Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

В. Н. Касьянов, И. Л. Мирзуитова

SLICING:
СРЕЗЫ ПРОГРАММ И ИХ ИСПОЛЬЗОВАНИЕ

Под редакцией
проф. Виктора Николаевича Касьянова

Новосибирск 2002

УДК 519.68; 681.3.06
ББК З 22.183.49+З 22.174.2

Книга содержит обзор существующих подходов и алгоритмов, используемых при построении срезов программ, а также различных приложений срезов, среди которых отладка, интеграция программ, тестирование потока данных, повторное проектирование и сопровождение программного обеспечения и является восьмой в серии книг, издаваемых Институтом систем информатики СО РАН по проблемам конструирования и оптимизации программ.

Представляет интерес для системных программистов, а также студентов и аспирантов, специализирующихся в области системного и теоретического программирования.

Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 01-01-794) и Научной программы Минобразования РФ «Университеты России» (грант УР.04.01.023).

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

V. N. Kasyanov, I. L. Mirzuitova

SLICING:
PROGRAM SLICES AND THEIR APPLICATIONS

Edited by
prof. V. N. Kasyanov

Novosibirsk 2002

This volume presents an overview of program slicing, including various approaches and algorithms used to compute slices, as well as an overview of applications of program slicing, which include debugging, program integration, dataflow testing, reengineering and software maintenance. It is the eights among books published in A. P. Ershov Institute of Informatics Systems on problems of program construction and optimization.

The volume is of interest for system programmers, students and postgraduate students working in the field of system and theoretical programming.

1. ВВЕДЕНИЕ

Современные технологии создания программ (программных систем) предполагают так называемый *slicing* — выделение из программы ее определенных частей, называемых *срезами программы* (*program slices*). Каждый срез определяется по отношению к некоторому интересующему нас *критерию среза (slicing criterion),* который обычно задается парой (точка программы, множество переменных). Если критерий среза С задан, то те элементы программы, которые прямо или косвенно влияют на значения, вычисляемые в указанных переменных в заданной точке программы, составляют *срез программы относительно критерия С*. Задача вычисления срезов программы в русскоязычной литературе получила название *слайсинга* или *срезания*.

Понятие среза программы было введено и исследовано Вайзером в работах [136–139]. В них Вайзер утверждал, что срезы соответствуют мысленным абстракциям, которые держат в уме программисты при отладке программ, и пропагандировал использование техник срезов программ в отладочных средах.

С тех пор было предложено несколько отличных друг от друга вариантов понятия среза, а также методов их вычисления. Основной причиной такого разнообразия стал тот факт, что различные приложения требуют от срезов различных свойств. Вайзер определил срез программы S как *сокращенную исполняемую программу*, полученную из исходной программы P путем удаления операторов, так что S воспроизводит часть поведения P. Другое общепринятое определение гласит, что срез — это *подмножество* операторов и выражений исходной программы, которые прямо или косвенно влияют на значения, вычисляемые в точке критерия среза, но не обязательно составляют исполняемую программу. Важное различие проводится между *статическими* и *динамическими* срезами. Вычисление статических срезов не требует принятия предположений о входных данных программы, тогда как вычисление динамических срезов предполагает определенные входные значения.

На рис. 1, *а* представлен пример программы, которая запрашивает значение переменной n, а затем вычисляет сумму и произведение первых n целых чисел. На рис. 1, *б* показан срез этой программы относительно критерия (10, product). Как видно из рисунка, все вычисления, не имевшие отношения к окончательному значению переменной product, были удалены.

```
(1)     read(n);                    read(n);
(2)     i := 1;                     i := 1;
(3)     sum := 0;
(4)     product := 1;               product := 1;
(5)     while i <= n do             while i <= n do
        begin                       begin
(6)       sum := sum + i;
(7)       product := product * i;     product := product * i;
(8)       i := i + 1                  i := i + 1
        end;                        end;
(9)     write(sum);                 write(product)
(10)    write(product)
```

а *б*

Рис. 1. Исходная программа (*а*) и ее срез (*б*) относительно критерия (10, product)

В подходе Вайзера срезы вычисляются путем создания последовательных множеств транзитивно подходящих операторов, в соответствии с имеющимися зависимостями по данным и по управлению. Для вычисления срезов используется только доступная статически информация; таким образом, этот тип срезов обозначается как *статический*. Альтернативный метод вычисления срезов был предложен К. Оттенштейном и Л. Оттенштейном [114], которые переформулировали проблему статического среза в терминах проблемы достижимости в так называемом *графе зависимости по данным (ГПЗ)* [54, 99]. ГПЗ представляет собой ориентированный граф с вершинами, соответствующими операторам и выражениям программы, и дугами, соответствующими зависимостям по управлению и по данным между ними. Критерий среза идентифицируется вершиной в данном графе, а срез соответствует всем вершинам графа, из которых достижима интересующая нас вершина. Различные подходы к срезам программ, рассматриваемые ниже, используют модифицированные и расширенные версии ГПЗ в качестве своих внутренних представлений. Еще один подход был предложен Бергеретти и Карре [34], которые определяют срезы в терминах отношений информационного потока, извлекаемых из программы с помощью синтаксического анализа программы.

Все срезы, упоминавшиеся ранее, вычисляются при помощи объединения операторов и предикатов (управляющих выражений) при *обратном*

обходе (т.е. при движении по дугам графа против их направления) управляющего графа или графа зависимостей по данным, начиная с той точки, где находится критерий среза. Поэтому полученные таким образом срезы называются *обратными* статическими срезами. Бергеретти и Карре [34] впервые определили понятие *прямых* статических срезов, хотя сам этот термин был впервые упомянут Репсом и Брикером [120]. Неформально прямой срез состоит из всех операторов и предикатов, зависимых от критерия среза; оператор «зависит» от критерия среза, если значения, вычисляемые оператором, зависят от значений, вычисляемых критерием, либо если значения, вычисляемые критерием, определяют, будет ли исполняться рассматриваемый оператор или нет. И обратные, и прямые срезы[1] вычисляются сходным образом; последние требуют прослеживания зависимостей в прямом направлении, а не в обратном.

Хотя термин «динамический срез программы» был впервые введен Корелом и Ласки [96], само понятие динамического среза можно рассматривать как неинтерактивную вариацию Бальцеровского понятия обратного потокового анализа [26]. В процессе обратного потокового анализа исследуется, как именно информация передается по программе для получения конкретного заданного значения: пользователь интерактивным образом обходит граф, представляющий зависимости по управлению и по данным программы. Например, если значение, вычисляемое оператором s, зависит от значений, вычисляемых оператором t, то пользователь может проследить путь от вершины, соответствующей s, к вершине, соответствующей t. Недавно Чои и др. [48, 111] предложили эффективную реализацию обратного потокового анализа для параллельных программ.

В случае динамического среза программ к рассмотрению принимаются только те зависимости между элементами программы, которые возникают при *данном конкретном* ее исполнении. *Критерий динамического среза* определяет входные данные и отсекает неподходящие вхождения операторов в истории исполнения; обычно этот критерий задается тройкой (входные данные, вхождение оператора, переменная). Другими словами, основное различие между статическим и динамическим подходами к построению срезов программ состоит в том, что динамический срез предполагает *фиксированные* входные данные, тогда как статический срез не делает предположений о входных данных программы. Различные гибридные подходы, использующие для вычисления срезов определенные комбинации статиче-

[1] Если явно не отмечено иное, под срезами в данной статье понимаются обратные срезы.

ской и динамической информации, также описаны в литературе. Чои и др. [48], Дюстервальд и др. [52] и Камкар [83] используют статическую информацию для уменьшения количества вычислений, производимых во время исполнения. Венкатеш [131], Нинг и др. [112], Филд и Тип [57] и Филд и др. [56] рассматривают ситуации с ограниченным *подмножеством* входных данных программы.

На рис. 2 представлена исходная программа и ее динамический срез относительно критерия (n = 2, 8^1, x), где 8^1 обозначает первое вхождение оператора 8 в истории исполнения программы. Заметим, что при входном значении n, равном 2, цикл исполняется дважды, причем каждое из присваиваний x := 17 и x := 18 исполняется по одному разу. В данном примере ветвь **else** оператора **if** может быть исключена из динамического среза, поскольку присваивание 18 переменной x на первой итерации цикла «уничтожается» присваиванием ей 17 на второй итерации[2]. Для сравнения отметим, что *статический* срез программы рис. 2, *а* относительно критерия (8, x) состоит из всей программы целиком.

```
(1)   read(n);                    read(n);
(2)   i := 1;                     i := 1;
(3)   while i <= n do             while i <= n do
      begin                       begin
(4)     if (i mod 2 = 0) then       if (i mod 2 = 0) then
(5)       x := 17                     x := 17
        else                        else
(6)       x := 18;                          ;
(7)     i := i + 1                  i := i + 1
      end;                        end;
(8)   write(x)                    write(x)
```

а *б*

Рис. 2. Программа (*а*) и ее динамический срез (*б*) относительно критерия (n = 2, 8^1, x)

Основной областью применения, которую Вайзер имел в виду при введении понятия разреза, была отладка программ [136—139]: если программа вычисляет ошибочное значение некоторой переменной x в некоторой точке

[2] Фактически можно утверждать, что конструкцию **while** можно заменить оператором **if** из его тела. Этот тип срезов будет описан в разд. 6.

программы, ошибка с наибольшей вероятностью будет найдена в срезе программы относительно переменной x в данной точке. Применением срезов в отладке занимались далее Лайл и Вайзер [109], Чои и др. [48], Агравал и др. [18], Фрицсон и др. [58], Пэн [115] и Пэн и Спаффорд [116].

Кроме этого, было предложено значительное количество других возможных приложений: распараллеливание [138], дифференциация и интеграция программ [68, 72], сопровождение программного обеспечения [61], тестирование [30, 63, 86, 116], обратное проектирование [78, 79, 139] и настройка компиляторов [105]. В разд. 5 приводится обзор использования техники срезов в каждой из этих областей.

Параллельно с исследованиями срезов и до настоящего времени независимо от них велись работы В. Н. Касьянова [4–12, 88–91] по так называемой *конкретизации программ*. Само понятие конкретизации было введено им в конце 70-х годов (т.е. примерно в то же время, что и понятие среза программ) в рамках работ по исследованию технологических возможностей оптимизации программ и развитию трансформационного подхода к программированию. Вместе с тем оно охватывает существенно более широкий класс срезов программ, чем тот, который был рассмотрен Вайзером и его последователями, и включает различные процессы семантической обработки программ при построении их срезов (например, их оптимизацию), которые начинают появляться в работах по срезам программ лишь в последнее время.

Оставшаяся часть данной работы организована следующим образом. В разд. 2 вводятся основные понятия большинства алгоритмов построения срезов: понятия зависимостей по данным и по управлению. Читатели, знакомые с этими понятиями, могут пропустить этот раздел и обращаться к нему по мере необходимости. В разд. 3 содержится обзор методов статических срезов. Вначале мы рассматриваем простые примеры срезов структурированных программ с одними лишь скалярными переменными. Затем включаем в рассмотрение алгоритмы срезов в присутствии процедур, неструктурированного потока управления, составных переменных и указателей, а также параллелизма. Раздел завершается сравнением и классификацией методов статических срезов. Разд. 4 посвящен динамическим методам срезов; структурно он организован подобно разд. 3. Области применения срезов программ рассматриваются в разд. 5. В разд. 6 обсуждаются последние работы по применению техник компиляторной оптимизации для получения более точных срезов. В разд. 7 рассматривается конкретизация программ, позволяющая строить более разнообразные и оптимизированные срезы программ. Разд. 8 — заключение; в нем перечисляются другие подобные работы и сведены воедино основные результаты данного обзора.

2. ЗАВИСИМОСТИ ПО ДАННЫМ И ЗАВИСИМОСТИ ПО УПРАВЛЕНИЮ

Между операторами программы существуют два типа отношений, реализуемых в выполнениях программы: связь по передаче управления – *управляющая связь*, когда один оператор выполняется непосредственно вслед за другим; связь по передаче информации – *информационная связь,* когда один оператор при своем исполнении использует значение переменной, выработанное другим оператором. Поток управления (система управляющих связей) в программе обычно задается в виде так называемого *управляющего графа* (или *уграфа*) [9, 13]. Это ориентированный граф, вершины которого соответствуют операторам, а дуги, соединяющие вершины, отражают возможность передачи управления от одного оператора к другому. Обычно управляющий граф содержит две выделенные вершины Start и Stop, называемые *начальной* (или *входной*) и *конечной*, а все остальные вершины делятся на *преобразователи*, из которых исходит по одной дуге, и *распознаватели*, из каждой из которых исходит не менее двух дуг. Таким образом, каждому выполнению программы соответствует некоторый путь по уграфу от его начальной вершины. Мы будем рассматривать уграфы программ на языках высокого уровня, преобразователи которых — это простые операторы (оператор присваивания и др.), а распознаватели получаются из управляющих выражений, образующих структурные операторы (условный оператор и др.).

Для каждой вершины i уграфа DEF(i) обозначает множество *определяемых* оператором i переменных (*результатов* оператора i) — переменных, значения которых могут определиться или измениться при выполнении оператора i, а REF(i) – множество *используемых* оператором i (*аргументов* оператора i) — переменных, значения которых могут использоваться при его выполнении. Различают несколько типов зависимостей по данным, таких как *истинная* (*информационная* или *потоковая*) зависимость, *выходная* зависимость и *антизависимость* [54]. Истинные зависимости, в свою очередь, подразделяются на *циклически порождаемые* и *циклически независимые*, в зависимости от того, возникают ли зависимости между итерациями цикла или нет. Для построения разрезов используются только потоковые зависимости, причем различие между циклически порождаемыми и циклически независимыми зависимостями не является существенным. Неформально оператор j является *истинно зависимым* от оператора i, если между оператором i и оператором j существует информационная связь. Результаты операторов часто полезно разделять на обязательные и необязательные [9]; переменная является *обязательным* результатом некоторого оператора в

том случае, если каждое выполнение этого оператора в программе обязательно задает значения всем компонентам данной переменной. Причиной необязательности результатов является косвенная адресация и связанная с ней синонимия (совмещение) имен – так называемый aliasing [101, 102], а также структурность переменных и операторов. В случае, когда все результаты операторов обязательные, между операторами i и j существует потоковая зависимость, если существует такая переменная x, что $x \in DEF(i)$, $x \in REF(j)$ и имеется путь из i в j, не содержащий внутри себя операторов, определяющих x. В этом случае говорят, что между операторами i и j существует *потоковая зависимость* (*информационная связь*) по переменной x и определение x в вершине i является *достигающим определением* для вершины j.

Зависимость по управлению обычно определяется в терминах так называемых обязательных преемников (или пост-доминаторов). Вершина j управляющего графа называется *обязательным преемником* (или *пост-доминатором*) вершины i, если все пути, ведущие из i в Stop, проходят через вершину j (см., например [9]). Вершина j является *зависимой по управлению* от вершины i, если справедливы следующие два свойства:

1) существует путь P из i в j такой, что j является обязательным преемником каждой вершины в P, отличной от i и j;

2) j не является обязательным преемником вершины i.

Проблема нахождения зависимостей по управлению в программе с произвольным потоком управления исследуется Ферранте и др. [54]. Для языков структурного программирования все зависимости по управлению легко определяются с помощью простого синтаксического анализа [70]: операторы, стоящие непосредственно[3] внутри операторов **if** или **while**, зависимы по управлению от вершин, соответствующих их выражениям.

На рис. 3 приведен пример управляющего графа для программы рис. 1,*а*. Вершина 7 истинно (информационно) зависит от вершины 4, поскольку в вершине 4 определяется переменная product, в вершине 7 используется переменная product, и существует путь $4 \to 5 \to 6 \to 7$, не содержащий внутри себя определений переменной product. Вершина 7 является зависимой по управлению также от вершины 5, так как существует путь $5 \to 6 \to 7$, в котором вершина 7 является обязательным преемником вершины 6, но не является обязательным преемником вершины 5.

[3] Если оператор, принадлежащий оператору **if**, находится внутри другого оператора **if**, он зависим по управлению только от выражения внутреннего оператора **if**.

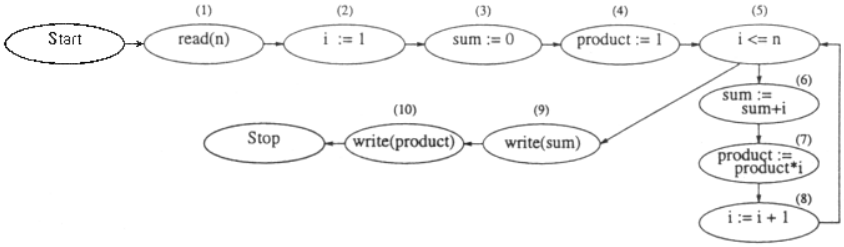


Рис. 3. Управляющий граф для программы на рис. 1, *а*

Многие подходы к срезам, которые будут обсуждаться далее, используют не уграф, а граф программных зависимостей (ГПЗ) в качестве представления программы [54, 98]. В отличие от уграфа дуги ГПЗ представляют отношения зависимостей по данным и по управлению между вершинами. Дуги зависимостей в ГПЗ определяют частичный порядок на множестве операторов программы; операторы должны исполняться согласно этому порядку, чтобы семантика программы сохранялась.

В графе программных зависимостей, предложенном Хорвитц и др. [70, 72, 73, 75], проводится различие между циклически порождаемыми и циклически независимыми потоковыми зависимостями, а также вводит дополнительный тип дуг — def-order-зависимости. Хорвитц и др. утверждают, что их вариант ГПЗ является *адекватным*: если две программы имеют изоморфные ГПЗ, они являются строго эквивалентными. Это означает, что при одинаковых входных данных эти программы либо выдают одинаковые значения, либо обе не применимы. Утверждается, что вариант ГПЗ из [70] является минимальным в том смысле, что удаление дуг зависимостей любой разновидности или игнорирование разницы между циклически порождаемыми и циклически независимыми зависимостями приведет к тому, что неэквивалентные программы могут иметь изоморфные графы. Однако для вычисления срезов программ мы нуждаемся только в знании потоковых зависимостей и зависимостей по управлению. Соответственно, далее будут рассматриваться только эти типы зависимостей.

На рис. 4 приведен пример графа программных зависимостей для программы рис. 1, *а*. Здесь использован ГПЗ в редакции Хорвитц и др. [75]. «Толстые» дуги представляют зависимости по управлению[4], а «тонкие» —

[4] Общепринятая маркировка дуг зависимостей по управлению здесь опущена, поскольку их различия несущественны для нашей задачи, а также не изображаются дуги циклически порождаемых потоковых зависимостей от вершины к ней самой, так как эти зависимости не используются при вычислении срезов.

зависимости по данным. Различие в окраске вершин будет объяснено в п. 3.1.3.

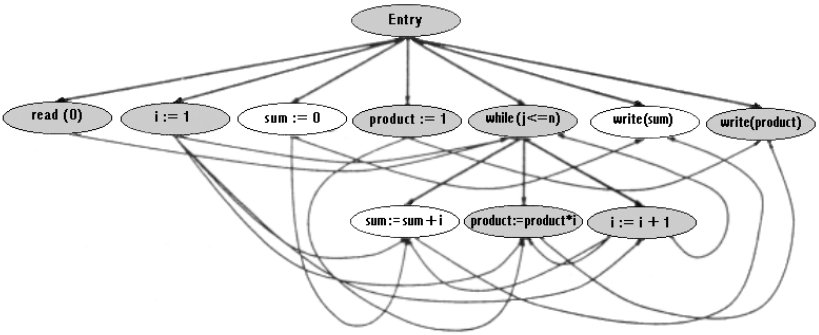


Рис. 4. Граф программных зависимостей для программы рис. 1

3. МЕТОДЫ СТАТИЧЕСКИХ СРЕЗОВ

3.1. Базовые алгоритмы

В этом разделе рассматриваются базовые алгоритмы построения статических срезов для структурированных однопроцедурных программ со скалярными переменными. Эти алгоритмы дают в итоге одинаковые результаты, но вычисляют их различными способами.

3.1.1. Уравнения для потока данных

Первоначальное определение срезов программ, данное Вайзером в работе [138], основано на итеративном решении уравнений потока данных[5]. Вайзер определяет *срез* как *исполняемую* программу, полученную из исходной программы путем удаления нуля или большего числа операторов. *Критерий среза* C состоит из пары (n, V), где n — вершина в управляющем графе программы CFG, а V — подмножество переменных программы. Подмножество S операторов программы P называется срезом относительно критерия (n, V), если справедливы следующие два свойства:

1) S — допустимая исполняемая программа;

2) всегда, когда P останавливает исполнение на заданных входных данных, S также останавливает исполнение на этих входных данных, вычисляя те же значения для переменных из V, когда исполняется оператор, соответствующий вершине n.

Для любого критерия существует по крайней мере один срез: сама исходная программа. Срез является *минимальным*, если ни один другой срез для того же критерия не содержит меньшего числа операторов. Вайзер утверждает, что минимальный срез не обязательно является единственным и что задача нахождения минимальных срезов неразрешима.

Вайзер описывает итеративный алгоритм для вычисления приближенных минимальных срезов. Важно понимать, что этот алгоритм использует *два* различных «слоя» итерации, которые можно охарактеризовать следующим образом:

[5] Данное Вайзером определение операторов ветвей, косвенно подходящих к срезу, содержит ошибку. В настоящей статье используется модифицированное определение, данное в [107].

— трассировка транзитивных зависимостей по данным. Это итеративный процесс при наличии циклов;

— трассировка зависимостей по управлению, которое приводит к включению определенных распознавателей в срезы. Для каждого такого выражения шаг 1 повторяется для включения в срез операторов, от которых он информационно зависит.

Алгоритм определяет последовательные множества *подходящих переменных*, из которых выводятся множества *подходящих операторов*; результирующий срез получается из них как неподвижная точка в ходе последовательного схождения алгоритма. Вначале определяются *непосредственно подходящие переменные*: это пример первого шага итеративного процесса, описанного выше. Множество непосредственно подходящих переменных в вершине i управляющего графа CFG обозначается $R^0_C(i)$. Итерация начинается с начального значения $R^0_C(n) = V$, и $R^0_C(m) = \varnothing$ для любой вершины $m \neq n$. На рис. 5 изображено множество уравнений, которые определяют, как множество подходящих переменных в *конце* j дуги $i \rightarrow_{CFG} j$ управляющего графа CFG влияет на множество подходящих переменных в *начале* i этой дуги. Минимальной неподвижной точкой этого процесса является множество непосредственно подходящих операторов в вершине i. Из R^0_C выводится множество *непосредственно подходящих операторов* S^0_C. На рис. 5 показан процесс вывода S^0_C как множества всех таких вершин i, которые определяют переменную v, подходящую в качестве потомка i в управляющем графе.

Для каждой дуги $i \rightarrow_{CFG} j$ управляющего графа CFG:
$R^0_C(i) = R^0_C(i) \cup \{v \mid v \in R^0_C(j), v \notin DEF(i)\} \cup \{v \mid v \in REF(i), DEF(i) \cap R^0_C(j) \neq \varnothing\}$
$S^0_C = \{i \mid (DEF(i) \cap R^0_C(j)) \neq \varnothing, i \rightarrow_{CFG} j\}$

Рис. 5. Уравнения для определения *непосредственно* подходящих переменных и операторов

Как уже упоминалось, второй «слой» итерации в алгоритме Вайзера состоит из рассмотрения зависимостей по управлению. Переменные, являющиеся аргументами выражения структурного оператора **if** или **while**, считаются *косвенно* подходящими, если по крайней мере один из операторов, содержащихся в данном структурном операторе, является подходящим. *Зона влияния* INFL(b) оператора ветви b определяется как множество опе-

раторов, зависимых по управлению[6] от b. На рис. 6 представлено определение операторов ветвей B^k_C, которые являются косвенно подходящими в силу влияния, которое они оказывают на вершины i в S^k_C. Далее определяется множество косвенно подходящих переменных $R^{k+1}_C(i)$. В дополнение к переменным из $R^k_C(i)$ множество $R^{k+1}_C(i)$ содержит переменные, которые являются подходящими из-за того, что они имеют транзитивную зависимость по данным с операторами из B^k_C. Наличие этой зависимости определяется путем проведения еще одной итерации первого типа (то есть трассировки транзитивных зависимостей по данным) относительно множества критериев (b, REF(b)), где b — оператор ветви в B^k_C (см. рис. 6). На рис. 6 также показано определение множества S^{k+1}_C *косвенно подходящих операторов* на итерации k+1. Это множество состоит из вершин B^k_C и вершин i, в которых определяются переменные, подходящие в контексте R^{k+1}_C к преемнику j в управляющем графе.

Множества R^{k+1}_C и S^{k+1}_C являются неубывающими подмножествами переменных и операторов программы соответственно; неподвижная точка множеств S^{k+1}_C и представляет собой искомый срез программы.

В качестве примера рассмотрим срез программы рис. 1, *а* относительно критерия (10, {product}). В табл. 1 сведены воедино множества DEF, REF, INFL и множества подходящих переменных, вычисленных алгоритмом Вайзера. Управляющий граф программы был представлен ранее на рис. 3. Из информации, представленной в таблице, и определения среза мы получаем: $S^0_C = \{2, 4, 7, 8\}$, $B^0_C = \{5\}$, $S^1_C = \{1, 2, 4, 5, 7, 8\}$. В нашем примере неподвижная точка множеств косвенно подходящих переменных достигается на множестве S^1_C. Соответствующий срез относительно критерия C ≡ (10, {product}), вычисленный алгоритмом Вайзера, совпадает с программой, показанной на рис. 1, *б*, за исключением того, что выходной оператор write(product) отсутствует в этом срезе.

Лайе [107] представляет модифицированную версию алгоритма Вайзера. Помимо некоторых изменений в терминологии, по существу алгоритм является тем же самым.

[6] В действительности Вайзер определяет INFL(b) как множество операторов, транзитивно зависимых по управлению от b. Определение, которое приводим мы, предложено Трипом [131] и приводит в итоге к эквивалентным срезам, но является более эффективным.

Таблица 1

Результаты алгоритма Вайзера для программы на рис. 1, *а* и критерия среза (10, product)

B.	DEF	REF	INFL	R^0_C	R^1_C
1	{n}	∅	∅	∅	∅
2	{i}	∅	∅	∅	{n}
3	{sum}	∅	∅	{i}	{i, n}
4	{product}	∅	∅	{i}	{i, n}
5	∅	(i, n}	{6,7,8}	{product, i}	{product, i, n}
6	{sum}	{sum, i}	∅	{product, i}	{product, i, n}
7	{product}	{product, i}	∅	{product, i}	{product, i, n}
8	{i}	{i}	∅	{product, i}	{product, i, n}
9	∅	{sum}	∅	{product}	{product}
10	∅	{product}	∅	{product}	{product}

$$B_C^k = \{b \mid \exists i \in S_C^k, i \in \mathrm{INFL}(b)\}$$

$$R_C^{k+1}(i) = R_C^k(i) \cup \bigcup_{b \in B_C^k} R^0_{(b,\mathrm{REF}(b))}(i)$$

$$S_C^{k+1} = B_C^k \cup \{i \mid DEF(i) \cap R_C^{k+1}(j) \neq \varnothing, i \rightarrow_{\mathrm{CFG}} j\}$$

Рис. 6. Уравнения для определения *косвенно* подходящих переменных и операторов

Хослер [67] переформулировал алгоритм Вайзера в функциональном стиле. Для каждого типа оператора (пустой оператор, оператор присваивания, составной оператор, оператор **if**, оператор **while**) он определяет две функции, δ и α. Грубо говоря, эти функции выражают то, каким образом оператор преобразует множество подходящих переменных R^i_C и множество подходящих операторов S^i_C соответственно. Функции δ и α строятся с использованием правил композиции. Для пустых операторов и операторов присваивания функции δ и α могут быть выведены из операторов с помощью простого синтаксического анализа. Функции δ и α для составных операторов и операторов **if** могут быть получены из функций δ и α их компо-

нент. Функции δ и α для оператора **while** находятся посредством эффективного преобразования его в бесконечную последовательность вложенных операторов **if**.

3.1.2. Отношения информационного потока

Бергеретти и Карре [34] вводят некоторое количество *отношений информационного потока*, которые могут быть использованы для вычисления срезов. Для оператора (или последовательности операторов) S, переменной v и выражения (которое является управляющим структурного оператора или правой частью оператора присваивания) e, которое встречается в S, вводятся отношения λ_S, μ_S и ρ_S. Эти отношения информационного потока обладают следующими свойствами:

1) $(v, e) \in \lambda_S$ тогда и только тогда, когда значение переменной v на его входе в S потенциально влияет на значение, вычисляемое выражением e;

2) $(e, v) \in \mu_S$ тогда и только тогда, когда значение, вычисляемое выражением e, потенциально влияет на значение переменной v на выходе из S;

3) $(v, v') \in \rho_S$ тогда и только тогда, когда значение переменной v на входе в S может влиять на значение переменной v' на выходе из S.

Множество E^v_S *всех* значений e, для которых $(e, v) \in \mu_S$, может быть использовано для конструирования *частичных операторов*. Частичный оператор для оператора S, ассоциированный с переменной v, получается путем замены всех операторов S, не содержащих выражений из E^v_S, пустыми операторами. Таким способом получается срез относительно последнего значения v.

