Санкт-Петербургский Государственный
Университет
Математико-механический факультет

Кафедра системного программирования

Анализ зависимостей в среде PRANLIB

Дипломная работа студента 544 группы
Сыча Геннадия Александровича

Научный руководитель	..... /подпись/	к.ф.-м.н. Д.Ю. Булычев
Рецензент	..... /подпись/	Е.Н. Вигдорчик
“Допустить к защите” зав. кафедры	..... /подпись/	д.ф.-м.н., проф. А.Н. Терехов

Санкт-Петербург
2007

Содержание

1 Введение

В настоящее время такое представление программы как граф зависимостей по данным [1, 4] имеет весьма широкое применение. Граф зависимостей служит мощным инструментом для понимания программ, их поддержки, отладки, тестирования, оптимизации, распараллеливания, а так же для многого другого. Построение графа зависимостей является неотъемлемой частью некоторых задач из области методов вычислений. Например, в работе [5] описывается основанный на анализе графа зависимостей метод автоматического построения программ дифференцирования математических функций, реализованных так же в виде программ. Еще одним примером применения такого представления является распараллеливание программ. Основным ограничением, накладываемым на распараллеливающие преобразования, является сохранение семантики исходной программы. Для выполнения этого требования необходимо руководствоваться информацией, полученной из графа зависимостей.

Ряд задач из области оптимизаций и кодогенерации содержат анализ зависимостей как неотъемлемую часть (например, оптимизации использования кэша данных и списковое планирование [2]).

PRANLIB [6] является библиотекой, в которой реализованы различные алгоритмы анализов потока управления. Мотивацией к данной работе послужило решение разработать модули для удобной реализации задач анализа потока данных и анализа зависимостей для библиотеки PRANLIB.

Задачей данной работы является расширение библиотеки PRANLIB представлением программы в виде графа зависимостей по данным и механизмами для решения задач анализа потока данных. Для этого необходимо разработать общее представление задач, реализовать задачи о достигающих определениях и о восходящем анализе использования переменных (см. описание ниже), а так же алгоритм построения графа зависимостей.

Все алгоритмы в среде PRANLIB реализованы в общем виде. Наличие в библиотеке общего представления задач анализа потока данных, а так же анализа зависимостей, послужит хорошим инструментом для реализации всевозможных задач анализа, которые можно будет эффективно переиспользовать и применять на различных конкретных представлениях программ.

В работе представлено описание модулей для реализации задач анализа потока данных, описан подход к построению графа зависимостей

на основе разработанного абстрактного представления программы и его реализация.

2 Анализ потока данных

В этом разделе вводятся используемые термины и обозначения, а также формально описывается задача анализа потока данных и ее решение.

2.1 Итеративный анализ потока данных

При решении задачи анализа потока данных множество фактов (решения) представляется в виде элементов *полурешетки*.

Определение 1 *Полурешеткой называется множество* L, *снабженное операцией* $\sqcap$, *обладающей свойствами:*

1. $\forall x \in L \quad x \sqcap x = x$;
2. $\forall x, y \in L \quad x \sqcap y = y \sqcap x$;
3. $\forall x, y, z \in L \quad (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$.

Естественным образом индуцируется отношение частичного порядка.

Определение 2 $\forall x, y \in L \quad x \leqslant y \quad \Leftrightarrow \quad x \sqcap y = x$,
$x < y \quad \Leftrightarrow \quad x \leqslant y \wedge x \neq y$.

Определение 3 *Частично-упорядоченное множество* X *будем называть множеством конечной высоты* N *тогда и только тогда, когда длины всех строго возрастающих последовательностей элементов* X *ограничены* N.

Определение 4 *Отдельно выделяют наибольший и наименьший элементы полурешетки:*

1. $\top \in L \quad \top \sqcap x = x \qquad \forall x \in L$;
2. $\bot \in L \quad \bot \sqcap x = \bot \qquad \forall x \in L$.

Определение 5 *Полурешетка называется ограниченной тогда и только тогда, когда в ней существуют наибольший и наименьший элементы.*

Определение 6 *Функция* $f : L \to L$ *называется монотонной, если* $\forall x, y \in L \quad x \leqslant y \quad \Rightarrow \quad f(x) \leqslant f(y)$.

Определение 7 *Под графом потока управления (ГПУ) будем понимать пару* (V, E), *где* V — *множество вершин графа, среди которых существует вершина, из которой достижимы все остальные (будем называть ее стартовой),* E — *множество ребер графа.*

Определение 8 (Формализация задачи анализа потока данных) *Пусть* $G = (V, E)$ — *граф потока управления,* L — *ограниченная полурешетка конечной высоты,* $\forall v \in V \quad f_v : L \to L$ — *монотонные потоковые функции. Тогда решением задачи анализа потока данных называется пара наименьших разметок* $before$ *и* $after$, *являющихся решением системы уравнений:*

$$\left\{ \begin{array}{l} before(v) = \bigsqcap_{w \in Pred(v)} after(w) \\ after(v) = f_v(before(v)) \end{array} \right\}_{v \in V}$$

Неформально говоря, полурешетка L представляет собой множество потоковых фактов, разметка $before$ описывает решение задачи анализа потоков данных до исполнения операторов в данной вершине графа, а разметка $after$ — после. Операция полурешетки описывает получение общей части нескольких решений.

