© 2012 г. Б.Я. ШТЕЙНБЕРГ, д-р техн. наук, с.н.с.,
С.А. ГУДА, канд. физ.-мат. наук,
Р.И. МОРЫЛЕВ, научный сотрудник,
А.П. БАГЛИЙ, аспирант,
И.С. СКИБА, аспирант
(Южный Федеральный Университет, Ростов-на-Дону)

АНАЛИЗ ИНФОРМАЦИОННЫХ ЗАВИСИМОСТЕЙ В ДВОР

Анализ информационных зависимостей – основа корректности преобразования программ. Усложнение архитектуры вычислительных устройств повышает актуальность систем автоматического и полуавтоматического анализа и оптимизации (распараллеливания) программ, поскольку ручная оптимизация становится все более затратной. Анализ информационных зависимостей в различных языках программирования имеет свои особенности. Наиболее часто автоматические преобразования производятся при повышении быстродействия программ. Быстрые программы разрабатываются, как правило, на языках, позволяющих генерировать быстрый код: ФОРТРАН, C, C++, ассемблер. Именно для этих языков наиболее востребованы и развиты технологии анализа информационных зависимостей.

ANALYSIS OF THE DATA DEPENDENCIES IN DVOR / B.Y. Steinberg, S.A. Guda, R.I. Morilev, A.P. Baglij, I.S. Skiba (Southern Federal University, Milchakova 8a, Rostov-on-Don 344090, Russia, E-mail: aidan@pisem.net). Analysis of the data dependencies is the basis of the correct program transformations. The increasing complexity of computer architecture increases the urgency of automatic and semi-automatic program analysis and optimization, as manual optimizations becoming more expensive. Analysis of the data dependencies in different programming languages is different. Usually automatic transformations is performed for program performance improvement. Fast programs are developed in the languages that generate faster code such as Fortran, C, C++, Assembler. It is for these languages most in demand and developed analysis of the data dependencies technology.

1. Введение

Анализ информационных зависимостей – основа корректности преобразования программ.

Эквивалентные преобразования программ используются при оптимизации и распараллеливании программ, при построении обфускаторов и деобфускаторов. Усложнение архитектуры вычислительных устройств повышает актуальность систем автоматического и полуавтоматического анализа и оптимизации (распараллеливания) программ, поскольку ручная оптимизация становится все более затратной.

Анализ информационных зависимостей в различных языках программирования имеет свои особенности. Наиболее часто автоматические преобразования производятся при повышении быстродействия программ. Быстрые программы разрабатываются, как правило, на языках, позволяющих генерировать быстрый код: ФОРТРАН, C, C++, ассемблер. Именно для этих языков наиболее востребованы и развиты технологии анализа информационных зависимостей.

В системах автоматического преобразования программ (компиляторах, конверторах, распараллеливающих системах) анализ производится во внутреннем представлении. Внутренние представления таких программных систем можно разделить на два вида: высокоуровневые и низкоуровневые.

Не все оптимизирующие преобразования программ используют анализ информационных зависимостей, но распараллеливающие преобразования – все. При этом информационные зависимости, допускающие распараллеливание циклов на архитектуры типа MIMD или SIMD или конвейер различны.

При оценке качества систем оптимизации программ обычно сравнивают результаты их работы на бенчмарках, не вникая в причины превосходства одной системы над другой. Но причины часто кроются в анализаторе зависимостей.

Следует отметить, что существуют нарушения корректности при преобразованиях программ, не связанные с информационными зависимостями [9].

2. Информационная зависимость

Использование информационных зависимостей при преобразованиях программ описано во многих источниках [1], [2], [3], [4], [5]. Вхождением переменной, как обычно, будем называть всякое появление переменной в тексте программы вместе с тем местом в программе, в котором эта переменная появилась. Всякому вхождению (при конкретном значении индексного выражения для массивов) соответствует обращение к некоторой ячейке памяти. Если при этом обращении происходит запись в ячейку памяти, то такое вхождение называется генератором. Остальные вхождения называются использованиями.