Отношения информационного потока вычисляются посредством восходящего синтаксического анализа. Для пустого оператора отношения λ_S и μ_S являются пустыми, а ρ_S — тождеством. Для присваивания $v := e$ λ_S содержит пары (v', e) для всех переменных v', которые встречаются в e, μ_S состоит из (e, v), а ρ_S содержит (v', v) для всех переменных v', которые встречаются в e, а также (v'', v'') для всех переменных $v'' \neq v$. На рис. 7 показано, каким образом отношения информационного потока для составных операторов, условных операторов и операторов циклов конструируются из отношений информационного потока для их составных частей. Здесь ε означает пустой оператор, $'\cdot'$ — композиция отношений[7], ID — отношение тождества, VARS(e) — множество переменных, встречающихся в выражении e, DEFS(S) — множество переменных, которые могут быть определены в опе-

[7] Композиция двух отношений R_1 и R_2 содержит все пары (e_1, e_3), для которых существует такое выражение e_2, что $(e_1, e_2) \in R_1$ и $(e_2, e_3) \in R_2$.

раторе S. Отношение-свертка для конструкта **while** получается путем эффективного преобразования его в бесконечную последовательность вложенных операторов **if**. Отношение ρ^* обозначает транзитивное и рефлексивное замыкание ρ.

Срез относительно значения переменной v в произвольном месте может быть вычислен с помощью вставки фиктивного оператора пересылки $v' := v$ в соответствующее место, где v' — это переменная, не встречавшаяся до этого в S. Срез относительно последнего значения v' в модифицированной программе будет эквивалентен срезу относительно v и выбранной точки исходной программы.

$\lambda_\varepsilon = \varnothing$ $\mu_\varepsilon = \varnothing$ $\rho_\varepsilon = ID$	$\lambda_{S1;S2} = \lambda_{S1} \cup (\rho_{S1} \cdot \lambda_{S2})$ $\mu_{S1;S2} = (\mu_{S1} \cdot \rho_{S2}) \cup \mu_{S2}$ $\rho_{S1;S2} = \rho_{S1} \cdot \rho_{S2}$	$\lambda_{v:=e} = VARS(e) \times \{e\}$ $\mu_{v:=e} = \{(e, v)\}$ $\rho_{v=e} = (VARS(e) \times \{v\}) \cup$ $(ID — (v, v))$

$\lambda_{\textbf{if e then S1 else S2}} = (VARS(e) \times \{e\}) \cup \lambda_{S1} \cup \lambda_{S2}$
$\mu_{\textbf{if e then S1 else S2}} = (\{e\} \times DEFS(S_1) \cup DEFS(S_2)) \cup \mu_{S1} \cup \mu_{S2}$
$\rho_{\textbf{if e then S1 else S2}} = (VARS(e) \times (DEFS(S_1) \cup DEFS(S_2)) \cup \rho_{S1} \cup \rho_{S2}$

$\lambda_{\textbf{while e do S}} = \rho^*_S \cdot ((VARS(e) \times \{e\}) \cup \lambda_S)$
$\mu_{\textbf{while e do S}} = (\{e\} \times DEFS(S)) \cup \mu_S \cdot \rho^*_S \cdot ((VARS(e) \times DEFS(S)) \cup ID)$
$\rho_{\textbf{while e do S}} = \rho^*_S \cdot ((VARS(e) \times (DEFS(S)) \cup ID)$

Рис. 7. Определение отношений информационного потока

Прямые статические срезы могут быть получены из отношения λ_S способом, близким к способу вычисления обратных статических срезов из отношения μ_S.

На рис. 8 представлено отношение информационного потока μ для программы, приведенной на рис. 1, *а*[8]. Из этого отношения следует, что множество выражений, которые потенциально влияют на значение product в

[8] Бергеретти и Карр не определяют отношений информационного потока для операторов ввода/вывода. Для наглядности изложения мы предполагаем, что оператор read(n) рассматривается как оператор n := НекотораяКонстанта, а операторы write(sum) и write(product) рассматриваются как пустые.

конце программы, равно {1, 2, 4, 5, 7, 8}. Соответствующий частичный оператор получается путем удаления всех операторов исходной программы, не содержащих выражений из этого множества, то есть обоих операторов присваивания sum и обоих операторов write. Полученный в результате срез идентичен срезу, вычисленному алгоритмом Вайзера (см. п. 3.1.1).

Номер выражения	Потенциально затрагиваемые переменные
1	{n, sum, product, i}
2	{sum, product, i}
3	{sum}
4	{product}
5	{sum, product, i}
6	{sum}
7	{product}
8	{sum, product, i}
9	∅
10	∅

Рис. 8. Отношение информационного потока μ для программы на рис. 1, *а*. Номер выражения соответствует номеру строки на рис. 1, *а*

3.1.3. Подходы, основанные на графах зависимостей

К. Оттенштейн и Л. Оттенштейн [114] впервые сформулировали задачу нахождения срезов как проблему достижимости в графе программных зависимостей (ГПЗ). Они использовали ГПЗ [54, 99] для построения статических срезов для однопроцедурных программ.

В подходах, основанных на графах зависимостей, критерий среза отождествляется с вершиной v управляющего графа. В терминологии Вайзера это соответствует критерию (n, V), где n — вершина управляющего графа, соответствующая v, а V — множество *всех* переменных, определяемых или используемых в v (за исключением введенных Джексоном и Роллинзом мелкозернистых графов зависимостей, о которых речь ниже). Однако мы считаем это различие несущественным; в разд. 3.6.2 будет обсуждаться вопрос, как можно «имитировать» методами ГПЗ-среза более точные критерии. Для однопроцедурных программ срез относительно критерия v состо-

ит из всех вершин, которые могут достигать v. Соответствующие фрагменты исходного текста программы могут быть найдены при помощи установления соответствия вершин ГПЗ и исходного текста во время построения графа зависимостей.

Вариант ГПЗ, предлагаемый К. Оттенштейном и Л. Оттенштейном [114], существенно более детализирован, чем вариант Хорвитц и др. [75]. В частности, он содержит вершины для каждого (под)выражения программы, а также в него явно включены дескрипторы файлов. В результате операторы read, связанные с не относящимися к критерию переменными, не «отрезаются» от программы, и срезы исполняются корректно при вводе полных входных данных исходной программы.

На рис. 4 представлен ГПЗ для программы на рис. 1, *а*. Затемненные области обозначают вершины, принадлежащие срезу программы относительно критерия write(product).

Джексон и Роллинз [78] вводят более мелкозернистый вариант графа зависимостей, в котором добавлены зависимости между отдельными переменными, определенными или использованными в соответствующих точках программы. Преимущество этого подхода состоит в том, что критерий среза является более детализированным, чем в обсуждавшихся ранее алгоритмах, основанных на графах зависимостей. Срез относительно любой переменной, которая определяется или используется в некоторой точке программы, может быть получен напрямую из графа зависимостей. Это дает возможность более точно распознать, какие переменные ответственны за включение того или иного оператора в срез.

Каждая вершина представляет собой *блок*, который содержит отдельный *порт* для каждой переменной, определенной в этой точке программы, а также для каждой переменной, используемой в этой точке. Отношения зависимости между переменными, используемыми в точке программы p, и переменными, определяемыми в этой точке, представлены *внутренними* дугами зависимостей *внутри* блока p. Зависимости по данным между операторами определяются обычным образом, в терминах достигающих определений. Зависимости по управлению между операторами моделируются как «фиктивные» зависимости по данным. Каждый блок имеет порт ε, который представляет «исполнение» соответствующего оператора. Предполагается, что управляющие предикаты создают временное значение, представленное портом τ. Если оператор из блока p зависим по управлению от оператора из блока q, это моделируется с помощью дуги зависимости от τ-порта p к ε-порту q. Наконец, зависимости от константных значений и

входных данных представлены портами γ — роль этих портов несущественна в контексте данного рассмотрения.

Джексон и Роллинз обобщают традиционное понятие критерия среза до пары (*источник*, *сток*), где источник (source) — множество портов определений, а сток (sink) — множество портов использований. Срез обобщается до так называемого *скола*: определяется подмножество операторов программы, которые оказывают влияние на элементы источника или элементы стока. Концептуально скол может быть произведен путем решения проблемы достижимости в модульном графе зависимостей. Однако Джексон и Роллинз формально определяют свой алгоритм (его мы здесь не приводим) в виде набора отношений между портами. Утверждается, что обычные понятия обратного и прямого срезов могут быть выражены в терминах скола.

3.2. Процедуры

Основная проблема, возникающая при построении межпроцедурных статических срезов, состоит в следующем: для того чтобы вычислить точные срезы, необходимо принимать в расчет структуру вызовов и возвратов на путях межпроцедурного исполнения. Простые алгоритмы, которые включают операторы в состав среза после *однократного* обхода программы (точнее, некоторого ее представления), имеют ту особенность, что они рассматривают *неосуществимые* пути исполнения, из-за чего срезы чересчур разрастаются в размерах. Некоторые варианты решения этой проблемы, часто называемой «проблемой контекста вызовов», будут рассмотрены ниже.

3.2.1. Уравнения для потока данных

Подход Вайзера к межпроцедурному статическому срезу [139] включает в себя три отдельные задачи.

1. Вычисляется межпроцедурная *сводная информация* с использованием ранее разработанных техник [138]. Для каждой процедуры P вычисляется множество MOD(P) переменных, которые могут быть модифицированы в P, и множество USE(P) переменных, которые могут быть использованы в P. В обоих случаях принимается в расчет эффект процедур, транзитивно вызываемых в P.

2. Вторым компонентом алгоритма Вайзера является алгоритм внутрипроцедурного среза. Этот алгоритм мы обсуждали ранее в п. 3.1.1. Однако здесь он слегка расширен, чтобы иметь возможность определять воздействие операторов вызова на те множества подходящих переменных и опера-

торов, которые вычислены. Это достигается с помощью информации, полученной на первом этапе. Вызов процедуры P обрабатывается как условный оператор присваивания «**if** некоторое выражение **then** MOD(P) := USE(P)», в котором формальные параметры заменены соответствующими фактическими. В случаях, когда программа вызывает внешние процедуры и недоступен их исходный код, нужно предполагать худшие возможности.

3. Третья часть представляет собой собственно алгоритм построения межпроцедурного среза, который итеративно генерирует новый критерий среза, относительно которого вычислялись внутрипроцедурные срезы на втором этапе. Для каждой процедуры P генерируются новые критерии для (i) процедур Q, вызываемых в P, и (ii) процедур R, которые вызывают P. Новый критерий (i) состоит из всех пар (n_Q, V_Q), где n_Q — последний оператор Q, а V_Q — множество подходящих переменных P в пределах Q (формальные переменные заменяют фактические). Новый критерий (ii) состоит из всех пар (N_R, V_R), где N_R — вызов P в R, а V_R — множество подходящих переменных в первом операторе P, находящемся в пределах Q (фактические переменные заменяют формальные).

Вайзер формализует обобщение нового критерия с помощью функций UP($\mathscr{S}$) и DOWN($\mathscr{S}$), которые отображают множество $\mathscr{S}$ критериев среза в процедуре P на множество критериев в процедурах, вызывающих P, и множество критериев в процедурах, вызываемых P, соответственно. Множество *всех* критериев, относительно которых вычисляются внутрипроцедурные срезы, состоит из транзитивного и рефлексивного замыкания отношений UP и DOWN; оно обозначается как $(UP \cup DOWN)^*$. Таким образом, для начального критерия C будут вычислены срезы для всех критериев из множества $(UP \cup DOWN)^*$ ({C}).

Вайзер определяет критерии в этом множестве «по запросу»: генерация нового критерия (шаг 3) и вычисление внутрипроцедурных срезов (шаг 2) перемешаны, итерации останавливаются, если не был сгенерирован новый критерий. Хотя количество внутрипроцедурных срезов, вычисляемых на шаге (2), может быть уменьшено путем объединения «сходных» критериев (например, замены двух критериев (n, V_1) и (n, V_2) одним критерием (n, $V_1 \cup V_2$)), Вайзер пишет, что «не было реализовано никаких трюков для ускорения» [139]. В действительности можно ожидать, что применение таких «трюков для ускорения» оказало бы существенное влияние на производительность его алгоритма. Дело в том, что для вычисления множеств UP и DOWN требуется, чтобы были известны множества подходящих переменных во всех точках вызова. Другими словами, вычисление этих множеств опирается на срезы соответствующих процедур. В процессе среза новые

переменные могут стать подходящими для встретившихся ранее точек вызова, и могут появиться новые точки вызова. Рассмотрим как пример программу на рис. 9. Ниже L обозначает точку программы, в которой находится оператор write (z), а M — точку программы, в которой находится последний оператор процедуры P. Вычисление среза относительно (L, {z}) требует n итераций в теле цикла **while**. Во время i-й итерации переменные x_1, ..., x_i будут подходящими в точке вызова, в силу чего необходимо включить критерий (M, {y_1, ..., y_i}); в DOWN(Main). Если не предпринимается никаких мер по объединению критериев в DOWN(Main), процедура P будет подвергнута обработке по построению среза n раз.

```
        program Main;
        ...                                  procedure P(y1, y2, ..., yn);
          while (...) do                       begin
            P(x1, x2, ..., xn);                  write(y1);
            z := x1;                             write(y2);
            x1 := x2;                        ...
            x2 := x3;                    (M)   write(yn);
         ...                                   end
            x(n-1) := xn
          end;
(L)     write(z);
        end
```

Рис. 9. Программа, в которой в процедуре P n раз осуществляется срез по алгоритму Вайзера для критерия (L, {z})

То, что алгоритм Вайзера не принимает в расчет, *какие* именно выходные параметры от *каких* входных параметров зависят, является источником неточности среза. На рис. 10, *а* представлен пример программы, которая иллюстрирует эту проблему. Для критерия (4, {d}) алгоритмом межпроцедурного среза Вайзера [139] будет вычислен срез, представленный на рис.10, *б*. Это срез содержит оператор a := 17 из-за мнимой зависимости между переменной a до вызова и переменной d после вызова. Алгоритм Хорвитц—Репса—Бинкли, который будет описан в п. 3.2.3, даст в итоге более точный срез, показанный на рис. 10, *в*.

```
     Program Example;      Program Example;      program Example;
     begin                 begin                 begin
       a := 17;              a := 17;                    ;
(1)    b := 18;              b := 18;              b := 18;
(2)    P (a, b, c, d);       P (a, b, c, d);       P (a, b, c, d);
(3)    write(d)                                    write(d)
(4)  end                   end                   end

     procedure             procedure             procedure
     P(v,w,x,y);           P(v,w,x,y);           P(v,w,x,y);
       x := v;                     ;                   ;
(5)    y := w              y := w                y := w
(6)  end                   end                   end
```

а *б* *в*

Рис. 10. Исходная программа (*а*); срез алгоритма Вайзера относительно критерия (4, {d}) (*б*); срез относительно того же критерия, выполненный алгоритмом Хорвитц—Репса—Бинкли (*в*)

```
     program Example;                  procedure Add (a; b);
     begin                             begin
(1)  read(n);                  (11)    a := a + b
(2)  i := 1;                           end
(3)  sum := 0;
(4)  product := 1;                     procedure Multiply (c; d);
(5)  while i <= n do                   begin
     begin                     (12)    j := 1;
(6)   Add (sum, i);            (13)    k := 0
(7)   Multiply (product, i);   (14)    while j <= d do
(8)   Add (i, 1)                       begin
     end;                      (15)     Add (k, c);
(9)  write(sum);               (16)     Add (j, 1);
(10) write(product)                    end;
     end                       (17)    c := k
                                       end
```

а *б*

Рис. 11. Пример многопроцедурной программы

Хорвитц и др. [75] отмечают, что алгоритм Вайзера для межпроцедурного среза является чересчур неточным из-за так называемой «проблемы контекста вызовов». Вкратце суть ее в следующем: когда вычисления «нисходят» к процедуре Q, вызываемой из процедуры P, они могут «восходить» ко *всем* процедурам, которые вызывают Q, а не только к P. Они включают *невыполнимые* пути, которые имеют началом вызов Q в P, а концом — выход из Q в другую процедуру. Обход таких невыполнимых путей дает в результате неточные срезы.

На рис. 11 представлен пример программы, иллюстрирующей проблему контекста вызовов. Предположим, что должен быть вычислен срез относительно критерия (10, product). Используя сводную информацию для того, чтобы приблизительно оценить действие вызовов, мы предполагаем, что в первом приближении срез состоит из всей основной процедуры за исключением строк 3 и 6. В частности, вызовы процедур Multiply (product, i) и Add (i, 1) будут включены в срез, так как, во-первых, переменные product и i считаются подходящими в этих точках и, во-вторых, можно с помощью межпроцедурного анализа определить, что MOD(Add) = {a}, USE(Add) = {a, b} , MOD(Multiply) = {c} и USE(Multiply) = {c, d}. Поскольку начальный критерий содержится в основной программе, мы получаем, что UP{(10, product)} = ∅ и что DOWN{(10, product)} содержит критерии (11, {a}) и (17, {c, d}). Результатом среза процедуры Add относительно критерия (11, {a}) и процедуры Multiply относительно критерия (17, {c, d}) будет включение этих процедур целиком. Заметим, что вызовы Add в строках 15 и 16 вызывают генерацию нового критерия (11, {a, b}) и, следовательно, повторного среза процедуры Add. Теперь можно увидеть, что для исходной программы существенна проблема контекста вызовов: поскольку строка 11 принадлежит срезу, новые критерии будут сгенерированы для *всех* вызовов Add. В их число входят уже включенные вызовы в строках 8, 15 и 16, а также вызов Add (sum, i) в строке 6. Новый критерий (6, {sum, i}), который будет сгенерирован, вызовет включение в срез строк 6 и 3. Следовательно, окончательный срез будет состоять из целой программы.

Можно видеть, что проблема контекста вызовов алгоритма Вайзера может быть решена с помощью наблюдения, что критерии во множествах UP должны включать только процедуры, которые (транзитивно) вызывают процедуру, содержащую исходный критерий[9]. Как только это сделано, не-

[9] Похожее наблюдение было сделано Джиангом и др. [81]. Однако они не поясняют, что этот подход работает только в том случае, когда вызов процедуры P обрабатывается как условное присваивание: **if** (предикат) **then** MOD(P) := USE(P).

обходимо вычислять *только* множества DOWN. Для исходного критерия C это соответствует определению множества критериев $DOWN^*$ (UP^* ({C})) и вычислению внутрипроцедурных срезов относительно каждого из этих критериев. Репс предположил, что это по существу совпадает с двумя проходами алгоритма Хорвитц—Репса—Бинкли (см. п. 3.2.3), если все множества UP вычисляются до определения первого из множеств DOWN.

Хванг и др. [76] предложили итеративное решение для проблемы межпроцедурного статического среза, основанное на замене (рекурсивных) вызовов экземплярами тела процедуры. С концептуальной точки зрения каждая итерация состоит из двух следующих шагов. Вначале производится открытая подстановка тел процедур на места вызовов с соответствующей заменой формальных параметров фактическими. Затем срез перевычисляется, при этом все оставшиеся вызовы обрабатываются как пустые, то есть считается, что они не оказывают никакого влияния на потоковые зависимости между окружающими их операторами. Этот итеративный процесс заканчивается, когда полученный на некоторой итерации срез идентичен срезу, полученному на предыдущей итерации, то есть, когда достигнута неподвижная точка. Предполагается, что между операторами различных расширенных версий программы и операторами исходной программы поддерживается имеющееся соответствие.

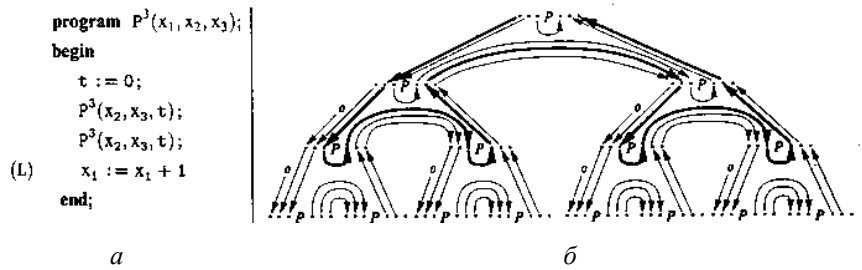


Рис. 12. Исходная программа (*а*); путь экспоненциальной длины, по которому алгоритм Хванга—Ду—Чоу обходит граф вызовов при межпроцедурном статическом срезе для критерия (L,x_3) (*б*)

Подход Хванга и др. не страдает от проблемы контекста вызовов, так как расширение рекурсивных вызовов не приводит к возникновению невыполнимых путей исполнения программы. Однако Репс [119, 121] показал, что для определенного семейства P^k рекурсивных программ этот алгоритм

требует времени исполнения $\mathbf{O}(2^k)$, то есть экспоненциального по отношению к длине программы. Пример такой программы представлен на рис. 12, *а*. На рис. 12, *б* показан экспоненциально длинный путь, который эффективно обходится алгоритмом Хванга—Ду—Чоу.

3.2.2. Отношения информационного потока

Бергеретти и Карре [34] поясняют, каким образом эффект от вызова процедуры может быть приближенно вычислен в отсутствии рекурсивных вызовов. Точные зависимости между входными и выходными параметрами вычисляются путем среза вызываемой процедуры относительно каждого выходного параметра (то есть вычисление μ-отношения для процедуры). Затем каждый вызов процедуры заменяется набором присваиваний, где каждому выходному параметру присваивается фиктивное выражение, содержащее входные параметры, от которых зависит данный выходной параметр. Поскольку рассматриваются только выполнимые пути исполнения, этот подход не страдает от проблемы контекста вызовов. Вызов функции, не вносящей побочных эффектов, может быть смоделирован путем замены ее на фиктивное выражение, содержащее все входные параметры.

Заметим, что вычисленные срезы не являются на самом деле межпроцедурными, так как они не были расширены до иных процедур, чем та, которая содержит критерий среза (как это было сделано в алгоритме, представленном в п. 3.2.1).

Для программы на рис. 11, *а* срез относительно последнего значения product будет включать все операторы, кроме sum := 0, Add(sum, i) и write(sum).

3.2.3. Графы зависимостей

Хорвитц и др. [75] описывают алгоритм для вычисления точных межпроцедурных статических срезов, состоящий из трех этапов:

1. Построение *графа системных зависимостей* (*ГСЗ*), являющегося теоретико-графовым представлением для многопроцедурных программ.
2. Вычисление сводной межпроцедурной информации. Эта информация имеет форму точных отношений зависимостей между входными и выходными параметрами каждого вызова процедуры и явно представлена в ГСЗ в форме *сводных дуг*.
3. Двухпроходной алгоритм извлечения межпроцедурных срезов из ГСЗ.

Мы начнем с краткого рассмотрения графов системных зависимостей. При дальнейшем обсуждении следует помнить, что параметр, передаваемый по значению-результату[10], моделируется следующим образом:

1) вызывающая процедура перед вызовом копирует свои фактические параметры во *временные* переменные;
2) формальные параметры вызываемой процедуры инициализируются с использованием соответствующих временных переменных;
3) перед возвратом вызываемая процедура копирует окончательные значения формальных параметров во временные переменные;
4) после возврата вызывающая процедура обновляет фактические параметры путем копирования значений соответствующих временных переменных.

Граф системных зависимостей содержит граф программных зависимостей для основной программы и граф процедурных зависимостей для каждой процедуры. Имеется несколько типов вершин и дуг графа системных зависимостей, отсутствующих в графах программных зависимостей. Для каждого оператора вызова в ГСЗ имеется *вершина точки вызова*, а также вершины *фактического входа* и *фактического выхода*, которые моделируют копирование фактических параметров во временные переменные и обратно. Каждый граф процедурных зависимостей имеет входную вершину и вершины *формального входа* и *формального выхода*, моделирующие копирование формальных параметров во временные переменные и обратно[11]. Вершины фактического входа и фактического выхода зависимы по управлению от вершины точки вызова; вершины формального входа и формального выхода зависимы по управлению от входной вершины процедуры. Наряду с этими дугами *внутрипроцедурных* зависимостей, граф системных

[10] Алгоритм Хорвитц—Репса—Бинкли [75] также подходит для передачи параметра по ссылке при условии, что решена проблема совмещения имен. К настоящему времени предложены два подхода: преобразование исходной программы в эквивалентную программу без совмещения имен, либо использование обобщенного понятия потоковой зависимости, которое принимает в расчет возможные схемы совмещения имен. Первый подход дает более точные срезы, а второй — разработанный далее Бинкли [41] — является более эффективным. За более подробным изложением механизмов передачи параметров читатель может обратиться к [20].

[11] Используя межпроцедурный потоковый анализ [27], можно определить множества переменных, которые могут быть изменены данной процедурой. Эта информация может быть использована для удаления вершин фактического выхода и вершин формального выхода для тех параметров, которые ни при каких условиях не могут быть изменены, что дает в итоге более точные срезы.

зависимостей содержит следующие типы дуг *межпроцедурных* зависимостей:

1) дуги зависимостей по управлению между вершиной точки вызова и входной вершиной соответствующего графа процедурных зависимостей;
2) дуги *ввода параметра* между соответствующими вершинами фактического входа и формального входа;
3) дуги *вывода параметра* между соответствующими вершинами формального выхода и фактического выхода;
4) *сводные* дуги, представляющие транзитивные межпроцедурные зависимости по данным.

Второй этап алгоритма Хорвитц—Репса—Бинкли состоит из вычисления сводных дуг между вершинами, представляющими входные и выходные параметры процедуры. Наличие подобной дуги отражает тот факт, что поступающее значение входного параметра может быть использовано в вычислении выдаваемого значения выходного параметра. Алгоритм Хорвитц—Репса—Бинкли определяет сводные дуги путем конструирования атрибутной грамматики, моделирующей отношения вызовов между процедурами (как в графе вызовов). Затем вычисляется граф подчиненных признаков для этой грамматики. Для каждой процедуры в программе в этом графе имеются дуги, которые соответствуют точным транзитивным потоковым зависимостям между его входными и выходными параметрами. Сводные дуги графа подчиненных признаков копируются в соответствующие разделы каждой точки вызова в графе системных зависимостей. Подробное изложение этого процесса можно найти в работе [75]. Недавно были предложены более эффективные алгоритмы для определения сводных дуг [53, 121, 122]; эти алгоритмы будут рассмотрены ниже.

Третий этап алгоритма Хорвитц—Репса—Бинкли представляет собой двухпроходной обход графа системных зависимостей. Сводные дуги ГСЗ помогают «перехитрить» проблему контекста вызовов. Предположим, что срезание начинается в некоторой вершине s. На первой фазе определяются все вершины, из которых s достижима *без передачи управления вызову процедуры*. Дуги транзитивных межпроцедурных зависимостей гарантируют, что вызовы процедур можно обойти без передачи управления на них. На второй фазе определяются остальные вершины среза путем передачи управления на ранее «обойденные» вызовы процедур.

На рис. 13 представлен граф системных зависимостей для программы рис. 11. Межпроцедурный анализ потока данных был использован для удаления вершин, обозначающих вторые параметры процедур Add и Multiply.

На рисунке тонкие сплошные линии обозначают потоковые зависимости; толстые сплошные линии — зависимости по управлению; тонкие пунктирные линии — зависимости вызова, ввода параметра и вывода параметра; толстые пунктирные линии — транзитивные межпроцедурные потоковые зависимости. Вершины, принадлежащие срезу относительно оператора write(product), помечены штриховкой: вершины, определенные на первом шагу алгоритма, заштрихованы слабее; вершины, определенные на втором шагу — сильнее. Ясно, что вершины sum := 0, Add(sum,i) и write(sum) не принадлежат срезу.

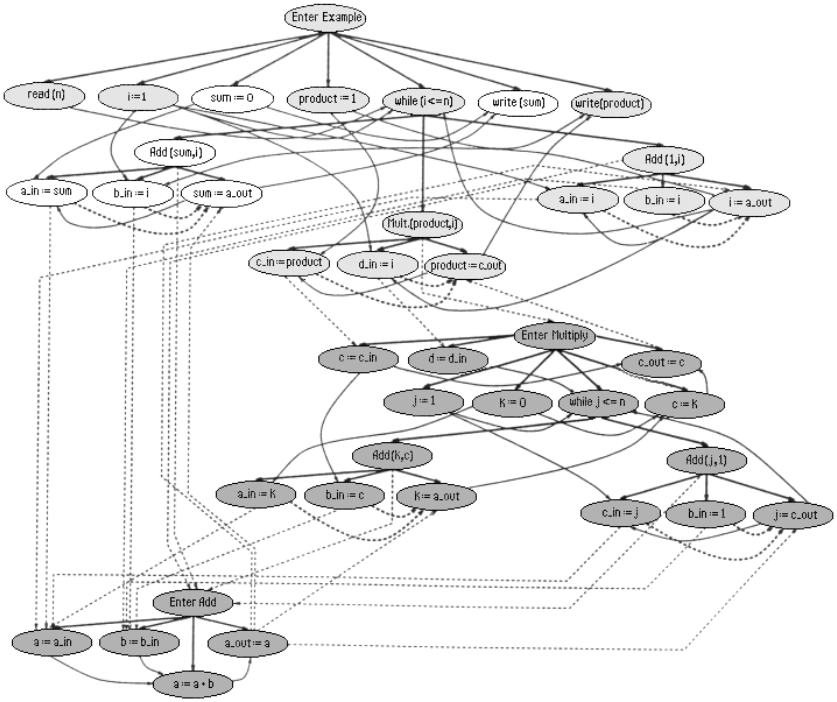


Рис. 13. Граф системных зависимостей для программы на рис. 11

Вычисленные алгоритмом Хорвитц—Репса—Бинкли [75] срезы не обязательно являются исполняемыми программами. Ситуации, когда только подмножество вершин для формальных и фактических параметров вклю-

чаются в срез, соответствуют процедурам, из которых «отрезаются» *некоторые* аргументы; для разных вызовов процедуры могут быть оставлены разные аргументы. Были предложены два подхода для преобразования таких неисполняемых срезов в исполняемые программы. Во-первых, в срез могут быть включены несколько вариантов процедуры [75]; недостаток этого способа состоит в том, что срез более не является сокращенным вариантом исходной программы. Второй способ решения проблемы состоит в расширении среза со всеми параметрами *некоторого* вызова на *все* вызовы, встречающиеся в данном срезе. Дополнительно к этому в срез должны быть включены также и все вершины, от которых зависимы добавленные вершины. Этот последний подход был исследован Бинкли [42]. Ясно, что он дает в итоге срезы большего размера, чем первый.

