Дерево значений: новая структура данных, объединяющая информацию о потоке управления и доминировании

Дроздов А.Ю., Боханко А.С.
ИМВС РАН, г. Москва
{sasha|ruff}@mcst.ru

Аннотация

Существует класс оптимизаций, для проведения которых необходима информация как о потоке данных, так и о доминировании. Использование для представления этой информации нескольких разных аналитических структур данных затрудняет реализацию упомянутых оптимизаций и может наложить ограничения на регион их применения. В работе прелагается новая структура данных, объединяющая в себе информацию о потоке данных и доминировании. Ее использование позволит упростить реализацию и одновременно расширить регион применения оптимизаций.

1. Введение

Для современного оптимизирующего компилятора большое значение имеют используемые промежуточное представление и аналитические структуры данных. Простые и одновременно мощные структуры данных позволяют проводить эффективные оптимизирующие преобразования, хорошо спроектированное промежуточное представление облегчает построение аналитических структур данных.

В настоящей работе используется представление программы в виде формы с единственным присваиванием (Static Single Assignment form, далее SSA-форма) [1, 2]. В программе, представленной в SSA-форме, любая переменная может быть определена только один раз. Для того чтобы соблюсти это ограничение и, тем самым, перевести программу в SSA-форму, необходимо выполнить следующие действия:

- Разместить в точках схождения потока управления φ-узлы. φ-узел для некоторой переменной – это операция, выбирающая среди множества значений переменной нужное.
- Переименовать все переменные, так чтобы каждому определению соответствовала своя, уникальная переменная.

На рис. 1 приведен пример перевода программы в SSA-форму (слева исходная программа, справа – программа, представленная в SSA-форме).

```
if (...) {
  foo = A;
} else
{
  foo = B;
}
bar = foo;
```

```
if (...) {
  foo1 = A;
} else
{
  foo2 = B;
}
foo3 = φ(foo1, foo2);
bar = foo3;
```

Рисунок 1. Пример перевода программы в SSA-форму

SSA-форма позволяет представить поток данных в удобном и компактном виде. Базовой формой выражения потока управления является граф потока управления, или, как его еще называют, управляющий граф [3]. Управляющий граф разбивает программу на линейные участки, внутри которых поток управления линеен. Таким образом, весь более сложный,

нелинейный поток управления оказывается выражен дугами управляющего графа, соединяющими линейные участки между собой.

Отношения доминирования между отдельными линейными участками можно представить в виде дерева доминаторов [4]. Линейный участок, расположенный выше в дереве доминаторов, доминирует (то есть встречается в потоке управления всегда раньше) над всеми линейными участками, расположенными ниже него.

Существует класс оптимизаций, для проведения которых необходима информация как о потоке данных, так и об отношении доминирования. В настоящей работе предлагается новая структура данных, объединяющая информацию о потоке данных и доминировании, необходимую упомянутому классу оптимизаций. В следующих разделах будет описана предлагаемая структура данных, приведен алгоритм ее построения, доказана его корректность и дана оценка сложности. Также будут представлены результаты тестирования и приведен пример оптимизаций, использующих предлагаемую структуру данных.

2. Дерево значений

Основной идеей новой структуры данных, которую предлагается назвать *деревом значений*, является объединение дерева доминаторов и операций и φ-узлов программы, представленной в SSA-форме, в одно целое. В результате такого объединения получается дерево, узлы и дуги которого совпадают с узлами и дугами дерева доминаторов, а в узлах хранятся вектора упорядоченных особым образом операций и φ-узлов.

В векторе, прежде всего, хранятся операции и φ-узлы, принадлежащие соответствующему линейному участку. Порядок хранения такой: сначала все φ-узлы (в произвольном порядке), затем операции. Порядок операций определяется их порядком в линейном участке.

Для переменных, всю работу с которыми можно точно описать статически, φ-узлов и операций одного линейного участка достаточно. Но есть переменные, работу с которыми точно описать статически невозможно. Это составные переменные (структуры, объединения, массивы), глобальные переменные, и переменные, на которые брался адрес.

