является ли граф потоков управления приводимым.

Описание алгоритма

CFG является приводимым, если все его отступающие рёбра являются обратными. Проверяем, совпадают ли списки обратных и отсутпающих рёбер.

Входные данные

Списки отсутпающих и обратных рёбер CFG.

Выходные данные

Булево значение, является ли граф приводимым.

Пример использования

```
```

3.4.6 Определение всех естественных циклов

Постановка задачи

Определение всех естественных циклов

От каких задач зависит(на каких задачах основывается), входные данные

На вход алгоритму подаются:

- Список базовых блоков

```
IEnumerable<T> blocks
```

- Граф потока управления

```
IGraph<T> graph
```

- Список обратных дуг

```
List<DomGraph.ValPair<T>> reverseEdges
```

- Дерево доминатора

```
IDominatorRelation<T>
```

Данный алгоритм основывается на задачах, которые занимаются построением соотвествующих структур данных, за исключением IDominatorRelation, который предоставляется как интерфейс:

```
    // Интерфейс для дерева доминатора
    public interface IDominatorRelation<T>
    {
        // Отношение доминирования первой вершины над второй
        bool FirstDomSeccond(T a, T b);

        // Все вершины, которые доминируют над текущей
        IEnumerable<T> UpperDominators(T a);
    }
```

Для каких задач нужна(для каких задач является основой), выходные данные

Выходные данные алгоритма - список циклов:

```
List<Cycle>
```

Данная задача является основой для задачи "Определение вложенности естественных циклов"

Реализация(интерфейс т.е. классы, методы)

Шаблон типа необходим для проведения тестирования алгоритма с помощью типа int, но алгоритм предназначен для использования с типом Block.

Данный алгоритм реализуется классами:

1. AllCycles< T > - находит все циклы типа CycleUsual (цикл типа CycleSpecialCase распознается как 2 цикла типа CycleUsual)

```
public class AllCycles<T> where T : IComparable<T>
{
     public List<Cycle<T>> cycles { get; protected set; }
     public AllCycles(IEnumerable<T> blocks, IGraph<T> graph,
     List<DomGraph.ValPair<T>> reverseEdges,
            IDominatorRelation<T> domTree);
}
```

2. AllCyclesSpecialCase< T > - находит все циклы типа CycleUsual и CycleSpecialCase

```
public class AllCyclesWithSpecialCase<T> : AllCycles<T>
where T: IComparable<T>
{
     public AllCyclesWithSpecialCase(IEnumerable<T> blocks,
                IGraph<T> graph,
     List<DomGraph.ValPair<T>> reverseEdges,
            IDominatorRelation<T> domTree);
}
```

Тип Cycle< T > является абстрактным классом цикла, который реализуется 2 классами:

```
// Класс цикла
public abstract class Cycle<T>
{
    // Вход в цикл
    public T N { get; set; }

    // Все вершины, принадлежащие циклу
    public List<T> DATA { get; set; }

    // Ребра - выходы из цикла
    public List<DomGraph.ValPair<T>> OUTS { get; set; }
}

// Цикл с одной обратной дугой
public class CycleUsual<T>: Cycle<T>
```

```
    {
        public CycleUsual(T n, List<T> data,
                List<DomGraph.ValPair<T>> outs, T d);

        // Вершина из обратного ребра
        public T D { get; set; }
    }

    // Цикл с двумя обратными дугами
    public class CycleSpecialCase<T> : Cycle<T>
    {
        public CycleSpecialCase(T n, List<T> data,
        List<DomGraph.ValPair<T>> outs, T d1, T d2);

        // Вершина из первого обратного ребра
        public T D1 { get; set; }

        // Вершина из второго обратного ребра
        public T D2 { get; set; }
    }
```

Других видов циклов не бывает.
List< Cycle< T > > - полиморфный контейнер, который хранит в себе 2 вида циклов.