Таким образом, мы рассмотрели, как межпроцедурные срезы могут быть получены из частичных графов системных зависимостей (соответствующих незавершенным программам либо программам, содержащим вызовы библиотечных процедур) и каким образом с помощью графов системных зависимостей могут быть вычислены *прямые* межпроцедурные срезы подобно тому, как только что были построены обычные (обратные) межпроцедурные срезы.

Недавно Репс и др. [121, 122] предложили новый алгоритм для вычисления сводных дуг графа системных зависимостей, который является асимптотически более эффективным, чем алгоритм Хорвитц—Репса—Бинкли [75] (временные характеристики обоих алгоритмов будут рассмотрены в п. 3.6.3). Подаваемый на вход алгоритма ГСЗ, еще не пополненный сводными дугами, представляет собой набор графов процедурных зависимостей, связанных дугами вызовов, ввода параметров и вывода параметров. Алгоритм использует список заданий для нахождения реализуемых путей *одного уровня*. Неформально реализуемый путь одного уровня согласуется со структурой вызовов и возвратов процедуры, при этом он начинается и заканчивается на одном и том же уровне (то есть в той же самой процедуре). Реализуемые пути одного уровня между вершинами формального ввода и формального вывода в процедуре P индуцируют сводные дуги между соответствующими вершинами фактического ввода и фактического вывода для любого вызова P. Алгоритм начинает с того, что существует реализуемый путь одного уровня нулевой длины из любой вершины формального вывода в нее же саму. Список заданий используется для выбора пути и расширения его путем добавления дуги к его началу. Репс и др. [122] также представили версию своего алгоритма, управляемую запросами, которая *пошагово* определяет сводные дуги графа системных зависимостей.

Лахотия [101] представляет алгоритм для вычисления межпроцедурных секций, также основанный на графах системных зависимостей. Алгоритм вычисляет срезы, идентичные полученным с помощью алгоритма Хорвитц—Репса—Бинкли. С каждой вершиной v графа системных зависимостей ассоциирован тэг с тремя возможными значениями: «⊥» обозначает, что вершина v не посещалась; «T» обозначает, что вершина v посещалась и *все* вершины, достижимые из v, должны быть также посещены; «β» обозначает, что вершина v посещалась и необходимо посетить *некоторые* из вершин, достижимых из v. Более точно, дуга из входной вершины к вершине вызова должна быть пройдена только в том случае, если вершина вызова помечена как T. Алгоритм с рабочим списком используется для организации посещений вершин таким образом, что все вершины, помеченные T, будут посещены раньше, чем любая из вершин, помеченных β. Когда этот процесс заканчивается, вершины, помеченные T или β, будут принадлежать срезу. Алгоритм Лахотия производит однократный обход графа системных зависимостей. Однако в отличие от алгоритма Хорвитц—Репса—Бинкли значение тэга здесь может изменяться *дважды*. Поэтому неясно, является ли на самом деле алгоритм Лахотия улучшением двухпроходного алгоритма Хорвитц—Репса—Бинкли.

Модель графа зависимостей, предложенная Джексоном и Роллинзом [78] (см. п. 3.1.3), является «модульной» в том смысле, что для каждого вызова процедуры имеется отдельный блок. Вместо связывания индивидуальных графов зависимостей для каждой процедуры программы, Джексон и Роллинз представляют вызовы процедур в более абстрактной манере: внутренние дуги зависимостей внутри процедурного блока соответствуют сводным дугам алгоритма Хорвитц и др. [108, 121]. В отличие от ранее обсуждавшихся методов, этот алгоритм обходит проблему контекста вызовов путем расширения срезов исключительно на *вызываемые* процедуры, а не на *вызывающие* (если это не было явно запрошено пользователем). Здесь «расширение среза на вызываемую процедуру» включает построение среза внутри отдельного графа зависимостей соответствующей процедуры относительно подходящих портов ее выходной вершины (т. е. соответствующих портам в точке вызова, входящей в срез).

Тогда как для простых операторов внутренние дуги зависимостей между портами соответствующего блока в графе зависимостей могут быть вычислены простым синтаксическим способом, для процедур требуется более сложная схема. В отсутствие рекурсии внутренние дуги зависимостей для процедуры выводятся из зависимостей внутри и вне блоков операторов, составляющих тело процедуры. Для рекурсивных процедур Джексон и Рол-

линз вкратце описали простую итеративную схему для определения внутренних дуг зависимостей и заявили, что их алгоритм является адаптацией решения, предложенного Эрнстом [53] (см. разд. 6). Суть предложенной ими схемы состоит в том, что внутренние дуги зависимостей для нерекурсивных вызовов определяются описанным выше способом и что изначально не имеется внутренних дуг зависимостей для вызовов в рекурсивном цикле. На каждом последующем шагу транзитивные зависимости между входными и выходными параметрами рекурсивной процедуры перевычисляются путем среза в графе, содержащем сводные дуги, определенные в предыдущем цикле. Затем к графу добавляются сводные дуги для тех зависимостей, которые не появлялись в предыдущем цикле. Этот итеративный процесс завершается, когда найти дополнительные транзитивные зависимости более нет возможности.

3.3. Неструктурированный поток управления

3.3.1. Уравнения для потока данных

Лайе [107] сообщает, что его версия алгоритма Вайзера для статического среза дает неточные срезы в присутствии неструктурированного потока управления: поведение среза не обязательно является проекцией поведения исходной программы. Он представляет консервативное решение для обработки операторов **goto**: каждый оператор **goto**, который имеет непустое множество подходящих переменных, ассоциированных с ним, включается в срез.

Галлахер [60] и Галлахер и Лайе [61] также используют вариацию метода Вайзера. Оператор **goto** включается в срез, если он передает управление на метку включенного в срез оператора[12]. Агравал [16] показывает, что этот алгоритм не во всех случаях дает корректные срезы.

[12] В действительности это является упрощением. Каждый линейный блок разбивается на помеченные блоки; *помеченный блок* представляет собой последовательность операторов линейного блока, начинающуюся с помеченного оператора и не содержащую других помеченных операторов. Оператор **goto** включается в срез, если он передает управление на метку, в помеченном блоке которой имеется оператор, включенный в срез.

Хианг и др. [81] расширяют метод среза Вайзера для программ на языке С с неструктурированным потоком управления. Они вводят некоторое количество дополнительных правил для «сбора» таких операторов неструктурированного потока управления, как **goto**, **break** и **continue**, которые являются частью среза. К сожалению, в [81] не дается никакого формального описания, как нужно обрабатывать конструкции неструктурированного потока управления. Агравал [16] показывает, что этот алгоритм также может давать некорректные срезы.

3.3.2. Графы зависимостей

Болл и Хорвитц [22, 25] и Чои и Ферранте [47] независимо друг от друга обнаружили, что обычные алгоритмы срезов, основанные на графах зависимостей по данным, производят некорректные срезы в присутствии неструктурированного потока управления: срезы могут вычислять значения, отличающиеся от вычисленных исходной программой. Эта проблема возникает из-за того, что алгоритмы не могут корректно определить, когда безусловные переходы, такие как **break**, **goto** и **continue**, должны быть включены в срез.

На рис. 14, *а* приведен вариант нашей исходной программы, в котором использован оператор **goto**. На рис. 14, *б* представлен граф программных зависимостей для этой программы. Вершины, имеющие транзитивные зависимости от оператора write(product), заштрихованы. На рис. 14, *в* показано текстовое представление программы, полученной таким способом. Понятно, что этот срез является неточным, поскольку он не содержит оператора **goto**, и по этой причине исполнение программы не завершается. Фактически, ранее описанный алгоритм, основанный на графе зависимостей по данным, будет включать **goto** в срез *только* тогда, когда этот оператор сам является критерием среза, поскольку ни один оператор не зависит от него ни по управлению, ни по данным.

Решение, предложенное в [22, 25], и первое решение, представленное в [47], весьма похожи друг на друга: безусловные переходы обрабатываются как *псевдовыражения*, у которых ветвь «true» состоит из оператора, на который происходит переход, а ветвь «false» — из *текстуально* следующего за **goto** оператора. Соответственно, две дуги выходят из этой вершины в *расширенном* управляющем графе. Только одна из этих дуг будет действительно пройдена во время исполнения; вторая дуга является неисполняемой. При построении расширенного графа программных зависимостей зависимости по данным вычисляются с использованием исходного управляющего графа, а зависимости по управлению — с использованием расши-

ренного управляющего графа. Срез определяется обычным образом, как проблема достижимости в графе. Метки, принадлежащие исключенным из среза операторам, переносятся в ближайший обязательный преемник (постдоминатор), встретившийся в срезе.

Основное отличие подхода Болла и Хорвитц и первого алгоритма Чои и Ферранте состоит в том, что последний использует несколько более ограниченный язык: условные и безусловные переходы **goto** включены, но нет конструкций структурированного потока управления. Хотя Чои и Ферранте утверждают, что эти конструкции могут быть преобразованы в условные и безусловные переходы **goto**, Болл и Хорвитц показывают, что в некоторых случаях в результате получаются чересчур большие срезы. Обе группы исследователей представили формальные доказательства того, что их алгоритмы вычисляют корректные срезы.

На рис. 14, *г* показан расширенный граф программных зависимостей для программы на рис. 14, *а*; заштрихованы вершины, из которых достижима вершина, помеченная write(product). Корректный срез, соответствующий этим вершинам, представлен на рис. 14, *д*.

Чои и Ферранте обнаружили два недостатка алгоритма построения срезов, основанного на расширенных графах программных зависимостей. Во-первых, расширенные графы программных зависимостей требуют больше места для размещения, чем обычные графы программных зависимостей, и их вычисление требует большего времени. Во-вторых, неисполняемые дуги зависимостей по управлению иногда приводят к появлению ложных зависимостей. Во втором варианте своего алгоритма Чои и Ферранте пользуются «классическим» графом зависимостей по данным. В качестве первого приближения используется стандартный алгоритм вычисления срезов, который сам по себе дает некорректные результаты в присутствии неструктурированного потока управления. Основная идея состоит в том, что для каждого оператора, *не входящего* в срез, добавляется новый оператор **goto**, ведущий к его непосредственному постдоминатору. Отдельной фазой алгоритма является удаление излишних каскадных операторов **goto**. Второй подход лучше первого в том плане, что он дает в итоге срезы меньшего размеры. Недостаток его в том, что полученные срезы могут содержать операторы **goto**, не присутствующие в исходной программе.

read(n);
i := 1;
sum := 0;
product := 1;
while true do
begin
if (i>n) **then**
goto L;
sum := sum + i;
product:=product*i;
i := i + 1
end;
L: write(sum);
write(product)
	read(n);
	i := 1;
	
	product := 1;
	while true do
	begin
	if (i>n) **then**
	;
	
	product:=product*i;
	i := i + 1
	end;
	
	write(product)
		read(n);
		i := 1;
		
		product := 1;
		while true do
		begin
		if (i>n) **then**
		goto L;
		
		product := product * i;
		i := i + 1
		end;
		L:
		write(product)

а	*в*	*д*

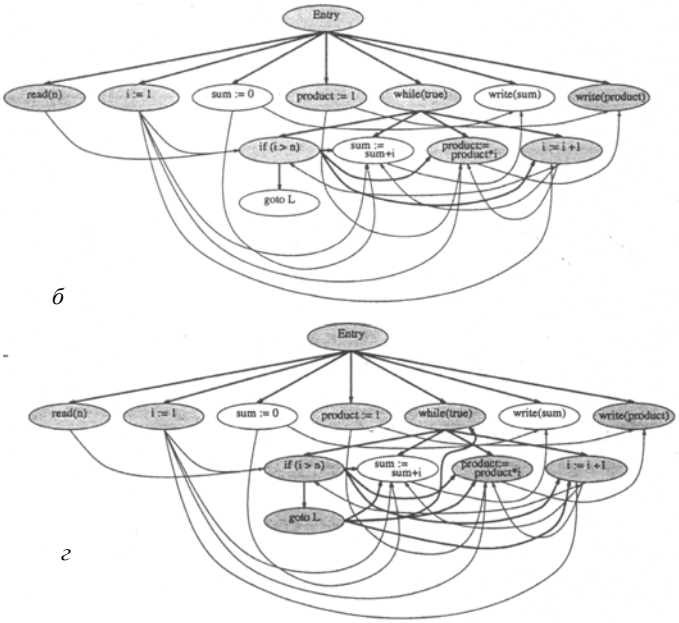


Рис. 14. Программа с неструктурированным потоком управления (*а*); граф программных зависимостей для этой программы (*б*); некорректный срез относительно оператора write(product) (*в*); расширенный граф программных зависимостей программы (*г*); корректный срез относительно оператора write(product) (*д*)

Еще один метод срезания для программ с неструктурированным потоком управления, основанный на графах программных зависимостей, был предложен Агравалом [16]. В отличие от Болла и Хорвитц [22, 25] и Чои и Ферранте [47], Агравал использовал *немодифицированный* ГПЗ. Он заметил, что оператор *условного* перехода вида **if** P **then goto** L должен быть включен в срез в том случае, если выражение P входит в срез, поскольку другой оператор, входящий в срез, зависит от него по управлению. Термин «условный алгоритм среза» был принят для обозначения стандартного метода среза, основанного на графах программных зависимостей, расширенного вышеуказанным способом, со включением условных переходов.

Ключевое наблюдение Агравала состоит в том, что оператор безусловного перехода J должен быть включен в срез тогда и только тогда, когда непосредственный пост-доминатор J, включенный в срез, отличается от непосредственного лексического преемника J, включенного в срез. Здесь оператор S′ называется *лексическим преемником* оператора S, если S текстуально предшествует S′ в программе[13]. Операторы, от которых вновь добавленный оператор транзитивно зависим, также должны быть включены в срез. Мотивация этого подхода может быть понятна, если рассмотреть последовательность операторов S_1; S_2; S_3, где S_1 и S_3 входят в срез, а S_2 содержит оператор безусловного перехода к оператору, непосредственным лексическим преемником которого не является S_3. Предположим, что S_2 не входит в срез. Тогда поток управления в срезе будет безусловно переходить от S_1 к S_3, хотя в исходной программе это могло быть и не так, поскольку оператор перехода мог передать управление в другую точку. Следовательно, оператор перехода должен быть включен в срез вместе со всеми операторами, от которых он зависит. Алгоритм Агравала обходит дерево постдоминаторов программы в ширину и рассматривает операторы перехода на предмет их включения. Алгоритм останавливается, когда уже не может быть добавлено ни одного оператора перехода; это необходимо, поскольку добавление оператора перехода (и зависящих от него операторов) может изменить лексический порядок преемников и постдоминаторов *в срезе* других операторов перехода, которые потом также, возможно, придется включать. Хотя и не приводя формального доказательства, Агравал утверждает, что его алгоритм вычисляет корректные срезы, идентичные вычисляемым с помощью алгоритмов Болла—Хорвитц и Чои—Ферранте.

[13] Как заметил Агравал, это понятие эквивалентно так называемым неисполнимым дугам расширенного управляющего графа, которые использовали как Боллом и Хорвитц, так и Чои и Ферранте.

Алгоритм Агравала [16] можно существенно упростить, если единственный тип условного перехода, встречающийся в программе, представляет собой *структурированный переход*, то есть переход к лексическому преемнику. Операторы языка C **break**, **continue** и **return** представляют структурированные переходы. Вначале требуется только однократный обход дерева постдоминаторов. Затем должны быть добавлены операторы перехода, в том случае, если они зависимы по управлению от предиката, включенного в срез. В этом случае операторы, от которых они зависят, уже содержатся в срезе. Для программ со структурированными переходами алгоритм может быть еще более упрощен до *консервативного* алгоритма путем включения в срез всех операторов перехода, которые зависимы по управлению от предиката, уже содержащегося в срезе.

Алгоритм Агравала включит оператор **goto** программы на рис. 14, *а* в срез, поскольку он зависит по управлению от уже содержащегося в срезе предиката оператора **if**.

3.4. Составные типы данных и указатели

Лайе [107] предлагает консервативное решение проблемы статических срезов при наличии массивов. Суть в том, что любое изменение элемента массива рассматривается как изменение массива.

Вариант графа программных зависимостей, предложенный К. Оттенштейном и Л. Оттенштейном [114], содержит вершину для каждого подвыражения; специальные операторы *select* и *update* служат для доступа к элементам массива.

В случае косвенной адресации (наличие указателей, параметров, подставляемых по имени, и т.д.) могут возникать ситуации, когда две или более переменных указывают на одну и ту же область памяти — феномен, обычно называемый *синонимией* или *совмещением имен*. Совмещение имен усложняет вычисление срезов, поскольку понятие потоковой зависимости зависит от того, какие переменные могут быть (потенциально) совмещены. Даже в случае внутрипроцедурных зависимостей проблема определения потенциальных кандидатов на совмещение имен в присутствии многоуровневых указателей является NP-трудной [102]. Однако срезы могут быть вычислены с использованием консервативных приближений зависимостей по данным, основанных на приближенной информации о совмещении. Консервативные алгоритмы определения потенциальных кандидатов на совмещение были предложены В.Н. Касьяновым [3, 9], Лэнди и Райдером [103] и Чои и др. [46].

Хорвитц и др. [69] предлагают несколько отличающийся подход к вычислению потоковых зависимостей в присутствии указателей. Вместо определения приближенных потоковых зависимостей в терминах определений и использований переменных, которые потенциально могут быть совмещены, определено понятие потоковой зависимости в терминах потенциальных определений и использований *абстрактных областей памяти*. Представлен алгоритм вычисления приближенных схем размещения в памяти, которые могут встретиться в каждой точке программы во время ее исполнения.

Предложенный Агравалом и др. [17] алгоритм для статических срезов, основанных на графах системных зависимостей, реализует сходную идею для обработки составных переменных и указателей. Их решение состоит в нахождении достигающих определений для скалярной переменной v в вершине n управляющего графа посредством установления всех путей из вершин, соответствующих определениям v, в n, не содержащих при этом других определений v. Когда рассматриваются составные типы данных и указатели, определения включают *левые выражения* вместо переменных. Левое выражение представляет собой любое выражение, которое может появиться в левой части оператора присваивания. Агравал и др. представляют новое определение достигающих определений, основанное на размещении областей памяти, потенциально назначаемых левым выражениям. Области памяти рассматриваются как абстрактные значения (например, массив a соответствует областям a[1], a[2],...). Тогда как определение скалярной переменной либо достигает, либо не достигает использования, ситуация становится более сложной, когда разрешены составные типы данных или указатели. Для def-выражения e_1 и use-выражения e_2 могут возникнуть следующие ситуации.

- *Полное пересечение*. Области памяти, соответствующие e_1, являются объемлющим множеством для областей памяти, соответствующих e_2. Примером может быть случай, когда e_1 определяет целую запись b, а e_2 представляет собой использование b.f.
- *Возможное пересечение*. Нельзя определить статически, совпадают ли области памяти, соответствующие e_1, областям памяти, соответствующим e_2. Такая ситуация возникает, когда e_1 является присваиванием элементу массива a[i], а e_2 — использованием элемента массива a[j]. Разыменование ссылок также может привести к ситуации возможного пересечения.
- *Частичное пересечение*. Области памяти, соответствующие e_1, являются подмножеством областей памяти, соответствующих e_2. Это

может случиться, к примеру, когда некоторый массив a используется в e_2, а некоторый элемент a[i] массива a определяется в e_1.

В свете этих понятий определяется расширенная функция достигающих определений, которая обходит управляющий граф в поисках полного пересечения (обязательного результата), делая наихудшие предположения в случаях возможного пересечения и продолжая поиск частей массива записей, пока еще не определенных, в случаях частичного пересечения.

Лайе и Бинкли [108] представляют подход к срезам в присутствии указателей, основанный на вариации символьного исполнения, описанного В.Н. Касьяновым [3, 9]. Алгоритм состоит из двух фаз. Вначале определяются все вершины управляющего графа, которые вводят адреса (либо в силу использования оператора C «&», либо в процессе динамического размещения нового объекта). Эти адреса протягиваются через управляющий граф, давая в итоге множество адресных значений для каждого указателя в каждой точке программы. Набор правил протягивания определяет, каким образом адреса протягиваются через операторы присваивания[14]. Затем с помощью информации, собранной на первой фазе, определяется, какие операторы должны быть включены в срез. Вторая фаза является обобщением алгоритма среза Лайе [107].

			DEF	REF	R^0_C		
(1)	p = &x;		{p}	{(-1)x}	∅	(1)	p = &x;
(2)	*p = 2;		{(1)p}	{p}	{p,(1)q}	(2)	
(3)	q = p;		{q}	{p}	{p,(1)q}	(3)	q = p;
(4)	write(*q)		∅	{q,(1)q}	{q,(1)q}	(4)	
	а			*б*			*в*

Рис. 15. Исходная программа (*а*); множества определенных, использованных и подходящих переменных для данной программы (*б*); некорректный срез относительно критерия C=(4, q, (1)q) (*в*)

Джианг и др. [81] представляют алгоритм для срезов программ на языке C с указателями и массивами, основанный на введенном Вайлем понятии

[14] Свое определение Лайе и Бинкли относят только к программам без циклов и утверждают, что проблемы, связанные с зависимостями по управлению, «ортогональны» проблемам зависимостей по данным, вызываемым указателями.

фиктивных переменных[15] [134]. Основная идея состоит в том, что для каждого указателя p вводится фиктивная переменная (1)p, обозначающая значение, на которое указывает p, а для каждой переменной x — фиктивная переменная (-1)x, обозначающая адрес x. Джианг и др. определяют зависимости по данным обычным способом, в терминах определений и использований фиктивных переменных. К сожалению, этот подход имеет свои недостатки[16]. На рис. 15 представлены исходная программа, множества DEF, REF и R^0_C для каждого ее оператора и некорректный срез, вычисленный для критерия C=(4,q,(1)q). Второй оператор удален некорректно из-за того, что он не определяет ни одной переменной, подходящей для оператора 3.

3.5. Параллелизм

Ченг [45] рассматривает статические срезы для параллельных программ с помощью графов зависимостей. Он обобщает понятия управляющего графа и графа программных зависимостей до *недетерминированной параллельной управляющей сети* и *сети программных зависимостей (СПЗ)* соответственно. Наряду с обычными для ГПЗ дугами зависимостей, СПЗ содержит дуги для зависимостей по выбору, зависимостей по синхронизации и зависимостей по коммуникации. *Зависимость по выбору* напоминает зависимость по управлению, но включает недетерминированные операторы выбора, такие как оператор ALT языка Occam-2. *Зависимость по синхронизации* отражает тот факт, что начало или окончание исполнения оператора зависит от начала или окончания исполнения другого оператора. *Зависимость по коммуникации* соответствует ситуациям, когда значение, вычисленное в одной точке программы, влияет на значение, вычисленное в другой точке программы, через коммуникацию процессов. Статические срезы вычисляются путем решения проблемы достижимости в СПЗ. К сожалению, Ченг не формулирует явно семантику зависимостей по синхронизации и коммуникации, равно как не заявляет и не доказывает никаких свойств вычисленных его алгоритмом срезов.

Интересно, что для построения СПЗ Ченг использует понятие *слабой* зависимости по управлению [45]. Множество слабых зависимостей по управлению является объемлющим по отношению к множеству зависимостей по управлению, разница между ними состоит в том, что слабая зависимость по управлению возникает между управляющим предикатом цикла и операто-

[15] Аналогичное понятие ранее рассматривалось В.Н. Касьяновым [3].

[16] Известен [130] контр-пример (рис. 15), который был предложен Сьюзен Хорвитц.

рами, следующими за ним. Например, операторы в строках 9 и 10 программы на рис. 1, *а* являются слабо (но не являются сильно) зависимыми по управлению от управляющего предиката в строке 5.

3.6. Сравнение

В этом разделе описанные ранее методы статических разрезов будут сравниваться и классифицироваться. Раздел организован следующим образом. В п. 3.6.1 будут сведены воедино проблемы, которым посвящены рассматриваемые работы. В пп. 3.6.2 и 3.6.3 будут сравниваться *точность* и *эффективность* методов построения статических срезов, решающих одни и те же задачи. Наконец, в п. 3.6.4 разбираются возможности комбинирования алгоритмов, решающих разные задачи.

3.6.1. Обзор

В табл. 2, предложенной Типом [130], сведены воедино большинство наиболее важных алгоритмов построения статических срезов. Для каждой статьи в таблице указан метод вычисления срезов и отмечены следующие моменты:

- могут ли быть вычислены межпроцедурные срезы или нет;
- какие конструкции управления рассматриваются;
- рассматриваемые типы данных;
- рассматривается ли параллельность.

Используются следующие сокращения и обозначения: *МВ (метод вычисления)*: D — уравнения для потока данных; F — функциональная/денотационная семантика; I — отношения информационного потока; G — достижимость в графе зависимостей; O — графы зависимостей в комбинации с оптимизационными техниками (см. раздел 6); R — трассировка зависимостей в терминах систем переписывания графов (см. разд. 6); *МР (межпроцедурные решения); ПУ (поток управления)*: С — структурированный, П – произвольный; *ТД (типы данных)*: С — скаляры, М — массивы/записи, У – указатели; *П (параллельность*); верхний индекс имеет следующий смысл: (1) решение некорректно (см. [47]), (2) решение некорректно (см. п. 3.4), (3) рассматриваются только программы без циклов, (4) только нерекурсивные процедуры, (5) принимает в расчет совмещение параметров, (6) полученные срезы являются исполняемыми процедурами.

Важно понимать, что элементы табл. 2 описывают только *рассматривавшиеся проблемы*; они *ничего не говорят* о «качестве» решений (за исключением того, что отдельно указаны некорректные решения). Более того,

в таблице также *не показано*, какие алгоритмы могут быть скомбинированы. Например, алгоритм межпроцедурного среза Хорвитц—Репса—Бинкли [75] теоретически может быть скомбинирован с любым из методов, основанных на графах зависимостей, для обработки неструктурированного потока управления [16, 25, 47]. Возможности таких комбинаций отчасти рассматриваются в п. 3.6.4. Работы Эрнста [53], Филда и др. [56, 57, 128], приведенные в конце табл. 2, опираются на техники, существенно отличающиеся от описанных ранее, и будут описаны отдельно в разд. 6.

Камкар [84] делает различие между методами вычисления срезов, являющихся исполняемыми программами, и методами вычисления срезов, представляющих собой множество «подходящих» операторов. Можно согласиться с наблюдением Хорвитц и др. [75], что для *статического* среза *однопроцедурных* программ — это вопрос представления. Однако для многопроцедурных программ различие *значительно*, как было отмечено в п. 3.2.3. Тем не менее некоторые (например, Тир [130]) полагают, что различие между исполняемыми и неисполняемыми межпроцедурными срезами в данном случае также можно игнорировать, поскольку эти две задачи родственны друг другу: Бинкли [42] показал, каким образом точные исполняемые межпроцедурные статические срезы могут быть получены из неисполняемых межпроцедурных срезов, вычисленных алгоритмом Хорвитц и др. [75].

Последнее замечание касается операторов ввода/вывода. Срезы, вычисленные алгоритмом Вайзера [139] и алгоритмом Бергеретти и Карре [34], вовсе не содержат операторов вывода, поскольку: (1) множество DEF оператора вывода пусто, так что ни один другой оператор не зависим по данным от него; (2) ни один оператор не зависим по управлению от оператора вывода. Хорвитц и Репс [70] предлагают алгоритм для того, чтобы выводимое значение было зависимым от всех предыдущих выводимых значений: они обрабатывают оператор write(v) как присваивание output := output || v, где output — строковая переменная, содержащая все выходные данные программы, а «||» обозначает операцию конкатенации строк. Операторы вывода могут быть включены в срез путем включения output в число переменных критерия.

Таблица 2

Методы статических срезов

Авторы	***МВ***	***МР***	***ПУ***	***ТД***	***П***
Вайзер [106, 139]	D	да	С	С	Нет
Лайе [107]	D	нет	П	С, М	Нет
Галлахер, Лайе [60, 61]	D	нет	П(1)	С	Нет
Хианг и др. [81]	D	да	П(1)	С, М, У(2)	Нет
Лайе, Бинкли [108]	D	нет	С(3)	С, У	Нет
Хослер [67]	F	нет	С	С	Нет
Бергеретти, Карре [34]	I(4)	да	С	С	Нет
Оттенштейн [114]	G	нет	С	С, М	Нет
Хорвитц и др. [72, 73, 123]	G	нет	С	С	Нет
Хорвитц и др. [75]	G	да	С	С	Нет
Бинкли [41]	G(5)	да	С	С	Нет
Бинкли [43]	G(6)	да	С	С	Нет
Джексон, Роллинз [78, 79]	G	да	С	С	Нет
Репс и др. [121, 122]	G	да	С	С	Нет
Лахотия [101]	G	да	С	С	Нет
Агравал и др. [17]	G	нет	С	С, М, У	Нет
Болл, Хорвитц [22, 25]	G	нет	П	С	Нет
Чои, Ферранте [47]	G	нет	П	С	Нет
Агравал [16]	G	нет	П	С	Нет
Ченг [45]	G	нет	С	С	Да
Эрнст [53]	O	да	П	С, М, У	Нет
Филд и др. [56, 57, 128]	R	нет	С	С, У	Нет

3.6.2. Точность

Хотя проблема определения срезов, минимальных по количеству операторов, неразрешима в общем случае, некоторые алгоритмы дают лучшие приближения к минимальным по количеству операторов срезам, чем другие. Для удобства изложения мы будем называть алгоритм *неточным*, если

другой алгоритм вычисляет срезы меньшего размера относительно того же самого критерия.

