МАТЕМАТИЧЕСКОЕ И АРХИТЕКТУРНОЕ ОБЕСПЕЧЕНИЕ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ
(Вычислительные системы)
1985 год Выпуск 109

УДК 519.665:681.324

АНАЛИЗ СВОЙСТВ ПРОГРАММ И ВЫДЕЛЕНИЕ НАБОРОВ ПЕРЕМЕННЫХ, ВЛИЯЮЩИХ НА ЭТИ СВОЙСТВА

С.А.Симонов

В в о д н ы е з а м е ч а н и я. Анализ свойств программ имеет важное значение, с одной стороны, как средство повышения уровня доверия к программам со стороны пользователя, а с другой - как средство явного выделения той информации о программе, которая позволяет преобразовывать и приспосабливать программу в соответствии с условиями ее использования. Такого анализа свойств требуют методы оптимизации, автоматического распараллеливания, конкретизации, обеспечения переносимости и слежения за точностью вычислений, обеспечения корректности программ.

Принято делить все свойства программ на статические, т.е. те, которые могут быть определены по тексту программы без его интерпретации, и динамические, требующие для своего определения интерпретации. Примерами свойств, определяемых статически, являются: отсутствие синтаксических ошибок в программе; количество операторов каждого типа; карта перекрестных ссылок при использовании идентификаторов, использование идентификаторов в каждом операторе (исходные данные, вводимые данные, фактические или формальные параметры, индексы и т.д.); наличие подпрограмм и функций, вызываемых в каждой программе; наличие неинициализированных переменных и переменных, которым были присвоены значения,но они никогда не использовались; наличие изолированных сегментов программ, которые не могут быть выполнены ни при каких входных данных;отклонения от стандартов программирования; неверное использование глобальных и локальных имен, переменных в общей области и в списке парамет - ров (неверное количество параметров, несоответствующие типы, неинициализированные входные параметры, выходные параметры, которым

не присвоены значения, выходные параметры, которым было присвоено значение, но оно нигде далее не используется, параметры, которые не используются ни как входные, ни как выходные и т.д.). Важной составной частью статического анализа является потоковый анализ программ, который традиционно разделяется на анализ потоков управления и анализ потоков данных. При этом анализ потоков управления кроме самостоятельного значения имеет и подчиненное, так как большинство задач анализа потоков данных невозможно без предварительного анализа потоков управления. Задачи и методы потокового анализа программ описаны в [5,6], задачи определения перечисленных выше статических свойств - в [I-3,II].

Примерами динамических свойств являются: обращение к несуществующим элементам массивов; наличие таких фрагментов, которые формально достижимы из начала программы, но нет такого набора входных данных, при котором эти фрагменты были бы выполнены; возможность замены некоторых условных операторов безусловными; наличие информационных зависимостей между отдельными операторами или фрагментами, такими как "витки" циклов (для простых переменных наличие информационных зависимостей может быть выявлено статически, а для переменных с индексами выявление зависимостей может быть сделано, в общем случае, только динамически); окончание циклов в программе; логико-термальная история выполнения программы; история использования программой данных и т.д. Эти и многие другие свойства, в общем случае, не могут быть выявлены без интерпретации программ в той или иной степени. Здесь степень интерпретации программ определяется как доля тех переменных, которые получают конкретные значения, и доля тех операторов, которые будут выполнены при интерпретации. Далее под интерпретацией оператора будет пониматься выполнение оператора в соответствии с его семантикой на конкретных данных. Для оператора присваивания интерпретация заключается в вычислении значения правой части и присваивании этого значения переменной из левой части, для оператора ввода - во вводе соответствующих значений. Интерпретация для условного оператора состоит в вычислении логического выражения и выполнении действий в зависимости от его значения.

Любые динамические свойства программы могут быть определены при исчерпывающем тестировании (т.е. при тестировании на всех возможных наборах входных данных). Действительно, проанализировав все возможные случаи исполнения программы, можно определить любые

ее свойства, но трудоемкость этого тестирования может быть фантастически огромна. Так, в [I] приведен пример несложного управляющего графа программы, для которого количество путей от начала до конца равно 10^{18}, а ведь этим не исчерпываются все наборы входных данных, так как по одному пути можно ходить с разными исходными данными, не влияющими на выбор пути.