Пример 1. В следующем операторе присваивания

```
A(I + B(J + 2)) = D(I - 1, B(I)) + 3 * A(1) - I + 5
1 2     3 4             5 6       7 8           9      10
```

десять вхождений переменных (их номера проставлены снизу) – и только первое является генератором.

Конец примера.

Пример 2. В следующем выражении языка C

```
A = B++
```

вхождение переменной B является одновременно и использованием и генератором.

Конец примера.

Говорят, что два вхождения порождают информационную зависимость, если при некоторых допустимых значениях индексных выражений они обращаются к одной и той же ячейке памяти.

Пример 3.

```
DO 99 I = 1, 100
99 A(I) = D(I, I - 1) + A(I - 1)
```

В операторе с меткой 99 вхождения переменной A информационно зависимы, поскольку оба обращаются к ячейке памяти A(10): вхождение A(I) на десятой итерации, а A(I - 1) – на одиннадцатой итерации цикла.

Конец примера.

3. Граф информационных связей

Граф информационных связей – это ориентированный граф, вершины которого соответствуют вхождениям, а дуга соединяет пару вершин (v,u), если выполняется одно из следующих условий:

1) Эти вхождения обращаются к одной и той же ячейке памяти (т.е. порождают информационную зависимость), причем вхождение v раньше, чем u и хотя бы одно из этих вхождений является генератором.
2) Вхождение u является генератором, а вхождение v принадлежит этому же оператору присваивания. Такие дуги называются тривиальными.

Генератор будем обозначать – out (output), а использование – in (input). Дуги графа информационных связей бывают трех типов в зависимости от типов инцидентных им вершин: out-in – истинная информационная зависимость (true dependence), in-out – антизависимость (antidependence), out-out – выходная зависимость (output dependence) [1, с.95].

Если информационная зависимость связывает два использования, то будем говорить об in-in – входной зависимости. Эти зависимости не влияют на эквивалентность преобразований программ и используются только при распределении данных в параллельной памяти.

Пример 4.

```
      DO 333 J = 2, N
      DO 333 I = 2, N
      A(I - 2, J) = X(2 * I - 1)
         v1              v2
      X(2 * I) = X(2 * I + 3) + A(I, J - 1)
        v3             v4              v5
      X(2 * I + 1) = A(J, I - 1)
           v6             v7
333 X(2 * I + 2) = A(I + K, J) + A(I - 2, J)
            v8             v9            v10
```

Выпишем список всех нетривиальных дуг графа информационных связей этого цикла: (v1; v7), (v1; v9), (v1; v10), (v4; v6), (v5; v1), (v6; v2), (v7; v1), (v8; v3), (v9; v1).

Если до выполнения программы неизвестно, какое значение примет K к началу выполнения цикла, то и характер зависимости между вхождениями v1 и v9 нельзя определить. В этом случае в графе информационной зависимости должны быть проведены обе дуги между этими двумя вхождениями.

Конец примера.

Пример 5.

```
 DO 333 I = 1, N
333 A(I + 2) = A(N - I)
      v1          v2
```

Для данного цикла список дуг графа имеет вид: (v1; v2), (v2; v1).

Конец примера.

Пример 6. В следующем цикле дуга графа информационных связей ведет от первого вхождения A(I) ко второму, но не наоборот. Вхождения переменной X соединяются двумя дугами в обе стороны: дугой антизависимости, поскольку на каждой итерации сначала X используется в первом операторе тела цикла, а потом перезаписывается; дугой истинной зависимости, поскольку на каждой итерации, начиная со второй, используется значение X, полученное на предыдущей итерации.

```
   DO 99 I = 1, N
   A(I) = B + X
   X = A(I)
99 CONTINUE
```

Конец примера.

Информационная зависимость между вхождениями называется циклически независимой (loop independent dependence), если эти вхождения обращаются к одной и той же ячейке памяти на одной и той же итерации цикла. Иначе зависимость называется циклически порожденной (loop carried dependence) [1, с.96].

В предыдущем примере дуга зависимости между вхождениями A(I) циклически независима, дуга зависимости между вхождениями X, ведущая сверху вниз тоже циклически независима, а дуга, ведущая снизу вверх – циклически зависима.

