Содержание

1 Введение

Система программирования PascalABC.NET - это реализация языка Object Pascal, сочетающая простоту языка Паскаль и огромные возможности платформы .NET. Система активно развивается и активно используется для обучения студентов и школьников основам программирования.

В апреле 2011 года вышла версия PascalABC.NET 1.7. Основным нововведением стало реализация директив OpenMP, что позволило параллельно выполнять циклы и секции. Директивы OpenMP являются простым механизмом для преобразования последовательной программы в параллельную, доступным для понимания начинающих программистов. Но дело в том, что не всегда применение этих директив приводит к эквивалентной программе. Результаты работы программы при последовательном и параллельном выполнении могут отличаться. Распараллеливанию могут мешать различные информационные зависимости, которые порождаются вхождениями переменных. Была поставлена задача: реализовать элементы автоматического распараллеливания в компиляторе PascalABC.NET, которые позволяют анализировать информационные зависимости внутри гнезда циклов и применять различные преобразования с целью получения эквивалентной параллельной программы.

При анализе программы с целью оптимизации (распараллеливания) удобно рассматривать данные об информационных зависимостях этой программы в виде графов. Существуют как более грубые способы описания зависимостей программы, например, граф информационных зависимостей, так и более тонкие, например, решетчатый граф и развертка решетчатого графа. В этой работе в качестве модели информационной зависимости был использован решетчатый граф. Решетчатый граф – одна из наиболее тонких моделей информационной зависимости. Решетчатый граф в частности позволяет для любой операции в правой части оператора присваивания

указать итерацию и генератор, выход которого является аргументом данной операции.

Решетчатый граф строится для линейного класса программ[1, с. 18]. Многие программы могут быть приведены к линейному классу с помощью различных преобразований [3, параграф 8.1]. Например, если в программе встречается произведение внешних переменных, то это произведение можно заменить новой внешней переменной. Если у некоторого оператора цикла шаг не равен +1, то с помощью канонизации циклов [33, 49] можно получить программу, в которой этот цикл имеет шаг +1. Согласно статистике [3, с. 341], с помощью различных приемов, можно привести к линейному классу 90–95% текста практически используемых программ. Рассмотрение программ, не принадлежащих линейному классу, выходит за рамки данной работы.

В рамках этой работы автором реализовано построение решетчатых графов для гнезд распараллеливаемых циклов в компиляторе PascalABC.NET. Эти графы строится на этапе синтактико-семантического преобразования для каждого из циклов, помеченных директивой OpenMP. На основе решетчатых графов определяются ParDO циклы, и, если распараллеливаемый цикл не принадлежит к их числу, выдается предупреждение, что существуют циклически порожденные зависимости, мешающие распараллеливанию. Кроме того, на основе решетчатых графов автором было реализовано унимодулярное распараллеливающее преобразование двумерных гнезд циклов для компилятора PascalABC.NET.

Работа состоит из 10 глав. В главах 1 и 2 описывается общая проблематика задачи. Глава 3 рассказывает об информационной зависимости в программе и модели рассматриваемых программ. В главе 4 подробно рассматриваются различные решетчатые графы, а так алгоритм построения элементарного решетчатого графа. Глава 5 описывает построение расширенной информации о вхождениях переменных. Глава 6 описывает под-

ключение целочисленной библиотеки параметрического программирования PipLib к компилятору PascalABC.NET. Глава 7 посвящена программной реализации построения решетчатых графов. Автоматическое распознавание ParDo циклов по решетчатому графу описано в главе 8. В этой же главе рассказывается о программной реализации этого распознавания. Глава 9 посвящена преобразованию циклов с помощью унимодулярных матриц и его реализации в компиляторе PascalABC.NET. Глава 10 описывает реализацию распараллеливания двумерных гнезд тесно вложенных циклов с помощью унимодулярного преобразования.

2 Постановка задачи

Была поставлена задача, включающая в себя следующее:

- построение расширенной информации о вхождениях переменных в теле распараллеливаемого цикла;
- формирование входных данных для библиотеки параметрического программирования PipLib;
- подключение библиотеки PipLib к компилятору PascalABC.NET;
- построение решетчатых графов для фрагмента программы с помощью библиотеки PipLib для компилятора PascalABC.NET;
- определение ParDo циклов по решетчатым графам для фрагмента программы;
- распараллеливание гнезд из двух тесно вложенных циклов с помощью унимодулярного преобразования.

3 Информационная зависимость в программе

Определение. Вхождением переменной будем называть всякое появление переменной в тексте программы вместе с тем местом в программе, в котором эта переменная появилась [1].

Не умаляя общности, все переменные, кроме счетчиков циклов, можно считать индексными. Всякому вхождению при конкретном значении индексного выражения соответствует обращение к некоторой ячейке памяти.

Если при этом обращении меняется состояние ячейки (вхождение в левую часть оператора присваивания, которое не входит в индексное выражение другого вхождения), то такое вхождение называется *генератором*. Остальные вхождения называются *использованиями*.

Пример 1. В следующем операторе присваивания

A[B[i + 1]]:=C[2*i - 2] + A [i + 2];

v1 v2 v3 v4

четыре вхождения v1, v2, v3, v4 и только вхождение v1 является генератором.

Конец примера 1.

Определение. Два вхождения информационно-зависимы, если обращаются к одной и той же ячейке памяти при некоторых допустимых значениях индексных выражений.

Информационные зависимости делятся на 4 типа:

1) выходная зависимость, если оба вхождения являются генераторами:

   X:=…

X:=…

2) потоковая или истинная зависимость, если первое вхождение является генератором, а второе вхождение – использованием:

X:=…

… :=X

3) анти зависимость, если первое вхождение является использованием, а второе вхождение – генератором:

... :=X

X:=…

4) входная зависимость, если оба вхождения являются использования:

... :=X

… :=X

Пример 2.

A := B + C;

v1 v2 v3

B := A + C;

v4 v5 v6

B := 2;

v7

В данном программном сегменте имеются следующие отношения зависимости:

1) потоковая зависимость использования v5 от генератора v1;
2) анти зависимость генератора v4 от использования v2;
3) выходная зависимость генератора v7 от генератора v4
4) входная зависимость использования v6 от использования v3

Конец примера 2.

Об информационных зависимостях можно прочитать, например, в [8].

3.1 Классификация зависимостей внутри цикла

Определение.([4]) Зависимость называется циклически порождённой, если оба вхождения обращаются к общей ячейке памяти на разных итерациях цикла.

Определение.([4]) Зависимость называется циклически независимой, если оба вхождения обращаются к одной и той же ячейке памяти на одной и той же итерации цикла.

Пример 3.

```
for var i:=0 to n do
  begin
    A[i]:= X[i];
    u1     u2
    X[i+1]:= A[i]-A[i-1];
    u3       u4    u5
  end;
```

Зависимость, которую порождают вхождения u1 и u4, – циклически независимая. Зависимости, которые образуют вхождения u2 и u3, u1 и u5 – циклически порожденные.

Конец примера 3.

*Условие распараллеливания цикла***:**

Допускаются только циклически независимые зависимости.

3.2 Модель рассматриваемых программ

Построение решетчатых графов и многие преобразования, на них основанные, возможны в рамках линейного класса [3, с. 340].]. Для полноты изложения опишем этот класс. Это класс программ, включающий в себя следующие элементы языка:

- скалярные и индексные переменные;
- операторы присваивания, правая часть которых является арифметическим выражением;

• все повторяющиеся операции описываются только с помощью циклов for языка программирования Pascal; структура вложенности циклов может быть произвольной; шаги изменения параметров циклов всегда равны +1(в случае языка программирования Pascal это всегда так); если у цикла нижняя граница больше верхней, то цикл не выполняется;

• условные и безусловные операторы перехода, передающие управление «вниз» по тексту; не допускается использование побочных выходов из циклов;

• все индексные выражения переменных, границы изменения параметров циклов и условия передачи управления задаются аффинными формами от совокупности параметров циклов и внешних переменных программы; все коэффициенты при переменных в этих формах являются целыми числами;

• внешние переменные программы всегда целочисленные.

Программы, принадлежащие этому классу, называются линейными.

Определение. Внешние переменные фрагмента программы – переменные, значения которых в этом фрагменте не изменяются и не могут быть определены на момент компиляции программы.

Пример 4. Следующий фрагмент принадлежит линейному классу программ:

```
for var i:=0 to N do
begin
  a[N-i+1] := 2*b[3*i];
end;
```

Конец примера 4.

Пример 5. Следующий фрагмент не принадлежит линейному классу программ:

```
for var i := 0 to N do
    for var j := 0 to a[i] do
    begin
      a[i + 2 * j] := b[i * j];
```

```
end;
```

Конец примера 5

.

по следующим причинам:

1. условие окончания работы второго цикла не аффинное;

2. индексное выражение массива b относительно счетчиков циклов не является аффинным.

В частности, следует отметить, что программы нахождения максимального элемента массива или сортировки не принадлежат линейному классу. Многие оптимизируемые (требующие больших объемов вычислений) участки программ классических численных методов принадлежат линейному классу или сводятся к таким. Не относятся к линейному классу программы для метода конечных элементов, поскольку матрицы в этих случаях разреженные и работа с ними предполагает косвенную адресацию.[4]

В защиту линейного класса программ заметим, что большинство оптимизирующих/распараллеливающих преобразований программ, как правило, носят локальный, а не глобальный характер. А это означает, что требования линейности предъявляются не ко всей программе, а только к оптимизируемому фрагменту.

Последний аргумент в пользу исследований линейного класса программ состоит в том, что традиционный граф информационных связей, который основан на неравенствах Банерджи или Омега-тесте, строится тоже только для линейного класса. [4]

4 Решетчатый граф, как модель информационной зависимости в гнездах циклов

При анализе программы с целью оптимизации (распараллеливания) удобно рассматривать данные об информационных зависимостях этой программы в виде графов. Существуют как более грубые способы описания зависимостей программы, например, граф информационных зависимостей, так и более тонкие, например, решетчатый граф и развертка решетчатого графа. В этом параграфе будут рассмотрены решетчатые графы.

Решетчатый граф – одна из наиболее тонких моделей информационной зависимости. Решетчатый граф в частности позволяет для любой операции в правой части оператора присваивания указать итерацию и генератор, выход которого является аргументом данной операции. Описание решетчатых графов в этом параграфе большей частью основано на [1,3,4].

4.1 Опорные пространства

Рассмотрим некоторый фрагмент программы из линейного класса. Занумеруем операторы циклов в этом фрагменте сверху вниз по тексту программы. Обозначим счетчик i-го цикла – I_i.

Определение. ([1]) Множество всех вложенных друг в друга циклов, в теле каждого из которых содержится некоторый оператор *Stmt*, называют *опорным гнездом циклов* для оператора *Stmt* и для всех вхождений, которые этот оператор содержит.

Определение. Множество значений вектора счетчиков циклов опорного гнезда, при которых исполняется оператор *Stmt*, будем называть *опорным пространством* для данного оператора и всех вхождений переменных, которые он содержит.

Далее будем считать, что в каждой строке программы находится только один оператор. Занумеруем все операторы присваивания фрагмента

программы сверху вниз по тексту программы и обозначим их через $Stmt_1$, $Stmt_2$, …, $Stmt_m$

Обозначим опорное пространство оператора $Stmt_i$ через V_i .

Определение. ([1]) *Пространством итераций фрагмента программы* назовем совокупность опорных пространств V_i для *i*=1, 2, …, *m*.

Пример 6. Рассмотрим гнездо циклов:

```
for var i1 := L1 to U1 do
    for var i2 := L2 to U2 do
    ...
      for var in := Ln to Un do
        loopbody(i1,i2,...,in);
```

Пространство итераций этого многомерного цикла (11) – это множество всех таких целочисленных векторов:

$$I = \{(I_1, I_2, \dots, I_n): L_1 \le I_1 \le U_1, L_2 \le I_2 \le U_2, \dots L_n \le I_n \le U_n\}$$

Границы L_i, U_i изменения счетчиков циклов I_i могут линейно зависеть от счетчиков объемлющих циклов. В этом случае пространство итераций многомерного цикла является целочисленным многогранником в n-мерном пространстве.

Конец примера 6.