Например, при статическом анализе программы, приведенной на рис. 2, невозможно сказать, вызовет ли запись по указателю `p` изменение значения переменной `foo`. То есть значение переменной `foo`, на которую брался адрес, может быть изменено *неявным* образом. Точно так же операция вызова функции может неявным образом изменить значение любой глобальной переменной, так как априори, не проведя межпроцедурный анализ, мы не можем утверждать, что она не изменит значения произвольной глобальной переменной. Но и межпроцедурный анализ может дать ответ лишь о *вероятном*, а не об *обязательном* изменении глобальной переменной. Индекс доступа к составной переменной часто вычисляется динамически, и одна и та же операция может обращаться к разным элементам составной переменной, или даже к разным составным переменным. То есть результат такой операции в разные моменты времени может определять разные переменные.

```
foo = bar = A;
p = &bar;
if (...)
    p = &foo;
*p = B;
```

Рисунок 2. Пример неявного изменения значения переменной

Далее, для краткости, мы будем называть переменные, работу с которыми невозможно точно описать статически, “не-SSA переменными”, понимая, впрочем, что это название очень условно и не совсем корректно.

Если в программе есть не-SSA переменные, то для полноты дерева значений необходимо, чтобы для любой операции O, читающей некоторую не-SSA переменную или множество не-SSA переменных, все операции, которые *могут* изменить значение читаемых переменных и встретиться в потоке управления ранее, входили или в вектор того узла, которому принадлежит O (причем встречались в этом векторе раньше нее), или в вектор одного из доминирующих узлов.

Для тех операций, что доминируют над O и могут изменить значения читаемых ею не-SSA переменных, никаких дополнительных действий проводить не надо – они и так гарантированно будут входить или в вектор узла дерева значений, которому принадлежит O, или в вектор одного из доминирующих узлов. Однако, есть еще операции, которые не доминируют над O, но, тем не менее, могут встретиться в потоке управления раньше нее. Известно, что в итерационный фронт доминирования (Iterated Dominance Frontier, далее IDF) [5] таких операций входит или сам узел O, или один из доминирующих над ним узлов. Поэтому достаточно для каждой операции O′, способной неявным образом изменить значения одной или нескольких переменных, и входящей в линейный участок N′, добавить операцию O′ в вектора всех узлов, входящих в IDF линейного участка N′.

Порядок таких недоминирующих операций в векторах значений также важен, мы подробно остановимся на этом при описании алгоритма.

Напомним, что IDF некоторого линейного участка N определяется как объединение IDF линейных участков, входящих в DF N. Линейный участок N′ входит в DF N, если N доминирует над одним из предшественников N′, но не доминирует (строго) над N′.

После добавления недоминирующих операций в векторах могут встречаться два вида операций: доминирующие и недоминирующие (φ-узлы всегда доминируют). Причем одна и та же операция может присутствовать и как доминирующая, так и как недоминирующая. Например, операция O с рис. 3 входит в вектор узла N1 как недоминирующая, а в вектор узла N2 как доминирующая.

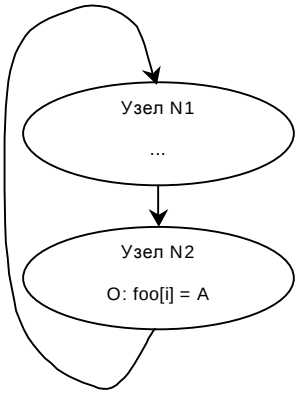

Рисунок 3. Пример вхождения одной операции в два вектора

Разумно в каждом элементе вектора кроме самой операции хранить также тип (доминирующая или не доминирующая) этой операции.

Дерево значений позволяет для любой операции O получить все операции и φ-узлы, способные изменить значение переменных, читаемых O, и встречающиеся в потоке управления раньше O; больше того, узнать информацию о доминировании этих операций над O. Таким образом, дерево значений объединяет в себе информацию о потоке данных (причем и для не-SSA переменных) и доминировании. В последующих разделах настоящей работы будет показано, как с помощью одного дерева значений можно проводить некоторые оптимизации, ранее требовавшие совместного использования нескольких аналитических структур данных.

На рис. 4 дерево значений представлено в наглядном виде.

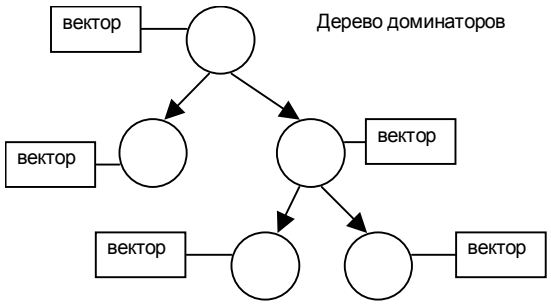

Рисунок 4. Схема дерева значений

3. Алгоритм построения дерева значений

Как было отмечено ранее, основным свойством дерева значений является то, что для любой операции O, все операции и φ-узлы, способные изменить значения переменных, читаемых O, и встречающиеся в потоке управления раньше O, входят в дерево значений также раньше O (то есть или входят в тот же вектор, что и O, но раньше, чем O, или входят в вектор одного из узлов, доминирующих над узлом, в который входит O).