Пример использования:

```
    //Входные данные
    List<Block> block;
    ControlFlowGraph graph;
    List<DomGraph.ValPair<Block>> reverseEdges;
    DomTree domTree;
    //...
    //Инициализация
    AllCycles<Block> allCyclesSpec =
        new AllCyclesWithSpecialCase<Block>(blocks, graph,
                    reverseEdges, domTree);
    AllCycles<Block> allCycles =
        new AllCycles<Block>(code.blocks, code.graph,
                reverseEdges, domTree);
    //Получить все циклы с учётом "специального случая"
    List<Cycle<Block>> cyclesSpec = allCycles.cycles;
    //Получить все циклы без учёта "специального случая"
    List<Cycle<Block>> cycles = allCycles.cycles;
```

Описание(что делает и как реализовано)

- Алгоритм ищет все циклы в графе потока управления.
- Считает граф потока управления приводимым.
- Ищет все циклы в графе вне зависимости от их вложенности.

- Алгоритм раелизован с помощью рекурсивного обхода вглубину по напрвлению дуг, обратных графу потока управления. Обход начинается с каждой точки выхода из цикла, и заканчивается обходом всего цикла.

Тесты(если есть и если не являются другой задачей)

Для прогона тестов необходимо вызвать

```
    AllCyclesTesting.TestingAllCycles();
```

Команда Альфа

3.4.7 Вложенность естественных циклов

Постановка задачи

Построить иерархию вложенности всех естественных циклов графа

От каких задач зависит, входные данные

На вход алгоритму подаетсятся список циклов

```
List<Cycle<T>> list
```

Основывается на задаче определения всех естественных циклов

Для каких задач нужна, выходные данные

Девево циклов

```
public Dictionary<Cycle<T>, List<Cycle<T>>> data { get; private set; }
```

Корень дерева циклов

```
public List<Cycle<T>> root { get; private set; }
```

Используется алгоритмом выделения областей

Реализация

Шаблон типа необходим для проведения тестирования алгоритма с помощью типа int, но алгоритм предназначен для использования с типом Block

```
public class AllCyclesHierarchy<T>
{
    public AllCyclesHierarchy(List<Cycle<T>> list);
    public List<Cycle<T>> root { get; private set; }
    public Dictionary<Cycle<T>, List<Cycle<T>>> data { get; private set; }
}
```

Описание

- Алгоритм строит иерархию вложенности циклов
- Использует только данные о вершинах, принадлежащих циклам
- Каждый цикл присутствует в дереве только один раз

- Если цикл вложен сразу в несколько циклов, то данный цикл становиться потомком самого наиболее удалённого от корня предка
- Корень может состоять из нескольких вершин

Тесты

Для прогона тестов использовать сл. метод:

```
AllCyclesHierarchyTesting.Test()
```

Команда Альфа

3.4.8 Глубина CFG

Постановка задачи

Необходимо реализовать алгоритм, осуществляющий нахождение глубины графа потока управления

Основная идея алгоритма

Основная идея заключена в проходе по графу потока управления и нахождении самого длинного пути от узла входа к узлу выхода без учета циклов. Для нахождения самого длинного пути заводится специальный счетчик, который увеличивается при переходе к узлу потомка. Реализация такого прохода включает в себя рекурсивный обход графа потока управления по каждой из его возможных ветвей и нахождение максимального значения счетчика из полученных.

Входные данные

На вход алгоритму подаются базовые блоки трехадресного кода и граф потока управления

Выходные данные

Числовое значение, соответствующее длине максимального пути

Пример использования

```
    ThreeAddrCode code;

    // ...

    int depth = DepthFinder<Block>.FindDepth(code.blocks, code.graph);

    // ...
```

3.5 Алгоритм выделения областей

3.5.1 Выделение областей

Постановка задачи

Выделение областей трехадресного кода

Входные данные

На вход алгоритму подается трехадресный код

Выходные данные

Ссылка на область, соответствующую первому блоку

Описание алгоритма

Алгоритм состоит из следующих шагов:

1. Для каждого типа областей реализован класс Region
2. Для работы алгоритма используется определение всех естественных циклов.
3. Изначально каждый блок представляется в виде простой области, т.е. один блок образует простую область;

3.1. Строится первоначальный граф;

3.2. На основе выходных данных, полученных из "определение всех естественных циклов", происходит сворачивание графа, начиная с самых вложенных циклов.

