Р. И. Идрисов

МЕТОДЫ МЕЖПРОЦЕДУРНОГО АНАЛИЗА[*]

1. ВВЕДЕНИЕ

Межпроцедурный анализ относится в первую очередь к анализу потока данных, который поступает при вызове в процедуру и из неё. Анализ используется для отслеживания передачи константных значений, данных, которые содержатся в одной ячейке, областей массивов. Такой вид анализа используется для контролирования системы типов в средах разработки и при выполнении преобразований в оптимизирующем компиляторе. Можно обойтись без межпроцедурного анализа, если осуществлять подстановку тела процедуры на место вызова (inlining). Это приводит к экспоненциальному увеличению анализируемого кода, но открывает возможность использования обычных методик анализа. Подстановку нельзя реализовать в полной мере, когда граф вызовов содержит контуры, поскольку это потребует неограниченного количества памяти. При частичной подстановке за счёт удаления мёртвого кода глубина рекурсии может быть определена на стадии компиляции, но размер анализируемого кода и в этом случае увеличится экспоненциально. Межпроцедурный анализ приобретает особую ценность в распараллеливающих компиляторах. Если не анализировать код вызываемых процедур, придётся предположить, что все параметры и глобальные переменные могут измениться в результате вызова. Это существенно снизит эффективность результирующего кода, потому что, например, циклы, содержащие вызовы, не будут распараллелены никогда. Алгоритмы межпроцедурного анализа зачастую являются трудоёмкими. Требуется сохранить баланс между скоростью проведения анализа и эффективностью распараллеливания, что сейчас и демонстрируют основные распараллеливающие системы. Целью данной работы является обзор существующих методик, выбор наиболее перспективных алгоритмов и направлений для развития в межпроцедурном анализе.

[*] Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 07-07-12050).

При подготовке обзора использовались публикации по основным системам автоматического распараллеливания FIAT, Polaris, PIPS, Fida, Parafrase-2, R^N, Parascope, PTRAN.

2. ОСНОВНЫЕ МЕТОДИКИ МЕЖПРОЦЕДУРНОГО АНАЛИЗА

Межпроцедурный анализ можно разбить на четыре части: анализ совмещений (alias analysis), протягивание констант (constant propagation), анализ использования переменных и анализ контекста использования. Такое разбиение условно. Эта информация может быть вычислена для каждой процедуры в программе, что позволяет уменьшить объём компиляции, необходимой при изменении кода одной из процедур. В визуальных системах программирования полезно получать эту информацию как можно быстрей для того, чтобы давать пользователю своевременные подсказки.

2.1. Анализ совмещений

Анализ совмещений. (Alias Analysis) [1], [2] помогает предотвратить появление неверного кода в результате оптимизационных преобразований. Например, последовательность присвоений $a = 5$; $b = 3$; $c = a * b$; можно оптимизировать как $c = 15$ при условии, что a и b нигде далее не используются, но это будет неверно, если a и b хранятся в одной ячейке памяти. В таком случае, после присвоения b значения 3, переменная a тоже примет значение 3, и результат будет равен 9. Также анализ совмещений может быть использован в системах разработки программного обеспечения. При разработке больших программных проектов иногда возникает необходимость изменения типа переменной или объекта, для сохранения корректности программы и исключения нежелательных конверсий типов используют анализ совмещений. Обычно выделяют три типа совмещений:

1. **Статическое совмещение** (explicit aliasing) — возникает, когда две переменные с помощью конструкций языка обозначаются как указывающие на одну ячейку памяти (например union в языке C или equivalence в языке Fortran). Анализ таких совмещений не вызывает сложностей.
2. **Совмещение через параметры** (parameter aliasing) — возникает, когда переменная передаётся в функцию в качестве нескольких из формальных параметров или выступает в качестве параметра, но также доступна из глобального окружения.

3. **Динамическое совмещение** или совмещение указателей (pointer aliasing) — возникает вследствие неизвестных значений переменных типа “указатель”, которые могут отвечать также за одну ячейку памяти.

Рассмотрим совмещение по параметрам и динамические совмещения более подробно.