Особым образом строится граф зависимости фрагмента программы, содержащего вызовы процедур и функций. Операция (оператор) вызова процедуры или функции рассматривается, как одна вершина графа. Если после замены формальных переменных фактическими, в теле процедуры (функции), имеется генератор (использование) некоторой переменной X, то из данной вершины (в данную вершину) графа идут дуги, как из генератора (использования) X. Более точно, вызов процедуры следует временно заменить соответствующим текстом программы, построить граф зависимостей для рассматриваемого фрагмента с этим текстом, потом весь подграф, соответствующий вставленному тексту, стянуть в одну вершину (построить фактор-граф по подграфу).

4. Неопределенные информационные зависимости

Во многих случаях в программе между вхождениями переменных сложно бывает определить наличие или отсутствие информационной зависимости. Существует много методов определения информационной зависимости между вхождениями переменной с индексными выражениями, аффинно зависящими от счетчиков циклов, содержащих эти вхождения. Обзор почти двух десятков таких методов приводится в диссертации Арапбаева [8]. Многие точные методы анализа слабо практикуются в виду высокой вычислительной сложности. Но повышение быстродействия процессоров, на которых производится такой анализ, повышают актуальность точных методов. Тем не менее, «точность» в данном случае относится только к описанному классу пар вхождений, причем без учета контекста.

Для пар вхождений переменных с неаффинными индексными выражениями или сложными контекстными условиями автоматически точно определить наличие или отсутствие зависимостей сложно. Теория определения зависимостей в таких случаях продолжает развиваться. Но в данной сфере практика опережает теорию, поскольку рынок требует программные продукты с автоматическим преобразованием программ уже сейчас. Поэтому, в условиях трудной автоматической определяемости информационной зависимости предполагается худшее – наличие этой зависимости. Приведем примеры с такими ситуациями.

1) Наличие внешних переменных в индексных выражениях. Внешние переменные – это переменные, значение которых определяется за пределами исследуемого фрагмента.
2) Наличие в индексных выражениях переменных, которые вычисляются внутри исследуемого фрагмента, но эти значения носят нерегулярный характер.
3) Наличие в индексных выражениях неаффинных зависимостей. Например, счетчик цикла находится в показателе степени или используется косвенная адресация.
4) Сложный контекст. Вхождения переменных, между которыми определяется зависимость, находятся на разных ветках управляющего графа программы и сложно определить к какому вхождению программа обращается раньше, а к какому – позже.
5) Сложный контекст. Вхождения переменных, между которыми определяется зависимость, находятся в разных функциях.

При преобразованиях программ при трудно определяемой зависимости предполагают худшее – считают, что такая зависимость существует, и на графе информационных связей соответствующие вершины соединяют дугой.

Пример 7.

```
   DO 99 I = 1, N
99 X(2 * I) = X(2 * I + k)
```

В этом примере, если число k нечетное, то между обоими вхождениями переменной X нет информационной зависимости; если k четное и отрицательное, то есть истинная информационная зависимость, делающая цикл рекуррентным; если k четное и неотрицательное, то имеет место антизависимость. Если до выполнения программы неизвестно, какое значение примет k к началу выполнения цикла, то и характер зависимости тоже нельзя определить. В этом случае в графе информационных связей должны быть проведены обе дуги между двумя вхождениями переменной X.

Конец примера.

Итак, неопределенные информационные зависимости – это объект для исследований в теории преобразования программ и резерв для повышения эффективности оптимизирующих компиляторов.

5. Граф потока управления

Для проведения анализа инфрмационных зависимостей необходимо знать, в какой последовательности выполняются операторы программы. Для предоставления информации об этой последовательности часто используется граф потока управления программы (управляющий граф) [10, с. 529]. Это ориентированный граф, вершины которого соответствуют операторам программы, а дуга соединяет пару вершин (v,u), если при выполнении программы оператор u может выполниться сразу после оператора v. Таким образом, граф потока управления содержит в себе информацию о передачах управления в программе. Приведем пример управляющего графа небольшого участка программы:

