

Московский государственный университет имени М. В. Ломоносова



Факультет Вычислительной Математики и Кибернетики
Кафедра Системного программирования

Курсовая работа
"Разработка и реализация алгоритма подстановки
выражений в последовательных программах с помощью
графа потока управления"

Выполнил:
студент 3 курса 327 группы
Кочармин Михаил Дмитриевич

Научный руководитель:
профессор Крюков Виктор Алексее-
вич

Научный консультант:
Колганов Александр Сергеевич

Москва, 2023

Содержание

1	Введение	2
2	Постановка задач	2
2.1	Подстановка выражений	2
2.2	Анализ живых переменных	2
2.3	Анализ приватных переменных	3
2.4	Входные данные	4
3	Решение	4
3.1	Подстановка выражений	4
3.2	Анализ живых переменных	5
3.3	Анализ приватных переменных	7
4	Возникшие проблемы	7
5	Результаты	8

1 Введение

Данная работа проводилась в рамках разработки программного обеспечения SAPFOR – системы для автоматизации распараллеливания Фортран-программ. Целью данной работы является добавление в SAPFOR реализаций алгоритмов подстановки выражений, анализа живых переменных и приватных переменных.

Система SAPFOR представляет собой набор проходов – отдельных стадий выполнения, которые последовательно запускаются с целью сбора информации и/или преобразования текста входной программы. Соответственно, в систему были добавлены три прохода с названиями SUBST_ANALYSIS_IR, LIVE_ANALYSIS_IR и PRIVATE_ANALYSIS_IR.

2 Постановка задач

2.1 Подстановка выражений

Подстановка выражений заключается в замене скалярных переменных в выражениях на другие, эквивалентные им, скаляры или выражения. При этом, стоит цель максимизировать количество замен в каждой цепочке подстановок.

Пример:

```
...
DO I = 1, N
  DO J = I, M
    P = J + I
    Q = J - I

    F = P / Q

    A(2*P, 3*Q) = F
  ENDDO
ENDDO
...
```

Листинг 1: До подстановки

```
...
DO I = 1, N
  DO J = I, M
    P = J + I
    Q = J - I

    F = (J + I) / (J - I)

    A(2*(J+I), 3*(J-I)) =
      (J+I)/(J-I)
  ENDDO
ENDDO
...
```

Листинг 2: После подстановки

Одной из целей данного прохода является удаление зависимостей по данным, которые могут мешать распараллеливанию циклов.

2.2 Анализ живых переменных

Класс живых переменных определяется следующим образом [2, с. 643]:

Определение 2.2.1 *Предположим, что трехадресная команда i присваивает значение переменной x . Если команда j имеет x в качестве операнда и управление может перейти от i к j по пути, на котором нет промежуточных присваиваний x , то команда j **использует** значение x , вычисленное в команде i .*

Определение 2.2.2 *При этом, переменная x называется **живой**, или **активной** (*live*) в команде i .*

Аналогично можно определить активность переменной не для конкретной инструкции, а для любой точки программы.

Задача прохода анализа живых переменных заключается в определении для каждого базового блока B множеств $IN_{live}[B]$ и $OUT_{live}[B]$ – множеств живых переменных соответственно перед первой и после последней инструкции рассматриваемого блока.

Для практической пользы, вместе с каждой живой переменной нужно сохранить информацию о том, в каких инструкциях используется значение переменной из данной точки. При этом, в целях экономии ресурсов, хранить номера только тех инструкций, в которых значение будет использоваться в первый раз.

Более формально, если I – множество номеров всех инструкций, в которых используется значение x из точки t перед первой (или после последней) инструкции базового блока, то нужно хранить только подмножество $I_0 \subseteq I$:

$$I_0 = \{ i \mid \forall i \in I \text{ существует путь от } t \text{ до } i, \text{ не содержащий } j \ \forall j \in I \}.$$

2.3 Анализ частных переменных

Определение 2.3.1 *Приватной переменной x цикла s , называется скалярная переменная, значение которой можно локализовать в рамках одного витка цикла [4].*

Это значит, что значение переменной x после любого витка цикла не будет использоваться ни одним другим витком, оно не будет использоваться после цикла и не используется значение x , которое она имела до цикла. При этом практический интерес имеют только те приватные переменные, которые имеют в цикле хотя бы одно переопределение.

Определение 2.3.2 *Также выделяется класс **lastprivate-переменных** (или **приватных по выходу**) – это приватные переменные, для которых нарушается вышеописанное требование для последнего витка, то есть значение такой переменной используется после цикла.*

Задача прохода анализа частных переменных – для каждого цикла построить множества частных и частных по выходу переменных.

