ИСПОЛЬЗОВАНИЕ ДЕРЕВЬЕВ ВЫБОРА ДЛЯ ОПИСАНИЯ СОСТОЯНИЙ В РАСПАРАЛЛЕЛИВАЮЩЕМ КОМПИЛЯТОРЕ

Арк.В. Климов

При автоматическом распараллеливании программ большое значение имеет выявление графа потоковых зависимостей [1] между операциями чтения и записи (для простой переменной или элемента массива): при каждом чтении значения важно знать, какой операцией записи это значение было записано. Для описания зависимостей нужны формальные средства, одним из которых являются деревья выбора [2]. В работе [3] предложено использовать деревья выбора для описания семантического эффекта фрагментов программ. В этой статье будет показано, как от описаний семантического эффекта фрагментов можно перейти к описаниям состояний памяти в точках программы, и далее – к описаниям потоковых зависимостей.

Под *семантическим эффектом* мы понимаем воздействие программного фрагмента на состояние массивов и переменных. Описать воздействие фрагмента P на массив A – значит задать правило, которое для каждого элемента массива A[i,j,…] говорит, был ли он изменен в результате исполнения данного фрагмента, и если да, то каким именно оператором присваивания. Причем надо не просто указать оператор в тексте программы, но и дать значения параметров всех циклов, объемлющих этот оператор, при которых и была сделана последняя запись в данный элемент массива.

Конечно, такое описание далеко не всегда возможно и, тем более, не всегда вычислимо. Мы ограничимся так называемыми *линейными* программами, для которых оно и возможно, и вычислимо. В них используются: операторы присваивания, правильно вложенные условные операторы if–then–else с линейными условиями, правильно вложенные циклы с линейными границами и единичным шагом, причем все индексные выражения линейные. *Линейные выражения* – это неоднородные линейные комбинации параметров циклов и внешних параметров с целыми коэффициентами.

Для линейного программного фрагмента его семантический эффект может быть выражен деревом выбора. В нем есть вершины вида (C–>T_1 : T_2), где C – условие вида e=0 или e>0 с линейными e (как «синтаксический сахар» допускаются и другие формы сравнения), а T_1 и T_2 – поддеревья для случаев True и False. Еще есть вершины «деления» вида ($e = m\boldsymbol{q}+\boldsymbol{r}$ –> T), где m – целое, а $\boldsymbol{q}$ и $\boldsymbol{r}$ – новые переменные. В листьях стоит либо терм None, либо терм вида $S\{e_1,e_2,\ldots\}$, где S – программное имя оператора присваивания.

Если всем переменным дерева дать числовые значения, все условия вычисляются до True или False, в вершинах «деления» значение e делится на m с получением частного $\boldsymbol{q}$ и остатка $\boldsymbol{r}$, и тогда все дерево вычисляется до терма с целочисленными аргументами. Терм None означает, что элемент A[i,j,…] не изменился. Терм $S\{n_1,n_2,\ldots\}$ говорит, что источник есть оператор S, исполненный в контексте объемлющих его циклов со значениями параметров циклов $n_1,n_2,\ldots$. Число аргументов должно соответствовать числу циклов, объемлющих оператор S (всех, а не только в данном фрагменте). В таблице 1 приведены примеры фрагментов линейных программ и их деревьев эффекта. Альтернативу :None для краткости опускаем.