Пример использования

```
    ThreeAddrCode code;

    // ...

    Region first = Region.RegionsDetermination(code);
```

Пример работы

Исходный трехадресный код:

```
      i = 0

@l2:  @t0 = i < 10
      if @t0 goto @l0

      goto @l1

@l0:  j = 0

@l5:  @t1 = j < 10
      if @t1 goto @l3

      goto @l4

@l3:  @t2 = i * j
```

```
      param @t2
      param endl
      call cout, 2
      j = j + 1
      goto @l5

@l4:  i = i + 1
      goto @l2

@l1:  <empty statement>
```

Тестовое представление областей:

```
Block 0
Cycle {
  Block 1
  Block 3
  Cycle {
    Block 4
    Block 6
  }
  Block 5
  Block 7
}
Block 2
Block 8
```

3.5.2 Реализация алгоритма на основе областей

В представленной версии ПО данный алгоритм реализован не был.

4 Сопутствующие задачи

4.1 GUI

Постановка задачи

Разработать графический пользовательский интерфейс

Описание

- Позволяет редактировать текст (исходный код)
- Имеет возможности по сохранению, загрузке, созданию файлов с исходными кодами
- Имеет возможность выводить оптимизированный код и IL-код
- Работает с текстовыми файлами с расширениями `.cn` (C-синтаксис) и `.pasn` (Pascal-синтаксис)
- Реализована возможность загрузки кода PascalABC.Net (`.pas`) и применения оптимизаций к нему (с преобразованием в трехадресный код и обратно) с последующим выводом временной оценки оптимизированного и не оптимизированного вариантов.

Реализация

Реализован с использованием **API Windows Forms C#**

Для обработки входных данных используется подпроект **iCompiler** и **PascalABC**.

4.2 Генерация IL-кода

Постановка задачи

Построение IL-кода по данному трехадресному коду

Входные данные

На вход алгоритму подается трехадресный код

Выходные данные

IL-код

Описание алгоритма

Алгоритм заключается в повторении для каждой строки кода следующей операции: В зависимости от типа строки генерируется участок IL-кода, соответствующий строке трехадресного кода.

Пример использования

```
    ThreeAddrCode code;

    // ...

    string ILCode = ILCodeGenerator.Generate(code);
```

Пример генерации

Исходный трехадресный код:

```
    i = 0

@l2:    @t0 = i < 10
    if @t0 goto @l0

    goto @l1

@l0:    param i
    param endl
    call cout, 2
    i = i + 1
    goto @l2

@l1:    <empty statement>
```

Сгенерированный IL-код:

```
.assembly extern mscorlib {}
.assembly program {}

.method static public void main() cil managed
{
    .entrypoint
    .maxstack 2
    .locals init (int32 i, bool @t0)

    ldc.i4 0
    stloc i
@l2:
    ldloc i
    ldc.i4 10
    clt
    stloc @t0
    ldloc @t0
    brtrue @l0
    br @l1
@l0:
    ldloc i
    call void [mscorlib]System.Console::Write(int32)
    call void [mscorlib]System.Console::WriteLine()
    ldloc i
    ldc.i4 1
    add
    stloc i
    br @l2
@l1:
    nop
    ret
}
```

4.3 Документация

Постановка задачи

Разработка шаблона для документации

Описание

Все компоненты документации находяться в папке Documentation в корне проекта

Документация представляет из себя дерево из папок и файлов сл. формата:

1. файл: номер.заголовок.txt (параграф)
2. папка: номер.заголовок (раздел)

Особенности

- Структуру документации можно менять, добавляя, удаляя папки и файлы, и изменяя номера соотвествующих папок и файлов

- Для собрания документации в единый текстовый файл необходимо вызвать программу DocumentationAssembler.exe. Эта программа:

1. Распологает текст из файлов в порядке их следования (по номерам), учитывая вложения
2. Добавляет заголовоки с номерами всех разделов (пр.: 1.4.2 Реализация RDV)
3. Учитывает глубину вложенности заголовка и добавляет перед ним соответствующее кол-во решеток (# для 0ого уровня, ## для 1ого, и т.д.) для соответсвия markdown
4. Добавляет пустую строку после файла
5. Корневые разделы документации распологать в папке Documentation
6. Результат в файле OptimizedCompilersProjectDocumentation.txt

Файлы/папки в одном разделе с одинаковыми номерами не использовать

Затем необходимо использовать утилиту для форматирования текстов в зависимости от формата файлов документции. В данном случае используется консольная утилита pundoc, которая поддерживает расширенный формат markdown.