Приведенная выше формализация задачи анализа потока данных описывает *прямую* задачу — разметка $before$ ассоциируется с входящими ребрами вершины. В *обратной* задаче $before$ ассоциируется с исходящими ребрами вершины, а $after$ с входящими. При этом потоковая информация распространяется снизу-вверх.

Утверждение 1 *Пусть* $L_1, ..., L_k$ — *ограниченные полурешетки конечной высоты* $\Rightarrow L = L_1 \times L_2 \times ... \times L_k$ — *ограниченная полурешетка конечной высоты:*

1. $\langle x_1, x_2, ..., x_k \rangle \sqcap \langle y_1, y_2, ..., y_k \rangle = \langle x_1 \sqcap y_1, x_2 \sqcap y_2, ..., x_k \sqcap y_k \rangle$
2. $\langle x_1, x_2, ..., x_k \rangle \leqslant \langle y_1, y_2, ..., y_k \rangle = x_1 \leqslant y_1 \wedge x_2 \leqslant y_2 \wedge ... \wedge x_k \leqslant y_k$
3. $\top_L = \langle \top_{L_1}, ..., \top_{L_k} \rangle$
4. $\bot_L = \langle \bot_{L_1}, ..., \bot_{L_k} \rangle$

Утверждение 2 *Пусть* $f_1, ..., f_k$ — *монотонные функции на* $L_1, ..., L_k$ $\Rightarrow \; f(\langle x_1, x_2, ..., x_k \rangle) = \langle f_1(x_1), f_2(x_2), ..., f_k(x_k) \rangle$ — *монотонная функция на декартовом произведении* $L_1, ..., L_k$.

Доказательства этих двух утверждений тривиальны. Теперь для описания решения задачи анализа потоков данных рассмотрим вновь систему из определения 8. Для каждой пары уравнений системы введем пару вспомогательных функций g_1, g_2:

$$before(v) = after(u_1) \sqcap ... \sqcap after(u_k) \Rightarrow g_1(x_1, ..., x_k) = x_1 \sqcap ... \sqcap x_k$$

$$after(v) = f_v(before(v)) \Rightarrow g_2(x) = f_v(x)$$

Утверждение 3 (Решение задачи анализа потока данных) *Пусть $G = (V, E)$ — граф потока управления. Тогда функция $F : L^{2\times|V|} \to L^{2\times|V|}$, определенная как*

$$F(\langle before_1, after_1, ..., before_{|V|}, after_{|V|}\rangle) =$$

$$\langle g_{11}(...), g_{12}(...), ..., g_{|V|1}(...), g_{|V|2}(...)\rangle$$

(функции g_{ij} определены выше) обладает слдующими свойствами:

1. *F монотонна на $L^{2\times|V|}$;*
2. *произвольная неподвижная точка F (т.е. такое x, что $F(x) = x$) является решением системы из определения 8;*
3. *$\exists n : F^n(\bot_{L^{2\times|V|}}, ..., \bot_{L^{2\times|V|}})$ — наименьшая неподвижная точка F.*

2.2 Стандартные задачи анализа потока данных

Среди задач анализа потока данных встречаются такие, в которых используются похожие полурешетки. Примерами являются задачи о доступных выражениях (available expressions), о живых переменных (live variables), о достигающих определениях (reaching definitions), о восходящем анализе использования переменных (upward exposed uses) — все они используют полурешетку, описанную ниже. В данной работе рассматривается задача о достигающих определениях и задача о восходящем анализе использования переменных.

Определение 9 *Пусть $G = (V, E)$ — граф потока управления. Будем говорить, что определение переменной x $(def(x))$ достигает использование $(use(x)) \Leftrightarrow$ существует путь в G $p = \{def(x), ..., use(x)\} : \quad \forall u \in p \setminus \{def(x), use(x)\} \quad u \neq def(x)$.*

Задача о достигающих определениях является прямой задачей. Решение содержит информацию о том, какие определения достигают каждую точку программы. В качестве полурешетки потоковых фактов фиксируется множество подмножеств присваиваний с операцией объединения:

1. A — множество присваиваний, $L = 2^A$ — полурешетка;
2. $\sqcap = \cup$;
3. $\top = A$;
4. $\bot = \varnothing$.

Очевидно, полурешетка имеет конечную высоту. Потоковая функция в каждой вершине добавляет к своему аргументу определения в данной вершине и убирает все определения в ту же переменную:

$$\forall v \in V \quad f_v(x) = (x \setminus KILL_v) \cup D_v,$$

где $KILL_v$ — множество определений, в которых присваивание происходит в ту же переменную, что и в вершине v, и D_v — множество определений в v.