Определение. Пусть имеются векторы $\mathrm{I} = (I_1, I_2, \dots I_{n1})$, и $\mathrm{J} = (J_1, J_2, \dots J_{n2})$. Пусть $m = \min(n_1, n_2)$. Тогда:

1. если существует $\mathrm{k} \in [1, \mathrm{m}]$, что $I_1 = J_1$, $I_2 = J_2$, $I_{k-1} = J_{k-1}$, $I_k < J_k$, то будем говорить, что вектор I *лексикографически меньше* вектора J. Будем обозначать этот факт $\mathrm{I} < \mathrm{J}$;
2. если $n_1 = n_2$ и для любого $\mathrm{k} \in [1, \mathrm{m}]$ $I_k = J_k$, то будем говорить, что вектор I *лексикографически равен* вектору J, и обозначать $\mathrm{I} = \mathrm{J}$; [1]

Пример 7.

Пусть $I = (1,2,3,8,5)$ и $\mathrm{J} = (1,2,3,4,5)$. Вектор J будет лексикографически меньше вектора I, поскольку при совпадающих первых трех коор-

динатах, третья координата вектора J строго меньше, третьей координаты вектора I. Значения всех последующих координат уже не важны.

Конец примера 7.

При последовательном выполнении гнезда циклов, точки пространства итераций просматриваются в лексикографическом порядке – это вытекает из семантики оператора цикла языка программирования Pascal.

4.2 Максимальные и минимальные решетчатые графы

Определение. ([1]) Множество вершин *максимального решетчатого графа* – все точки пространства итераций фрагмента программы. Дуга направляется из вершины $I \in V_i$ в вершину $J \in V_j$, если $I < J$, и существуют такие вхождения $u \in Stmt_i$ и $v \in Stmt_j$, что $u[I]$ и $v[J]$ обращаются к одной и той же ячейке памяти. При этом говорят, что вхождение u определяет начало дуги, а вхождение v – конец дуги.

В зависимости от того, чем являются вхождения u и v (генератором или использованием), различают максимальные решетчатые графы потоковой, выходной, анти - и входной зависимости.

Определение *минимальных решетчатых графов* дадим конструктивно ([1]).

Возьмем максимальный решетчатый граф зависимости типа *T*. Для каждой вершины *s* этого графа выполним:

1. разобьем множество дуг, входящих в вершину *s* на группы: к одной группе отнесем дуги, конец которых определялся одним и тем же вхождением;
2. в каждой группе выберем одну дугу, у которой начальная вершина лексикографически ближе всего к вершине s; остальные дуги удалим.

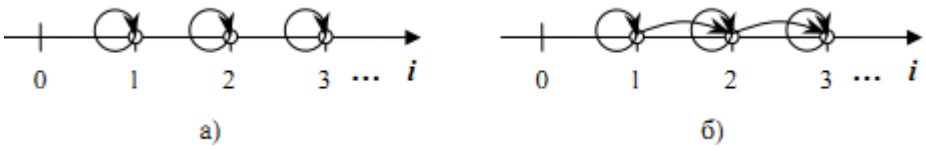
Полученный граф называется *минимальным снизу решетчатым графом* зависимости типа T.

Для зависимостей по выходу и по входу минимальные снизу и сверху решетчатые графы совпадают [3, с. 347]. В общем случае для анти- и потоковой зависимости минимальные сверху и снизу решетчатые графы различаются [3, с. 347].

Пример 8. Рассмотрим следующий фрагмент программы:

```
for var i := 1 to 10 do
  begin
    y[i] := x[i + 1];
            u1
    z[i] := x[i];
            u2
    x[i] := 5;
    u3
  end;
```

В этом цикле имеется анти зависимость вхождения u3 от вхождений u1, u2.Других анти зависимостей нет. Часть минимального снизу решетчатого графа анти зависимости для данного цикла изображена на рисунке 1.а, минимального сверху решетчатого графа анти зависимости изображена на рисунке 1.б.



а) б)

Рис.1. Решетчатые графы анти зависимости а) минимальный снизу и б) минимальный сверху.

Из приведенного примера видно, что эти графы не совпадают.

Конец примера 8.

В компиляторе PascalABC.NET автором реализовано построение минимальных сверху и минимальных снизу решетчатых графов.

4.3 Элементарные и полные решетчатые графы. Способы хранения

Определение решетчатого графа истинной зависимости переменной.([4])Пусть задан фрагмент программы и некоторая переменная (м.б. индексная) *X*. Множество вершин решетчатого графа истинной зависимости переменной – это пространство итераций заданного фрагмента. Две точки пространства итераций *I* и *J* соединяются дугой, если

1. в графе информационных связей существует дуга истинной зависимости *(u1, v1)*, решетчатый граф которой содержит дугу *(I,J)*;

2. для любой другой дуги *(u2, v1)* истинной зависимости графа информационных связей, решетчатый граф которой содержит некоторую дугу *(I2,J)* , входящую в вершину *J*, точка пространства итераций *I2* лексикографически меньше *I*, либо *I2 = I*, но вхождение *u2* в тексте программы встречается всегда раньше вхождения *u*.

Аналогично определяются решетчатые графы переменной для анти зависимости, выходной зависимости и входной зависимости.

Определение (полного) решетчатого графа фрагмента программы.([4]) Решетчатый граф фрагмента программы – это граф, вершинами которого являются все точки пространства итераций, а множество дуг является объединением множеств дуг решетчатых графов всех переменных данного фрагмента.

Из определения решетчатого графа вытекает следующее свойство:

- в каждую вершину элементарного решетчатого графа входит не более одной дуги.

Из этого свойства вытекает возможность хранения решетчатого графа в виде отображения подмножества пространства итераций в пространство итераций: концу дуги решетчатого графа сопоставляется начало этой дуги.

Традиционные способы хранения графов в виде матрицы смежностей или списка дуг не приемлемы для решетчатых графов. Действительно, для гнезд циклов реальных долго считаемых программ количество дуг решетчатого графа может быть так велико, что прочтение всех дуг может потребовать время, сравнимое со временем последовательного выполнения исходной программы, и распараллеливание, основанное на таком графе, потеряет смысл.

Функция, описывающая решетчатый граф, может занимать объем, не зависящий от размерностей циклов, описывающих пространство итераций. Эта функция является *кусочно-квазилинейной*. *«Квази-линейная»* - это значит, что кроме линейных операций над счетчиками циклов эта функция допускает взятие целой части числа или условные операторы. Во многих практически важных случаях эта функция оказывается кусочно-линейной[4, с. 83].

4.4 Построение решетчатого графа

Все минимальные решетчатые графы строятся на основе элементарных решетчатых графов. Поэтому здесь будет детально рассмотрен процесс построения элементарного решетчатого графа. Для упрощения изложения будем считать, что рассматриваемые фрагменты программ не содержат внешних переменных. Материал этого параграфа в значительной степени опирается на работы [1,с.68],[3, с.363].

Пусть вхождение u содержится в гнезде из n циклов, пространство итераций которого V_1. Вхождение v – в гнезде из m циклов, пространство итераций которого V_2. Пусть оператор, содержащий вхождение u находится раньше по тексту программы, чем оператор, содержащий вхождение v. Обозначим через $P(I)$ вектор, координатами которого являются индексы вхождения u, через $Q(J)$ – вектор индексов вхождения v.

Рассмотрим задачу построения элементарного решетчатого графа [1, с. 69], описывающего зависимость вхождения *v* от вхождения *u*. Зафиксируем некоторую вершину $J \in V_2$. Чтобы из вершины $I \in V_1$ выходила дуга в вершину $J \epsilon V_2$, необходимо выполнение условий:

$$P(I) = Q(J)\,, \qquad (1)$$

$$I \in V_1, J \in V_2, \qquad (2)$$

$$I \leq_{lex} J\,. \qquad (3)$$

Система (1) при условиях (2), (3) может иметь неединственное решение. На множестве всевозможных решений I из (1), (2), (3) нужно найти то решение I_o, которое является лексикографически максимальным (lex.max). Такое решение

$$I_0 = lex.\max I \qquad (4)$$

является единственным.

Для того чтобы решить указанную задачу, сначала нужно свести условия (1),(2), (3) к совокупности систем линейных неравенств. Условия $P(I) = Q(J)$ и $I \in V_1, J \in V_2$ сводятся к системе линейных неравенств как обычно в линейном программировании. Равенство $P(I) = Q(J)$ заменяется на два линейных неравенства:

$$P(I) \geq Q(J)$$

$$-P(I) \geq -Q(J)$$

Условия $I \in V_1, J \in V_2$ в случае цикла задаются его границами в виде линейных неравенств.

Рассмотрим подробно, как сводится к совокупности систем линейных неравенств условие $I \leq_{lex} J$. Пусть два указанных вхождения *u* и *v* имеют *d* общих циклов. Системы неравенств, к которым сводится условие $I \leq_{lex} J$, называются *альтернативными многогранниками* [3, с. 365]. Первая из этих систем имеет следующий вид:

$$I_1 = J_1;$$

$$I_2 = J_2; \qquad (5)$$

…

$$I_d = J_d$$

и рассматривается только в случае, если оператор, содержащий вхождение u находится раньше по тексту программы, чем оператор, содержащий вхождение v. Все остальные альтернативные многогранники рассматриваются во всех случаях. Второй многогранник имеет вид:

$$I_1 = J_1;$$

$$I_2 = J_2; \qquad (6)$$

…

$$I_d < J_d$$

В каждом следующем многограннике количество равенств уменьшается на единицу, неравенство всегда одно. Последний многогранник будет таким:

$$I_1 < J_1; \qquad (7)$$

Занумеруем все альтернативные многогранники. Пусть количество равенств в системе, описывающей некоторый альтернативный многогранник, равно m. Тогда номер этого многогранника положим равным $m+1$. Заметим, что номера альтернативных многогранников изменяются от 1 до $d+1$. По построению, любая точка I любого альтернативного многогранника удовлетворяет условию $I \leq_{lex} J$. Лексикографически ближе к J будет та точка, которая находится в альтернативном многограннике с большим номером [3, с. 366].

Подчеркнем, что если в решетчатом графе существует дуга из вершины I в вершину J, то компоненты указанных векторов удовлетворяют соотношениям (1), (2) и одной из систем линейных неравенств, определяющей некоторый альтернативный многогранник.

Задача (1) – (4) решается по следующей схеме. Организуется цикл, со счетчиком по k, в котором выполняются нижеследующие действия,

причем, если оператор, содержащий вхождение *u* находится по тексту программы раньше, чем оператор, содержащий вхождение *v*, то *k* изменяется от *d+1* до *1*, иначе *k* изменяется от *d* до *1*:

1. Система (2) и условия (3) сводятся к системе линейных неравенств, к которой дописывается набор линейных неравенств, определяющих альтернативный многогранник с номером *k*. В полученной системе есть набор неизвестных – компоненты вектора *I*, и набор параметров – компоненты вектора *J*.
2. Далее во множестве решений указанной системы находится лексикографический максимум относительно неизвестных с помощью метода параметрического целочисленного программирования [5]
3. Если лексикографический максимум найден, то выписываются функции, определяющие дуги решетчатого графа, и области их определения. Уменьшить *k* на единицу и перейти на пункт 1.

Таким образом, из алгоритма построения решетчатого графа видно, что каждой функции *Fk* можно поставить в соответствие единственный номер *s* альтернативного многогранника, который использовался при ее построении. Системе неравенств, которая определяет альтернативный многогранник с номером, будут удовлетворять компоненты каждой пары векторов *(I, J)*, для которой выполняется *I=Fk(J)*. Этот факт используется при определении *ParDo* циклов в программе[1].

Отметим, что решение задачи (1)-(4) дает в результате минимальный снизу элементарный решетчатый граф. Для построения минимального сверху элементарного решетчатого графа нужно решить следующую задачу[3, с. 372]:

$$P(I) = Q(J)\,, \qquad (8)$$

$$I \in V_1, J \in V_2, \qquad (9)$$

$$I \leq_{lex} J\,. \qquad (10)$$

$$I_0 = lex.\min I \quad (11)$$

Соответственно из-за последнего условия (11) в альтернативных многогранниках знаки неравенств поменяются на противоположные.