Алгоритм построения дерева значений определяется необходимостью соблюдения этого свойства. Для обычных переменных алгоритм тривиален: все, что нужно сделать – это создать в каждом линейном участке вектор, и заполнить созданный вектор φ-узлами и операциями линейного участка в том порядке, в котором они следуют в SSA-форме.

Для не-SSA переменных необходим дополнительный шаг: добавление операций, способных изменить (явно или неявно) значения таких переменных в начало векторов узлов, входящих в IDF этих операций. При этом сами операции следует обходить в порядке уменьшения их номеров в RPO-нумерации (Reverse Post Order numbering, [6]).

Введем несколько служебных функций, реализация которых тривиальна и в значительной степени определяется используемым способом хранения структур данных.

Служебные функции с константной алгоритмической сложностью:

`MaxRPONumber` (получение максимального RPO-номера)
`GetNodeByRPONum` (получение линейного участка по его RPO-номеру)
`SetValuesVector` (задание вектора значений заданного узла)
`GetValuesVector` (получение вектора значений заданного узла)
`append` (добавление элемента в конец вектора значений; первые два параметра задают вектор и операцию или φ-узел, третий – признак доминирования)
`insert` (добавление элемента в начало вектора значений; первые два параметра задают вектор и операцию или φ-узел, третий – признак доминирования)

Служебные функции с линейной алгоритмической сложностью:

`Nodes` (получение множества узлов дерева)
`SSANodes` (получение множества операций и φ-узлов линейного участка в порядке их следования в SSA-форме)
`NonSSAOpers` (получение множества операций линейного участка, которые могут изменить значения не-SSA переменных, в порядке из следования в SSA-форме)
`IDF` (получение множества узлов, входящих в IDF заданного линейного участка)

На рис. 5 представлен предлагаемый алгоритм построения дерева значений, написанный на алгоритмическом псевдоязыке, и легко переводимый в любой современный язык, поддерживающий структурное программирование.

```
1:      func BuildValuesTree(cfg, dom_tree) {
2:        foreach n in Nodes(dom_tree)
3:          CreateValuesVector(n);
4:        endfor;
5:
6:        for (i = MaxRPONumber(cfg); i > 0; i = i - 1)
7:          n = GetNodeByRPONum(i);
8:          InsertDefOpersIDF(n);
9:        endfor;
10:     }

11:     func CreateValuesVector(node) {
12:       v = {};
13:       foreach n in SSANodes(node)
14:         v = append(v, n, TRUE);
15:       endfor;
16:       SetValuesVector(node, v);
17:     }

18:     func InsertDefOpersIDF(node) {
19:       foreach o in NonSSAOpers(node)
20:         foreach n in IDF(node)
21:           v = GetValuesVector(n);
22:           v = insert(v, o, FALSE);
23:         endfor;
24:       endfor;
25:     }
```

Рисунок 5. Алгоритм построения дерева значений

Главная функция, собственно строящая дерево значений – `BuildValuesTree`. Она получает в качестве параметров управляющий граф и дерево доминаторов. Функция `CreateValuesVector` создает вектор значений и помещает в созданный вектор φ-узлы и операции линейного участка в порядке их следования в SSA-форме. Функция `InsertDefOpersIDF` добавляет операции линейного участка N, способные изменить одну или несколько не-SSA переменных, в начало векторов тех узлов, что входят в IDF N. Порядок обхода операций линейного участка N совпадает с порядком их следования в SSA-форме.

3.1. Доказательство корректности

Докажем, что алгоритм, предложенный в разделе 3, строит структуру данных, обладающую следующим свойством: для любой операции O, все операции и φ-узлы, способные изменить значения переменных, читаемых O, и встречающиеся в потоке управления раньше O, входят в дерево значений также раньше O (то есть или входят в тот же вектор, что и O, но раньше, чем O, или входят в вектор одного из узлов, доминирующих над узлом, в который входит O).