2.1.1. Совмещение через параметры

Для анализа совмещений через параметры в случае отсутствия контуров в графе вызовов программы достаточно обойти граф в прямом направлении и проанализировать каждую передачу параметра по ссылке на предмет одной и той же глобальной переменной. В этом случае мы получим множество пар переменных, которые могут быть совмещены в процессе выполнения программы. Не все эти совмещения могут возникать на практике, потому что алгоритм не учитывает условных переходов и различных контекстов вызова процедуры. В случае если в графе присутствуют контуры, можно использовать итеративный вариант алгоритма.

Для демонстрации алгоритма рассмотрим следующий пример программы:

```
Procedure p(var a,b,c,d,e: integer)
Begin
 If e=0 then exit;
 B=a+b;
 P(d,a,b,c,e-1);
End;

Begin
…
For i=0 to 9 do
 P(a[i*3], a[i*3+1], a[i*3+2], a[i*3+3],4);

End.
```

В данном примере процедура P, при входном параметре $e = 4$ вычисляет ряд частичных сумм (a, $a + b$, $a + b + c$, $a + b + c + d$). В результате работы данного участка программы каждый элемент массива будет дополнен суммой предшествующих элементов. Совмещений по параметрам в этом

случае не возникает, но если изменить вызов процедуры следующим образом:

```
Begin
…
For i=0 to 9 do
 P(a[i*3], a[i*3+1], a[i*3+1], a[i*3+2],4);

End.
```

Возникает совмещение b и c при первом вызове. Продемонстрируем действие итеративного алгоритма поиска совмещений. Граф вызовов для данной программы содержит петлю в вершине, относящейся к процедуре p, алгоритм при рассмотрении каждой процедуры строит множество совмещений, которое получается при её вызове, и протягивает эти данные для вызываемых процедур. Завершение происходит, когда на каком-то из шагов не происходит изменения множества совмещений. На первом шаге анализа процедуры множества совмещений выглядят следующим образом:

```
a->a
b->b,c
c->c,b
d->d
e->e
```

При анализе рекурсивного вызова процедуры множества меняются следующим образом:

```
a->a,b,c
b->a,b,c
c->a,b,c
d->d
e->e
```

Переменная *a* добавляется к множеству совмещаемых переменных *b* и *c*, потому что используется вторым аргументом при вызове функции. Алгоритм завершается после следующего шага и в результате получается, что первые четыре аргумента могут быть совмещены друг с другом.

Этот алгоритм не является точным, поскольку не учитывает возможности вызова процедуры из различных контекстов; реального совмещения всех аргументов при каждом вызове не происходит. Результат нужно рас-

сматривать как множество возможных совмещений переменной. Если переменные не содержатся в одном множестве совмещений, они не могут соответствовать одной ячейке памяти в ходе выполнения программы, а если содержатся — могут соответствовать, но не обязательно соответствуют. Можно увидеть, что результирующие множества никак не зависят от параметра *e*, но если задать $e = 0$ реального совмещения параметров *a* и *b* в ходе выполнения программы не будет, для выявления таких случаев анализ совмещений иногда объединяют с алгоритмом протягивания констант. Значения констант протягиваются вглубь графа вызова аналогично информации о совмещениях, что делает удобным объединение этих двух видов анализа [3].

Вернёмся к случаю $e = 4$. Если учитывать контексты вызова, можно разделить процедуру на несколько, значения совмещений для которых будут различными, а тела — одинаковыми. Такой анализ называется контекстно-чувствительным.

Можно заметить, что стандартный итеративный алгоритм имеет достаточное количество минусов.

2.1.2. Динамическое совмещение

Наибольший интерес и сложность представляет анализ динамических совмещений (совмещений указателей). Информация о совмещениях по параметру действительна на всём протяжении процедуры, а динамические совмещения могут быть различны; они могут быть вычислены для каждого узла (оператора) отдельно. Такой подход даёт более точные результаты, но имеет большие накладные расходы. Его называют узловым (flow-sensitive) анализом. Динамические совмещения могут быть вычислены для процедуры в целом. Такой алгоритм анализа менее точен, но осуществляется с гораздо меньшими затратами (flow-insensitive alias analysis). Как и в случае совмещений по параметру, алгоритмы могут быть чувствительны к пути исполнения (context-sensitive). Такие алгоритмы в русскоязычной литературе называются контекстно-чувствительными. Нечувствительные к пути исполнения алгоритмы дают большой выигрыш в скорости анализа, они исполняются за линейное время, это объясняет их большую распространённость.