Таблица 1. Примеры деревьев эффекта

Параметры	**Фрагмент кода**	**Действие на**	**Дерево эффекта**
i	`M: A(i-1)=Z`	A(p)	(p=i-1-> M{})

i,j	`if i<j then`
	`M1: A(i)= exp1`
	`else`
	`M2: A(j)= exp2`
		A(k)	(i<j –>
			(k=i –> M1{})
			: (k=j –> M2{})
			)

N	`M1: A(1)=0;`
	`for i = 2 to N do`
	`M2: A(i)=A(i-1)*x`
		A(k)	(k=1 –> M1{}
			: (2≤k –> (k≤N–> M2{k}
			)))

n	`for k = 1 to n do`
	`for j = 1 to k-1 do`
	`for i = k to n do`
	`M: A(i,j) = ...`
		A(p,q)	(p≤n –> (1≤q –> (q<p –>
			M{p,q,p}
			)))

Для построения дерева эффекта по коду фрагмента имеется ряд операций над деревьями. Операция SEQ(T_1,T_2) выдает дерево для последовательного исполнения фрагментов P_1 и P_2, имеющих деревья эффекта T_1 и T_2. Операция CONV(v, e_0, e_1, T), выполняет свертку (определяемую через SEQ) дерева T по переменной v, пробегающей диапазон от e_0 до e_1. Ее результат есть дерево для цикла **for** $v = e_0$ **to** e_1 **do** P, если T есть дерево

эффекта оператора *P*. Оно уже не зависит от переменной *v*, если только она не входит в выражения для границ e_0, e_1 (иначе говоря, переменная *v* считается здесь связанной внутри *T*, но не внутри e_0 и e_1). Посредством этих и некоторых других операций осуществляется построение дерева эффекта *T* по фрагменту *S* путем рекурсивного подъема по структуре фрагмента: *T*=Eff(*S*) [3].

Реализация функции CONV опирается на алгоритм решения задачи параметрического целочисленного программирования [2,4] относительно одиночной переменной. Все функции, вырабатывающие дерево выбора, перед выдачей результата пытаются его масимально упростить. Для этого вдоль каждого пути в дереве выполняется Омега-тест [5], проверяющий совместность совокупности целочисленных условий, после чего лишние ветви отсекаются.

Теперь займемся задачей получения *описания состояния* массива в заданной точке программы. Для представления состояний используются те же деревья выбора, но только терм None теперь означает, что элемент массива содержит свое исходное значение перед входом в программу. Параметрами дерева состояния относительно массива с элементами A[i,j,…] являются индексы i,j,…, параметры всех циклов, объемлющих данную точку, и другие фиксированные параметры программы. При фиксации числовых значений всех параметров дерево состояния превращается либо в терм None, либо в терм вида $S\{n_1,n_2,\ldots\}$, который задает точные координаты оператора присваивания, определившего содержимое элемента A[i,j,…] на момент прохождения управления через данную точку.

Опишем рекуррентный процесс построения описаний состояний для всех точек программы, начинающийся с входной точки и переходящий последовательно от точки к точке:

- Для входной точки состояние выражается деревом None.
- При переходе через оператор S состояние «после» определяется через состояние «до» по формуле:

$$T_{\text{после}} = \text{SEQ}\ (\ T_{\text{до}},\ \text{Eff}\ (\ S\ )\ ).$$

- При переходе через условие к началу любой ветви условного оператора состояние не изменяется.
- При входе в цикл **for** *v = e0* **to** *e1* **do** *P* состояние перед телом *P* определяется формулой:

$$T_{\text{перед}P} = \text{SEQ}\ (T_{\text{до}},\ \text{CONV}\ (v,\ e_0,\ v\text{–}1,\ \text{Eff}\ (\ P\ )\ )\ )$$

Здесь свертка выполняется по переменной *v*, которая также входит в верхнюю границу для свертки. В результирующем дереве *v* выражает текущее значение параметра данного цикла для момента, в котором определяется состояние. Смысл этой формулы в том, что при проходе через заголовок цикла из состояния перед циклом $T_{\text{до}}$ надо «пройти» все витки цикла с $v<v_{\text{текущ}}$. В записи результата функции CONV в качестве $v_{\text{текущ}}$ используется опять *v*.

Эти 4 правила позволяют «вычислить» дерево состояния в любой внутренней точке линейной программы.