Готовый для использования скрипт, который собирает, форматирует и конверитует текст в формат DOCX, находиться в файле toDOCX.bat

Реализация

Исходный код пргграммы DocumentationAssembler.exe находиться в проекте https://github.com/chekist131/DocumentationAssembler

Программа DocumentationAssembler.exe строит структуру данных дерево, состоящую из 2х типов вершин:

1. Параграфов
2. Разделов

Особенности

- Разделы могут включать в себя как параграфы, так и разделы
- Параграфы являются листьями.
- Обход данного дерева осуществляется в глубину в порядке их нумерации
- Для реализации использовался паттерн 'visiter'

Команда Альфа

4.4 Подсчёт статистики

Результаты

- Статистика по контрибьюторам и командам на 20.12.15

- Учитываются изменения, добавленные только после 28.09.15 включительно

TEAMS

Commits

No	Commiter	Count
1	TrueDevelopers	169
2	Альфа	66
3	DoubleK	35
4	Source_Code	35
5	Стрела в колено	33
6	Ultimate	25
7	Академ(	8

Additions + Deletions

No	Commiter	Count
1	TrueDevelopers	84998
2	DoubleK	34396
3	Source_Code	25286
4	Альфа	7390
5	Стрела в колено	2381
6	Ultimate	1258
7	Академ(	201

AVG Additions + Deletions per commit

No	Commiter	Count
1	DoubleK	982
2	Source_Code	722
3	TrueDevelopers	502
4	Альфа	111
5	Стрела в колено	72
6	Ultimate	50
7	Академ(	25

STUDENTS

Commits

No	Commiter	Count
1	Михайличенко	107
2	Валяев	51
3	Леонтьев	49
4	Гаджиев	27

5	Гончаров	23
6	Проскуряков	20
7	Гулканян	15
8	Батраков	13
9	Пыслару	13
10	Троицкий	13
11	Тухиков	12
12	Гончаров+Пак	12
13	Папиж	8
14	Рыбин	8

Additions + Deletions

No	Commiter	Count
1	Михайличенко	48208
2	Леонтьев	31228
3	Гаджиев	18078
4	Папиж	16318
5	Гончаров+Пак	14742
6	Гончаров	10544
7	Валяев	6555
8	Пыслару	5562
9	Проскуряков	1607
10	Гулканян	835
11	Батраков	804
12	Троицкий	774
13	Тухиков	454
14	Рыбин	201

AVG Additions + Deletions per commit

No	Commiter	Count
1	Папиж	2039
2	Гончаров+Пак	1228
3	Гаджиев	669
4	Леонтьев	637
5	Гончаров	458
6	Михайличенко	450
7	Пыслару	427
8	Валяев	128

9	Проскуряков	80
10	Батраков	61
11	Троицкий	59
12	Гулканян	55
13	Тухиков	37
14	Рыбин	25

Описание

Программа для сбора статистики по проекту

- Её можно найти по ссылке https://github.com/chekist131/GitHubProjectStatistic
- Использовалась библиотека octokit , реализующая доступ к GitHub через GitHub web api для платформы .NET
- Для многократного запуска программы требуется доступ с логином и паролем (ввод через аргументы программы, либо в исходном коде)
- Программа анализирует статистику по разработчикам и командам для данного проекта (количество добавлений, удалений и коммитов)
- Выводит полученные значения в виде таблиц в формате markdown

4.5 Интеграция с PascalABCNet

Суть интеграции заключалась в том, чтобы реализовать возможность преобразования синтаксического дерева **PascalABC.Net** в трехадресный код, выполнении оптимизаций над этим трехадресным кодом и обратной конвертации преобразованного трехадресного кода в синтаксическое дерево **PascalABC.Net**.

Предварительно синтаксичческое дерево подвергается так называемому lowering'у - замене высокоуровневых конструкций типа `while`, `repeat` и т.п. на простые конструкции, содержащие лишь операторы `goto`. Сложных `if` (к примеру, содержащие много строк в теле `then` или `else`) также упрощаются.

Для выполнения поставленной задачи было реализовано два класса - генератор трехадресного кода и генератор синтааксического дерева `PascalABC.Net`.