Реализовать алгоритм можно при помощи alias-переменных, которые сопоставляются также всем переменным, с которыми данная переменная может быть совмещена в ходе выполнения программы. Таким образом строятся классы эквивалентности. Здесь усматривается аналогия с анализом совмещений по параметрам, только для сбора данных о возможных совме-

щениях нужно проанализировать не только вызовы процедур, но и их код. В случае присутствия операций именования/разыменования вводятся дополнительные фиктивные переменные.

```
Int *b,*c, *d, e;
 b=&c;
 e=*d;
 d=b;
```

В данном случае совмещаются **b* и *c*, **d* и **b*, для **d* и *&c* будут созданы дополнительные фиктивные переменные. Динамические совмещения можно анализировать при помощи итеративного алгоритма, аналогичного алгоритму анализа совмещений по параметру, описанного в 2.1.1. Различия заключаются в определении наличия совмещения, например:

```
Int  a[20],*p, i;
p=&a[0];
For (i=0;i<10;i++) {
  *p=i*5;
    p++;
}
```

В данном случае при выполнении программы указатель *p* будет совмещён с первыми 10 элементами массива *a*, простой анализ совмещений укажет только на первый элемент, а более сложный, но не учитывающий значения констант/границ циклов покажет на возможное совмещение указателя *p* со всеми элементами массива. Анализ получается более сложным, если пытаться точно вычислить совмещения в случаях прямого изменения указателя *p*. В случае, если к значению указателя добавляется число, которое программа получает в качестве входных данных, или случайное число, совмещения не могут быть вычислены точно на этапе статического анализа. Это не единственный случай, в котором невозможно точно вычислить множества совмещений, как и для совмещений по параметру наличие переменной во множестве указывает только на возможное совмещение в ходе выполнения программы, но не на обязательное. Существуют алгоритмы, которые создают различные множества для возможных и обязательных совмещений (переменные, которые будут обязательно совмещены в ходе выполнения программы). Такой анализ является более точным.

В системах разработки программного обеспечения используются алгоритмы, которые способны извлекать информацию не только о переменных, ссылающихся на одну ячейку памяти, но и о переменных, для которых происходит присвоение. Это производится для контроля типов. Например:

```
Int a,b,c,d;
A=5;
B=7;
C=a+b;
D=c*b;
```

При изменении типа переменной *c* на real желательно изменить тип переменной *d* для того, чтобы избежать нежелательной конверсии типов. Такой анализ не является анализом совмещений в чистом виде, но называется так же. Алгоритмы в этом случае классифицируются как отождествляющие (unification-based), для которых наличие присвоения $y = x$ в теле программы вызывает отождествление вершин y и x в графе совмещений (points-to graph), и «не отождествляющие» (subset-based) алгоритмы, для которых аналогичный случай создаёт зависимость $y \in x$ в графе совмещений. Обычно отождествляющие алгоритмы хранят информацию о совмещениях в виде множеств, alias- переменных или пар возможных совмещений, построение ориентированного графа совмещений характерно для не отождествляющих алгоритмов. Эта информация позволяет проследить цепочку получения значения другого типа. Отождествляющие алгоритмы дают более грубый результат, но исполняются за линейное время. «Не отождествляющие» имеют полиномиальную сложность $O(n^3)$ [7], но предоставляют гораздо более детальную информацию о совмещениях переменной. В системах визуального программирования предпочтительно использование «не отождествляющих» алгоритмов, поскольку по такой информации пользователю будет гораздо проще ориентироваться в динамических совмещениях, возникающих в коде программы. В системах, которые предназначены для анализа больших объёмов кода, иногда используются индексированные базы данных [2] для хранения межпроцедурной информации и быстрого доступа оптимизирующих алгоритмов.