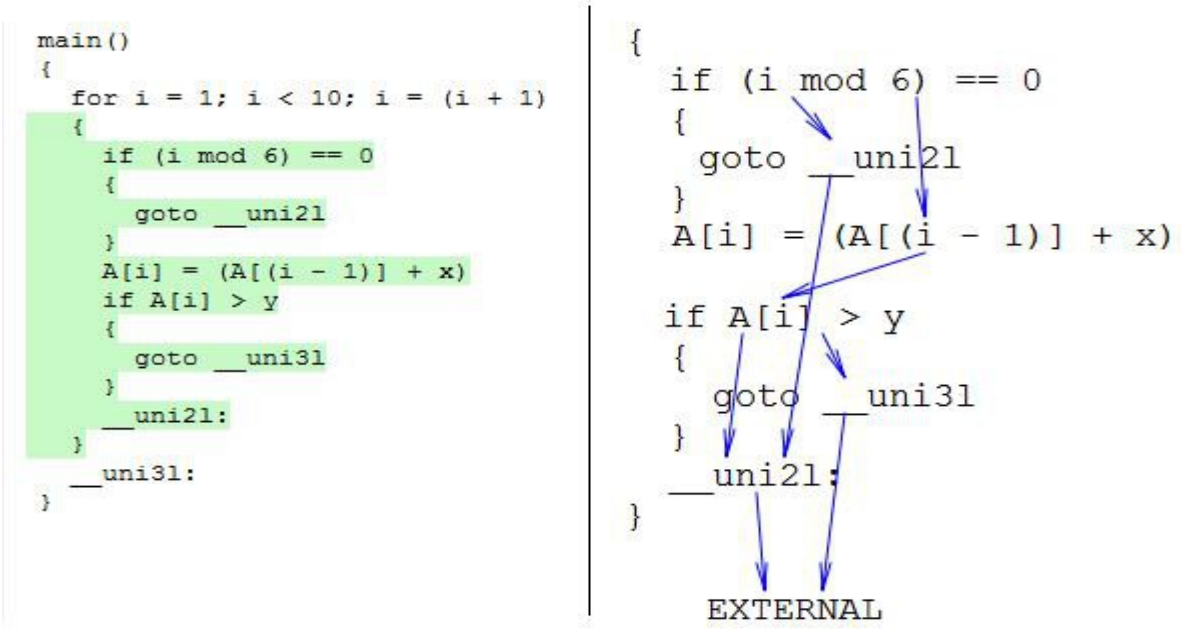

Рис. 1. Граф потока управления участка программы, выделенного цветом слева. Дуги, входящие в вершину «EXTERNAL», означают передачу управления при завершении программы.

Граф потока управления используется при построении графа информационных связей. Используется он для определения того, может ли один оператор в программе выполниться после другого, то есть существует ли на графе потока управления путь между этими двумя операторами. Если путь существует и в этих двух операторах есть обращение к одной ячейке памяти, то есть и информационная зависимость между вхождениями. Так, в примере на рис. 2 два вхождения X[i] находятся в разных ветках условного оператора, поэтому между ними нет дуги информационной зависимости.