Известно, что если операция O′ может встретиться в потоке управления раньше O, то O′ либо доминирует над O, либо один из доминаторов O входит в IDF O′. Это утверждение непосредственно следует из определения IDF [5].

Те операции, что доминируют над O, добавляются в дерево значений в строках 2-3 алгоритма. Те же, в IDF которых входит один из доминаторов O, или добавляются в строках 2-3 (уже представленные φ-узлами в SSA-форме) или, для не-SSA переменных, добавляются в строках 5-8.

Что и требовалось доказать.

3.2. Оценка сложности

Алгоритм, предложенный в разделе 3, обладает алгоритмической сложностью $O(M_1 + M_2 * N)$, где M_1 – число операций и φ-узлов SSA-формы, M_2 – число операций, N – число линейных участков в управляющем графе.

Сложность алгоритма определяется действиями, проводимыми в строках с номерами 2-3 и 5-8.

В строках 2-3 происходит обход всех операций и φ-узлов SSA-формы, что дает слагаемое M_1, а в строках 5-8 операции, способные изменить значения не-SSA переменных, добавляются в вектора дерева значений согласно IDF соответствующих линейных участков, что дает слагаемое $M_2 * N$.

5. Результаты тестирования

Авторы реализовали дерево значений в рамках оптимизирующего компилятора проекта “Эльбрус 3М”; кроме того, нами было проведено тестирование размера новой структуры данных.

Тестирование проводилось на тестах пакета SPEC92, в который входят достаточно разнообразные тесты, написанные на языках C и Fortran.

В таблице 1 приведены результаты тестирования. Они показывают, что размер дерева значений, как правило, лишь незначительно превосходит число операций и φ-узлов процедуры, и не требует значительного объема памяти.

Таблица 1. Результаты тестирования размера дерева значений

	Название теста	Описание теста	Средний размер дерева значений процедуры	Отношение размера дерева значений к размеру SSA-формы
1	008.espresso	Минимизация булевских функций	272.07	1.28
2	022.li	Интерпретатор языка LISP	88.74	1.25
3	023.eqntott	Перевод булевских выражений в таблицу истинности	286.63	1.45
4	026.compress	Компрессор / декомпрессор	274.63	1.36
5	072.sc	Табличные вычисления	279.16	1.42
6	085.gcc	Компилятор	134.07	1.20
7	047.tomcatv	Генерация сетей	1713	1.15
8	052.alvinn	Модель нейросети	132.88	1.22
9	056.ear	Модель ушной полости	129.66	1.23
10	078.swm256	Модель водной поверхности	446.5	1.16
11	089.su2cor	Квантовая физика; моделирование методом Монте-Карло	904.28	1.48
12	090.hydro2d	Астрофизические вычисления	356	1.11
13	093.nasa7	7 математических программ	544.18	1.17
14	094.fpppp	Квантовая химия	2257.92	1.76

Под “размером дерева значений” понимается суммарное число элементов векторов, под “размером SSA-формы” – суммарное число операций и φ-узлов. Тесты 1-5 – целочисленные, 6-14 – с плавающей точкой.

5. Оптимизации, использующие дерево значений

В этом разделе мы приведем пример оптимизаций, использующих дерево значений. Все эти оптимизации хорошо известны, но до сего момента для их применения приходилось использовать несколько разных структур данных, вследствие чего регион действия оптимизаций часто был ограничен. Как мы покажем, использование дерева значений позволяет снять эти ограничения.

5.1. Замена одних вычислений на результат других: GCP, RLE

Задачи оптимизаций Глобальное распространение копий (Global Copy Propagation, далее GCP) и Удаление излишних чтений (Redundant Loads Elimination, далее RLE) похожи – замена одних вычислений на другие (выполняющиеся в потоке управления раньше). Основное их отличие заключается в том, что GCP работает с SSA-объектами, а RLE – с не-SSA объектами.

На рис. 6 приведен пример работы оптимизаций GCP и RLE (слева – программа до применения оптимизаций, справа – после).

```
for (i = 0; i < 100; i++) {
  a[i] = A;
  foo = i;
  b[foo] = a[i];
}
```

```
for (i = 0; i < 100; i++) {
  a[i] = A;
  foo = i;
  b[i] = A;
}
```

Рисунок 6. Пример применения оптимизаций GCP и RLE

Оптимизация GCP заменила чтения переменной `foo` при вычислении индекса массива `b` на чтение переменной `i`; оптимизация RLE заменила чтение элемента массива `a` на чтение переменной, значение которой ранее было записано в соответствующем элементе массива.