2.2. Распространение констант

В некоторых случаях значения или множество возможных значений переменных может быть определено на стадии компиляции программы. Строго говоря, множество значений переменной всегда ограничено её типом и может быть определено на стадии компиляции, но в данном изложении будем считать, что множество значений переменной в таких случаях не определено. Рассмотрим следующий пример:

```
Procedure P(var a,b:integer);
Begin
 If a<50 then exit;
 b=a+b;
End;
…
Begin
 …
i=5;
P(i,j);
…
For i=1 to 20 do
 P(i,j);

End.
```

В этом примере процедура всегда вызывается с параметром $a < 50$, и, следовательно, может быть полностью удалена, потому что суммирование $b = a + b$ не выполняется никогда. В случаях, когда значение переменной может быть определено на стадии компиляции, используется следующая методика:

Распространение констант (constant propagation) — распространение информации о возможных значениях переменной внутрь процедуры, осуществляемое на стадии компиляции. В случае, когда значение единственное, осуществляется замена переменной её значением в теле процедуры [4, 5].

Информация о значениях используется при анализе индуктивных переменных, границ и шагов циклов. Через границы циклов могут быть определены используемые области массивов [6]. В работах PIPS информация о возможных значениях переменной называется начальными условиями (preconditions) и вычисляется для каждого из анализируемых контекстов. Начальное условие не обязательно представляет информацию о конкретном значении переменной; это может быть множество возможных значений переменной. Эта информация может быть очень полезной при анализе возможности распараллеливания цикла. Даже то, что переменная принимает значения строго больше ноля, может сказаться на возможности параллельного исполнения итераций цикла. Также в статьях о системе PIPS вводится такое понятие, как преобразователи (transformers), которое отражает характер изменения переменной в результате выполнения операции. Преобразо-

ватель — функция, определённая для каждого изменяемого параметра функции, определённой в языке программирования, преобразующая множество значений параметра до выполнения функции во множество возможных значений после её выполнения. Начальные условия для следующей из последовательно исполняемых команд получаются в результате действия преобразователя предыдущей команды на начальные условия для неё. Например, если задано начальное условие i > 0 для операции $i = i + 1$, тогда начальные условия для следующей операции будут i > 1. Начальные условия распространяются в прямом направлении, а преобразователи — в обратном.

Простейший алгоритм анализа начальных условий и преобразователей, при отсутствии в графе вызовов циклов, можно осуществить в два прохода. На первом (обратном) проходе можно получить все преобразователи, затем на втором (прямом) проходе получить все начальные условия. Конечно, здесь не имеется в виду, что преобразователи вычисляются точно, это не всегда возможно, потому что в точную функцию преобразователя войдут параметры, которые невозможно вычислить на стадии компиляции, и сама функция будет являться срезом процедуры относительно параметра [7].

Для распространения констант можно использовать итеративный алгоритм, аналогичный 2.1.2, основные отличия алгоритмов, используемых в оптимизирующих компиляторах, заключаются в представлении множества возможных значений переменной.

1) значения переменных представлены с помощью единственного значения, если такое может быть вычислено,
2) значения переменных представлены в виде интервала, например [0, 9] — означает, что переменная может принимать значения внутри этого интервала,
3) значения переменных представлены в виде множественных интервалов,
4) значения переменных представлены в форме kx + b с заданным шагом k, смещением b и диапазоном изменения x в виде интервала;

Также должна быть определена операция объединения двух множеств значений переменной и преобразователи для функций языка.

Иногда распространение констант объединяют с анализом совмещений, называя протягиванием межпроцедурной информации, а остальную часть межпроцедурного анализа — анализом использования данных. Особую важность распространение констант имеет вследствие того, что границы

массивов отсчитываются по границам изменения индексных переменных, которые могут быть переданы внутрь процедуры в качестве параметра.

2.3. Анализ использования переменных

При анализе процедуры нам будут интересны четыре множества переменных:

1) READ — множество переменных, используемых для чтения в теле подпрограммы,
2) WRITE — множество переменных, используемых для записи в теле подпрограммы,
3) IN –множество используемых для чтения внешних переменных и параметров процедуры,
4) OUT — множество переменных, вычисляемых и записываемых в процедуре, используемых последующем коде.