Задача о восходящем анализе использования переменных является обратной задачей. Ее решение предоставляет информацию о том, какие определения достигают использование в данной точке программы. Полурешетка для этой задачи совпадает с полурешеткой задачи о достигающих определениях за исключением того, что A — множество использований. Потоковая функция для данной вершины убирает из аргумента все использования переменных, которые стоят в левых частях присваиваний в вершине, и добавляет переменные, использующиеся в ней:

$$\forall v \in V \quad f_v(x) = (x \setminus D_v) \cup U_v$$

3 Анализ зависимостей

В данном разделе описаны существующие подходы к анализу зависимостей.

3.1 Граф зависимостей

Определение 10 *Пусть G — граф потока управления. Будем говорить, что исполнение оператора S_1 предшествует исполнению оператора S_2 (обозначается как $S_1 \lhd S_2$), если существует путь в графе G из вершины, содержащей оператор S_1, в вершину, содержащую оператор S_2.*

Определение 11 *Обычно рассматривают три типа зависимостей по данным между двумя операторами S_1 и S_2. Пусть $S_1 \lhd S_2$. Тогда*

1. *если в S_1 происходит присваивание в переменную, которая затем используется в S_2, тогда говорят, что имеет место потоковая (или истинная) зависимость, и пишут $S_1\ \delta^f S_2$;*

2. *если в S_1 используется переменная, которая присваивается в S_2, то это — анти-зависимость, которая обозначается как $S_1\ \delta^a S_2$;*

3. *если и в S_1, и в S_2 происходит присваивание в одну и ту же переменную, то говорят, что это — выходная зависимость, и пишут $S_1\ \delta^o S_2$.*

Определение 12 *Пусть $G = (V, E)$ — граф потока управления, в котором каждой вершине соответствует только один оператор исходной программы. Графом зависимостей по данным называется граф $G' = (V, E')$, где E' — множество ребер такое, что для каждого ребра (w, v) оператор S_2, содержащийся в вершине v, зависит от оператора S_1 в вершине w, и $S_1 \lhd S_2$.*

Граф зависимостей строится на основе решения задачи о достигающих определениях и задачи о восходящем анализе использования переменных. При этом решение первой используется для обнаружения потоковых и выходных зависимостей, а решение второй для обнаружения анти-зависимостей.

3.2 Дэг зависимостей для элементарного блока

Одним из методов планирования инструкций для элементарного блока является метод *спискового планирования*. Его применение требует наличия построенного графа зависимостей, который накладывает ограничения на построение расписаний. Элементарный блок не содержит циклов, поэтому граф зависимостей всегда является дэгом[1]. Зависимости

[1]От англ. слова DAG — Directed Acyclic Graph.

между инструкциями в элементарном блоке могут возникать (вдобавок к описанным выше) в случаях когда:

1. невозможно определить, можно ли переместить инструкцию (возникает, например, когда операция загрузки следует сразу же за операцией выгрузки, которые используют различные регистры, — в этом случае мы не можем определить, будут ли перекрываться области памяти);

2. инструкции «соревнуются» в одновременном получении одного и того же вычислительного ресурса.

При построении дэгов зависимостей для элементарного блока анализ потока данных не требуется, т.к. блок не содержит циклов.

3.3 Анализ зависимостей в циклах

Анализ зависимостей в циклах имеет несколько иной характер, хоть и ту же природу. Задача анализа — выявить зависимости между итерациями гнезда циклов (вложенных циклов). Зависимость между итерациями I_1 и I_2 возникает, если существует зависимость между оператором в итерации I_1 и оператором в итерации I_2. Основной интерес при данном анализе представляют вырезки. Существует несколько подходов для анализа вырезок [12]. Достаточно распространено представление зависимостей между итерациями в виде векторов дистанции и направления[2]. *Вектор дистанции* для гнезда из k циклов — k-мерный вектор $\overrightarrow{d} = \langle d_1, ..., d_k \rangle$, в котором d_i — целые числа. Вектор дистанции указывает, что для любой итерации $\overrightarrow{i}$ итерация $\overrightarrow{i} + \overrightarrow{d}$ зависит от $\overrightarrow{i}$, т.е. $\overrightarrow{d} = \overrightarrow{i_1} - \overrightarrow{i_2}$ для некоторых итераций i_1 и i_2. *Вектор направления* — аппроксимация векторов дистанции, и вычисляется он следующим образом: $\overrightarrow{d} = sign(\overrightarrow{i_1} - \overrightarrow{i_2})$.