Заметим, что для выявления определенных свойств программ не обязательно интерпретировать все операторы программы - надо интерпретировать лишь те операторы, от которых это свойство зависит и, очевидно, чем меньше операторов будет интерпретироваться, тем меньшее количество исходных данных будет использовано и тем большее количество тестовых случаев будет охвачено.

П о с т а н о в к а з а д а ч и и м е т о д е е р е ш е н и я. Возникает задача: как определить то минимальное количество интерпретируемых операторов, при котором полностью определяется требуемое свойство?

Для более формальной постановки задачи будем рассматривать исследуемые программы в виде последовательности операторов $S = (s_n, n=\overline{1,N})$, s_1 - первый оператор текста программы, s_N - последний. Обозначим все имена переменных программы a_m $(m=\overline{1,M})$. С каждым оператором программы s_n свяжем пару множеств имен переменных: $R(s_n)$ - множество имен переменных, значения которых используются при исполнении s_n, и $W(s_n)$ - множество имен переменных, получающих новое значение в результате исполнения s_n. Например, для оператора A[L+1,K]:=A[L+1,K] + SIN(K) * B множества R и W будут соответственно {L,K,A,B} и {A}. Тогда задача состоит в следующем.

Выбрать $S' \subseteq S$ $(S'=(s'_n, n=\overline{1,N'}))$ такое, что: а) каждому пути по программе S взаимно однозначно соответствует путь по программе S', б) операторы S' вычисляют все значения, необходимые для определения нужного свойства, в) если переменная a получает новое значение при выполнении $s'_n \in S'$ (т.е. $a \in W(s'_n)$, $s'_n \in S'$), то S' содержит все операторы, вычисляющие переменные из $R(s'_n)$.

Можно заметить, что определение большинства описанных свойств программ связано с определением значений собственного подмножества всех переменных программ. Выделение такого подмножества не представляет труда для каждого конкретного случая, например, обращение к несуществующим элементам массивов связано со значениями тех переменных, которые используются в индексных выражениях мас-

сивов, выявление недостижимых участков - со значениями переменных, использующихся в условных операторах ветвления; завершаемость циклов - с переменными, управляющими циклами.

Таким образом, определение операторов исследуемой программы, которые должны интерпретироваться при выявлении того или иного свойства, связано с выделением переменных, значения которых влияют на это свойство. После того, как такие переменные выделены, для определения свойства достаточно интерпретировать лишь те операторы, которые изменяют значения этих переменных. Такое,трудно формулируемое в общем случае, но тем не менее интуитивно понятное для каждого конкретного свойства, определение интерпретируемых операторов имеет один недостаток.На примере оператора присваивания этот недостаток имеет следующий вид. Пусть выделен оператор, присваивающий значение переменной,которая используется для определения заданного свойства. Для исполнения этого оператора надо знать значения всех переменных в правой части. Следователь - но, такие переменные также являются влияющими на выявляемое свойство (не прямо, но косвенно) и должны интерпретироваться все операторы, присваивающие значения таким переменным.Но и эти переменные тоже, в свою очередь, могут зависеть от других переменных. Таким образом, решение простой задачи может вылиться в длинную цепочку зависимых задач. Сложность решения таких задач усугубляется возможной сложностью графа управления исследуемой программы.

Поэтому рассматриваемый ниже алгоритм предлагается для решения задачи о минимальном достаточном количестве интерпретируемых операторов программы, требуемом для выявления некоторых, описан - ных ниже, ее динамических свойств; трудоемкость алгоритма существенно меньше трудоемкости просмотра всех путей управляющего гра - фа.

Основная идея алгоритма состоит в построении набора интерпретируемых операторов на основе разбиения множества всех переменных на два класса - влияющих (прямо или косвенно) на выявляемое свойство и не влияющих на него. Разбиение множества всех переменных происходит во время итерационного пополнения класса переменных, явно влияющих на выявляемое свойство, переменными, необходимыми для вычисления их значений.

Алгоритм показан на простом примере фортрано-подобного языка, а затем описан в более общем случае.