Первые три множества могут быть получены при анализе тела подпрограммы и вызываемых ею подпрограмм, а четвёртое множество — только путём анализа всего кода, который исполняется после процедурного вызова. Такой анализ не производится, если не учитывается возможность вызова процедуры из различных контекстов и влияние контекста на тело процедуры вообще. Множество OUT отражает характеристики не самой подпрограммы, а последующего кода, и может быть использовано для межпроцедурных оптимизаций. Например, может оказаться, что некоторые или все внутрипроцедурные вычисления могут быть удалены как мёртвый код.

Множества READ, WRITE и IN можно получить при помощи итеративного алгоритма в один проход:

1. Изначально все множества принимаются пустыми.
2. Для каждого узла:
   - если присутствует чтение из некоторой переменной, и она не содержится во множестве WRITE — её следует добавить во множество READ;
   - если добавленная переменная не является локальной — она добавляется в IN;
   - если присутствует запись в некоторую переменную — её следует добавить во множество WRITE;
   - если переменная, добавленная во множество WRITE, не является локальной — она добавляется так же в OUT.

3. Для каждого вызова подпрограммы потребуется предварительное вычисление его множеств IN и OUT (хотя бы приблизительное), далее вызов рассматривается как обычный узел, использующий переменные IN и вычисляющий (записывающий) переменные OUT.

Здесь под локальными переменными понимаются переменные, область видимости которых ограничивается данной процедурой. В результате мы получаем первые три множества и верхнюю оценку множества OUT, для точного вычисления которого требуется проанализировать остаток кода программы на предмет использования (чтения) переменных этого множества. Запись без чтения автоматически убирает переменную из множества OUT, поскольку в этом случае результат вычисления не используется, с оговоркой, что запись производится для любого из контекстов вызова процедуры (для любого пути исполнения в графе вызовов программы).

Если при анализе переменных рассматривать каждый массив как одно целое (запись в элемент массива считать записью массива, а чтение переменной массива считать чтением массива) — результат получается слишком грубым. Для более точного анализа следует рассматривать компоненты массива отдельно, что и производится при анализе итераций циклов, использующих массивы, но для глобального анализа этот способ имеет слишком много накладных расходов. Компромиссом является рассмотрение отдельных областей массива в качестве атомарных единиц. Существует множество способов описания областей, и всегда можно выбрать тот, который наиболее подходит для конкретной системы по точности/скорости работы и требуемым объёмам памяти. Рассмотрим основные способы описания областей массивов. Их можно разделить на две группы:

1) точные — дают такой же результат, что и анализ каждой компоненты массива как отдельной переменной;
2) неточные — дают приближённое описание областей массивов, которое требует меньше памяти и вычислительного времени, но даёт огрублённый результат.

2.3.1. Неточные алгоритмы описания областей массивов

Кроме рассмотренных ниже алгоритмов анализа, разработчики Fida [8] включают в состав неточных алгоритмов “Пессимистический (Pessimistic) алгоритм”, который заключается в том, что все массивы предполагаются изменяемыми в процессе работы процедуры (межпроцедурный анализ не осуществляется), и “Классический (Classic)” — анализ массивов как скаляров. Это делается для сравнения их на тестах другими алгоритмами.

2.3.1.1. Регулярные секции (Regular Sections)

Этот алгоритм впервые предложили Каллаган и Кеннеди [9] (Callahan, Kennedy). Области массивов задаются через значения индексных переменных размерностей массива. Регулярные секции делятся на два вида: ограниченные регулярные секции (restricted regular sections) и описания с помощью триплетов (bounded regular sections). В методе ограниченных регулярных секций каждой из размерностей массива сопоставляется одно из следующих выражений:

1) $\perp$ — если индексная переменная может принимать любое значение;
2) α — если переменная принимает только одно, константное значение;
3) $\alpha \pm j_k$ — если индексная переменная связана с другой индексной переменной этого же массива константным смещением.

Таким образом, этот алгоритм может описывать строки, диагонали и ещё некоторые виды регионов в массиве, но несложно придумать такую область, описание которой даст весь массив целиком, потому что не найдётся другого возможного описания. При объединении и пересечении областей требуется процесс нормализации (приведения к зависимости от одной индексной переменной в случае, если есть записи типа 3). Сложность алгоритма объединения областей — $O(d^2)$, где d — размерность массива.