С первого взгляда может показаться, что сравнение методов статических срезов усложняется тем фактом, что некоторые методы допускают более общие критерии срезания, чем другие. В методах построения срезов, основанных на уравнениях потока данных и отношениях информационного потока, критерий среза представляет собой пару (s, V), где s — оператор, а V — произвольный набор переменных. Однако в методах, основанных на графах программных зависимостей (за исключением «модульных» графов зависимостей Джексона и Роллинза [78]), критерию среза соответствует пара (s, VARS(s)), где s — оператор, а VARS(s) — множество *всех* переменных, *определяемых или используемых в s*.

Впрочем с помощью методов, основанных на графах программных зависимостей, можно вычислять срезы и относительно критериев вида (s, V) для произвольного V, если выполнить следующие три шага. Во-первых, определяется вершина управляющего графа n, соответствующая вершине s графа программных зависимостей. Во-вторых, определяется множество N вершин управляющего графа, соответствующих всем определениям, достигающим переменные из V в вершине n. В-третьих, определяется множество S вершин управляющего графа, соответствующих вершинам из N. Искомый срез состоит из всех вершин, из которых достижимы вершины, содержащиеся в S. Другой способ состоит в том, что в интересующей нас точке программы помещается оператор v := e, где v — некоторая фиктивная переменная, не встречавшаяся ранее в программе, а e — некоторое выражение, содержащее все переменные из V; перевычисляются граф программных зависимостей и срез относительно вновь добавленного оператора.

Рассматривая вопрос о точности алгоритмов, мы можем теперь прийти к некоторым заключениям.

Базовые алгоритмы. При *внутрипроцедурном* статическом срезе точность методов, основанных на уравнениях для потока данных [139] (п. 3.1.1), отношениях информационного потока [34] (п. 3.1.2) и графах программных зависимостей [114] (п. 3.1.3) по существу не различается для разных методов, хотя представление вычисленных срезов различно: Вайзер определяет свои срезы как исполняемые программы, тогда как в двух других методах срезы определяются как подмножества операторов исходной программы.

Процедуры. Алгоритм *межпроцедурного* статического среза Вайзера [139] является неточным по двум причинам. Во-первых, межпроцедурная

сводная информация, используемая для оценки воздействия вызовов процедур, устанавливает отношение между множеством *всех* входных параметров и множеством *всех* выходных параметров; напротив, подходы, предложенные в [34, 75, 76, 121, 122], определяют для каждого выходного параметра, от каких в точности входных параметров он зависит. Во-вторых, алгоритм не в состоянии учесть структуру вызовов и возвратов по межпроцедурным путям исполнения. Эти проблемы детально рассматриваются в п. 3.2.1.

Алгоритм Бергеретти и Карре [34] не вычисляет межпроцедурных срезов в полном смысле этого слова, поскольку срезанию подвергается только основная программа. Более того, он не в состоянии обрабатывать рекурсивные программы. Срезы Бергеретти-Карре являются точными в том смысле, что используются точные зависимости между входными и выходными параметрами и не встречается проблема контекста вызовов.

Решения, предложенные в [75, 76, 121, 122], вычисляют точные межпроцедурные статические срезы вплоть до предположения об исполнимости путей и способны обрабатывать рекурсивные программы (см. пп. 3.2.2 и 3.2.3). Эрнст [53] и Джексон и Роллинз [78] также предлагают решения для межпроцедурных статических срезов, которые называют точными при предположении об исполнимости путей, но не представляют доказательств их точности.

Бинкли расширил алгоритм Хорвитц—Репса—Бинкли [75] в двух аспектах: решение для межпроцедурного статического среза для программы с совмещением параметров [41] и решение для получения исполняемых межпроцедурных статических срезов [42].

Неструктурированный поток управления. Лайе предложил весьма консервативный метод для вычисления статических срезов в присутствии неструктурированного потока управления (см. п. 3.3.1). Агравал [16] показал, что решения, предложенные Галлахером и Лайе [60, 61] и Хиангом и др. [81], являются некорректными. Корректные решения проблемы статических срезов для программ с неструктурированным потоком управления были предложены Боллом и Хорвитц [22, 25], Чои и Ферранте [47] и Агравалом [16] (см. п. 3.3.2). Считается, что эти три подхода являются одинаково точными.

Составные переменные и указатели. Обзор решений для срезов в присутствии составных переменных и указателей был рассмотрен в п. 3.4. Лайл [107] предложил консервативный алгоритм для статических срезов в присутствии массивов. Решение Хианга и др. [81] для построения срезов для

программ с массивами и указателями оказалось неточным. Лайе и Бинкли [108] описали подход к вычислению более точных срезов в программах с указателями, но рассматривали только программы без циклов. Агравал и др. предложили алгоритм статического среза в присутствии массивов и указателей более точный, чем алгоритм Лайе [107].

Параллелизм. Единственный алгоритм статических срезов для параллельных программ был предложен Ченгом [45] (см. п. 3.5).

3.6.3. Эффективность

Описанные ранее методы будут рассмотрены ниже с точки зрения эффективности.

Базовые алгоритмы. Алгоритм Вайзера для *внутрипроцедурных* срезов, основанный на уравнениях потока данных [139], может вычислять срезы за время $\mathbf{O}(v * (n + e))$[17], где v — количество переменных в программе, n — количество вершин управляющего графа, e — количество дуг управляющего графа.

Бергеретти и Карре [34] утверждают, что отношение μ_S для оператора S может быть вычислено за время $\mathbf{O}(v^2 * n)$. Имея μ_S, срезы для всех переменных в S можно вычислить за константное время.

Построение графа программных зависимостей включает вычисление в программе всех зависимостей по данным и по управлению. Для структурированных программ зависимости по управлению могут быть вычислены синтаксическим способом за время $\mathbf{O}(n)$. В присутствии неструктурированного потока управления зависимости по управлению в однопроцедурной программе могут быть вычислены за время $\mathbf{O}(e)$ [50, 82]. Вычисление зависимостей по данным фактически соответствует нахождению достигающих определений для каждого использования. Для скалярных переменных это может быть выполнено за время $\mathbf{O}(e * d)$, где d — количество определений в программе (см., например, [122]). Из $d \le n$ следует, что граф зависимостей по данным может быть построен за время $\mathbf{O}(e * n)$.

Одно из очевидных преимуществ методов, основанных на графах зависимостей по данным, состоит в следующем: после того как ГПЗ построен, срезы могут быть вычислены за линейное время, $\mathbf{O}(V + E)$, где V и E —

[17] Вайзер [139] приводит в качестве верхней оценки $\mathbf{O}(n*e*\log(e))$; эта оценка для количества расширенных операций над битовыми шкалами длины $\mathbf{O}(v)$. Однако современные методы позволяют осуществлять вычисление той же информации за время $\mathbf{O}(v*(n+e))$.

количество вершин и дуг *среза*, соответственно. Это бывает особенно полезно, если требуется вычислить несколько срезов для одной программы. В худшем случае, когда срез состоит из всей программы целиком, V и E соответствуют количеству вершин и дуг ГПЗ. В некоторых случаях наблюдается квадратичное возрастание количества дуг потоковой зависимости в ГПЗ, т.е. $E = \mathbf{O}(V^2)$. Нам не известно, существуют ли алгоритмы срезов, использующие более эффективные представления программ, такие как SSA-форма [21]. Однако Янг и др. [140] используют специальные графы представления программ как базис для алгоритма программной интеграции, который делает возможными сохраняющие семантику преобразования. Этот алгоритм основан на технике, близкой к срезам.

Процедуры. Далее по тексту Visible обозначает максимальное количество параметров и переменных, являющихся видимыми в пределах любой процедуры; Params — максимальное число вершин формального входа в любом графе процедурных зависимостей общего графа системных зависимостей; TotalSites — общее количество точек вызова в программе; N_p и E_p — количество вершин и дуг в управляющем графе процедуры p; $Sites_p$ — количество точек вызова в процедуре p.

Вайзер не дает оценки сложности своего алгоритма межпроцедурного среза [139]. Однако можно заметить, что для начального критерия C множество критериев в $(UP \cup DOWN)^*$ (C) содержит максимум $\mathbf{O}(Visible * Sites_p)$ критериев для каждой процедуры p. Вычисление внутрипроцедурного среза процедуры p требует времени $\mathbf{O}(Visible * (N_p + E_p))$; вычисление сводной межпроцедурной информации может быть проведено за время $\mathbf{O}(Globals * TotalSites)$ [49]. Отсюда получаем следующее выражение, представляющее верхнюю границу времени, необходимого для построения среза по всей программе:

$$\mathbf{O}(Globals * TotalSites + Visible^2 * \Sigma_p(Sites_p * (N_p + E_p))).$$

Подход Бергеретти и Карре требует, чтобы в худшем случае отношение μ вычислялось для каждой процедуры. Вызов в каждой точке заменяется наиболее видимыми присваиваниями. Следовательно, стоимость среза процедуры p равна $\mathbf{O}(Visible^2 * (n + Visible * Sites_p))$, а полная стоимость построения среза всей программы оценивается как

$$\mathbf{O}(Visible^2 * \Sigma_p(n + Visible * Sites_p)).$$

Как описывалось в п. 3.2.1, подход Хванга и др. требует времени, экспоненциального по отношению к размеру программы.

Построение индивидуального графа процедурных зависимостей в графе системных зависимостей требует времени $\mathbf{O}(\Sigma_p(E_p * N_p))$. Алгоритм Хорвитц—Репса—Бинкли [75] для вычисления сводных дуг требует времени:

$$\mathbf{O}(\text{TotalSites} * E^{PDG} * \text{Params} + \text{TotalSites} * \text{Sites}^2 * \text{Params}^4),$$

где Sites — максимальное количество точек вызова в любой процедуре, а E^{PDG} — максимальное количество дуг зависимостей по управлению и данным в любом графе процедурных зависимостей (подробнее см. [75, 121]). Подход, который предложили Репс-Хорвитц-Сагив-Росей для вычисления сводных дуг, требует времени:

$$\mathbf{O}(P * E^{PDG} * \text{Params} + \text{TotalSites} * \text{Params}^3).$$

Здесь P обозначает число процедур в программе. Поскольку количество процедур P обычно бывает намного меньшим, чем количество точек вызова TotalSites, оба слагаемых выражения сложности для алгоритма Репса—Хорвитц—Сагива—Росея являются асимптотически меньшими, чем слагаемые выражения сложности для алгоритма Хорвитц—Репса—Бинкли.

После того как граф системных зависимостей будет построен, срез можно извлечь из него (в два прохода) за время $\mathbf{O}(V + E)$, где V и E — количество вершин и дуг среза, соответственно. В худшем случае V и E равны соответственно количеству вершин и дуг графа системных зависимостей.

Бинкли не приводит оценок стоимости своего алгоритма для межпроцедурных срезов для программ с совмещением параметров [41]. Стоимость «расширения» этого алгоритма для вычисления исполняемых межпроцедурных срезов [42] из «неисполняемых» межпроцедурных срезов является линейной по отношению к размеру графа системных зависимостей.

Джексон и Роллинз [78], использующие адаптацию алгоритма Эрнста для нахождения сводных дуг зависимостей (см. п. 3.2.3), приводят оценку $\mathbf{O}(v * n^2)$, где v обозначает количество переменных, а n — количество портов в графе зависимостей. Заметим, что каждый порт в действительности является парой: (переменная, оператор). В подходе Джексона и Роллинза извлечение среза производится во время однократного обхода графа зависимостей, что требует времени $\mathbf{O}(V + E)$, где V и E обозначают количество вершин (то есть портов) и дуг среза.

Неструктурированный поток управления. Лайе [107] представляет консервативное решение для обработки неструктурированного потока управления. Его алгоритм является слегка модифицированной версией алгоритма

Вайзера для *структурированного* потока управления [139], который требует **O**(v ∗ (n + e)) времени.

Нет оценок сложности в работах [16, 25, 47]. Однако различия между приведенными там алгоритмами и «стандартным» алгоритмом срезов, основанным на графе зависимостей по данным, незначительны: Болл и Хорвитц [25] и Чои и Ферранте [47] используют несколько иной подграф зависимостей по управлению, связанный с подграфом зависимостей по данным, а Агравал [16] использует стандартный граф программных зависимостей, связанный с деревом лексических преемников, которое может быть построено за линейное время **O**(n). Таким образом, можно ожидать, что эффективность этих алгоритмов в первом приближении эквивалентна эффективности описанного выше стандартного алгоритма.

Составные переменные и указатели. Подход Лайе к срезам программ с массивами [107] дает ту же оценку сложности, что и алгоритм Вайзера для срезов программ со скалярными переменными, поскольку длина путей достигающих определений в худшем случае остается той же самой.

Стоимость построения графов программных зависимостей для программ с составными переменными и ссылками при использовании алгоритма Агравала и др. [17] такая же, как при построении графов программных зависимостей для программ с одними только скалярными переменными, поскольку длина путей (потенциальных) достигающих определений в худшем случае остается той же самой, а определение возможных пересечений и частичных пересечений (см. п. 3.4) может быть выполнено за константное время.

Лайе и Бинкли не приводят оценок сложности для своего подхода к срезам для программ с указателями [108].

Нужно заметить, что можно получить более точные статические срезы в присутствии нескалярных переменных, если провести более совершенный (но более дорогой) анализ зависимостей по данным (см., например, [110, 141]).

Параллелизм. Ченг [45] также не приводит оценок сложности вычисления зависимостей по выбору, по синхронизации и по коммуникации. Для извлечения среза требуется время **O**(V+E), где V и E обозначают количество вершин и дуг сети программных зависимостей, соответственно.

3.6.4. Комбинирование алгоритмов статических срезов

В табл. 3 (см. [140]) представлены «ортогональные» направления, усложняющие задачу построения статических срезов: процедуры, неструкту-

рированной поток управления, нескалярные переменные и параллелизм. Для каждого вычислительного метода в таблице указаны работы, предлагающие решения этих проблем. В принципе, решения для различных проблем могут быть скомбинированы, если они появляются в одной и той же строке табл. 3 (т.е., если они относятся к одному и тому же вычислительному методу).

Таблица 3

Ортогональные направления усложнения задачи статических срезов

Основа алгоритма	*Процедуры*	*Неструктур. поток управления*	*Составные переменные*	*Параллелизм*
Уравнения потока данных	Вайзер [106, 139]	Лайле[107]	Лайл [107]	-
Отношения информац. потока	Бергеретти, Карре [34]	-	-	-
Графы программных зависимостей	Хорвитц и др[75] Лахотия [100] Репс и др.[121, 122] Бинкли [42]	Болл, Хорвитц [22, 25] Чои, Ферранте [47] Агравал [16]	Агравал и др. [17] (*)	Ченг [45]

(*) Алгоритмы для вычисления консервативных приближений зависимостей по данным в присутствии совмещения имен могут быть использованы здесь (см. п. 3.4).

4. МЕТОДЫ ДИНАМИЧЕСКИХ СРЕЗОВ

4.1. Базовые алгоритмы

В этом разделе рассматриваются базовые алгоритмы построения динамических срезов структурированных однопроцедурных программ со скалярными переменными.

4.1.1. Динамические представления потока

Корел и Ласки [96, 97] описывают, каким образом могут быть вычислены динамические срезы. Они формализуют историю исполнения программы как *траекторию*, состоящую из последовательности «вхождений» операторов и выражений (управляющих предикатов). Метки служат для различения разных вхождений одного оператора в историю исполнения. На рис. 16 представлена траектория для программы на рис. 2, *а* для входного значения n = 2.

Критерий динамического среза представляет собой триплет (x, I^q, V), где x обозначает входное значение программы, вхождение оператора I^q является q-м элементом траектории, а V — подмножеством переменных программы[18]. Корел и Ласки определяют *динамический срез* относительно критерия (x, I^q, V) как *исполняемую* программу S, полученную из программы P путем удаления из нее нуля или большего числа операторов. Три ограничения налагаются на S. Во-первых, при исполнении с входным значением x траектория S идентична траектории P, если удалить из нее все операторы, не появляющиеся в S. Во-вторых, и программой, и ее срезом вычисляются идентичные значения для всех переменных из V в точке вхождения оператора, обозначенной в критерии. В-третьих, требуется, чтобы оператор I, соответствующий обозначенному в критерии экземпляру оператора I^q, входил в S. Корел и Ласки заметили, что введенное ими понятие динамического среза обладает следующим свойством: если цикл встречается в срезе, он проходится столько же раз, сколько и в исходной программе.

[18] Определение критерия динамического среза, данное Корелом и Ласки, Тир [130] считает несколько несогласованным (оно предполагает, что траектория заранее известна, хотя она неоднозначно определяется входным значением x) и предлагает определять критерий динамического среза как триплет (x, q, V), где q — количество вхождений оператора в траекторию, индуцированную входным значением x.

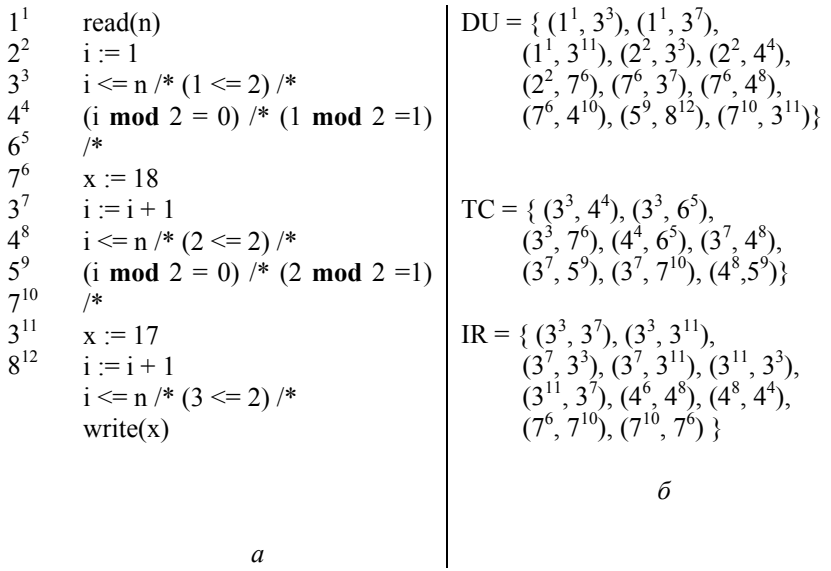


Рис. 16. Траектория программы рис. 2, *а* для входного значения n = 2 (*а*); динамические представления потока для этой траектории (*б*)

Для того чтобы вычислить динамические срезы, Корел и Ласки вводят три *динамических представления потока*, формализующие зависимости между вхождениями операторов в траектории. Отношение DU (Definition-Use) связывает использование переменной с последним ее определением. Заметим, что для конкретной траектории это отношение определяется единственным образом. Отношение TC (Test-Control) связывает самое последнее вхождение управляющего предиката с вхождениями операторов траектории, зависимыми по управлению от него. Это отношение определяется синтаксически только для структурированных конструкций программы. Вхождения одного и того же оператора составляют отношение IR (Identity). На рис. 16, *б* приведены динамические представления потока для траектории на рис. 16, *а*.

Динамические срезы вычисляются итеративным образом, путем определения последовательных множеств S^i, состоящих из прямо и косвенно подходящих операторов. Для критерия среза (x, I^q, V) начальное приближение S^0 содержит последние определения переменных из V на траектории перед вхождением экземпляра оператора I^q, а также тестовые операции на траек-

тории, от которых I^q зависим по управлению. Приближение S^{i+1} определяется следующим образом:

$$S^{i+1} = S^i \cup A^{i+1},$$

где A^{i+1} определяется по следующему правилу:

$$A^{i+1} = \{X^p \mid X^p \notin S^i, (X^p, Y^t) \in (DU \cup TC \cup IR) \text{ для некоторого } Y^t \in S^i, p < q\},$$

где q — «метка» вхождения оператора, обозначенного в критерии среза. Динамический срез является неподвижной точкой S_C этого итеративного процесса (поскольку q конечно, она всегда существует): любой оператор X, экземпляр X^p которого входит в S_C, будет принадлежать срезу. Затем оператор I, соответствующий I^q, добавляется в срез.

Рассмотрим в качестве примера вычисление динамического среза для траектории на рис. 16 и критерия (n=2, 8^{12}, {x}). Поскольку последний оператор не зависит по управлению от любого другого оператора, начальное приближение среза состоит из последнего определения x: $A^0 = \{5^9\}$. В результате последующих итераций получим:

$$A^1 = \{3^7, 4^8\}, A^2 = \{7^6, 1^1, 3^3, 3^{11}, 4^4\}, A^3 = \{2^2, 7^{10}\}.$$

Из чего следует: $S_C = \{1^1, 2^2, 3^3, 4^4, 7^6, 3^7, 4^8, 5^9, 7^{10}, 3^{11}, 8^{12}\}$.

Таким образом, динамический срез относительно критерия (n=2, 8^{12}, {x}) включает все операторы, кроме оператора 5, соответствующего оператору 6^5 траектории. Это срез был представлен ранее на рис. 2, *б*.

Роль IR-отношения нуждается в некотором пояснении. Рассмотрим траекторию программы, приведенной на рис. 2, *а* для входного значения n = 1. Траектория представлена на рис. 17, *а*; динамические представления потока для этой траектории — на рис. 17, *б*; срез относительно критерия (n = 1, 8^8, {x}) — на рис. 17, *в*. Заметим, что срез, полученный таким способом, является завершающейся программой: поскольку экземпляр оператора 3^3 включен в срез, IR-отношение вызовет включение также 3^7, которое, в свою очередь, приведет к включению в срез 7^6 в силу DU-отношения. Однако, вычисляя срез и игнорируя при этом IR-отношение, мы получим незавершающуюся программу, показанную на рис. 17, *г*. Причиной этого феномена (и, соответственно, введения IR-отношения) является тот факт, что отношения DU и TC обходят траекторию только в *обратном* направлении. Целью введения IR-отношения является как обход траектории в *обоих* направлениях, так и включение в срез всех операторов и управляющих предикатов, необходимых для обеспечения завершения всех циклов среза. К сожалению, не приводится доказательства того, что эта мера всегда является достаточной.

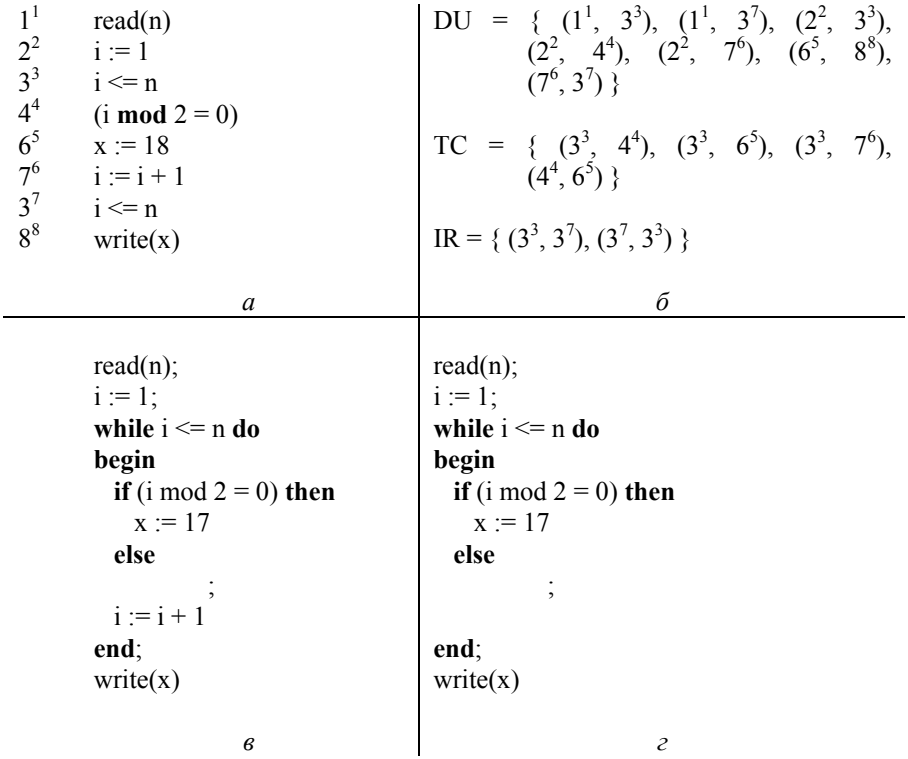


Рис. 17. Траектория программы рис. 2, *а* для входного значения n = 1 (*а*); динамические представления потока для этой траектории (*б*); динамический срез для критерия (n=1, 8^8, {x}) (*в*); незавершающийся срез относительно того же самого критерия, получаемый при игнорировании IR-отношения (*г*)

К сожалению, обход IR-отношения в «обратном» направлении приводит к включению в срез операторов, которые не являются необходимыми для завершения. Например, на рис. 18, *а* представлена чуть модифицированная версия программы рис. 2, *а*; на рис. 18, *б* представлена траектория для этой программы. Из этой траектории видно, что (7^6, 7^{11}) ∈ IR, (6^5, 7^6) ∈ DU, (5^{10}, 7^{11}) ∈ DU. Следовательно, оба оператора (5) и (6) будут включены в срез относительно критерия (n=2, 9^{14}, {z}), хотя оператор (6) не требуется ни для вычисления последнего значения z, ни для сохранения завершения.

```
(1)     read(n);
(2)     i := 1;
(3)     while i <= n do
        begin
(4)       if (i mod 2 = 0) then
(5)         x := 17
          else
(6)         x := 18;
(7)       z := x;
(8)       i := i + 1
        end;
(9)     write(z)
```

а

```
1^1     read(n)
2^2     i := 1
3^3     i <= n
4^4     (i mod 2 = 0)
6^5     x := 18
7^6     z := x
8^7     i := i + 1
3^8     i <= n
4^9     (i mod 2 = 0)
5^10    x := 17
7^11    z := x
8^12    i := i + 1
3^13    i <= n
9^14    write(z)
```

б

Рис. 18. Исходная программа (*а*);
траектория программы для входного значения n = 2 (*б*)

```
(1)     read(n);
(2)     x := 0;
(3)     i := 1;
(4)     while i < n do
        begin
(5)       x := 1;
(6)       y := 10/x;
(7)       i := i + 1
        end
```

а

```
1^1     read(n)
2^2     x := 0
3^3     i := x
4^4     i < n
5^5     x := 1
6^6     y := 10/x
7^7     i := i + 1
4^8     i < n
5^9     x := 1
6^10    y := 10/x
7^11    i := i + 1
4^12    i < n
```

б

```
read(n)
x := 0;
i := x;
while i < n do
begin
              ;
  y := 10/x;
  i := i + 1
end
```

в

Рис. 19. Исходная программа (*а*);
траектория программы для входного значения n = 2 (*б*);
динамический срез относительно критерия (n=2, 6^{10}, {i}), который приводит к ошибке во время исполнения, когда на вход подается то же значение (*в*)

Предполагается, что если ограничить IR-отношение только теми экземплярами операторов, которые соответствуют управляющим предикатам программы, то можно получить срезы меньшего размера. С другой стороны, любопытно было бы исследовать вопрос, нельзя ли вместо IR-отношения использовать динамическую вариацию понятия слабой зависимости по управлению, введенного Подгурски и Кларком [117].

Завершая изложение алгоритма Корела—Ласки, следует обратить внимание на небольшую проблему[19]. Рассмотрим пример программы рис. 19, *а*, траекторию этой программы на рис. 19, *б* и динамический срез относительно критерия (n = 2, 6^{10}, {i}) рис. 19, *в*. Мы видим, что оператор x := 1 исключен из среза, так что при исполнении с входным значением n = 2 срез приводит к ошибке — делению на нуль. Эта проблема возникает из-за того, что критерий среза I^q добавляется к последнему приближению S_C среза, при этом в срез не обязательно добавляются операторы, от которых он зависит. По мнению Типа [130] указанная проблема может быть решена довольно просто — инициализацией S_0 с критерием среза I^q вместо последнего определения переменных из V.

4.1.2. Динамические отношения зависимостей

Гопал [62] описывает подход к вычислению динамических срезов, использующий динамические отношения зависимостей. Он вводит динамические версии используемых Бергеретти и Карре отношений информационного потока[20] λ_S, μ_S и ρ_S (см. п. 3.1.2). Отношение $\dot{\lambda}_S$ содержит все такие пары (v, e), что оператор e зависит от входного значения v во время исполнения программы. Отношение $\dot{\mu}_S$ содержит все такие пары (e, v), что выходное значение v зависит от исполнения оператора v. Пара (v, v′) принадлежит отношению $\dot{\rho}_S$, если выходное значение v′ зависит от входного значения v. В этих определениях предполагается, что S исполняется при некотором фиксированном входном значении.

Для пустых операторов, операторов присваиваний и составных операторов отношения зависимостей Гопала в точности совпадают с отношениями для статического случая. Статические отношения информационного потока для условного оператора выводятся из самого оператора и из операторов, составляющих его ветви. Однако для вычисления *динамических* отношений

[19] Этот момент был впервые замечен анонимным рецензентом статьи [130].