Рассмотрим в качестве примера следующий фрагмент, который заимствован из [1, стр.385]:

```
S1:  for i = 1 to n do
S2:    for j = 1 to n do
S3:      A(i+j) = A(2*n+1-i-j)
```

Вложенные друг в друга операторы S1, S2, S3 имеют семантический эффект (относительно A[k]):

```
T3 = Eff(S3) = (i+j=k –> S3{i,j})
T2 = Eff(S2) = CONV(j,1,n,T3) = (k≥i+1 –> (k≤i+n –> S3{i,k-i} ) )
T1 = Eff(S1) = CONV(i,1,n,T2) =
        (n≥1 –> (k≥n+2 –> (k≤2*n –>  S3{n,k-n}
                                :      (k≥2 –> S3{k-1,1} ) ) ) )
```

Ниже приводятся деревья выбора, представляющие состояния в точках перед операторами S1, S2 и S3. Они были вычислены компьютером согласно вышеприведенным правилам и лишь немного подредактированы для улучшения читабельности.

```
TS1 = None
TS2 = SEQ(TS1, CONV(i,1,i-1,T2) =
        (i≥2 –> (k≥i+1 –> (k≤n+i-1 –> S3{i-1,k+1-i} )
                :          (k≥2 –> (n≥1 –> S3{k-1,1} ) ) ) ) )
TS3 = SEQ(TS2, CONV(j,1,j-1,T3) =
        (k≥i+1 –> (k≤i+j-1 –> S3{i,-i+k}
                  :          (i≥2 –> (k≤n+i-1 –> S3{i-1, k+1-i} ) ) )
         :          (k≥2 –> (n≥1 –> S3{k-1,1} ) ) )
```

Имея в каждой точке программы дерево состояния, легко получить описание источника для произвольного оператора *R* чтения элемента массива A[e_1,e_2,…]. Пусть дерево *T* описывает состояние массива

A[i,j,…] перед оператором *R*. Сделаем подстановку: $T_R=T//i->e_1,j->e_2,\ldots$. Новое дерево T_R определяет (в зависимости от параметров циклов, объемлющих оператор *R*, и других параметров программы) полные координаты оператора присваивания, записавшего читаемое значение. Если проделать эту операцию для нашего примера, то получим дерево, которое описывает тот же граф (те же линейные функции, на тех же многогранниках, только чуть по-другому заданных), что построен в [1].

Таким образом, информация о состояниях во всех точках дает основу для вычисления всех потоковых зависимостей. Верно и обратное: если есть алгоритм построения потоковых зависимостей, то легко вычислить состояние массива A[i,j,…] в определенной точке. Для этого достаточно в заданную точку программы формально поместить оператор чтения элемента A[i,j,…] и провести построение зависимостей.

Задача построения зависимостей рассматривалась много раз [1,2,4,5]. При этом обычно для каждой пары чтение-запись решается своя задача целочисленного программирования, после чего результаты объединяются по всем операциям записи. В работе [2] для представления решений также используются деревья выбора. Предлагаемый подход проще концептуально и, вероятно, эффективнее, поскольку, в частности, он избегает повторной работы для нескольких однотипных операторов чтения.

Работа выполнена в рамках НИР «ВЕГА-Пр-Ст-2011» (Рег. № 01.2.00951607) по программе фундаментальных исследований №3 Программы Президиума РАН "Фундаментальные проблемы системного программирования".

ЛИТЕРАТУРА:

1. В.В. Воеводин, Вл.В. Воеводин. Параллельные вычисления. СПб: БХВ-Петербург, 2004. – 599 с.
2. P. Feautrier. Dataflow analysis of array and scalar references. // *International Journal of Parallel Programming*, 20(1), February 1991.
3. А.В. Климов. Деревья выбора и их использование для описания семантического эффекта программы. // *Подано в сборник трудов конференции МСО-2009*, Москва, 2009.
4. V. Maslov. Lazy Array Data-Flow Dependence Analysis. // *In: Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1994, pp. 311-325.
5. W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. // *Comm.of the ACM*, August 1992.