При описании с помощью триплетов (bounded regular sections) каждой размерности массива сопоставляется тройка чисел (l:u:s), где l — нижняя граница значений индексной переменной, u — верхняя граница, а s — шаг изменения значения индексной переменной. Также возможны значения (α.α.α) — если индексная переменная принимает константное значение, ($\perp$.$\perp$.$\perp$) — если значение индексной переменной неизвестно. Этот алгоритм не требует нормализации при объединении областей, его сложность — O(d).

2.3.1.2. Дескрипторы доступа к данным (Data Access Descriptors)

Впервые предложен Валасундарам и Кеннеди [10] (Balasundaram, Kennedy). В алгоритме области массива описываются простыми секциями (simple sections), которые имеют ортогональные и диагональные границы. Ортогональные границы накладывают условие на максимальное и минимальное значение индексной переменной размерности, диагонали накладывают ограничение на пару индексных переменных различных размерностей. Границы хранятся в следующем виде:

1) $\alpha \le x_i \le \beta, i \in [1,n]F$ — ортогональная граница,
2) $\alpha \le x_i - x_j \le \beta, \forall i, j \in [1,n], i \ne jF$ — диагональ,
3) $\alpha \le x_i + x_j \le \beta, \forall i, j \in [1,n], i \ne jF$ — обратная диагональ.

Время построения минимальной оболочки для набора циклов не может быть выражено через их количество и размерность массива. Скорость построения объединения O(d) согласно [11], где d — количество уравнений в описании области массива.

2.3.1.3. Регионы (Regions)

Алгоритм заключается в описании областей массива в виде минимальной выпуклой оболочки (convex hull) [12], которая ограничивается линейными неравенствами. Неравенства делятся на набор индексных выражений (region) и контекст исполнения (execution context). Контекст исполнения отражает ограничения на управляющие параметры цикла. Здесь, в отличие от предыдущего метода, линейные неравенства не ограничиваются только диагональными и ортогональными границами. Это позволяет описать более сложные области, но увеличивает время построения оболочки. Скорость объединения также O(d) согласно [11].

2.3.2. Точные алгоритмы описания областей массивов

2.3.2.1. Образы (Atom Images)

Этот алгоритм впервые предложен Ли и Ю [3]. Идея — описать области доступа к массиву с помощью неравенств на каждую из индексных переменных. Предлагаются две структуры для хранения этих данных:

- **Атом** хранит информацию о ссылке на массив, получаемую из самой ссылки. Это двумерный массив, в котором строки соответствуют индексным размерностям массива, а столбцы строки — коэффициентам при индексных переменных в выражении, используемом для доступа к массиву. Нулевой столбец содержит константу — инвариант цикла. Также с каждым рядом ассоциируется флаг, который отвечает за линейность размерности.
- **Атомное изображение** аналогично **атому**, но содержит также информацию о пределах изменения индексных переменных. Эта информация дописывается в конец массива: после k рядов атома (где k — количество объемлющих циклов), идёт ряд k + 1, в котором находится линейное выражение, отвечающее за верхнюю границу изменения переменной 1,

аналогично для k + i. Все циклы предполагаются нормализованными, начинающимися с 1.

Этим методом точно описываются области доступа к массиву, потому что границы получаются напрямую из границ изменения индексных переменных объемлющих циклов. Главный недостаток такого алгоритма в том, что невозможно получить объединение двух областей. В этом заключается принципиальное отличие от метода **2.3.1.1 Регулярных секций**.

2.3.2.2. Линеаризация (Linearization)

Этот подход предложен Бурке и Сайтроном [13] (Burke, Cytron). В методе память рассматривается как одномерный массив, и все ссылки на массивы преобразуются в ссылки на память. При слиянии двух областей можно огрублять результат, предполагая, что результирующая область полностью занимает всё пространство между дальними границами областей массивов, но тогда метод становится неточным. Автоматически решаются некоторые проблемы анализа совмещений (Aliasing), но серьёзный недостаток этого метода в том, что требуется хранить большое количество неравенств для каждого из массивов, вследствие чего скорость проверки на зависимость понижается. При построении объединения участки одной области просто дописываются к участкам другой, а при построении пересечения нужно проанализировать $m_1 * m_2$ комбинаций, где m — количество участков, используемое для описания области.