Для того, чтобы находить зависимости между вырезками внутри итераций и на разных итерациях, необходимо решить систему неравенств (чаще всего линейных). Приведём пример проверки обнаружения зависимостей внутри итерации. Для того, чтобы выяснить, есть ли зависимость между двумя операторами цикла, изображенного на рис. 1, необходимо решить следующую систему:

$$\begin{cases} 2i + 1 = 3i - 5 \\ 1 \leqslant i \leqslant 4. \end{cases}$$

[2]Distance and directions vectors

```
for(int i = 1; i <= 4; i++)
{
    b[i] = a[3*i-5] + 2;
    a[2*i+1] = 1/i;
}
```

Рис. 1: Цикл без межитерационных зависимостей.

Из первого уравнения очевидно получается $i = 6$, что не удовлетворяет неравенству.

В общем случае нахождение зависимостей требует решения системы диофантовых уравнений и неравенств. Эта задача NP-полна [2] (сводится к целочисленному программированию), поэтому все существующие алгоритмы предоставляют неточное решение.

Существует довольно много библиотек, реализующих алгоритмы обнаружения зависимостей в цикле. Например, GCD test (greatest common divisor) [7], Delta test [8], Fourier-Motzkin test [9] и [10], Omega test [11] и еще достаточно других с разной степенью эффективности и вычислительной сложностью.

3.4 Символьный анализ

Символьный анализ [13] — метод анализа программ, при котором информация извлекается путем манипулирования символьными выражениями (например, сравнивая их и/или интерпретируя абстрактно). Символьный анализ служит промежуточным шагом в оптимизации программ, он не выполняет никаких преобразований, но «добывает» знания о смысле программы с целью увеличения производительности последующих стадий. Так, результаты символьного анализа используются, например, при распространении констант, предсказании производительности, обнаружении зависимостей — в этом случае анализ, сравнивая символические подвыражения, предоставляет информацию о том, какие доступы к массивам перекрываются по памяти, а какие нет.

4 Реализация

PRANLIB является библиотекой, в которой собраны различные алгоритмы анализа потока управления. Разработка библиотеки

осуществляется на языке Objective Caml[3] [14].

Objective Caml является функциональным языком, поэтому ему присущи такие черты как выразительность (по сравнению с императивными языками), удобная работа со ссылочными типами данных, статическая типизация, благодаря которой значительно снижается процент ошибок при разработке, и многие другие. Наличие мощной модульной системы облегчает разработку крупных проектов, позволяя разбить их на части, компилируемые раздельно, облегчает понимание программ, увеличивает степень переиспользования кода и удобство сопровождения, предоставляя возможность реализации параметризованных модулей — функторов, делает программы более безопасными при помощи механизма описания типов модулей (интерфейсов).

4.1 Представление программы

Решение двух задач анализа потока данных и построение графа зависимостей производятся на абстрактном представлении программы — графе потока управления, в вершинах которого содержится два множества: Def — множество определяемых переменных и Use — множество используемых переменных. Переменные в множествах хранятся в виде номеров. Такое представление содержит минимум необходимой информации для решения поставленных задач.

4.2 Анализ потока данных

В ходе разработки архитектуры было принято решение обобщить реализацию задач анализа потоков данных с целью достичь наибольшей степени переиспользования кода. Так, было решено создать абстракцию программы и задач анализа в виде следующих модулей и типов модулей (интерфейсов).

При реализация какой-либо задачи анализа прежде всего нужно описать модуль полурешетки, который должен удовлетворять типу:

[3]Язык Objective Caml разрабатывается в Institut National de Recherche en Informatique et en Automatique (INRIA)

```
module type Semilattice.Sig =
  sig

    type t

    val cap : t -> t -> t
    val top : t
    val bottom : t
    val equal : t -> t -> bool

  end
```

В приведенном типе модуля `t` — тип элементов полурешетки, `cap` — операция пересечения $\sqcap$, `top` и `bottom` — наибольший и наименьший элементы, `equal` — функция проверки элементов на равенство.

Тип модуля **Repr** представляет общий интерфейс представления программы в виде графа потока управления:

```
module type Repr =
  sig

    type node
    type edge

  end
```

Здесь `node`, `edge` — типы информации, содержащейся в вершинах и на ребрах ГПУ.

Для представления информации о потоковых функциях используется **Adapter**.

```
module type Adapter =
  sig

    module P : Repr

    module L : Semilattice.Sig

    type edge = P.edge * L.t
    type flow = edge list * edge list -> L.t list * L.t list

    val flow : P.node -> flow
    val init : P.node ->
        (P.edge list * P.edge list -> L.t list * L.t list)

  end
```

Потоковая информация хранится на дугах — каждая дуга представляется в виде пары, первый элемент которой содержит значение дуги в каком-то представлении, описываемом модулем `P` и имеющем тип `Repr`, а второй является элементом полурешетки.

Разные задачи анализа потока данных по-разному преобразуют потоковую информацию, поступающую в рассматриваемую вершину ГПУ. В общем случае потоковая функция модифицирует информацию на входящих и исходящих дугах (например, при двунаправленном анализе). Для обобщения представления потоковых функций вводится тип `flow`. Этот тип описывает функции, аргументами которых являются пары из списков входящих и исходящих дуг, а возвращаемыми значениями — пары списков элементов полурешетки, которые обновляют информацию на дугах. Функция `init` используется для получения начальной разметки.