[20] Гопал использует в своей работе обозначения s^S_v, v^S_v и v^S_s. Для того чтобы избежать путаницы и сделать явной связь с работой Бергеретти и Карре (см. п. 3.1.2), мы используем обозначения $\dot{\lambda}_S$, $\dot{\mu}_S$ и $\dot{\rho}_S$, соответственно.

зависимостей в рассмотрение берутся только те зависимости, которые появляются на ветви, реально выбираемой во время исполнения программы. Так же, как в [34], отношение зависимости для оператора **while** (которое мы здесь не приводим) выражено в терминах отношений зависимостей для вложенных условных операторов. Однако если в статическом случае циклы эквивалентно заменяются их бесконечными развертками, в динамическом случае требуется развернуть цикл ровно столько раз, сколько раз цикл реально исполняется. Найденные определения получаются свернутыми, так как отношения зависимостей для тела цикла могут меняться на каждой итерации. Следовательно, простая операция транзитивного замыкания, такая как для статического случая, была бы в этом случае недостаточной.

На рис. 20 сведены воедино динамические отношения зависимостей Гопала. Здесь DEFS(S) обозначает множество переменных, которые подвергаются изменению при исполнении оператора S. Заметим, что определение DEFS является «динамическим» в том смысле, что оно принимает во внимание, какая ветвь оператора **if** реально исполняется. Определив указанные отношения, мы можем теперь определить динамический срез относительно последнего значения переменной v как $\sigma^{P}_{v} \equiv \{e \mid (e,v) \in \mu^{-}_{P}\}$.

$\lambda^{-}_{\varepsilon} = \varnothing$ $\mu^{-}_{\varepsilon} = \varnothing$ $\rho^{-}_{\varepsilon} = ID$	$\lambda^{-}_{S1;S2} = \lambda^{-}_{S1} \cup \rho^{-}_{S1} \cdot \lambda^{-}_{S2}$ $\mu^{-}_{S1;S2} = \mu^{-}_{S1} \cdot \rho^{-}_{S2} \cup \mu^{-}_{S2}$ $\rho^{-}_{S1;S2} = \rho^{-}_{S1} \cdot \rho^{-}_{S2}$	$\lambda^{-}_{v:=e} = VARS(e) \times \{e\}$ $\mu^{-}_{v:=e} = \{(e, v)\}$ $\rho^{-}_{v=e} = (VARS(e) \times \{v\}) \cup$ (ID - (v, v))

$\lambda^{-}_{\textbf{if e then } S1 \textbf{ else } S2} = (VARS(e) \times \{e\}) \cup \lambda^{-}_{S1}$, если e оказывается true;
$\lambda^{-}_{\textbf{if e then } S1 \textbf{ else } S2} = (VARS(e) \times \{e\}) \cup \lambda^{-}_{S2}$, если e оказывается false;
$\mu^{-}_{\textbf{if e then } S1 \textbf{ else } S2} = (\{e\} \times DEFS(S_1)) \cup \mu^{-}_{S1}$, если e оказывается true;
$\mu^{-}_{\textbf{if e then } S1 \textbf{ else } S2} = (\{e\} \times DEFS(S_2)) \cup \mu^{-}_{S2}$, если e оказывается false;
$\rho^{-}_{\textbf{if e then } S1 \textbf{ else } S2} = (VARS(e) \times (DEFS(S_1)) \cup \rho^{-}_{S1}$, если e оказывается true;
$\rho^{-}_{\textbf{if e then } S1 \textbf{ else } S2} = (VARS(e) \times DEFS(S_2)) \cup \rho^{-}_{S2}$, если e оказывается false.

Рис. 20. Определение динамических отношений зависимостей

На рис. 21, *а* показано отношение μ^- для всей программы рис. 2, *а*[21]. Из этого отношения следует, что множество выражений, влияющих на значение x в конце программы для входного значения n = 2, составляет {1, 2, 3, 4, 5, 7}. Соответствующий динамический срез почти идентичен срезу, приведенному на рис. 1, *б*, за тем исключением, что алгоритм Гопала не включает в срез последний оператор write(x) из строки 8.

В некоторых случаях алгоритм Гопала может выдавать незавершающиеся срезы завершающихся программ. На рис. 22, *а* представлен срез для программы на рис. 2, *а* и входного значения n = 1, вычисленный в соответствии с алгоритмом Гопала.

1	{i, n, x}
2	{i, x}
3	{i, x}
4	{i, x}
5	{x}
6	∅
7	{i, x}
8	∅

Рис. 21. Отношение μ^- для программы на рис. 2, *а* и входного значения n = 2. Выражения обозначены номерами строк на рис. 2, *а*

Преимущество использования отношений зависимостей состоит в том, что в определенных случаях алгоритм Гопала вычисляет срезы меньшего размера, чем алгоритм Корела и Ласки. Например, на рис. 22, *б* показан срез относительно последнего значения z программы на рис. 18, *а* для входного значения n = 2. Заметим, что присваивание x := 18, которое входит в срез, вычисленный алгоритмом из п. 4.1.1, не включено в срез, вычисленный алгоритмом Гопала.

[21] Гопал не определяет отношения информационного потока для операторов ввода/вывода. Для удобства изложения нашего примера мы считаем, что оператор read(n) рассматривается как присваивание n := *НекотораяКонстанта*, а операторы write(sum) и write(product) рассматриваются как пустые.

```
read(n);
i := 1;
while i <= n do
begin
 if (i mod 2 = 0) then

 else
 x := 18;

end
```

а

```
read(n);
i := 1;
while i <= n do
begin
 if (i mod 2 = 0) then
 x := 17
 else
 ;
z := x;
i := i + 1
end
```

б

Рис. 22. Незавершающийся срез для программы на рис. 2, *а* относительно последнего значения x для входного значения n = 1 (*а*); срез для программы на рис. 18, *а* относительно последнего значения x и входного значения n = 2 (*б*)

4.1.3. Графы зависимостей

Миллер и Чои [111] впервые ввели динамическую вариацию ГПЗ, назвав ее *динамическим графом программных зависимостей*. Разработанный ими параллельный отладчик программ использует упомянутые графы для проведения *обратного потокового анализа* [26], наращивая их по мере необходимости. Перед исполнением строится вариация статического ГПЗ, а объектный код программы дополняется кодом, генерирующим лог-файл. Вдобавок к этому генерируется пакет эмулятора. Программы разбиваются на так называемые блоки эмуляции (обычно это процедуры). Во время отладки отладчик, используя лог-файл, статический ГПЗ и пакет эмуляции, перевычисляет блок эмуляции и получает информацию, необходимую для построения части динамического ГПЗ, соответствующей этому блоку. В случае, когда пользователь желает применить обратный потоковый анализ к еще не построенным частям графа, перевычисляется больше блоков эмуляции.

Агравал и Хорган [19] разработали подход, использующий графы зависимостей для вычисления динамических срезов. Первые два их алгоритма являются неточными, но они полезны для понимания окончательного вари-

анта. Самый первый вариант использует ГПЗ, описанный в п. 3.1.3[22], и помечает вершины, которые исполняются на заданном наборе входов. Динамический срез определяется путем вычисления статического среза в подграфе ГПЗ, образованном помеченными вершинами. По построению полученный срез содержит только те вершины, которые были исполнены. Это решение неточное, так как оно не распознает ситуации, когда существует дуга потоковой зависимости между помеченными вершинами v_1 и v_2, но определения v_1 реально не используются в v_2.

На рис. 23, *а* представлен ГПЗ для программы на рис. 2, *а*. Предположим, что нам нужно вычислить срез относительно последнего значения x при входном значении n = 2. Все вершины ГПЗ исполняются, следовательно, все они будут помечены. Статический алгоритм среза (п. 3.1.3), таким образом, выдаст в качестве среза всю программу целиком, хотя присваивание x := 18 является неподходящим. Оно будет включено в срез, поскольку существует дуга зависимости от вершины x := 18 к вершине write(x), несмотря на то, что эта дуга представляет зависимость, не проявляющуюся на второй итерации цикла. Говоря более точно, эта зависимость реализуется только на тех итерациях цикла, где управляющая переменная i имеет нечетное значение.

Второй подход состоит в том, что помечаются *дуги* ГПЗ, когда соответствующие им зависимости реализуются во время исполнения. Срез опять получается при помощи обхода ГПЗ, но в этот раз обход совершается только по помеченным дугам. К сожалению, данный алгоритм также дает неточные срезы в присутствии циклов, так как дуга, помеченная на некоторой итерации, будет присутствовать на всех последующих итерациях независимо от того, реализуется ли эта зависимость на более поздних итерациях или нет. На рис. 23, *б* представлен ГПЗ для программы рис. 18, *а*. Для входного значения n = 2 будут помечены все дуги зависимостей, в силу чего срез будет состоять из всей программы целиком. В работе [19] показано, что потенциальное усовершенствование второго подхода, состоящее в *снятии пометок* с дуг, помеченных на предыдущих итерациях, не будет корректным.

[22] Вариант графа зависимостей в [19] не имеет входной вершины. Ее отсутствие не сказывается на окончательном виде среза. Для удобства изложения все графы, изображенные на рисунках в нашей работе, имеют входную вершину.

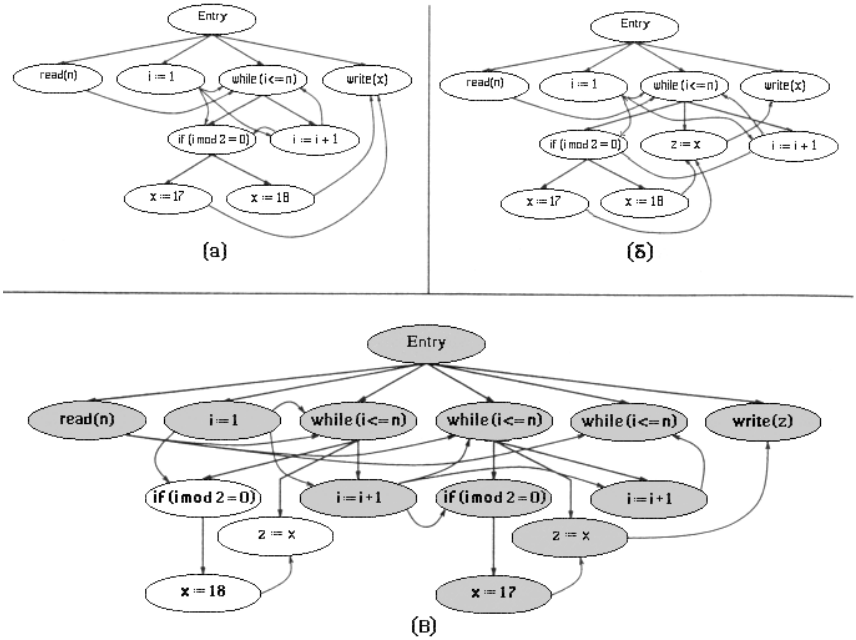


Рис. 23. Граф программных зависимостей для программы рис. 2, *а* (*а*); ГПЗ для программы рис. 18, *а* (*б*); динамический граф зависимостей для программы рис. 18, *а* (*в*)

Агравал и Хорган обращают внимание на тот интересный факт, что их второй подход дает в итоге срезы, идентичные[23] полученным в результате действия алгоритма Корела и Ласки (см. п. 4.1.1). Однако ГПЗ программы (с возможно помеченными дугами) требует только $\mathbf{O}(n^2)$ памяти (n обозначает количество операторов программы), тогда как траектории Корела и Ласки требуют **O**(N) памяти, где N обозначает количество исполненных операторов.

Второй подход Агравала и Хоргана вычисляет чересчур большие срезы, поскольку он не принимает в расчет тот факт, что разные вхождения опера-

[23] В действительности имеются некоторые расхождения из-за того, что вычисленные алгоритмом Корела—Ласки срезы могут привести к ошибкам времени исполнения. Это было предметом обсуждения в п. 4.1.1.

тора в истории исполнения могут быть (транзитивно) зависимы от разных операторов. Это наблюдение привело к разработке третьего подхода: созданию отдельной вершины в графе зависимостей для каждого вхождения оператора в истории исполнения. Этот вид графа был назван *динамическим графом зависимостей* (ДГЗ). Критерий динамического среза идентифицируется с вершиной ДГЗ, и динамический срез вычисляется путем определения всех вершин ДГЗ, из которых критерий может быть достигнут. Оператор или управляющий предикат включается в срез в том случае, когда критерий может быть достигнут по меньшей мере из одной вершины, соответствующей вхождению этого оператора или предиката.

На рис. 23, *в* представлен ДГЗ для программы рис. 18, *а*. Критерий среза соответствует вершине write(z), и все вершины, из которых он может быть достигнут, заштрихованы. Заметим, что критерий нельзя достичь из вершины x := 18. Следовательно, соответствующее присваивание не принадлежит срезу.

Недостатком ДГЗ является то, что количество вершин в графе равно количеству исполненных операторов. Однако количество динамических срезов в худшем случае равно $\mathbf{O}(2^n)$, где n — количество операторов в обрабатываемой программе. На рис. 24 представлена программа Q^n, для которой имеется $\mathbf{O}(2^n)$ динамических срезов. Программа читает количество значений в переменных x_i ($1 \le i \le n$) и позволяет вычислить сумму $\Sigma_{x \in S} x$ для любого количества подмножеств $S \subseteq \{x_1, ..., x_n\}$. Важнейшее наблюдение заключается в том, что на каждой итерации внешнего цикла срез относительно оператора write(y) будет содержать в точности операторы read(x_i) для $x_i \in S$. Поскольку множество с n элементами имеет 2^n различных подмножеств, программа Q^n имеет $\mathbf{O}(2^n)$ различных динамических срезов.

Агравал и Хорган [19] предложили сокращать количество вершин ДГЗ путем слияния вершин, у которых транзитивные зависимости покрывают одно и то же множество операторов. Другими словами, новая вершина вводится только в том случае, когда она вызывает появление нового динамического среза. Очевидно, что такая проверка вызывает некоторое увеличение издержек времени исполнения. Полученный граф называется *сокращенным динамическим графом зависимостей* (СДГЗ) программы. Срезы, вычисленные с помощью СДГЗ, имеют ту же точность, что и вычисленные с помощью ДГЗ.

В ДГЗ рис. 23, *в* вершины, представляющие оператор i := i + 1, и две правые вершины, представляющие оператор i <= n, имеют одни и те же транзитивные зависимости; эти вершины зависят от операторов 1, 2, 3 и 8 программы на рис. 18, *а*. Таким образом, СДГЗ для этой программы (для

входного значения n = 2) получается путем слияния указанных четырех вершин ДГЗ в одну.

```
program Q^n;
begin
    read(x_1);
    ...
    read(x_n);
    MoreSubsets := true;
    while MoreSubsets do
        begin
          Finished := false;
          y := 0;
          while not (Finished) do
              begin
              read(i);
              case (i) of
                  1: y := y + x_1;
                  ...
                  i: y:=y+ x_i;
                  ...
                  n: y := y + x_n;
              end
              read(Finished);
              end;
          write(y);
          read(MoreSubsets);
        end
end.
```

Рис. 24. Программа Q^n, имеющая $\mathbf{O}(2^n)$ различных динамических срезов

Агравал и Хорган [19] представляют алгоритм построения СДГЗ, не опирающийся на полную историю исполнения. Для построения требуется следующая информация:

1) для каждой переменной — вершина, соответствующая ее последнему определению;

2) для каждого предиката — вершина, соответствующая его последнему исполнению;

3) для каждой вершины СДГЗ — динамический срез относительно этой вершины.

4.2. Процедуры

Агравал и др. [17] рассматривают динамические срезы для процедур с передачей параметров по значению, по ссылке, по результату и по значению-результату. Главной особенностью этого метода является то, что динамические зависимости по данным определяются в терминах определений и использований *областей памяти*; этот способ хорош тем, что глобальные переменные не требуют специальной обработки и не нужно проводить проверку совмещения имен. Агравал и др. описывают, каким образом каждый из механизмов передачи параметров может быть смоделирован с помощью набора фиктивных присваиваний, которые вставляются до и/или после каждого вызова процедуры. Далее по тексту мы предполагаем, что процедура P с формальными параметрами f_1, ..., f_n вызывается с фактическими параметрами a_1, ..., a_n. Передача параметров *по значению* может быть смоделирована последовательностью присваиваний $f_1 := a_1$; ... $f_n := a_n$, которая исполняется перед входом в процедуру. Чтобы определить местонахождение ячеек памяти для корректной активации записи, множества USE для фактических параметров a_i определяются *до* входа в процедуру, а множества DEF для формальных параметров f_i определяются *после* входа в процедуру. Для передачи параметров *по значению-результату* необходимо произвести дополнительные присваивания формальных параметров фактическим после выхода из процедуры. Передача параметров *по ссылке* не требует дополнительных действий, так как одна и та же ячейка памяти ассоциирована с соответствующими формальным и фактическим параметрами a_i и f_i.

Альтернативный подход к динамическим межпроцедурным срезам был предложен Камкаром и др. [85, 87]. От алгоритма Агравала и др. его отличает тот факт, что авторы в основном концентрировались на срезах процедурного уровня. Более точно, они исследовали проблему определения множества *точек вызова* в программе, влияющих на значение переменной в конкретной точке вызова.

Во время исполнения строится *(динамический) сводный граф зависимостей*. Вершины этого графа, называемые *экземплярами процедур*, соответ-

ствуют активациям процедуры, аннотированным их параметрами[24]. Дуги сводного графа представляют собой либо дуги активации, соответствующие вызовам процедур, либо сводные дуги зависимостей. Второй тип отражает транзитивные зависимости по управлению и данным между входными и выходными параметрами экземпляров процедуры.

Критерий среза определяется как пара, содержащая экземпляр процедуры и входной или выходной параметр связанной с ней процедуры. После построения сводного графа срез относительно критерия определяется в два этапа. Вначале находятся те части сводного графа, из которых достижим критерий; полученный подграф обозначается как *рабочий срез*. Вершины рабочего среза представляют собой *частичные* экземпляры процедур, так как некоторые параметры могли быть «*отрезаны*». Межпроцедурный срез программы состоит из всех точек вызова в программе, для которых частичный экземпляр входит в состав рабочего среза.

Рассматриваются три подхода к построению сводных графов. В первом случае *внутрипроцедурные* зависимости по данным вычисляются статически: в итоге могут получиться неточные срезы в присутствии условных переходов. Во втором случае все зависимости определяются во время исполнения. Хотя в итоге получаются более точные динамические срезы, зависимости для процедуры P приходится перевычислять при каждом вызове процедуры P. Третий подход пытается объединить эффективность «статического» подхода с точностью «динамического», вычисляя зависимости внутри линейных участков статически, а зависимости между участками — динамически. Во всех трех подходах зависимости по управлению[25] вычисляются статически. Неясно, насколько хорош третий подход в присутствии составных переменных и указателей, когда внутри лучей зависимости времени исполнения не могут быть вычислены статически: во время исполнения приходится проводить дополнительную проверку на совмещение имен.

Камкар [83] приспосабливает метод межпроцедурных срезов, разработанный им ранее (Камкар и др. [85, 87]), для вычисления межпроцедурных срезов операторного уровня (т.е. срезов, состоящих из множества операторов, а не множества точек вызова). Вкратце, это достигается введением

[24] Более точно, Камкар говорит о входных и выходных переменных процедуры. Это понятие также применяется к глобальным переменным, которые используются или определяются процедурой.

[25] Камкар и др. используют понятие зависимостей по управлению, *сохраняющих завершаемость программы*; оно сходно с введенным Подгурски и Кларком понятием *слабой* зависимости по управлению [117].

вершины для каждого *экземпляра оператора* (вместо экземпляра процедуры) в сводный граф. Для построения сводного графа могут быть использованы все те же три подхода (статический, динамический и комбинированный).

Чои и др. [48] предложили подход к выполнению обратного межпроцедурного потокового анализа. Вначале предполагается, что вызов процедуры может изменять значение любой глобальной переменной; для этого статический ГПЗ дополняется *дугами связей*, обозначающими *потенциальные* зависимости по данным. На втором этапе сводная межпроцедурная информация используется для того, чтобы либо заменить дугу связи дугой зависимости по данным, либо удалить ее из графа. Некоторые дуги связи могут оставаться в графе; они будут распознаны во время исполнения.

4.3. Составные типы данных и указатели

4.3.1. Составные типы данных и указатели

Корел и Ласки [97] производят срезы в присутствии составных переменных, рассматривая каждый элемент массива или каждое поле записи как отдельную переменную. Динамические структуры данных обрабатываются как *два* отдельных объекта, а именно — сам указатель и объект, на который тот указывает. Для динамически размещаемых объектов они предложили решение присваивать уникальное имя каждому объекту.

4.3.2. Графы зависимостей

Агравал и др. [17] представляют алгоритм для динамических срезов в присутствии составных типов данных и указателей, основанный на графах зависимостей. Их решение состоит в выражении множеств DEF и USE в терминах *фактических областей памяти*, предоставляемых компилятором. Алгоритм, описанный в [17], похож на алгоритм *статического* среза в присутствии составных типов данных и указателей тех же авторов (см. п. 3.4). Однако во время вычисления *динамических* достигающих определений не могут возникать возможные пересечения — только полные и частичные пересечения.

Чои и др. [48] расширяют метод обратного потокового анализа Миллера и Чои [111] (см. п. 4.1.3) для того, чтобы работать с массивами и указателями. Для массивов они добавляют к своим статическим ГПЗ *дуги связи*; эти дуги отображают *потенциальные* зависимости по данным, а во время ис-

полнения они либо удаляются, либо заменяются реальными дугами зависимостей. Переходы по ссылкам рассматриваются во время исполнения с записью всех случаев использования указателей в лог-файл.

4.4. Параллельность

4.4.1. Динамические представления потока

Корел и Фергюсон [95] расширяют метод динамических срезов Корела и Ласки [96, 97] на распределенные программы с Ада-подобными рандеву-коммуникациями (см., например, [28]). Для распределенной программы история исполнения формализуется в виде *пути распределенной программы*, который для каждой задачи содержит, во-первых, последовательность исполняемых ею операторов (траекторию) и, во-вторых, последовательность триплетов A, C, B, определяющих каждое рандеву, в которое вовлечено задание. Здесь A обозначает оператор **accept** задания, B обозначает другое задание, участвующее в коммуникации, а C — входной оператор вызова в задании, вовлеченном в рандеву.

Критерий динамического среза распределенной программы задает: (1) входные значения каждого задания; (2) путь распределенной программы P; (3) задание w; (4) вхождение оператора q в траектории w; (5) переменную v. Динамический срез относительно такого критерия является исполняемой проекцией программы, полученной путем удаления из нее операторов. Однако программа гарантированно сохраняет свое изначальное поведение только в том случае, если рандеву появляются в срезе в том же относительном порядке, что и в исходной программе. (Заметим, что не все рандеву программы должны входить в срез).

Метод Корела-Фергюсона для вычисления срезов распределенных программ [95] по существу является обобщением метода Корела-Ласки [96, 97], хотя и изложен в несколько иной манере. Дополнительно к описанным ранее динамическим представлениям потока (см. п. 4.1.1), вводится понятие *влияния коммуникации*, отражающее взаимозависимости между заданиями. Авторы также представляют распределенную версию своего алгоритма, которая задействует отдельный процесс построения среза для каждого задания.

4.4.2. Графы зависимостей

Дюстервальд и др. [51] представляют алгоритм вычисления динамических срезов распределенных программ, основанный на графах зависимо-

стей. Они вводят *распределенный граф зависимостей* (РГЗ) для представления распределенных программ.

Распределенная программа P состоит из множества процессов P_1, ..., P_n. Коммуникация между процессами предполагается синхронной и недетерминированной и производится посредством операторов **send** и **receive**. *Критерий динамического среза распределенной программы* (I_1, X_1), ..., (I_n, X_n) задает для каждого процесса P_i его входное значение I_i и набор операторов X_i. *Распределенный динамический срез* S представляет собой исполняемый набор процессов P'_1, ..., P'_n такой, что операторы P'_i являются подмножеством операторов P_i. Срез S вычисляет те же значения на операторах каждого X_i, что и программа S, при тех же входных значениях. Это достигается посредством того, что *все* входные операторы включаются в срез, а недетерминированные операторы коммуникации в программе заменяются детерминированными операторами коммуникации в срезе.

РГЗ содержит одну вершину для каждого оператора и управляющего предиката в программе. Зависимости по управлению между операторами определяются статически, перед исполнением. Дуги зависимостей по данным и по коммуникации добавляются к графу во время исполнения. Срезы вычисляются обычным образом, путем определения множества вершин РГЗ, из которых могут быть достигнуты вершины, обозначенные в критерии. И построение РГЗ, и вычисление срезов проводятся распределенным образом; отдельный процесс построения РГЗ и процесс срезания присваиваются каждому процессу P_i программы; эти процессы связываются друг с другом, когда встречается оператор **send** или **receive**.

В силу того, что одна вершина РГЗ представляет все вхождения оператора в истории исполнения, неточные срезы могут вычисляться в присутствии циклов (см. п. 4.1.1). Например, срез относительно последнего значения z для программы на рис. 18 при входном значении n = 2 будет представлять собой всю программу целиком.

Ченг [45] представляет альтернативный алгоритм вычисления динамических срезов распределенных и параллельных программ, также основанный на графах зависимостей. Представление параллельной программы в виде сети программных зависимостей (см. п. 3.5) используется для вычисления динамических срезов. Алгоритм Ченга является, по существу, обобщением первоначального подхода, предложенного Агравалом и Хорганом [19]: вершины СПЗ, соответствующие исполненным операторам, помечаются, и статический алгоритм построения срезов из разд. 3.5 применяется к подграфу СПЗ, индуцированному помеченными вершинами. Как говорилось в п. 4.1.3, в результате получаются неточные срезы.

Чои и др. [48, 111] описывают, каким образом их подход к обратному потоковому анализу может быть расширен на параллельные программы. Рассматриваются разделяемые переменные с семафорами, коммуникации с передачей сообщений и Ада-подобный механизм рандеву. С этой целью вводится параллельный динамический граф, который содержит вершины синхронизации для операций синхронизации (таких, как P и V над семафором) и дуги синхронизации, которые представляют зависимости между параллельными процессами. Чои и др. поясняют, каким образом с помощью анализа параллельного динамического графа могут быть обнаружены конфликты чтения/записи и записи/записи между параллельными процессами.

4.5. Сравнение

В этом разделе сравниваются и классифицируются методы построения динамических срезов, рассмотренные выше. Раздел имеет следующую структуру. В п. 4.5.1 дается общий взгляд на методы динамических срезов. В пп. 4.5.1 и 4.5.2 сравниваются *точность* и *эффективность* методов срезания, решающих одинаковые задачи. Наконец, в п. 4.5.4 исследуются возможности «комбинирования» алгоритмов, решающих разные задачи.

4.5.1. Обзор

В табл. 4, предложенной Типом [130], сведены воедино динамические алгоритмы срезов и выделены основные характеристики каждого подхода. Используются следующие сокращения и обозначения: *МВ (метод вычисления)*: D — уравнения для потока данных; I — динамические отношения зависимостей; G — достижимость в графе зависимостей; R — отслеживание зависимостей в терминах систем переписывания графов (см. раздел 6); *МР (межпроцедурные решения);И (исполняемые срезы);ТД (типы данных)*: С — скаляры, М — массивы/записи, У – указатели; *П (параллельность)*. Для каждой статьи в таблице представлены: (1) метод вычисления срезов; (2) являются ли полученные срезы вычисляемыми программами; (3) поддерживается ли межпроцедурное вычисление; (4) рассматриваемые типы данных; (5) рассматривается ли параллельность. Как и табл. 2, табл. 4 описывает только *рассматривавшиеся проблемы*. В ней *не показано*, как могут быть скомбинированы различные алгоритмы, а также *ничего не говорится* о «качестве» решений. Работы Филда и др. [56, 57, 128, 129], приведенные в табл. 4, опираются на техники, существенно отличающиеся от ранее описанных алгоритмов динамического среза, и будут рассматриваться отдельно в разд. 6.

В отличие от статического случая, имеется значительное различие между методами, вычисляющими исполняемые срезы [51, 95—97], и подходами, дающими в результате только набор операторов [17, 19, 62]. Последний тип срезов может не быть исполняемым из-за отсутствия присваивания возрастающим счетчикам циклов[25]. Для удобства изложения мы будем далее называть такие срезы «неисполняемыми». Как было отмечено в п. 4.1.1, алгоритмы, вычисляющие исполняемые динамические срезы, могут выдавать неточные результаты в присутствии циклов.

Кроме работы Венкатеша [131], любой другой метод вычисления «неисполняемых» срезов почти не занимается проверкой семантической правильности. Алгоритмы, приведенные в [19, 45, 85, 87, 111], решают проблему достижимости в графе, вычисляя множество операторов, которые явно или неявно «влияют» на значения, вычисляемые критерием среза. Кроме самих алгоритмов, не уделяется никакого или почти никакого внимания формальному определению и описанию таких срезов.

Таблица 4

Обзор методов динамических срезов

Авторы	МВ	И	МР	ТД	П
Корел, Ласки [96, 97]	D	да	нет	С, М, У	Нет
Корел, Фергюсон [95]	D	да	нет	С, М	Да
Гопал [62]	I	нет	нет	С	Нет
Агравал, Хорган [19]	G	нет	нет	С	Нет
Агравал и др. [15, 17]	G	нет	да	С, М, У	Нет
Камкар и др. [85, 87]	G	нет	да	С	Нет
Дюстервальд и др. [51]	G	да	нет	С, М, У	Да
Ченг [45]	G	нет	нет	С	Да
Чои и др. [48, 111]	G	нет	да	С, М, У	Да
Филд и др. [56, 57, 128, 129]	R	да	нет	С, М, У	Нет

4.5.2. Точность

Базовые алгоритмы. Срезы, вычисляемые алгоритмом Корела и Ласки [96, 97] (см. подробнее в п. 4.1.1), больше по размеру, чем срезы, вычис-

[25] Конечно, такой срез все равно может быть исполнен; однако он может быть незавершающимся.