Стоить отметить, что для потоковой зависимости функции решетчатого графа по концу дуги возвращают ее начало, а для анти зависимости – по началу дуги возвращают ее конец.

5 Построение расширенной информации о вхождениях переменных

Для построения решетчатых графов необходимо иметь информацию обо всех вхождениях переменных в данный фрагмент программы. При этом производится упрощение выражений с помощью приведения подобных и раскрытия скобок. Для этих целей был описан класс – визитор по синтаксическому дереву.

5.1 Паттерн посетитель (визитор)

Визитор(посетитель) – это паттерн проектирования. Он обходит дерево и выполняет дальнейшее преобразование программы. Дерево лишь хранит информацию о программе. Посетители (визиторы) задают алгоритм преобразования, это позволяет использовать синтаксическое дерево для многих целей.

Все узлы синтаксического дерева наследуются от базового класса syntax_tree_node

```
public class syntax_tree_node
{
    public syntax_tree_node();
    public syntax_tree_node(SourceContext source_context);
    public SourceContext source_context { get; set; }
    public virtual void visit(IVisitor visitor);
}
```

Для реализации механизма визиторов в каждом узле дерева определен метод:

```
public virtual void visit(IVisitor visitor);
{
     visitor.visit(this);
}
```

Для обхода дерева необходимо написать класс визитор, в котором будет переопределена функция visit для всех типов узлов

```
class OccurCollectorSyntaxVisitor: IVisitor
{
     public void visit(syntax_tree_node _tree_node)
          {}
          …
```

```
}
```

5.2 Реализация визитора для построения расширенной информации о вхождениях переменных

Для сбора информации о вхождениях переменных был реализован класс **OccurCollectorSyntaxVisitor** – наследник **WalkingVisitor**. **WalkingVisitor** – это реализация интерфейса **IVisitor**, которая производит обход переданного на вход синтаксического поддерева. В классе **OccurCollectorSyntaxVisitor** переопределен обход узлов, необходимых для построения информации о вхождениях переменных в теле цикла.

```
class OccurCollectorSyntaxVisitor: WalkingVisitor
{
      /// <summary>
      /// Хранит генераторы внутри цикла
      /// </summary>
      public Dictionary<string, List<Occurrence>> Generators;
      /// <summary>
      /// Хранит использования внутри цикла
      /// </summary>
      public Dictionary<string, List<Occurrence>> Uses;
      /// <summary>
      /// текущее множество счетчиков цикла
      /// </summary>
      private HashSet<string> cur_Iterators;
      /// <summary>
      /// Множество счетчиков цикла
      /// </summary>
      public HashSet<for_node> Iterators;
      /// <summary>
      /// Список ссылок на циклы
      /// </summary>
      private List<for_node> cur_loops;

            ...
}
```

После того как визитор обойдет поддерево, он возвращает генераторы и использования, которые были встречены внутри тела цикла, в полях **Generators** и **Uses**. В этих ассоциативных массивах, по имени переменной возвращается список всех ее вхождений. Класс **Occurrence**, хранящий информацию о вхождении переменной, будет описан позднее. Кроме того, в

поле **Iterators** хранятся ссылки на все операторы циклов внутри данного гнезда циклов.

В этом классе переопределены методы **visit** для узлов синтаксического дерева **for_node** и **assign**. При их обработке как раз и заполняются поля **Generators** и **Uses**.

Стоит отметить, что логические условия условных операторов не обрабатываются. Предполагается, что обе ветки могут сработать. Также не допускается вызов функций в теле цикла, так как рассматривается только линейный класс программ.

5.3 Структуры данных для хранения информации о вхождении переменной

Для хранения информации о вхождении переменной был описан класс **Occurrence**. Он может быть представлен UML-диаграммой (Рис.2):

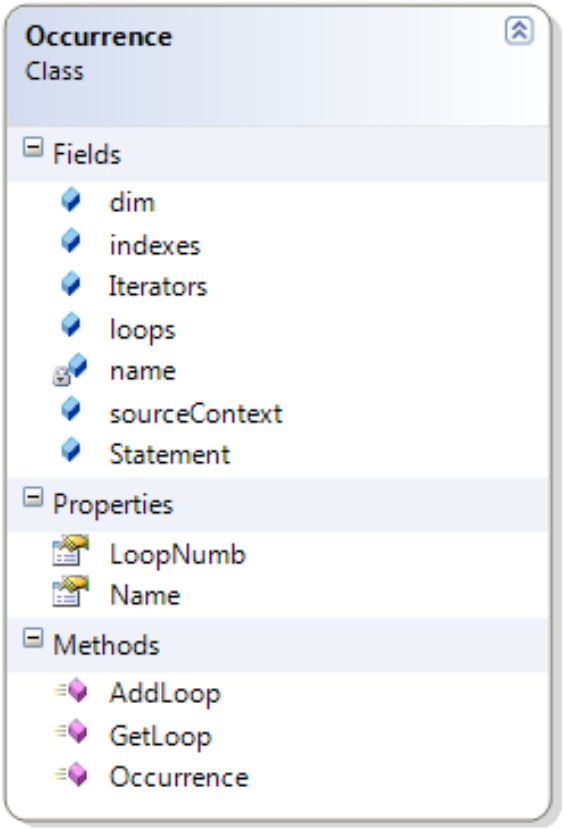


Рис. 2. UML-диаграмма класса Occurrence

Рассмотрим поля, свойства и методы этого класса.

dim – это размерность переменной. Для скалярной переменной это поле равно нулю.

indexes –это поле хранит индексные выражения переменной и представляет собой список ассоциативных массивов. Количество элементов этого списка равно размерности переменной. Каждый элемент – ассоциативный массив, представляет собой отдельное индексное выражение массива. В этом массиве хранятся коэффициенты при каждом счетчике цикла, коэффициенты при внешних переменных и значение свободного члена.

Пример 9. Рассмотрим вхождение переменной

a[2*i+3*j +5*n+3,i+j+1]

В этом случае поле **indexes** выглядит, как это показано на рис. 3.

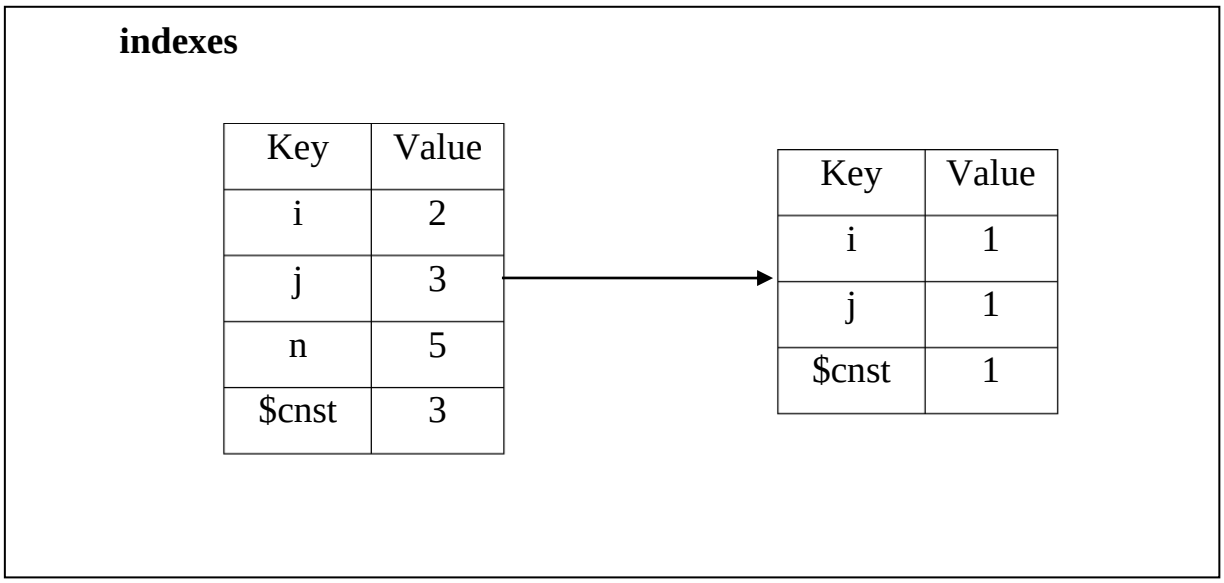


Рис. 3. Вид разобранного индексного выражения

Конец примера 9.

iterators – множество имен счетчиков циклов, внутри которых находится данное вхождение.

loops – список ссылок на операторы циклов данного гнезда.

name – имя переменной.

sourceContext – положение переменной в тексте программы. Хранится строка и столбец начала и конца вхождения переменной.

LoopNumb – количество циклов for, в которые вложена данная переменная.

Name – это свойство возвращает значение приватного поля **name**.

Этот класс содержит конструктор и методы для добавления и получения заданного цикла.

5.4 Упрощение выражений

При обходе синтаксического поддерева с помощью визитора **OccurCollectorSyntaxVisitor** перед созданием нового экземпляра класса **Occurrence** происходит упрощение индексных выражений путем раскрытия скобок и приведения подобных. Для этих целей служит класс **ProcessIndexer**. Он получает на вход узел синтаксического дерева **indexer**, который соответствует вхождению переменной с индексами.

Этот класс имеет следующий интерфейс:

```
class ProcessIndexer{
    private indexer ind;// обрабатываемый индексер
    public int dim;// размерность массива
    public string name;// имя массива
    public List<Dictionary<string, int>> indexes;// Список
    индексных выражений
    public ProcessIndexer(indexer ind);
    private static Dictionary<string, int>
          summKoeffs(Dictionary<string, int> left,
                     Dictionary<string, int> right);
    private static Dictionary<string, int>
          MinusUnExp(Dictionary<string, int> val);
    private static Dictionary<string, int>
          diffKoeffs(Dictionary<string, int> left,
                     Dictionary<string, int> right);
    private static Dictionary<string, int>
          multKoeffs(Dictionary<string, int> left,
                     Dictionary<string, int> right);
    public static Dictionary<string, int>
          processExpr(expression exp);
    public void Run()// основной метод класса
}
```

После создания объекта этого класса необходимо вызвать метод **Run()**. В результате работы этого метода поля **dim**, **name** и **indexes** содержат соответственно размерность массива, его имя и список разобранных индексных выражений. Это метод рекурсивно обходит узел синтаксического дерева, вызывая для каждого индексного выражения метод **processExpr**. Этот метод производит упрощение выражения и возвращает результат своей работы в виде, как представлено на рис. 3.

Рассмотрим подробнее, как происходит приведение подобных и раскрытие скобок. Выражение хранится в синтаксическом дереве в разобранном виде. Заводится структура данных словарь (ассоциативный массив), в котором в качестве ключа выступает имя переменной, а значение – коэффициент при этой переменной. Запускается рекурсивный обход выражения:

1. Если выражение – это идентификатор, то добавляем ее в словарь с соответствующим коэффициентом и выходим из обработки.
2. Если выражение – это константа, то добавляем ее в словарь и выходим из обработки.
3. Если выражение – это бинарное выражение, то вызываем рекурсивно обработку для каждого из подвыражений. Результатом рекурсивного вызова будет также словарь такого же вида как на рис. 3. К этим словарям в зависимости от бинарной операции применяем соответствующую функцию, выход которой и есть результат обработки.
4. Если выражение – это унарное выражение, то вызываем рекурсивно обработку для подвыражения, а у результата рекурсивного вызова поменяем знаки коэффициентов на противоположные. Возвращаем полученный словарь.

При сложении двух словарей подвыражений ищутся элементы с одинаковыми ключами, а их коэффициенты суммируются и попадают в результирующий словарь с таким же ключом-именем переменной, как и в исходных словарях. Остальные элементы с несовпадающими ключами копируются в результирующий словарь без изменений. При вычитании двух словарей подвыражений действия аналогичные. А вот при умножении проверяется, что один из сомножителей константа, так как иначе это выражение не соответствует линейному классу программ. На эту константу умножаются все коэффициенты другого сомножителя, который также представлен словарем.

6 Библиотека целочисленного параметрического программирования PipLib

Для построения решетчатого графа необходимо решить задачу целочисленного параметрического программирования (1) - (4) для построения минимального снизу решетчатого графа или задачу (8) - (11) – для минимального сверху решетчатого графа, как это было описано ранее в главе «Построение решетчатого графа». Для решения этой задачи была использована библиотека **PipLib (Parametric Integer Programming Library)**, разработанная П. Фотрье (P. Feautrier) и распространяемая по GNU-лицензии. Скачать ее можно на официальном сайте проекта[7].