С целью облегчения реализации модуля для однонаправленного анализа (прямого или обратного) был создан интерфейс **UniAdapter** и два функтора — **ForwardAdapter** и **BackwardAdapter**, которые строят по модулю, реализующему **UniAdapter**, модуль с типом **Adapter** для прямого или обратного анализа соответственно.

```
module type UniAdapter =
  sig

    module P : Repr

    module L : Semilattice.Sig

    val flow : P.node -> (L.t -> L.t)
    val init : P.node -> L.t

  end
```

В приведенном типе модуля функция `flow` по вершине ГПУ возвращает потоковую функцию.

Обычно задача анализа использует не всю информацию о программе, а лишь ее часть, представленную в виде какой-либо абстракции. Например, при решении задачи о достигающих определениях из всех операторов программы важны лишь определения. Интерфейс **Abstractor** предоставляет функциональность, благодаря которой анализ может пользоваться абстрактным представлением, вместо конкретного.

```
module type Abstractor =
  sig

    module Concrete : Repr

    module Abstract : Repr

    val node : Concrete.node -> Abstract.node
    val edge : Concrete.edge -> Abstract.edge

  end
```

Пара функций `node` и `edge` реализуют отображение конкретного представления программы в абстрактное представление какой-либо задачи.

Теперь перейдем к описанию главного типа модуля абстракции программы — **ProgramView**. Он имеет следующий вид:

```
module type Sig =
  sig

    module Adapter : Adapter

    module Abstractor : Abstractor with
      type Abstract.node = Adapter.P.node and
      type Abstract.edge = Adapter.P.edge

    module L : Semilattice.Sig with
      type t = Adapter.L.t

    module G : CFG.Sig with
      type Node.info = Abstractor.Concrete.node and
      type Edge.info = Abstractor.Concrete.edge

    val flow : G.Node.info -> Adapter.flow
    val init : G.Node.info ->
        (Adapter.P.edge list * Adapter.P.edge list ->
                    Adapter.L.t list * Adapter.L.t list)

  end
```

Данный тип модуля содержит всю необходимую информацию для проведения анализа. Функтор `ProgrpamView.Make` конструирует модуль, имеющий приведенный тип, в котором функция `flow` возвращает потоковую функцию, используя `Abstractor`:

```
let flow n = Adapter.flow (Abstractor.node n)
```

Параметризованный тип модуля `DFAEngine.Generic` представляет интерфейс для получения решения задачи потока данных.

```
module Generic (P : ProgramView.Sig) (O : Order.Sig with module G = P.G) :
  sig

    val get : P.G.Edge.t -> P.L.t

  end
```

Функтор принимает абстракцию программы `P` и модуль порядка вершин (нумерацию) `O`, в соответствии с которым происходит обход графа потока управления. Функция `get` получает решение для данной дуги.

Модуль `DFAEngine` содержит так же специализированные алгоритмы итерирования: для обхода в порядке обратной нумерации [3]; в порядке, обратном порядку обратной нумерации; и для двунаправленных обходов — сначала в порядке обратной нумерации, потом в обратном порядке; и наоборот.

При представлении потоковых задач, описанном выше, достигается значительное переиспользование кода на разных уровнях.

Существуют классы задач анализа потока данных, которые используют одно и то же абстрактное представление программы. Такими задачами, например, являются: достигающие определения, живые переменные, задача о восходящем анализе использования переменных. Организация вышеприведенных модулей анализа потока данных позволяет переиспользовать любую реализацию интерфейса **Repr** в задачах с одинаковым представлением программы.

Из описания задач анализа в разделе 2.2 видно, что существуют задачи, использующие одинаковые полурешетки. Такое переиспользование возможно очевидным образом из-за инкапсуляции полурешетки в отдельный модуль.

Алгоритмы итерирования по своей сути не привязаны ни к какой задаче анализа потока данных, поэтому модуль итерирования был реализован независимым от представления задач.

При реализации задачи о восходящем анализе использования переменных были переиспользованы модули задачи о достигающих определениях, а именно модули полурешетки и абстрактного представления и модуль реализации потоковых функций.