ляемые алгоритмами Агравала и Хоргана [19] (п. 4.1.3) и Гопала [62] (п. 4.1.2). Причиной этого является ограничение, введенное Корелом и Ласки: их срезы должны быть исполняемыми программами.

Процедуры. Алгоритмы динамических межпроцедурных срезов, основанные на графах зависимостей, были предложены Агравалом и др. [17] и Камкаром и др. [85, 87] (разд. 4.2). Неясно, вычисляет ли один из этих алгоритмов более точные срезы, чем другой.

Составные переменные и указатели. Корел и Ласки [97] (п. 4.1.1) и Агравал и др. (п. 4.1.3) предложили методы для динамических срезов в присутствии составных переменных и указателей. Однако неизвестно, имеется ли различие в точности этих методов.

Параллельность. Алгоритмы Корела и др. [95] (п. 4.4.1) и Дюстервальда и др. (п. 4.4.2) вычисляют исполняемые срезы, но справляются с недетерминированностью разными способами. Первый подход основан на механизме, воспроизводящем рандеву в срезе программы в том же относительном порядке, в каком они появлялись в исходной программе, анализируя заранее записанный лог-файл. Второй подход заменяет недетерминированные операторы коммуникации, встречающиеся в программе, детерминированными операторами коммуникаций в срезе, так что срез может быть вторично исполнен и даст те же результаты. Ченг [45] и Чои и др. [48, 111] (п. 4.4.2) не ставят эту проблему, так как от вычисляемых их алгоритмами срезов не требуется, чтобы они были исполняемыми программами. Динамические методы для срезов Ченга и Дюстервальда и др. являются неточными, поскольку они основаны на «статических» графах зависимостей, в которых не делается различий между разными вхождениями оператора в истории исполнения (п. 4.1.3).

4.5.3. Эффективность

Поскольку при динамических срезах требуется информация, доступная только во время исполнения, неудивительно, что все изложенные здесь методы динамического среза имеют временные ограничения, зависящие от количества N исполненных операторов (или вызовов процедуры в случае алгоритмов из [85, 87]). Все алгоритмы затрачивают по меньшей мере **O**(N) времени во время исполнения для записи в память истории исполнения программы или для обновления графов зависимостей. Некоторые алгоритмы (например, [95—97]) обходят историю исполнения и извлекают из нее срез, на что тратится еще *как минимум* **O**(N) времени *на каждый срез*, то-

гда как другие алгоритмы затрачивают меньшее (иногда даже константное) время. Далее при обсуждении временных ограничений время, затраченное на создание историй исполнения или графов зависимостей, не будет включаться в общий итог. Ограничения на размер памяти везде будут исчисляться точно.

Базовые алгоритмы. Решение Корела и Ласки [96, 97] (п. 4.1.1) требует **O**(N) места для размещения траектории и **O**(N^2) места для размещения динамических представлений потока. Вычисление представлений потока требует **O**(N*(v+n)) времени, где v и n — число переменных и операторов в программе, соответственно. Извлечение одного среза из вычисленных представлений потока может быть проведено за время **O**(N).

Алгоритм Гопала [62] (п. 4.1.2) требует **O**(N) места для размещения истории исполнения и **O**(n*v) места для размещения отношения μ^{-}_{S}. Время, требующееся для вычисления отношения μ^{-}_{S} для программы S, ограничено **O**(N^2*v^2). Из этого отношения срезы могут быть извлечены за время **O**(v).

Как уже говорилось в п. 4.1.3, предложенный Агравалом и Хорганом метод срезов требует не более чем **O**(n^2) места, где n — количество операторов в программе. Поскольку вершины сокращенного динамического графа зависимостей аннотированы соответствующими им срезами, срезы могут быть извлечены из этого графа за время **O**(1).

Процедуры. Метод для межпроцедурных динамических срезов, предложенный Камкаром и др. [85, 87] (п. 4.2), требует **O**(P^2) места для размещения сводного графа, где P — количество исполненных вызовов процедур. Обход этого графа, необходимый для извлечения среза, требует **O**(P^2) времени.

Временные и емкостные оценки для алгоритма Агравала и др. [17] соответствуют оценкам для базисного алгоритма Агравала—Хоргана, описанного выше.

Составные переменные и указатели. Алгоритмы Корела и Ласки [97] (п. 4.3.1) и Агравала и др. [17] (п. 4.3.2), строящие срезы в присутствии составных переменных и указателей, являются адаптациями базовых алгоритмов Корела—Ласки и Агравала—Хоргана, соответственно (см. выше). Эти адаптации, заключающиеся в изменении функций достигающих определений, используемых при вычислении зависимостей по данным, не влияют на поведение алгоритмов в худшем случае. Следовательно, мы можем предполагать затраты времени и памяти, близкие к соответствующим значениям для скалярного случая.

Параллельность. Алгоритмы Ченга [45] и Дюстервальда и др. [51] основаны на *статических* графах программных зависимостей. Следовательно, только $\mathbf{O}(n^2)$ места требуется для размещения графа зависимостей, и срезы могут быть извлечены из него за время $\mathbf{O}(n^2)$. Распределенный алгоритм среза Дюстервальда и др. [51] использует отдельный процесс среза для каждого процесса программы; процесс среза для процесса P_i требует времени $\mathbf{O}(e_i)$, где e_i — количество дуг в ГПЗ для процесса P_i. Издержки коммуникации между процессами срезов занимают не более чем $\mathbf{O}(e)$ времени, где e — количество дуг в полном графе.

4.5.4. Комбинирование алгоритмов динамических срезов

В табл. 5 представлены решения для «ортогональных» проблем динамического среза: обработка процедур, составных переменных и указателей, коммуникация между процессами. Алгоритмы, основанные на динамических представлениях потока и работающие с составными переменными и указателями [97] и с параллельными программами [95], могут быть интегрированы почти без проблем. Однако для алгоритмов, основанных на графах зависимостей, ситуация может быть более сложной по следующим причинам.

- Использование разных представлений графа. Агравал и др. [17], Камкар и др. [85, 87] и Чои и др. [48, 111] используют динамические графы зависимостей с отдельными вершинами для каждого вхождения оператора в истории исполнения, тогда как Дюстервальд и др. [51] и Ченг [45] пользуются вариациями статических ГПЗ.
- Динамический метод срезов Агравала и др. [17] основан на определении и использовании областей памяти. Все другие методы срезов, использующие графы зависимостей, основаны на определении и использовании имен переменных.

Заметим также, что остается неясным, является ли комбинированный статически/динамический алгоритм Камкара практичным в присутствии составных переменных и указателей, поскольку внутриблочные зависимости не могут быть определены статически в данном случае, и во время исполнения потребуется дополнительный анализ совмещения имен.

Таблица 5

Ортогональные измерения пространства динамических срезов

Базис алгоритма	Процедуры	Составные переменные	Параллелизм
Концепции динамического потока	—	Корел и Ласки [96, 97]	Корел, Фергусон [95]
Отношения динамических зависимостей	Гопал [62]	—	—
Графы зависимостей	Агравал и др. [17], Камкар и др. [85, 87]	Агравал и др. [17]	Дюстервальд и др. [51], Ченг [45], Чои и др. [48, 111]

5. ОБЛАСТИ ПРИМЕНЕНИЯ СРЕЗОВ ПРОГРАММ

5.1. Отладка и анализ программ

Отладка является сложной задачей, если программист работает с большой программой и не представляет себе, в каком месте программы искать ошибку. Срезы программы полезны для отладки, поскольку срез программы потенциально позволяет игнорировать большое число операторов программы во время локализации ошибки [107]. Если программа вычисляет ошибочное значение в переменной x, только операторы, входящие в срез относительно x, с наибольшей вероятностью внесли вклад в появление этой ошибки. В данном случае *наиболее вероятно*, что ошибка появляется в одном из операторов среза. Однако *не всегда* ошибка находится именно в срезе, так как она может содержаться в операторе, который был неумышленно исключен из среза. Тем не менее эта техника полезна, так как она позволяет определить возможное место ошибки с большей точностью, чем без ее применения.

Прямые срезы также полезны при отладке. Прямой срез относительно оператора s может показать, как значение, вычисленное в s, будет далее последовательно использоваться, и может помочь программисту убедиться в том, что s закладывает инварианты, допускаемые более поздними операторами. Это может пригодиться, например, при поиске ошибки занижения или завышения на единицу. Другой целью прямого среза может быть инспекция путей в программе, которые могут быть затронуты предложенной модификацией, для проверки, не имеется ли непредвиденных последствий в поведении программы.

Лайе и Вайзер [109] вводят *нарезку программы кубиками:* метод, комбинирующий информацию из разных срезов. Основная идея состоит в следующем: когда программа вычисляет корректное значение для переменной x и некорректное — для переменной y, ошибка *с большей вероятностью* будет находиться в операторах среза относительно y, а не среза относительно x. Этот подход не является безотказным при наличии множественных ошибок, а также в случаях, когда при неверных начальных данных вычисляются верные конечные данные (ситуация *случайной корректности* по терминологии Агравала [15]). Авторы утверждают, что нарезка программы кубиками дает приемлемые результаты при ослаблении указанных предположений.

Бергеретти и Карре [34] поясняют, каким образом статические методы срезов определяют «мертвый» код, т.е. те операторы программы, которые никак не влияют на результат вычислений по программе. Часто такие операторы являются неисполняемыми из-за наличия ошибки. Статический срез может также применяться для нахождения использований неинициализированных переменных, другого симптома наличия ошибки в программе, а также других свойств, говорящих о неправдоподобности программы. Понятие *правдоподобности* (*plausibility*) программы было введено В.Н. Касьяновым и И.В. Поттосиным [92] и содержательно означает, что все изображаемые программой исполнения осмыслены и согласуются с программным текстом. Оно складывается из следующих прагматических соображений: программа, как правило, в точности решает некоторую задачу в том смысле, что она не должна быть избыточной по отношении к своей задаче; те программные средства, которые выбраны для решения задачи, должны применяться в программе естественным образом; результаты программы не зависят от того, как будут исполнены семантически неопределенные действия. Авторы отмечают в [92], что проверка правдоподобности программы является алгоритмически неразрешимой проблемой, и предлагают использовать при отладке проверку программы на свойства неправдоподобности, выявление которых можно осуществлять статически, в том числе с помощью методов и техники оптимизации программ. Список основных *свойств неправдоподобности* (*properties of implausibility* or *anomalies*), выделенных В.Н. Касьяновым и И.В. Поттосиным [91, 92], включает свойства, связанные с незаданностью переменных (например, использование незаданной переменной), возможностью бесконечного исполнения (бесконечная рекурсия, бесконечный цикл или бесконечное ожидание), существованием неиспользуемых объектов (например, «мертвые» операторы или описанные, но не используемые процедуры и т.д.), существованием излишних вычислений, несоответствием используемых конструкций и алгоритмических действий (переусложненная структура управления, избыточная организация структур данных и т.д.), наличием побочного эффекта в совместных исполнениях, наличием семантически запрещенных или неопределенных конструкций, а также свойство абсолютной неправдоподобности, говорящее о том, что программа совсем не содержит правдоподобных исполнений.

При отладке программист часто бывает заинтересован в специфическом исполнении программы, проявляющем ее аномальное поведение. *Динамические* срезы особенно полезны в этом случае, так как они отражают только актуальные зависимости конкретного исполнения, что дает срезы меньшего размера, чем в статическом случае. Диссертация Агравала [15] содержит

детальное рассмотрение вопроса, каким образом статические и динамические срезы могут использоваться для полуавтоматической отладки программ [17, 19]. Он предлагает вариант, когда пользователь постепенно «поднимается» из области программы, где обнаружила себя ошибка, путем последовательного рассмотрения все более увеличивающихся срезов, фиксирующихся на данных и на управлении. *Срез, фиксирующийся на данных*, получается, когда принимаются в расчет только статические или динамические зависимости по данным; *срез, фиксирующийся на управлении*, состоит из множества управляющих предикатов, окружающих языковую конструкцию. Замыкание всех фиксирующихся на данных и на управлении срезов относительно выражения является статическим или динамическим срезом относительно множества переменных, входящих в это выражение. Информация о нескольких динамических срезах может комбинироваться и давать некоторое представление о местоположении ошибки. К настоящему времени предложено несколько операций на срезах, таких как объединение, пересечение и разность. Операция разности является динамической версией нарезки программы на кубики, предложенной Лайе и Вайзером [109]. Очевидно, что операции комбинирования срезов могут приводить к ошибочным решениям при наличии множественных ошибок или случаев случайной корректности. Агравал и др. [18] обсуждают реализацию программного средства отладки, основанного на идеях предыдущих работ тех же авторов [15, 17, 19].

Пэн и Спэффорд [115, 116] представляют некоторое количество эвристик для локализации ошибки. Эти эвристики описывают, каким образом динамические срезы (вариации представленных Агравалом и др. [19]) используются для выбора множества подозрительных операторов, которые могут содержать ошибку. Подход Пэна и Спэффорда состоит из двух частей. Вначале программа исполняется для большого количества тестов, и каждый тест классифицируется как *вскрывающий ошибку* или *не вскрывающий ошибку* в зависимости от того, обнаруживает ли он присутствие ошибки или нет. Второй шаг состоит из фактических *эвристических правил* для комбинирования информации, содержащейся в динамических срезах для этих тестов, различными способами. Например, можно потребовать вывести список операторов, которые встречаются в *каждом* динамическом срезе для вскрывающего ошибку теста — в этих операторах с большой вероятностью содержится ошибка. Другие эвристики зависят от *частоты встречаемости* или *частоты влияния* операторов в динамических срезах. Частота встречаемости, как ясно из названия, обозначает количество срезов, в которых встречается конкретный оператор, а частота влияния — ко-

личество раз, которое оператор в конкретном срезе встречается в виде «объекта ссылки» в терминах зависимости по данным и по управлению. Например, одна из приводимых Пэном и Спэффордом эвристик состоит в выборе операторов с «высокой» частотой влияния в срезе для выбранного (вскрывающего ошибку) теста. Заметим, что при этом требуется, чтобы пользователь определил *порог*, обозначающий границу между «высокой» и «низкой» частотой. Утверждается, что эту границу можно интерактивно передвигать, чтобы постепенно увеличивать количество рассматриваемых операторов.

Чои и др. [48] описывают структуру и эффективную реализацию отладчика параллельных программ, включающего обратный потоковый анализ — понятие, введенное в работе Бальцера [26]. В двух словах, обратный потоковый анализ определяет, каким образом вычисление значений зависит от более раннего вычисления других значений. Различие между обратным потоковым анализом и основанным на графах динамическим срезом состоит в том, что первый позволяет интерактивно просматривать граф зависимостей, а второй представляет собой множество всех путей программы, соответствующих вершинам графа, из которых может быть достигнута заданная вершина, а именно критерий.

Фрицсон и др. используют статические [58] и динамические [83, 87] межпроцедурные срезы для алгоритмической отладки [126, 127]. Алгоритмический отладчик частично автоматизирует задачу локализации ошибки путем сравнения *подразумеваемого* поведения программы с ее *фактическим* поведением. Подразумеваемое поведение определяется посредством опроса пользователя, ведет ли себя модуль программы (например, процедура) правильно или нет. Используя данные пользователем ответы, отладчик может локализовать ошибку на модульном уровне программы. Применяя процесс алгоритмической отладки к *срезу* относительно неверно вычисляемой переменной, а не к целой программе, можно существенно сократить число задаваемых вопросов.

5.2. Дифференциация и интеграция программ

Под дифференциацией программ [68] понимается анализ старой и новой версий программы и определение набора компонент новой версии программы, представляющего синтаксические и семантические изменения. Такая информация является полезной, поскольку при необходимости нужно будет тестировать только компоненты программы, отражающие измененное поведение. Ключевым моментом дифференциации программ явля-

ется разбиение компонентов старой и новой версий таким образом, чтобы два компонента принадлежали одному сегменту в том и только том случае, если они ведут себя одинаково. Описываемый ниже алгоритм интеграции программ, разработанный Хорвитц и др. [72], сравнивает срезы для того, чтобы распознать эквивалентное поведение. Однако альтернативная техника разбиения, предложенная Янгом и др. [131, 140], основанная не на срезах, а на сравнении небольших модулей программ, дает более точный результат, поскольку может быть снабжена сохраняющими семантику преобразованиями (например, протягиванием копий).

Хорвитц и др. [72] используют статический алгоритм среза для однопроцедурных программ, разработанный Хорвитц и др. в работе [75], как основу для алгоритма интегрирования изменений вариантов программы. На вход алгоритма подается программа Base и два варианта A и B, полученных из Base. Алгоритм состоит из следующих шагов.

1. Строятся графы программных зависимостей G_{Base}, G_A и G_B. Соответствия между «родственными» вершинам этих графов предполагаются доступными.
2. Определяются множества затрагиваемых точек G_A и G_B относительно G_{Base}; они состоят из тех вершин G_A (G_B), которые имеют отличающиеся срезы в G_{Base}[26].
3. Объединенный граф G_M строится из G_A, G_B и множеств затрагиваемых точек, определенных в п. (2).
4. Используя G_A, G_B, G_M и множества затрагиваемых точек, определенные в п. (2), алгоритм определяет, сохраняется ли поведение программ A и B в G_M. Это достигается путем сравнения срезов относительно затрагиваемых точек G_A (G_B) в G_M и G_B (соответственно — G_A). Если найдены различающиеся срезы, тогда изменения противоречат друг другу и интеграция невозможна.
5. Если нет противоречий между изменениями A и B, алгоритм проверяет, является ли G_M настоящим ГПЗ, т.е. соответствует ли он какой-либо программе. Если да, алгоритм строит программу M из графа G_M. Если нет, изменения в A и B не могут быть интегрированы.

[26] Эти множества затрагиваемых точек могут быть эффективно вычислены с помощью прямых срезов относительно всех прямо затрагиваемых точек, т.е. всех вершин G_A, которые не встречаются в G_{Base}, и всех вершин, которые имеют разные наборы входящих дуг в G_A и G_{Base} [74].

Сравнение срезов (шаг 4) полагается на существование отображения между различными компонентами. Однако если такое отображение недоступно, могут быть использованы техники Хорвитц и Репса [73] для сравнения двух срезов за время, линейное относительно суммы их размеров. Семантическая правомерность однопроцедурного алгоритма среза Хорвитц и др. [75] и алгоритма интеграции программ Хорвитц и др. [72] доказана Репсом и Янгом [123]. В этой статье формализуется отношение между поведениями программ при их исполнениях (срезами этих программ), а также между вариантами программ и соответствующей интегрирующей версией.

Репс [118] представляет альтернативную формулировку алгоритма интеграции программ Хорвитц—Принса—Репса, основанную на брауэровых алгебрах. Алгебраические законы, действующие в этих алгебрах, используются для переформулировки алгоритма и для доказательства таких свойств, как ассоциативность последовательных интеграций.

Бинкли и др. обобщают алгоритмы интеграции Хорвитц и др. [72] на многопроцедурные программы. Показано, что такие программы не могут быть интегрированы попроцедурно (поведение программы не будет сохраняться во всех случаях) и что прямое расширение с использованием алгоритма межпроцедурного среза Хорвитц—Репса—Бинкли является недостаточно мощным (оно слишком часто находит противоречия между версиями). Хотя расширенное изложение теории, лежащей в основе алгоритма многопроцедурной интеграции Бинкли—Хорвитц—Репса, выходит за рамки темы статьи, можно отметить, что алгоритм основан на обратных и прямых межпроцедурных срезах и представлении программы в виде графа системных зависимостей.

5.3. Сопровождение программного обеспечения

Одна из проблем сопровождения программ состоит в определении того, будет ли изменение в некотором месте программы влиять на поведение других частей программы. Галлахер и Лайе [60, 61] используют статические срезы для декомпозиции программы в набор компонент (т.е. усеченных программ), каждая из которых воспроизводит часть поведения исходной программы. Они представляют набор директив для сопровождения компонент, которые, если следовать им, предотвращают изменения в поведении других компонент. Более того, они описывают, каким образом изменения в одной компоненте программы могут быть встроены в целую программу семантически непротиворечивым способом.

Галлахер и Лайе используют понятие *декомпозиционного среза* для декомпозиции программ. В общих чертах декомпозиционный срез воспроизводит часть поведения программы, а его дополнение воспроизводит поведение остальной части программы. Декомпозиционный срез относительно переменной v определяется как множество всех операторов, которые могут влиять на «наблюдаемое» значение v в некоторой точке программы; он вычисляется как объединение срезов, взятых относительно v и любого оператора, который выдает значение v, а также конечного оператора программы. *Ограниченный по выводу* декомпозиционный срез (ОВД-срез) есть декомпозиционный срез, из которого удалены все операторы вывода. Два ОВД-среза являются *независимыми*, если у них нет общих операторов; ОВД-срез является *сильно зависимым* от другого ОВД-среза, если он является подмножеством последнего. ОВД-срез, который не является сильно зависимым ни от какого другого ОВД-среза, является *максимальным*. Оператор, встречающийся в более чем одном ОВД-срезе, называется *зависимым*, в противном случае он является *независимым*. Переменная является *зависимой*, если она вычисляется некоторым зависимым оператором, и *независимой*, если она вычисляется только независимыми операторами. Только максимальные ОВД-срезы содержат независимые переменные, а объединение всех максимальных ОВД-срезов эквивалентно исходной программе без операторов вывода. *Дополнение* ОВД-среза определяется как исходная программа минус все независимые операторы ОВД-среза и минус операторы вывода.

Важное наблюдение Галлахера и Лайе [61] состоит в том, что независимые операторы в срезе не влияют на поток данных и управления в дополнении среза. Это дает возможность применять следующие директивы для модификации.

- Независимые операторы могут быть удалены из композиционного среза.
- Присваивания независимым переменным могут быть вставлены в любом месте декомпозиционного среза.
- Логические выражения и операторы вывода могут быть вставлены в любом месте декомпозиционного среза.
- Новые управляющие операторы, которые окружают любой зависимый оператор, будут влиять на поведение дополнения среза.

Новые переменные могут рассматриваться как независимые при том условии, что нет конфликтов имен с переменными дополнения среза. Если требуются такие изменения, которые затрагивают зависимую переменную v, пользователь может либо расширить срез таким образом, что v станет независимой, либо ввести новую переменную. Слияние измененных ком-

понентов в целую программу является тривиальной задачей. Поскольку алгоритм гарантирует, что изменения в ОВД-срезе не влияют на его дополнение, необходимо проверить только измененный срез.

5.4. Тестирование

Программа удовлетворяет «общепринятому» критерию *проверки потока данных*, если все пары def-use встречаются в успешном тестовом задании. Дюстервальд и др. [52] предложили более строгий критерий тестирования, основанный на срезах программ: каждая пара def-use должна быть выполнена в успешном тестовом задании; кроме того, она должна *влиять на выходные данные*, т. е. влиять как минимум на одно выходное значение. Пара def-use является влияющей на выходные данные, если она появляется в *выходном срезе*, то есть в срезе относительно оператора вывода. Задачей пользователя либо автоматического генератора тестов является составление достаточного количества тестовых заданий, так что все пары def-use будут протестированы. Используются три подхода к построению срезов на основе различных графов зависимостей. Статические срезы вычисляются с использованием *статических графов зависимостей* (подобных графам программных зависимостей у Хорвитц и др. [75]), динамические срезы вычисляются с использованием *динамических графов зависимостей* (подобных динамическим графам зависимостей у Агравала и Хоргана [19], но отличающихся тем, что экземпляры одной и той же вершины сливаются вместе, немного снижая точность) и *гибридные срезы* вычисляются с использованием графов зависимостей, основанных на комбинации статической и динамической информации. При гибридном подходе множество переменных программы разбивается на два непересекающихся подмножества таким образом, что все переменные одного подмножества не ссылаются на переменные из другого подмножества. Статические зависимости вычисляются для одного подмножества (обычно это скалярные переменные), динамические — для второго (обычно это массивы и указатели). Преимущество этого подхода состоит в том, что он объединяет разумную эффективность и разумную точность.

Камкар и др. [86] расширяют алгоритмы Дюстервальда, Гупты и Соффа для многопроцедурных программ. Они ввели пригодное к использованию понятие межпроцедурных пар def-use. Межпроцедурный метод динамического среза Камкара и др. [85, 87] используется для определения того, какие пары def-use влияют на точное выходное значение в заданном тестовом задании. Представление программы в виде сводного графа, описанное в п.

4.2, используется здесь в слегка измененном виде, вершины и дуги здесь аннотированы def-use-информацией. С помощью такого представления множество пар def-use, реализованных срезом, может быть эффективно найдено.

Регрессивное тестирование заключается в повторном тестировании только тех частей программы, которые были затронуты модификацией протестированной ранее программы, при этом сохраняется степень покрытия исходного набора тестов. Гупта и др. [63] описывают подход к регрессивному тестированию, использующий срезы. Обратные и прямые статические срезы служат для нахождения путей программы, затрагиваемых изменениями, и только те тестовые задания, в которых выполняются затрагиваемые пары def-use, должны быть исполнены повторно. Срезы вычисляются при помощи обратного и прямого обхода управляющего графа программы, начиная в точке модификации. Однако алгоритмы Гупты и др. [63] были разработаны для нахождения информации, необходимой только для регрессивного тестирования (то есть затрагиваемых пар def-use).

Бинкли [36] описывает способ уменьшения стоимости регрессивного тестирования *многопроцедурных* программ при помощи, во-первых, сокращения количества тестов, которые должны повторно исполняться, и, во-вторых, уменьшения размера программы, на которой они должны исполняться. Это достигается путем определения множества точек программы, *затрагиваемых* модификацией, и множества *сохраненных* точек программы (см. разд. 5.2). Множество затрагиваемых точек используется для построения меньшей по размеру и более эффективной программы, которая воспроизводит только модифицированное поведение исходной программы; все тестовые задания, которые должны быть повторно исполнены, могут применяться к вновь полученной программе. Множество сохраненных точек используется для распознавания того, какие тестовые задания не требуют повторного исполнения.

Бэйтс и Хорвитц [30] используют вариацию понятия графа программных зависимостей, введенного Хорвитц и др. [72], для инкрементного тестирования программы. Критерии тестирования определяются в терминах понятий ГПЗ: критерий тестирования «по всем вершинам» удовлетворяется, если каждая вершина ГПЗ выполняется набором тестов (т.е. исполняется каждый оператор и управляющий предикат программы). Критерий «по всем управляющим дугам» определяется подобным же образом. Имея тестированную и впоследствии модифицированную программу, мы можем использовать срез для определения: (1) операторов, затрагиваемых модификацией и (2) тестовых заданий, которые могут быть вновь использованы

для модифицированной программы. Грубо говоря, в первое множество входят операторы, которые не встречались ранее, а также операторы, срезы для которых отличаются от их предыдущих версий. Объекты второго множества требуют разбиения операторов исходной и модифицированной программы на эквивалентные классы; операторы принадлежат одному классу, если они имеют один и тот же срез «по управлению» (несколько модифицированная версия стандартного понятия). Бэйтс и Хорвитц доказывают, что операторы из одного и того же класса исполняются одними и теми же тестовыми заданиями.

5.5. Настройка компиляторов

Ларус и Чандра [105] представляют подход к настройке компиляторов, в котором динамические срезы используются для определения потенциальных вхождений избыточных общих подвыражений. Обнаружение общего подвыражения означает, что код приближается к оптимальному.

Объектный код снабжается генерирующими след инструкциями. Регенератор следа читает след и производит поток событий, таких как чтение и запись областей памяти. Этот поток событий является входом для программы-аудитора (например, ответственного за удаление общих подвыражений), который строит динамические срезы относительно текущих значений регистров. Ларус и Чандра используют вариацию подхода Агравала и Хоргана [19]: динамический срез представлен ориентированным ациклическим графом, содержащим все операторы и операнды, которые производят текущее значение регистра. Общее подвыражение имеется тогда, когда изоморфные графы строятся для двух разных регистров. Однако описанная ситуация означает только то, что общее подвыражение присутствует в *данном конкретном* исполнении. Общее подвыражение наличествует на *всех* путях исполнения, если его входные значения одни и те же при всех исполнениях. Это проверяется следующим образом: (1) счетчик программы PC1 для первого вхождения общего подвыражения является доминатором счетчика программы PC2 для его второго вхождения; (2) регистр, содержащий первое вхождение общего подвыражения, не модифицировался ни на одном пути между PC1 и PC2; (3) входные значения общих подвыражений также не модифицируются ни на одном пути между PC1 и PC2. Хотя третье условие в общем случае не поддается проверке, вполне возможно проверить его для некоторых количества специальных случаев. В общем случае проверка условия (3) передается на усмотрение разработчика компилятора.

5.6. Повторное проектирование

Необходимость перепроектирования и переноса на современные платформы больших программных комплексов, время жизни которых приходилось на несколько последних десятилетий, поставила программистов перед серьезной проблемой. Миллионы строк программного кода, написанного на популярных некогда языках Кобол и PL/I, отсутствие спецификаций решаемых задач и комментариев, неструктурность программ, вызванная как особенностями языка программирования, так и многочисленными поправками к исходному коду программистов сопровождения, множество разнородных компонентов и неявных связей между ними (типичное приложение на языке Кобол включает в себя несколько программ, описания таблиц баз данных, экранных форм, последовательных и индексных файлов) — все это приводит к неоправданно высоким затратам времени на простую адаптацию программиста к программному окружению, понимание алгоритмов и внутренних связей приложения. По оценкам фирм, занимающихся ручным переводом старых программ, доля этих затрат составляет от 50 до 90%.