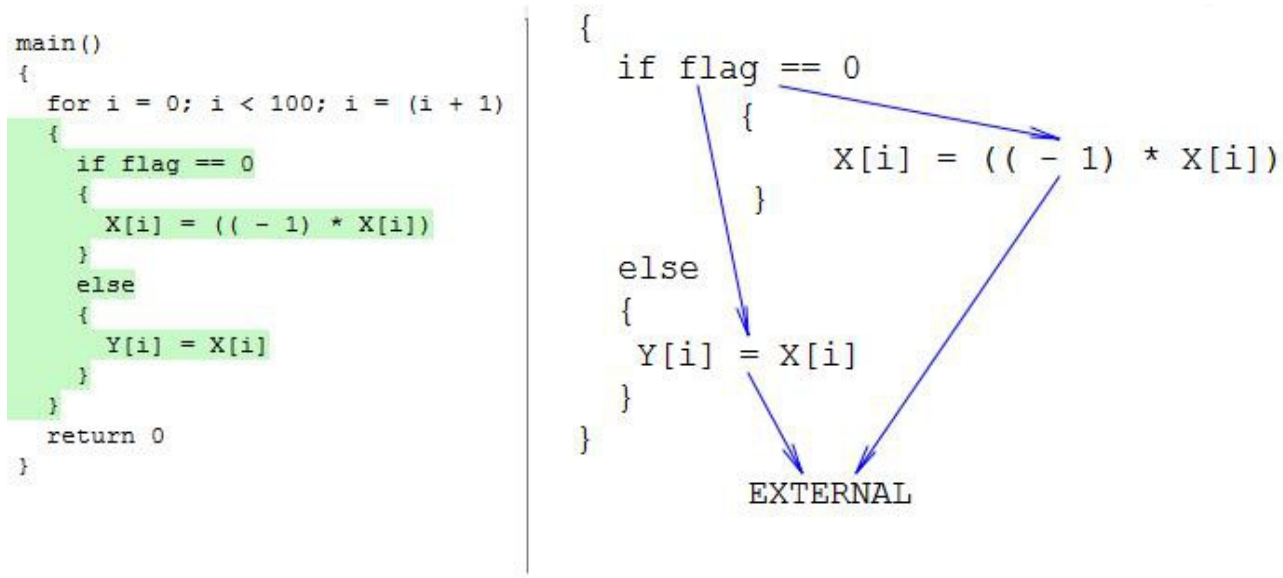

Рис. 2. Граф потока управления тела цикла слева. На этом графе нет пути, ведущего из оператора X[i]=(-1)*X[i] в оператор Y[i]=X[i] и наоборот, поэтому между вхождениями X[i] в этих операторах нет информационной зависимости.

Граф потока управления также используется при построении специального представления программы – SSA-формы [10, с. 369]. Это представление удобнее для анализа информационных зависимостей.

Следует заметить что, в графе потока управления могут быть участки, способные помешать анализу программы. Это участки структурного зацикливания и недостижимые участки. Неформально их можно определить так: структурное зацикливание – участок управляющего графа, из которого поток управления не может выйти, то есть это подмножество множества вершин графа, которые достижимы из остальных вершин, но из которых не достижимы остальные. Недостижимый участок - подмножество вершин, которые не достижимы из других вершин графа. Такие участки графа легко обнаруживаются при анализе графа потока управления.

6. Граф вызовов подпрограмм

Для анализа межпроцедурных зависимостей может использоваться граф вызовов подпрограмм [10, с. 904]. Это ориентированный граф в котором вершины соответствуют подпрограммам, а дуги – их вызовам. Этот граф хранит информацию обо всех вызовах подпрограмм, совершаемых в теле каждой отдельной подпрограммы.

7. Алгоритм построения графа информационных связей в ДВОР

Граф информационных связей является комплексным инструментом для анализа информационных зависимостей. Для его построения используются

1) внутреннее представление программы,
2) граф потока управления,
3) анализ псевдонимов,
4) решетчатый граф.

На вход модулю построения графа информационных связей подается программа, в виде дерева объектов внутреннего представления. Затем проводится анализ псевдонимов. В результате все вхождения указанного фрагмента программы разбиваются на группы псевдонимов – алиасы. Одно вхождение может принадлежать сразу нескольким группам. Два вхождения из одной группы соединяются дугой информационной зависимости, если между ними существует путь на графе потока управления. Этот этап в большинстве случаев дает недостаточно точные результаты. На графе могут присутствовать лишние дуги. Далее производится уточнение каждой дуги с помощью различных инструментов, например, решетчатым графом.

8. Анализ псевдонимов в ДВОР

Для анализа псевдонимов в ДВОР используется алгоритм [6]. Модуль анализатора виртуально исполняет код программы и следит за адресным содержимым ячеек памяти. Под адресными данными понимаются: адрес переменной, динамически выделенной области памяти, области памяти на стеке программы или ссылка на функцию. Анализатор псевдонимов, виртуально исполняя код программы, изменяет соответствующим образом структуру ProgramState, отражающую возможное содержимое всех ячеек памяти. Чтобы ограничить количество обходов циклов на графе потока управления, каждой вершине графа приписываются хеши структур ProgramState, с которыми ее посещал анализатор. Если обнаруживается, что данную вершину анализатор уже посещал ранее с равным текущему ProgramState, то рекурсивная ветвь обхода графа завершается.