PipLib находит лексикографический максимум (минимум) среди целых точек выпуклого многогранника. Для этих целей используется параметризованный симплекс метод и параметризованный метод отсечений Гомори [5, с. 33].

6.1 Входные данные PipLib

На вход библиотека получает выпуклый многогранник, заданный системой неравенств вида:

$$Ax \geq 0,$$

где **A** – матрица коэффициентов неизвестных и параметров. Эта матрица называется *доменом* и имеет специальное представление в **PipLib**. Рассмотрим его на примере.

Пример 10. Пусть ограничения заданы системой неравенств:

$$\begin{cases} -i + m \geq 0 \\ -j + n \geq 0 \\ j + i - k \geq 0 \end{cases}$$

Тогда домен задается матрицей следующего вида:

eq/in	i	j	k	m	n	cnst
1	-1	0	0	1	0	0
1	0	-1	0	0	1	0
1	1	1	-1	0	0	0

Каждая строка в такой матрице задает отдельное неравенство системы. Первый элемент каждой строки определяет тип ограничения(0 означает равенство, 1 – неравенство $\geq$). Далее следуют коэффициенты неизвестных (i, j), коэффициенты параметров – внешних переменных (k, m, n), значение свободного члена (cnst).

Конец примера 10.

Помимо домена, на вход **PipLib** поступают ограничения на параметры – *контекст*. Именно он определяет, сколько первых столбцов домена являются коэффициентами неизвестных. Он задается матрицей такого же вида, как и домен. Если домен имеет N столбцов, а контекст M, то первые $N - M$ столбцов матрицы домена после первого столбца, определяющего тип неравенства $(=, \geq)$, соответствуют коэффициентам неизвестных. Оставшиеся столбцы кроме последнего соответствуют параметрам. Поэтому контекст необходимо всегда указывать, даже если нет ограничений на параметры. В этом случае количество строк матрицы контекста равно нулю, а количество столбцов определяет количество параметров +2 (2 – это один столбец для указания типа неравенства и еще одни столбец для коэффициента константы)

Пример 11. Пусть контекст задан один ограничением:

$$-k + m + n \geq 0$$

Тогда контекст задается следующей матрицей

eq/in	k	m	n	cnst
1	-1	1	1	0

Конец примера 11.

6.2 Использованные структуры данных и функции PipLib

Библиотека **PipLib** реализована на языке *C* и доступна в исходных кодах. Для преставления домена и контекста использована следующая структура данных [6]:

```
struct pipmatrix
{
     unsigned NbRows;     /* Количество строк. */
     unsigned NbColumns;  /* Количество столбцов. */
     Entier ** p;    /*Массив указателей на строки матрицы. */
     Entier * p_Init;     /*Матрица строк в памяти. */
     int p_Init_size;     /*Для внутреннего использования. */
}
typedef  struct pipmatrix PipMatrix;
```

Для решения задачи целочисленного параметрического программирования есть специальная функция:

```
PipQuast * pip_solve
(    PipMatrix * domain,      /* Домен. */
     PipMatrix * context,     /* Контекст. */
     int Bg,                  /* Индекс большого параметра. */
     PipOptions * options     /* Опции. */
)
```

Функция **pip_solve** решает задачу, поданную ей на вход в виде параметров. Первые три параметра описывают задачу, которую пользователь хочет решить. Последний параметр описывает следующие опции, настройки библиотеки для решения данной задачи. Опишем подробнее каждый из параметров функции.

domain: имеет тип **PipMatrix** и содержит матрицу ограничений на неизвестные и параметры, имеющая вид, который описан в предыдущей главе.

context: имеет тип **PipMatrix** и содержит матрицу ограничений на параметры. Если есть параметры, но нет ограничений на них, то передается матрица, у которой 0 строк, а количество столбцов равно количеству параметров +2.

Bg: номер столбца для большого параметра. Может использоваться для решения некоторых задач (например, для перехода от задачи на

минимум к задаче на максимум). Если он равен -1, то не используется.

options: указатель на структуру данных, содержащую настройки управляющие поведением **PipLib**. Подробнее о них можно почитать в документации [6, с.23]

Эта функция возвращает решение в виде дерева заданного структурой **PipQuast**:

```
struct pipquast
{
     PipNewparm * newparm ; /* Список новых параметров. */
     PipList * list solution; /* Решение(если нет условия).*/
     PipVector * condition ; /* Условие.*/
     struct pipquast * next_then ; /* Указатель на ветку then
     условия.*/
     struct pipquast * next_else /* Указатель на ветку else
     условия.*/
     struct pipquast * father ; /* Ссылка на родительский
     PipQuast.*/

} ;
typedef  struct pipquast PipQuast;
```

PipQuast задает решение в виде дерева условных операторов. В качестве условия условного оператора выступает линейное неравенство, вида:

$$Ax \geq 0$$

Это неравенство задается вектором (типа **PipVector)** коэффициентов параметров, коэффициентов новых параметров и значения свободного члена. Если условие выполняется, то выполняется ветка **next_then**, иначе ветка **next_else**. Если поле **condition** равно NULL, то этот кваст является листовым элементом, и в поле **solution** содержится решение. Решение – это квазиаффинная функция, заданная в виде списка коэффициентов параметров, коэффициентов новых параметров и значения свободного члена.

Рассмотрим подробнее это на примере.

Пример 12. Пусть домен задачи – это домен из примера 10, а контекст возьмем из задачи 11. Тогда, решив эту задачу, получим следующее решение (для вывода решения есть специальная функция):

```
(if #[ -1 0 1 0]
     (list
     #[ 0 0 0 0]
     #[ 1 0 0 0]
     )
     (list
     #[ 1 0 -1 0]
     #[ 0 0 1 0]
     )
)
```

Это решение следует читать следующим образом. У нас три параметра: *k, m, n*. Поэтому условие выглядит так:

$$-1 * \mathrm{k} + 0 * \mathrm{m} + 1 * \mathrm{n} + 0 ** 1 \geq 0.$$

Если упростить, то получим $n - k \geq 0$. Неизвестных у нас два: *i, j*. Поэтому по ветке then условного оператора (первый list в выводе) имеем:

$$i = 0 * k + 0 * m + 0 * n + 0 * 1 = 0$$

$$j = 1 * k + 0 * m + 0 * n + 0 * 1 = k$$

По ветке else условного оператора (второй list в выводе) имеем:

$$i = 1 * k + 0 * m + (-1) * n + 0 * 1 = k - n$$

$$j = 0 * k + 0 * m + 1 * n + 0 * 1 = n$$

Окончательно решение можно выписать так:

```
if n-k>=0 then <0,k>
else <k-n,n>
```

Конец примера 12.

Следует отметить, что вид решения немного отличается от того, который рассматривается в работах [1], [3]. В этих работах решение представляет собой набор квазиафинных функций и их областей определения. А при использовании **PipLib** решение – это дерево условных операторов. Листья дерева задают функции, а их области определения могут быть получены из условий условных операторов.

6.3 Подключение PipLib к компилятору PascalABC.NET

Компилятор PascalABC.NET написан на языке программирования *C#*, а **PipLib** – на неуправляемом *C*. Поэтому, для того чтобы использовать эту библиотеку, потребовалось ее откомпилировать в управляемый код.

6.3.1 Управляемый PipLib

Был создан проект **CLRPip**, в котором помимо исходных кодов **PipLib** добавлены функции, чтобы можно было использовать эту библиотеку в управляемом приложении.

В заголовочном файле **CLRPip.h** описан управляемый класс с единственным необходимым для наших целей методом **PipSolve**:

```
public ref class PipLib
{
     public:
       static IntPtr PipSolve(IntPtr domain, IntPtr context,
                              int Bp, IntPtr options)
       {
          return IntPtr((void*)PipSolve_internal
               ((PipMatrix*)(domain.ToPointer()),
               (PipMatrix*)(context.ToPointer()), Bp,
               (PipOptions*)(options.ToPointer())));
       }
};
```

Это метод решает задачу целочисленного параметрического программирования и возвращает результат в управляемом виде, вызывая в процессе работы функцию **PipSolve_internal**, которая описана в файле **CLRPip.cpp**:

```
PipQuast* PipSolve_internal(PipMatrix * domain, PipMatrix *
context, int Bp, PipOptions * options)
{
     PipQuast * solution;
     FixMatrix(domain);
     FixMatrix(context);
     solution = pip_solve(domain, context, Bp, options) ;
     pip_close() ;

     return solution;
}
```

В этой функции вызывается неуправляемый **pip_solve** самой библиотеки **PipLib**, но перед этим вызывается функция **FixMatrix** для домена и контекста. Функция **FixMatrix** заполняет поле **p** класса **PipMatrix** указателями на строки матрицы, как того требует библиотека.

Результатом компиляции этого проекта является управляемая DLL, 32-х или 64-х разрядная, в зависимости от выбранных опций компиляции.

6.3.2 Враппер для работы с управляемой PipLib

Для работы с управляемой **PipLib** был реализован класс **PipWrapper**, который выполняет передачу данных в неуправляемую память из управляемой памяти, и обратно. Его UML-диаграмма:

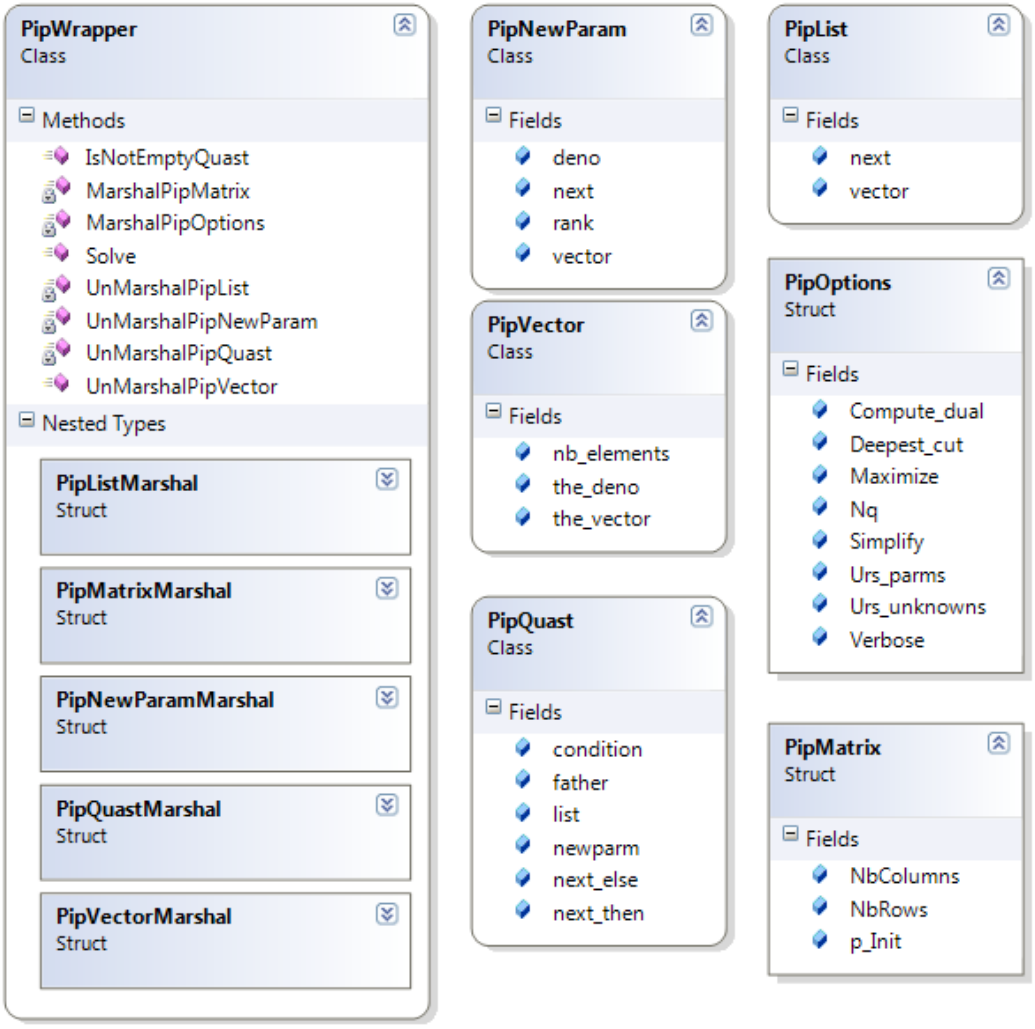


Рис. 4. UML-диаграмма классов, реализующих работу с Piplib в компиляторе PascalABC.NET

Рассмотрим подробнее методы и поля этого класса.