4.3 Построение графа зависимостей

Задачи о достигающих определениях и о восходящем анализе использования переменных были реализованы классическим способом — с использованием битовых векторов в качестве элементов полурешетки и представлением потоковых функций в виде пары *gen* и *kill* [2, 1]. Модуль представления для обеих задач выглядит следующим образом:

```
type info = { gen : Bvect.t; kill : Bvect.t }

module Repr =
  struct
    type node = info
    type edge = unit
  end
```

Здесь `Bvect` — модуль битовых векторов. Элементы полурешети выглядят так (тоже одинаковы для обеих задач):

```
type elem = Vect of Bvect.t | Top
```

Операции полурешетки также описаны классическим способом. Модуль, реализующий тип **UniAdapter**, организован очевидным образом — функция `flow` по вершине `node` представления возвращает функцию потока управления, которая имеет вид:

```
f(x) = node.gen | (x & not node.kill)
```

Для применения анализов к разработанному абстрактному представлению были реализованы два модуля типа **Abstractor**. Для реализации этих модулей строятся два отношения «вершина — `gen`, `kill`», которые затем используются для построения функции `Abstractor.node`. В первом из них биты отвечают определениям (для задачи о достигающих определениях), во втором — использованиям (для задачи о восходящем анализе использования переменных).

После реализации всех необходимых модулей, можно приступить к конструированию модуля алгоритма итерирования. Выглядит это примерно так:

```
let module T = DFST.Make (G) in
let module RDAdapter = ProgramView.ForwardAdapter (RD.Adapter) in
let module RDPV = ProgramView.Make (RDAdapter) (RDAbstr) (G) in
let module RDEngine = DFAEngine.Post (RDPV) (T) in
...
(* далее следует разбор результатов решения,
   с использованием RDEngine.get *)
```

Здесь `G` — модуль ГПУ, `DFST` — модуль глубинно-остовного дерева, `RD.Adapter` — модуль, реализующий **UniAdapter**, а `RDAbstr` имеет тип **Abstractor**.

Подобное конструирование модулей происходит и для задачи о восходящем анализе использования переменных.

Теперь опишем способ построения графа зависимостей. После решения задачи о достигающих определениях в каждой вершине графа содержится информация о том, какие определения достигают данную вершину.

Для получения информации о том, какие переменные (и в каких вершинах) использовались до прихода потока управления в

рассматриваемую вершину, использовалось решение «развернутой» задачи о восходящем анализе использования переменных, т.е. обход в ходе итерирования осуществлялся как в прямой задаче — по направлению дуг, вместо обхода против направления.

На основе этой информации довольно очевидным образом строится граф зависимостей.

Вершины графа зависимостей совпадают с вершинами графа потока управления. Будем помечать дуги следующим образом: `Flow` — для потоковой зависимости, `Anti` — для анти-зависимости и `Output` — для выходной зависимости. Рассмотрим вершину u.

1. Для каждого определения d_i, достигающего u и производящего присваивание в переменную v, проверяем, есть ли использование v в u, если да, то добавляем `Flow` дугу с началом в вершине, содержащей d_i, и концом в u. Теперь, если вершина u содержит присваивание в переменную v, то добавляем `Output` дугу с такими же началом и концом как и в предыдущем случае.

2. Если u содержит определение переменной v, то для каждого использования переменной v, которое достигает (в смысле решения «развернутой» задачи о восходящем анализе использования переменных) вершину u, добавляем `Anti` дугу с началом в вершине, содержащей использование v, и концом в u.

Выше был рассмотрен случай переменной. Обобщение построения графа зависимостей с учетом вырезок требует реализации анализа указателей, который позволяет определить, какие ссылки на массив указывают на одну и ту же область памяти, или анализа индексов вырезки (такие подходы описаны в разделе 3.3). В рамках данной работы вырезки оцениваются консервативным образом, т.е. весь массив трактуется как одна переменная.

5 Применение

В данном разделе описываются примеры использования анализа зависимостей и представления анализа потока данных.

5.1 Простой пример

Для апробации проделанной работы было разработано простое абстрактное представление программы. В данном представлении в

каждой вершине графа потока управления содержится один из операторов следующего вида:

1. `Assgn(num, expr)`: присваивание, где источником является выражение `expr`, а приемником — переменная с номером `num`.
2. `Cond expr`: представление выражения в условиях (для ветвлений и циклов).

В свою очередь выражение `expr` может быть:

1. `Unary(op, expr)`: унарная операция, где `op` — непосредственно сама операция;
2. `Binary(op, expr, epxr)`: бинарная операция;
3. `Var num`: переменная с номером `num`;
4. `Const num`: константа с номером `num`;
5. `Call(num, expr_list)`: вызов процедуры с номером `num` и списком параметров `expr_list`.

Рассмотрим применение алгоритма на следующем примере. Пусть дана программа:

```
1. i = 0;
2. while(i < 10)
   {
3.     j = i + 1;
4.     if(i % 2 == 0)
       {
5.         k = 0;
       }
       else
       {
6.         k = 1;
       }
7.     i = f(i, j, k);
   }
8. z = 0;
```

Представление данной программы в виде графа потока управления изображено на рис. 2. Граф зависимостей по данным, который был построен для программы, приведен на рис. 3.