2.4 Входные данные

В качестве входных данных для прохода подстановки выражений предоставляются:

- внутреннее представление программы в виде графов базовых блоков функций (графы потока управления)
- результаты анализа достигающих определений для этого графа потока управления
- внутреннее представление программы в виде дерева операторов и выражений (часть библиотеки *Sage++* [3])

Результаты анализа достигающих определений поставляются в виде словарей $IN_{RD}[B]$ и $OUT_{RD}[B]$ для каждого базового блока B . Ключами данного словаря являются переменные, а значениями – множества номеров инструкций, которые порождают определения, достижимые в точке соответственно до и после базового блока B .

В проходе анализа живых переменных используется только граф потока управления.

В проходе анализа приватных переменных – граф потока управления, граф циклов программы, Sage++-дерево [3] и результаты анализа живых переменных.

3 Решение

3.1 Подстановка выражений

Алгоритм подстановки выражений работает отдельно с каждым базовым блоком. Для каждой инструкции $i = 1..m$ вычисляется словарь $available[i]$. Ключами в нём являются переменные, а значениями – выражения, которыми можно заменить данные переменные в точке сразу после этой инструкции.

Пример:

[1] P = I - J
[2] Q = I + J
[3] F = P / Q
[4] REF P
[5] REF Q
[6] STORE A 2 F

$$\begin{aligned}
 available[1] &= \{P \rightarrow I - J\} \\
 available[2] &= \{ \\
 &\quad P \rightarrow I - J, \\
 &\quad Q \rightarrow I + J \\
 &\} \\
 available[3] &= \{ \\
 &\quad P \rightarrow I - J, \\
 &\quad Q \rightarrow I + J, \\
 &\quad F \rightarrow P / Q \\
 &\} \\
 available[4, 5, 6] &= available[3]
 \end{aligned}$$

Сначала, перед итерацией по инструкциям блока, по множеству IN_{RD} строится базовый словарь $available[0]$. Потом для каждой операции с номером $i = 1..m$ по $available[i - 1]$ строится $available[i]$ следующим образом:

1. $available[i] = available[i - 1]$
2. Пусть инструкция i создаёт новые определения $a_1 \rightarrow e_1, \dots, a_k \rightarrow e_k$, и определяет переменные b_1, \dots, b_r выражениями, которые невозможно скопировать в другое место (например, если выражение содержит вызов функции)
3. $\forall j$ если в $available[i]$ есть запись $a_j \rightarrow expr_j$ (или $b_j \rightarrow expr_j$), то во всех хранимых в $available[i]$ выражениях делается замена b_j (или a_j) на $expr_j$
4. $\forall j$ из $available[i]$ удалятся все записи с ключём a_j (или b_j) или те, которые содержат a_j (или b_j) в выражении-значении
5. $\forall j$ в $available[i]$ добавляется запись $a_i \rightarrow e_i$

После построения очередного множества $available[i - 1]$ можно выполнить подстановку переменных в подвыражении, соответствующее инструкции с номером i . Необходимо заменить в этом подвыражении все переменные, которые есть в $available[i - 1]$, на соответствующие выражения. Так как после выполненных замен в этом подвыражении могут появиться другие неподставленные переменные, подстановку стоит осуществлять до тех пор, пока доступные замены не закончатся.

3.2 Анализ живых переменных

Сначала рассмотрим задачу для графа отдельной функции, поскольку именно в таком виде реализовано внутреннее представление.

Введём для базового блока два множества def_B и use_B так же, как и в [2, с. 733]:

- def_B — множество переменных, определенных (т.е. получающих значения) в блоке B до любых их использований в этом блоке;
- use_B — множество переменных, значения которых могут использоваться в блоке B до любых определений этих переменных.

Построив для каждого базового блока B множества def_B и use_B можно записать соотношения для $IN_{live}[B]$ и $OUT_{live}[B]$ [2, с. 734]:

$$IN_{live}[B] = use_B \cup (OUT_{live}[B] - def_B)$$

$$OUT_{live}[B] = \bigcup_{S \text{ — преемник } B} IN_{live}[S]$$

С граничным условием

$$OUT_{live}[\text{Выход}] = \emptyset$$

Существует итеративный алгоритм, который просматривает все базовые блоки и обновляет множества $IN_{live}[B]$, $OUT_{live}[B]$ до тех пор, пока хотя бы в один базовый блок были внесены изменения [2, с. 735].