Как отмечается в [3], решение этой задачи может быть основано на построении срезов программ, а также их модификаций, позволяющих использовать механизм срезов для глубокого анализа старых программ, которое может помочь разработчику найти и выделить из сложной программы необходимую ему функциональность, поместить соответствующую часть исходной программы в отдельный модуль и переиспользовать в дальнейшем, например, для перевода на современную языковую платформу.

Джексон и Роллинз представляют инструмент повторного проектирования, называемый Chopshop [79], который основан на техниках, разработанных в [78] (см. пп. 3.1.3 и 3.2.3). Этот инструмент предоставляет средства для визуализации срезов программ в виде графических диаграмм. Наряду со «сколом» (см. п. 3.1.3), Chopshop может также производить «реферирование» срезов, удаляя все вершины графа, не являющиеся точками вызовов, и получая граф, содержащий только вершины точек вызова и дуги транзитивных зависимостей между этими вершинами.

Нинг и др. [112] описывают набор инструментов для выделения компонентов из больших Кобол-систем. Эти инструменты включают средства для *сегментации программ*, т.е. разделения их на куски функционально связанного кода. Вдобавок к обратным и прямым статическим срезам могут быть определены срезы, *основанные на условии*. Для такого среза критерий определяет ограничения на значение определенной переменной.

5.7. Другие применения срезов

Вайзер [138] описывает способ применения срезов для *распараллеливания* исполнения последовательной программы. Несколько срезов исходной программы исполняются параллельно, а выходные данные этих срезов *склеиваются* вместе таким образом, что поведение исходной программы относительно ввода/вывода сохраняется. Естественное требование к алгоритму склеивания Вайзера состоит в том, чтобы набор всех срезов «покрывал» поведение целой программы. Склеивание не основывается на какой-либо отдельной технике срезов; любой метод вычисления исполняемых статических срезов может быть использован. Рассматриваются только программы со структурированным потоком управления, поскольку алгоритм склеивания опирается на то, что поведение программы может быть выражено в терминах так называемых регулярных выражений программы. Реконструкция исходного поведения программы относительно ввода/вывода становится неразрешимой задачей в присутствии невосстановимого потока управления.

Отт и Тусс [113] рассматривают модуль программы как множество элементарных процессоров, работающих совместно для вычисления выходных значений модуля. Они разбивают модули на *классы сцепления* (согласно характеру взаимоотношений между элементарными процессорами), сравнивая срезы относительно различных выходных переменных. *Низкое сцепление* соответствует ситуации, когда модуль разбивается на отдельные множества не связанных между собою элементарных процессоров. Каждое множество вовлечено в вычисление отдельного выходного значения, и между срезами не имеется перекрытий. *Сцепление по управлению* возникает, когда имеются два или более множества не связанных элементарных процессоров, каждое из которых зависит от общего входного значения; пересечение срезов состоит из управляющих предикатов. *Сцепление по данным* соответствует ситуации, когда данные перетекают от одного множества элементарных процессоров к другому; срезы будут иметь непустое пересечение и нетривиальные различия. Ситуации с *высоким сцеплением* напоминают конвейер. Данные перетекают от элементарного процессора к его преемнику; срезы модулей с высоким сцеплением перекрываются весьма значительно. В статье не делается упора на конкретный метод срезов и не приводится никаких численных характеристик.

Бинкли [43] представляет семантику переписывания графов для ГСЗ, используемую для проведения межпроцедурного протягивания констант. Алгоритм межпроцедурного среза Хорвитц—Репса—Бинкли используется

для нахождения срезов, которые при исполнении могут давать константные значения.

Бек и Айхманн [33] рассматривают случай, когда используется «стандартный» модуль для модуля абстрактного типа данных, при этом требуется только часть его функциональности. Их цель состоит в «отрезании» всего лишнего от кода модуля. Они обобщают понятие статических срезов на модульные программы. Для того, чтобы получить урезанную версию модуля, строится *граф интерфейсных зависимостей* (*ГИЗ*). Этот граф содержит вершины для всех определений типов и глобальных переменных, а также подпрограмм внутри модуля. Он также содержит дуги для всех отношений def-use между вершинами. *Интерфейсный критерий рассечения* состоит из модуля и подмножества операций абстрактного типа данных. Вычисление интерфейсных срезов соответствует решению проблемы достижимости в ГИЗ-графе. Межмодульные срезы, соответствующие ситуациям, когда модули импортируют другие модули, могут быть вычислены с помощью вывода нового критерия для импортированных модулей.

6. ПОСЛЕДНИЕ РАЗРАБОТКИ

В этом разделе рассматриваются недавние работы по улучшению точности методов срезов, которые опираются на отказ от двух важнейших ограничений, характерных для ранее описанных алгоритмов построения срезов:

1) срез состоит из подмножества операторов исходной программы; иногда дополнительно требуется, чтобы срез представлял собой синтаксически правильную программу.
2) срезы вычисляются путем трассировки зависимостей по данным и по управлению.

Оба этих ограничения отрицательно влияют на точность вычисляемых срезов. Кроме того, важно понимать, что они связаны друг с другом: в большинстве случаев отказ от первого ограничения делает возможным отказ от второго.

Вайзер уже рассматривал некоторые проблемы, вызванные первым ограничением, в своей диссертации [136], где он замечает: «Для построения хорошего среза на исходном языке требуются преобразования помимо удаления операторов». Это замечание можно легко понять, если рассмотреть ситуацию, где язык программирования не допускает существования оператора **if** с пустыми ветвями, но при этом используемый алгоритм среза пытается удалить все операторы на одной ветвей условного оператора. Понятно, что все эти операторы никогда не будут удалены из среза, поскольку результатом их удаления будет синтаксически неправильная программа. Хванг и др. [77] описывают некоторое количество родственных проблем и заключают, что на практике удаление операторов само по себе не является адекватным методом получения срезов.

Второе ограничение — тот факт, что срезы вычисляются путем обхода зависимостей по данным и по управлению — также должно быть устранено, если единственное требование к срезам состоит в том, что они должны быть насколько возможно малы. Рассмотрим для примера программу на рис. 25, *а*. Здесь статический срез относительно оператора write(n), вычисленный любым «обычным» статическим алгоритмом, состоит из всей программы целиком[27]. Однако если при построении среза будут использованы такие техники оптимизации, как протягивание констант [9, 133] или подобные ей, полученные срезы могут быть более точными. Для программы на

[27] Некоторые алгоритмы [34, 139] исключили бы оператор write.

рис. 25, *а*, например, можно заметить, что значение i является константой и что ветвь **else** условного перехода никогда не выбирается. Следовательно, становится возможным вычисление более точного среза, представленного на рис. 25, *б*. Более того, если допускается замена всего оператора **if** одним из операторов на его ветви, можно получить минимальный срез, представленный на рис. 25, *в*.

```
read(n);                read(n);                read(n);
i := 1;                 i := 1;
if (i > 0) then         if (i > 0) then
  n := n + 1              n := n + 1              n := n + 1
else                    else
  n := n * 2;                     ;               ;
write(n)                write(n)                write(n)
```

а | *б* | *в*

Рис. 25. Исходная программа = статический срез относительно оператора write(n) (*а*); более точный срез, полученный после применения протягивания констант (*б*); минимальный срез (*в*)

Другие техники компиляторной оптимизации[28], символьное исполнение и разнообразные сохраняющие семантику преобразования также могут быть использованы для получения более точных срезов. На рис. 26, *а* представлена программа, для которой нужно построить срез относительно ее последнего оператора write(y). Вновь мы видим, что традиционные алгоритмы построения среза дадут в результате целую программу. Более точный срез может быть получен после «слияния» двух операторов **if**. Результат этого сохраняющего семантику преобразования показан на рис. 26, *б*. Ясно, что алгоритм построения среза, который может производить такое преобразование, способен в принципе вычислять более точные срезы, подобно приведенному на рис. 26, *в*.

[28] Смотри, например, обстоятельный обзор в [141] или книгу [9] .

```
Read(p);
read(q);
if (p = q) then
  x := 18
else
  x := 17;
if (p <> q) then
  y := x;
else
  y := 2;
write(y)
```

а

```
read(p);
read(q);
if (p = q) then
  begin
    x := 18;
    y := 2
  end
else
  begin
    x := 17;
    y := x;
  end
write(y)
```

б

```
read(p);
read(q);
if (p = q) then

else
  x := 17;
if (p <> q) then
  y := x;
else
  y := 2;
write(y)
```

в

Рис. 26. Исходная программа = статический срез относительно оператора write(y) (*а*); преобразованная программа (*б*);
более точный срез, полученный после рассечения преобразованной программы (*в*)

Подходы, использующие техники оптимизации для получения более точных срезов, как на рис. 25 и 26, были предложены Филдом и др. [56, 57, 128] и Эрнстом [53]. На концептуальном уровне эти подходы состоят из следующих компонентов:

- перевод программы в подходящее промежуточное представление (ПП);
- преобразование и оптимизация ПП;
- поддержка отображения между исходным текстом программы, исходным ПП и преобразованным ПП;
- извлечение срезов из ПП.

Филд и др. [56, 57, 128] используют промежуточное представление для императивных программ, называемое P_{IM} [55], как базис для своего алгоритма построения срезов. И перевод программы в ее P_{IM}-представление и последующая оптимизация P_{IM}-графов определяются посредством эквациональной логики, которая может быть реализована как переписывание термов [94] или переписывание графов [29]. Соответствие между исходным

текстом программы, ее изначальным P_{IM}-графом и вычисляемым впоследствии оптимизированным P_{IM}-графом поддерживается автоматически с помощью техники, называемой *динамической трассировкой зависимостей* [57, 128]. Эта техника, определяемая для произвольных систем переписывания термов, отслеживает то, каким образом новые вершины, динамически создаваемые в процессе переписывания, *зависят* от ранее существовавших вершин. Эти соответствия хранятся в P_{IM}-графах в виде аннотаций к вершинам подобно тому, как информация хранится в сокращенном динамическом графе зависимостей, описанном Агравалом и др. [19] (см. п. 4.1.3). Извлечение среза относительно указанного выражения включает поддержку указателя на P_{IM}-подграф для этого выражения и получение динамической информации о зависимостях, размещенной в этом P_{IM}-подграфе. За подробным изложением того, каким образом это производится, читатель может обратиться к [56, 128].

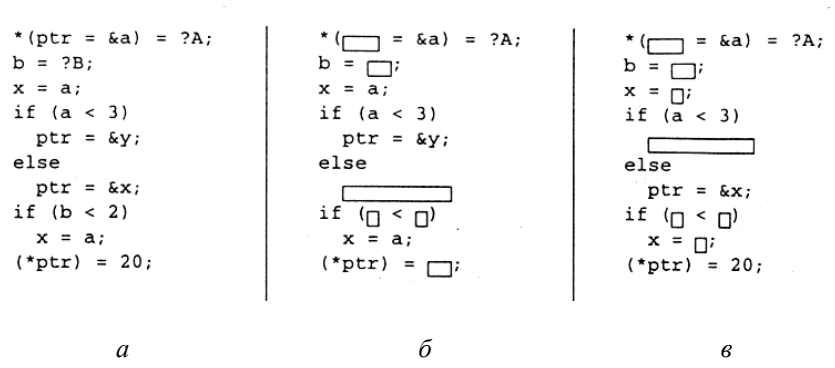
Как P_{IM}-графы, так и динамическая трассировка зависимостей были реализованы с использованием мета-окружения ASF+SDF, генератора программного окружения [29], разработанного в известном институте CWI (г. Амстердам). Недавние эксперименты дали обнадеживающие результаты. В частности, были вычислены срезы, представленные на рис. 25, *б* и 26, *в*. Недавно Тип [128, 129] показал, что динамическая трассировка зависимостей может быть также использована для вычисления точных динамических срезов из простой алгебраической спецификации [35], определяющей интерпретатор.

Эрнст [53] использует *граф зависимостей по значению* (*ГЗЗ*) [135] в качестве промежуточного представления своего алгоритма построения срезов. Вершины ГЗЗ соответствуют вычислениям, а дуги представляют передачи значений между вычислениями. Наиболее существенными характеристиками ГЗЗ являются следующие: (1) поток управления представлен как поток данных, (2) циклы моделируются вызовами рекурсивных функций, (3) все значения и вычисления в программе, включая операции над динамическими областями и потоками ввода/вывода, явно представлены в ГЗЗ. Преобразования и оптимизации ГЗЗ более или менее детально описаны в [135]. Эрнст обращается к проблеме поддержки соответствия между ГЗЗ и графом исходной программы в процессе оптимизации, но никаких деталей относительно того, каким образом поддерживается это соответствие, не приводит. Для извлечения срезов из ГЗЗ Эрнст использует простой и эффективный алгоритм достижимости в графе, подобный тому, которым пользовались К. Оттенштейн и Л. Оттенштейн [114].

К настоящему времени не проведено глубокое сравнение подходов Филда и др. и Эрнста, как по причине использования сложных оптимизаций, так и по причине отсутствия информации о соответствии представлений, используемых Эрнстом. Однако некоторые различия между этими работами выглядят довольно очевидными.

- Язык, рассматриваемый Эрнстом, существенно шире того, который исследуется в статье Филда—Рамалингама—Типа. Эрнст реализовал алгоритм рассечения для полного языка C (включая рекурсивные процедуры, см. п. 3.2.3), тогда как Филд и др. пока не занимаются проблемами, вызванными процедурами и неструктурированным потоком управления.
- Подход Филда и др. разрешает использование различных вариаций P_{IM}-логики для обработки циклов в соответствии с разными «уровнями ленивости» семантики языка. В зависимости от выбранного варианта вычисленные алгоритмом срезы будут напоминать либо неисполняемые срезы Агравала и Хоргана [19], либо исполняемые срезы Корела и Ласки [97]. Из статьи Эрнста неясно, предоставляет ли его подход подобную гибкость.
- Филд и др. разрешают вычисление срезов при наличии любого набора ограничений на входные данные программы и определяют соответствующее понятие *ограниченного* среза, которое обобщает традиционные понятия статических и динамических срезов. Это достигается при помощи переписывания P_{IM}-графов, которые содержат переменные (соответствующие неизвестным значениям) в комбинации с P_{IM}-правилами, моделирующими символьное исполнение. Эрнст не рассматривает вопрос подобной гибкости своего алгоритма.
- Филд и др. определяют срезы как *подконтекст* (то есть «связанное» множество функциональных символов) абстрактного синтаксического дерева программы. Операторы или выражения программы, которые не входят в срез, представлены «дырками» (т.е. отсутствующими подтермами) в контексте. Хотя это понятие среза не составляет исполняемую программу в общепринятом смысле слова, полученные в результате срезы *являются* исполняемыми программами в том смысле, что такой срез может быть переписан в виде P_{IM}-графа, содержащего то же самое значение для выражения, указанного в критерии среза, при соблюдении заданных ограничений на входные значения программы.

На рис. 27 представлен пример программы, взятый из [56], и некоторые из ограниченных срезов, полученные при помощи подхода Филда и др.[29] Интуитивно эти срезы понять несложно: «заклеенное» выражение среза может быть заменено любым другим выражением, что не повлияет на вычисление указанного в критерии значения при заданных ограничениях на входные значения программы. Хотя и чересчур хитроумным на первый взгляд способом, этот пример иллюстрирует несколько важных моментов. В силу отказа от требования синтаксической корректности, мы можем выбирать между операторами присваивания, у которых в срез включена правая часть, и операторами присваивания с включенной левой частью, как показано на рис. 27, *б*. Заметим, что можно установить, что значения, проверяемые условным оператором, не относятся к срезу, хотя тело этого оператора относится к нему. В целом, этот подход дает возможность различать множество нюансов, на что неспособны традиционные алгоритмы срезания.



а *б* *в*

Рис. 27. Исходная программа (*а*); ее ограниченный срез относительно последнего значения x при заданном ограничении ?A := 2 (*б*); условный ограниченный срез относительно последнего значения x при заданном ограничении ?A > 5 (*в*)

[29] В данном примере выражения, начинающиеся с вопросительного знака (например, «?A»), представляют неизвестные значения или входные данные. Подтермы абстрактного синтаксического дерева программы, которые не встречаются в срезах на рис. 27(б) и 27(в), «заклеены».

Применение техники оптимизации программ особенно актуально для динамических срезов, размер которых может быть неоправданно большим для реальных программ.

В работе Бежедэса и др. [37] описан алгоритм построения обратных динамических срезов для С программ, который, по словам авторов, может использоваться для обработки реальных программ, причем реальных как в смысле размера обрабатываемых программ, так и в смысле использования в них разных конструкций языка С, таких как указатели или вызовы функций. Алгоритм работает в параллель с исполнением обрабатываемой программы и является глобальным в том смысле, что после завершения исполнения последнего оператора обрабатываемой программы алгоритм выдает срезы для всех ее операторов, ранее исполненных. Метод не конструирует ни граф динамических зависимостей, ни какую-либо его вариацию, а просто всегда хранит множество всех операторов, информационно влияющих на оператор, исполняемый в данный момент. Основным достоинством алгоритма называется то, что его емкостная сложность пропорциональна количеству различных элементов памяти, используемых обрабатываемой программой. Их количество во многих ситуациях существенно меньше длины цепочки исполнения, которая, по существу, является абсолютной верхней границей для алгоритмов динамических срезов.

Среди других последних работ, посвященных срезам программ, следует также выделить работу Ларсена и Харрольда [104], в которой рассматривается решение задачи для языков объектно-ориентированного программирования. Ими описана конструкция графов системных зависимостей для объектно-ориентированных программ, на основании которых можно осуществлять эффективное построение срезов. Графы могут строиться для отдельных классов, групп взаимодействующих классов и объектно-ориентированных программ целиком. Для неполной программной системы, состоящей из единственного класса и группы взаимодействующих классов, строится некоторый граф процедурных зависимостей, моделирующий все возможные обращения к общим методам класса. Для полной системы этот граф конструируется по главной программе системы. Одним из достоинств предложенного подхода является то, что графы системной зависимости могут строиться инкрементально, что поддерживает переиспользование представлений классов. Второе достоинство – это то, что данный подход позволяет строить срезы для неполных объектно-ориентированных программ, таких как классы или библиотеки классов. Авторы представляют свои методы для языка С++, но отмечают, что их методика применима к

другим объектно-ориентированным языкам со статической проверкой типов, например, к языку Ada-95.

До последнего времени основное внимание большинства исследований в области построения срезов уделялось алгоритмическим аспектам. Тому, каким образом лучше визуализировать и интерактивно отображать или просматривать срезы [53, 79], уделялось намного меньше внимание. Недавно разработанное программное средство SeeSlice Болла и Эйка [23] представляет некоторые интересные новые идеи по визуализации срезов больших программ.

7. КОНКРЕТИЗАЦИЯ ПРОГРАММ

7.1. Понятие конкретизации

В данном разделе рассматривается задача конкретизации программ, которая охватывает существенно более широкий класс срезов программ, чем тот, который был рассмотрен Вайзером и его последователями, и включает различные процессы семантической обработки программ при построении этих срезов, такие, как, например, рассмотренные в разделе 6. Понятие конкретизации было введено В.Н. Касьяновым [4—12, 88—91] в рамках его работ по исследованию технологических возможностей оптимизации программ и развитию трансформационного подхода к программированию.

В общем случае под *конкретизацией,* или, что то же самое, *оптимизацией в контексте,* понимается такое преобразование программы в рамках одного языка, при котором сохраняется смысл программы и повышается ее качество в некотором подмножестве ее применений и относительно заданного критерия качества. Другими словами, задачей конкретизации является построение оптимизированного среза исходной программы.

Каждое применение программы состоит в том, что оно воспринимает и (или) вырабатывает значения объектов, являющихся по отношению к программе внешними (глобальными) аргументами и (или) результатами. Суженное множество применений может характеризоваться сокращением не только множества внешних аргументов, но и множества внешних результатов — тех внешних объектов, вычисление которых и рассматривается в качестве суженной цели применений программы. Сужение множества применений программы может также проявляться в ограничении на комбинации значений внешних аргументов. Например, возможна ситуация, когда многовариантный параметрический модуль используется для обработки только тех вариантов, для которых выполняется определенное свойство для значений параметров (в частности, только тех вариантов, в которых некоторые из параметров имеют известные постоянные значения: см., например, задачу смешанных вычислений [1]). Понятие качества программ является многоплановым. В практике программирования одновременно широко используются такие критерии качества, как эффективность, надежность, наглядность и т. п., каждый из которых является также многоплановым; в частности, эффективность обычно рассматривается отдельно по отношению к таким разным ресурсам ЭВМ, как время и память. При этом, как правило, различные показатели качества программы являются противоречивыми (например, более эффективная программа очень часто ока-

зывается менее наглядной или менее надежной, а программа, работающая быстрее, при своем выполнении, как правило, занимает большую память).

На рис. 28 содержится пример универсальной программы Паскаль-процедуры EXAM1, которая осуществляет вычисление в X решения системы уравнений с матрицей коэффициентов A, имеющей над главной диагональю все нули. Тогда процедура EXAM2 является конкретизацией процедуры EXAM1 в контексте, предполагающем диагональный вид матрицы A и эффективность в качестве критерия оптимизации. Если контекст применения процедуры EXAM1 предполагает ее внешним результатом только X [1], то процедура EXAM3 является конкретизацией процедуры EXAM1 при любом естественном критерии качества.

```
procedure EXAM1 (A:MATRIX; B:VECTOR; var X:VECTOR);
        var I, J, Z: integer;
begin  X[1]:=B[1]/A[1, 1];
        for I:=2 to N do
                begin  Z:=0; for J:=1 to I-1 do Z:=Z+A[I,J]*X[J];
                        X[I]:=(B[I]-Z)/A[I,J]
                end
end;

procedure EXAM2 (A: MATRIX; B: VECTOR; var X: VECTOR);
        var I: integer;
begin for I:=1 to N do X[I]:=B[I]/A[I, I] end;

procedure EXAM3 (A: MATRIX; B: VECTOR; var X: VECTOR);
        begin X[1]:=B[1]/A[1,1] end;
```

Рис. 28. Пример Паскаль-процедуры и двух ее возможных конкретизаций

Для того чтобы, используя конкретизацию, осуществить построение простого среза некоторой программы P относительно некоторого заданного критерия (n, X), можно, например, построить программу P', добавив в точку n программы P набор из |X| фиктивных операторов присваивания y := x значений переменных x $\in$ X новым глобальным переменным y $\in$ Y, а затем осуществить конкретизацию P' относительно контекста, рассматривающего Y в качестве результатов, например, с помощью редукции (см. разд. 7.3) .

7.2. Аннотированное программирование

Основная идея конкретизации – это получение преимуществ от известного контекста, в котором только некоторые программные выполнения допустимы и только некоторые из их результатов используются, с целью реализации более качественного (в смысле, определенном контекстом) вычисления используемых результатов. Однако современные языки программирования высокого уровня не содержат достаточно средств для описания контекста применения программ.

Поэтому естественным представляется подход, предложенный В.Н. Касьяновым [4—12, 88—91] для решения задач конкретизации, в рамках так называемого *аннотированного* программирования, основанного на расширении исходного (*базисного*) языка программирования формализованными комментариями (*аннотациями*), в которых может быть выражена контекстная информация. Аннотации, предназначенные для спецификации контекста, могут иметь форму *утверждений* и *директив*. Утверждение является предикатным ограничением на допустимые состояния памяти в соответствующей точке аннотированной программы, а директива — это либо оператор, который будет изменять текущее состояние памяти каждый раз, как эта аннотация проходится при исполнении аннотированной программы, либо имя конкретизирующего преобразования, применение которого в соответствующей точке аннотированной программы разрешено контекстом.

Например, фрагмент программы рис. 29, *а*, в котором S1, S2, S3 — произвольные операторы, а все три аннотации являются утверждениями и задают, что перед исполнением фрагмента всегда X=2 и Y=3, а после его исполнения не используются текущие значения переменных A и B, эквивалентен фрагменту рис. 29, *б*, базисная часть которого состоит из единственного оператора присваивания, являющегося срезом исходного базисного фрагмента программы по отношению к контексту, заданному аннотациями.

Класс корректных преобразований аннотированных программ охватывает различные виды работы с базисными программами и позволяет конструировать инструменты семантической обработки программ различных типов [6, 9—12, 88—91]. По существу, механизм аннотаций является полным в том смысле, что позволяет реализовать любой алгоритм обработки программ [89].

Аннотированное программирование позволяет специализирующие и обобщающие преобразования базисных программ сводить к эквивалентным преобразованиям аннотированных программ и применять для их исследования методы эквивалентных преобразований, разработанные в рамках тео-

рии схем программ. Другое преимущество аннотированного программирования — это возможность реализации глобальных преобразований базисных программ итеративным применением так называемых *бесконтекстных* преобразований (*трансформаций*) аннотированных программ.

а

```
begin {$ X=2} {$ Y=3}
  case X of
    1: S1;
    2: if Y>2 then begin A:=X+1; B:=Y+2 end else S2 end;
    3: S3
  end;
  A:=A+1;
  W:=B+1
  {$ DEAD(A, B)}
end;
```

б

```
begin {$ X=2} {$ Y=3}
        W:=5
        {$ DEAD(A, B)}
end;
```

Рис. 29. Примеры эквивалентных аннотированных программ

Наличие аннотаций, содержащих априорно известную информацию о свойствах программы в рамках суженного контекста ее применения, требует обобщения постановки потокового анализа (задачи анализа свойств[26]). При этом возникают две трудности. Во-первых, в общем случае та информация, которая содержится в аннотации, с одной стороны, не может быть получена из остальной части программы, а с другой — может быть дополнена при рассмотрении программы. Во-вторых, наличие априорной информации требует учета противоположных зависимостей в изучаемых свойствах (например, в прямой[27] задаче о равенстве переменных константам наличие информации о

[26] Задача анализа свойств состояний программы [9] в общем виде представляет собой задачу построения по тексту программы набора утверждений (инвариантов), отнесенных ко всем точкам программы и дающих достоверную информацию о поведении программы в этих точках при любом ее исполнении.

[27] В традиционных постановках задачи анализа свойств извлекаются лишь такие свойства, которые для каждой точки программы говорят либо о том, что может случиться после прохождения этой точки (*обратная* задача), либо о том, что может произойти перед тем, как управление

константности переменной в каком-то операторе может означать константность переменных не только в преемниках, но и в предшественниках этого оператора).

Задача анализа свойств состояний, допускающая наличие априорной информации о программе и одновременное существование прямых и обратных зависимостей в изучаемых свойствах, рассматривается в [8].

7.3. Класс редуцирующих конкретизаций

Если зафиксировать класс программ и рассмотреть множество всех конкретизирующих преобразований в этом классе, то подклассы преобразований, конкретизирующих программы относительно разных направлений качества, имеют непустое пересечение (рис. 30). Конкретизирующие преобразования из этого пересечения получили название редуцирующих [7].

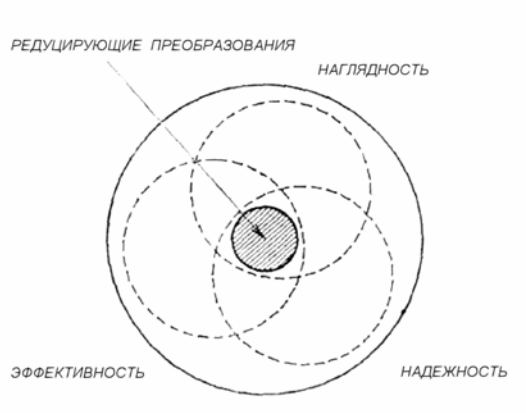


Рис. 30. Структура класса конкретизирующих преобразований и редукции

Класс *редуцирующих конкретизаций* характерен тем, что в процессе редукции повышение тех или иных показателей качества программы происходит не за счет перестройки программы, которая может привести к пониже-

---

достигнет этой точки (*прямая* задача). В прямой задаче в качестве точек программы рассматриваются все входы ее операторов и предполагается известным инвариант для входной точки программы. В обратной задаче множество точек программы образуется всеми выходами ее операторов и инвариант является известным для выходной точки программы.

нию других показателей качества, а за счет удаления из универсальной программы тех объектов и конструкции, которые становятся избыточными в силу сужения множества решаемых с ее помощью задач.

Следует заметить, что многие традиционные оптимизирующие преобразования, по существу, реализуют редукцию отдельных конструкций программы на основе контекста использования этой конструкции в программе, найденного при ее потоковом анализе.

В [7, 9, 90, 91] описаны редуцирующие преобразования для разных представлений базисных программ и аннотаций к ним.

8. ЗАКЛЮЧЕНИЕ

Данная работа — первая в отечественной литературе попытка обзора так называемого слайсинга (slicing), одного из эффективных и активно развиваемых направлений современных технологий конструирования программ. Цель ее вполне понятна – дать возможность отечественным разработчикам познакомиться с методами построения и применения так называемых срезов программ (program slices), широко используемых в западных разработках.

В качестве основы для классификации техник срезов мы, следуя Типу [130], взяли статический или динамический способ их вычисления, а также некоторое количество таких характеристик языков программирования, как процедуры, неструктурированный поток управления, составные переменные и указатели, параллелизм. По существу, решение проблемы построения среза в присутствии одной из указанных особенностей языка «ортогонально» решению для каждой из остальных особенностей. При рассмотрении динамических методов построения срезов также учитывался вопрос о том, являются ли полученные срезы исполняемыми программами, воспроизводящими поведение или часть поведения исходной программы. Когда это было возможно, различные решения одной и той же проблемы сравнивались путем применения каждого алгоритма к одной и той же исходной программе. Рассмотрены возможности и проблемы интеграции решений для «ортогональных» особенностей языка. Дополнительно нами рассмотрены подходы к конкретизации программ, позволяющей осуществлять построение срезов программ более общего вида.