2.3.2.3. Межпроцедурный Омега-тест (Interprocedural Omega-test)

Рассмотрим следующий пример:

```
For i=1 to 100 do
a[i,i]=a[50,i];
```

для того, чтобы показать, что области, используемые в качестве левого и правого значения присвоения пересекаются; достаточно найти целочисленное решение системы уравнений

```
i=i
50=i
```

В данном примере сразу же видно, что система имеет решение при $i = 50$, что лежит в диапазоне изменения индексной переменной. Проблема заключается в том, что целочисленное решение произвольной системы уравнений является np-полной задачей.

Метод, предложенный Тангом (Tang) заключается в том, что все границы и шаги циклов записываются в виде системы уравнений и неравенств. Для построения пересечения областей используется целочисленный метод,

являющийся расширением метода исключения переменных Фурье—Моцкина [14]. Этот алгоритм имеет экспоненциальную сложность в худшем случае, но тесты показывают, что его можно использовать в реальных компиляторах и для большинства задач время его работы полиноминально [15].

2.3.3. Комбинированные методы

В реальных компиляторах возможно использование методов, в которых комбинируются точные и не точные алгоритмы описания областей массивов. Наиболее известным комбинированным методом является FIDA (Full Interprocedural Data Dependence Analysis) Майкла Хинда (Hind [8]) из IBM. В этом методе использованы алгоритмы линеаризации и образов. Для определения зависимости производится частичная подстановка процедурного кода на место вызова (результирующий код содержит вызовы без подстановки, подстановка производится только для анализа). Данные, которые при этом не являются необходимыми (для анализа зависимости), отбрасываются (вычисляется срез [7]). Линеаризация используется в тех случаях, когда размерность массива изменяется при вызове процедуры, или используются смещения начала массива. За счёт использования подстановки алгоритм позволяет использовать стандартные методы нахождения зависимости (во многих случаях).

2.4. Анализ контекста использования процедуры

Анализ контекста использования направлен на уточнение оптимизаций путём анализа различных вариантов использования процедуры. Информация о совмещениях, диапазоны изменения переменных и, как следствие, набор возможных оптимизаций зависят от контекста вызова процедуры. Рассмотрим следующий пример:

```
function a(a,b,c)
begin
 if (c=1) a=sin(b);
 else b=asin(a);
end
```

В этом примере подпрограмма может изменять параметр *b* или параметр *a* в зависимости от значения параметра *c*. Контекст вызова процедуры определяется набором параметров совмещений (alias) и информацией о

возможных значениях переменных, которая генерируется в процессе протягивания констант. В данном примере оптимизирующий компилятор может принять решение о замене этой процедуры двумя следующими:

```
function a_1(a,b)
begin
 a=sin(b);
end

function a_2(a,b)
begin
 b=asin(a);
end
```

Будет произведена замена вызовов процедуры *a* на вызовы процедур *a_1* и *a_2* в зависимости от значения параметра *c*. Оговоримся, что такое решение принимается только в результате анализа контекста использования процедуры *a*, потому что возможны такие варианты программы, при которых такая оптимизация не будет возможной. Например:

```
Случай 1:
....
a(a,b, rand(0, 5))

Случай 2:
a(a,b,1);
a(c,d,1);
```

В первом случае явная разделимость процедуры *a* на две разные процедуры не даёт ничего, а во втором — просто не требуется (при условии, что это все вызовы *a* в программе), потому что используется только при c = 1, что даёт возможность найти и удалить «мёртвый код» ветви c <> 1. Замена одной процедуры несколькими называется клонированием (procedure cloning). Предельный случай клонирования процедур (когда процедура клонируется столько раз, сколько вызывается) является аналогом прямой подстановки (inlining). Клонирование процедур уменьшает огрубление информации о поведении процедуры в конкретном контексте. Решение о клонировании принимается на основе анализа контекстов возможного использования и тела процедуры. Разработчики SUIF утверждают, что таким обра-

зом они достигают точности прямой подстановки без излишнего увеличения размера анализируемого кода. В случае, если выполняется частичная подстановка, решение о подстановке принимается на основе анализа графа вызовов. Подстановка применяется к висячим процедурам.