Набор структур **PipListMarshal**, **PipMatrixMarshal**, **PipNewParamMarshal**, **PipQuastMarshal**, **PipVectorMarshal** заполняются из неуправляемой памяти для дальнейшего формирования структур с результатами работы библиотеки. Для заполнения этих структур и получения результирующих данных, которые будут использованы для построения решетчатых графов, был реализован набор методов **UnMarshalPipVector**, **UnMarshalPipList**, **UnMarshalPipNewParam**, **UnMarshalPipQuast**. Метод **MarshalPipOptions** передает в неуправляемую память настройки библиотеки, которые были рассмотрены ранее в главе посвященной **PipLib**. Метод **MarshalPipMatrix** служит для передачи матрицы домена и контекста в управляемую память. Метод **IsNotEmptyQuast** возвращает истина, если решение не пусто, иначе – ложь. И, наконец, основной метод **Solve**, управляя всеми структурами и методами, возвращает решение задачи, вызывая аналогичную функцию из **PipLib**. Набор классов **PipMatrix**, **PipVector**, **PipNewParam**, **PipList**, **PipQuast** служит для представления результатов, возвращаемых **PipLib**, в управляемой памяти.

Следует отметить, что при реализации взаимодействия с **PipLib**, особое внимание уделялось тому, чтобы вся память, которая выделялась в процессе работы с неуправляемой памятью, была освобождена, и не было ее утечек.

7 Программная реализация построения решетчатого графа

В компиляторе PascalABC.NET реализованы директивы OpenMP для параллельного выполнения секций и циклов. Автором было реализовано построение решетчатых графов минимальных снизу и минимальных сверху для распараллеливаемого цикла, т.е. цикла помеченного директивой OpenMP:

```
{$omp parallel for [clause [ clause]…]}
  for var i:=0 to 10 do
  begin
    ...
  end;
```

Построение решетчатых графов происходит на этапе синтактико-семантического преобразования, перед построением параллельной версии цикла for с целью определения циклически порожденных зависимостей, мешающих распараллеливанию.

7.1 Формирование домена и контекста для построения решетчатого графа

Для построения решетчатого графа необходимо построить домен и контекст для библиотеки целочисленного параметрического программирования **PipLib**. Как было сказано ранее, визитор по синтаксическому дереву (**OccurCollectorSyntaxVisitor**) собирает информацию обо всех вхождениях переменных в распараллеливаемый цикл. Этот цикл помечен директивой OpenMP. Информацию о вхождениях нужно преобразовать в задачу, решаемую **PipLib**. Для этих целей был реализован класс **PipData**. Рассмотрим подробнее его поля и методы.

```
class PipData
{
     Occurrence oc1, oc2;
     int numUnkns = 0;
     HashSet<string> CommonIters;
     PipMatrix domain;
     PipMatrix context;
```

```
    public Dictionary<string, int> NameToInd;
    public Dictionary<string, int> ParamsToInd;

    public PipData(Occurrence oc1, Occurrence oc2){…}

    …
}
```

Конструктор этого класса принимает два вхождения переменных, для которых будет построен домен и контекст. Поле **numUnkns** хранит количество неизвестных в результирующем домене. Поле **CommonIters** содержит имена общих счетчиков циклов. Если вхождения переменных вложены в разное количество циклов, то нужно брать для построения домена и контекста только счетчики общих циклов. Словари **NameToInd** и **ParamsToInd** хранят соответствие имен неизвестных, параметров и номеров столбцов домена.

В этом классе есть методы **CanonizeDomain** и **CanonizeContext**, которые осуществляют канонизацию домена и контекста[1]. Эти преобразования затрагивают непосредственно контекст и домен, не влияя на сам узел цикла во внутреннем представлении. Так как в языке программирования Паскаль шаг цикла for всегда равен единице, то такие преобразования заключаются в пересчете левой и правой границы каждого цикла вложенного в гнездо и замене соответствующих элементов в матрице домена и контекста. Канонизация является всегда эквивалентным преобразованием и служит основой для многих других преобразований циклов[4].

Следующая группа статических методов класса **PipData** служит для вывода в поток (в файл или на консоль) решения, возвращаемого **PipLib**, и его отдельных элементов:

```
public static void print_quast(StreamWriter sw, PipQuast
solution, int indent); // вывод решения
public static void  pip_vector_print(StreamWriter sw,
PipVector vector);// вывод вектора
public static void  pip_newparm_print(StreamWriter sw,
PipNewParam newparm, int indent); // вывод новых параметров
```

[1] Цикл имеет канонический вид, если его левая граница равна 0 или 1,а шаг цикла равен 1

```
static void pip_list_print(StreamWriter sw, PipList list, int
indent);// вывод списка
public static void print_pipMatrix(StreamWriter sw,PipMatrix
matr);// Вывод матрицы
```

Построения домена и контекста выполняют методы **ComputeDomain** и **ComputeContext.** Рассмотрим подробнее этот процесс.

Класс **PipData** содержит два поля, соответствующих вхождениям переменных: *oc1* и *oc2*. Ищется зависимость вхождения *oc2* от *oc1*. Для построения домена надо объединить в одной матрице столбцы для счетчиков общих циклов обоих вхождений, столбцы их внешних переменных (параметров) и один столбец для свободного члена. После этого необходимо заполнить эту матрицу, как это описано в главе о построении решетчатого графа. Для контекста необходимо взять все параметры обоих вхождений и построить матрицу ограничений для них. Решается задача нахождения, при каких значения счетчиков циклов индексные выражения первого вхождения *oc1* равны индексным выражениям второго вхождения *oc2*. При этом счетчики циклов для второго вхождения *oc2* выступают в качестве параметров задачи.

Пример 13. Пусть есть гнездо циклов:

```
var t := 0;
{$omp parallel for private(t)}
for var i := 1 to n do
  for var j := 1 to m do
  begin
    a[i, j] := 5;
    u1
    t := a[i - 1, j - 1];
         u2
  end;
```

В этом гнезде циклов есть два вхождения переменной *a* – *u1* и *u2*. Рассмотрим потоковую зависимость использования *u2* от генератора *u1*. Оба вхождения находятся в гнезде из двух циклов со счетчиками *i* и *j*. Имеется две внешних переменных *n* и *m*. Производится поиск, при каких значениях счетчиков циклов, индексные выражения обоих вхождений сов-

падают. При этом счетчики циклов второго вхождения *u2* выступают в качестве параметров. Домен будет иметь следующий вид:

in/eq	i	j	i$	j$	n	m	$cnst
0	1	0	-1	0	0	0	1
0	0	1	0	-1	0	0	1
1	1	0	0	0	0	0	-1
1	-1	0	0	0	1	0	0
1	0	1	0	0	0	0	-1
1	0	-1	0	0	0	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Первые две строки задают равенства индексных выражений обоих вхождений. К имени счетчиков цикла второго вхождения добавляется знак $, чтобы их отличать от счетчиков циклов первого цикла. Первые две строки задают равенства:

$$i = i\$ - 1$$

$$j = j\$ - 1$$

Следующие четыре строки домена задают ограничения на счетчики цикла:

$$1 \leq i \leq n$$

$$1 \leq j \leq m$$

Последние две строки матрицы домена зарезервированы под альтернативные многогранники. Количество таких зарезервированных строк матрицы равно количеству общих циклов у двух вхождений. В зависимости от того какой элементарный решетчатый граф (минимальный снизу или сверху) строится, вид альтернативных многогранников будет меняться.

Контекст для данного примера будет содержать ограничения на счетчики циклов (*i$* и *j$*) второго вхождения переменной *a*:

in/eq	i$	j$	n	m	$cnst
1	1	0	0	0	-1
1	-1	0	1	0	0
1	0	1	0	0	-1
1	0	-1	0	1	0

Конец примера 13.

7.2 Реализация построения элементарных решетчатых графов

Каждый элементарный решетчатый граф хранится в виде списка функций, задающих его дуги. Для элементарного решетчатого графа потоковой зависимости каждая такая функция возвращает по концу дуги ее начало, а для элементарного решетчатого графа анти зависимости – по началу дуги ее конец. Для представления такой функции в программе был описан класс, имеющий следующий интерфейс:

```
class LatticeFunc
{
     public LatticeFunc(PipQuast quast, int num, Occurrence
     oc1, Occurrence oc2)
     public Occurrence oc1, oc2;
     public PipQuast quast;
     public int altPolyNumb; //номер альтернативного
     //многогранника, который использовался для построения
     //функции
}
```

Каждая функция элементарного решетчатого графа хранит:

- вхождения переменных *oc1* и *oc2*, для которых она была построена;
- номер альтернативного многогранника, который использовался для построения функции

- представление функции в виде дерева решения *quast* , которое содержит область определения функции и ее явное задание через параметры задачи.

Класс **ElemLatticeGraph** реализует построение элементарного решетчатого графа:

```
class ElemLatticeGraph
{
     Occurrence oc1, oc2;
     public List<LatticeFunc> Lfuncs;
     public ElemLatticeGraph(Occurrence oc1, Occurrence oc2,
     GraphKind gkind);
}
```

Этот класс по заданных вхождениям переменных и типу графа (минимальный снизу или минимальный сверху) производит заполнение списка функций, образующих граф. Сначала производится построение домена и контекста, как это описано в предыдущем параграфе. Затем перебираются все альтернативные многогранники, как заполняя зарезервированные строки домена. Домен, контекст и настройки библиотеки поступают на вход функции, осуществляющей решение задачи средствами **PipLib**. Полученное решение , если оно не пусто, формирует функцию элементарного решетчатого, а эта фунцкия добавляется в общий список.

В примере 13 для потоковой зависимости использования *u2* от генератора *u1* будет построен один альтернативный многогранник и соответсвенно функция элементарного решетчатого графа. Поле *quast* этой функции имеет следующий вид:

```
(if #[ 1 0 0 0 -2]
 (if #[ 0 1 0 0 -2]
  (list
   #[ 1 0 0 0 -1]
   #[ 0 1 0 0 -1]
  )
  ()
 )
 ()
)
```

7.3 Реализация построения полных решетчатых графов

Для построения полного решетчатого графа был описан класс:

```
class LatticeGraph
{
     public List<ElemLatticeGraph> Graph;
     public LatticeGraph(Dictionary<string, List<Occurrence>>
     Gens, Dictionary<string, List<Occurrence>> Uses,
     DependenceKind depKind, GraphKind gKind);
}
```

Этот класс принимает на вход список всех использований и генераторов гнезда циклов и заполняет список элементарных решетчатых графов. Кроме того, в конструкторе нужно указать тип зависимости(выходная, потоковая или анти зависимость) и тип графа (минимальный снизу или сверху). В соответствии с типом зависимости и типом графа, перебираются вхождения из списка генераторов и использований. При этом строятся элементарные решетчатые графы, которые потом объединяются в единый полный решетчатый граф.

8 Автоматическое распознавание ParDo циклов по решетчатому графу

В данном параграфе рассматривается метод распознавания циклов ParDo [3] в программе, основанный на теории и методах построения решетчатых графов. Циклы, обладающие свойством ParDo, часто приходится определять при распараллеливании программ. Если цикл обладает таким свойством, то все его итерации можно выполнять в любом порядке, в том числе и одновременно[3].