Для проведения GCP достаточно найти в дереве значений доминирующую операцию O (говоря “доминирующая”, мы имеем в виду вхождение в вектор с признаком доминирования), вырабатывающую то же значение, что читает операция O′, входящая в дерево значений ниже (то есть входящая в вектор одного из нижележащих узлов, или входящая в тот же вектор, что и O, но позже). Для определения равенства значений можно использовать анализ, известный как Нумерация значений (Value Numbering, [7]).

Для проведения RLE, кроме того, необходимо проверить, что O и O′ *всегда* работают с одинаковыми элементами одинаковой не-SSA переменной V (в случае массивов это можно сделать с помощью индексного анализа, [8]), и между O и O′ в дереве значений не встречаются операции, которые *могут* изменить значение V.

Так как дерево значений глобально для всей процедуры, то и проводить на ее основе оптимизации GCP и RLE также можно глобально. То есть рамки проведения этих оптимизаций ограничены только деревом значений.

Более того, использование дерева значений позволяет снять для оптимизации RLE требование доминирования *одной* операции над чтением не-SSA переменной, так как дерево значений позволяет получить множество *совокупно доминирующих* операций.

5.2. Удаление избыточных вычислений: DCE, RSE

Оптимизации Удаление мертвого кода (Dead Code Elimination, далее DCE) и Удаление изливших записей (Redundant Stores Elimination, далее RSE) в некотором смысле обратны оптимизациям GCP и RLE. Они удаляют операции, результаты которых нигде более не потребляются. Например, применение оптимизации GCP на программе с рис. 6 сделало операцию определения переменной foo ненужной; ее можно смело удалить с помощью оптимизации DCE. Точно таким же образом, только для не-SSA объектов, работает оптимизация RSE.

Также как и оптимизации, “обратным” должно быть и дерево значений. То есть оно должно быть основано на дереве постдоминаторов, в алгоритме построения обход узлов надо проводить в порядке увеличения их RPO-номеров, операции и φ-узлы нужно обходить в порядке, обратном

их порядку в SSA-форме, а функция dfa_NonSSAOpers должна возвращать (опять-таки, в обратном порядке) операции, которые могут прочитать значения не-SSA переменных.

Разумеется, применение дерева значений не ограничивается описанными выше оптимизациями; они были использованы нами только в качестве иллюстрации гибкости и мощи новой структуры данных.

Заключение

В настоящей работе была предложена новая структура данных – дерево значений. Представлен алгоритм построения дерева значений, доказана его корректность, дана оценка сложности.

Далее были приведены результаты тестирования, показывающие, что новая структура данных является достаточно компактной и поэтому может применяться для решения практических задач.

В заключение было показано, как дерево значений позволяет упростить реализацию нескольких хорошо известных оптимизаций, увеличив при этом мощность и регион применения как минимум некоторых из них.

Список литературы

1. Ronald Cytron, Jean Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck, “An Efficient Method of Computing Static Single Assignment Form”, Conference Record of the 16th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Austin, 1988, pp. 23-25

2. Ronald Cytron, Jean Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck, “Efficiently Computing Static Single Assignment Form and the Program Dependence Graph”, ACM TOPLAS, Vol. 13, No. 4, Oct. 1991, pp. 451-490

3. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, “Compilers: Principles, Techniques, and Tools”, Addison-Wesley, Reading, 1986, chapter 9.4

4. Thomas Lengauer, Robert Tarjan, “A Fast Algorithm for Finding Dominators in a Flowgraph”, ACM TOPLAS, Vol. 1, No. 1, July 1979, pp. 121-141

5. Vugranam C. Sreedhar, Guang R. Gao, “A Linear Time Algorithm for Placing φ-nodes”, Conference Record of the 22nd ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, San Francisco, 1995, pp. 62-73

6. Steven S. Muchnick, “Advanced Compiler Design and Implementation”, Morgan Kauffman, San Francisco, 1997, chapter 7.2

7. Loren Taylor Simpson, “Value-Driven Redundancy Elimination”, Ph.D. Thesis, Rise University, Houston, Texas, 1996

8. Utpal Banerjee, “Loop Transformations for Restructuring Compilers”, Vols. 1 – 3, Kluwer Academic Publishers, Boston, 1993-1997