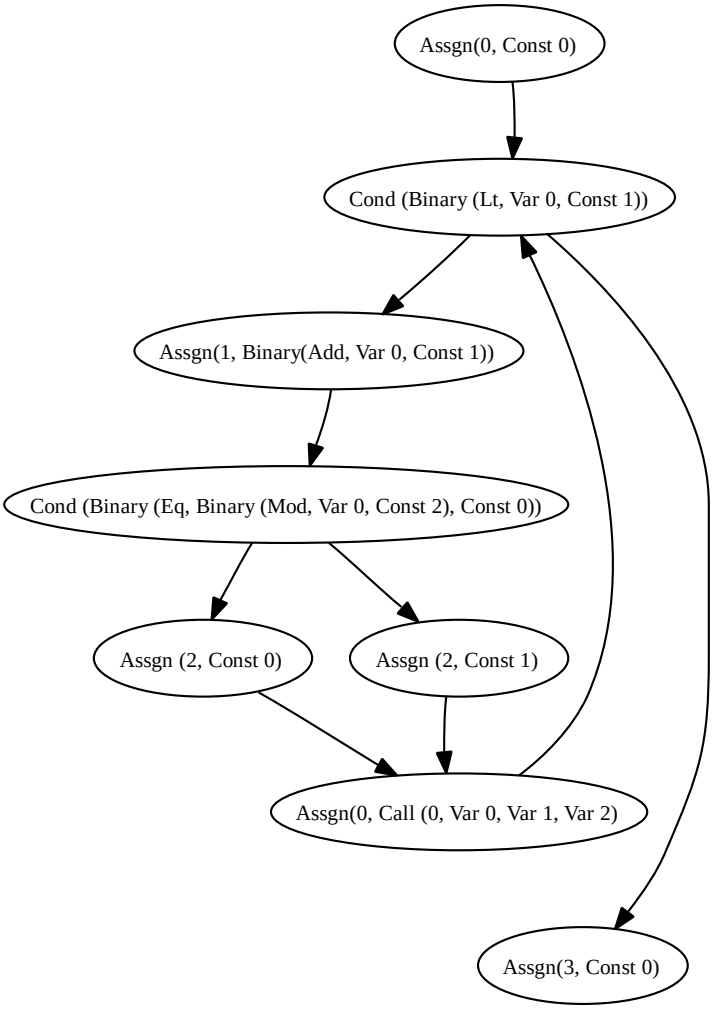

Рис. 2: Граф потока управления.