8.1 Определение ParDo цикла в программе

Определение. (Цикл ParDo по решетчатому графу G.) ([1,с. 86]) Пусть G – минимальный решетчатый граф фрагмента программы. Пусть цикл L находится на глубине вложенности k в некотором гнезде циклов данного фрагмента. Цикл L будем называть *циклом ParDo по графу G*, если в графе G не существует вершин $I = (I_1, I_2 \dots I_{n1}) \in V_i$ и $J = (J_1, J_2 \dots J_{n2}) \in V_j$, для которых одновременно выполняются условия:

1. $I_l = J_l$, для всех $l \in [1 \dots k-1]$;
2. $I_k < J_k$;
3. решетчатый граф G содержит дугу, идущую из вершины I в вершину J;
4. операторы $Stmt_i$ и $Stmt_j$, опорные пространства которых есть V_i и V_J соответственно, содержатся в цикле L.

Для того чтобы определить, можно ли итерации некоторого цикла исполнять одновременно, необходимо рассмотреть три минимальных решетчатых графа: потоковой, анти - и выходной зависимости.

Определение. ([1,с. 91]) Пусть G – решетчатый граф, полученный объединением минимальных решетчатых графов, описывающих потоковую, анти- и выходную зависимость по значению, а именно: мини-

мального снизу решетчатого графа потоковой зависимости, минимального сверху графа анти зависимости, минимального снизу или сверху графа выходной зависимости. Цикл, являющийся ParDo циклом по решетчатому графу *G*, будем называть *ParDo циклом*.

Утверждение 1. Все итерации ParDo цикла могут быть исполнены в любом порядке, в том числе и одновременно.

Доказательство этого факта можно найти в работе [1, с. 91].

8.2 Алгоритм распознавания ParDo циклов по решетчатым графам

Резюмируя все вышесказанное, можно привести следующий алгоритм определения ParDo циклов.

Алгоритм определения ParDo циклов в произвольном гнезде [1]:

1. Для рассматриваемого гнезда циклов строим минимальные решетчатые графы, описывающие потоковую, анти- и выходную зависимость по значению.

2. Обрабатываем каждую функцию *F*, описывающую дуги одного из построенных графов:

- будем считать, что общие циклы для пары вхождений, которые определяют дуги описываемые функцией *F*, занумерованы от 1 в порядке вложенности, начиная с самого внешнего;
- если при построении функции *F* использовался альтернативный многогранник с номером *k*, то пометим цикл с номером *k* как *DoSeq*; эта метка означает, что итерации цикла исполнять одновременно нельзя.

3. Все циклы, которые не были помечены как *DoSeq*, являются ParDo циклами.

8.3 Программная реализация распознавания ParDo циклов для компилятора PascalABC.NET

Распознавание ParDo циклов в компиляторе PascalABC.NET производится для циклов, помеченных директивой OpenMP. В случае если распараллеливаемый цикл не является ParDo, выдается предупреждение, что существуют циклически порожденные зависимости, мешающие параллельному выполнению. Так как определение ParDo циклов основано на решетчатых графах, следовательно, оно работает только для линейного класса программ.

Для реализации преобразований программ, основанных на решетчатых графах, был разработан класс **LatticeGraphAlgorithms**. У этого класса есть метод:

```
/// <summary>
/// Метод, определяющий циклы, которые можно выполнять
    параллельно
/// </summary>
/// <param name="gens">Генераторы</param>
/// <param name="uses">Использования</param>
/// <param name="iterators">Счетчики циклов</param>
/// <param name="reductions">Список редукций</param>
/// <param name="privVars">Список приватных переменных</param>
/// <returns>возвращает список циклов for, которые можно
            выполнять параллельно</returns>
public static HashSet<SyntaxTree.for_node>
            FindParDoCycles(Dictionary<string,
            List<Occurrence>> gens, Dictionary<string,
            List<Occurrence>> uses,
            HashSet<PascalABCCompiler.SyntaxTree.for_node>
            iterators, List<ReductionDirective> reductions,
            List<string> privVars)
```

Это метод получает на вход гнездо циклов **iterators**, все генераторы и использования данного гнезда и возвращает список ссылок на циклы, которые являются ParDo. При этом учитываются переменные редукции и приватные переменные, объявленные в директиве OpenMP.

При реализации этого метода строятся три решетчатых графа: минимальный снизу решетчатый граф потоковой зависимости, минимальный сверху граф анти зависимости, минимальный снизу граф выходной за-

висимости. Затем перебираются все функции, описывающие графы, и вычеркиваются из списка ParDo циклов те циклы, номера которых равны номерам альтернативных многогранников, используемых при построении функций. Сначала все циклы гнезда считаются ParDo циклами.

Пример 14. Рассмотрим гнездо циклов:

```
{$omp parallel for }
  for var i := 1 to n do
    for var j := 1 to m do
    begin
      a[i, j] := a[i - 1, j - 1];
    end;
```

В этом гнезде циклов внешний цикл по *i* не является ParDo. А внутренний цикл по *j* является ParDo.

Конец примера 14.

8.4 Сравнение с другими методами распознавания ParDo циклов

Распознавать ParDo циклы можно с помощью векторов направления зависимости [8] или, что эквивалентно, с помощью носителей зависимости [9]. Однако подобные способы имеют ряд недостатков [1, с. 95]:

1. Один вектор направления зависимости может описывать не одну зависимость. Например, один вектор направления может описывать потоковую зависимость использования v от генератора u и, одновременно, описывать самозависимость генератора u. Поэтому, при использовании указанных представлений невозможно точно распознать тип зависимости, по которому цикл обладает или не обладает свойством ParDo.
2. Вектор направления зависимости никак не учитывает ложные зависимости.
3. Эффективные методы построения векторов направления зависимости, основанные на неравенствах Банержи [8, 9], дают лишь достаточные условия даже в случае отсутствия внешних

переменных. Поэтому такие методы иногда могут не распознать ParDo цикл.

Метод распознавания ParDo циклов, предложенный в данной работе, лишен перечисленных недостатков.

9 Преобразование циклов с помощью унимодулярных матриц

Любое преобразование тесно вложенного гнезда циклов, состоящее из комбинации таких преобразований, как перестановка циклов, скашивание гнезда циклов, инверсия цикла, может быть реализовано с помощью унимодулярных матриц.

Унимодулярные преобразования позволяют изменить порядок исполнения итераций цикла, благодаря чему можно распараллелить некоторые виды циклов, правильно подобрав матрицу преобразования.

Необходимость использования унимодулярных матриц состоит в том, что, переходя к новым индексным переменным в цикле, мы получаем целочисленный вектор тогда, когда исходный вектор индексных переменных целочисленный, и — наоборот. Преобразованный цикл будет состоять из тех же экземпляров операторов, но экземпляры в новом цикле будут исполняться в порядке возрастания новых индексных переменных. В случае с целочисленными матрицами операции сложения, вычитания, умножения, скалярного умножения на целое число и транспонирования всегда приводят к целочисленной матрице. Однако вычисление обратной матрицы не всегда приводит к целочисленной матрице. Важным же свойством унимодулярной матрицы является то, что обратная к ней матрица — также целочисленная и унимодулярная.

При унимодулярных преобразованиях необходимо генерировать новые границы гнезда циклов. Для унимодулярных преобразований точные границы нового гнезда циклов находятся с помощью алгоритма Фурье-Моцкина.

9.1 Описание входных данных для унимодулярного преобразования

Рассматриваются только тесные гнезда циклов вида [2]:

```
for var i1 := L1 to U1 do
    for var i2 := L2(i1) to U2(i1) do
      //...
      for var in := Ln(i1, ..., in-1) to Un(i1, ..., in-1) do
      begin
        LoopBody(i1, ..., in);
 end;
```

Здесь:

a. Шаг каждого цикла равен 1

b. L_i и U_i (i=1...n) – соответственно нижняя и верхняя границы изменения счётчиков циклов. Все L_i и U_i (i = 1..n) – целочисленные линейные функции от счетчиков циклов такие, что:

$L_i = \sum_{k=1}^{i-1} a_i^k \cdot i_k + \alpha_i$, $U_i = \sum_{k=1}^{i-1} d_i^k \cdot i_k + \beta_i$, где a_i^k и d_i^k – константы, а α_i и β_i могут быть константами или внешними переменными

c. LoopBody($i_1, \ldots, i_n$) – тело цикла, в котором индексные выражения массивов – целочисленные линейные функции от счетчиков циклов.

9.2 Описание пространства итераций гнезда циклов в матричной форме

Пространство итераций гнезда циклов может быть описано в матричной форме. Учитывая, что $\sum_{k=1}^{i-1} a_i^k \cdot i_k + \alpha_i \le i_i$ и $i_i \le \sum_{k=1}^{i-1} d_i^k \cdot i_k + \beta_i$Тем самым, объединяя эти два неравенства, получим:

$$\sum_{k=1}^{i-1} a_i^k \cdot i_k + \alpha_i \le \sum_{k=1}^{i-1} d_i^k \cdot i_k + \beta_i$$

Отсюда следует:

$$\sum_{k=1}^{i-1} (a_i^k - d_i^k) \cdot i_k \le \beta_i - \alpha_i$$

Таким образом, можно записать пространство итераций гнезда циклов в матричном виде $A \cdot I \le \beta$

Пример 15. Рассмотрим гнездо циклов:

```
for var i1 := 0 to 7 do
    for var i2 := i1 to 8 do
      for var i3 := i1 + i2 to i1 + 10 do
          LoopBody(i1, i2, i3);
```

Матричная форма будет иметь вид:

$$\begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & -1 \\ -1 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} i1 \\ i2 \\ i3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 7 \\ 0 \\ 8 \\ 0 \\ 10 \end{pmatrix}$$

Конец примера 15.

9.3 Матрица преобразования

Для унимодулярного преобразования требуется матрица следующего вида:

$T = (a_{i,j})_{i=1,j=1}^{n,n} \in M_{n\times n}(Z)$ – невырожденная матрица, где $M_{n\times n}(Z)$ – множество матриц порядка $n \times n$ с целыми элементами.

Матрица подбирается в зависимости от преобразования, которое мы хотим получить. Матрицы для некоторых преобразований циклов, можно посмотреть в приложении А.

Определение. Невырожденная матрица T называется унимодулярной, если её определитель равен ± 1.

Теорема. Если матрица T – унимодулярная, то T^{-1} – также унимодулярная.

Доказательство непосредственно следует из свойств определителя.

Таким образом, получается что, если матрица преобразования T унимодулярная, то T и T^{-1} переводят целочисленные векторы в целочисленные векторы.

9.4 Преобразование пространства итераций матрицей преобразования

Пусть есть унимодулярная матрица преобразования T и пространство итераций гнезда циклов в матричной форме, т.е. $A \cdot I \leq \beta$. Тогда можно "подействовать" матрицей преобразования T на пространство итераций гнезда циклов, другими словами преобразовать пространство итераций гнезда циклов с помощью матрицы преобразования.

Для этого из неравенств:

$$A \cdot I \leq \beta$$

$$T \cdot I = J$$

Учитывая, что матрица T – обратима, можно получить следующие неравенства:

$$T^{-1} \cdot J = I$$

$$A \cdot T^{-1} \cdot J \leq \beta$$

Заменим $A * T^{-1}$ на A' и получим:

$A' \cdot J \leq \beta$ – *преобразованное пространство итераций в матричной форме.*

При этом если матрица преобразования унимодулярная, то новые границы циклов, которые находятся с помощью алгоритма Фурье-Моцкина по преобразованному пространству итераций, будут точными и могут быть непосредственно использованы для построения нового гнезда циклов.

9.5 Условия эквивалентности унимодулярного преобразования циклов

Прежде чем применять данное преобразование, необходимо проверить условие эквивалентности [2][10].

Пусть дана матрица T – матрица преобразования и гнездо циклов. Построим векторы расстояния зависимости для вхождений гнезда циклов. Тогда преобразование будет эквивалентно, если выполнены условия:

1. T – невырожденная матрица.

2. Выходные, анти- и потоковые зависимости не изменятся, если преобразованные векторы расстояния зависимостей лексикографически положительны[2], т.е. $\forall d_i \;\; T * d_i > \bar{0}$, где d_i – векторы направления зависимостей.

9.6 Алгоритм Фурье-Моцкина

Границы нового гнезда циклов могут быть найдены из матричного неравенства $A' \cdot J \leq \beta$ с помощью алгоритма Фурье-Моцкина. Опишем подробно этот алгоритм [10].