Было принято решение оптимизировать этот алгоритм:

Шаг 0. Положить

$$\forall B : IN_{live}[B], OUT_{live}[B] = \emptyset.$$

Шаг 1. Игнорируя в графе базовых блоков обратные дуги, привести его к напращенному дереву. После этого перевести это дерево в массив базовых блоков A , для которого выполнено соотношение:

$$\forall i, j, i < j, \text{ в построенном дереве нет пути из } A[i] \text{ в } A[j].$$

Шаг 2. Итерация по массиву $A[i]$ с его начала до конца (то есть в направлении от листьев к корню). На каждой итерации сначала обновляются множества $IN_{live}[B]$ и $OUT_{live}[B]$. Потом, $\forall j$: до удаления существовала обратная дуга $A[j] \rightarrow A[i]$:

1. Заново вычислить $IN_{live}[A[j]]$, $OUT_{live}[A[j]]$ уже рассматривая обратные дуги (то есть информация из блока $A[i]$ должна распространиться в блок $A[j]$).
2. Если были внесены изменения в $IN_{live}[A[j]]$, то продолжить итерацию с элемента $A[j]$, то есть передвинуть указатель назад: $i := j$.
3. Иначе продолжить просмотр обратных дуг.
4. Если все обратные дуги просмотрены и перехода назад не произошло (или если обратных дуг, заканчивающихся в $A[i]$ нет), то продолжить итерацию: $i = i + 1$.

Данный алгоритм тоже будет всегда сходиться, так как размеры множеств IN_{live} ограничены и не убывают.

Ожидается, что в худшем случае этот алгоритм будет использовать работать намного быстрее.

Теперь следует учесть межпроцедурную часть анализа. На этой стадии рассматриваются глобальные живые переменные после вызовов функции. Чтобы распространить информацию о них внутрь вызова, следует просто запустить описанный выше алгоритм ещё раз, но только с соответствующим, вообще говоря, непустым, граничным условием.

3.3 Анализ частных переменных

Имея результаты анализа живых переменных, найти для цикла L множество частных переменных $PRI(L)$ можно построив вспомогательное множество $USE(L)$, определяемое следующим образом: переменная x входит в $USE(L)$ тогда и только тогда, когда существует инструкция, которая может использовать значение переменной x с какого-то из предыдущих витков цикла L или значение, которое имело x при входе в этот цикл.

Очевидно, что $USE(L)$ совпадает с множеством всех переменных, которые имеют хотя бы одно достигающее определение на входе в первый базовый блок рассматриваемого цикла.

Введём множество $PRI^+(L)$:

$$PRI^+(L) = DEF(L) - USE(L)$$

Где $DEF(L)$ – множество всех переменных, которые могут переопределяться в цикле L .

Множество $PRI^+(L)$ является объединением множеств частных и частных по выходу переменных.

Таким образом, множество частных по выходу переменных

$$PRI_{last}(L) = \{ x \mid x \in PRI^+(L) : x \text{ жива на выходе из цикла } L \}$$

и

$$PRI(L) = PRI^+(L) - PRI_{last}(L).$$

4 Возникшие проблемы

Подстановка выражений и массивы

Поскольку достигающие определения строятся только для скаляров, при подстановке нет возможности использовать выражения, содержащие массивы, из других базовых блоков.

Циклы в графе вызова процедур

Первая проблема связана с циклами в графе вызова процедур. Для построения множества def_B и use_B нужно уже иметь результат прохода анализа живых переменных для каждой вызываемой функции. В текущей реализации обработка программ с циклами в графе вызова процедур не реализовано, поэтому предполагается, что этот граф является деревом, и анализ проходит функции в направлении от листьев к корню.

Разрастание занятой памяти

При анализе больших проектов с большим количеством глобальных переменных, для сохранения результатов прохода живых переменных тре-

буется слишком много памяти. Чтобы это исправить, пришлось изменить структуру данных, пожертвовав скоростью работы этого прохода.

5 Результаты

В результате проведённой работы вышеописанные алгоритмы были успешно реализованы и интегрированы в систему SAPFOR. Добавленный код был тщательно протестирован на многочисленных проектах, в том числе на пакете NAS Parallel Benchmarks[1]:

Тест	количество подставленных переменных	количество записей о живых переменных*	количество частных переменных**
BT	2486	72411	348
CG	82	4428	54
EP	33	1282	20
FT	228	3519	98
LU	3045	60864	406
MG	304	7504	71
SP	2269	117580	562

* считается сумма размеров множеств *IN* и *OUT* всех базовых блоков

* сумма размеров множеств *PRI* для всех циклов

Список литературы

- [1] *NAS Parallel Benchmarks*. URL: <https://www.nas.nasa.gov/software/npb.html>.
- [2] Альфред В. Ахо и др. *Компиляторы: принципы, технологии и инструментарий*, 2-е изд. 2006.
- [3] *Библиотека Sage++*. URL: <http://www.extreme.indiana.edu/sage/>.
- [4] А.С. Колганов и Н.Н. Королев. “Статический анализ частных переменных в системе автоматизированного распараллеливания Фортран-программ”. В: *Параллельные вычислительные технологии (ПаВТ’2018)*. URL: <http://omega.sp.susu.ru/pavt2018/short/018.pdf>.