Рассмотрим фрагмент программы, строящей по заданным ребрам графа связные списки смежности [2], т.е. для каждой вершины графа свой список вершин, в которые ведут выходящие из нее ребра.

```
      DO 99 J=1,100                    (1)
      ADJ(J)=0                         (2)
99    NEXT(J)=0                        (3)
      READ(P)                          (4)
      I=P+1                            (5)
100   READ(M,N)                        (6)
      IF(M.LT.0)GOTO 200               (7)
      ADJ(I)=N                         (8)
      NEXT(I)=NEXT(M)                  (9)
      NEXT(M)=I                       (10)
      I=I+1                           (11)
      ADJ(I)=M                        (12)
      NEXT(I)=NEXT(N)                 (13)
      NEXT(N)=I                       (14)
      I=I+1                           (15)
      GOTO 100                        (16)
      . . .
200   . . .
```

Этот фрагмент вводит значение P - число вершин в графе, а затем на основании вводимых пар чисел-ребер графа (ребро графа задается номерами вершин, соединяемых этим ребром) строит списки смежности. Фрагмент заканчивает свою работу, если номер вершины, инцидентной очередному введенному ребру, отрицателен.

Для определения порядка выполнения операторов программы необходимо знать значения, присваиваемые переменной M, так как предикат в единственном условном операторе фрагмента зависит от этой переменной. Единственный оператор, присваивающий значение переменной M, - это оператор (6). Таким образом, для определения последовательности выполняемых операторов фрагмента достаточно интерпретировать операторы (6) и (7), а вместо выполнения остальных операторов - просто констатировать факт их исполнения. Для определения того, что только операторы (6) и (7) влияют на ход вычислительного процесса, необходимо два просмотра программы: на первом просмотре определяется, что переменная M влияет на порядок выполнения операторов, а на втором - что для определения значения M

должен быть выполнен оператор (6). Конечно, в данном случае можно обойтись одним просмотром тела программы, но тогда для каждой переменной надо хранить список всех операторов, в которых она может изменить значение.

Рассмотрим теперь другое свойство того же фрагмента, для определения которого применим описываемый алгоритм, а именно определение последовательности обращений к переменным программы, т.е. историю использования переменных (при этом обращением к переменной с индексами считается обращение к соответствующей компоненте этой переменной, а не ко всей переменной). Такая история использования переменных может быть полезна при определении информационных зависимостей между операторами, например, в задачах оптимизации программ или при анализе возможности их параллельного исполнения. Ясно, что для определения всех обращений к данным необходимо вычислять те переменные, которые влияют на последовательность выполнения операторов программы или встречаются в индексных выражениях. Но, как это и оказалось в данном случае, набор только таких переменных {J,M,N,I} необходим, но недостаточен для определения свойства. Выяснилось это во время второго последовательного просмотра тела программы, когда оказалось,что для вычисления значения переменной I в операторе (5) необходимо использовать значение переменной P. Набор переменных, значения которых должны вычисляться для определения последовательности обращений ко всем переменным программы, есть {J,M,N,I,P} и соответственно вместо всего фрагмента достаточно выполнять операторы (I),(4),(5),(6), (7), (II),(I5) и (I6), а вместо исполнения остальных операторов фиксировать факты обращений к переменным, которые произошли бы,если бы эти операторы исполнялись.

Доведем анализ этого свойства до конца, т.е. опишем все действия над программой для определения последовательности обращений к переменным программы. При первом просмотре текста программы замечаем, что переменные J,M,N,I должны вычисляться, при втором просмотре к ним добавляется переменная P, а при третьем отмечаются те операторы, которые присваивают значения выделенным переменным. Затем исходная программа формально преобразуется к следующему виду [4,I0]:

```
      DO  99  J=1,100
      CALL ACT(ADJ,J,"W")
      CALL ACT(NEXT,J,"W")
99    CONTINUE
      CALL ACT(P,"W")
      READ(P)
      CALL ACT(P,"R",I,"W")
      I=P+1
100   READ(M,N)
      CALL ACT(M,"W",N,"W",M,"R")
      IF(M.LT.0) GOTO 200
      CALL ACT(N,"R",I,"R",ADJ,I,"W")
      CALL ACT(M,"R",NEXT,M,"R",NEXT,I,"W")
      CALL ACT(I,"R",M,"R",NEXT,M,"W")
      CALL ACT(I,"R",I,"W")
      I=I+1
      CALL ACT(M,"R",I,"R",ADJ,I,"W")
      CALL ACT(N,"R",NEXT,N,"R",I,"R",NEXT,I,"W")
      CALL ACT(I,"R",N,"R",NEXT,N,"W")
      CALL ACT(I,"R",I,"W")
      I=I+1
      GOTO  100
      . . .
200   . . .
```