Помимо работы Типа [130] можно указать достаточно большое количество описаний структуры методов срезов, а также несколько обзоров этих методов.

Венкатеш [131] приводит формальное определение нескольких типов срезов в терминах денотационной семантики. Он различает три независимых критерия, в соответствии с которыми нужно классифицировать срезы: статические или динамические, обратные или прямые, срезы-замыкания или исполняемые программы. Некоторые методы срезов в литературе были классифицированы в соответствии с этими критериями [19, 72, 75, 97, 114, 139].

Лахотия [99] переформулирует некоторые методы срезов [72, 113, 139], а также предложенный Хорвитц и др. [72] алгоритм программной интегра-

ции, в терминах операций на ориентированных графах. Он предлагает единую структуру *срезов графов*, различая *синтаксические* свойства срезов, которые могут быть выведены исключительно из теоретико-графовых построений, и *семантические* свойства, которые включают интерпретацию графового представления срезов. Хотя упомянутая работа посвящена только статическим методам срезов, утверждается, что некоторые методы динамических срезов [19, 97] могут быть промоделированы подобным же образом.

Гупта и Соффа представляют обобщенный алгоритм *статического среза* и решение для сопутствующих проблем, связанных с потоком данных (таких, как распознавание достигающих определений), основанные на обходе потокового графа [64]. Алгоритм характеризуется следующими параметрами:

- *направление*, в котором обходится управляющий граф (обратный или прямой обход),
- *тип* рассматриваемой зависимости (по управлению или по данным),
- *ширина* поиска (а именно: должны ли рассматриваться только непосредственные зависимости или также и транзитивные),
- возможность анализа только зависимостей, встречающихся на *всех* путях управляющего графа, или также зависимостей, которые появляются только на *некоторых* из путей.

Критерием среза является либо *множество переменных* в определенной точке программы, либо *множество операторов*. Для срезов, при вычислении которых в расчет принимаются зависимости по данным, пользователь может выбирать значения переменных *перед* оператором или *после* него.

Хорвитц и Репс [74] представляют обзор работ, проводившихся в Университете г. Мэдисон в области срезов, дифференциации и интеграции однопроцедурных и многопроцедурных программ как операций на графах программных зависимостей [68, 71—73, 75, 123]. Наряду с описанием наиболее важных определений, алгоритмов, теорем и выводов о сложности детально обсуждается мотивация упомянутых исследований.

Первичная классификация статических и динамических методов срезов была представлена Камкаром [83, 84]. Различие между работой Камкара и данной заключается в следующем. Во-первых, наша работа является более полной и современной; к примеру, Камкар вообще не рассматривает статьи, посвященные срезам при наличии неструктурированного потока управления [16, 22, 25, 47], или методы вычисления срезов, основанные на отношениях информационных потоков [34, 62]. Во-вторых, различны способы организации самих работ: тогда как Камкар рассматривает каждый метод сре-

зания и его применение отдельно от других методов, наша работа организована в терминах «ортогональных» проблем решения задачи, возникающих из-за процедур, составных переменных, совмещения имен, указателей. Такой подход делает возможным комбинированные решения для различных направлений. В-третьих, в отличие от Камкара, мы сравниваем точность и эффективность методов рассечения (применяя их к одним и тем же или похожим тестовым программам) и пытаемся определить их фундаментальные достоинства и слабости (т.е. плюсы и минусы, которые не зависят от исходного представления). Наконец, Камкар не уделяет внимания ни одной из недавних работ [53, 56, 57], посвященных повышению точности рассечения путем применения техник компиляторной оптимизации.

СПИСОК ЛИТЕРАТУРЫ

1. **Ершов А. П.** Избранные труды. — Новосибирск: Наука, 1994. — 416 с.
2. **Друнин А. В.** Автоматизированное построение программных компонент на основе устаревших программ // Автоматизированный реинжиниринг программ. — СПб: СПбГУ, 2000. — С. 184—204.
3. **Касьянов В. Н.** О нахождении аргументов и результатов в схемах с косвенной адресацией // Программирование. — 1976. — № 1. — С. 6—15.
4. **Касьянов В. Н.** Практический подход к оптимизации программ. — Новосибирск, 1978. — 43 с. — (Препр./АН СССР. Сиб. отд-ние. ВЦ; № 133).
5. **Касьянов В. Н.** Смешанные вычисления и оптимизация программ // Кибернетика. — 1980. — № 2. — С. 51—54.
6. **Касьянов В. Н.** Вопросы конкретизации программ // Проблемы системного и теоретического программирования. — Новосибирск: НГУ, 1982. — С. 35—45.
7. **Касьянов В. Н.** Редуцирующие преобразования программ // Трансляция и оптимизация программ. — Новосибирск: ВЦ СО АН СССР, 1983. — С. 86—98.
8. **Касьянов В. Н.** Учет априорной информации при анализе свойств состояний программ // Математическая теория программирования. — Новосибирск: ВЦ СО АН СССР, 1985. — С. 150—157.
9. **Касьянов В. Н.** Оптимизирующие преобразования программ. — М.: Наука, 1988. —335 с.
10. **Касьянов В. Н.** Аннотирование программ и их преобразование // Программирование. — 1989. — № 4. — С. 3—16.
11. **Касьянов В. Н.** Трансформационный подход к конкретизации программ // Кибернетика. — 1989. — № 6. — С. 28—32.
12. **Касьянов В. Н.** Трансформационные методы и средства конструирования эффективных и надежных программ // Кибернетика и системный анализ. — 1993. — № 2. — С. 30—39.
13. **Касьянов В. Н., Поттосин И. В.** Методы построения трансляторов.— Новосибирск: Наука, 1986. — 344 с.
14. **Agesen O., Holzle U.** Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages // Proc. of the Object-Oriented Programming Systems, Languages and Applications. — 1995. — P. 91—107.
15. **Agrawal H.** Towards Automatic Debugging of Computer Programs: PhD thesis. — Purdue University, 1991.
16. **Agrawal H.** On slicing programs with jump statements // ACM SIGPLAN Notices. — 1994. — Vol. 29**,** N 6. — P. 302—312.
17. **Agrawal H., DeMillo R. A., Spafford E. H.** Dynamic slicing in the presence of unconstrained pointers // Proc. of the ACM Fourth Symposium on Testing, Analysis, and Verification (TAV4). — 1991. — P. 60—73.

18. **Agrawal H., DeMillo R. A., Spaffbrd E. H.** Debugging with dynamic slicing and backtracking // Software — Practice and Experience. —1993. — Vol. 23. — P. 589—616.
19. **Agrawal H., Horgan J. R.** Dynamic program slicing // ACM SIGPLAN Notices. — 1990. — Vol. 25, N 6. — P. 246—256.
20. **Aho A. V., Sethi R., Ullman J. D.** Compilers. Principles, Techniques and Tools. — Addison-Wesley, 1986.— 796 p.
21. **Alpern B., Wegman M. N., Zadeck F. K.** Detecting equality of variables in programs // Conf. Record of the 15th ACM Symp. on Principles of Programming Languages. — San Diego, 1988. — P. 1—11.
22. **Ball T. J.** The Use of Control-Flow and Control Dependence in Software Tools: PhD thesis. — Univ. of Wisconsin-Madison, 1993.
23. **Ball T., Eick S. G.** Visualizing program slices // Proc. of the IEEE Symp. on Visual Languages. — St. Louis, Missouri, 1994 —P. 288—295.
24. **Ball T., Eick S. G.** Software visualizing program in large // IEEE Computer. — 1996. — Vol. 29, N 4. — P. 25—39.
25. **Ball T., Horwitz S.** Slicing programs with arbitrary control-flow // Lect. Notes Comput. Sci. — 1993. — Vol. 749. — P. 206—222.
26. **Balzer R. M.** EXDAMS — Extendable Debugging And Monitoring System // Proc. of the AFIPS SJCC, 34 (1969) 567—586.
27. **Banning J. P.** An efficient way to find the side effects of procedure calls and the aliases of variables // Conf. Record of the Sixth ACM Symp. on Principles of Programming Languages. — 1979. — P. 29—41.
28. **Bames J. G. P.** Programming in Ada. — Addison-Wesley, 1982.
29. **Barendregt H.P., van Eekelen M. C. J. D., Glauert J. R. W., Kennaway J. R., Plasmeijer M. J., Sleep M.R.** Term graph rewriting // Lect. Notes Comput. Sci. — 1987. —Vol. 259. — P. 141—158.
30. **Bates S., Horwitz S.** Incremental program testing using program dependence graphs // Conf. Record of the 12th ACM Symp. on Principles of Programming Languages. — Charleston, SC, 1993. — P. 384—396.
31. **Barth J. M.** A practical interprocedural data flow analysis algorithm // Communs ACM. — 1978. — Vol. 21. — P. 724—736.
32. **Beck J.** Program and interface slicing for reverse engineering // Proc. of the 15th Intern. Conf. on Software Engineering. — Baltimore, 1993.
33. **Beck J., Eichmann D.** Program and interface slicing for reverse engineering // Proc. of the 15th Intern. Conf. on Software Engineering. — Baltimore, 1993. — P. 509—518.
34. **Bergeretti J.-F., Carre B. A.** Information-flow and data-flow analysis of while-programs // ACM Trans. Progr. Lang. and Systems. — 1985. — Vol. 7. — P. 37—61.
35. **Algebraic** Specification / Ed. by J. A. Bergstra, J. Heering, P. Klint. — The ACM Press in cooperation with Addison-Wesley, 1989.
36. **Binkley D.** Using semantic differencing to reduce the cost of regression testing // Proc. of the IEEE Conf. on Software Maintenance. — Orlando, Florida, 1992.

37. **Beszedes A., Gergely T., Szabo Z.M, Farago C., Gyimothy T.** Forward computation of dynamic slices of C programs. — Szeged, 2000. — (Tech. Rep. / RGAI; TR-2000-001).
38. **Beszedes A., Gergely T., Szabo Z,M., Csirik J., Gyimothy T.** Dynamic slicing method for maintenance of large C programs // Proc. of 5th European Conf. on Software Maintenanence and Reengineering (CSMR 2001). — Lisbon, 2001. — P. 105—113.
39. **Bieman J.M., Ott L.M.** Measuring functional cohesion // IEEE Trans. Software Eng. — 1994. — Vol. 20, N 8. — P. 644—657.
40. **Binkley D.** Using semantic differencing to reduce the cost of regression testing // Proc. of Conf. on Software Maintenance. — 1992. — P. 41—50.
41. **Binkley D.** Slicing in the presence of parameter aliasing // Proc. of the 3d Software Engineering Research Forum. — Orlando, Florida, November 1993. — P. 261—268.
42. **Binkley D.** Precise executable interprocedural slices // ACM Letters Progr. Lang. and Systems. —1993. — Vol. 2. — P. 31—45.
43. **Binkley D.** Interprocedural constant propagation using dependence graphs and a data-flow model // Lect. Notes Comput. Sci. — 1994. — Vol. 786, — P. 374—388.
44. **Binkley D., Gallagher K.** Program slicing // Advances in Computers. — Academic Press, 1996. — Vol. 43.
45. **Cheng J.** Slicing concurrent programs — a graph-theoretical approach // Lect. Notes Comput. Sci. — 1993. — Vol. 749. — P. 223—240.
46. **Choi J.-D., Burke M., Carini P.** Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects // Conf. Record of the 20th ACM Symp. on Principles of Programming Languages. — ACM Press, 1993. — P. 232—245.
47. **Choi J.-D., Ferrante J.** Static slicing in the presence of goto statements // ACM Trans. Progr. Lang. and Systems. — 1994. — Vol. 16. — P. 1087—1113.
48. **Choi J.-D., Miller P. B., Netzer R. H. B.** Techniques for debugging parallel programs with flowback analysis // ACM Transactions on Programming Languages and Systems. — 1991. — Vol. 13. — P. 491—530.
49. **Cooper K.D., Kennedy K.** Interprocedural side-effect analysis in linear time // ACM SIGPLAN Notices. — 1988. — Vol. 23, N 7. — P. 57—66.
50. **Cytron R., Ferrante J., Rosen B. K., Wegman M. N., Zadeck F. K.** Efficiently computing static single assignment form and the control dependence graph // ACM Trans. Progr. Lang. and Systems. — 1991. — Vol. 13. — P. 451—490.
51. **Duesterwald E., Gupta R., Sofia M. L.** Distributed slicing and partial reexecution for distributed programs // Proc. of the 5th Workshop on Languages and Compilers for Parallel Computing. — New Haven, Connecticut, 1992. — P. 329—337.
52. **Duesterwald E., Gupta R., Sofia M. L.** Rigorous data flow testing through output influences // Proc. of the 2nd Irvine Software Symp. (ISS'92). — California, 1992. — P. 131—145.
53. **Ernst M.** Practical fine-grained static slicing of optimized code. — Redmond, WA, 1994. — (Tech. Rep. / Microsoft Research; MSR-TR-94-14).

54. **Ferrante J., Ottenstein K. J., Warren J. D.** The program dependence graph and its use in optimization // ACM Trans. Progr. Lang. and Systems. — 1987. — Vol. 9. — P. 319—349. — Русск.пер.: **Ферранте Дж., Оттенштейн К., Уоррен Дж.** Граф программных зависимостей и его применение в оптимизации // Векторизация программ: теория, методы, реализация. — М.: Мир, 1991. — С. 141—182.
55. **Field J.** A simple rewriting semantics for realistic imperative programs and its application to program analysis // Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. — 1992. — P. 98—107.
56. **Field J., Ramalingam G., Tip F.** Parametric program slicing // Conf. Record of the 22nd ACM Symp. on Principles of Programming Languages. — San Francisco, CA, 1995. — P. 379—392.
57. **Field J., Tip F.** Dynamic dependence in term rewriting systems and its application to program slicing // Lect. Notes Comput. Sci. — 1994. — Vol. 844. — P. 415—431.
58. **Fritzson P., Shahmehri N., Kamkar M., Gyimothy T.** Generalized algorithmic debugging and testing // ACM Letters Progr. Lang. and Systems. — 1992. — Vol. 1. — P. 303—322.
59. **Calder A., Grunwald D.** Reducing indirect function call overhead in C++ programs // Conf. Record of 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. — 1994. — P. 397—408.
60. **Gallagher K. B.** Using Program Slicing in Software Maintenance: PhD thesis. — University of Maryland, 1989.
61. **Gallagher K.B., Lyie J.R.** Using program slicing in software maintenance // IEEE Transactions on Software Engineering. — 1991. — Vol. 17. — P. 751—761.
62. **Gopal R.** Dynamic program slicing based on dependence relations // Proc. of the Conf. on Software Maintenance. — 1991.— P. 191—200.
63. **Gupta R., Harrold M. J., Soffa M. L.** An approach to regression testing using slicing // Proc. of the Conf. on Software Maintenance. — 1992. — P. 299—308.
64. **Gupta R., Soffa M. L.** A framework for generalized slicing. — Pittsburg, 1992. — (Tech. Rep. / University of Pittsburgh; TR-92-07).
65. **Gyimothy T., Beszedes A., Forgacs I.** An efficient relevant slicing method for debugging // Lect. Notes Comput. Sci. — 1999. — Vol. 1687. — P. 303—321.
66. **Harrold M. J., Rothermel G.** Performing dataflow testing on classes // Proc. of the 2nd ACM SIGSOFT Symp. on the Foundations of Software Engineering. — 1994. — P. 154—163.
67. **Hausler P.** Denotational program slicing // Proc. of the 22nd Hawaii International Conf. on System Sciences. — Hawaii, 1989. — P. 486—494.
68. **Horwitz S**. Identifying the semantic and textual differences between two versions of a program // ACM SIGPLAN Notices. — 1990. — Vol. 25, N 6. — P. 234—245.
69. **Horwitz S., Pfeiffer P., Reps T.** Dependence analysis for pointer variables // ACM SIGPLAN Notices. — 1989. —Vol. 24, N 7.
70. **Horwitz S., Prins J., Reps T.** On the adequacy of program dependence graphs for representing programs // Conf. Record of the 15th Annual ACM Symp. on Principles of Programming Languages. — 1988. — P. 146—157.

71. **Horwitz S., Prins J., Reps T.** Integrating non-interfering versions of programs // Conf. Record of the ACM SIGSOFT/SIGPLAN Symp. on Principles of Programming Languages. — 1988. — P. 133—145.
72. **Horwitz S., Prins J., Reps T.** Integrating noninterfering versions of programs // ACM Trans. Progr. Lang. and Systems. — 1989. — Vol. 11. — P. 345—387.
73. **Horwitz S., Reps T.** Efficient comparison of program slices // Acta Informatica. — 1991. — Vol. 28. — P. 713—732.
74. **Horwitz S., Reps T.** The use of program dependence graphs in software engineering // Proc. of the 14th International Conf. on Software Engineering. — Melbourne, 1992. — P. 392—411.
75. **Horwitz S., Reps T., Binkley D.** Interprocedural slicing using dependence graphs // ACM Trans. Progr. Lang. and Systems. — 1990. — Vol. 12. — P. 26—61.
76. **Hwang J. C., Du M. W., Chou C. R.** Finding program slices for recursive procedures // Proc. of the 12th Annual Intern. Computer Software and Applications Conf. — Chicago, 1988.
77. **Hwang J. C., Du M. W., Chou C. R.** The influence of language semantics on program slices // Proc. of the 1988 Intern. Conf. on Computer Languages. — Miami Beach, 1988.
78. **Jackson D., Rollins E. J.** A new model of program dependences for reverse engineering // Proc. of the 2nd ACM SIGSOFT Conf. on Foundations of Software Engineering. — New Orleans, LA, 1994. — P. 2—10.
79. **Jackson D., Rollins E. J.** Abstraction mechanisms for pictorial slicing // Proc. of the IEEE Workshop on Program Comprehension. — Washington, 1994. — P. 82—88.
80. **Jacobson I., Lindstrom F.** Reengineering of old systems to an object-oriented architecture // ACM SIGPLAN Notices. — 1991. — Vol. 26, N 11. — P. 340—350.
81. **Jiang J., Zhou X., Robson J. D.** Program slicing for C — the problems in implementation // Proc. of the Conf. on Software Maintenance. — 1991. — P. 182—190.
82. **Johnson R., Pearson D., Pingali K.** The program structure tree: computing control regions in linear time // ACM SIGPLAN Notices. — 1994. — Vol. 29, N 6. — P. 171—185.
83. **Kamkar M.** Interprocedural Dynamic Slicing with Applications to Debugging and Testing: PhD thesis. — Linkoping University, 1993.
84. **Kamkar M.** An overview and comparative classification of program slicing techniques // J. Systems Software. — 1995. — Vol. 31. — P. 197—214.
85. **Kamkar M, Fritzson P., Shahmehri N.** Three approaches to interprocedural dynamic slicing // Microprocessing and Microprogramming. — 1993. — Vol. 38. — P. 625—636.
86. **Kamkar M, Fritzson P., Shahmehri N.** Interprocedural dynamic slicing applied to interprocedural data flow testing // Proc. of the Conf. on Software Maintenance. — Montreal, Canada, 1993. — P. 386—395.
87. **Kamkar M., Shahmehri N., Fritzson P.** Interprocedural dynamic slicing and its application to generalized algorithmic debugging // Proc. of the Intern. Conf. on Programming Language Implementation and Logic Programming (PLILP '92), 1992.

88. **Kasyanov V. N.** Transformational approach to program concretization // Theor. Comput. Sci. —1991. — Vol. 90, N 1. — P. 37 — 46.
89. **Kasyanov V. N.** On completeness of mechanism of annotation-directives // Bull. of the Novosibirsk Computing Center. Ser.: Comput. Sci. — 1995. — N 3. — P. 59—68.
90. **Kasyanov V. N.** Formal methods for program reusability // Proc. 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics. — 1997. — Vol. 4. — P. 761 – 766.
91. **Kasyanov V. N.** A support tool for annotated program manipulation // Proc. of 5th European Conf. on Software Maintenanence and Reengineering. — Lisbon: IEEE Computer Society Press, 2001. — P. 85—94.
92. **Kasyanov V. N., Pottosin I. V.** Application of optimization techniques to correctness problems // Proc. IFIP Working Conf. Constructing Quality Software / Ed. by P. G. Hidbard and S. A. Schuman. — Amsterdam: North-Holland, 1979. — P.237—248. — Русск. пер.: Касьянов В.Н., Поттосин И.В. Применение методов оптимизации к проверке правильности программ // Создание качественного программного обеспечения: Тр. рабочей конф. Междунар. федерации по обработке информации. — Новосибирск: ВЦ СО АН СССР, 1978. — Т. 1. — С. 225—237.
93. **Klint P.** A meta-environment for generating programming environments // ACM Trans. Software Engineering and Methodology. — 1993. — Vol. 2. — P. 176—201.
94. **Klop J. W.** Term rewriting systems // Handbook of Logic in Computer Science. — Oxford University Press, 1992. — Vol. 2. — P. 1—116.
95. **Korel B., Ferguson R.** Dynamic slicing of distributed programs // Applied Mathematics and Computer Science. —1992. — Vol. 2. — P. 199—215.
96. **Korel B., Laski J.** Dynamic program slicing // Inform. Processing Letters. — 1988. — Vol. 29. — P. 155—163.
97. **Korel B., Laski J.** Dynamic slicing of computer programs // J. Systems and Software. — 1990. — Vol. 13. — P. 187—195.
98. **Korel B., Yalamanchili S.** Forward computation of dynamic program slices // Proc. of the 1994 Intern. Symp. on Software Testing and Analysis (ISSTA). — Seattle, Washington, 1994.
99. **Kuck D.J., Kuhn R.H., Padua D.A., Leasure B., Wolfe M.** Dependence graphs and compiler optimizations // Conf. Record of the 8th ACM Symp. on Principles of Programming Languages. — 1981. — P. 207—218.
100. **Lakhotia A.** Graph theoretic foundations of program slicing and integration. — Lafayette, 1991. —(Rep. / Univ. of Southwestern Louisiana; CACS TR-91-5-5).
101. **Lakhotia A.** Improved interprocedural slicing algorithm. — Lafayette, 1992. — (Rep. / Univ. of Southwestern Louisiana; CACS TR-92-5-8).
102. **Landi W., Ryder B..G.** Pointer-induced aliasing: A problem classification // Proc. of the 18th ACM Symp. on Principles of Programming Languages. — 1991. — P. 93—103.

103. **Landi W., Ryder B. G.** A safe approximate algorithm for interprocedural pointer aliasing // ACM SIGPLAN Notices. — 1992. — Vol. 27, N 7. — P. 235—248.
104. **Larsen L., Harrold M.J.** Slicing object-oriented Software // Proc. ICSE-18. — Berlin, 1996. — P. 495—505.
105. **Larus J.R., Chandra S.** Using tracing and dynamic slicing to tune compilers. — Madison, 1993. — (Comput. Sci. Tech. Rep. / Univ. of Wisconsin—Madison; N 1174).
106. **Leung H. K. N., Reghbati H. K.** Comments on program slicing // IEEE Trans. Software Eng. — 1987. — Vol. 13. — P. 1370—1371.
107. **Lyie J. R.** Evaluating Variations on Program Slicing for Debugging: PhD thesis. — Univ. of Maryland, 1984.
108. **Lyie J. R., Binkley D.** Program slicing in the presence of pointers // Proc. of the 3rd Software Engineering Research Forum. — Orlando, Florida, 1993. — P. 255—260.
109. **Lyie J.R., Weiser M.** Automatic bug location by program slicing // Proc. of the 2d Intern. Conf. on Computers and Applications. — Beijing, 1987. — P. 877—883.
110. **Maydan D. E., Hennessy J. L., Lam M. S.** Efficient and exact data dependence analysis // ACM SIGPLAN Notices. — 1991. — Vol. 26, N 6. — P. 1—14.
111. **Miller B.R., Choi J.-D.** A mechanism for efficient debugging of parallel programs // ACM SIGPLAN Notices. — 1988. — Vol. 23, N 7. — P. 135—144.
112. **Ning J.Q., Engberts A., Kozaczynski W.** Automated support for legacy code understanding // Communs ACM. — 1994. — Vol. 37. — P. 50—57.
113. **Ott L. M., Thuss J. J.** The relationship between slices and module cohesion // Proc. of the llth Intern. Conf. on Software Engineering. — 1989. — P. 198—204.
114. **Ottenstein K. J., Ottenstein L. M.** The program dependence graph in a software development environment // ACM SIGPLAN Notices. — 1984. — Vol. 19, N 5. — P. 177—184.
115. **Pan H.** Software Debugging with Dynamic Intrumentation and Test-Based Knowledge: PhD thesis. — Purdue University, 1993.
116. **Pan H., Spafford E. H.** Fault localization methods for software debugging // J. Computer and Software Eng. — 1994.
117. **Podgurski A., Clarke L. A.** A formal model of program dependences and its implications for software testing, debugging, and maintenance // IEEE Trans. Software Eng. — 1990. — Vol. 16. — P. 965—979.
118. **Reps T.** Algebraic properties of program integration // Sci. Computer Progr. — 1991. — Vol. 17. — P. 139—215.
119. **Reps T.** On the sequential nature of interprocedural program-analysis problems. — University of Copenhagen, 1994.
120. **Reps T., Bricker T.** Illustrating interference in interfering versions of programs // ACM SIGSOFT Software Engineering Notes. — 1989. — Vol. 17, N 7.
121. **Reps T., Horwitz S., Sagiv M., Rosay G.** Speeding up slicing // ACM SIGSOFT Software Engineering Notes. — 1994. — Vol. 19, N 5. — P. 11—20.
122. **Reps T., Sagiv M., Horwitz S.** Interprocedural dataflow analysis via graph reachability.— Copenhagen, 1994. — (Rep. / Univ. of Copenhagen; DIKU TR 94-14).

123. **Reps T., Yang W.** The semantics of program slicing and program integration // Lect. Notes Comput. Sci. — 1989. — Vol. 352. — P. 60—74.
124. **Rothermel G., Harrold M. J.** Selecting regression tests for object-oriented software // Proc. of Conf. on Software Maintenance. — 1994. — P. 14—25.
125. **Rothermer G., Harrold M. J.** Selecting tests and identifying test coverage requirements for modified software // Proc. of ISSTA'94. — Seattle, Washington, 1994. — P. 169—183.
126. **Shapiro E.Y.** Algorithmic Program Debugging. — MIT Press, 1982.
127. **Shahmehri N.** Generalized Algorithmic Debugging: PhD thesis. — Linkoping University, 1991.
128. **Tip F.** Generation of Program Analysis Tools: PhD thesis. — Univ. of Amsterdam, 1995.
129. **Tip F.** Generic techniques for source-level debugging and dynamic program slicing // Proc. of the 6th Intern. Joint Conf. on Theory and Practice of Software Development. — 1995. — P. 516—530.
130. **Tip F.** A survey of program slicing techniques // J. Programming Languages. — 1995. — Vol. 3. — P. 121—189.
131. **Venkatesh G. A.** The semantic approach to program slicing // ACM SIGPLAN Notices. — 1991. — Vol. 26, N 6. — P. 107—119.
132. **Venkatesh G. A.** Experimental results from dynamic slicing of C programs // ACM Trans. Progr. Lang. and Systems. — 1995. — Vol. 17. — P. 197—216.
133. **Wegman M. N., Zadeck F. K.** Constant propagation with conditional branches // ACM Trans. Progr. Lang. and Systems. — 1991. — Vol. 13. — P. 181—210.
134. **Weihl W. E.** Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables // Conf. Record of the 7th ACM Symp. on Principles of Programming Languages. — 1980. — P. 83—94.
135. **Weise D., Crew R. F., Ernst M., Steensgaard B.** Value dependence graphs: Representation without taxation // Conf. Record of the 21st ACM Symp. on Principles of Programming Languages. — 1994. — P. 297—310.
136. **Weiser M.** Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method: PhD thesis. — Univ. of Michigan, Ann Arbor, 1979.
137. **Weiser M.** Programmers use slices when debugging // Communs ACM. — 1982. — Vol. 25. — P. 446—452.
138. **Weiser M.** Reconstructing sequential behaviour from parallel behaviour projections // Inform. Processing Letters. — 1983. — Vol. 17. — P. 129—135.
139. **Weiser M.** Program slicing // IEEE Trans. Software Eng. — 1984. — Vol. 10. — P. 352—357.
140. **Yang M., Horwitz S., Reps T.** A program integration algorithm that accommodates semantics-preserving transformations // ACM SIGSOFT Software Engineering Notes. — 1990. — Vol. 15, N 6. — P. 133—143.
141. **Zima H., Chapman B.** Super compilers for Parallel and Vector Computers: Frontier Series. — New York: ACM Press, 1991.

ОГЛАВЛЕНИЕ

В. Н. Касьянов, И. Л. Мирзуитова

SLICING:
СРЕЗЫ ПРОГРАММ И ИХ ИСПОЛЬЗОВАНИЕ

Под редакцией
проф. Виктора Николаевича Касьянова

Рукопись поступила в редакцию 28.12.2001
Ответственный за выпуск Г. П. Несговорова
Редактор З. В. Скок

Подписано в печать 30.05.02
Формат бумаги 60 × 84 1/16 Объем 6.3 уч.-изд.л., 6.9 п.л.
Тираж 50 экз.

НФ ООО ИПО “Эмари” РИЦ, 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6