Чтобы возвратить информацию о псевдонимах, анализатор во время виртуального исполнения программы приписывает каждому вхождению ссылки на ячейки памяти, к которым происходит обращение в данном месте. Рассмотрим пример

```
int *p, a, b, c;
if  (c > 0)  p = &a;
else  p = &b;
*p = 1;
a = b + 1;
```

Данную программу анализатор виртуально исполнит следующим образом:

1) посетит оператор if и рекурсивно вызовет два исполнителя: для ветвей true и false
2) первый исполнитель занесет в свой экземпляр ProgramState p = &a и, в соответствии с этим, припишет затем вхождению *p псевдоним a
3) второй исполнитель занесет в свой экземпляр ProgramState p = &b и добавит ко вхождению *p еще один псевдоним b

В итоге по данным анализатора псевдонимов модуль графа информационных связей проведет из вхождения *p во вхождения a и b две дуги *p→a выходной зависимости и *p→b истинной.

Помимо обнаружения псевдонимов в ДВОР анализатор алиасов применяется для определения факта возможного изменения содержимого вхождения во фрагменте программы; возможности приватизации переменных (включая указатели на динамически выделенную память) при распараллеливании циклов; функции, вызываемой в данном месте программы по указателю.

9. Решетчатый граф

Решетчатый граф информационных связей строится в ДВОР, используя алгоритм параметрического целочисленного программирования П.Фотрье [7]. На выходе алгоритма получается набор кусочно квазиаффинных функций, описывающих зависимости между итерациями гнезда циклов. Фрагмент программы, для которого строится решетчатый граф, должен принадлежать линейному классу (см. [7]). При этом должны отсутствовать нетривиальные альясы между переменными. Аффинные формы в индексных выражениях и границах циклов могут зависеть от внешних параметров. Решетчатый граф позволяет определить множество значений внешних параметров, для которых два вхождения связаны информационной зависимостью.

В ДВОР решетчатый граф применяется

1) При уточнении зависимостей. Например, в программе

```
a[0] = 1;
for (i = 1; i < n / 2; i++)  a[n - i] = 2 * a[i] + 1;
```

решетчатый граф определит, что между вхождениями a[0], a[n - i], a[i] отсутствуют какие бы то ни было зависимости.

2) При определении носителей зависимости. В примере

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      for (k = 0; k < n; k++)
         a[j][k] += 2 * a[j][k - 1] + 2;
```

решетчатый граф определит, что носителями зависимости между вхождениями a[j][k] и a[j][k - 1] являются первый и третий циклы (по i и по k), итерации же второго цикла по j независимы, их можно выполнять параллельно.

3) При проверке применимости преобразований программ. Например, решетчатый граф определит, что преобразование перестановка циклов применимо к гнезду

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      a[n - i][j] = 2 * a[i][j] + 1;
```

10. Особенности графа информационных связей в ДВОР

Одной из отличительных особенностей графа информационных связей в ДВОР является его построение на основе высокоуровневого внутреннего представления. Большинство компиляторов используют низкоуровневое представление программ, которое оперирует регистрами процессора, что не всегда удобно для преобразований программ.

Еще одна отличительная особенность графа информационных связей — это его визуализация. Она представляет собой текст фрагмента программы с дугами графа информационных связей отображенными поверх. С помощью компьютерной мыши вершины графов (соответствующие операторы или операции программы) вместе с

соответствующими дугами можно передвигать для получения лучшего визуального представления.