Клонирование процедур и частичную подстановку относят к межпроцедурным оптимизациям [6], также медпроцедурной оптимизацией считается перенос кода за границы процедуры, который является промежуточным вариантом подстановки.

3. ЗАКЛЮЧЕНИЕ

Рассмотренные алгоритмы позволяют сказать, что для каждого из видов анализа не существует какого-либо универсального и точного решения поставленной задачи. Алгоритмы могут быть подобраны по производительности и точности. Область межпроцедурного анализа является развивающейся, различные источники используют разную терминологию. В целом отмечаются тенденции развития точных алгоритмов анализа, основанных на представлении программы в виде графа и тенденции развития быстрых алгоритмов межпроцедурного анализа совмещений для систем визуального программирования. Межпроцедурный анализ на сегодняшний день является обязательной частью современного оптимизирующего компилятора.

СПИСОК ЛИТЕРАТУРЫ

1. Steensgaard B. Points-to analysis in almost linear time // Proc. of POPL'96. — St. Petersburg Beach, Florida, January 1996. — P. 32–41.
2. Heintze N., Tardieu O. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second // Proc. of 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation, Snowbird, Utah, USA, June 20-22, 2001. — ACM Press, 2001 — P. 254–263.
3. Li Z., Yew P.-Ch. Efficient Interprocedural Analysis for Program Parallelization and Restructuring // Proc. of the ACM/SIGPLAN PPEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems, New Haven, Connecticut, July 19-21, 1988. — SIGPLAN Notices. — 1988. — Vol. 23(9). — P. 85–99
4. Schouten D.A. An Overview of interprocedural analysis technologies for high performance parallelizing compilers — Illinois, 1990 — 62 p. — (Tech. Rep. / Center for Supercomputing Research and Development. Univ. of Illinois; N 1005).
5. Евстигнеев В.А., Серебряков В.А. Методы межпроцедурного анализа (обзор) // Программирование. — 1992 — № 3. — С. 4–15.

6. Keryell R. et al. PIPS — A Workbench for Interprocedural Program Analyses and Parallelization / R. Keryell, C.Ancourt, F.Coelho, B.Creusillet, F.Irigoin, P.Jouvelot — Paris, 1996 — 24 p. — (Tech. Rep. / Centre de Recherche en Informatique. Ecole Nationale Sup'erieure des Mines de Paris)
7. Касьянов В.Н., Мирзуитова И.Л. SLICING: Срезы программ и их использование — Новосибирск, 2002 — 116 с.
8. Hind M. et al. An Empirical Study of Precise Interprocedural Array Analysis / M. Hind, M. Burke, P. Carini, S. Midkiff // Scientific Programming — 1994 — Vol. 3, N 3 — P. 255–271
9. Callahan D., Kennedy K. Analysis of Interprocedural Side Effects in a Parallel Programming Environment // J. of Parallel and Distributed Computing. — 1988. — Vol. 5. — P. 517–550.
10. Balasundaram V., Kennedy K. A technique for summarizing data access and its use in parallelism enhancing transformations // Proc. of the SIGPLAN '89 Conf. on Program Language Design and Implementation. — Portland, Orgen. June 1989. —Vol. 24 (7). — P. 41–53.
11. Антонов А.С. Современные методы межпроцедурного анализа программ // Программирование. — 1998. — № 5. — С. 3–14.
12. Triolet R., Irigoin F., Feautrier P. Direct Parallelization of Call Statements // Proc. of the SIGPLAN Symp. on Compiler Construction. — SIGPLAN Notices. — 1986. — Vol. 21 (7). — P. 176–185.
13. Burke M., Cytron R. Interprocedural dependence analysis and parallelization // Proc. of the SIGPLAN Symp. on Compiler Construction — SIGPLAN Notices. — 1986. — Vol. 26 (6). — P. 145–156.
14. Pugh W. The Omega Test: a fast and practical integer programming algorithm for dependence analysis — Univ. of Maryland, 1991 — 10 p. — (ACM 0-89791-459-7/91/0004).
15. Tang P. Exact Side Effects for Interprocedural Dependence Analysis // Proc. of the ACM Internat. Conf. on Supercomputing, Tokyo, Japan, July 1993 — P. 137–146.