Пусть тесное гнездо содержит d вложенных циклов. k – номер текущего цикла, для которого находятся границы. Сначала k = d, то есть алгоритм определяет границы, начиная с самого внутреннего цикла. Пусть матрица A' имеет m строк и k столбцов. Матричную форму $A' \cdot J \leq \beta$ можно записать как систему из m неравенств:

$\sum_{j=1}^{k} a_{ij} \cdot J_j \leq b_i$, где $1 \leq i \leq m$. (*)

В цикле по k от d до 1 выполняем следующие шаги:

Шаг 1. *Переупорядочивание строк матрицы* A'. Переставляем строки матрицы A' согласно значениям элементов $a_{ik}(1 \leq i \leq m.)$ последнего столбца. Сначала располагаем строки, в которых $a_{ik} > 0$. Пусть таких строк p штук. Далее располагаем строки, у которых $a_{ik} > 0$. Пусть это строки с номерами $p < i \leq q$. Оставшиеся строки с номерами $q < i \leq m$ имеют нулевой элемент a_{ik}. После такого переупорядочивания получим три множества неравенств:

$$a_{ik} \cdot J_k \leq \; b_i - \sum_{j=1}^{k-1} a_{ij} \cdot J_j$$

$$-b_i + \sum_{j=1}^{k-1} a_{ij} \cdot J_j \leq (-a_{ik}) \cdot J_k \quad (**)$$

[2] Вектор лексикографически положителен, если его первая ненулевая координата положительна

$$\sum_{j=1}^{k-1} a_{ij} \cdot J_j \leq b_i$$

Шаг 2. Генерация границ. Первые p неравенств в этой системе определяют верхние границы на счетчик цикла J_k. Следующие $q - p$ неравенств определяют нижние границы этого счетчика цикла. Для того чтобы сгенерировать верхнюю границу счетчика цикла J_k, берется каждое из первых p неравенств, находится отношение правой части к коэффициенту a_{ik}. Так в счетчиках циклов допустимы только целые выражения, к каждому элементу множества полученных отношений применяется функция *floor*[3]. Если $a_{ik} = 1$, то функцию *floor* можно опустить. Потом среди множества таких выражений берется минимум, что и является результирующей верхней границей. Если всего одно неравенство ограничивает сверху счетчик цикла J_k ($p = 1$), то функцию минимум можно опустить. Аналогичные рассуждения производятся для нижних границ счетчика цикла J_k. Отличия состоят лишь в том, что вместо функции *floor* для отношений берется функция *ceil*[4], и для всех выражений потом вместо минимума берется максимум. Аналогичные упрощения также можно произвести.

Шаг 3.Создание новой системы неравенств. После того, как новые границы для счетчика цикла J_k сгенерированы, производится перестройка матрицы A'. При этом производится исключение последнего столбца этой матрицы, соответствующего счетчику цикла J_k. Для этого перебираются все пары неравенств вида:

$L \leq c_1 \cdot J_k$ и $c_2 \cdot J_k \leq U$, где $c_1 > 0$ и $c_2 > 0$.

Это неравенства из первых двух множеств неравенств системы (**). Всего таких пар $p \cdot (q - p)$штук. Каждая такая пара заменяется неравенством:

[3] Функция *floor* возвращает наибольшее целое, меньшее или равное переданному в качестве параметра числу *n*

[4] Функция *ceil* возвращает наименьшее целое, которое больше значения аргумента *n* или равно ему

$$\frac{L}{c_1} \le \frac{U}{c_2}$$

Откуда следует:

$$c_2 * L \le c_1 * U$$

Чтобы избежать переполнения последнее неравенство можно разделить на НОД(c_1, c_2). Таким образом, первые q неравенств системы (**) заменяются на $p \cdot (q - p)$ неравенств следующего вида:

$$\sum_{j=1}^{k-1} (c_{i'} \cdot a_{ij} + c_i \cdot a_{i'j}) \cdot J_j \le c_{i'} \cdot b_i + c_i \cdot b_{i'}$$

Здесь:

$$1 \le i \le p, \qquad p < i' \le q$$

$$g = \text{НОД}(a_{i'k}, a_{ik})$$

$$c_i = \frac{a_{ik}}{g} \text{ и } c_{i'} = -\frac{a_{i'k}}{g}$$

Последние $m - q$ неравенств системы (**) переписываются в новую систему, исключая столбец с номером k.

Шаг 4. Уменьшаем k на единицу. Если $k = 0$ выход. Иначе переходим на шаг 1, решая новую систему неравенств.

Пример 16. Рассмотрим гнездо циклов:

```
for var i1 := 10 to 15 do
    for var i2 := 1 to 3 do
      for var i3 := 1 to 50 do
          LoopBody(i1, i2, i3);
```

И дана матрица преобразования:

$$T = \begin{pmatrix} 0 & 6 & 1 \\ 1 & -3 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Обратная матрица к T:

$$T^{-1} = \begin{pmatrix} 0 & 1 & 3 \\ 0 & 0 & 1 \\ 1 & 0 & -6 \end{pmatrix}$$

Расширенная матрица преобразованного пространства итераций:

$$(A'|b) = (A \cdot T^{-1}|b) = \left(\begin{array}{ccc|c} 0 & -1 & -3 & -10 \\ 0 & 0 & -1 & -1 \\ -1 & 0 & 6 & -1 \\ 0 & 1 & 3 & 15 \\ 0 & 0 & 1 & 3 \\ 1 & 0 & -6 & 50 \end{array}\right)$$

Применим к этой матрице метод исключений Фурье-Моцкина и получим следующую последовательность преобразований:

$$\left(\begin{array}{ccc|c} -1 & 0 & 6 & -1 \\ 0 & 1 & 3 & 15 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & -1 & -1 \\ 0 & -1 & -3 & -10 \\ 1 & 0 & -6 & 50 \end{array}\right) \rightarrow \left(\begin{array}{cc|c} 0 & 1 & 12 \\ 1 & 2 & 80 \\ -1 & -2 & -21 \\ 0 & -1 & -1 \\ 0 & 0 & 5 \\ 0 & 0 & 49 \\ 0 & 0 & 2 \\ -1 & 0 & -7 \\ 1 & 0 & 68 \end{array}\right) \rightarrow \left(\begin{array}{c|c} 1 & 78 \\ 1 & 68 \\ -1 & 3 \\ -1 & -7 \\ 0 & 5 \\ 0 & 49 \\ 0 & 2 \\ 0 & 59 \\ 0 & 11 \end{array}\right)$$

Результирующее гнездо циклов будет выглядеть так:

```
for var $i1 :=max(-3,7) to min(68,78) do
  for var $i2 :=max(1,ceil((21-$i1)/2))
          to min(12,floor((80-$i1)/2)) do
     for var $i3 :=max(1,ceil((10-$i2)/3),ceil(($i1-50)/6))
          to min(3, floor((15-$i2)/3)), floor(($i1-1)/6)) do
               LoopBody($i2+3*$i3, $i3, (-6)*$i3);
```

Конец примера 16.

9.7 Программная реализация алгоритма Фурье-Моцкина в компиляторе PascalABC.NET

В компиляторе PascalABC.NET автором реализован параметрический алгоритм Фурье-Моцкина. Этот алгоритм был использован для генерации новых границ цикла после поворота пространства итераций, который выполняется таким образом, чтобы цикл можно было выполнять параллельно. Это преобразование ищет векторы расстояний зависимостей для двумерного гнезда циклов, и, если эти векторы параллельны друг другу, то происходит поворот пространства итераций на ранее найденный угол. Подробнее это преобразование будет описано ниже.

9.7.1 Входные структуры данных для алгоритма Фурье-Моцкина

Автором работы был разработан класс **FourierMotzkinData**, который формирует информацию о гнезде циклов в виде расширенной матрицы:

$$(A|Params\ |Const) \qquad (12)$$

Здесь A – матрица коэффициентов счетчиков циклов, $Params$ – матрица коэффициентов параметров, $Const$ – столбец значений свободного члена. Эта расширенная матрица представляет следующую систему неравенств (в матричной форме):

$$A \cdot I \leq Params \cdot Z + Const, \qquad (13)$$

где I –вектор-столбец счетчиков циклов, Z – вектор-столбец параметров.

Рассмотрим подробнее класс **FourierMotzkinData**. Он имеет UML-диаграмму, как изображено на рис.5.

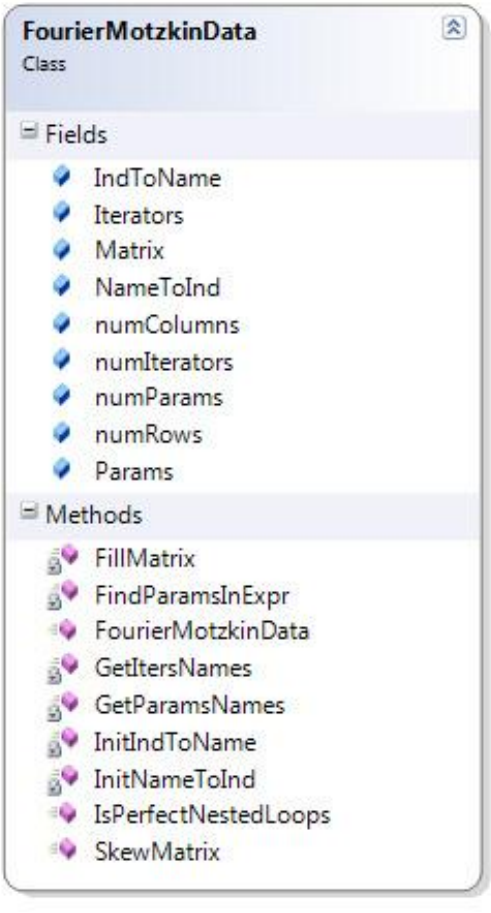


Рис. 5. UML-диаграмма класса FourierMotzkinData.

Поля:

Matrix – расширенная матрица системы, вида (12).

numColumns – общее количество столбцов расширенной матрицы.

numRows – общее количество строк расширенной матрицы.

numIterators – количество счетчиков циклов.

numParams – количество параметров(внешних переменных) цикла.

IndToName – словарь, который по номеру столбца возвращает имя переменной.

NameToInd – словарь, который по имени переменной возвращает номер соответствующего ей столбца.

Методы:

FillMatrix – метод, который заполняет расширенную матрицу системы ограничениями.

FindParamsInExpr – метод, который получает множество имен параметров из выражения.

GetItersNames – метод, который проходит по гнезду циклов возвращает его итераторы.

GetParamsNames – метод, который возвращает список всех параметров заголовка цикла.

InitNameToInd – метод, который инициализирует таблицу соответствия имен переменных и номеров столбцов в расширенной матрице.

InitIndToName – метод, который инициализирует таблицу соответствия номеров столбцов в расширенной матрице и имен переменных.

IsPerfectNestedLoops – метод, который возвращает истина, если гнездо циклов является тесным, а иначе – ложь.

SkewMatrix – метод, который производит умножение на унимодулярную матрицу скашивания с заданным фактором искажения.

Конструктор этого класса принимает на вход гнездо циклов во внутреннем представлении и заполняет поля этого класса. Сначала формируется

список имен счетчиков циклов с помошью метода **GetItersNames** и список параметров заголовка цикла с помощью метода **GetParamsNames**. По этим данным определяются размеры будущей расширенной матрицы. После этого заполняются таблицы соответствий **IndToName** и **NameToInd**. Наконец, заполняется расширенная матрица системы. Причем при разборе выражений производится приведение подобных и раскрытие скобок.

9.7.2 Реализация метода исключений Фурье-Моцкина

Автором работы метод исключений Фурье-Моцкина был реализован в параметрическом виде, с генерацией новых границ цикла в виде списка узлов синтаксического дерева. Вся соответствующая работа производится в классе **FourierMotzkinElimination**. Реализация сделана в соответствии с вышеописанным материалом.