Процесс такого преобразования может быть легко запрограммирован. Вызываемая, заранее написанная подпрограмма ACT должна быть добавлена к преобразованной программе для дальнейшей совместной трансляции. Во время своей работы она фиксирует факты обращения к соответствующим компонентам переменных, указанных в списке параметров. Параметры этой подпрограммы (их может быть произвольное число) разбиты на группы. Группы отделяются друг от друга символами "R" или "W"; первый означает, что надо зафиксировать факт чтения переменной, имя которой стоит первым в этой группе, ли-

бо ее компоненте, указанной значениями второго и далее параметров (если они есть) в группе, второй - что должен быть зафиксирован факт обращения к переменной по записи.

Затем полученная преобразованная программа вместе с добав - ленной подпрограммой АСТ транслируется и выполняется обычным образом. Результатом ее работы будет зафиксированная вызовами подпрограммы АСТ последовательность (история) обращений к переменным.

Более строго этот алгоритм может быть представлен в виде следующих шагов. Пусть LIST1 и LIST2 - два динамических списка имен переменных, определяемых алгоритмом.

Шаг I. LIST 1:={$\emptyset$} - пустой список;

LIST 2:={ все переменные, явно влияющие на определяемое свойство }.

Шаг 2. while LIST 1$\neq$LIST2 do LIST1:=LIST 2; для каждого оператора исследуемой программы s_n, для каждой переменной a_m

$$\text{if}((a_m \in \text{LIST1}) \;\&\; (a_m \in W(s_n)) \text{ then LIST2}:=\{\text{LIST2} \cup R(s_n)\}$$

(этот шаг повторяется, пока очередное его исполнение добавляет к списку что-то новое).

Шаг 3. Помечаем оператор s_n ($n=\overline{1,N}$) как интерпретируемый (принадлежащий S'), если найдется хоть одна такая переменная a_m, что $(a_m \in W(s_n)) \;\&\; (a_m \in \text{LIST2})$, либо если сам s_n - оператор безусловного перехода.

Очевидно, что цикл в шаге 2 завершится за ограниченное число повторений, так как при каждом повторении список LIST2 может быть только увеличен. В худшем случае цикл завершится за M повторений, где M - общее число имен переменных. Общая трудоемкость алгоритма $T \leq (2+M)Nt$, где t - средняя трудоемкость просмотра одного оператора исследуемой программы.

Легко показать, что приведенный алгоритм решает поставленную задачу. Действительно, выполнение свойства "а" постановки задачи гарантируется тем, что все операторы, изменяющие последовательность выполнения операторов, попадут в список интерпретируемых операторов S'. Выполнение свойства "б" обеспечивается заданием начального множества интерпретируемых переменных. Выполнение свойства "в" обеспечивается конечностью работы алгоритма.

Приведенный алгоритм строит достаточное множество операторов, требуемое для имитационного [4] выполнения программы. Это

множество может обладать избыточностью по сравнению с необходимым множеством. Однако эта избыточность сокращается в том случае, когда исследуемые программы являются структурированными либо в них следуют таким ограничениям: избегают переходов назад по тексту программы и каждую переменную программы используют в тексте только в одном смысле (например, если переменная используется как переменная цикла, то она не используется в качестве обычной переменной вне тела цикла).

Описанный алгоритм был использован при реализации имитационного выполнения программ для выявления параллелизма в циклах [4]. При этом для выявления информационных связей, препятствующих параллельному выполнению отдельных витков циклов, использовалась история использования программой данных, способ получения которой описан в статье.