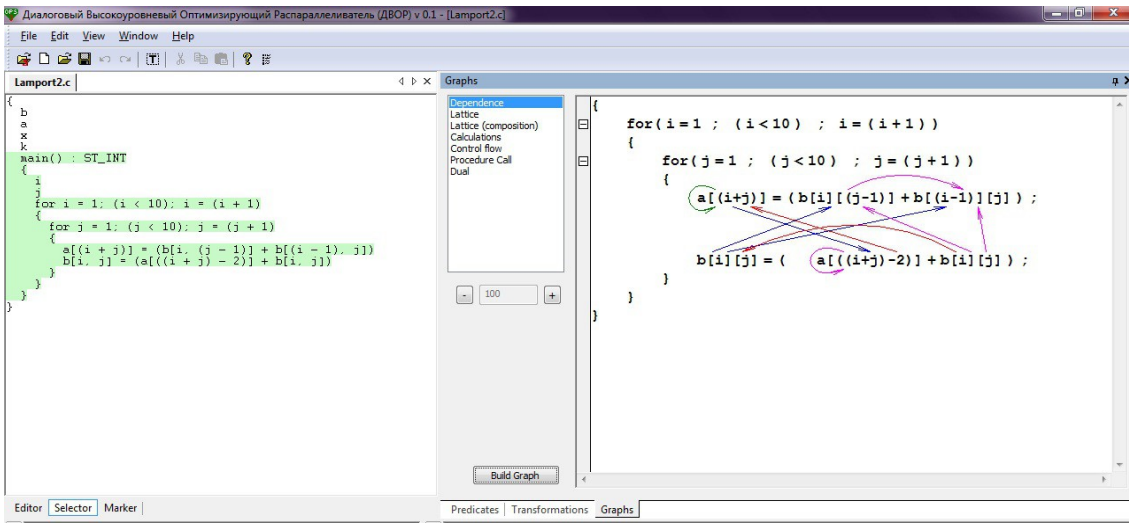

Рис. 3. Визуализация графа информационных связей в диалоговом высокоуровневом оптимизирующем распараллеливателе версии 1.

11. Диалоговое построение графа информационных связей

Не все дуги ГИС (графа информационных связей) можно проанализировать автоматически на этапе компиляции. Некоторые зависимости могут возникать из-за неопределенных значений переменных, входящих в состав индексных выражений вхождений. Возможные значения таких переменных известны только программисту и в тексте программы не содержатся. Так, в индексном выражении вхождения v9 в Примере 5 содержится вхождение переменной K, в зависимости от значения которой, вид ГИС фрагмента может меняться.

Для того чтобы проанализировать такие ситуации, необходимо получить дополнительную информацию о переменных программы у пользователя. Для этого в ДВОР реализован режим диалогового построения ГИС. В общем виде этот процесс можно представить в виде следующих шагов:

Пользователь выбирает интересующий фрагмент программы

Выбранный фрагмент анализируется и составляется список вопросов пользователю, которые помогут уточнить ГИС

Пользователь отвечает на поставленные вопросы

Происходит окончательное построение ГИС, который потом используется другими компонентами системы.

Таким образом, удается удалить дуги ГИС, мешающие проведению оптимизирующих преобразований и распознать параллелизм там, где раньше его найти не удавалось.