Реализация

Для генерации трехадресного кода из синтаксического дерева используется класс

```
class Gen3AddrCodeVisitor : WalkingVisitorNew {
  public iCompiler.ThreeAddrCode CreateCode();
}
```

Метод `CreateCode();` генерирует трехадресный код (который и возвращает) из данных, полученных в процессе выполнения обхода дерева. Также именно в этой функции создается таблица имен.

Для генерации синтаксического дерева из трехадресного кода реализован класс `PascalABCTreeGenerator`:

```
public class PascalABCTreeGenerator {
  public block Generate(iCompiler.ThreeAddrCode code);
}
```

Стоит отметить, что при использовании делегата `compiler.SyntaxTreeChanger`, который и выполняет преобразование над синтаксическим деревом, для корректного использования вышеописанного генератора синтаксического дерева, последовательность действий внутри основной функции `Change(..)` этого делегата должна быть следующей:

```
public class SyntaxTreeChanger : ISyntaxTreeChanger {
  public void Change(syntax_tree_node sn) {
    sn.visit(new LoweringVisitor());
    ...
    var visitor = new Gen3AddrCodeVisitor();
    sn.visit(visitor);

    try {
      Code = generator.CreateCode();
      ...
      // тут выполняем оптимизации, если необходимо
      ...
      var generator = new PascalABCTreeGenerator();
      var program_block = generator.Generate(Code);
      (sn as program_module).program_block = program_block;
      ...
    }
  }
}
```

Нереализованные возможности

В настоящий момент трехадресный код преобразуется в синтаксическое дерево "как есть". В процессе разработки была идея реализовать сворачивание трехадресных выражений в полное выражение; например, преобразовать

```
@t1 = 2 + 3
a = 2*@t1
```

в

```
a = 2 * (2+3)
```

и уже выражение в таком виде помещать в синтаксическое дерево. Необходимый функционал был реализован. Однако позже мы пришли к выводу, что в процессе оптимизаций строка `@t1 = 2 + 3`, которую после подобного сворачивания нужно было удалить (чтобы не проводить лишних вычислений), может быть использована кодом ниже, и подобное удаление не возможно. Это лишь часть из проблем, с которыми мы столкнулись при решении этой задачи.

Таким образом, задача по подобным преобразованиям является не такой уж и тривиальной, и от нее было решено отказаться по причине наличия более приоритетных задач.

Заключение

Генерация трехадресного кода поддерживает стандартные синтаксические конструкции (if, while, repeat, for) и вызовы функций (к примеру, write/writeln/read/readln).

Для тестирования результатов оптимизаций в GUI данного проекта есть возможность загрузить код в файле `.pas` и увидеть скорость работы оптимизированного и неоптимизированного вариантов.

5 Заключение

В рамках курса был разработан программный комплекс, который позволяет генерировать из синтаксических деревьев двух представленных грамматик трехадресный код. Для этого трехадресного кода реализован ряд различных оптимизаций, действующих как в пределах базового блока, так и вне его - т.е., учитывающих информацию из всего программного кода.

Для удобства демонстрации доступных возможностей реализован дружелюбный GUI, который имеет возможность редактирования исходного кода, компиляции его как в трехадресный и IL-код, так и исполняемый файл, а также позволяет осуществлять вручную выполнение доступных оптимизаций над трехадресным кодом.

В полном объеме была произведена интеграция с компилятором `PascalABC.Net` - т.е. преобразование синтаксического дерева этого компилятора в трехадресный код, выполнение оптимизаций и преобразование этого кода обратно в синтаксическое дерево. Пункт загрузки файлов с исходным кодом `PascalABC.Net` (`.pas`) также доступен в GUI. При этом выводится информация о сравнении скорости выполнения оптимизированного и не оптимизированного исходных кодов для данного языка - как и сам вид оптимизированной программы.

К сожалению, одна из задач - а именно задача об анализе на основе областей - реализована не была. Во многом этому способствовало то, что слишком много команд участвовало в написании кода для решения этой задачи. Это значительно усложнило решение этой, вообще говоря, реализуемой в рамках курса задачи. Т.е., по нашему мнению, лучший результат мог бы быть достигнут при назначении полного объема работ по этой задаче одной, максимум двух командам, плотно взаимодействующим друг с другом.

Несмотря на этот неприятный факт, считаем, что команда успешно выполнила поставленные перед ней задачи.