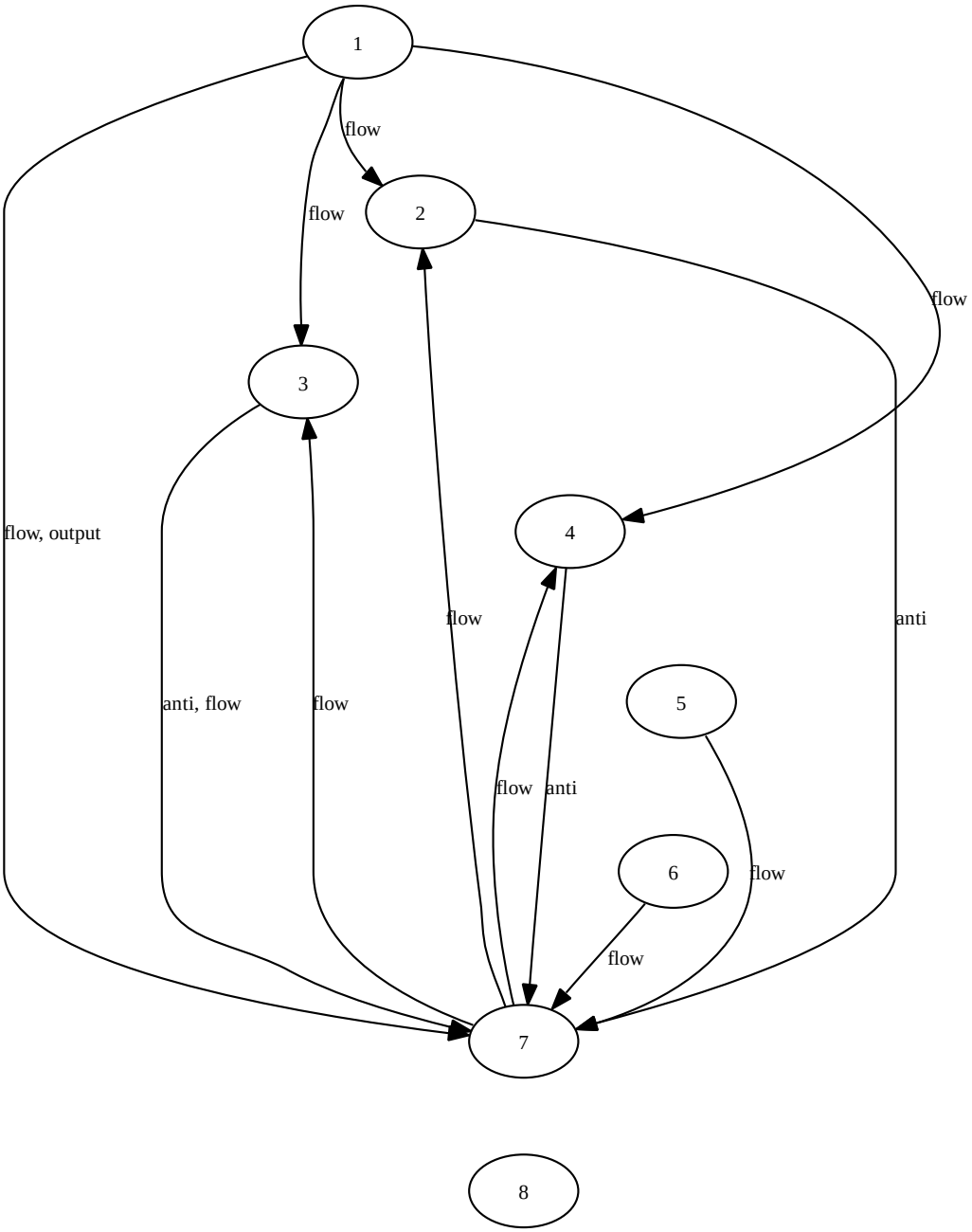

Рис. 3: Граф зависимостей.

5.2 Применение представления анализов потока данных

Разработанное представление задач анализа потока данных было использовано при реализации анализа указателей в проекте Creen.

Данный проект направлен на создание коллекции анализов программ, написанных на языке C. Разработка ведется на языке Objective Caml. Для построения графа зависимостей и других внутренних представлений используется открытый компилятор языка C CIL [15]. Взаимодействие между CIL и PRANLIB возможно, благодаря наличию в библиотеке модуля конвертации графа из CIL в граф библиотеки.

Реализация модуля представления задачи анализа указателей (**Repr**), модуля абстракции конкретного представления (**Abstractor**), модуля полурешетки анализа и модуля потоковых функций (**Adapter**) заняло 265 строчек исходного кода.

Для реализация задач о достигающих определениях и о восходящем анализе использования переменных на представлении программы, описанном в разделе 5.1, потребовалось 300 строчек кода.

6 Заключение

В ходе данной работы было разработано представление задач анализа потока данных для библиотеки PRANLIB. Организация представления позволяет достичь значительной степени переиспользования кода, что было продемонстрировано в ходе реализации задачи о достигающих определениях и задачи о восходящем анализе использования переменных для абстрактного представления программы. Эти задачи были использованы при разработке алгоритма построения графа зависимостей по данным, который теперь также является частью библиотеки.

Реализованные механизмы анализа потока данных уже используются в проекте Creen, направленном на разработку коллекции анализов программ на языке C.

Список литературы

[1] A.Aho, R.Sethi and J.Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986.

[2] Steven S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, July 1997.

[3] Касьянов В.Н., Евстигнеев В.А. Графы в программировании: обработка, визуализация и применение. Санкт-Петербург, БХВ-Петербург.

[4] M.Wolfe and U.Banerjee. Data Dependence and its application to Parallel Processing. International Journal of Parallel Processing, 16(2):137—178, 1987.

[5] Laurent Hascoët. The Data-Dependence Graph of Adjoint Programs. INRIA, BP93, 06902 Sophia-Antipolis, France, April 2001.

[6] http://oops.tercom.ru/projects/pranlib/

[7] Banerjee, Uptpal. Dependence Testing in Ordinary Programs. M.S. thesis. Dept. of Comp. Sci., University of Illinois, Urbana-Champaign, IL, November 1976.

[8] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical Dependence Testing. Proc. of the SIGPLAN '91 Symp. on Programming Language Design and Implementation, Toronto, Ontario, published as SIGPLAN Notices, Vol. 26, No. 6, June 1991.

[9] George Dantzig, and B.C. Eaves. Fourier-Motzkin Elimination and Its Dual, J. of Combinatorial Theory A, Vol. 14, 1973.

[10] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. An Efficient Method for Exact Dependence Analysis. Proc. of the SIGPLAN '91 Symp. on Programming Language Design and Implementation. Toronto, Ontario, published as SIGPLAN Notices, Vol. 26, No. 6, June 1991.

[11] William Pugh, and David Wannacott. Eliminating False Data Dependences Using the Omega Test. Proc. of the SIGPLAN '92 Symp. on Programming Language Design and Implementation. San Francisco, CA, published as SIGPLAN Notices, Vol. 27, No. 7, July 1992.

[12] Dror E. Maydan, Saman P. Amarasinghe and Monica S. Lam. Data Dependence and Data-Flow Analysis of Arrays. Computer System Laboratory, Stanford University, CA 94305, 1992.

[13] Paul Havlak. Interprocedural Symbolic Analysis. A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree, Rice University, Houston, Texas, May 1994.

[14] http://caml.infria.fr

[15] http://manju.cs.berkeley.edu/cil/