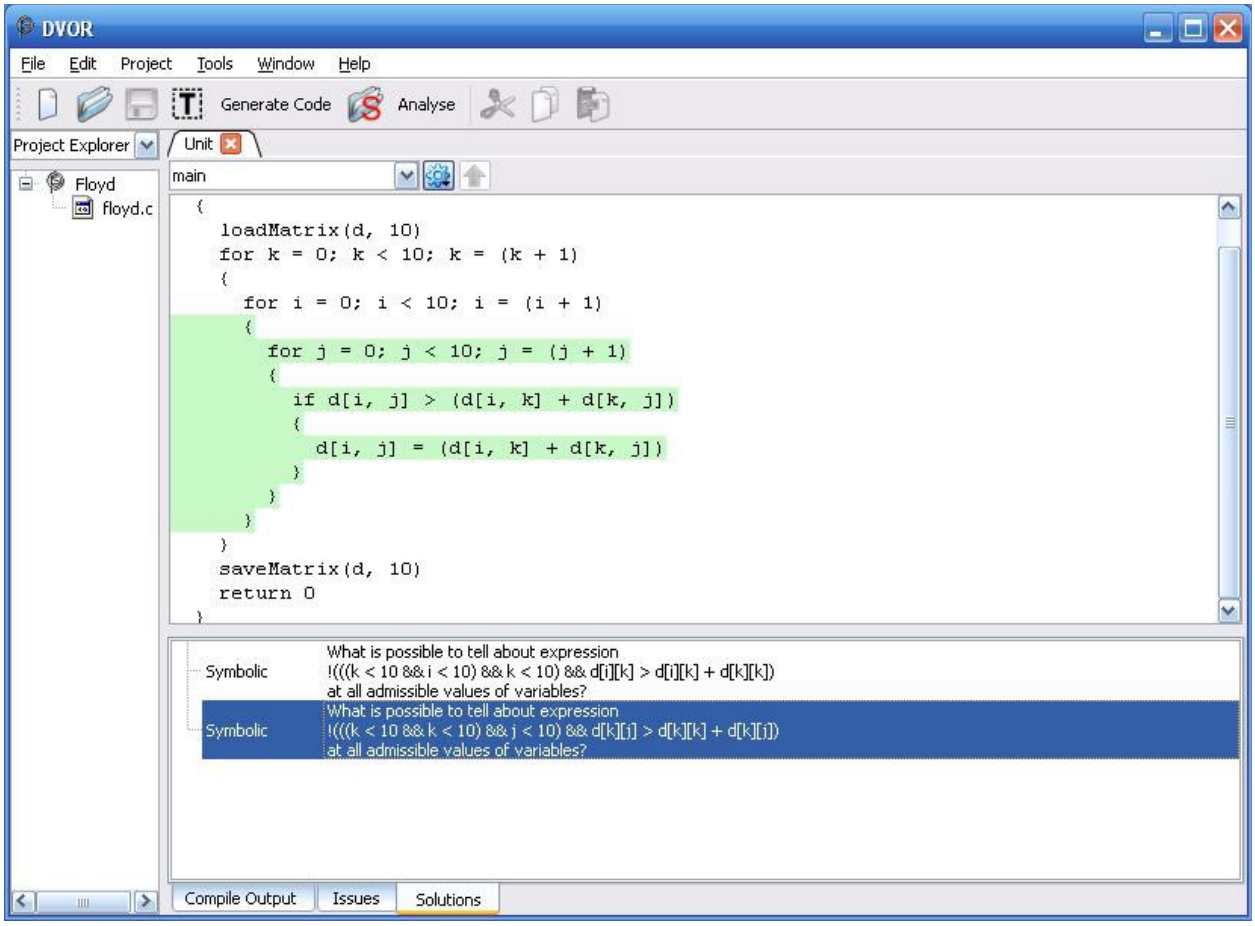

Рис. 4. Экранная форма ДВОР со списком вопросов пользователю, сгенерированных при анализе программы реализующей алгоритм Флойда.

СПИСОК ЛИТЕРАТУРЫ

1) Векторизация программ. // Векторизация программ: теория, методы, реализация. / Сборник переводов статей М.: Мир, 1991. С. 246 — 267.
2) *Штейнберг Б.Я.* Математические методы распараллеливания рекуррентных программных циклов на суперкомпьютеры с параллельной памятью. // Ростов-на-Дону, Издательство Ростовского университета, 2004 г., 192 с.
3) *Allen R., Kennedy K.* Optimizing compilers for Mordern Architetures // Morgan Kaufmann Publisher, Academic Press, USA, 2002, 790 p.
4) *Lamport L.* The parallel execution of DO loops // Commun. ACM.- 1974.- v.17, N 2, p. 83-93.
5) *Wolf M., Banerjee U.* Data Dependence and its Application to Parallel Processing/ International Journal of Parallel Programming // Vol.16, N 2, 1987, p. 137-178.
6) *Полуян С.В.* Анализ указателей для распараллеливания // Параллельные вычисления и задачи управления (PACO'2010): Труды V Международной конференции, ИПУ РАН, г. Москва, 26-28 октября 2010 г. С. 1065-1069.
7) *Feautrier P.* Parametric integer programming, RAIRO Recherche Operationnelle, 1988, v. 22, №3, p.243-268.

8) *Арапбаев Р.Н.* Анализ зависимостей по данным: тесты на зависимость и стратегии тестирования // Диссертация на соискание ученой степени кандидата физико-математических наук
9) *Нис З.Я.* Корректность преобразований // Труды конференции РАСО'2006
10) *Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman* Compilers: principles, techniques, and tools - second edition // Addison-Wesley, USA, 2007, 1009 p.