Конструктор этого класса имеет следующий интерфейс:

```
public FourierMotzkinElimination(FourierMotzkinData FMD,
HashSet<SyntaxTree.for_node> iterators);
```

Он получает на вход расширенную матрицу системы, которая хранится в параметре **FMD** и список ссылок на циклы гнезда. В конструкторе производится вся работа по вычислению новых границ. Результат этой работы будет записан в поле **LoopsBounds –** список пар верхних и нижних границ счетчиков циклов, которые хранятся в порядке от самого внутреннего цикла до самого внешнего:

```
public List<LoopBounds> LoopsBounds;//Список границ цикла
```

Границы каждого цикла представлены объектами класса:

```
/// <summary>
/// Границы цикла
/// </summary>
class LoopBounds
{
    public SyntaxTree.expression low;// Нижняя
    public SyntaxTree.expression up;// Верхняя
    public LoopBounds(SyntaxTree.expression low,
       SyntaxTree.expression up)
}
```

И верхняя, и нижняя границы цикла представлены узлами синтаксического дерева SyntaxTree.expression, которые могут быть непосредственно подставлены в дерево и участвовать в дальнейшей компиляции.

При генерации границ производится некоторая оптимизация сгенерированных границ – если это возможно опускаются вызовы функций max, min, floor, ceil. Стоить отметить, что оптимизация неполная, и, если встретится выражение вида max(1,35) , оно не вычисляется на этом этапе. Попав в синтаксическое дерево, такой узел еще в дальнейшем подвергается оптимизации на этапе семантического анализа.

10 Распараллеливание двумерных гнезд тесно вложенных циклов с помощью унимодулярного преобразования

Предположим, что производится распараллеливание двумерного тесного гнезда циклов. С помощью алгоритма распознавания ParDo мы можем определить, что внешний цикл не распараллеливается, так как существуют зависимости, мешающие этому.

Теорема. Всегда существует унимодулярная матрица T размера 2×2 такая, что преобразованное гнездо эквивалентно исходному гнезду, и внутренний цикл может быть исполнен параллельно [11].

Если есть вектор расстояния зависимости (D_1, D_2) такой, что $D_1 > 0$ и $D_2 < 0$, тогда обозначим через $b = \lfloor \max\{-D_2/D_1\} + 1 \rfloor$, где максимум берется из всех векторов расстояния зависимости (D_1, D_2) в исходном гнезде таких, что $D_1 > 0$ и $D_2 < 0$. Иначе берем b = 1.

Возьмем унимодулярную матрицу $T = \begin{pmatrix} b & 1 \\ 1 & 0 \end{pmatrix}$. Тогда применяя эту матрицу преобразования к исходному гнезду, можно получить цикл внутренние итерации, которого можно выполнять параллельно.

Пример 17. Исходное гнездо циклов:

```
for var i := 1 to 6 do
    for var j := 1 to 5 do
      a[i, j] := a[i - 1, j + 1] + 1;
```

Вектор расстояния зависимости (1,–1).

Решетчатый граф изображен на рис. 6.

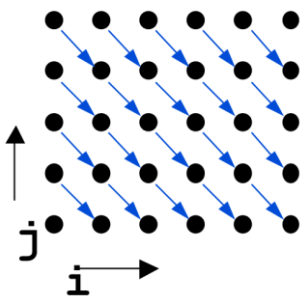


Рис.6. Решетчатый граф исходного гнезда циклов

Используем преобразование скашивания циклов со следующей унимодулярной матрицей:

$$T = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

В результате применения такого преобразования, получим следующее гнездо циклов:

```
for var i' := 2 to 11 do
    for var j' := max(i'-5,1) to min(6,i'-1) do
      a[j',i'-j'] := a[j' - 1, i' - j' + 1] + 1;
```

Внутренний цикл можно выполнять параллельно. Решетчатый граф изображен на рис. 7.

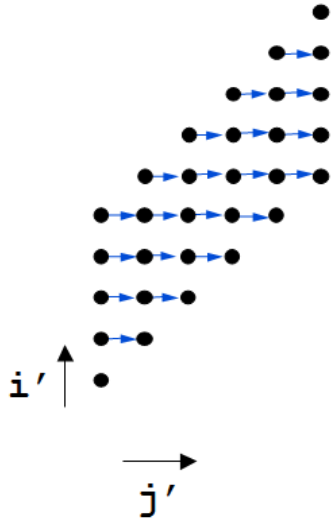


Рис.7 Решетчатый граф преобразованного гнезда циклов

Конец примера 17.

10.1 Унимодулярная матрица скашивающего преобразования

Для распараллеливающего преобразования двумерных гнезд циклов была использована унимодулярная матрица следующего вида:

$T = \begin{pmatrix} b & 1 \\ 1 & 0 \end{pmatrix}$, где b – фактор искажения. (14)

$T^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -b \end{pmatrix}$ – обратная матрица к матрице преобразования. (15)

Эта матрица, применяется к исходному гнезду циклов, а затем для полученной системы выполняется алгоритм Фурье-Моцкина генерации

новых границ циклов. После этого генерируется новое гнездо циклов, которое попадает в синтаксическое дерево и участвует в дальнейшей компиляции.

10.2 Программная реализация распараллеливания гнезд двух тесно вложенных циклов

Для реализации распараллеливания двумерных гнезд тесно вложенных циклов с помощью унимодулярного преобразования в классе **LatticeGraphAlgorithms** автором был описан метод **SkewLoopNest**, производящий скашивание гнезда циклов:

```
public static SyntaxTree.for_node SkewLoopNest
(OccurCollectorSyntaxVisitor occurCol,
 SyntaxTree.for_node for_node, DirectiveInfo dirInfo);
```

Этот метод возвращает преобразованный синтаксический узел гнезда циклов.

Входные параметры:

occurCol – визитор, содержащий информацию о вхождених переменных внутри тела цикла;

for_node – ссылка на исходное гнездо циклов в синтаксическом дереве программы;

dirInfo – содержит информацию о директиве, переменных редукции и приватных переменных.

В процессе своей работы этот метод строит три решетчатых графа: минимальный снизу решетчатый граф потоковой зависимости, минимальный сверху решетчатый граф анти зависимости и минимальный снизу граф выходной зависимости. По этим трем графам находятся векторы расстояния зависимости, и проверяется, что они параллельны друг другу(метод **CheckAndConvertVector**). Если есть векторы расстояния зависимости, мешающие распараллеливанию, то скашивание гнезда

циклов не производится. После построения этих векторов, находится базисный минимальный вектор, которому параллельны все остальные.

Следующий этап – это нахождение фактора искажения для унимодулярной матрицы преобразования. За это отвечает метод **FindResultVectorAngle**. Найденный фактор искажения передается в матрицу преобразования, которая применяется к исходному гнезду циклов. Генерируются новые границы циклов, и производится замена переменных. К имени счетчиков циклов дописываются слева символ $. Это гарантирует, что имя новой переменной не будет совпадать с именами переменных внутри цикла, так как в языке запрещено использовать этот символ в идентификаторах, а на этапе синтактико – семантического преобразования это уже не проверятся. Затем в начало тела цикла добавляется операторы присваивания, для того чтобы выразить старые переменные через новые. В общем виде преобразование можно представить схемой на рис. 8.

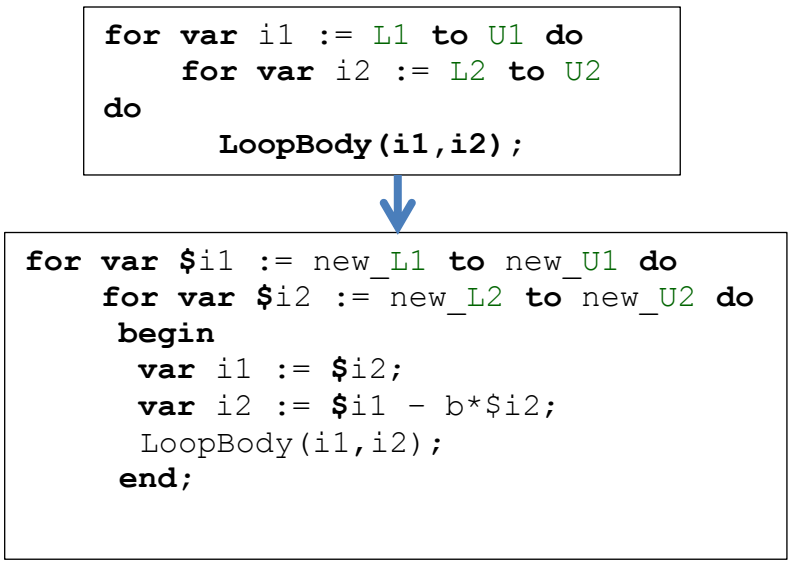


Рис. 8. Схема преобразования цикла

Заключение

В диссертационной работе представлена реализация элементов автоматического распараллеливания в компиляторе PascalABC.NET: различные виды решетчатых графов и алгоритмов на них.

В процессе разработки были решены следующие задачи:

- произведено построение расширенной информации о вхождениях переменных в теле распараллеливаемого цикла в компиляторе PascalABC.NET. При этом сделано упрощение выражений с приведением подобных и раскрытием скобок;
- подключена библиотека целочисленного параметрического программирования PipLib к компилятору PascalABC.NET;
- произведено построение различных видов решетчатых графов (минимальных сверху и снизу) фрагмента программы с помощью библиотеки PipLib на основе расширенной информации о вхождениях
- реализовано распознавание ParDo циклов для компилятора PascalABC.NET
- реализовано распараллеливание двумерных гнезд циклов, с помощью унимодулярного скашивающего преобразования.

В результате для линейного класса задач в компиляторе PascalABC.NET стало возможно параллельное выполнение внутренних циклов двумерных гнезд. Для этого же класса задач выдается предупреждение в случае, если программист пытается распараллелить цикл с циклически порожденными зависимостями. Кроме того, построенные решетчатые графы могут быть использованы для дальнейшей работы по преобразованию программ.

Литература

1. Шульженко А.М. Исследование информационных зависимостей программ для анализа распараллеливающих преобразований. Диссертация на соискание учёной степени кандидата технических наук, 2006.

2. Шилов М.В. Преобразования гнёзд циклов в открытой распараллеливающей системе. Магистерская диссертация, 2006.

3.Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. Санкт-Петербург «БХВ-Петербург» 2002. – 608 с.

4. Штейнберг Б.Я. Информационные зависимости и высокоуровневые распараллеливающие преобразования программ. Учебные материалы к спецкурсу «Параллельные вычисления и преобразования программ». Ростов-на-Дону, 2007г.

5. Feautrier P. Parametric integer programming// Operationnelle/ Operations Research, 22(3):243--268, September 1988.

6. Feautrier P. PIP/PipLib documentation
http://www.bastoul.net/cloog/pages/download/piplib.pdf

7. http://www.piplib.org/

8. Wolfe M., Banerjee U. Data Dependence and Its Application to Parallel Processing// International Journal of Parallel Programming. 1987. Vol.16. No.2. P.137-178.

9. Аллен Р., Кеннеди К. Автоматическая трансляция Фортран-программ в векторную форму. Векторизация программ: теория, методы, реализация. Сб. статей. Москва: Мир. 1991, с.77 - 140.

10. Aart J.C. Bik, Harry A.G. Wijshoff. Implementation of Fourier-Motzkin Elimination

11. Осмонов Р.А. Методы распараллеливания алгоритмов унимодулярными преобразованиями.

Приложение А.

Рассмотрим примеры унимодулярных матриц размера 2×2 для тесных гнезд двух вложенных циклов их применение для преобразования программы. Пусть исходное гнездо циклов имеет вид:

```
for var i1 := 1 to M do
    for var i2 := 1 to N do
          LoopBody(i1, i2);
```

При унимодулярных преобразованиях вводятся новые переменные $i1 и $i2, которые потом подставляются в тело цикла. Ниже приведены преобразования циклов.

Инверсия цикла:

Матрица преобразования	Результат

$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$	`for var $i1 := -M to -1 do`
	`  for var $i2 := 1 to N do`
	`  LoopBody(-$i1, $i2);`

Перестановка циклов:

Матрица преобразования	Результат

$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	`for var $i1 := 1 to N do`
	`  for var $i2 := 1 to M do`
	`  LoopBody($i2, $i1);`

Скашивание гнезда циклов:

Матрица преобразования	Результат

$\begin{pmatrix} 1 & 0 \\ p & 1 \end{pmatrix}$	`for var $i1 := 1 to M do`
	`  for var $i2 := 1+p*i1 to N+p*i1 do`
	`  LoopBody($i1, $i2 - p*$i1);`

p–фактор искажения.