В качестве еще одного приложения алгоритма опишем использование его для облегчения символического выполнения программ.

Пример использования метода. Символическим выполнением программ [8,9] обычно называют выполнение не с традиционной семантикой, при которой значения выражений вычисляются по правилам арифметики (и, как только выражение вычислено, последовательность действий по вычислению выражения никак не отражается в результате), а с "алгебраической" семантикой, при которой значением выражения является алгебраическая формула, по которой это выражение вычислено. При реализации символического выполнения выделяются две задачи, их можно назвать прямой и обратной. В общем случае постановка этих задач такова: прямая задача - по заданным условиям на значения входных переменных определить ход вычислительного процесса, т.е. последовательность выполнения операторов и символьные значения результирующих переменных программы; обратная задача - по заданной последовательности выполненных операторов программы определить реализуемость данного пути и условия на значения входных переменных, при которых эта последовательность реализуема. Обычной целью символического выполнения является проверка реализуемости путей управляющего графа и построение системы тестов, удовлетворяющей некоторому критерию тестирования (покрытие всех путей графа управления программы, покрытие всех операторов, выполнение каждого условного оператора хотя бы один раз или другие). Среди трудностей на пути реализации символического выполнения отмечают [8]: а) неоднозначность, возникаю-

щую при использовании индексных выражений у элементов массивов (например, условное выражение A[I+1]=A[K] ≡ true, если K=2 и индекс не выходит за пределы массива, но для определения этого надо знать значение K, которым при символическом выполнении может быть нетривиальное алгебраическое выражение); б) быстрый рост размеров формул с увеличением количества витков циклов. Поскольку решение обратной задачи связано с решением систем неравенств или автоматическим доказательством теорем, а решение прямой задачи - с проверкой совместимости неравенств, то трудоемкость решения задач символического выполнения существенно зависит от количества неравенств и размера формул в неравенствах. Поскольку количество соотношений между символическими переменными растет линейно, в зависимости от числа витков внутренних циклов, а размер этих соотношений увеличивается также линейно, то можно считать, что рост соотношений квадратичный.

С помощью приведенного алгоритма все переменные исследуемой программы можно разбить на два непересекающихся класса: интерпретируемые переменные, т.е. переменные, организующие вычислительный процесс и влияющие на его ход, и неинтерпретируемые, т.е. пассивно подвергающиеся изменениям в ходе вычислительного процесса. После этого естественно организовать смешанное выполнение [7] (здесь задержанными являются неинтерпретируемые переменные), при котором над переменными первого класса производятся обычные вычисления арифметических и логических выражений, а над переменными второго класса определяются символические выражения. Такой подход позволяет существенно сократить сложность организации символического выполнения.

Продемонстрируем достоинства совместного применения описанного алгоритма и символического выполнения на примере решения прямой задачи для рассмотренного фрагмента фортран-программы. Пусть фрагмент в процессе символического выполнения вводит символические значения $\alpha,\beta,\gamma,\delta,\epsilon,\zeta,\eta$ ... .

Условия на входные значения: $\alpha = 10$, $\beta > 0$, $\delta > 0$, $\zeta < 0$. Тогда полные протоколы для чисто символического выполнения (А) и смешанного выполнения с интерпретацией переменных J, I, P, M, N (Б) будут иметь вид:

А	Б
J:=1	ADJ(1):=0
ADJ(1):=0	

```
NEXT(1):=0                                  NEXT(1):=0
J:=1+1                                      ADJ(2):=0
      1+1 < 100                             NEXT(2):=0
ADJ(1+1):=0
NEXT(1+1):=0                                . . .
J:=1+1+1
     1+1+1 < 100
. . .
. . .                                       . . .
J:=1+1+1+...+1                              ADJ(100):=0
     ¬(1+1+1+...+1 < 100)                   NEXT(100):=0
P:=α
I:=α+1
M:=β
N:=γ
    ¬(β < 0)
ADJ(α+1):=γ                                 ADJ(11):=γ
NEXT(α+1):=NEXT(β)                          NEXT(11):=NEXT(β)
NEXT(β):=α+1                                NEXT(β):=11
I:=α+1+1
ADJ(α+1+1):=β                               ADJ(12):=β
NEXT(α+1+1):=NEXT(γ)                        NEXT(12):=NEXT(γ)
NEXT(γ):=α+1+1                              NEXT(γ):=12
I:=α+1+1+1
M:=δ
N:=ε
   ¬(δ < 0)
ADJ(α+1+1+1):=ε                             ADJ(13):=ε
NEXT(α+1+1+1):=NEXT(δ)                      NEXT(13):=NEXT(δ)
NEXT(δ):=α+1+1+1                            NEXT(δ):=13
I:=α+1+1+1+1
ADJ(α+1+1+1+1):=δ                           ADJ(14):=δ
NEXT(α+1+1+1+1):=NEXT(ε)                    NEXT(14):=NEXT(ε)
NEXT(ε):=α+1+1+1+1+1                        NEXT(ε):=14
I:=α+1+1+1+1
M:=ζ
N:=η
ζ<0
```

всего 428 записей
из них 103 неравенства

всего 212 записей

Легко заметить, что протокол Б в полтора раза короче, чем А и не содержит громоздких символьных значений типа $\alpha+1+1+1$ и $ADJ(\alpha+1+1+1)$. С ростом числа витков внутреннего цикла n (где n - число ребер графа) протокол Б будет увеличиваться как $6n$, в то время, как протокол А как $(6+5)n = 11n$. Размер символьных выражений протокола Б не зависит от n, а в протоколе А возрастает линейно.

З а к л ю ч е н и е. Выполнение программ на ЭВМ может преследовать различные цели. С одной стороны, это получение результатов счета, а с другой - исследование свойств программ при различных условиях их выполнения. Последний случай имеет место не только при тестировании программ в процессе отладки, но и при их модификации в процессе эксплуатации. При этом часто нет необходимости исполнять все тело программы, а достаточно исполнения лишь некоторых, ключевых в смысле поведения программы, операторов. Поэтому алгоритм определения достаточных наборов таких операторов, описанный в статье, может быть весьма полезным, он расширяет возможности и упрощает символическое выполнение программ. Он также может найти применение при построении профилей исполнения программ и для динамического слежения за априорной точностью вычислений.

Замена итерационного характера алгоритма на работу с динамическими списками взаймовлияний переменных не улучшает ни его трудоемкости, ни наглядности. Реализация алгоритма, как показал опыт его использования [4], не представляет сложности и может быть осуществлена либо как самостоятельная сервисная часть системы программирования, либо как дополнительная возможность отладочного компилятора.

Л и т е р а т у р а

1. МАЙЕРС Г. Надежность программного обеспечения. -М.: Мир, 1980. - 360 с.

2. ГУДМАН С., ХИДЕТНИЕМИ С. Введение в разработку и анализ алгоритмов. -М.: Мир, 1981. - 368 с.

3. МИРЕНКОВ Н.Н. Параллельные алгоритмы и корректность параллельных программ. - Новосибирск, 1983. - 15 с. (Препринт/ИМ СО АН СССР: ОВС-17).

4. МИРЕНКОВ Н.Н., СИМОНОВ С.А. Выявление параллелизма в циклах методом имитации их выполнения. - Кибернетика, 1981, №3, с. 28-33.

5. КАСЬЯНОВ В.Н. Методы анализа программ. -Новосибирск, 1982. - 92 с. (Нов.Гос.ун-т.)

6. КАСЬЯНОВ В.Н. Анализ структур программ. - Кибернетика, 1980, №1, с.48-61.

7. ЕРШОВ А.П. О сущности трансляции.-Программирование, 1977, №5, с.21-39.

8. БИЧЕВСКИЙ Я.Я. Автоматическое построение систем примеров.-Программирование, 1977, №3, с.60-69.

9. HUANG J.C. An approach to program testing. - ACM Computing Surveys,1975,v.9, N 3,p.113-128.

10. HUANG J.C. Program instrumentation and software testing. -Computer,1978,v.11,april,p.25-32.

11. FAIRLEY R.E. Tutorial: Static analysis and dynamic testing of computer software. - Ibid,p.14-23.

Поступила в ред.-изд.отд.
17 мая 1985 года